HUMAN-POWERED DATA MANAGEMENT

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Aditya G. Parameswaran

September 2013

This thesis is dedicated to my dear (late) thatha, Dr. S. Narayana Iyer, whose passion for science and engineering will forever remain an inspiration.

Acknowledgments

First and foremost, I'd like to thank Hector Garcia-Molina for being the ideal advisor a graduate student could ever hope for. Hector is absolutely brilliant; a clear thinker with the uncanny ability to discover and communicate crisp, simple insights in almost any problem domain, even those alien to him. Hector allowed me the freedom to pursue a range of problems in a variety of topics (I worked in at least four separate areas!), and the freedom to actively seek out and execute collaborations with others. Hector was always available for advice on any topic, ranging from faculty interviews, to reviewing and teaching, all the way to writing recommendation letters — I've lost track of the number of times I have knocked on his door on a Saturday or Sunday morning with the most banal questions in hand. I will forever treasure the advice I've received from Hector over the last five years, and I'm sure I have yet to learn so much more from him.

Next, I'd like to express my gratitude to Jennifer Widom. Jennifer contributed so frequently and so generously to my Ph.D. training and research that she is, for all intents and purposes, my second advisor. Even when I was a newbie graduate student, still unsure of my footing in the research community, she welcomed me into her research group and made sure I was always kept up-to-speed. (I miss those heated Trio group discussions!) When we started working on Deco, Jennifer was kind enough to devote an hour of her time solely on me every week; I inevitably left every meeting with her with some unifying insight, new avenues to explore, and encouragement for my half-baked ideas mixing theory and practice. Jennifer's razor-sharp intellect is only matched by her ability to write and present; her turn of phrase is truly a joy to witness (in real time).

I'm fortunate to have been around when Alkis Polyzotis was doing his sabbatical at Stanford. Alkis is a remarkable, brilliant, and yet incredibly nice person: he took me and my crazy ideas seriously even though I was still an unaccomplished junior grad student. Alkis was the first senior researcher to treat me as a peer rather than a student: I've spent many hours brainstorming, proving theorems, and thinking "big" with Alkis. I will forever cherish our time laying the road-map for the scoop project in our CIDR paper together. Our times together have helped me grow and mature as a researcher.

Throughout my years in the Infolab, Marianne Siroker was truly a godsend — I've counted on her help on numerous occasions. Jure Leskovec was a exceptional source of inspiration and advice, especially during the faculty search season. I'm grateful to Balaji Prabhakar, who I worked with during my first year at Stanford, for putting up with grad student teething pains.

I am forever indebted to collaborators from whom I've learned so very much—all the way from formulating a problem to writing the final paper : Arvind Arasu, Kedar Bellare, Nilesh Dalvi, Anish Das Sarma, Alon Halevy, Raghav Kaushik, Georgia Koutrika, Anand Rajaraman, Rajeev Rastogi, Vibhor Rastogi, Hyunjung Park, Jeffrey Ullman, and others. I'd like to single out Alon, Rajeev, and Jeff for special gratitude: these experienced researchers in industry and academia took the time out of their busy schedules to meet and brainstorm with me; even the mere fact that such well-known researchers were excited about the same topics meant the world to me as an early-stage grad student.

Even before my time at Stanford, my professors at IIT Bombay helped me understand and appreciate the beauty of computer science, after which I was convinced that I would pursue an academic career in some form. In particular, I'd like to express my thanks to my undergraduate advisor, Supratik Chakraborty, for getting me initiated into research.

I found the Infolab to be an incredibly social environment: I'm going to miss the intellectually stimulating (and sometimes not so intellectual) conversations over coffee and lunch. I'd like to thank Paul Heymann, Robert Ikeda, Manas Joglekar, Andrej Krevl, David Menestrina, Julian McAuley, Andreas Paepcke, Semih Salihoglu, Steven Whang, among many others.

I'd like to thank my other Bay Area buddies: Parag Agrawal, Onkar Dalal, Nandan Dixit, Yashodhan Kanoria, Karthik Ramaswamy, Brinda Ramachandran, Arunanshu Roy, Neesha Subramaniam, Ankur Taly, Venkat Viswanathan, for putting up with the unexpected ups and downs grad school brings. It has been fun — I will miss you guys! I'm also glad to have been part of a close-knit group of amazing IIT Bombay Computer Science friends: to Bhaskara, Rajhans, Lakulish, Bodicherla, Alekh, Ansari, and Luv, many thanks! To my other IITB friends who are far too many to name: much love, and I promise to do a better job of staying in touch!

My heartfelt gratitude to my parents and grandparents, without whom I would not have made it this far. I have counted on their wisdom, encouragement, and advice time and time again, and words cannot even begin to express my love and respect for them.

Lastly, to Dipti, for understanding, as only a fellow academic would, the pleasures and tribulations of a PhD. I'm indeed fortunate to have had Dipti's boundless love, unwavering support, and calm intellect by my side for the last six years. I look forward to our journey ahead, together.

Abstract

Fully automated algorithms are inadequate for a number of data analysis tasks, especially those involving images, video, or text. Thus, there is often a need to combine "human computation" (or crowdsourcing), together with traditional computation, in order to improve the process of understanding and analyzing data. However, most data management applications currently employ crowdsourcing in an ad-hoc fashion; these applications are not optimized for low monetary cost, low latency, or high accuracy. In this thesis, we develop a formalism for reasoning about human-powered data management, and use this formalism to design: (a) a toolbox of basic data processing algorithms, optimized for cost, latency, and accuracy, and (b) practical data management systems and applications that use these algorithms. We demonstrate that our techniques lead to algorithms and systems that expend very few resources (e.g., time waiting, human effort, or money spent), while providing just as high quality results, as compared to approaches currently used in practice.

Contents

Acknowledgments **v**

Abstract **vii**

1 Introduction **1**

1.1 Thesis Overview and Contributions 2

1.1.1 Part I: Crowd-Powered Algorithms 3

1.1.2 Part II: Crowd-Powered Systems/Applications 6

1.2 Overview of Related Perspectives 7

2 Background **9**

2.1 Crowdsourcing Mechanisms 9

2.2 Interacting with a Crowdsourcing Marketplace 10

2.3 Notation 15

3 Algorithm 1: Filtering **18**

3.1 Introduction 18

3.1.1 Outline of Chapter 19

3.2 Preliminaries 20

3.2.1 Formal Definitions 21

3.2.2 Metrics 24

3.2.3 The Problems 26

3.2.4 Computing Probabilities at Grid Points 27

3.3 Deterministic Strategies 27

3.3.1 The Paths Principle 28

3.3.2 Shapes and Ladders 29

3.4 Probabilistic Strategies . . . 36
3.5 Other Formulations . . . 39
3.5.1 Problem 3.2.2 . . . 39
3.5.2 Problems 3.2.3 and 3.2.4 . . . 39
3.5.3 Problem 3.2.5 . . . 40
3.6 Multiple Filters . . . 40
3.7 Experiments . . . 41
3.8 Related Work . . . 50
3.9 Conclusions . . . 52

4 Algorithm 1 (Variants): Filtering Generalizations 53
4.1 Introduction . . . 53
4.1.1 Outline of Chapter . . . 54
4.2 The Answer-Record Representation . . . 55
4.2.1 Setting with Worker Abilities . . . 55
4.2.2 Incorporating Prior Information . . . 61
4.3 The Posterior-Based Representation . . . 62
4.3.1 Basic Setting . . . 62
4.3.2 Generalization of Worker Abilities . . . 66
4.3.3 Incorporating Prior Information . . . 69
4.4 Other Generalizations . . . 70
4.4.1 Outline and Summary of Results . . . 71
4.4.2 Overall Approach to Generalizations . . . 72
4.4.3 Improvement Generalization: Difficulty . . . 73
4.4.4 Functionality Addition Generalization: Picking Workers with Costs . . . 74
4.4.5 Functionality Addition Generalization: Latency . . . 75
4.4.6 Functionality Addition Generalization: Scoring . . . 76
4.5 Conclusions . . . 77

5 Algorithm 2: Finding 78
5.1 Introduction . . . 78
5.1.1 Outline of Chapter . . . 80
5.2 Definitions . . . 81
5.2.1 Setting . . . 81

5.2.2 Performance Metrics 83
5.2.3 Optimization Problems 84
5.3 Deterministic Setting 85
5.3.1 Problem 5.2.2: Optimal Cost 86
5.3.2 Problem 5.2.3: Multiplicative Approximation 87
5.3.3 Problem 5.2.4: Additive Approximation 88
5.3.4 Discussion 89
5.4 Uncertain Human Answers 89
5.4.1 Model of Uncertainty 90
5.4.2 Algorithms for $k_0 = 0$ 91
5.4.3 Pass and Fail Items 98
5.4.4 Cost Computation 99
5.4.5 Discussion 100
5.5 Problem 5: Expected Cost and Latency 101
5.6 Related Work 104
5.7 Conclusions 105

6 Algorithm 3: Maximum **106**
6.1 Introduction 106
6.1.1 Outline of Chapter 108
6.2 Judgment Problem 109
6.2.1 Problem Setup 109
6.2.2 Maximum Likelihood 111
6.2.3 Computational Complexity 113
6.2.4 Heuristic Strategies 115
6.2.5 Experiments 121
6.3 Next Votes Problem 126
6.3.1 Maximum Likelihood 126
6.3.2 Computational Complexity 128
6.3.3 Selection and Evaluation of Additional Votes 131
6.3.4 Experiments 134
6.4 Related Work 139
6.5 Conclusion 140

7 Algorithm 4: Categorization **142**

7.1 Introduction . . . 142

7.1.1 Outline of Chapter . . . 144

7.2 Dimensions . . . 144

7.2.1 Summary of Results and Outline . . . 145

7.3 The Categorization Problem . . . 146

7.4 Single Target Category . . . 149

7.4.1 Single-Bounded . . . 149

7.4.2 Single-Unlimited . . . 155

7.5 Multiple Target Nodes . . . 157

7.5.1 Multi-Bounded . . . 157

7.5.2 Multi-Unlimited . . . 163

7.6 Experimental Study . . . 164

7.6.1 Methodology . . . 166

7.6.2 Results . . . 166

7.7 Related Work . . . 168

7.8 Conclusions . . . 169

8 Application 1: Datasift **170**

8.1 Introduction . . . 170

8.1.1 Outline of Chapter . . . 174

8.2 Preliminaries and Components . . . 174

8.2.1 Crowdsourced Components . . . 175

8.2.2 Automated Components . . . 175

8.3 Configurations . . . 178

8.4 Implementation . . . 179

8.5 Initial Evaluation on Textual Queries . . . 181

8.6 Rich Queries and Parameter Tuning . . . 184

8.7 Related Work . . . 190

8.8 Conclusion . . . 191

9 Application 2: Peer Evaluation in MOOCs **192**

9.1 Introduction . . . 192

9.2 Peer Evaluation Experiments . . . 193

9.3 Conclusion . . . 205

10 Related Work **206**

10.1 Surveys . . . 206

10.2 Human Computer Interaction . . . 207

10.2.1 Novel Worker and User Interfaces . . . 207

10.2.2 Games With a Purpose . . . 207

10.3 Machine Learning and Artificial Intelligence . . . 208

10.3.1 Active Learning . . . 208

10.3.2 Quality Estimation . . . 208

10.3.3 Decision Theory . . . 209

10.4 Social Science . . . 210

10.5 Game Theory and Pricing . . . 210

10.6 Systems . . . 210

10.6.1 Crowd-Powered Database Systems . . . 210

10.6.2 Domain-Specific Toolkits . . . 211

10.6.3 Generic Toolkits . . . 211

10.7 Algorithms . . . 212

10.7.1 Sorting, Max, and Top-K . . . 212

10.7.2 Entity Resolution and Clustering . . . 212

10.7.3 Other Algorithms . . . 213

11 Conclusions **214**

11.1 Summary of Contributions . . . 214

11.2 Future Work . . . 214

11.2.1 Medium-Term Research Agenda . . . 215

11.2.2 Long-Term Research Agenda . . . 216

11.3 Outlook . . . 217

Bibliography **218**

List of Tables

1.1 Summary of Thesis . . . 3
1.2 Summary of Perspectives . . . 7

2.1 Data Processing Question Types . . . 15
2.2 Table of Notation . . . 16
2.3 Table of Notation (continued) . . . 17

4.1 Complexity of Generalizations . . . 72

6.1 Predictions using each heuristic for Figure 6.1. . . . 120

7.1 Summary of Results in Categorization . . . 146

8.1 Components of DataSift . . . 177
8.2 List of Textual Queries for Initial Evaluation . . . 181
8.3 List of Rich Queries . . . 185
8.4 Breakdown of Monetary Costs . . . 187

10.1 Summary of Perspectives . . . 206

List of Figures

2.1 Interacting with a Marketplace 11
2.2 Filtering Task 12
2.3 Rating Task 13

3.1 Strategy Representation 21
3.2 Depiction of Error at Grid Points 22
3.3 Equations for p_0/p_1 25
3.4 Shape Representation 30
3.5 Pruning Shapes 30
3.6 Experiments on Varying m, s, e_1 42
3.7 Experiments on Varying m on Average, and for Fixed Parameters 45
3.8 Experiments on Running Time and Accuracy on Varying s 48

4.1 Mapping Between Representations 63

5.1 Possible states of the system on asking up to 3 questions. 86
5.2 Comparing Algorithms on ET and EC 104

6.1 Representation for Max: Matrix and Graph 110
6.2 Precision at 1 (P@1) versus Edge Coverage. 123
6.3 Experiments for Judgment Problem 125
6.4 Selecting Next Votes 133
6.5 Experiments for Next Votes 134
6.6 Additional Experiments for Next Votes: Incremental Gain 135
6.7 Additional Experiments for Next Votes: Larger Scenarios 136
6.8 Additional Experiments for Next Votes: Varying Complete Tournament 136

6.9 Additional Experiments for Next Votes: Item Types 138

7.1 Categorization 143
7.2 Hardness Proof for Single-Bounded 151
7.3 Partition Example 153
7.4 Hardness Proof for Multi-Bounded 158
7.5 Triviality of Multi-Unlimited 164
7.6 Experiment on Varying the Number of Questions Asked 165
7.7 Experiment on Varying the Size of the Tree 167
7.8 Experiment on Varying the Number of Phases 168

8.1 DataSift Overview 172
8.2 DataSift Example 172
8.3 Experiments 182
8.4 Experiments: Precision Curve with Error Bars 183
8.5 Experiments on Easy vs. Hard Queries 184
8.6 Experiments on Textual vs. Rich Queries 185
8.7 Heat Map of True Positives 188
8.8 Experiments on Varying Sampled Items 189
8.9 Experiments on Varying Sampled Items for Fewer Suggestions 190
8.10 Experiments on Varying Target Number of Items 191

9.1 Basic Comparison 198
9.2 Varying Class Size 199
9.3 Variance with Factor 200
9.4 Bias with Factor 201
9.5 Bias Vs. Variance 202
9.6 Computational Cost of Varying Class Size 202
9.7 Computational Cost of Varying Class Size 203
9.8 Relative Impacts of Parameters 204

Chapter 1

Introduction

We are drowning in information, while starving for wisdom.
— E. O. Wilson

With the advent of the "data deluge" [176], organizations world-wide have been struggling with designing algorithms and systems to better process and analyze the massive quantities of data collected every day. It is estimated that 80% of this data is unstructured [101,191], i.e., consisting largely of images, videos, and raw text. While there have been significant advances in automated mechanisms for interpreting and extracting information from unstructured data, algorithms to fully comprehend unstructured data have not been developed yet. It is widely acknowledged that we are at least several decades away from this goal [123, 165].

For this reason, using humans to analyze certain aspects of unstructured data can be crucial. Humans have an innate understanding of language, speech, and images; they are able to process, reason about, and provide solutions to problems faced often in managing and processing unstructured data. Moreover, the abundance of cheap and reliable internet connectivity throughout the world has given rise to "crowdsourcing" marketplaces, such as Mechanical Turk [14] and ODesk [17], enabling the use of humans to process data on demand.

In particular, crowdsourcing has been applied in the following large-scale unstructured data management applications:

- *Content Moderation:* Humans in crowdsourcing marketplaces are often used for content moderation of images uploaded on web sites [5]. That is, humans are asked to determine whether each user-uploaded image is appropriate to be viewed by a general audience.
- *Web Extraction:* Humans are also used for information extraction from web sites. That is, humans are asked to provide specific information by looking up web sites and finding, say,

phone numbers or addresses of restaurants. Humans may also be used to aid semi-automatic information extraction systems—for instance, Yahoo! [24] uses crowdsourcing to build web extraction wrappers, and to verify extracted information [47, 66, 90, 91, 151].

- *Search Relevance:* Most search companies, e.g., Bing [15], Google [11], Yahoo!, use humans to evaluate the performance of their search algorithms [31].
- *Entity Resolution:* Entity Resolution, or deduplication [83] refers to the problem of identifying if two textual records refer to the same entity. Groupon and Yahoo! both use crowdsourcing for entity resolution [39, 106, 107].
- *Text Processing:* Crowdsourcing is used in spam identification [145], text classification [34, 173], translation [195], and text editing [41].
- *Video and Image Processing:* Crowdsourcing is used in video analysis [56], for image labeling [164, 183], and in visual aids [45].

Unfortunately, in all of these applications, and overall, crowdsourcing can be subjective or error-prone; it can be time-consuming (humans take longer than computers); and it can be relatively costly (humans need to be paid). Moreover, these three aspects—accuracy, latency, and cost—are correlated in complex ways, making it difficult to optimize the trade-offs among them while designing data processing algorithms and systems.

Returning to content moderation of images (the first item in the list of applications above), we can ask one human to verify if each image is appropriate, but they may make mistakes. So, we may need to ask multiple humans to verify each image. However, asking multiple humans has higher monetary cost, and may have higher latency. Furthermore, we can ask multiple humans to verify each image in parallel, or ask humans in sequence. The former option will have lower latency, while the latter may have lower monetary cost (since we can choose to not ask subsequent questions based on answers to previous ones).

Therefore, all of these applications (and many others) could benefit significantly from an effort to optimize accuracy, cost, and latency in human-powered data processing algorithms and systems.

1.1 Thesis Overview and Contributions

The goal of this thesis is to:

develop a formalism for reasoning about human-powered data processing, and use this formalism to design: (a) a toolbox of basic data processing algorithms, optimized for cost, latency, and accuracy, and (b) practical data management systems and applications that

Part I: Crowd-Powered Algorithms	Simple Filter	Chapter 3	[152]
	Advanced Filter	Chapter 4	[149]
	Find	Chapter 5	[168]
	Max	Chapter 6	[92]
	Categorize	Chapter 7	[155]
Part II: Crowd-Powered Applications	DataSift	Chapter 8	[150]
	Peer Evaluation	Chapter 9	[149]

Table 1.1: Thesis Summary. (Deco [153, 154, 156, 158], another crowd-powered application we have contributed to, is not included in this thesis.)

use these algorithms.

Our first and primary objective is to develop the toolbox of basic data processing algorithms, and the description of the toolbox forms a large part of this thesis. Subsequently, we use these algorithms to develop practical data management systems, which we also describe in this thesis. The main parts of the thesis, as well as the associated chapters and published papers, are shown in Table 1.1. In addition, Chapter 2 introduces some necessary background on crowdsourcing, while Chapter 10 discusses related work.

1.1.1 Part I: Crowd-Powered Algorithms

In the first part of the thesis, our focus is on designing algorithms that use humans as "data processors", i.e., when human involvement can be abstracted as functions or subroutines applied on data elements — we call these *crowd-powered algorithms*. As an example, we may design a crowd-powered sorting algorithm where humans perform pairwise comparisons, or provide ratings to individual items. We design crowd-powered algorithms for the following problems:

1) Simple Filter: (Chapter 3, Reference: [152])
Here, our goal is to design an algorithm to filter a set of items using humans. That is, we want to use humans to identify which of a large set of data items satisfy a certain predicate. This problem is commonplace in crowd-powered applications, such as content moderation, spam identification, and search relevance. Since humans make mistakes, it is not sufficient to simply check for every item, by asking one human, whether it satisfies the predicate. We may instead show each item to multiple people and somehow combine the answers, e.g., if 2 out of 3 vote yes, then we say the item satisfies the predicate. But what is the right way to combine the votes? Simply taking the majority may not

always be the best approach, for instance, when it is known that most items satisfy the predicate. Furthermore, we may not have an unlimited budget to do our filtering. So how do we select a filtering strategy that, for instance, minimizes cost (e.g., total number of questions asked) while keeping overall error below a desired threshold? We find that strategies can be completely characterized using a two-dimensional representation. We show that while the number of strategies is exponential, there are efficient algorithms that can find optimal or near-optimal strategies, optimizing cost and accuracy. We demonstrate that these strategies achieve 20-30% cost reduction as compared to heuristics used in practice.

2) Advanced Filter: (Chapter 4, Reference: [149])
Here, our goal is to generalize our filtering algorithm from Chapter 3 in a number of directions. We provide algorithms for: (a) Incorporation of abilities: adapting filtering to take into account individual worker abilities. (b) Integration with prior information: combining filtering with prior information about items, say from a machine learning algorithm. (c) Latency constraints: providing latency guarantees in addition to cost and error guarantees. (d) Scoring: identifying the scores of items as between $1 \ldots u$, instead of boolean filtering. All of these generalizations require significant changes to the representation of the strategies considered in filtering. In some cases, these representations themselves become intractable to store. For these cases, we describe a new representation scheme that is (a) approximate, but asymptotically optimal—that is, the representation will tend to capture all the information in the previous (intractable) representation as we increase the granularity of the representation (b) efficient—that is, the representation is compact to store, and the algorithm that determines the optimal strategy for filtering has low latency.

3) Find: (Chapter 5, Reference: [168])
Here, our goal is to design an algorithm to find, in a given data set, a pre-specified number of items that satisfy a certain predicate. For instance, we may want humans to identify 20 travel photos from a data set of 10,000 photos to display on a travel website, or a candidate set of 10 resumes that meet certain job requirements from a large pool of 500 applicants. For this problem, it is not sufficient to simply adapt techniques from filtering; in fact, we show that using techniques optimized for filtering can be arbitrarily monetarily expensive for the finding problem. In fact, for finding, we need to focus on a small set of items that are the most "promising", i.e., that are likely to help solve the problem. However, it is not clear how small this set should be: for instance, if we focus on too small a set, we may take too long to find the desired set of items, and if we focus on too large a set, then we may end up spending too much money. We develop algorithms to generate strategies for the finding problem

that are optimal, i.e., the strategies lie on the skyline of cost and latency, for a fixed accuracy threshold.

4) Maximum: (Chapter 6, Reference: [92])
Given that each item in a data set has a certain inherent quality, our goal is to design an algorithm to find the item with the highest quality. For instance, we may want to use humans to find the profile photo that best depicts a given person; or the song that is the catchiest among all the songs of a given musical band. We focus on pairwise votes, that is, we provide humans two items and ask them to pick the one of higher quality. When asking two humans to compare the same pair of items, they may give us different answers, because one or both may have made a mistake, or because the question is inherently subjective. Either way, we may need to aggregate pairwise votes from multiple humans, possibly asking multiple humans the same question, in order to increase our confidence in the final answer. Of course, executing more votes increases the cost as well as latency of the algorithm. We study two problems: given a set of pairwise votes, (a) what is our current best estimate for the highest quality item? (b) if we want to invoke more votes, which are the most effective ones to issue to the crowdsourcing marketplace? We show that both these problems are NP-Hard, even for a relatively simple model of human error. We also design heuristic algorithms that do extremely well in solving both these problems.

5) Categorize: (Chapter 7, Reference: [155])
Here, our goal is to design an algorithm for categorization, i.e., an algorithm that uses humans to categorize an item into a taxonomy of concepts. For instance, we may want to use humans to identify the most appropriate category in a taxonomy for a new Amazon product. We focus on asking people categorization questions of the form "does this item belong to a specific category?" Asking categorization questions corresponding to nodes or categories in the taxonomy that are close to the roots (very *general* questions) are more likely to receive a positive answer, while asking categorization questions corresponding to nodes that are close the the leaves (very *specific* questions) are more likely to receive a negative answer. Asking categorization questions in the "middle" nodes may give more information. Furthermore, the order of questions asked is important as well. Getting a YES or a NO answer to a categorization question may eliminate the need to ask some other categorization questions. We study the problem of deciding which categorization questions to ask, on varying various dimensions: (a) number of target categories: one or multiple, (b) type of taxonomy: general graph or tree, (c) objective: eliminate as many categories as possible, or precisely identify the target categories. We find that while the general problem is NP-Hard, there are special cases for which we can find efficient solutions that eliminate 10X more nodes than naive schemes.

1.1.2 Part II: Crowd-Powered Systems/Applications

In the second part of the thesis, our focus is on designing novel systems and applications that use crowd-powered algorithms as building blocks, thereby demonstrating the utility of these algorithms. In addition, this part reveals that there are additional optimization challenges in designing and assembling crowd-powered applications and systems beyond those tackled in Part I. We study the following applications:

1) DataSift: (Chapter 8, Reference: [150])
Here, our goal is to improve information retrieval using humans. Traditional information retrieval systems have limited functionality. For instance, they are not able to adequately support queries containing non-textual fragments such as images or videos, queries that are very long or ambiguous, or semantically-rich queries over non-textual corpora. To address these problems, we designed DataSift, an expressive and accurate crowd-powered search toolkit that can connect to any corpus, e.g., corpora containing products, web sites, or images. We consider a number of alternative configurations for DataSift using crowdsourced and automated components, and demonstrate gains of 2–3x on precision over traditional retrieval schemes using experiments on real corpora.

2) Peer Evaluation System: (Chapter 9, Reference: [149])
Here, our goal is to improve crowdsourced evaluation in MOOCs (Massive Open Online Courses) [36]. MOOCs have gained prominence in the last few years, with several institutions and organizations, such as Coursera, Udacity, and EdX, offering MOOCs on a variety of subjects. Many of these subjects have assignments and exams that cannot be automatically evaluated, e.g., psychology, sociology, literature, poetry, and history. Thus, peer grading is used to evaluate these assignments. However, peer grading is currently used in an ad-hoc manner—each submission is evaluated by a fixed number of randomly selected peers. We adapt crowd-powered algorithms from Chapter 4 for more effective peer grading. Our techniques provide the same cost guarantees as the current ad-hoc mechanisms, while significantly improving the accuracy of the results. As we show in real experiments on MOOC data, our techniques can reduce error in grade estimation by up to 40%.

We now briefly describe yet another system that we have contributed to that could benefit from our optimized crowd-powered algorithms. This system is not included as part of this thesis.

3) Deco: (References: [153, 154, 156, 158])
Here, our goal is to design a crowd-powered database system. In many applications, we can view human-generated data as a *data source*, so naturally, one would like to seamlessly integrate the crowd

Discipline	Perspectives
Human Computer Interaction	Better interface design; novel interaction mechanisms
Machine Learning	Using humans for training data; improving crowdsourcing
Social Science	Behavioral experiments; motivations; demographic studies
Game Theory	Pricing; incentives; game design
Algorithms and Databases (Us)	Using humans as data processors; optimization

Table 1.2: Summary of Perspectives

data source with other conventional sources, so that the end user can interact with a single, unified database. Furthermore, one would like a declarative system, where the end user describes the needs, and the system dynamically figures out what crowd data to obtain and how it must be obtained, and how it must be integrated with other data. Deco is a a database system that answers declarative queries posed over stored relational data, the collective knowledge of the crowd, as well as other external data sources. While declarative access to crowd data is appealing and natural, there are many challenges. For instance, it is not obvious (a) how the schema designer can specify mechanisms using which data can be obtained from the crowd (b) what the crowdsourced database system should store — cleansed, or uncleansed data (c) how the query processor should deal with the complexity of human data sources — for instance, the latency of humans responses, the resolution of disagreements, and the satisfaction of constraints, e.g., latency, cost, or accuracy.

1.2 Overview of Related Perspectives

In this section, to situate this thesis in relation to other work on crowdsourcing, we present a brief overview of how different academic communities approach crowdsourcing research, summarized in Table 1.2. We defer a detailed study of related work to Chapter 10, where the perspectives of these communities will be considered in greater depth. (Work related to specific chapters will be presented as part of those chapters.)

The Human Computer Interaction (HCI) community has been focused on developing new platforms for interaction with human workers. This community was the first to design a toolkit for harnessing crowdsourcing through programs [130]. More recently, this community has studied novel interfaces for collaboration (for editing [41] or planning [126]) and assistance (for disabled people [45] or photographers [40]).

The Machine Learning (ML) community views humans as providers of training data for ML algorithms. There has been a lot of work in Active Learning [169] on selecting training examples to label to best improve ML. There has also been some recent work on applying ML to optimize crowdsourcing. For instance, ML has been used to estimate the quality of workers once data has been collected, for example, see [162].

The Social Science community has primarily used crowdsourcing for behavioral research, for example, see [118, 137], while the Game Theory community has looked into incentivizing human workers better, via games [183], and by varying pricing [55].

In our work, we view humans as (potentially error-prone) data processors or data sources. Our focus is on systematically optimizing the use of humans in data processing algorithms and systems.

Chapter 2

Background

In this chapter, we begin by providing a primer on crowdsourcing and crowdsourcing marketplaces as applicable to this thesis, followed by our perspective on humans as data processors, and then conclude with a description of notation that is used across chapters.

2.1 Crowdsourcing Mechanisms

There are many conflicting opinions [159] on what "crowdsourcing" actually means, and whether crowdsourcing is indeed the same concept as "human computation". We avoid this debate by re-defining crowdsourcing or human computation to mean the same thing:

> From [181]: *"Crowdsourcing (or Human Computation) is a paradigm that utilizes human processing power to solve problems that computers cannot yet solve."*

Crowdsourcing Marketplaces: There are a number of online crowdsourcing marketplaces where users post tasks, and specify a monetary compensation and time limit for those tasks. The canonical example of a crowdsourcing marketplace is Amazon's Mechanical Turk [14]; other examples include Samasource [18], ODesk [17], Clickworker [3], and Crowdflower [6]. There are estimated to be over 30 crowdsourcing marketplaces, and these marketplaces are growing rapidly, quadrupling in overall size in 2010 and 2011, with the total revenue reaching $400M US Dollars in 2011 [20].

The structure of marketplaces vary, but a representative workflow is as follows: Human workers (or simply humans, or workers) who are online in a marketplace can browse through tasks, choose the ones they are interested in, and work on them. Workers who solve the tasks within the time limit are awarded the monetary compensation specified by the task creator. The same task may be attempted simultaneously by multiple workers. If so, the workers work on the task independently, and each one is compensated.

In this thesis, we use “tasks” to mean what is conventionally referred to as microtasks, i.e., those that take a few minutes at most to complete (e.g., labeling an image) instead of long-running tasks that take days or weeks to complete (e.g., writing an essay).

Voluntary or Gaming-based Crowdsourcing: In addition to crowdsourcing marketplaces, there are other mechanisms to get humans to work on tasks. One such mechanism is to solicit volunteers to work on tasks for a worthy cause. As an example, volunteers were asked to help translate tweets during the Haiti earthquake [197], or help identify galaxies in astronomical images [160, 190]. Yet another mechanism relies on games [183]. In this mechanism, people play games for fun, without realizing that the games are, in fact, tasks that need to be solved.

Even though our focus is on crowdsourcing marketplaces, the optimized crowd-powered algorithms developed in this thesis can also be used in voluntary or gaming mechanisms, since there is still a limited budget of “human attention” that those mechanisms have that can be treated as analogous to monetary cost in crowdsourcing marketplaces.

2.2 Interacting with a Crowdsourcing Marketplace

We now describe how our crowd-powered algorithms and systems (as described in Section 1.1.1 and 1.1.2) interact with a crowdsourcing marketplace. An informal diagram of the interaction is shown in Figure 2.1. Our algorithms and systems operate on data items, e.g., images, videos, or text, and construct tasks to be asked to humans. These tasks are expressed using HTML (hosted externally in our case), and posted on the crowdsourcing marketplace using an API specific to the marketplace. (Recall, as we discussed in the previous section, while posting the task, we need to also specify the monetary reward and the time limit, but our focus in this section is on tasks.) These tasks are answered by humans independently. Once answers to these tasks are provided back to the crowd-powered algorithm or system, the algorithm or system may choose to issue additional tasks once again, or may instead terminate.

Since humans may be concurrently working on different tasks, we can view the algorithm or system as making humans work on tasks “in parallel”, waiting for their responses, then making humans work on additional tasks “in parallel”, and so on. However, note that the system can in fact issue new tasks to the crowdsourcing marketplace before the outstanding ones are complete.

We provide examples of these tasks next.

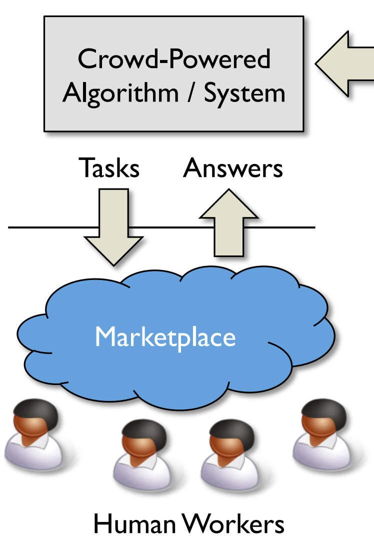


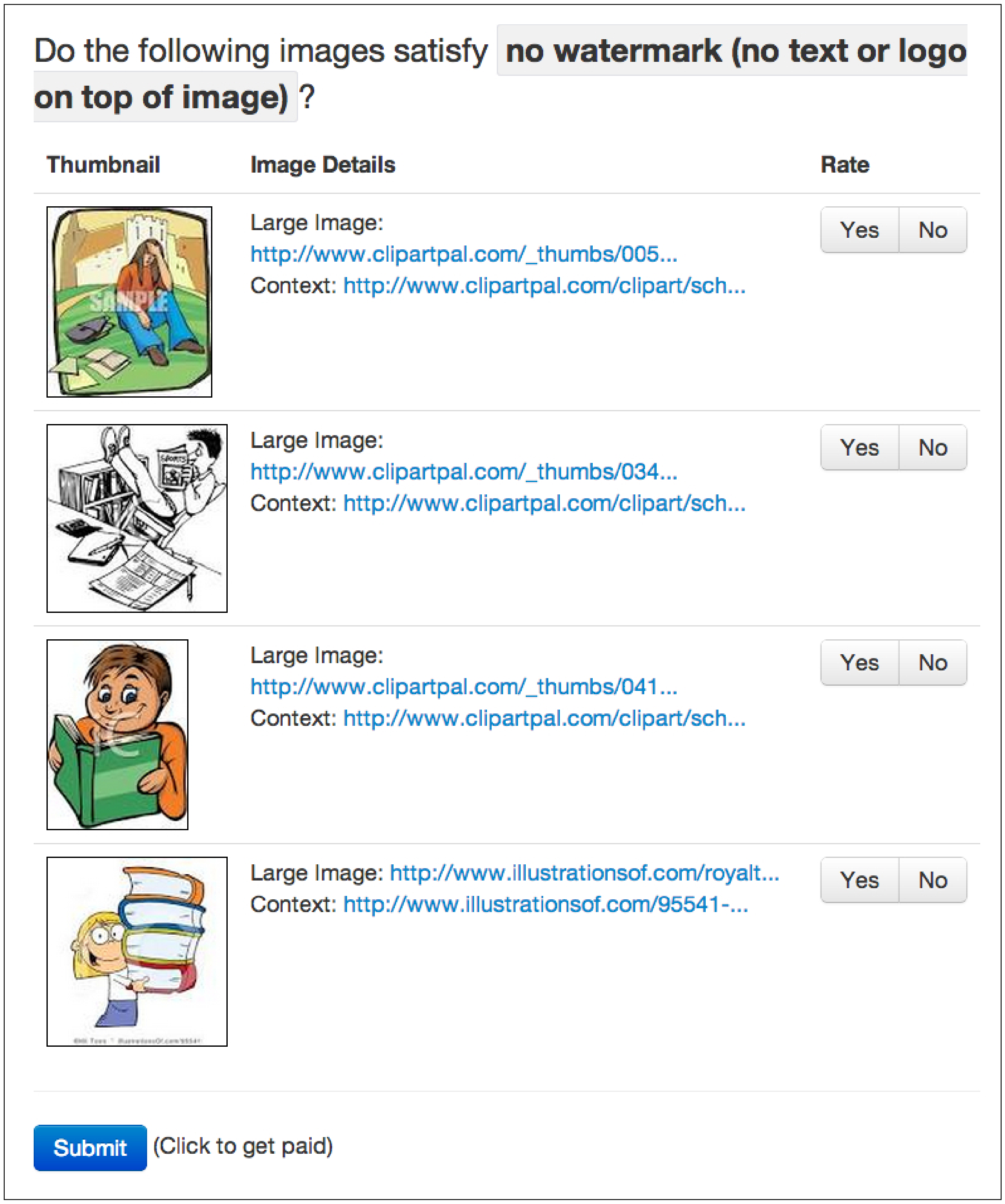
Figure 2.1: Interacting with a Marketplace

Example Tasks

We show two example tasks, as seen by workers, in Figures 2.2, and 2.3. Once a human worker completes either of these tasks, he/she can click on the submit button to get compensated for their effort.

The first task consists of a batch (specifically, four) of "filtering questions", that is, questions checking if specific items (in this case, images) satisfy a given filtering predicate (in this case, whether they do or do not have a watermark). In this task, notice that only the last image does not have a watermark; while it is easy to make out the watermark in the first and third images, the watermark in the second image is much harder to distinguish from the rest of the image (that is, human workers may be more likely to make a mistake on this image instead of the other images). Thus, ensuring that we get correct answers for filtering questions on some items may be more difficult than others.

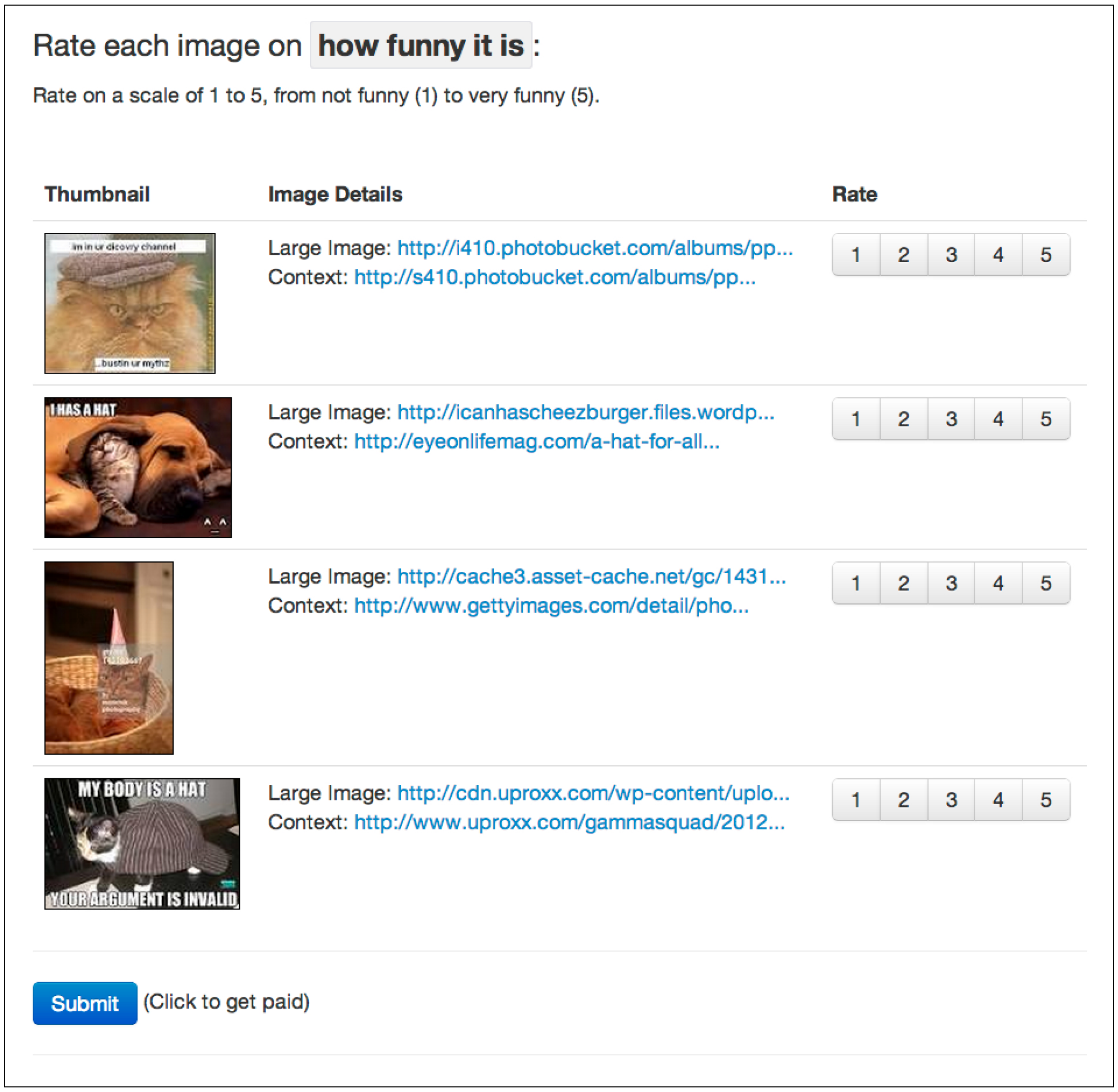
The second task consists of a batch of four "rating questions", that is, questions requesting ratings for specific items (once again, images) for the predicate "how funny it is". In this task, since humor is subjective, different human workers may have different opinions on what constitutes a funny image; furthermore, some workers may be much more generous than others in providing high ratings. Thus, given various worker answers, inferring the true rating for each image is not trivial.


Do the following images satisfy **no watermark (no text or logo on top of image)** ?

Thumbnail	Image Details	Rate
	Large Image: http://www.clipartpal.com/_thumbs/005... Context: http://www.clipartpal.com/clipart/sch...	Yes No
	Large Image: http://www.clipartpal.com/_thumbs/034... Context: http://www.clipartpal.com/clipart/sch...	Yes No
	Large Image: http://www.clipartpal.com/_thumbs/041... Context: http://www.clipartpal.com/clipart/sch...	Yes No
	Large Image: http://www.illustrationsof.com/royalt... Context: http://www.illustrationsof.com/95541-...	Yes No

(Click to get paid)

Figure 2.2: Filtering Task



Rate each image on **how funny it is** :

Rate on a scale of 1 to 5, from not funny (1) to very funny (5).

Thumbnail	Image Details	Rate
	Large Image: http://i410.photobucket.com/albums/pp... Context: http://s410.photobucket.com/albums/pp...	1 2 3 4 5
	Large Image: http://icanhascheezburger.files.wordp... Context: http://eyeonlifemag.com/a-hat-for-all...	1 2 3 4 5
	Large Image: http://cache3.asset-cache.net/gc/1431... Context: http://www.gettyimages.com/detail/pho...	1 2 3 4 5
	Large Image: http://cdn.uproxx.com/wp-content/uplo... Context: http://www.uproxx.com/gammasquad/2012...	1 2 3 4 5

Submit (Click to get paid)

Figure 2.3: Rating Task

Humans as Data Processors

In general, we treat humans answering questions as function or subroutine calls with data items as arguments. For instance, in the questions that comprise the tasks above, the arguments are images, and the function or subroutine call takes as input an image, and: (a) returns a boolean YES/NO output for the first type of question (b) returns a rating 1—5 for the second type of question. This abstraction enables us to treat human involvement as potentially error-prone, costly, and time-consuming "data processors" that can be called from within crowd-powered systems and algorithms. Naturally, the latency, cost, and accuracy of each function or subroutine call will depend on the specific human (some humans may be better than others, and as we saw for the rating case, different humans may have different biases), and on the item(s) under consideration (some items may be more prone to errors than others like in the filtering case).

Of course, treating humans as data processors is certainly a simplification: humans are incredibly complex, and simple models or abstractions are likely to not be accurate depictions of human behavior. Nevertheless, even with these simple models we will find that (a) the problems are still challenging (b) we can obtain substantial benefits in practice. It remains to be seen if more intricate abstractions of human involvement will provide additional benefits.

Questions Answerable by Humans

The filtering and rating questions described above are just two types of questions we consider. The entire list of question types we consider in this thesis is shown in Table 2.1 annotated with the respective chapters. Most of these questions take as input one or more items from domain $\mathcal{I}$. They may also take other inputs (e.g., for categorization, we take as input a category from domain $\mathcal{C}$).

These questions types are sufficient and powerful enough to capture all human interaction in our systems and algorithms. Furthermore, these questions are amenable to analysis and optimization, unlike more free-form questions, for example, "write an essay on topic X".

Given a question or a batch of questions, there are many interesting issues in designing the interface to improve worker satisfaction and answer quality (including pop-out, anchoring [118]). While these considerations are important and useful, they are not the focus of study in this thesis. In this thesis, we leverage prior work to build pre-optimized interfaces for individual questions, and our goal is to use the pre-optimized question templates or interfaces in the best manner to minimize cost, latency, and error in applications.

Problem	Chapter No.	Question Type	Signature
Filter, Find	3, 4, 5	"Does this item satisfy the filter?"	$(\mathcal{I} \rightarrow \{0, 1\})$
Advanced Filter	4	"What is the score of this item?"	$(\mathcal{I} \rightarrow \{0 \ldots u\})$
Maximum	6	"Which of these items has higher quality?"	$(\mathcal{I} \times \mathcal{I} \rightarrow \{0, 1\})$
Categorize	7	"Does this item fall under this category"	$(\mathcal{I} \times \mathcal{C} \rightarrow \{0, 1\})$

Table 2.1: Data Processing Question Types

2.3 Notation

For easy reference, we summarize the notation used in the thesis in Tables 2.2 and 2.3. The notation that applies to the entire thesis forms the first block of rows, while the subsequent blocks list notation that applies to specific chapters. The table is presented here for the reader's future reference; we do not expect the reader to understand all the symbols at this point.

Relevant Chapters	Quantity	Notation
Entire Thesis	Set of Items	$\mathcal{I}$
	Number of Items	n
	Individual Item	I
	Multiset of Questions Asked	$\mathcal{Q}$
	Error Rate	e
	Error in Output	E
	Monetary Cost	C
	Latency	T
	Additional Number of Questions	b
	Probabilities	p
Filtering, Filtering Generalizations, and Peer Evaluation (Chapters 3, 4, 9)	Filters	$f_1, \ldots, f_l$
	Filtering Strategy	$\mathcal{F}$
	Cost Threshold	m
	Intrinsic Value of item	V
	State Variable	S, R
	Selectivity	s
	Fractional Paths	$path$
	Probabilistic Decision	$a_{pass/fail/cont}$
	Workers	$w_1, \ldots, w_r$
	Ratings	$0 \ldots u$
	Distinct Selectivities	l
	Number of Discrete Difficulty Values	d
	Discretization Factor	δ
Finding (Chapter 5)	Desired Items	k
	State of Knowledge	$\mathcal{SK}$
	Filtering Algorithm	$\mathcal{A}$
	Approximation factor	α, β
	Cost to Next Item	C_{next}
	Number of Questions	a
	Number of Phases	q

Table 2.2: Table of Notation

Relevant Chapters	Quantity	Notation
Maximum (Chapter 6)	Inherent Quality	c_i
	Permutation	π, π_d
	Vote Matrix and Votes	W, w
	Scoring Functions	$score, final$
	Vote Graph	$G_v(V, A)$
	Damping Factor	γ
	Degree of Vertex	d
Categorization (Chapter 7)	Taxonomy	$G(V, E), \|V\| = n$
	Nodes in Graph	u, v
	Set of Nodes	U
	Answers to Questions	$\mathsf{q}(u, U)$
	Preceding Set, Reachable Set	pset, rset
	Candidate Set of Categories	cand
DataSift (Chapter 8)	Query	Q, TQ
	Number of Humans	h
	Number of Suggested Reformulations	s
	DataSift Components	G, F, R, W, S
	Number of Positive and Negative Responses	x, y
	Number of Desired Items	k
	Weights	w
	Number of Items Retrieved	n

Table 2.3: Table of Notation (continued)

Chapter 3

Algorithm 1: Filtering

3.1 Introduction

In this chapter, we develop algorithms that design optimized strategies for crowd-powered filtering[1]. For example, we may have a data set of images, and we may want to find all images that satisfy a given set of properties. We need to use humans to decide if the images have the particular properties. We use the term *filter* for each of the properties we wish to check. For instance, one filter could be "image shows a scientist," and another could be "people in image are looking at camera." If we apply both filters, we should obtain images of scientists looking at the camera. The emphasis in this chapter is on applying a single filter, however, we also provide extensions to conjunctions of filters.

This chapter, being our first technical chapter, will also serve to formalize our model of human computation (i.e., abstracting human involvement as calls to error-prone data processors), as well our objectives of cost and error.

At first sight, the problem of checking which of a set of items satisfy a filter seems trivial: Simply take each item in turn, and ask a human a *question*: Does this item satisfy the filter? The solution is the subset of items that received a positive answer. Since humans may make mistakes, we may not get a desirable solution with such a simple strategy. We may instead show each item to multiple people and somehow combine the answers, e.g., if 2 out of 3 vote yes, then we say the item satisfies the filter. But what is the right way to combine the votes? And how many questions should we ask per item? Does it matter if humans are more likely to make false positive mistakes, as opposed to false negatives? And what if we know a priori that most items are very likely to satisfy the filter? How does

[1]This chapter is adapted from our paper [152], published at ACM SIGMOD 2012, written jointly with Aditya Ramesh, Hector Garcia-Molina, Hyunjung Park, Neoklis Polyzotis, and Jennifer Widom

this knowledge change our filtering approach? Furthermore, we may not have an unlimited budget or time to do our filtering. So how do we select a filtering strategy that, for instance, minimizes costs (e.g., total number of questions asked) while keeping overall error below a desired threshold, or vice versa?

Filtering with humans is reminiscent of statistical hypothesis testing [187]. While the underlying model is similar to ours (our hypothesis is that an item satisfies a filter), the fundamental difference with our work is in the optimization criteria: With hypothesis testing, one wishes to ensure that the decision on each item meets an error or cost bound. In our case, on the other hand, we filter a large set of items, and our bounds are on the *overall* error or cost. In Section 3.2 we argue that these optimization criteria are often desirable in crowdsourcing for large data sets. Furthermore, in Section 3.7 we show that use of our criteria can lead to substantial cost savings over the traditional, more conservative approach.

Emerging crowdsourcing applications [31, 34, 130, 173, 195] have implemented various types of filtering (e.g., majority voting), but have not studied how to implement optimal filtering strategies. Prior work on using human feedback for database information integration [139] (which we will show is worse than our strategies) uses a heuristic approach to solve the problem. Previous work has also looked at the problem of deciding how and when to obtain labels for machine learning [81, 171, 188], but not the problem of minimizing cost. We revisit related work in more detail in Section 3.8.

Filtering is critical in crowd-powered database systems, since it can be used for the implementation of the selection operator that is present in most database queries. In addition, filtering can be used for data cleaning (when the decision of whether a data item is valid or not requires human input) and for data generation and population (when humans provide missing data items or attribute values). Furthermore, the optimization objectives in our approach are designed specifically to give the most savings over very large data sets, a typical scenario in database systems.

Filtering is also of independent interest in human computation. Indeed, almost every crowdsourcing application performs some form of filtering [31, 34, 86, 130, 135, 173, 195]. For example, detecting spam websites in a set of websites, or categorizing a set of images into a few categories (both common tasks on Amazon's Mechanical Turk [14]), are instances of the single-filter problem. Relevance judgments for search results [31], where the task is to determine whether a web-page is relevant or not for a given web-search query, is another instance.

3.1.1 Outline of Chapter

Here is the outline of this chapter:

- We identify the interesting dimensions of the single filter problem, which in turn reveal which parameters of the problem can be constrained or optimized in a meaningful manner. We formulate five different versions of the problem under constraints on these parameters. We also develop a novel grid-based representation to reason about strategies for the problems that we describe. (Section 3.2)
- For deterministic strategies, we develop an algorithm that designs an approximately optimal deterministic strategy (for the cases where we definitely want a deterministic strategy due to simplicity of representation or not having to generate random numbers). (Section 3.3)
- For probabilistic strategies, we develop a linear programming solution to produce the optimal probabilistic strategy. (Section 3.4)
- We discuss other versions of the problem constructed by constraining cost and accuracy in various ways. (Section 3.5)
- We describe how our techniques can be extended to the problem of multiple filters. (Section 3.6)
- We show through experiments that our algorithms perform exceedingly well compared to standard statistics approaches as well as other naive and heuristic approaches. (Section 3.7)

3.2 Preliminaries

For the majority of this chapter, we consider the single filter problem, but we do generalize to multiple filters in Section 3.6. Also, we assume that our filters are "binary" (i.e., they simply return YES or NO), rather than the "n-ary" filtering case, where the filters may return one out of a set of n disjoint independent alternatives. (This "n-ary" problem is relevant when we are classifying each item into exactly one out of a set of indepenent classes, e.g., assigning colors.) Our techniques generalize in a straightforward fashion to the "n-ary" filtering problem.

Filtering into n disjoint independent classes is different from and simpler than the generalization of scoring, which we consider in Chapter 4, where we score items as being one out of $1 \ldots n$. In the scoring problem, the n alternatives are not independent of each other (e.g., score of 1 is closer to a score of 2 than a score of 5). Lastly, filtering into n independent classes is different from and simpler than the generalization of categorization into a taxonomy, which we consider in Chapter 7. In the latter case, once again, the alternatives are not independent of each other, and are in fact related to each other via a hierarchical relationship.

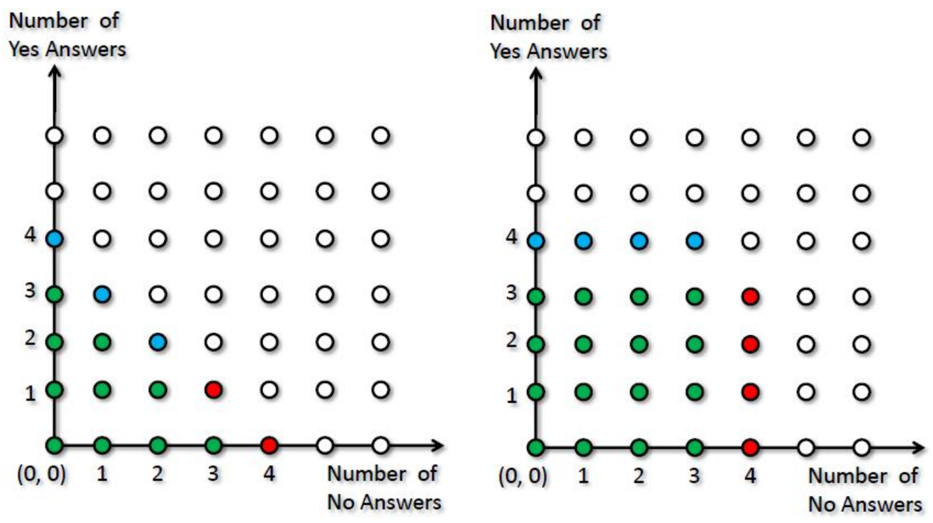


Figure 3.1: (a) Representation of a Triangular Strategy (b) Representation of a Rectangular Strategy

3.2.1 Formal Definitions

We are given a set of items $\mathcal{I}$, where $|\mathcal{I}| = n$. We introduce a random variable V that controls whether an input item satisfies the filter ($V = 1$) or not ($V = 0$). The selectivity of our filter, s, gives us the probability that $V = 1$ (over all possible items). The selectivity may be estimated by sampling on a small number of items or by using prior history. (The sampling approach is commonly used by query optimizers to estimate the size of a selection operator.)

However, we assume there is no automated mechanism to determine whether an item satisfies the filter or not. The only type of action the algorithm can perform on an item is to ask a human a *question*. The human can tell us YES (meaning that he/she thinks the item satisfies the filter) or NO. The human can make mistakes, and in particular:

- The false positive rate is: $Pr[\text{answer is YES}|V = 0] = e_0$
- The false negative rate is: $Pr[\text{answer is NO}|V = 1] = e_1$

(We use two distinct probabilities of error because our experience with Mechanical Turk indicates that the false positive rate and false negative rate for a filter can be very different.) We can ask different humans the same question to get better accuracy, and we assume that their errors are independent. (Thus we assume all humans are able to answer the question with the same degree of accuracy.) As we will see in the rest of this chapter, filtering is a challenging problem even after making this assumption.

We consider the case of correlated answers, differing degrees of competence of humans, and

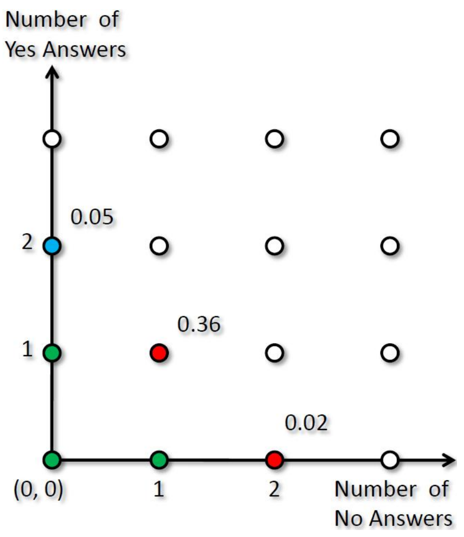


Figure 3.2: Error at Terminating Points vs. Overall Error

differing costs for humans with different expertise in the next chapter.

A *strategy* $\mathcal{F}$ is a computer procedure that takes as input any one item, asks one or more humans questions on that same item, and eventually outputs either "Pass" or "Fail". A Pass output represents a belief that the item satisfies the filter, while Fail represents the opposite. Of course, a strategy can also make mistakes, and our job will be to design good strategies, i.e., ones that "make few mistakes without asking too many questions". We will express this goal more formally later. Note that in relation to Figure 2.1, the strategy is the crowd-powered algorithm. In this chapter we also use the term "algorithm" when describing our procedure that, given parameters and constraints, generates a strategy.

The state of processing of an item can be completely represented using the pair (x, y), where x is the number of NOs received so far, and y is the number of YESs received so far. Therefore, a strategy can be visualized or represented using a two-dimensional grid like the one in Figure 3.1(a). This two-dimensional grid captures what the strategy does at each state. The Y axis represents the number of YES answers obtained so far from humans, while the X axis is the number of NO answers so far.

A grid point at (x, y) determines what the strategy does after x NOs and y YESs have been received from humans: A blue grid point indicates that the strategy outputs Pass at this point, i.e., $\mathcal{F}(x, y) = \text{Pass}$, while a red point indicates a Fail decision, i.e., $\mathcal{F}(x, y) = \text{Fail}$. We call a point that is either blue or red a *termination point*. At a green point no decision is made and the strategy issues another question, and thus moves to either $(x, y+1)$ (if an additional YES is received) or to $(x+1, y)$ (if a NO is received). We call green points *continue points*, i.e., we continue to ask questions, and we have $\mathcal{F}(x, y) = \text{Cont}$. Notice that we are overloading our use of the term "strategy" to mean two things: (a) the code that operates on an item, and eventually returns Pass or Fail, (b) a decision function, defined over all reachable states, that takes as input a state (x, y), and outputs a decision for that state (Pass, Fail, Cont). The use will be clear from the context.

Overall, the evaluation of an item starts at $(0, 0)$ (no questions have been asked), and moves through the grid until hitting either a blue or red grid point; the white points are not reachable under any circumstances. Our example in Figure 3.1(a) depicts a strategy that always asks a fixed number of questions, in this case 4. Thus, the termination points are along the $x + y = 4$ line. As a second example of a strategy, consider Figure 3.1(b). In this strategy, we stop as soon as we get four YESs or four NOs. Thus, the total number of questions will vary between 4 and 7.

The strategies of Figure 3.1(a) and 3.1(b) are *deterministic*, i.e., the output is the same for the same sequence of answers. In Section 3.4 we discuss a generalization of deterministic strategies, called *probabilistic* strategies where the decision at each grid point is probabilistic. For instance, at a particular grid point we may determine Pass with probability 0.3 and Fail with probability 0.1, and may continue asking questions with probability 0.6. Thus we represent each grid point by a triple like $\mathcal{F}(x, y) = (0.3, 0.1, 0.6)$. For deterministic strategies, the only triple values allowed are $(0, 0, 1)$ (green point), $(1, 0, 0)$ (blue point) or $(0, 1, 0)$ (red point).

Our strategies are defined to be naturally *uniform*, i.e., the same strategy is applied to each item in the dataset, and *complete*, i.e., the strategy tells us what to do at each reachable point in the grid. In addition, we want our strategies to be *terminating*, i.e., the strategy always terminates in a finite number of steps, no matter what sequence of YES/NO answers are received to the questions (a terminating strategy effectively corresponds to a closed shape around the origin). Note that the strategies in Figures 3.1(a) and 3.1(b) are terminating. We will enforce a termination constraint in our problem formulations in Section 3.2.3. Lastly, we want our strategies to be *fully determined in advance*, so that we can plan beforehand how we would like to expend cost across all items, and so that we do not need to do any computation on the fly while the answers are received. Thus, we present algorithms that compute entire strategies in advance. However, our algorithms may also be used to compute strategies on-the-fly: The strategy may be computed while the answer to the first question for all the items is being requested from the crowd. Subsequently, the computed strategy can then be applied on all the items for the next batch of questions.

Strategies are therefore instances of the well-studied Markov Decision Processes (MDPs) [175]. MDPs are represented by a set of states (here, all possible S), possible decisions for each state (here, Pass, Fail, Cont), and a probability distribution over next states when a given decision is taken at a given state (here, when Cont is the decision taken at state S, then the next state can be one of at most 2 states—corresponding to a YES/NO answer from one of the r workers; when Pass/Fail is the decision taken, then the strategy terminates). MDPs have a single reward function (or metric) associated with each state. In our case, since we are considering multiple metrics — cost and accuracy — the standard value and policy iteration techniques used for MDPs do not apply. To enable our chapter to be self-contained, we describe our approach without using the MDP formalism.

3.2.2 Metrics

To determine which strategy is best, we study two types of metrics, involving error and cost. We start by defining two quantities, given a strategy:

- $p_1(x, y)$ is the probability that the strategy reaches point (x, y) and the item satisfies the filter $(V = 1)$; and
- $p_0(x, y)$ is the probability that the strategy reaches point (x, y) and the item does not satisfy the filter $(V = 0)$.

Below we give a simple example to illustrate these quantities, and in Section 3.2.4 we show how to compute them in general.

We can now define the following metrics:

- $E(x, y)$ (only of interest when $\mathcal{F}(x, y) \neq$ Cont) is the probability of error given that the strategy terminated at (x, y). If $\mathcal{F}(x, y) =$ Pass, then an error is made if $V = 0$, so $E(x, y)$ is $p_0(x, y)$ divided by the probability that the strategy reached (x, y) (i.e., divided by $p_0(x, y) + p_1(x, y)$). The Fail case is analogous, so we get:

$$E(x, y) = \begin{cases} \frac{p_0(x,y)}{[p_0(x,y)+p_1(x,y)]} & \text{if } \mathcal{F}(x, y) = \text{Pass} \\ \frac{p_1(x,y)}{[p_0(x,y)+p_1(x,y)]} & \text{if } \mathcal{F}(x, y) = \text{Fail} \\ 0 & \text{else} \end{cases} \tag{3.1}$$

- E is the expected error across all termination points. That is:

$$E = \sum_{(x,y)} E(x, y) \times [p_0(x, y) + p_1(x, y)]. \tag{3.2}$$

- $C(x, y)$ (only of interest when $\mathcal{F}(x, y) \neq$ Cont) is the number of questions used to reach a decision at (x, y), i.e., simply $x + y$. We consider $C(x, y)$ to be zero at all non-termination points.

- C is the expected cost across all termination points, i.e.,:

$$C = \sum_{(x,y)} C(x,y) \times [p_0(x,y) + p_1(x,y)] \tag{3.3}$$

To evaluate our n items using the same strategy, we will incur an expected cost of nC. Notice that we are not explicitly considering latency as part of our metrics. We will consider incorporating latency constraints in Chapter 4.

To illustrate the metrics above, consider the simple deterministic strategy in Figure 3.2, and assume that $s = 0.5$, $e_1 = 0.1$, $e_0 = 0.2$. At each termination point (x, y) we show $E(x, y)$. For example, $E(0, 2) = 0.05$. This number can be interpreted as follows: In 5% of the cases where we end up terminating at $(0, 2)$, an error will be made. In the remaining 95% of the cases a correct (Pass) decision will be made. To compute $E(0, 2)$, we need the p_0 and p_1 values. Since there is only a single way to get to $(0, 2)$ (two consecutive YESs) the computation is simple: $p_0(0, 2)$ is $(1 - s)$ (V must be 0) times e_0 (i.e., 0.02) squared (thus, $p_0(0, 2) = (1 - s) \times e_0^2$). Similarly, $p_1(0, 2) = s \times (1 - e_1)^2 = 0.5 \times 0.9 \times 0.9 = 0.405$. Thus, $E(0, 2) = 0.02/[0.02 + 0.405] = 0.05$. When there are multiple ways to get to a point (e.g., for $(1, 1)$) the p_0, p_1 computation is more complex, and will be discussed in Section 3.2.4.

The expected error E for this sample strategy is the error across all termination points. In this case, it turns out that $E = 0.115$. Notice the difference between the overall error E and the individual termination point errors: The error at $(1, 1)$ is quite high, but because it is not very likely that we end up at $(1, 1)$, that error does not contribute as much to the overall E.

$$p_0(x,y) = \begin{cases} p_0(x-1,y)(1-e_0) + p_0(x,y-1)e_0 & \text{if } \mathcal{F}(x,y-1) = \text{Cont} \wedge \mathcal{F}(x-1,y) = \text{Cont} \\ p_0(x,y-1)e_0 & \text{if } \mathcal{F}(x,y-1) = \text{Cont} \wedge \mathcal{F}(x-1,y) \neq \text{Cont} \\ p_0(x-1,y)(1-e_0) & \text{if } \mathcal{F}(x,y-1) \neq \text{Cont} \wedge \mathcal{F}(x-1,y) = \text{Cont} \\ 0 & \text{if } \mathcal{F}(x,y-1) \neq \text{Cont} \wedge \mathcal{F}(x-1,y) \neq \text{Cont} \end{cases}$$

$$p_1(x,y) = \begin{cases} p_1(x,y-1)(1-e_1) + p_1(x-1,y)e_1 & \text{if } \mathcal{F}(x,y-1) = \text{Cont} \wedge \mathcal{F}(x-1,y) = \text{Cont} \\ p_1(x,y-1)(1-e_1) & \text{if } \mathcal{F}(x,y-1) = \text{Cont} \wedge \mathcal{F}(x-1,y) \neq \text{Cont} \\ p_1(x-1,y)e_1 & \text{if } \mathcal{F}(x,y-1) \neq \text{Cont} \wedge \mathcal{F}(x-1,y) = \text{Cont} \\ 0 & \text{if } \mathcal{F}(x,y-1) \neq \text{Cont} \wedge \mathcal{F}(x-1,y) \neq \text{Cont} \end{cases}$$

Figure 3.3: Recursive Equations for p_0 and p_1 (We define $\mathcal{F}(a, b) = \text{Pass}$ when $a < 0$ or $b < 0$.)

3.2.3 The Problems

Given input parameters selectivity s, false positive rate e_1, and false negative rate e_0, the search for a "good" or "optimal" strategy $\mathcal{F}$ can be formulated in a variety of ways. Since we want to ensure that the strategy terminates, we enforce a threshold m on the maximum number of questions we can ask for any item (which is nothing but a maximum budget for any item that we want to filter).

We start with the problem we will focus on in this chapter:

Problem 3.2.1 (Core) *Given an error threshold τ and a budget threshold per item m, find a strategy $\mathcal{F}$ that minimizes C under the constraint $E < \tau$ and $\forall(x, y)$ $C(x, y) < m$.*

An alternative would be to constrain the error at each termination point:

Problem 3.2.2 (Core: Per-Item Error) *Given an error threshold τ and a budget threshold per item m, find a strategy $\mathcal{F}$ that minimizes C under the constraint $\forall(x, y)$ $E(x, y) < \tau$ and $C(x, y) < m$.*

In Problem 3.2.2 we ensure that the error is never above threshold, even in very unlikely executions. This formulation may be preferred when errors are disastrous, e.g., if our filter is checking for patients with some disease, or for defective automobiles. However, in other cases we may be willing to tolerate uncertainty in some individual decisions (i.e., high $E(x, y)$ at some points), in order to reduce costs, as long as the overall error E is acceptable. For instance, say we are filtering photographs that will be used by an internet shopping site. In some unlikely case, we may get 10 YES and 10 NO votes, which may mean we are not sure if the photo satisfies the filter. In this case we may prefer to stop asking questions to contain costs and make a decision, even though the error rate at this point will be high no matter what we decide. In Section 3.7 we study the price one pays (number of questions) for adopting the more conservative approach of Problem 3.2.2 under the same error threshold.

Instead of minimizing the overall cost, one could minimize the *maximum* cost at any given point. In this case, the strategy will issue a fixed number of questions per item (corresponding to the maximum cost), and hence the final outcome can be computed only after all answers are received. This allows for a more compact representation of the strategy, as it is not necessary to keep track of the number of YES and NO answers obtained in between. In addition, the optimal strategy can be computed quickly, and hence the next two formulations are preferable when time and compactness of representation are more important than cost.

Problem 3.2.3 (Maximum Cost) *Given an error threshold τ, find a strategy $\mathcal{F}$ that minimizes the maximum value of $C(x, y)$ (over all points), under the constraint $E < \tau$.*

Another variation is to minimize error, given some constraint on the number of questions. We specify two such variants next. In the first variant we have a maximum budget for each item we want to filter, while in the second variant we have an additional constraint on the expected overall cost.

Problem 3.2.4 (Error) *Given a budget threshold per item m, find a strategy $\mathcal{F}$ that minimizes E under the constraint $\forall(x, y)\ C(x, y) < m$.*

Problem 3.2.5 (Error - II) *Given a budget threshold per item m and a cost threshold α, find a strategy $\mathcal{F}$ that minimizes E under the constraints $\forall(x, y)\ C(x, y) < m$ and $C < \alpha$.*

3.2.4 Computing Probabilities at Grid Points

In this subsection we show how to compute the p_0 and p_1 values defined in the previous subsection. We focus on a deterministic strategy (probabilistic strategies are discussed in Section 3.4).

We compute the p values recursively, starting at the origin. To start with, note that $p_0(0,0)$ is $(1-s)$ and $p_1(0,0)$ is s. (For instance, $p_0(0,0)$ is the probability that the item does not satisfy the filter and the strategy visits the point $(0,0)$ — which it has to.) We can then derive the probability p_0 and p_1 for every other point in the grid as shown in Figure 3.3. (Recall that $\mathcal{F}(x, y) \neq \text{Cont}$ means that (x, y) is a termination point, either Pass or Fail.)

To see how these equations work, let us consider the first case for p_0, where neither $(x-1, y)$ nor $(x, y-1)$ are termination points. Note that we can get to (x, y) in two ways, either from $(x, y-1)$ on getting an extra YES, or from $(x-1, y)$ on getting an extra NO. Thus, the probability of an item not satisfying the filter and getting to (x, y) is the sum of two quantities: (a) the probability of the item not satisfying the filter and getting to $(x, y-1)$ and getting an extra YES, and (b) the probability of the item not satisfying the filter and getting to $(x-1, y)$ and getting an extra NO. The probability of getting an extra YES given that the item does not satisfy the filter is precisely e_0, and the probability of getting a NO is $(1-e_0)$. We can write similar equations for p_1, as shown in the figure.

Given values for p_0 and p_1, we can use the definitions of Section 3.2.1 to compute the errors and costs at each point, and the overall error and cost. Note that we do not have to compute the p values at all points, but only for the reachable points, as all other points have zero error $E(x, y)$ and cost $C(x, y)$.

3.3 Deterministic Strategies

This section develops algorithms that find effective deterministic strategies for Problem 3.2.1.

3.3.1 The Paths Principle

Given a deterministic strategy, using Equations 3.1, 3.2, 3.3 and Figure 3.3 given earlier, it is easy to see that the following theorem holds:

Theorem 3.3.1 (Computation of Cost and Error) *The expected cost and error of a strategy can be computed in time linear in the number of reachable grid points.*

Based on the above theorem, we have a brute force algorithm to find the best deterministic strategy, namely by examining strategies corresponding to all possible assignments of Pass, Fail or Continue (i.e., continue asking questions) to each point in $(0,0)$ to (m,m). (There are 3^{m^2} such assignments.) Evaluating cost and error for each strategy takes time $O(m^2)$ using the recursive equations. We select the one that satisfies the error threshold, and minimizes cost. We call this algorithm naive3.

Note that some of these strategies are not terminating. However, termination can also be checked easily for each strategy considered in time proportional to the number of reachable grid points in the strategy. (If the p_0 and p_1 values at all points on the line $x+y=m'$, for some $m' \leq m$ is zero, then the strategy is terminating.)

Theorem 3.3.2 (Best Strategy: Naive3) *The* naive3 *algorithm finds the best strategy for Problem 3.2.1 in $O(m^2 3^{m^2})$.*

We are able to reduce significantly the search space of the naive algorithm by excluding the provably suboptimal strategies. Our exclusion criterion is based on the following fundamental theorem.

Theorem 3.3.3 (Paths Principle) *Given s, e_1, e_0, for every point (x,y), the function*

$$g(x,y) = \frac{p_0(x,y)}{p_0(x,y)+p_1(x,y)}$$

is a function of (x,y), independent of the particular (deterministic or probabilistic) strategy.

Proof 3.3.4 *Consider a single sequence of x NO answers and y YES answers. The probability that an item satisfies the filter, and gets the particular sequence of x NO answers and y YES answers is precisely $a = s \times e_1^x \times (1-e_1)^y$, while the probability that an item does not satisfy the filter and gets the same sequence is $b = (1-s) \times e_0^y \times (1-e_0)^x$. The choice of strategy may change the number of such paths to the point (x,y), however the fraction of p_0 to p_0+p_1 is still $b/(a+b)$. (Note that each path precisely adds a to p_1 and b to p_0. Thus, if there are r paths, $p_0/(p_0+p_1) = (r \times b)/[r \times (a+b)] = b/(a+b)$.) The proof also generalizes to probabilistic strategies. (In that case, r can be a fractional number of paths.)*

Intuitively, this theorem holds because the strategy only changes the number of paths leading to a point, but the characteristics of the point stay the same.

Using the previous theorem, we have the result that in order to reduce the error, for every termination point (x, y), Passing or Failing is independent of strategy, but is based simply on $g(x, y)$.

Theorem 3.3.5 (Filtering Independent of Strategy) *For every optimal strategy $\mathcal{F}$, for every point (x, y), if $\mathcal{F}(x, y) \neq \mathsf{Cont}$ holds, then:*

- *If $g(x, y) > 1/2$, then $\mathcal{F}(x, y) = \mathsf{Fail}$*
- *If $g(x, y) < 1/2$, then $\mathcal{F}(x, y) = \mathsf{Pass}$*

Proof 3.3.6 *Given a strategy, let there be a point (x, y) for which $\mathcal{F}(x, y) = \mathsf{Pass}$, but $g(x, y) > 1/2$. Let the error be:*

$$E = E_0 + E(x, y)(p_0(x, y) + p_1(x, y))$$

(We split the error into two parts, one dependent on other terminating points, and one just dependent on (x, y).) Currently, since $\mathcal{F}(x, y)$ is Pass, $E(x, y)$ is $p_0(x, y)/(p_0(x, y) + p_1(x, y))$. Thus, $E = E_0 + p_0(x, y)$. If we change (x, y) to Fail, we get a new strategy $\mathcal{F}'$, with $E' = E_0 + p_1(x, y)$, then $E' < E$. Thus, by flipping (x, y) to Fail, we can only reduce the error. Similarly, if $g(x, y) < 1/2$ and $\mathcal{F}(x, y) = \mathsf{Fail}$ holds, we can only reduce the error by flipping it to Pass.

Thus, we only need to consider 2^{m^2} strategies, namely those where for each point, we can either set it to be a continue point or a termination point, and if it is a termination point, then using the previous theorem, we can infer whether it should be Pass or Fail. The algorithm that considers all such 2^{m^2} strategies is called naive2. Thus, we have the following theorem:

Theorem 3.3.7 (Best Strategy: Naive2) *The naive2 algorithm finds the best strategy for Problem 3.2.1 in $O(m^2 2^{m^2})$.*

3.3.2 Shapes and Ladders

In practice, considering all 2^{m^2} strategies is computationally feasible only for very small m. In this section, we design algorithms that only consider a subset of these 2^{m^2} strategies and thereby can only provide an approximate solution to the problem (i.e., the expected cost may be slightly greater than the expected cost of the optimal solution). The subset of strategies that we are interested in are

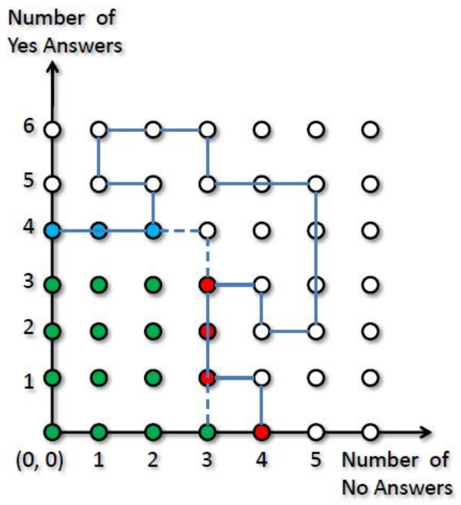


Figure 3.4: A Shape

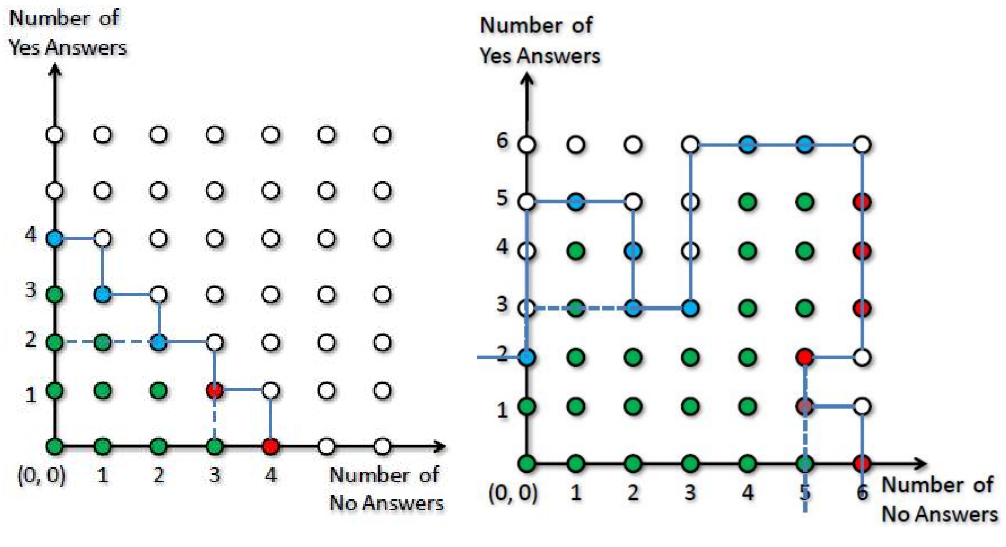


Figure 3.5: (a) Triangular Strategy Corresponds to a Shape (b) Ladder Shape Pruning

those that correspond to *shapes*. We will describe an efficient way of obtaining the best strategy that corresponds to a shape.

Shapes: A *shape* is defined by a connected sequence of (horizontal or vertical) segments on the grid, beginning at a point on the y-axis, and ending at a point on the x axis, along with a special point somewhere along the sequence of segments, called a *decision point*. We also assume that each segment intersects at most with two other lines, namely the ones preceding and following it in the sequence.

As an example, consider the shape in Figure 3.4 (ignore the dashed lines for now) This shape begins at (0, 4), has a sequence of 14 segments, and ends at (4, 0). The decision point (not shown in the figure) is (for example) at (5, 5). As seen in the figure, the segments are allowed to go in any direction (up/down or left/right).

Each shape corresponds to precisely one strategy, namely the one defined as follows:

- For each point in the sequence of segments starting at the point on the y axis, until and including the decision point, we color the point blue (i.e., we designate the point as Pass). In the figure, all points in the sequence of segments starting from (0, 4) until and including (5, 5) are colored blue.
- For each point in the sequence of segments starting at the point after the decision point, until and including the point on the x axis, we color the point red (i.e., we designate the point as Fail). In the figure, all points after (5, 5) on the sequence of segments are colored red.
- For all the points inside or on the shape that are reachable, we color them green; else we color them white. (Some points colored blue or red previously may actually be unreachable and will be colored white in this step.) In the figure, some of the blue points, such as (2, 5), (1, 5), (1, 6), . . ., (5, 5), and some of the red points (5, 4), . . ., (4, 3), and (4, 1) are colored white, while some of the unreachable points inside the shape, such as (3, 4) and (4, 4) are colored white as well. The reachable points inside the shape, like (1, 1) or (2, 3) are colored green.

The strategies that correspond to shapes form a large and diverse class of strategies. In particular, the triangular strategy and rectangular strategy both correspond to shapes. Consider Figure 3.5(a), which depicts a shape corresponding to the triangular strategy of Figure 3.1(a). (Again, ignore dashed lines in the figure.) The shape consists of eight connected segments, beginning at (0, 4) and ending at (4, 0), each alternately going one unit to the right or down. The decision point in this case is the point (3, 2). Note that all points in the segments leading up to (3, 2) are either blue or white (unreachable), while all points in the segments after (3, 2) are red or white. In this case, all the points on the interior of the shape are continue points.

The rectangular strategy of Figure 3.1(b) corresponds to the shape formed by two segments, one from (0, 4) to (4, 4) and one from (4, 4) to (4, 0). The point (4, 4) is the decision point. As another example, consider Figure 3.5(b). If we consider the shape corresponding to the solid lines, we have a segment from (0, 2) to (0, 5), another from (0, 5) to (2, 5), and so on until (6, 6) — which is the decision point. Then we have five segments from (6, 6) to (0, 6). Once again, notice that some of the points before the decision point in the sequence of segments, such as (3, 4) and (2, 5) are unreachable, and some points after the decision point, like (6, 1) and (6, 2) are unreachable. The decision point is unreachable as well. In this case, all internal points are reachable (and thus colored green).

As an example of a strategy that does not correspond to a shape, if the strategy in Figure 3.5(b) had an additional terminating point, say Fail at point (1, 1), then it can never correspond to a shape.

Why shapes? One objection one may have to studying shapes is that the best strategy corresponding to shapes may be much worse than the best strategy overall.

However, the properties that shapes obey make intuitive sense. First, note that the strategies that correspond to shapes only have terminating points on the "boundary" of the strategy, and not on the interior. This makes sense because it is not worthwhile to have a termination point inside the boundary of termination points, since we might as well move the boundary earlier. Second, the strategies have a single decision point; this makes sense because it is not useful to alternate between red and blue points on the boundary, since the more YES answers we get relative to NO answers, the more likely the item should satisfy the filter, hence we should be able to improve the strategy by converting it to one with a single point where the colors change.

In addition, we found that over 100 iterations of the naive2 algorithm for random instances of the parameters s, e_1, e_0 and m, by inspection all of the optimal strategies corresponded to shapes. In addition, as we will see in the experiments, on varying parameters, the best strategy given by naive2 is no better than the best strategy corresponding to a shape.

Hence, we pose the following conjecture:

Conjecture 3.3.8 *Given a problem, the best deterministic strategy is one that corresponds to a shape.*

Proving this conjecture remains open.

Ladder Shapes: From all strategies that correspond to shapes, if we wanted to find the best strategy, we can prove that we only need consider the subset of shapes that we call *ladder shapes*. A ladder shape is formed out of two *ladders* connected at the decision point. We first define a *ladder* to be a connected sequence of (flat or vertical) segments connecting grid points, such that the flat lines go "right", i.e., from a smaller x value to a larger x value, and the vertical lines go "up", i.e., from a smaller y value to a larger y value. As an example, in Figure 3.5(b), the sequence of (dashed and solid) segments (0, 2)-(0, 3)-(3, 3)-(3, 6)-(6, 6) forms a ladder, while the sequence of segments (0, 2)-(0, 5)-(2, 5)-(2, 3)-(3, 3)-(3, 6)-(6, 6) is not a ladder because (2, 5) to (2, 3) goes from a larger y value to a smaller y value. As another example, the sequence of solid segments from (0, 4) to (4, 0) in Figure 3.5(a) is not a ladder because the vertical segments go from a larger y value to a smaller y value, while the segment (0-2)-(3,2) is a ladder.

We define the set of ladder shapes to be those shapes that contain a decision point and two ladders, i.e., the connected sequence of segments from the point on the y axis to the decision point forms one ladder and the connected sequence of segments from the point on the x axis to the decision point forms the second ladder. Thus, ladder shapes are a subset of the set of all shapes. Intuitively, ladder shapes are the shapes that we would expect to be optimal: the shape is smaller at the sides (close to the x and y axis, where we are more certain whether the item satisfies the filter or not), and larger close to the center (away from both the x and y axis, where we are more uncertain about the item).

As before, the strategy that corresponds to a ladder shape is the one formed by coloring all the points in the "upper" ladder blue, and all the points in the "lower" ladder red, then coloring all remaining reachable points inside the shape green, and coloring all unreachable points inside or on the boundary of the shape white. In the following, we provide a few examples of how we can convert any shape into a ladder shape such that the strategy corresponding to the ladder shape has the same or lower cost than the strategy corresponding to the shape. Since the set of ladder shapes is a subset of all possible shapes, if we want to find the best strategy corresponding to a shape, we can thus focus on ladder shapes instead of all shapes. Thus, we have the following theorems:

Theorem 3.3.9 (Transformation) *Any shape can be converted into a ladder shape yielding lesser cost and the same error.*

We now describe some examples of how the conversion algorithm works. The algorithm along with an informal proof can be found next. The algorithm essentially prunes redundant portions of the shape to give a ladder shape.

Theorem 3.3.10 (Best Shape) *For problem 3.2.1, the best strategy from the set of shapes has equal cost to the best strategy from the set of ladder shapes.*

The above theorem gives us an algorithm, denoted ladder, which considers a small subset of the set of all shapes, namely all the ladder shapes. Note that this algorithm is still worst-case exponential; however, as we will see in the experiments, this algorithm performs reasonably well in practice, and in particular, much better than naive2.

Examples of Converting Shapes to Ladder Shapes: First, consider the triangular strategy shown in Figure 3.5(a). As it stands, the shape (formed from the solid lines in the figure) is not a ladder shape, since the sequence of segments leading to the decision point (3, 2) from the point on the x axis as well as the point on the y axis don't form ladders. While the ladder from the y axis has segments that go "down" instead of "up", the ladder from the x axis has segments that go "left" instead of "right". In the strategy corresponding to the shape, note that asking questions at points above the line $y = 2$

is redundant, because once we cross $y = 2$ (i.e., 2 YES answers), we will always reach a Pass point. Similarly, notice that asking questions at points on the right of the line $x = 3$ is redundant. Thus, we can convert this triangular strategy into a rectangular strategy with the same error and lower cost simply by pruning the regions to the top and to the right of the decision point, and having termination points earlier. Notice that this corresponds to the ladder shape formed by the two dashed ladders (one from (0, 2) to (3, 2) and one from (3, 0) to (3, 2), with the decision point (3, 2)). Thus, the shape (giving the triangular strategy) can be converted into a ladder shape (giving a rectangular strategy) with lower cost and the same error.

As another example, consider Figure 3.5(b). Here the solid blue line represents a shape corresponding to the strategy formed by the blue, green and red dots. The shape has 10 lines: (0, 2)-(0, 5)-(2, 5)-(2, 3)-(3, 3)-(3, 6)-(6, 6) (which is also the decision point), (6, 6)-(6, 2) and so on. Now consider the shape corresponding to the dashed blue line in the figure. (This shape is the same as the solid shape, except for the portion (0, 3) to (3, 3) which bypasses the segment portions (0, 3)-(0, 5)-(2, 5)-(2, 3)-(3, 3), and the portion (5, 0) to (5, 1) which bypasses the portions (6, 0)-(6, 1)-(5, 1)). Notice that this shape corresponds to a ladder shape (with one ladder beginning at (0, 2) and ending at (6, 6), and another beginning at (5, 0) and ending at (6, 6), with a decision point at (6, 6)). This ladder shape corresponds to the strategy where there is a blue point at (1, 3) and a red point at (5, 0). Notice that for the strategy that corresponds to the shape, asking questions at (1, 3) and (1, 4) is redundant because the item will "Pass" no matter what answers we get at (1, 3) and (1, 4). Thus, moving the segment portion down to (1, 3) and to (5, 0) gives us a ladder shape that corresponds to a strategy that asks fewer questions to obtain the same result.

As yet another example, consider Figure 3.4, here, the shape corresponding to the solid lines in the figure (with decision point (5, 5)), can be replaced by the ladder shape corresponding to the two ladders (0, 4)-(3, 4) and (3, 0)-(3, 4), with decision point (3, 4).

Thus, we have shown some examples of how to convert shapes into ladder shapes such that the strategy corresponding to the ladder shape has lower cost than the strategy corresponding to the shape.

Conversion Algorithm: We now present our algorithm to convert shapes into ladder shapes. Readers may skip the rest of the material in this section without loss of continuity.

Our algorithm proceeds in two steps: First, we prune unreachable regions. Second, we prune redundant regions.

Step 1: Pruning Unreachable Regions: We define *y-path* to be the sequence of lines leading to the decision point from the point on the y axis in the shape (and including the decision point), and *x-path* to be the sequence of lines leading to the decision point from the point on the x axis in the shape (and including the decision point). We begin by pruning some of the the unreachable portions from the shape. Notice that if the *x-path* ever goes "down", we can instead add a segment that goes "right" until we once again intersect the shape. Additionally, if the *y-path* ever goes "left" we can instead add a segment that goes upwards until we once again intersect the shape. (The decision point may need to be moved if it is unreachable.) After this procedure, we may assume that the *x-path* now has segments that go right, upwards or to the left, while the *y-path* now has segments that go right, upwards and downwards. (Essentially all that we are saying is that for every shape with these unreachable portions, there is another shape without it, with same cost.) This also means that for a given y value there is a single point on the *x-path*, and for a given x value, there is a single point on the *y-path*.

Step 2: Pruning Redundant Regions: We now convert both *x-path* and *y-path* into ladders. We describe our algorithm for the *x-path*, the algorithm for the *y-path* follows similarly. (As we describe the algorithm, we also provide an informal explanation as to why it works.) The algorithm is essentially a scan along the x axis that incrementally builds the ladder.

We begin at $x = 0$, and scan the points corresponding to the *x-path* for the given x coordinate. For some $x = x_i$, we may find that there are some points along the *x-path* that have the x coordinate set to x_i. Find one such point which has the largest such y (say y_i). Now, we add the portion $(x_i, 0)-(x_i, y_i)$ to the ladder. We can do this because no matter what answers we get to the questions to the right of $(x_i, 0)$, with a y coordinate less than y_i, we will always end at a Fail terminating point. Now, we ignore the *x-path* portion below $y = y_i$. (We can effectively assume that we moved the x axis to $y = y_i$ (and the origin to (x_i, y_i)). We now repeat the same procedure. Let us say the next x coordinate for which we find an *x-path* point is $x = x_j$. We add $(x_i, y_i) - (x_j, y_i)$ to the ladder. Let the point on *x-path* with the largest y value at x_j be y_j. We then add $(x_j, y_i) - (x_j, y_j)$ to the ladder. Subsequently, we ignore all *x-path* points below y_j.

In other words, for each x value, we always add a segment that connects the ladder to the ladder until $x - 1$, and optionally a segment that goes from a smaller y value to a larger y value. We keep building the ladder until we hit the decision point. Note that the decision point is the point on the *x-path* with the largest y coordinate, thus we will always hit it. In this manner, we maintain the invariant that all points to the right of the ladder being constructed are all points on *x-path*, and not on *y-path*. (Note that there cannot be any points on the *y-path* to the right of this ladder apart from the decision point because otherwise we will violate the property that the shape has no unreachable regions.) Thus, we ensure that if we ever reach a point on the lower ladder, we are sure to end at a *x-path* point and not an *y-path* point.

Similarly, we build the upper ladder corresponding to the *y-path*. Here the invariant being maintained is that all points above this ladder must be *y-path* points. The decision point is then the point on the *y-path* that has the largest x coordinate. The two ladders meet at the decision point.

3.4 Probabilistic Strategies

In this section, we consider probabilistic strategies. Recall that a probabilistic strategy is again represented in a grid, however, each point has a triple $\mathcal{F}(x,y) = (a_{pass}, a_{fail}, a_{cont})$ corresponding to the probability of returning Pass (blue), Fail (red), or Continue asking questions (green); as a shorthand, we let $a_{pass}(x,y)$ denote the value of a_{pass} in $\mathcal{F}(x,y)$.

We can pose Problem 3.2.1 as a set of constraints, where the objective is to minimize the expected cost C, given a constraint that $E < \tau$, along with some additional constraints, which are essentially the counterparts of the equations described in Section 3.2.

- We have the probabilistic counterpart of Equation 3.1:

$$\begin{aligned} &\forall (x,y); x+y \leq m: \\ &\quad E(x,y) = a_{pass}(x,y) \times g(x,y) + a_{fail}(x,y) \times (1 - g(x,y)) \end{aligned} \tag{3.4}$$

  (Recall that $g(x,y)$ is defined in Theorem 3.3.3.) The error at a certain point is simply the probability that the strategy terminates at that point with a Pass, times the probability of error if it was a Pass, plus a similar quantity if the strategy terminated with a Fail. Note that $g(x,y)$ is a constant, independent of the strategy.

- The cost at a given point is simply the probability that the strategy terminates at that point times the total number of questions asked to get to the point.

$$\forall (x,y); x+y \leq m: C(x,y) = (a_{pass}(x,y) + a_{fail}(x,y)) \times (x+y) \tag{3.5}$$

- The error and cost equations stay the same as Equations 3.2–3.3.

$$E = \sum_{(x,y);x+y\leq m} E(x,y) \times [p_0(x,y) + p_1(x,y)] \tag{3.6}$$

$$C = \sum_{(x,y);x+y\leq m} C(x,y) \times [p_0(x,y) + p_1(x,y)] \tag{3.7}$$

- The counterpart of the equations in Figure 3.3 is simpler since we ask an additional question at $(x, y-1)$ with probability $c(x, y-1)$ and at $(x-1, y)$ with probability $c(x-1, y)$.

$$\begin{aligned}\forall(x,y);\quad x+y\le m:\quad & p_0(x,y) = \\ & e_0 \cdot p_0(x,y-1)\cdot a_{cont}(x,y-1)+ \\ & (1-e_0)\, p_0(x-1,y)\cdot a_{cont}(x-1,y)\end{aligned} \tag{3.8}$$

$$\begin{aligned}\forall(x,y);\quad x+y\le m:\quad & p_1(x,y) = \\ & e_1 \cdot p_1(x-1,y)\cdot a_{cont}(x-1,y)+ \\ & (1-e_1)\, p_1(x,y-1)\cdot a_{cont}(x,y-1)\end{aligned} \tag{3.9}$$

- In addition, we have the following constraints:

$$\forall(x,y); x+y=m:\quad a_{cont}(x,y)=0 \tag{3.10}$$

  The constraint above forces the strategy to terminate at m questions.

$$\forall(x,y); x+y\le m:\quad a_{pass}(x,y)+a_{fail}(x,y)+a_{cont}(x,y)=1 \tag{3.11}$$

  This constraint simply forces the probabilities of termination and continuation at each point on the grid to add up to one.

This program is not linear, due to constraints 3.6, 3.7, 3.8 and 3.9 (all of which involve a product of two variables). A key technical result of our work is that we can reason about "paths" using a variant of the Paths Principle (Theorem 3.3.3) to transform the program into a linear program.

Transformed Program: We introduce a new pair of variables $path_{pass}$, $path_{fail}$ and $path_{cont}$ to replace $p_0, p_1, a_{path}, a_{fail}, a_{cont}$ for every point in the grid. The variable $path_{pass}$ corresponds to the (fractional) number of paths in the strategy from $(0,0)$ to (x,y) that are stopped and Pass is returned for those paths, $path_{fail}$ corresponds to the (fractional) number of paths that are stopped at (x,y) and Fail is returned for those paths, while $path_{cont}(x,y)$ corresponds to the (fractional) number of paths that continue onwards beyond (x,y). Thus, $path_{pass}(x,y)+path_{fail}(x,y)+path_{cont}(x,y)$ represents the number of paths reaching (x,y). For instance $path_{cont}(x,y)=0$ implies that all paths reaching (x,y) terminate at (x,y).

We let the constant $const_1(x,y) = s \times e_1^x \times (1-e_1)^y$ (i.e., the probability that the item satisfies the filter, and we get a given sequence of x no answers and y yes answers), and constant $const_0(x,y) = (1-s)\times e_0^y \times (1-e_0)^x$ (i.e., the probability that the item does not satisfy the filter, and we get a given sequence of x no answers and y yes answers). Note that the constant $g(x, y)$ can be expressed as:

$$g(x, y) = \frac{const_0(x, y)}{(const_0(x, y) + const_1(x, y))}$$

The following relationships are immediate:

$$p_0(x, y) = const_0(x, y) \times (path_{pass}(x, y) + path_{fail}(x, y) + path_{cont}(x, y))$$

$$p_1(x, y) = const_1(x, y) \times (path_{pass}(x, y) + path_{fail}(x, y) + path_{cont}(x, y))$$

These relationships hold because the probability of getting to a point when the item satisfies the filter (or not) is simply the total number of paths times the probability of a single path when the item satisfies the filter (or not).

Additionally, since the probability a_{pass} is simply the fraction of paths that stop at (x, y) with a Pass, we have:

$$a_{pass}(x, y) = \frac{path_{pass}(x, y)}{path_{pass}(x, y) + path_{fail}(x, y) + path_{cont}(x, y)}$$

We have similar equations for Fail and Cont. The constraints may now be rewritten in terms of the path variables:

- Constraints 3.4, 3.5, 3.6 and 3.7 transform into:

$$E = \sum_{(x,y);x+y\leq m} path_{pass}(x, y) \times const_0(x, y) + path_{fail}(x, y) \times const_1(x, y)$$

$$C = \sum_{(x,y);x+y\leq m} (path_{pass}(x, y) + path_{fail}(x, y)) \cdot (x + y) \cdot (const_0(x, y) + const_1(x, y))$$

  (Recall that $const_0, const_1$ are constants, so the constraints are linear.) In other words, E is precisely the number of paths leading to the point that terminate at that point, times the smaller of the two error probabilities. The cost C is simply the cost for all paths terminating at (x, y) (each such path has probability $const_0 + const_1$).

- Constraints 3.8 and 3.9 transform into:

$$\forall (x, y); x + y \leq m : path_{pass}(x, y) + path_{fail}(x, y) + path_{cont}(x, y) =$$
$$path_{cont}(x, y - 1) + path_{cont}(x - 1, y)$$

In other words, the number of paths into (x, y) are precisely those that come from $(x, y-1)$ and $(x-1, y)$

- We replace constraint 3.10 with the following, which implies that no paths go beyond $x+y=m$.

$$\forall (x, y); x+y=m: \quad path_{cont}(x, y) = 0$$

- We also have the constraint that there is a single path to $(0, 0)$, i.e., $path_{pass}(0,0)+path_{fail}(0,0)+path_{cont}(0,0) = 1$

No additional constraints exist.

The linear program (LP) has a total of $O(m^2)$ variables, with the total size of the LP encoding being $O(poly(m))$. As a result, the complexity of the solution is $O(m^{2^{3.5}} \cdot \log(poly(m))$, i.e., $O(m^7 \log m)$

Theorem 3.4.1 (Best Probabilistic Strategy) *The best probabilistic strategy for Problem 3.2.1 can be found in $O(m^7 \log m)$.*

We denote the algorithm corresponding to the linear program above as linear.

3.5 Other Formulations

3.5.1 Problem 3.2.2

For Problem 3.2.2, we can show that we simply need to compute $g(x, y)$ for every point in $(0, 0)$ to (m, m), bottom up, and for every point where we find that $\min(g(x, y), 1-g(x, y)) < \tau$, we make the point a terminating point, returning Pass if $g(x, y) \le 0.5$ and Fail otherwise.

In fact, we can actually terminate earlier if we find that p_0 and p_1 are 0 for all points (x, y) : $x+y=m'$ at some $m' < m$. In this case, we do not need to proceed beyond points along $x+y=m'$.

Note that a feasible strategy that terminates and satisfies $E(x, y) < \tau$ for every terminating point may not exist.

Thus, we have the following theorem:

Theorem 3.5.1 *The best strategy for Problem 3.2.2 can be found in $O(m^2)$.*

3.5.2 Problems 3.2.3 and 3.2.4

We begin by considering Problem 3.2.4 first. This problem formulation is identical to Maximum A-Posteriori (MAP) estimation. In this case we simply ask all m questions (since there is no gain to asking fewer than m questions). We can estimate the p_0 and p_1 at all points along $x + y = m$ for this triangular strategy. If $p_0 > p_1$, we return Fail at that point and Pass otherwise. We can then estimate the error E over all the terminating points.

Thus, we have the following theorem:

Theorem 3.5.2 *The best strategy for Problem 3.2.4 can be found in* $O(m^2)$.

(We can actually compute the best strategy in $O(m)$ if binomial expressions involving m can be computed in $O(1)$ time.)

Subsequently, we can solve Problem 3.2.3 by performing repeated doubling of the maximum cost m, until we find a triangular shape for which $E < \tau$, followed by binary search between the two thresholds m and $m/2$. Thus, we have the following theorem:

Theorem 3.5.3 *The best strategy for Problem 3.2.4 can be found in* $O(m^2 \log m)$.

(We can actually compute the best strategy in $O(m \log m)$ if binomial expressions involving m can be computed in $O(1)$ time.)

3.5.3 Problem 3.2.5

For Problem 3.2.5, we can use the same linear programming formulation as in Section 3.2.4, except that we constrain C and minimize E. We thus have the following:

Theorem 3.5.4 *The best probabilistic strategy for Problem 3.2.5 can be found in* $O(m^7 \log m)$.

3.6 Multiple Filters

Now, we consider the case when we have independent filters $f_1, f_2, \ldots, f_l$, with independent selectivities $s_1, s_2, \ldots, s_l$, and independent error rates $e_{10}, e_{11}, e_{20}, e_{21}, \ldots, e_{l0}, e_{l1}$, where e_{i0} is the probability that a human answers YES when the item actually does not pass the ith filter, and e_{i1} is the probability that the human answers NO when the item actually does pass the ith filter. Note that the multiple filters problem is much harder than the single filter one, since a strategy not only must decide when to continue asking questions, but must also now decide *which* question to ask next (i.e., for what filter). The problem becomes even harder when the filters are correlated, however, we leave it for future work.

We can visualize the multiple filters case as a $2l$ dimensional grid, where each point $(x_{10}, x_{11}, x_{20}, x_{21}, \ldots, x_{l0}, x_{l1})$ corresponds to the number of NO and YES answers received from humans for each of the filters. (x_{i0} indicates the number of NO answers for the ith filter, and x_{i1} indicates the number of YES answers for the ith filter.)

In its most general form, the multiple filters problem allows us, at any point on this $2l$-dimensional grid, to ask a question corresponding to any of the l filters. Thus, each point can correspond to "Pass", "Fail", or "Ask ith Filter", where i can be one from $1 \dots l$.

We can in fact represent the problem as a linear program, generalizing the linear program in Section 3.4. The counterpart of the variables $path_{pass}$, $path_{fail}$ and $path_{cont}$ are $path_{pass}$, $path_{fail}$ and $path_{cont1}, \dots, path_{contl}$, representing respectively, the number of paths terminating at a given point with pass or fail and the number of paths continuing in the direction of each of the l filters. Similarly, the total number of paths coming into a point is simply the paths coming in from each of the l directions.

Theorem 3.6.1 *The best probabilistic strategy for the multiple filters version of Problem 3.2.5 can be found in* $O(m^{7l} \log ml)$.

This algorithm is exponential in the number of filters l, which may not be very large. However, any algorithm whose output is a strategy using our representation would need $\Omega(m^{2l})$, since we need to provide a decision for each point in the $2l$ dimensional cube of size m^{2l}. It remains to be seen if we can find an optimal or approximately optimal strategy whose representation is smaller.

3.7 Experiments

The goal of our experiments is to study the runtime and expected cost and errors of the strategies output by our algorithms with respect to those output by other naive and approximate algorithms. We continue to focus on Problem 3.2.1.

In our experiments we explored wide ranges of values for parameters m, e_0, e_1, τ, s. In some cases we manually selected the values, to study scenarios that interested us or to study extreme cases in the parameter space. In other cases, we synthetically generated random instances of the parameter values (over given ranges), to explore the average behavior.

Algorithms:

We considered the following exact deterministic algorithms:

- naive3: The naive algorithm that considers all 3^{m^2} strategies.
- naive2: The naive algorithm that considers all 2^{m^2} strategies (after pruning strategies that violate the Paths Principle).

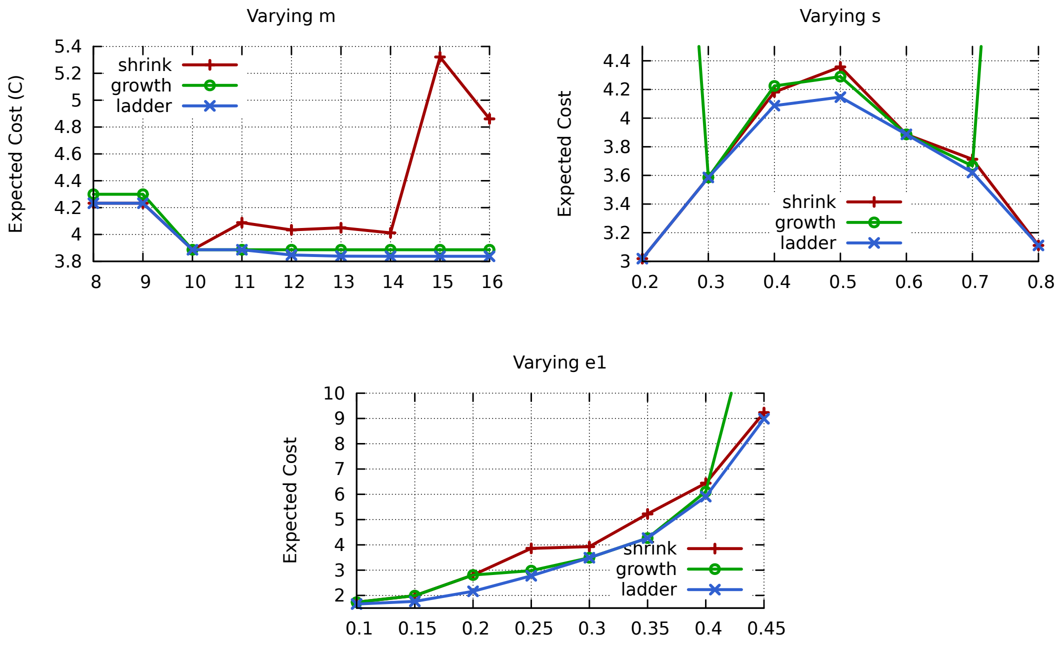


Figure 3.6: For fixed values of parameters: (a) Varying m (b) Varying s (c) Varying e_1

We also considered the following heuristic deterministic algorithms:

- ladder: This algorithm returns the best strategy corresponding to a ladder shape. This algorithm always returns a better deterministic strategy than the heuristic proposed in [139], as we will see in Section 3.8.
- growth: This greedy algorithm "grows" a strategy until the constraints are met. It begins with the null strategy at $(0, 0)$ (i.e., terminate and return Pass or Fail). Then, the algorithm "pushes the boundary ahead". In other words, in each iteration, the algorithm in turn considers moving each termination point (x, y) to $(x + 1, y)$ and $(x, y + 1)$ and computes the ratio of change in cost to change in error. The algorithm decides to move the termination point that yields the smallest increase in this ratio. This "pushing" continues until the error constraint is satisfied.
- shrink: This greedy algorithm "shrinks" a strategy until the cost cannot be decreased any longer. It begins with the triangular strategy that asks all m questions, and "pushes the boundary in". In other words, in each iteration, the algorithm for each terminating point (x, y) in turn, considers adding a terminating point at $(x, y - 1)$ or $(x - 1, y)$ and computes the ratio of the change in cost to change in error. The algorithm decides to add a terminating point that yields the largest increase in this ratio. The "shrinking" continues as long as the error constraint is satisfied.

- rect: This algorithm tries all rectangular strategies that fit in (m, m).

In addition, we have the optimal probabilistic algorithm:

- linear: This algorithm returns the strategy computed by the linear program in Section 3.4.

Also, we compared our algorithms against the best algorithm for Problem 3.2.2.

- point: This algorithm ensures that at every termination point, $E(x, y) < \tau$. Thus, this algorithm returns the optimal strategy for Problem 3.2.2. When the algorithm cannot find a feasible solution for Problem 3.2.2, we modify the solution to ensure that at least the cost constraint $C(x, y) < m$ is satisfied. That is, for an infeasible solution, we add termination points along the boundary $x + y = m$, if those points are reachable.

Note that there may be parameters for which some of the algorithms return an infeasible solution (i.e., where the error constraint is violated). It can be shown that for all algorithms except growth and point, either all algorithms return feasible solutions or none of them do. Algorithms growth and point, in addition to failing whenever other algorithms fail, also fail in some other cases.

Comparison of Heuristic Deterministic Algorithms:

Results on Varying m: ladder *results in large cost savings compared to other heuristic deterministic algorithms, and furthermore its cost decreases as m increases*

Figure 3.6(a) presents the results of an experiment that supports this finding. In this scenario, the parameters are $s = 0.6, e_0 = 0.2, e_1 = 0.25, \tau = 0.05$, and m (horizontal axis) is varied from 8 to 16. The vertical axis shows the expected cost C returned by the strategy found by the heuristic deterministic algorithms. For instance, when m is 14, the cost for ladder is about 3.85 (i.e., we need around 3.85 questions on average to get the desired expected error), growth is about 3.9 and shrink is about 4. The plot for rect is not shown, but rect is a straight line at about 5.6.

Note that even these small differences in expected cost can result in major cost savings overall. If there are a million items that need to be filtered, where each question costs 10 cents and m is 14, ladder results in at least $0.05 * 10^6 * 0.1 = \$5000$ of savings over the shrink and growth, and a whopping \$100000 of savings over the rect.

While at first one might think that increasing the question limit m will *increase* the overall cost, observe that in reality the opposite is true for ladder and rect. The reason is that as m increases, the number of shapes available for consideration strictly increases, giving the optimizer more choices. The same cannot be said of shrink and growth, since these are both heuristic greedy search algorithms that can get stuck in local minima. For instance, the cost for shrink increases as m goes from 10 to 11, and once again from 14 to 15 in the experiment above. Note also that none of the algorithms give a feasible solution when $m < 8$ for the set of constraints.

As mentioned earlier, space constraints prevent us from including multiple results per finding. The extensive additional experiments we performed support all of our findings.

Results on Varying s: growth *sometimes gets stuck in local minima; if not,* shrink *and* growth *outperform* rect.

Our second experiment, depicted in Figure 3.6(b), illustrates this finding. We fixed parameters $e_0 = 0.2$, $e_1 = 0.25$, $\tau = 0.05$, $m = 10$, and varied s from 0.2 to 0.8 and compared the same algorithms as before. The expected cost is plotted as a function of s. For instance, when s is 0.5, the cost for ladder is about 4.15, growth is about 4.28, rect (not depicted in figure) is about 5.68 and shrink is about 4.35. Once again, ladder performs the best. As expected, the cost increases when s is close to 0.5 since that situation is the most uncertain (and therefore we need to ask more questions). Additionally, when $s < 0.3$ or $s > 0.7$, growth gives an infeasible answer (i.e., it gives a strategy that does not satisfy the constraint on error) — depicted in the graph as the cost being set to ∞. Since the growth strategy does a local search around the origin, it can get stuck in an infeasible local minimum where the error can no longer be reduced by growing the strategy. The algorithm rect is much worse than the other algorithms with an additional expected cost of at least 1 over the other algorithms. However, it does give a feasible solution when growth does not.

Results on Varying e_1: Cost increases superlinearly as e_1 increases for all algorithms.

In the third experiment, depicted in Figure 3.6(c), we fixed parameters $e_0 = 0.25, s = 0.7, \tau = 0.1, m = 15$, and varied e_1 from 0.1 to 0.45 and compared the algorithms. The expected cost is plotted as a function of e_1. Once again, ladder performs the best, while growth and shrink each perform well in some situations. Also, the expected cost increases as e_1 increases, in a superlinear fashion. growth once again returns an infeasible answer for large e_1, and there are some points where shrink does extremely poorly compared to ladder, such as $e_0 = 0.25$ (with a difference in expected cost of 1, translating to > 33% in savings). The experimental results on varying e_0 are similar and therefore not shown.

Comparison of ladder**,** point**, and** linear **algorithms:**

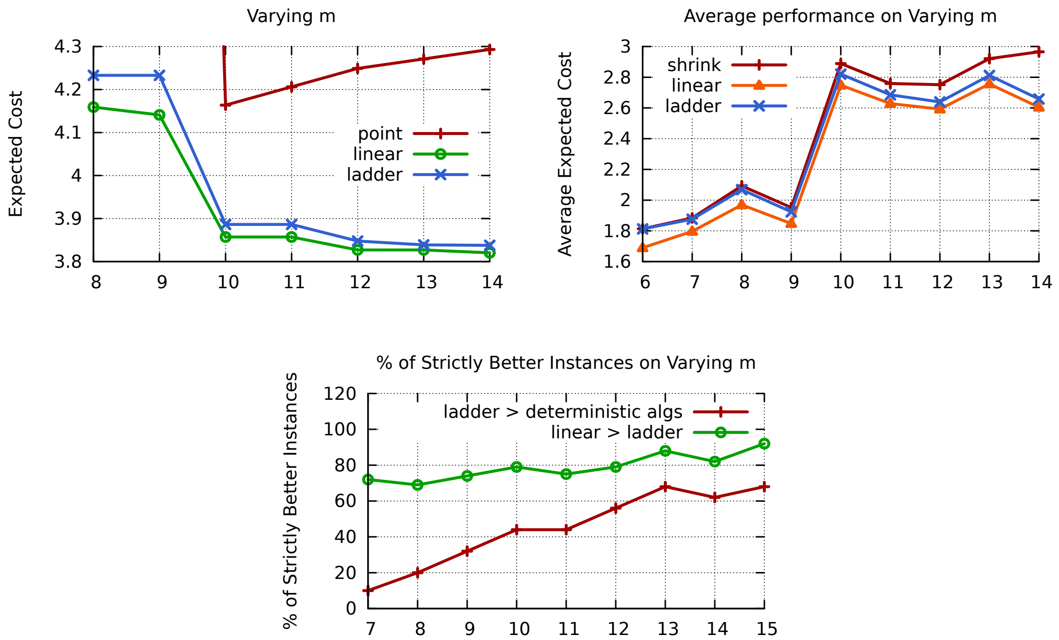


Figure 3.7: (a) Varying m for fixed values of parameters (b) Varying m on Average (c) Fraction of strictly better shapes

Having shown that ladder is the best heuristic deterministic algorithm (at least in terms of the quality of the results; run time performance is explored later on), the next natural question is how it compares to our other choices.

Results on Varying m: linear *performs even better than* ladder*, both of which perform significantly better than* point*, which sometimes returns infeasible solutions.*

To illustrate our finding, we repeated the same set-up of the experiment in Figure 3.6(a) — the results are depicted in Figure 3.7(a). We find that point returns an infeasible solution even for the case when $m = 8$ or 9, but for $m = 10$ onwards it returns a feasible solution. This is probably because point terminates too early (and has too "narrow" a strategy) when m is small, while linear and ladder have a "wider" shape. Interestingly, the cost of point also increases as m increases; this is because the strategy looks identical for any m and $m + 1$, except that the termination points along $x + y = m$ may have been moved up to $x + y = m + 1$.

While point is infeasible for $m = 9$, linear has a cost of nearly 0.1 less than ladder, representing significant savings in overall cost. Also, the cost returned by linear only decreases as m increases since a larger space of solutions are considered.

The cost difference between point and linear is greater than what we have observed in our earlier graphs. This difference highlights the benefits of using an expected error bound (Problem 3.2.1) as opposed to a point error bound (Problem 3.2.2). And as mentioned earlier, if we are filtering one million items, the savings are multiplicative. Thus, for crowdsourcing applications where expected error is adequate, linear (or ladder if a deterministic strategy is desired) is clearly the way to go.

Comparison of Algorithms on Average:

So far we have illustrated our findings with particular scenarios, i.e., particular parameter settings. To further validate our findings, we now explore the average behavior of our algorithms on varying parameters randomly over a range of values.

Results on Varying m: (1) The linear *and* ladder *algorithms continue to outperform the other algorithms in our average scenario, especially for large m. (2) For smaller m,* shrink *does quite well.*

In the first experiment, depicted in Figure 3.7(b), we compared the algorithms on varying m for 100 synthetic instances where s, e_0, e_1, τ were sampled from $[0, 1]$, $[0.1, 0.4]$, $[0.1, 0.4]$, $[0.005, 0.1]$ respectively. (We made sure that all three algorithms were feasible for each random instance used.) The expected cost is then averaged over all 100 instances, and plotted as a function of m. We focus on the best algorithms so far, ladder and linear. We also show shrink for comparison. (rect is always much worse, and growth and point have cases where they returns infeasible answers, and are typically not much better than shrink when they are feasible).

We can see in Figure 3.7(b) that linear performs much better than all algorithms for all m, with at least a difference of 0.1 on average. For lower m values, for these settings we do not see much of an improvement on average between ladder and shrink, which indicates that we may be able to use shrink for cases when m is small. However, for larger m, we find that the difference between ladder and shrink is at least 0.1. This is because there are more opportunities for optimization for ladder and linear compared to shrink as m increases.

Interestingly, we find that the cost does not strictly decrease for linear and ladder as m increases; this is an artifact of our experimental setup. For each m, we restrict our instances to those for which all algorithms are feasible. Thus, for smaller m we are not considering many of the "harder" cases, which are handled by larger m, instead, we are only consider the "easier" cases, whose expected cost is smaller. On the other hand, allowing a larger m means that we can explore a larger space of solutions.

Thus, the variation with m is not as predictable as in the earlier case, except that it seems to gradually increase with some small variations.

Results on Varying m: linear *yields strictly better (lower cost) strategies than* ladder *in a substantial majority of the scenarios. Furthermore,* ladder *outperforms the rest of the deterministic algorithms in a substantial number of scenarios.*

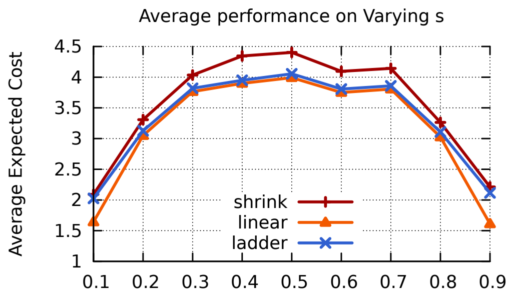
The previous experiment indicates that, on average, we get significant improvements in cost by using linear and ladder. However, we would like to verify if this behavior is because of a few instances where there is a high difference in cost (while the rest of the instances return the same cost). To see how often we get strictly better strategies by using either ladder or linear, we counted the number of instances where linear gave a strictly better strategy than ladder, and the number of cases where ladder gave a strictly better strategy than the rest of the algorithms for the same experimental setup described above. The results are depicted in Figure 3.7(c).

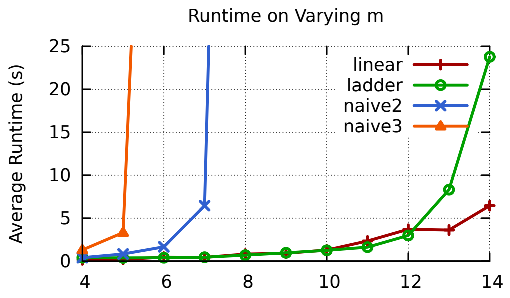
We find that for $m = 10$, linear gives a strictly better strategy than ladder in 80% of the cases (and in the remaining 20% of the cases, gives a strategy of the same cost as ladder). On the other hand, ladder gives a strictly better strategy than the rest of the algorithms for over 45% of the cases (and in the remaining 55%, gives a strategy of the same cost as the best heuristic algorithm).

Further, the number of scenarios where linear outperforms ladder (and where ladder outperforms the others) continues to increase as m increases. For $m = 15$, linear gives a strictly better strategy than ladder in 90% of the cases, while ladder gives a strictly better strategy than the rest of the algorithms in 65% of the cases.

Results on Varying s: The linear *algorithm outperforms* ladder *much more when s is away from* 0.5, *while* ladder *outperforms the other algorithms more when s is close to* 0.5.

In the second experiment, depicted in Figure 3.8(a), we compared the algorithms on varying s from 0.1 to 0.9 for 100 synthetic instances where $m = 14$ and e_0, e_1, τ were sampled from $[0.1, 0.4]$, $[0.1, 0.4]$, $[0.005, 0.1]$ respectively. The average expected cost was plotted as a function of s. In this case, we find that linear on average yields a smaller cost than the other two algorithms by around 0.1, but more so when s is away from 0.5. On the other hand, shrink on average yields a larger cost than ladder, and more so when s is closer to 0.5. In fact, when $s = 0.5$ shrink asks 0.5 higher questions on average.





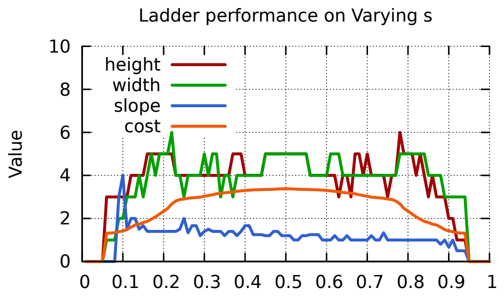


Figure 3.8: (a) Varying *s* on average (b) Runtime variation with m (c) Ladder Variation with s

Comparison to Naive Algorithms:

Cost for Random Instances: ladder *performs identically to* naive2 *and* naive3 *for each random instance generated.*

In Section 3.3.2, we informally argued that any optimal deterministic strategy should be found by the ladder algorithm, although we do not have a formal proof yet. In our next experiment we checked for cost differences between strategies found by ladder and those found by naive2, an exhaustive algorithm that does find the optimal deterministic strategy. (naive3 gives the same result as naive2.) We generated 100 synthetic instances for each m from 4 to 8 (with s, τ, e_1, e_0 all uniformly sampled from $[0, 1]$, $[0.1, 0.4]$, $[0.1, 0.4]$, $[0.005, 0.1]$ respectively). (Beyond 8, naive2 was impractical to use.) We found that ladder returns a strategy of the same cost as naive2 for each instance. In fact, in none of the many other experiments we performed did we find an instance where ladder does not return the optimal deterministic strategy. This result is a strong indicator that ladder is indeed optimal, however, the proof is still open.

Runtime Comparisons

Runtime results on Varying m: naive3 *and* naive2 *become impractical to use even for very small m, while* ladder *and* linear *are efficient alternatives for m as large as 15. For very large* $m \approx 40$, shrink, growth, rect *are able to return a strategy fairly quickly.*

In the first experiment, we compared the average runtimes (in seconds) of the naive algorithms with the best deterministic and best probabilistic algorithms across 100 synthetic instances for each m from 4 to 14 (with s, τ, e_1, e_0 all uniformly sampled from $[0, 1]$, $[0.1, 0.4]$, $[0.1, 0.4]$, $[0.005, 0.1]$ respectively). Comparing average runtimes lets us see how much time we might take for an algorithm for any set of parameters on average. Figure 3.8(b) shows the average runtimes as a function of m. While naive2 and naive3 take more than 5 minutes even for m as small as 5 and 7 respectively, ladder takes less than a minute even until $m = 14$, while linear takes even less time. Thus, the optimizations of Section 3.3 let us design strategies for larger m. (Our implementation of linear uses exact arithmetic, and as a result takes longer. If we were willing to get a slightly worse solution, we may be able to reduce the running time even further.) Note that if we are willing to use a slightly worse deterministic algorithm, we can get strategies for substantially larger values of m. For instance, growth is able to return a solution for $m = 40$ within seconds, while shrink is able to return a solution for $m = 40$ within a minute.

Variation of Ladder Shape: In our final set of experiments we studied two interesting properties of the ladder shapes selected by ladder. We define the *height* and *width* of a ladder shape to be, respectively, the largest distance between the two ladders along the y axis and along the x axis respectively. We also define the *slope* of a ladder shape to be the ratio of the y coordinate to the x coordinate of the decision point. Since there is no counterpart to ladders and decision points in the strategies output by linear, we cannot study them in this way.

Results on Varying s: The optimal ladder shape for ladder *has a decision point closer to the y axis (larger slope) when s is small and closer to the x axis (smaller slope) when s is large. In addition, the closer s is to 0.5, the further away the decision point of the ladder shape is from the origin.*

We fix $e_1 = 0.2$, $e_0 = 0.2$, $\tau = 0.05$, $m = 12$, and vary s in increments of 0.005 from 0 until 1. The results are depicted in Figure 3.8(c). We find that the slope of the optimal ladder shape moves from 4 all the way down to 0 gradually as we increase s. (Note that we set slope = 0 when the strategy is simply a terminating point at the origin.) This result can be explained as follows: When s is very small, it is unlikely that we will get any YES answers, so more questions need to be asked to verify that an item actually passes the filter, but not when the item does not pass the filter. On the other hand, when s is large, it is unlikely that we will get any NO answers, so more questions are needed to ascertain whether an item fails the filter. Note that cost increases until 0.5 and then decreases, as expected. Height and width are mirror images of each other across $x = 0.5$: this is expected since $e_0 = e_1$, thus the best shape for s is the best shape for $1 - s$.

3.8 Related Work

The work related to this chapter falls under four categories: crowdsourced schema matching, active sampling, filtering applications, and statistical hypothesis testing.

Crowdsourced schema matching: Recent work by McCann et al. [139] has considered the problem of using crowdsourcing for schema matching. The core of the problem is similar: the crowd provides (possibly incorrect) answers, and the goal is to determine whether a match is true or not. The strategy used is the following: ask at least v_1 questions, and stop either when the difference between the number of YES and NO answers reaches a certain threshold δ, or when the total number of questions asked is v_2. The authors prove probabilistic bounds on the maximum error from this strategy, under specific assumptions for the performance of human workers. The proposed strategy can be mapped to a shape in our framework. In fact, we can convert the shape into a ladder shape (as described in Section 3.3.2), and obtain a new strategy that asks fewer questions while providing the same error guarantees. Moreover, since [139] does not optimize for the number of questions, our cost-optimized algorithms can provide even further improvements.

Active Sampling: (We discuss active learning in detail in Chapter 3.8. Here, we provide a quick summary, and also relationships to the results in this chapter.) The field of active learning [169] addresses the problem of actively selecting training data to ask an "oracle" (for instance, the oracle could be an expert user) that would help train a classifier with the least error. Our metric for error (i.e., expected error) is the same as the 0-1 loss used in machine learning.

Typical papers studying active learning do not assume that the oracle makes mistakes, and in any case, repeating the same question to the oracle would typically not help. However, there are some papers that do consider the case when many interchangeable humans can be used as oracles, and we discuss them next.

Sheng et al. [171] consider the problem of obtaining labels for training data in the context of machine learning. Specifically, the problem is whether to ask for another label for an existing data item or whether to acquire more data items, in order to maximize the utility of the training data set for the machine learning algorithm. Similar work has considered a pool of workers where the most "informed" worker is asked for a label on the fly on the most uncertain item [81,188], and the accuracy of the worker is learned as labels are obtained. Other work [163, 189] considers a setting where the worker's labels are provided beforehand, and the goal is to infer the labels of items and the accuracy of different workers.

None of the previous studies deal with the problem of optimizing the number of questions asked to the crowd. In large-scale human computation, especially in marketplaces such as Mechanical Turk, this metric is the most critical cost factor that needs to be optimized. One previous paper has considered optimizing for low cost for filtering in conjunction with learning error rates for workers [112], however, they only provide worst-case guarantees—in fact, they are able to show that there are no improvements in the worst-case from dynamically changing the number of answers requested from workers, which is clearly not true in our scenario (which focuses on the expected case). In our work so far we have not taken into account knowledge of or discrepancies in worker accuracies, assuming a rapidly changing and replaceable worker pool (as in Mechanical Turk). As future work we will explore incorporating worker accuracy into our theory and algorithms.

Statistical Hypothesis Testing: Our problem is also related to the field of Statistical Hypothesis Testing [187]. This field is concerned with the problem of trying to estimate whether a certain hypothesis is true or not given the observed data. One such method of estimation is to compute the LR (Likelihood Ratio), i.e., the ratio of the probability that a given hypothesis is true given the data to the probability that an alternative hypothesis is true given the same data. Subsequently the LR is checked to see if it is statistically significant. In our case, our two hypotheses (for a given data item) are simply whether or not the item satisfies the filter. Unlike typical applications of hypothesis testing, where the goal is to estimate the parameters of some distribution, here the distribution is provided to us (i.e., that the item satisfies the filter with probability s). For Problem 3.2.2, in fact, our algorithm computes the LR to see if it is greater than the threshold (τ) for all reachable points. However, hypothesis testing techniques do not help us address the problem of minimizing overall cost while filtering a large set of data items.

Filtering Applications: Several practical applications have used heuristic strategies for filtering, typically a majority vote over a fixed number of workers, in the context of sentiment analysis and NLP tasks [173], categorization of URLs [34], search result evaluation [31], and evaluation of competing translations [195]. In fact, for all these strategies (which correspond to the triangular strategy described earlier), we can replace them with an equivalent rectangular strategy with much lower cost and the same answers. If the ladder or linear algorithms are used to design strategies, the cost can be reduced even further.

3.9 Conclusions

In this chapter, we designed algorithms for crowd-powered filtering. Filtering has wide applicability in image, video, and text analysis. Furthermore, filtering corresponds to a fundamental relational operator in crowd-powered database systems.

We designed algorithms that efficiently find filtering strategies that result in significant cost savings compared to commonly-used strategies in crowdsourcing applications, while ensuring the same accuracy. In fact, the cost savings can be up to 20-30% in practice.

In the next chapter, we show how we can generalize the algorithms in this chapter, by removing some of the simplifying assumptions (i.e., all workers have the same error rate or need to be paid the same amount, all items are equally difficult, or that we have no auxiliary information from automated schemes and no latency constraints).

Chapter 4

Algorithm 1 (Variants): Filtering Generalizations

4.1 Introduction

In the previous chapter, we described algorithms to find filtering strategies under a simple setting: filtering a set of items (with no prior knowledge on items), using a single infinite pool of identical workers, with equal costs (i.e., they all need to be paid the same amount). We designed algorithms that generate strategies with guarantees on expected cost and expected error.

In this chapter, we consider significant generalizations of our algorithms in the previous chapter. We focus on algorithms that generate probabilistic strategies (since probabilistic strategies subsume deterministic strategies), and consider generalizations of two types:

(a) *Improvement-based:* Improving the performance of the algorithms (in terms of expected cost and error) by removing some of the simplifying assumptions used in Chapter 3. For instance, we may wish to take into account individual worker abilities, or take into account individual item difficulties.

(b) *Functionality Addition-based:* Enhancing the algorithms to support additional functionality needed by many applications in practice. For instance, we may wish to support a constraint on latency, which the algorithms in Chapter 3 do not support.

The entire list of generalizations we consider can be found in Table 4.1. (We do not expect the reader to fully understand the table at this point.) We initially focus on one important improvement-based generalization (type (a) in the list above), that of *worker abilities*. The previous chapter assumes that all human workers have the same error rates — all workers are equally capable at answering questions. In practice, there are some workers that are much better than others, possibly because they do a much more careful job.

In this chapter, we demonstrate that we can generalize the algorithms from Chapter 3 to take into account worker abilities; we get a strategy that is optimal, but is computationally intractable to design as well as store. Then, to combat the computational intractability, we also provide algorithms that can generate a strategy that is efficient to design and store, but is approximate.

Later on, in Chapter 9, we will demonstrate that the algorithms that take into account worker abilities yield a significant reduction in cost (for fixed error) as compared to algorithms considered in the previous chapter, which make a range of simplifying assumptions. In this chapter, however, our focus will be on algorithm design, rather than experimental evaluation.

Our second focus, after worker abilities, will be on incorporating prior information into our algorithms, which is another improvement-based generalization. The algorithms in Chapter 3 assume that we have no information about any of the items to begin with; however, there may be cases where we have "prior information"—that is, knowledge that some items are more likely to pass the filter than others. This prior information may originate from an automated algorithm, such as a machine learning algorithm, that outputs probabilities for whether each item is likely to pass the filter or not. We will discuss other generalizations in Section 4.4.

4.1.1 Outline of Chapter

- We describe the *answer-record* representation, which is a straightforward extension of the representation in the previous chapter. This representation provides the optimal solution to all the generalizations that we consider, but can be expensive to compute in some cases (Section 4.2).
  - We describe the complete generalization for the case of many worker abilities (Section 4.2.1).
  - We provide the key ideas for generalizing the answer-record representation when incorporating prior information (Section 4.2.2).
- We describe the *posterior-based* representation that provides an efficient but approximate solution to all generalizations (Section 4.3).
  - We describe the representation for the basic setting considered in Chapter 3, and show that the expected cost of the optimal strategy in this representation converges in the limit

to the cost of the optimal strategy in the answer-record representation (Section 4.3.1).

  - We describe the complete generalization using this representation for the case of many worker abilities (Section 4.3.2).
  - We provide the key ideas for generalizing the posterior-based representation when incorporating prior information (Section 4.3.3).

- We discuss other generalizations (Section 4.4).

We do not consider related work in this chapter since prior work related to filtering was already covered in Chapter 3.

4.2 The Answer-Record Representation

In this section, we describe our first representation. To enable this chapter to be relatively self-contained (independent of the previous chapter), we describe the problem setup in entirety, but under the generalization of worker abilities. To illustrate the power of the representation, we will then discuss the generalization of incorporating prior information. We will describe other generalizations in Section 4.4.

4.2.1 Setting with Worker Abilities

Setting: We are given a set of items $\mathcal{I}$, where $|\mathcal{I}| = n$. A random variable V controls whether an input item satisfies the filter ($V = 1$) or not ($V = 0$). The selectivity of our filter, s, gives us the probability that $V = 1$ (over all possible items).

As before, we assume that there is no automated mechanism to examine an item and determine for certain whether that item satisfies the filter or not. The only type of action we can perform on an item is to ask a specific human worker w_i, $i \in 1 \ldots r$ a *question*. The human can tell us YES (meaning that she thinks the item satisfies the filter) or NO. The human worker w_i can make mistakes, and in particular:

- The false positive rate is: $Pr[w_i\text{'s answer is YES}|V = 0] = e_0(w_i)$
- The false negative rate is: $Pr[w_i\text{'s answer is NO}|V = 1] = e_1(w_i)$

The error rates $e_0(w_i), e_1(w_i)$ are estimated either by evaluating worker w_i on questions with known correct answers, or by using prior history on worker performance. Overall, there may be some workers who are less error-prone at answering questions than others, possibly because they are more diligent or more capable.

We can ask different humans the same question to get better accuracy, and we assume that their errors are independent. However, if we ask the same human the question on the same item, we will get the same answer. Therefore, we will ask the question on a given item to a given human at most once.

Note that our techniques are easily adapted to the somewhat simpler setting when there are different worker classes, each with an infinite number of workers. In this setting, there are r classes of workers with error rates $e_0(w_i), e_1(w_i), i \in 1 \ldots r$, such that there are infinitely many workers in each class (and each worker in a given class has the same error rate). In this scenario, even after receiving an answer from a worker in one class, we may still get additional (possibly different) answers from workers in the same class (with the same error rate). We return to this simpler setting in the posterior-based representation section (Section 4.3).

Strategies: A *strategy* $\mathcal{F}$ is a computer procedure that takes as input one item, asks one or more humans questions on that item, and eventually outputs either Pass or Fail. A Pass output represents a belief that the item satisfies the filter, while Fail represents the opposite. We define an *algorithm* to be a procedure that, given parameters and constraints, generates a strategy.

Since humans may have different error rates, the state of processing for an item after some questions are asked can be completely represented using a state variable $S = (x_1, y_1, x_2, y_2, \ldots, x_r, y_r)$, where x_i is an indicator variable indicating whether worker w_i answered YES for the question on the item, and y_i indicates whether worker w_i answered NO for the question on the item. If $x_i = y_i = 0$, then w_i has not been asked a question yet on the item; if $x_i = 1, y_i = 0$, then w_i has been asked a question and has answered YES; and if $x_i = 0, y_i = 1$, then w_i has been asked a question on the item and has answered NO. The case where $x_i = y_i = 1$ can never arise.

Note that the reason why the state variable S is a complete description of the state of processing is that the order in which the answers are provided by humans is not important; only the set of YES/NO answers is important, and the identity of the workers who provided the answers. Therefore, the state variable S captures all the relevant information about an item necessary for a strategy to maintain. Recall that in Chapter 3, S was simply the count of the YES and NO answers. This information was sufficient for a strategy to maintain because all workers were assumed to be equally error-prone in that setting, and therefore, the identity of the worker giving a specific answer was not important.

We therefore call this representation the *Answer Record Representation* (i.e., the state or processing is the complete set of answers).

At a given state S, a strategy $\mathcal{F}$ probabilistically does one of the following: (a) stop executing, and return Pass on the item, (b) stop executing, and return Fail on the item, or (c) continue executing (Cont), i.e., ask an additional question for that item. Thus, at a given state S, the strategy returns Pass, i.e., $\mathcal{F}(S)$ = Pass, with probability $a_{pass}(S)$, returns Fail, i.e., $\mathcal{F}(S)$ = Fail with probability $a_{fail}(S)$, and will ask another human a question on that item, $\mathcal{F}(S)$ = Cont with probability $a_{cont}(S) = 1 - a_{pass}(S) - a_{fail}(S)$. If either Pass or Fail returned at a state, we say that the strategy terminates. If Cont is returned at a state, then an answer is requested from one of the unasked human workers–those for whom $x_i = y_i = 0$, with equal probability. Thus, in this scenario, we do not control which human worker answers our question—this scenario is relevant in marketplaces like Mechanical Turk. We consider the case where we may control which worker is asked to answer the question in Section 4.4.4. (Recall that in Chapter 3, we referred to strategies with probabilistic decisions as probabilistic strategies; here, for convenience, we simply refer to them as strategies.)

Metrics: To determine which strategy is best, we study the two metrics of error and cost. We start by defining two quantities, given a strategy:

- $p_1(S)$ is the probability that the strategy reaches S and the item satisfies the filter ($V = 1$); and
- $p_0(S)$ is the probability that the strategy reaches S and the item does not satisfy the filter ($V = 0$).

We can now define the following metrics:

- E is the sum of expected errors across all states. The expected error at a state S is simply the probability that the strategy terminated at S with an error being made.

$$E = \sum_{S} a_{pass}(S) \cdot p_0(S) + a_{fail}(S) \cdot p_1(S) \tag{4.1}$$

- C is the sum of expected cost across all states. The cost at a state S is the probability that the state was reached, i.e., $(p_0(S) + p_1(S))$, multiplied by the probability that a termination decision was taken, i.e., $(a_{pass}(S) + a_{fail}(S))$, multiplied by the cost, i.e., the total number of answers so far: $\sum_{i \in 1 \ldots r}[x_i + y_i]$.

$$C = \sum_{S} [p_0(S) + p_1(S)] \cdot (a_{pass}(S) + a_{fail}(S)) \cdot (\sum_{i \in 1 \ldots r} [x_i + y_i]) \tag{4.2}$$

The probabilities p_0 can be iteratively computed using the following equations:

$$\begin{aligned} p_0(x_1, y_1, \ldots, x_r, y_r) &= \sum_{1 \le i \le r;\ x_i=1:\ R=(x_1,y_1,\ldots,x_{i-1},y_{i-1},0,0,\ldots,x_r,y_r)} \frac{1}{b} \cdot e_0(w_i) \cdot p_0(R) \cdot a_{cont}(R) \ + \\ & \sum_{1 \le i \le r;\ y_i=1;\ R=(x_1,y_1,\ldots,x_{i-1},y_{i-1},0,0,\ldots,x_r,y_r)} \frac{1}{b} \cdot (1 - e_0(w_i)) \cdot p_0(R) \cdot a_{cont}(R) \\ p_0(0, 0, \ldots, 0, 0) &= (1 - s) \end{aligned}$$

where b is the number of x_i or y_i that are 0, i.e., the number of workers who have not been asked yet. Thus, we simply sum up the probabilities that the strategy reaches $(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, 0, 0, \ldots, x_n, y_n)$ and gets a YES from a specific worker w_i (out of b) who has not been asked before, for all i. And we add this sum to the probabilities that the strategy reaches $(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, 0, 0, \ldots, x_n, y_n)$ and gets a NO from a specific worker w_i (out of b) who has not been asked before, for all i. Therefore, the probability of getting to a state S (and the item not satisfying the filter) is the sum of the probabilities of getting to one of the previous states S' that is identical to S but has one x_i or y_i diminished by 1 (with the item not satisfying the filter), and getting the appropriate answer (YES/NO) from the appropriate worker i. (Naturally, the states S' that are invalid are omitted from the summation.)

For a strategy, we define the states S for which p_0 or p_1 are non-zero as *reachable states* (that is, there is a non-zero probability of reaching them while executing a strategy.)

Problem: Given input parameters selectivity s, false negative rates $e_1(w_i)$, and false positive rates $e_0(w_i)$, we consider the following problem:

Problem 4.2.1 (Abilities) *Given an error threshold τ and a budget threshold per item m, find a strategy that minimizes C under the constraint $E < \tau$ and*

$$\forall \text{ reachable } (x_1, y_1, \ldots, x_r, y_r) : \sum_{i \in 1 \ldots r} (x_i + y_i) \le m$$

Since we want to ensure that the strategy terminates, we enforce a threshold m on the maximum number of questions we can ask for any item (which is nothing but a maximum budget for any item that we want to filter).

Solution: We present a solution to Problem 4.2.1 that generalizes the solution presented in Chapter 3. As in Chapter 3, our solution leverages linear programming.

Even though the equations describing relationships between variables are highly non-linear (ref. Equation 3.2, 3.3), we can convert these into linear equations by considering the flow of *paths*. A path is a specific sequence of answers that can be used to get to a given state S. For instance, if the number of workers $r = 2$, and S_0 is $(1, 0, 1, 0)$ (i.e., worker w_1 answered YES and w_2 answered YES), then one path to get to S may be w_1 answered YES first, followed by w_2. The only other possible path is for w_2 to answer YES followed by w_1 answering YES. Of course, not all paths may be possible in a strategy. That is, it may be possible that a strategy stops and returns Pass when w_2 answers YES, and therefore, the latter path is not feasible.

We define a new variable called $path(S)$ to denote the number of paths entering S. Alternatively, this variable can be regarded as the number of ways we can get from the origin $(0, \ldots, 0)$ to S within a strategy. In our example above, say that the two possible paths are feasible, and assume the strategy always continues from $(1, 0, 0, 0)$ and from $(0, 0, 1, 0)$. Then $path(S)$ is 2. However, assume that from $(1, 0, 0, 0)$ the strategy continues only with 0.5 probability. Then in this case $path(S)$ is 1.5.

Notice that $path(S)$ can be computed recursively. Continuing with our example, observe that the number of ways of getting to S_0, i.e., $path(S_0)$ is equal to the sum of the number of ways of reaching S_0 via $S'_0 = (0, 0, 1, 0)$, and the number of ways of reaching S_0 via $S''_0 = (1, 0, 0, 0)$. If, for instance, $a_{cont}(S'_0) = 0$, that is, the strategy always terminates at S'_0, then, the former number is 0. If, on the other hand, $a_{cont}(S'_0) = 0.5$, that is, with probability half, the strategy asks an additional question at S'_0, then the former number is $0.5 \times path(S'_0)$, i.e., the number of paths leaving S'_0 is half the number of paths reaching S'_0 (alternatively, half the number of ways to reach S'_0).

From each state S, therefore, the incoming paths, $path(S)$, flow onward to other states in the following manner: $path_{pass}(S)$ is the fraction of paths that stop at S (with the strategy returning Pass); $path_{fail}(S)$ is the fraction of paths that stop at S (with the strategy returning Fail); and $path_{cont}(S)$ is the fraction of paths that continue onward to other states.

We therefore have:

$$\begin{aligned} path_{pass}(S) &= a_{pass}(S) \times path(S) \\ path_{fail}(S) &= a_{fail}(S) \times path(S) \\ path(S) &= path_{cont}(S) + path_{pass}(S) + path_{fail}(S) \end{aligned}$$

That is, a_{pass} and a_{fail} alternatively represent the fraction of the path at a state that is lost by returning a Pass or Fail decision respectively. The remaining (fractional) number of paths, $path_{cont}$, continue onward to other states. Via conservation of paths, we have:

$$\begin{aligned} &path(x_1, y_1, x_2, y_2, \ldots, x_r, y_r) = \\ &\sum_{0 \le i \le r;\ x_i = 1} path_{cont}(x_1, y_1, x_2, y_2, \ldots, x_{i-1}, y_{i-1}, 0, 0, \ldots, x_r, y_r) + \\ &\sum_{0 \le i \le r;\ y_i = 1} path_{cont}(x_1, y_1, x_2, y_2, \ldots, x_{i-1}, y_{i-1}, 0, 0, \ldots, x_r, y_r) \end{aligned}$$

In other words, path flow can come to a state by asking any one of the r workers and getting any one of the two possible answers (YES/NO).

It can be shown that the rest of the variables are linear equalities on the $path$ variables. For some constants $const(S)$, $const'(S)$ (independent of the strategy and only dependent on S) we have:

$$\begin{aligned} p_0(S) &= const(S) \times path(S) \\ p_1(S) &= const(S) \times path(S) \\ E = \sum_S & const(S) \cdot path_{pass}(S) + const'(S) \cdot path_{fail}(S) \\ C = \sum_S & const(S) \cdot path_{pass}(S) + const'(S) \cdot path_{fail}(S) \end{aligned}$$

Thus, we have linear equations relating all the variables of interest, along with corner cases:

$$\begin{aligned} path(0, \ldots, 0) &= 1 \\ \forall S = (x_1, y_1, \ldots, x_n, y_n);\ \sum_{i \in 1 \ldots r} [x_i + y_i] = m + 1 :\ path(S) &= 0 \end{aligned}$$

And the objectives are linear as well, enabling a linear programming solution.

The Linear Program (LP) has a total of $O(m^{2r})$ variables, with total size of the LP encoding being $O(poly(m^r))$. As a result, the complexity of the solution is: $O((m^{2r})^{3.5} \times \log(poly(m^r))$, i.e., $O(m^{7r} r \log m)$. Thus, we have:

Theorem 4.2.2 (Abilities) *We can find the optimal strategy for Problem 4.2.1 in* $O(m^{7r} r \log m)$.

Discussion: The astute reader may have noticed that the complexity, especially when r is large, can be rather high, due to r being present in the exponent of the complexity expression. Therefore, the linear programming approach will not scale if r is large. Instead, we will need to resort to an approximate approach, presented in the next section.

4.2.2 Incorporating Prior Information

We now generalize our algorithms for when we have prior information about items. For instance, we may have a machine learning algorithm (say a classifier) that analyzes items and assigns probabilities of passing the filter to each item. As an example, if we were doing content moderation of images, there are automated algorithms that analyze each image (perhaps by using the distribution of colors or by looking for specific patterns) and provide a probability for whether the image is likely to be inappropriate for children.

Modified Setting: Instead of having a single prior probability or selectivity s for all items, we now have prior probabilities s_i for each item i, representing the probability of the item satisfying the filter. We let $s'_1, s'_2, \ldots, s'_l$ be the *distinct* set of priori probabilities. We expect l to be much smaller than n, the total number of items. For instance, if $s_1 = s_2 = s_5 = 0.8, s_3 = s_4 = s_6 = 0.3$, then $s'_1 = 0.8, s'_2 = 0.3$, i.e., $l = 2$.

Solution: Our algorithm will effectively design a distinct strategy for each distinct probability s'_j. We characterize the new state space as: $(x_1, y_1, \ldots, x_r, y_r, j), j \in 1 \ldots l$. The last coordinate in this state space effectively encodes the a-priori probability of the item we are operating on.

Each item with priori probability $s_i = s'_j$ will then begin filtering at state $(0, \ldots, 0, j)$, i.e., the start state for the strategy corresponding to s'_j. On asking a question and getting an answer from a worker, we transition from $(x_1, y_1, \ldots, x_r, y_r, j)$ to $(x_1, y_1, \ldots, x_i + 1, y_i, \ldots, x_r, y_r, j)$ or $(x_1, y_1, \ldots, x_i, y_i + 1, \ldots, x_r, y_r, j)$ — that is, the last coordinate remains fixed, while one of the other coordinates is incremented based on the given worker answer.

When computing the strategies in the previous section, we had set $p_0(0, 0, \ldots, 0) = 1 - s$. Here, we have a different probability p_0 depending on the last coordinate. We have,

$$p_0(0, 0, \ldots, 0, j) = frac(s'_j) \times (1 - s'_j)$$

The probability above is a product of two factors: The first factor is the fraction of items that begin at $(0, \ldots, 0, j)$, i.e., the probability that any item has prior probability s'_j. The second factor is the probability that an item does not satisfy the filter, given that it begins processing at $(0, \ldots, 0, j)$—i.e., $(1 - s'_j)$

Given these modifications, once again, the path flow property can be leveraged to derive a linear program, giving us:

Theorem 4.2.3 (Problem 3.2.1+Priors) *We can find the optimal strategy for Problem 3.2.1 with priors* *in* $O(m^{7r} r l^{3.5} \log(ml) + m^{2r} l)$, *where* l *is the number of distinct* s_i.

The proof of complexity is a straightforward extension of the argument in the previous section.

Discussion Notice that even if $r = 1$, l can be as large as n (the total number of items), and as a result, this approach can be rather inefficient when l is large. Instead, we will need to resort to an approximate approach, presented in the next section.

4.3 The Posterior-Based Representation

We now describe the posterior-based representation. This representation, unlike the previous representation, is approximate, that is, the representation does not comprise a complete record of the state of processing of an item — some information is lost. However, the amount of information that is lost is user-controlled; i.e., there is a parameter that allows the user to specify how much information is lost. We first describe the representation for the basic setting considered in Chapter 3—that is, all workers are assumed to be equally error-prone. Considering the basic setting allows us to demonstrate, for a simple case, how the states in the answer-record representation, discussed previously, map to those the posterior-based representation. We then show that while the strategies computed using the posterior-based representation may not have optimal cost, as the amount of information recorded is increased, the strategies computed using the posterior-based representation tend towards optimal cost.

Then, in the next section, we will provide generalizations for worker abilities (that we considered in Section 4.2.1), and then for incorporating prior information (that we considered in Section 4.2.2). Unlike the answer-record representation whose state space scales rapidly (with some parameter dependent on the generalization considered), the dimensionality of the posterior-based representation stays constant independent of which generalization is considered.

4.3.1 Basic Setting

Outline: We begin by describing the mapping between representations, followed by showing that no information is lost (for the basic setting) when considering the initial version of the posterior-based representation. Then, we introduce approximations, and prove that even though approximations lead to sub-optimal strategies due to loss of information, in the limit, the the strategy computed using the approximate posterior-based representation is asymptotically optimal.

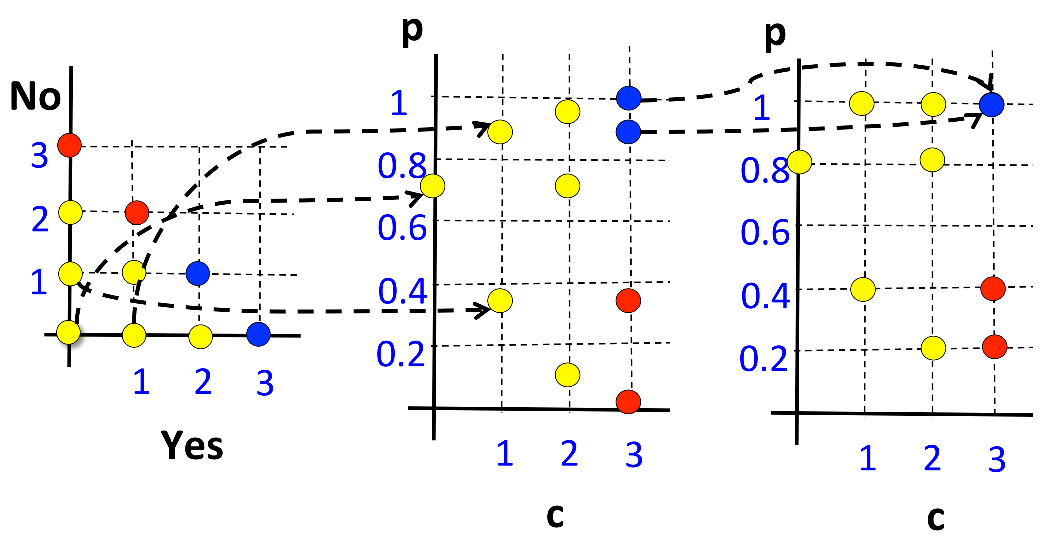


Figure 4.1: Mapping Between Representations: Left: Answer-record Representation; Middle: Continuous Posterior-based Representation; Right: Discretized Posterior-based Representation

Preliminaries: In the basic setting, the answer-record representation had a collection of states S represented using a pair (x, y), where x is the total number of YES answers so far, while y is the total number of NO answers so far, as shown in Figure 4.1 (left). The figure shows a strategy that has a Cont decision (Yellow) for all states (x, y) such that $x + y \leq 2$, and has Pass (blue) for (3, 0) and (2, 1), and Fail (red) for (0, 3), and (1, 2).

Instead, the posterior-based representation has a new state space represented using two components (p, c), encoding a probability p, which represents the probability that an item has $V = 1$ given the answers obtained so far, denoted:

$$p = \Pr[V = 1|(x, y)]$$

and cost c, which represents the total cost incurred so far, i.e., $(x + y)$.

As in the answer-record representation, in this representation, a strategy takes as input a state (p, c), and outputs a probabilistic decision: "Fail", "Pass", or "Continue".

Mapping: We show the correspondence by mapping states in the strategy in the answer-record (left) representation to the posterior-based (middle) representation. The mapping is shown in Figure 4.1

for $s = 0.7, e_0 = e_1 = 0.2$ with dashed lines for three states (other mappings are omitted for clarity).

As can be seen in the figure, the state $(0,0)$ maps to precisely $(s,0) = (0.7,0)$, since $\Pr[V = 1|(0,0)] = s = 0.7$, and since $x + y = 0 + 0 = 0$. The state $(1,0)$ maps to precisely $(0.903,1)$ since $\Pr[V = 1|(1,0)] = \frac{se_1}{se_1+(1-s)(1-e_0)} = 0.903$, and since $x + y = 1 + 0 = 1$, while state $(0,1)$ maps to precisely $(0.368,1)$ since $\Pr[V = 1|(0,1)] = 0.368$, and since $x + y = 0 + 1 = 1$.

Thus, each state (x,y) (in the left in the figure) maps to precisely one state in the new representation (in the middle in the figure). It is also easy to see that each state in the new representation can correspond to at most one state (x,y). To see this, let there be two distinct states (x',y') and (x,y) in the answer-record representation that map to the same state in the posterior-based representation. Thus, $x' + y' = x + y$. Now, without loss of generality, let $x' < x$. Then, the p value associated with (x',y') is smaller than that associated with (x,y), leading to a contradiction. Thus, strategies in the answer-record representation may be represented in the posterior-based representation without any loss in information.

Optimality: Every strategy in the answer-record representation can be represented in the posterior-based representation, by substituting the decisions for every state in the posterior-based representation from the answer-record representation, as we saw in Figure 4.1. Furthermore, there are no better strategies in the posterior-based representation, that is, the best strategy in the posterior-based representation is no better than the best strategy in the answer-record representation. This is because the only states in the posterior-based representation that matter are the ones that are reachable in any strategy; those are precisely the ones that have a corresponding state in the answer-record representation. Thus, we have:

Theorem 4.3.1 (Optimality) *For the basic setting, the optimal strategy in the posterior-based representation would be no better, and no worse, in terms of expected monetary cost, than the optimal strategy in the answer-record representation.*

Approximation: The theorem stated above states that the optimal strategy in the posterior-based representation would have same expected cost as the optimal strategy in the answer-record representation. Instead of storing the set of reachable states in the posterior-based representation and computing a strategy using those states, we instead compute strategies on an approximate discretized version of the entire posterior-based representation states; when we consider the generalization of worker abilities, approximations will become more necessary. As we will see in Chapter 9, even our approximate solutions have very good performance (i.e., very low cost for same error threshold).

In particular, we approximate the state (p,c) by using discretization. We discretize the probability

p into intervals. (Note that c is already discrete.) We use a discretization factor δ, and we divide the $[0,1]$ interval for p into δ intervals of size $1/\delta$ each. For instance, if $\delta = 2$, then $[0,1]$ will be divided into two parts: $[0, 0.5]$, $(0.5, 1]$. The larger the discretization factor δ, the smaller will be our intervals, and our intervals will be more in number.

Our state space S is now restricted to (p, c) where p is an integer multiple of $1/\delta$. The discretized state space is depicted in Figure 4.1 (Right) for $\delta = 5$. As can be seen in the figure, the two blue states in the full posterior-based representation (middle) map to the same state in the approximate posterior-based representation (right); there are infinitely many other mappings from the states in the full representation to the approximate discrete one, but we omit them from the figure for clarity.

Now, if, on getting an answer a for a question asked when at state (p, c), (where p is an integer multiple of $1/\delta$), our updated posterior probability value is p' and the cost is c', we round p' up to p'', the nearest integer multiple of $1/\delta$, and the new state will be (p'', c').

Approximate Solution: Now that we have a discrete set of states, we can once again use path-based reasoning, and linear programming on paths to find the best strategy, as in the answer-record representation. (Notice that our approximate posterior-based representation is once again a Markov Decision Process: we have a set of states, decisions to be made at each state, and probabilistic transitions from each state based on the decision that is made.)

We therefore have the following theorem, which uses a straightforward extension of the complexity argument of Theorem 4.2.2:

Theorem 4.3.2 *We can find a strategy using the approximate posterior-based representation for the basic problem in Chapter 3, in* $O(\delta^{3.5} m^{3.5} \log(m\delta))$, *where* δ *is the discretization factor.*

Thus, by adjusting the value of δ, the user can control how much computational cost she wishes to use to compute the strategy. The more computational cost used by the user, the better the strategy will be, as we will see next.

Convergence: As we saw in Figure 4.1, multiple states in the answer-record representation may map to the same state in the approximate posterior-based representation. For example, the two blue states in the answer-record representation map to two separate states in the full posterior-based representation, both of which map to a single discrete blue state in the approximate posterior-based representation.

However, as we increase the discretization factor (and therefore the number of intervals), the likelihood that multiple states in the answer-record representation will map to the same discrete state in the approximate posterior-based representation will go down; as a result, the cost of the optimal strategy in the approximate discrete posterior-based representation tends towards optimal cost. Formally,

Theorem 4.3.3 (Asymptotic Optimality) *As $\delta \to \infty$, the cost of the optimal strategy in the approximate posterior-based representation will tend to the cost of the optimal strategy in the answer-based representation.*

Discussion: In this subsection, we demonstrated that even though the strategies computed using the approximate posterior-based representation do not achieve the same low monetary cost of the exact answer-record representation, we can get as close as we want to that cost by varying the user-controlled discretization factor δ.

This guarantee seems to not be that useful for the basic setting of Chapter 3, where the answer-record representation leads to a tractable solution. However, we will find that similar guarantees hold for the generalizations considered next. While tractable solutions are not possible for those generalizations using the answer-record representation, they are indeed possible with the posterior-based representation.

4.3.2 Generalization of Worker Abilities

Recall that in Section 4.2.1, for the answer-record representation, we found that representing the answers from each worker individually led to an explosion in the state space. In this section, we describe how we may leverage the posterior-based representation when we have worker abilities.

Also in Section 4.2.1, we had briefly mentioned that our techniques would directly apply to the simpler generalization of infinite worker classes, where instead of having r distinct workers, we had r infinite worker classes, with each class having a distinct error rate. The key difference is that if one of the r workers answers a question on an item, that worker will not be asked from that point on; while in the r infinite worker classes case, the same worker class may be used multiple times on the same item.

Here, we revisit that generalization: our guarantees for asymptotic optimality only hold for the simpler generalization of r worker classes, and do not hold for the generalization of r distinct workers. This is because even as δ increases without bound, for the case of r distinct workers, if we do not record *exactly which worker gave us which answer* (like we do in the answer record representation), we will not be able to achieve optimal cost. Thus, in adapting to r distinct workers, there are two sources of approximation: one, from discretization (like we saw in the previous section), and second, from using the r infinite worker classes generalization for the r distinct worker case.

We begin by describing the changes in representation that apply to the entire section, then discuss the infinite worker classes case (along with the associated optimality guarantees), and then discuss the r distinct workers case.

Changes in Representation: Unlike in the answer-record representation, where we had $2r$ coordinates in the representation corresponding to the r workers, here, the posterior-based representation continues to use two coordinates (p, c). Thus, the size of the posterior-based state space does not change when we have many workers with different abilities — but, as we will see later, the cost of computing the strategy does change.

We briefly discuss how transitions happen in this new representation; in short, the only aspect that changes is how the probability p is updated on receiving an answer. At a state (p, c), if a Cont decision is made (instead of a Pass or a Fail), then a worker is asked the question on the item, and the probability p (of $V = 1$ given the set of answers, including the new answer), is updated to p', and c is updated to c' (based on the cost of having a worker answer a question).

Infinite Worker Classes: We first consider the full posterior-based representation before discretization, and then discuss discretization. Recall that in the infinite worker classes case, instead of having r workers with different abilities, we have r infinite worker classes. That is, there are r classes of workers, such that, at any state, with probability $1/r$, our question is answered by a worker with error rate $e_0(w_1), e_1(w_1)$, with probability $1/r$, by a worker with error rate $e_0(w_2), e_1(w_2)$, and so on. These classes are infinite; that is, if we sample a worker from class 1, the probability of getting a worker from class 1 does not change in the future. In this scenario, the answer-record representation is the same $S = (x_1, y_1, \ldots, x_r, y_r)$: but with one difference; in the previous setting, at most one of x_i or y_i is 1; here x_i or y_i can both be as large as m (because there may be as many as m YES or NO answers from a given worker class.)

We state the following lemma without proof.

Lemma 4.3.4 (No Loss of Information) *With infinite worker classes, all states $s_1, s_2, \ldots, s_a$ in the answer-record representation that map to the same state $s = (p, c)$ in the full posterior-based representation have the following properties:*

- *For the same answer obtained at $s_1, \ldots, s_a$ (YES/NO from any worker class), the resulting states $s'_1, s'_2, \ldots, s'_a$ all map to the same state $s' = (p', c')$.*
- *The probability of getting a YES or a NO from a specific worker class at $s_1, \ldots, s_a$, given that a question is asked, is identical for each of $s_1, \ldots, s_a$.*

We can now state the following theorem:

Theorem 4.3.5 (Optimality with Worker Classes) *With infinite worker classes, the optimal strategy in the full posterior-based representation has the same cost as the optimal strategy in the answer-record representation.*

Proof 4.3.6 *The proof of the above theorem is not as straightforward as the proof of Theorem 4.3.1, where in we could simply show a one-to-one correspondence between states in the answer-record and posterior-based representations.*

For convenience, we prove the above theorem when there are two worker classes, but our technique will work for the r-worker class case. Let $\mathcal{S}'$ be the subset of states in the posterior-based representation that are reachable if no Pass/Fail decisions are made at any state until $c = m$ (i.e., the probabilities of Pass or Fail are 0 until $c = m$), while $\mathcal{S}$ is the set of all states in the answer-record representation. For each state $s \in \mathcal{S}$ there is a state $s' \in \mathcal{S}'$ which it maps to. For each state $s' \in \mathcal{S}'$, there can be multiple states $s_1, s_2, \ldots, \in \mathcal{S}$ that map to it.

We show that for every strategy in the answer-record representation, we can find a better strategy in the answer-record representation such that the same decision is made for all states that map to the same state in the posterior-based representation. This strategy is therefore a strategy in the posterior-based representation.

We use induction starting from states at $(*, m)$. Consider the set of all reachable states in $\mathcal{S}$ that map to a specific (p, m_0) in the posterior-based representation. It is easy to show that the same decision must be made at all those states (Pass or Fail). Else, we can improve the strategy by ensuring that the same (better) decision is made at all states. Thus, all the states in $\mathcal{S}$ mapping to (p, m) must have the same decision.

Now assume that all states until $c = i$ have been mapped to their corresponding states, with the identical decisions for all states in $\mathcal{S}$. Consider states at $c = i - 1$. Let there be two states such that they both map to $(p, i - 1)$. These states are identical in that they transition to the same states after asking a question. Without loss of generality, assume that Pass is better at the states than Fail. In that case, the two states can only be improved by having the Fail probability set to 0. Now, we have two states such that the Continue probability is x for one, and y for the other with $x < y$. Let the current path flow into the first state be f_1 and the path flow into the second state be f_2. The lemma above indicates that the path flow from both these states will be indistinguishable to future states. If we set the continue probability for both states to be $(f_1 x + f_2 y)/(f_1 + f_2)$, then the same amount of path flow leaves both the states, and the strategy is unchanged in terms of cost and error. This generalizes to the case when we have multiple states mapping to the same one.

Therefore, all states in $\mathcal{S}$ mapping to $(p, i-1)$ for all p in the optimal strategy must have the same decision.

Hence proved.

Furthermore, we have:

Theorem 4.3.7 (Asymptotic Optimality) *For infinite worker classes, as $\delta \to \infty$, the cost of the optimal strategy in the approximate posterior-based representation will tend to the cost of the optimal strategy in the full posterior-based representation.*

Approximation to Worker Abilities: While we have proved optimality for the posterior-based representation for infinite worker classes, the proof does not capture the worker abilities generalization discussed in Section 4.2.1 precisely, because as soon as a worker answers a question (with a YES/NO), the worker can no longer be asked any further questions. Therefore, by representing the state using just two numbers p, c, we are certainly losing information of which workers have already answered questions, and our solution will be necessarily approximate.

We now further approximate via discretization (as discussed in the previous section). Thus, in this case, there are two sources of approximation.

However, as we will see in the experiments in Chapter 9, the two approximations we have made do not hurt performance; our solution is still near-optimal. We have the following, which uses a straightforward extension of the complexity argument of Theorem 4.2.2:

Theorem 4.3.8 *We can find a posterior-based strategy for the Problem 4.2.1 with worker abilities provided, in $O(m^{3.5}\delta^{3.5}\log(mr\delta) + m\delta r)$, where δ is the discretization factor.*

Notice that r appears as a logarithmic factor in first term of the complexity. This is because the linear equations in the linear program scale up by $O(r)$ — we need to consider transitions from each state (p, c) based on r possible answers: YES/NO from each worker. Since the complexity is no longer exponential in r, it is much faster to compute the optimal strategy in the approximate posterior based representation than it is in the answer-record representation.

4.3.3 Incorporating Prior Information

We now consider the generalization described in Section 4.2.2. Recall that our approach for generalization in the answer-record representation was to have a strategy computed for each individual distinct prior probability s'_j value as provided by an automated algorithm or human. This number could be as large as $O(n)$, where n is the number of items. As a result, our generalization, even when the number of workers or worker classes is small, ended up being difficult to compute.

We now discuss how we may leverage the posterior-based representation $S = (p, c)$ for this generalization. We discuss the generalization for the basic setting, that is, all workers are alike and independent, though our technique is easily generalizable to when we have distinct worker abilities.

The key idea that we use for this generalization is to set the path flow into $(s'_j, 0)$ to be equal to $frac(s'_j)$, i.e., the fraction of items with prior probability s'_j. Thus, the total path flow into all states with cost $c = 0$ is still 1, as before.

With the full posterior-based representation, the optimal strategy has just as low cost as the answer-record representation, formalized in the theorem below:

Theorem 4.3.9 (Optimality) *With $s_1, s_2, \ldots, s_n$ provided, the optimal strategy in the posterior-based representation has the same cost as the optimal strategy in the answer-record representation.*

We will now discretize the probability p, as before. As we increase the discretization factor δ, the cost of the optimal strategy in the discretized posterior-based representation will tend to the cost of the optimal strategy in the full posterior-based representation.

Theorem 4.3.10 (Asymptotic Optimality with Priors) *As $\delta \to \infty$, with priors, the cost of the optimal strategy in the approximate posterior-based representation will tend to the cost of the optimal strategy in the answer-record representation.*

We then have:

Theorem 4.3.11 *We can find a posterior-based strategy for Problem 4.2.1 with prior probabilities provided, in $O(\delta^{3.5} m^{3.5} \log(ml\delta) + lm\delta)$, where δ is the discretization factor.*

Thus, unlike the answer-record representation for this generalization, this representation does not have a computationally expensive $O(n^{3.5})$ factor.

4.4 Other Generalizations

We now discuss other generalizations described in the introduction. We first discuss generalizations that improve the cost (for fixed error), then discuss generalizations that provide enhanced functionality. For the sake of clarity, we describe the addition of each individual aspect one at a time to the basic setting discussed in the previous section. In practice, we may wish to use all the aspects at once. It is straightforward to construct the solution involving all aspects at the same time.

Furthermore, we describe our generalization for the answer-record representation. It is straightforward to construct the posterior-based representation based on our solution for the answer-record representation.

We describe the following generalizations:

(*a*) *Improvements:*

(1.) *Difficulty:* The algorithms in Chapter 3, and in this chapter so far assume that all items are equally hard or equally easy to filter—that is, they assume that all humans have the same error rate on every item. However, this assumption may not hold in practice. As an example, checking if a blurry picture contains a cat is much harder to do (and is more error-prone) than a clear picture.

(*b*) *Additions:*

(1.) *Requesting Specific Workers:* The algorithms in Chapter 3 and in this chapter so far do not request that specific workers answer, nor pay workers differently. In Mechanical Turk, for instance, there are workers with qualifications who are paid more while workers without qualifications are paid less, and for any question, we may choose to use a more qualified or less qualified worker. Here, we will consider the addition of the functionality of being able to request that specific workers answer and being able to pay them different amounts.

(2.) *Latency:* The problem statements described so far only have monetary cost and error as objectives, not latency. Latency is important in many crowdsourcing applications. We will consider the addition of a latency constraint in our problem statement.

(3.) *Scoring:* The problem statements described so far only consider binary filtering: we would also like to perform scoring, i.e., identifying the appropriate score or rating of an item, say from $1 \dots 5$. Furthermore, we allow weighted error objectives, i.e., different ways of assess the result of filtering. For instance, it is much worse to score an item with true rating 1, as a 5, instead of a 2.

4.4.1 Outline and Summary of Results

In Table 4.1, we show the complexity results for each of the generalizations considered; the two columns correspond to the complexity of algorithm computing the strategy using the answer-record representation, and the complexity of the algorithm computing the strategy using the posterior-based representation.

Functionality	Answer-record	Approximate Posterior-based (δ)
Problem 4.2.1	$m^{7r} r \log m$	$(m\delta)^{3.5} \log(mr\delta) + m\delta r$
Problem 4.2.1+Priors	$m^{7r} r l^{3.5} \log(ml) + m^{2r} l$	$(m\delta)^{3.5} \log(mrl\delta) + m\delta rl$
Problem 4.2.1+Difficulty	$m^{7r} r \log(md) + m^{2r} d$	$(m\delta)^{3.5} \log(mrd\delta) + m\delta rd$
Problem 4.2.1+Picking Workers	$m^{7r} r \log m$	$(m\delta)^{3.5} \log(mr\delta) + m\delta r$
Problem 4.2.1+Latency	$m^{7r} r t_o^{3.5} \log(mt_o) + m^{2r} t_o$	$(m\delta t_o)^{3.5} \log(mr\delta t_o) + m\delta r t_o$
Problem 4.2.1+Scores	$m^{3.5ru} ru \log m + m^{ru}$	$(m\delta^{(u-1)})^{3.5} u \log(mr\delta) + m\delta^{u-1} r$

Table 4.1: Comparison of complexity: For clarity, we only show the complexity of adding one generalization at a time to the setting of Problem 4.2.1

We divide the rows into two parts: the complexity on adding each of the individual improvement-based generalizations to the setting with worker abilities, followed by the complexity on adding each of the functionality addition-based generalizations to the setting with worker abilities.

4.4.2 Overall Approach to Generalizations

Our approach in addressing the generalizations will be to construct a representation that records all possible information pertaining to the state of processing of an item (possibly by adding additional dimensions), and then leveraging path flows so that the optimal strategy can be found using linear programing. That is, we need to show that p_0, p_1, E, C are linear functions of the *path* variables under new assumptions. This process in fact requires the same steps we have used above for the two generalizations of worker abilities and prior information:

1. Describing the new assumptions and the new metrics.
2. Constructing the state space and arguing that the state space captures all the information necessary for a strategy to make a decision at a state.
3. Describing the set of possible decisions made at a state (sometimes more than Pass, Fail, Continue).
4. Capturing the metrics of a strategy using non-linear equations like Equations 3.2, 3.3 above
5. Arguing that by considering path flow, the transformation of the non-linear equations into linear equations and constraints is correct.

Since describing all the items enumerated above is rather tedious, we will instead only describe the key ideas associated with each of the generalizations.

We begin with the improvement generalization, then discuss functionality addition generalizations.

4.4.3 Improvement Generalization: Difficulty

In the previous section, and in the previous chapter, we assumed that the error rate e_0, e_1 is the same for every item — that is, e_0 and e_1 are fixed. However, this assumption may not hold in practice. For instance, when identifying if a picture contains a cat, humans are much more likely to answer the question correctly if the picture is clear than if it is blurry.

Instead of having a single false positive or false negative rate for all items, we assume that there is an inherent known distribution across error rates for items. We have the following (discrete) distributions — estimated in advance using sampling, or using prior history:

$$\Pr[e_0|V = 0] \text{ and } \Pr[e_1|V = 1]$$

For instance, it may be the case that for items that satisfy the filter ($V = 1$), 10% of the items are really hard for a human to judge, with $e_1 = 0.4$ (i.e., most people make mistakes for those items), and 90% of the items are really easy for a human to judge, with $e_1 = 0.1$ (i.e., most people answer questions correctly for those items).

Then, p_0 — the probability of arriving at a state $S = (x_1, y_1, \ldots, x_r, y_r)$ and the item satisfying the filter — is now a sum of quantities instead of a single quantity: (where $const(S), const'(S)$ are constants that depend just on S and not on the strategy)

$$\begin{aligned}
p_0(S) &= \Pr[\text{reach}(S), V = 0] = \Pr[\text{reach}(S)|V = 0] \cdot \Pr[V = 0] \\
&= \sum_e \Pr[e_0 = e|V = 0] \cdot \Pr[\text{reach}(S)|V = 0 \wedge e_0 = e] \cdot \Pr[V = 0] \\
&= path(S) \sum_e \Pr[e_0 = e|V = 0] \cdot const(S) \\
&= path(S) \cdot const'(S)
\end{aligned}$$

Similar relationships hold for p_1. It is easy to see that E, C are also linear functions of the *path* variables. Thus, we have:

Theorem 4.4.1 (Difficulty) *We can find the optimal strategy for Problem 4.2.1 with difficulty information in $O(m^{7r} r \log mrd + m^{2r} d)$, where d is the number of discrete values for e_0, e_1.*

Incidentally, notice that difficulty also captures the case where errors between humans are correlated; the fact that many humans are making mistakes on the same item can be explained by the item having a higher error rate or difficulty than a different non-ambiguous item.

4.4.4 Functionality Addition Generalization: Picking Workers with Costs

Unlike the situation in Section 4.2.1, where the strategy is assigned a worker randomly when it chooses to ask a question, we now consider the case where we can choose to ask any one of our r workers, who have differing costs $c_1, \ldots, c_r$. This scenario is relevant in may crowdsourcing marketplaces, such as ODesk, where the worker who is asked a question may be controlled by the application.

The state space is identical to the one described in Section 4.2.1, that is: $S = (x_1, y_1, x_2, y_2, \ldots, x_r, y_r)$ However, the decision that is made at any state is now not restricted to "Pass", "Fail" or "Continue" — when continuing to process the item, we can choose to ask any one of the r workers (assuming they haven't been asked before). We have:

$$\begin{aligned} path(x_1, y_1, x_2, y_2, \ldots, x_r, y_r) &= path_{fail}(x_1, y_1, x_2, y_2, \ldots, x_r, y_r) \\ &+ path_{pass}(x_1, y_1, x_2, y_2, \ldots, x_r, y_r) \\ &+ \sum_{1 \le i \le r;\ x_i = y_i = 0} path^i_{ask}(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, 0, 0, x_{i+1}, y_{i+1}, \ldots, x_r, y_r) \end{aligned}$$

where $path^i_{ask}$ represents the path flow based on asking the i th worker an additional question (if the worker has not answered a question before). Moreover, we have:

$$path(x_1, y_1, x_2, y_2, \ldots, x_r, y_r) = \sum_{1 \le i \le r;\ x_i = 1 \text{ or } y_i = 1} path^i_{ask}(x_1, y_1, \ldots, x_{i-1}, y_{i-1}, 0, 0, x_{i+1}, y_{i+1}, \ldots, x_r, y_r)$$

where $path_{ask}$ is now replaced by $path^i_{ask}$, the path flow on asking the ith human worker. We then have:

Theorem 4.4.2 (Picking Workers with Costs) *We can find the optimal strategy for Problem 4.2.1 with the ability of picking specific workers in* $O(m^{7^r} r \log m)$.

Discussion: This generalization suffers from the same problem as the related generalization in Section 4.2.1: when r is large, the complexity can be rather high, due to r being present in the exponent. Therefore, the approach will not work if r is large.

Furthermore, if the workers have different (non-integral) costs, then our cost coordinate may be rational rather than integral, as a result, we may need to discretize the rational numbers. Adding an additional discretization would only make our solution more approximate.

4.4.5 Functionality Addition Generalization: Latency

So far, we have not considered latency as part of our optimization, focusing instead on cost and accuracy. The reason is the following: our latency is, in the wost case, as large as the time taken for m questions to be answered in sequence. We ask one question on each item, and stop processing for each item based on the strategy, and we do so for all items in parallel. Thus, even in the worst case, the latency is not very large. If latency is critical, we can add a constraint to our objective, in the following manner.

We define latency as the number of round-trips taken to the crowdsourcing marketplace; we assume that multiple questions may be asked in parallel, and if asked in parallel, then would be all answered in one unit of time.

Then, our state space can be represented as: $S = (x_1, y_1, \ldots, x_r, y_r, t)$, where t is the total number of round-trips to the crowdsourcing marketplace taken so far in processing a given item.

We modify Problem 3.2.1 to the following by adding a constraint on the absolute number of round-trips:

Problem 4.4.3 (Problem 4.2.1+Latency) *Given an error threshold τ, a latency threshold per item t_0, and a budget threshold per item m, find a strategy that minimizes C under the constraint $E < \tau$ and $\forall$ reachable $(x_1, y_1, \ldots, x_r, y_r, t)$ $\sum_i [x_i + y_i] \leq m, t \leq t_0$.*

At each state, the strategy may ask one, two, . . ., or t_0 questions in parallel. Thus, instead of having one variable $path_{ask}$, we have $path^1_{ask} \ldots path^{t_0}_{ask}$, corresponding to path flow on asking $1, 2, \ldots, t_0$ questions in parallel. Also, we have:

$$\begin{aligned} path(x_1, y_1, \ldots, x_r, y_r, t) &= path^1_{ask}(x_1, y_1, \ldots, x_r, y_r, t) + \ldots + path^{t_0}_{ask}(x_1, y_1, \ldots, x_r, y_r, t) \\ &+ path_{pass}(x_1, y_1, \ldots, x_r, y_r, t) + path_{fail}(x_1, y_1, \ldots, x_r, y_r, t) \end{aligned}$$

We then have:

Theorem 4.4.4 (Latency) *We can find the optimal strategy for Problem 4.4.3 in $O(m^{7r} r t_0^{3.5} \log(m t_0) + m^{2r} t_0)$.*

Note that in addition to an upperbound on latency, we can also capture constraints on expected latency across items. For instance, we can enforce the constraint that the expected latency across all items should be less than some number t', or we can minimize expected latency, while enforcing constraints on expected cost and expected error.

Latency is a lot more critical in Finding (Chapter 5), since in that case, latency can be as large as the number of items. We will consider latency in greater depth in that chapter.

4.4.6 Functionality Addition Generalization: Scoring

So far, we have considered boolean filtering, where an item either satisfies or does not satisfy a filter. We can also handle the case where each item can have one of k scores or ratings $V = 1, \ldots, u$, with accuracies or error rates:

$$\Pr[w_k\text{'s answer is } j | V = i] = p_{(i,j)}(w_k)$$

where $p_{(i,j)}(w_k)$ is the probability that worker answers that the rating of an item is j when its actual rating is i. Thus, there is a set of $u \times (u-1)$ numbers that define the accuracy of a worker. (For each $i \in 1 \ldots u$, we need to know $(u-1)$ probability values corresponding to $p_{(i,j)}$ — the last probability value can be inferred from the remaining $u-1$.)

Then, at any point, we can represent the current set of answers using:

$$S = (x_1^1, x_1^2, \ldots, x_1^u, \ldots, x_r^1, x_r^2, \ldots, x_r^u)$$

That is, we record whether each worker has scored the item as $1, 2, \ldots, u$. Once again, the path flow property and linear programming may be leveraged to find the optimal strategy in the answer-record representation.

Moreover, we can also handle error metrics that penalize differently for assigning i or i', $i \neq i'$ to an item whose actual rating is j. For instance, we may penalize i higher than i' if i' is closer to j than i. The function $0 \leq pen(i,j) \leq 1$ refers to the penalty for judging an item with true value j, as i. Now, E is set to be the following expression:

$$E = \sum_S \Big[\; p_1(S) \times (\sum_{i,i\neq 1} pen(i,1) \cdot a_i(S)) + p_2(S) \times (\sum_{i,i\neq 2} pen(i,2) \cdot a_i(S)) + \ldots + \\ p_u(S) \times (\sum_{i,i\neq u} pen(i,u) \cdot a_i(S)) \; \Big]$$

where a_i is the probability that a score i is assigned to an item, and $p_i(S)$ is the probability that an item arrives at S with true score i.

4.5 Conclusions

In this chapter, we discussed a number of filtering generalizations. We provided extensions of the strategy computation techniques developed in the previous chapter that enable us to address all the generalizations, but lead to intractability in the representation and computation of the strategy for some generalizations. We then developed the posterior-based representation which does not suffer from the intractability issue in the answer-record representation, but leads to strategies that may not be optimal. We did, however, show that these strategies converge to optimal ones in the limit.

In Chapter 9, we experimentally evaluate the techniques for distinct worker abilities in the context of peer evaluation systems. That is, we want to use distinct peer graders to score submissions on a scale from 0—5.

In the next chapter, we consider crowd-powered finding. So far, we have assumed that we want to filter *all* items. In finding, we do not want to filter all items, we just want to *find a small number* of items that satisfy (or do not satisfy) the filter.

Chapter 5

Algorithm 2: Finding

5.1 Introduction

In this chapter, we develop algorithms for crowd-powered finding[1]. The basic version of the finding problem is the following: Given a (large) set of items and a number k, we want to use humans to find k items that satisfy a given predicate or filter.

The finding problem has several applications. As examples, a company may want to build a team of 20 young Java programmers from a large set of pre-screened resumes, or a travel website may want to identify 10 photos containing the Eiffel Tower from a dataset of 100,000 travel photos. Of course, one could apply filtering (Chapter 3 or 4), e.g., find all photos of Eiffel Tower in a set of 100,000 photos. But adapting filtering would result in considering the entire set of 100,000 items, and hence would be highly inefficient when we only need a small number of items satisfying the predicate, e.g., only 10 Eiffel Tower photos are desired. Therefore, we focus our efforts on identifying a small subset of input items with desired properties, instead of finding all items with desired properties.

Unlike filtering, where we our primary focus was on optimizing cost for a fixed accuracy bound (or vice versa), here, we focus on optimizing *both cost and latency* for a fixed accuracy bound. The reason is simple: in filtering, since we need to filter all items anyway, we do not save cost by filtering items in sequence, we might as well filter all items in parallel. On the other hand, in finding, we definitely save cost by evaluating one item at a time (and stopping once we have enough items), while incurring a higher latency.

Thus, unlike the filtering problem, the finding problem has an crucial **cost-latency tradeoff** which

[1]This chapter is adapted from our paper [168], written jointly with Anish Das Sarma, Hector Garcia-Molina, and Alon Halevy, but some details and generalizations are omitted for the sake of clarity.

this chapter studies in detail: At one end of the spectrum, there are solutions that require high monetary cost, but the overall latency of solving the problem is low, while at the other end, there are solutions that minimize cost but incur heavy latency. The following example illustrates this tradeoff.

Example 5.1.1 *Consider a data set $\mathcal{I}$ of images, from which we want to find 10 images that satisfy a predicate or filter f, e.g., whether it is a photo of a cat. We consider each image, and ask humans the question, e.g., "does this image show a cat?". Suppose on average that 20% of the photos are of cats. For the purposes of this example, we assume that humans do not make mistakes while answering questions.*

Since crowdsourcing marketplaces have high latencies, we consider algorithms where we ask (and get answers to) many questions in parallel, *and we do so over multiple "phases", where in each phase a number of questions are asked in parallel and answered together in one unit of time. Consider the following algorithms:*

1. **Sequential:** *Pick one image at a time, ask a human if the image shows a cat (in one phase), and then stop as soon as enough images satisfying f are gathered. The expected number of questions and phases for this algorithm is $\frac{10}{0.2} = 50$. Note that this algorithm is* cost-optimal, *i.e., it only asks as many questions as strictly necessary.*

2. **Parallel:** *Ask all images in parallel in a single phase, incurring a heavy monetary cost depending on $n = |\mathcal{I}|$. If $n = 10000$, the number of questions for this algorithm is 10000, while the number of phases is 1.*

3. **Hybrid-1:** *Ask multiple images in the same phase, but no more than necessary: Ask 10 images in parallel in the first phase, and if x_1 images were found that satisfied f, ask $10 - x_1$ in the next phase, and if x_2 were found in the second phase, ask $10 - x_1 - x_2$ in the third and so on. In this case, the expected number of phases for this algorithm is much smaller than 50, while the expected number of questions is the same as the sequential algorithm.*

4. **Hybrid-2:** *A slight modification of the algorithm above might ask more than 10 images in the first phase, hoping to get all 10 images in the first phase, but without incurring much more cost than necessary. For instance, we may ask $\frac{10}{0.2} = 50$ images in the first phase, with an expected number of 10 images satisfying f. The expected number of phases for this algorithm is less than Hybrid-1, while the expected number of questions is larger.*

As in Section 4.4.5, we measure time or latency by the number of *phases* of crowdsourcing we need to perform, with one or more items asked in each phase, and measure cost (as in the previous chapter) as the total number of questions asked.

We provide algorithms for the finding problem that are "optimal", i.e., they lie along the *skyline of solutions* of the cost-time tradeoff, and the balance between cost and time can be configured by the application. In other words, we provide knobs that let application designers control the point along the skyline that is desired.

We explore the cost-time tradeoff under two models of accuracy of human answers:

- *Deterministic:* In this setting, every human gives an accurate response to every question, which is a reasonable assumption for properties that are easily evaluated by humans, e.g., whether a resume mentions a date of birth before 1990. Recall that in the filtering problem, the deterministic setting was trivial — i.e., simply ask a human a question for every single item. Here, the deterministic setting is non-trivial due to the cost-time tradeoff.
- *Uncertain:* In this setting, humans may give erroneous responses, which is often a more realistic assumption for properties that may be hard to evaluate, e.g. checking if a blurry photo contains the Eiffel Tower.

5.1.1 Outline of Chapter

Here is the outline for this chapter:

- We study the cost-time tradeoff for the deterministic setting as follows: (Section 5.3)
  - Given the cost-optimal sequential algorithm $\mathcal{A}$, we find an algorithm $\mathcal{A}'$ that asks the same questions as $\mathcal{A}$, but minimizes the number of phases of the algorithm. Essentially, we find the best parallelization $\mathcal{A}'$ of $\mathcal{A}$.
  - Given the cost-optimal sequential algorithm $\mathcal{A}$, and an (additive or multiplicative) approximation bound α, we find an algorithm $\mathcal{A}'$ that asks at most α more questions than $\mathcal{A}$ for every input instance, but minimizes the number of phases of the algorithm. Essentially, we can use this algorithm to find any optimal point in the cost-time tradeoff.
- We study the cost-time tradeoff when humans may give erroneous responses as follows: (Section 5.4)
  - We find a cost-optimal sequential algorithm $\mathcal{A}$ that minimizes cost. Unlike in the deterministic setting, this algorithm is non-trivial.
  - Given the sequential algorithm $\mathcal{A}$, we find an algorithm $\mathcal{A}'$ that asks the same questions as $\mathcal{A}$, but minimizes the number of phases of the algorithm.
  - Given the sequential algorithm $\mathcal{A}$ and an approximation bound α, we present two algorithms that ask at most α more questions than $\mathcal{A}$, but have provable guarantees on the number of phases.

   - We formally show that adaptations of filtering for the uncertain setting are arbitrarily worse in both cost and number of phases than our algorithms.
- We show that our techniques can be extended to the case when we desire bounds on expected cost and expected number of phases. (Section 5.5)

5.2 Definitions

We start by defining the finding problem (Section 5.2.1), and then presenting our metrics for comparing crowdsourcing solutions to each problem (Section 5.2.2). We then describe the formal problems we address.

5.2.1 Setting

As in filtering, we are given a (large) set of items $\mathcal{I}, |\mathcal{I}| = n$. Every input item either satisfies a given boolean filter f, or not. Our goal is to find k_1 or more items that satisfy the filter, and k_0 or more items that do not satisfy the filter. (In many practical scenarios, one of k_1 and k_0 may be 0.) We can also handle any general monotonic goals or output conditions based on one or more filters [168] — e.g., find k_1 items that satisfy filter$_1$ OR (k_0 items that satisfy the filter$_1$ AND k_3 that don't satisfy the filter$_2$) — but we omit the details for ease of exposition. We state our goals more formally as part of the problem description in Section 5.2.3.

For every item, we may ask humans a question on that item: the human may answer YES if he/she believes the item satisfies the filter, and NO otherwise. After having asked humans a few questions about a given item, the state of processing of an item I can be completely represented using the pair (x, y). (We once again assume that all humans are able to answer the question with the same degree of accuracy.) We define the pair of an item along with its state of processing, i.e., $(I, (x, y))$, x, y not both 0, to be *a partially evaluated item*.

We assume we are given a filtering strategy $\mathcal{F}$ from Chapter 3, that given a state (x, y), outputs one of three responses $\mathcal{F}((x, y)) = \mathsf{Pass}/\mathsf{Fail}/\mathsf{Cont}$. We consider two scenarios of human error in this chapter:

- Deterministic: In the deterministic setting, humans don't make errors while answering questions. Here, $\mathcal{F}$ is the simple strategy that returns Pass after one YES answer is obtained, and returns Fail after one NO answer is obtained, and Cont at $(0, 0)$.
- Uncertain: In the uncertain setting, humans may make errors. Here, $\mathcal{F}$ may be one of the

optimized filtering strategies from Chapter 3. Our finding algorithms are filtering strategy-agnostic, i.e., it can be used in conjunction with any filtering strategy.

Finding Algorithms: A finding algorithm, denoted $\mathcal{A}$, is given a data set of items, also called an *input instance* $\mathcal{I}$ and the output condition represented by a pair (k_1, k_0) (recall that the objective is to find k_1 items that satisfy the filter, and k_0 items that don't). The algorithm $\mathcal{A}$ maintains a *state of knowledge*, which is the set of partially evaluated items $\mathcal{SK} = \{I_i, (x_i, y_i)\}$. That is, $\mathcal{SK}$ contains the items, along with the state of processing for each item. As the algorithm asks more questions to humans for each item, this information grows by either gathering new property information for existing items (only necessary when human answers may be uncertain), or finding out properties of some new items.

A finding algorithm proceeds in *phases*. In phase i, the algorithm performs the following operations:

- **Item Selection:** (lines 1-6 in Algorithm 1) The algorithm picks a multiset of items to ask humans questions for in phase i. That is, algorithm A picks a multiset $\mathcal{Q} = \{I_j\}$, where $I_j \in \mathcal{I}$. If the algorithm decides to ask a question for an item that has not been seen before, then a new item is retrieved from the item database $\mathcal{I}$.
- **Human Questions:** (lines 7-10 in Algorithm 1) The algorithm *in parallel* asks different humans questions on the items in $\mathcal{Q}$. The set of newly obtained answers is added to the state of knowledge of the algorithm $\mathcal{SK}$.
- **Test for Solution:** (lines 11-14 in Algorithm 1) If the state of knowledge satisfies the output condition (k_1, k_0), i.e., it consists of k_1 items for which $\mathcal{F}(x, y) = \mathsf{Pass}$, and k_0 items for which $\mathcal{F}(x, y) = \mathsf{Fail}$, then the algorithm halts. Else, the algorithm moves on to the next phase.

An algorithm is said to return *a correct solution* if it finds k_1 items that satisfy the filter (based on the strategy $\mathcal{F}$) and k_0 items that don't.

Naturally, the core logic of the filtering algorithm rests in Line 1, wherein the multiset of questions $\mathcal{Q}$ is selected. That logic will be the focus of our chapter.

To enable us to compare the executions of various algorithms, we make a few natural assumptions on the execution of algorithms: (1) We assume that the order in which "new" items are retrieved by the algorithms from $\mathcal{I}$ is identical. (2) For each item, different algorithms receive the same answer for each question. For instance, if one algorithm receives an answer a when a question on item I is asked for the r'th time, then every algorithm would get the answer a for the r'th question on item I.

Algorithm 1: Algorithm Outline

```
Data: $k_1, k_0, \mathcal{F}$
Result: $\mathcal{SK}, T, C$
$T, C \leftarrow 0$;
$flag \leftarrow 0$;
while $flag == 0$ do
1       $\mathcal{Q} \leftarrow$ set of questions to ask in current phase;
2       $\mathcal{H} \leftarrow \varnothing$;
3       for $I \in \mathcal{Q}$ do
4           if $I \notin \mathcal{SK}$ then
5               $I \leftarrow$ get new item from $\mathcal{I}$;
6           $\mathcal{H} \leftarrow \mathcal{H} \cup \{I\}$;
7       ask $\mathcal{H}$ in parallel using crowdsourcing service;
8       $\mathcal{SK} \leftarrow \mathcal{SK} \cup$ answers of $\mathcal{H}$;
9       $T \leftarrow T + 1$;
10      $C \leftarrow C + |\mathcal{Q}|$;
11      $tmp1, tmp2 \leftarrow 0$;
12      for $(I_i, (x_i, y_i)) \in \mathcal{SK}$ do
13          if $\mathcal{F}(x_i, y_i) ==$ Pass then
                $tmp1 \leftarrow tmp1 + 1$;
            if $\mathcal{F}(x_i, y_i) ==$ Fail then
                $tmp0 \leftarrow tmp0 + 1$;
14      if $tmp0 \geq k_0 \wedge tmp1 \geq k_1$ then
            $flag = 1$;
```

5.2.2 Performance Metrics

We assume that each human question takes one unit of cost to answer, and that a batch of questions being asked in parallel take one unit of time to be answered. The performance of any algorithm as described above can be expressed in terms of two quantities on a fixed input instance I.

- **Latency** T: The total number of phases T of an algorithm is the latency of the algorithm. We ignore any other computation time, such as testing whether the current set of items satisfies the output condition, since that is an automated test that is negligible compared to human tasks. Notice that we are implicitly assuming that all humans take around the same amount of time to answer questions. While this assumption may not hold exactly in real scenarios, in practice, we find that a bulk of the answers arrive at the same time, while a small number of answers arrive much later; a common strategy, therefore, is to cancel the outstanding answers and then issue a new batch of questions. Our algorithms also apply to the case where we cancel outstanding answers.

- **Cost** C: The total monetary cost C is the number of human questions asked by the algorithm. If x_i is the number of human questions in the ith phase, then:

$$C = \sum_{i=1}^{i=T} x_i$$

Notice that we do not explicitly consider error E, because that is captured implicitly by our filtering strategy $\mathcal{F}$. If the filtering strategy $\mathcal{F}$ asks a lot of questions, then the error on each item will be low.

5.2.3 Optimization Problems

This section describes the optimization problems addressed in the rest of this chapter; in all definitions below k_1, k_0 refer to the number of items needed to satisfy the output condition. First, we describe the sequential problem, which is trivial for the case when humans do not make mistakes, but non-trivial for the uncertain setting:

Problem 5.2.1 (Sequential) *Given the output condition (k_1, k_0), design an algorithm $\mathcal{A}$ that asks one question in each phase and returns the correct solution incurring the least monetary cost on each input instance I.*

Recall once again that a solution for a finding problem identifies k_1 or more items that are inferred to satisfy the predicate, and k_0 or more items that are inferred to not satisfy the predict.

We define the *optimal cost* on a certain input instance $\mathcal{I}$, $C_{opt}(\mathcal{I})$ to be the cost taken by the sequential algorithm on the data set (i.e., the solution to Problem 5.2.1).

Then, we have the following problem which leverages the maximum parallelism possible while ensuring optimal cost on every input instance.

Problem 5.2.2 (Cost-optimal Max-parallel) *Given (k_1, k_0), design an algorithm $\mathcal{A}$ that for every input instance $\mathcal{I}$*

- *Algorithm $\mathcal{A}$ returns a correct solution and has $C(\mathcal{I}) = C_{opt}(\mathcal{I})$*
- *No other algorithm $\mathcal{A}'$ (which for every $\mathcal{I}$, returns a solution and has $C(\mathcal{I}) = C_{opt}(\mathcal{I})$) has lower latency.*

However, by trading off some cost, we may be able to achieve better parallelism.

Problem 5.2.3 (α-multiplicative-approx. MP) *Given (k_1, k_0), design an algorithm $\mathcal{A}$ such that for every $\mathcal{I}$:*

- *Algorithm $\mathcal{A}$ returns a correct solution and has $C(\mathcal{I}) \leq \alpha C_{opt}(\mathcal{I})$*
- *No other algorithm $\mathcal{A}'$ (which for every $\mathcal{I}$, returns a solution and has $C(\mathcal{I}) \leq \alpha C_{opt}(\mathcal{I})$) has lower latency.*

In other words, we provide an α-approximate cost on every input instance $\mathcal{I}$, and increase parallelism as much as possible. We can also define the problem based on additive approximation.

Problem 5.2.4 (α-additive-approx. MP) *Given (k_1, k_0), design an algorithm $\mathcal{A}$ such that for every input instance $\mathcal{I}$:*

- *Algorithm $\mathcal{A}$ returns a correct solution and has $C(\mathcal{I}) \leq \alpha + C_{opt}(\mathcal{I})$*
- *No other algorithm $\mathcal{A}'$ (which for every $\mathcal{I}$, returns a solution and has $C(\mathcal{I}) \leq \alpha C_{opt}(\mathcal{I})$) has lower latency.*

The problems described so far capture *instance-specific guarantees*, i.e., the goal is to design algorithms that provide approximation guarantees per data set instance $\mathcal{I}$, relative to the sequential algorithm. Next, we describe a problem wherein the goal is to quantify *expected monetary cost* and *expected latency* of algorithms. Note that while expected monetary cost and latency guarantees do not translate to instance-specific guarantees—i.e., there may be algorithms whose expected costs and latencies are low, but may do extremely poorly on some input instances—having expected cost and latency guarantees allows us to compare algorithms relative to each other on the two dimensions of cost and time.

Problem 5.2.5 (Expected Monetary Cost and Latency) *Given a finding algorithm $\mathcal{A}$, find its expected monetary cost and latency.*

5.3 Deterministic Setting

This section describes solutions to the problems defined in Section 5.2 when humans do not make mistakes. Recall that the main logic of Algorithm 1 lies in Line 1, where the algorithm selects a set of items $\mathcal{Q}$ to be asked questions in parallel, based on the current state of knowledge $\mathcal{SK}$. Moreover, since humans do not make mistakes, each item may be asked as part of $\mathcal{Q}$ at most once. Thus, we can simply represent $\mathcal{Q}$ using a single integer x, which signifies the number of new items to be asked in each phase.

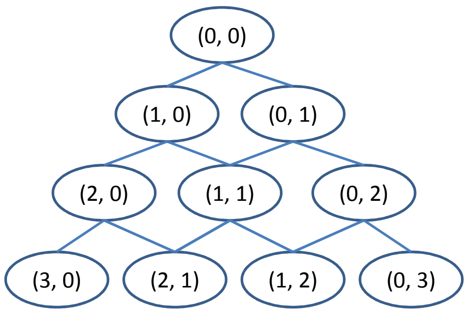


Figure 5.1: Possible states of the system on asking up to 3 questions.

When there is a single predicate with no errors in human answers, the state of execution $\mathcal{SK}$ can be simply represented using a pair (a, b), where a is the number of items that have been verified to satisfy the filter, and b is the number of items that have been verified to not satisfy the filter.

Example 5.3.1 *Figure 5.1 depicts the reachable states on asking up to 3 questions when we have not asked any questions yet (i.e., state (0, 0)). For the rest of this section, the output condition specifies that two or more items that satisfy the filter are desired, i.e.,* $k_1 = 2, k_0 = 0$. *Thus, states (2, 0), (2, 1) and (3, 0) are states that satisfy the output condition.*

5.3.1 Problem 5.2.2: Optimal Cost

We first consider the problem of minimizing phases, while keeping the cost the same as the sequential algorithm. In our example, it is clear that 2 questions may be asked in the first phase since there are no states when one question is asked that satisfy the output condition. Subsequently, if the resulting state of knowledge after the answers are obtained is $\mathcal{SK} = (1, 1)$, only one question may be asked in the next phase, while if the state is $(0, 2)$, then two questions may be asked in the next phase.

We now describe an online algorithm, called `OptCost`, that solves Problem 5.2.2. The algorithm proceeds as follows: Let the current state of knowledge be (a, b). Then, in the next phase, the algorithm asks $x_m + 1$ questions where x_m is the largest x such that:

$$\forall i, j \geq 0, \quad 0 \leq i + j \leq x, \quad (a + i < k_1) \vee (b + j < k_0) = TRUE \tag{5.1}$$

In other words, the algorithm asks precisely as many questions in each phase until one of the reachable states is one for which the output condition is met.

We have the following theorem:

Theorem 5.3.2 *Algorithm* `OptCost` *solves Problem 5.2.2.*

Proof 5.3.3 *We need to show that* `OptCost` *(a) has the same cost on every input instance as C_{opt} (b) uses as few phases as possible on every input instance. It is easy to see that (a) is true, since we do not at any point ask more questions than necessary. For (b), we show that Algorithm* `OptCost`*, has lower cost than any other algorithm that satisfies condition (a). Let one such algorithm be called* `Other`*.*

We show that for any input, at any phase i, `OptCost` *has asked more questions than* `Other`*. Since our output condition is monotonic, this property will imply that* `OptCost` *reaches a termination state in the same or fewer phases than* `Other`*, for every input instance.*

We use induction. Consider the first phase. Algorithm `OptCost` *asks $x_m + 1$ questions. Any algorithm asking greater than $x_m + 1$ risks asking more questions than necessary. Thus, at phase one,* `OptCost` *has asked as many or more questions than* `Other`*. Assume that* `OptCost` *has asked more questions than* `Other` *until phase i, with* `OptCost` *having asked y questions, while* `Other` *has asked $y' \leq y$ questions in total. Now, assume that* `Other` *has asked more questions in total than* `OptCost` *at phase $i + 1$, i.e.,* `OptCost` *has asked z questions, while* `Other` *has asked $z' > z$ questions in total. Moreover, by definition of Equation 5.1, starting from the state corresponding to y questions, there is a state corresponding to z questions in total that satisfies the output condition. Now, since* `Other` *begins at $y' \leq y$, there is an input instance where $z - y'$ additional questions in total are necessary to get to the terminating state, but* `Other` *asks $z' - y'$, which is more than necessary. Hence proved.*

5.3.2 Problem 5.2.3: Multiplicative Approximation

The algorithm, called α-`MultApprox`, proceeds as follows: Let the current state of knowledge $\mathcal{SK}$ be (a, b), and the number of questions asked so far be $y = a + b$. Then, the algorithm asks $\alpha \times (y + x_m + 1) - y$ questions in the next phase, where x_m is as defined in Section 5.3.1. Thus, the algorithm asks up to $(\alpha - 1)(y + x_m + 1)$ additional questions beyond what `OptCost` asks.

Consider our example with $\alpha = 2$. Consider the case when the resulting state of knowledge after the first phase is $(1, 3)$. (Thus, $y = 1 + 3 = 4$.) In this case, on asking one question, we can reach a state for which the output condition is satisfied $(2, 3)$ – thus `OptCost` will ask one question. However, α-`MultApprox` will ask $\alpha(y + x_m + 1) - y = \alpha(4 + 1) - 4 = 6$ questions in parallel in the next phase.

We have the following theorem:

Theorem 5.3.4 *Algorithm α-*`MultApprox` *solves Problem 5.2.3.*

Proof 5.3.5 *We need to show that the algorithm (a) has less than $\alpha \times C_{opt}$ on every input instance (b) uses as few phases as possible. First, we show that (a) is true. Notice that at any state, we only ask $\alpha \times (y + x_m + 1) - y$ where $(x_m + 1)$ is questions necessary to get to the closest reachable state that satisfies the output condition. Note also that from any state, the only reachable terminating states are those where $(x_m + 1)$ or more questions are used (by definition of x_m). For all those terminating states, the total cost is greater than or equal to $y + x_m + 1$. Since α*-`MultApprox` *asks at most $\alpha \times (y + x_m + 1) - y$ and since the current state has y questions asked already, α*-`MultApprox` *asks $\alpha \times (y + x_m + 1)$, which is less than or equal to $\alpha \times C_{opt}$ for every terminating state. Thus, (a) is true.*

For (b), we show that Algorithm α-`MultApprox`*, at any phase, has asked as many or more more questions than any other algorithm, called* `Other`*, that satisfies (a). As in the previous proof, this implies that the algorithm will terminate sooner.*

We use induction for the proof. Consider the first phase. Algorithm α-`MultApprox` *asks $\alpha(x_m + 1)$. Any algorithm that asks more questions risks asking more questions than necessary, since there exists a terminating state with $x_m + 1$ cost, thus, asking $> \alpha(x_m + 1)$ will not satisfy (a). Now assume that α*-`MultApprox` *has asked as many or more questions than* `Other` *until phase i, where α*-`MultApprox` *has asked y questions in total, while* `Other` *has asked (say) $y' \leq y$ questions in total. Now, let* `Other` *have asked more questions than α*-`MultApprox` *at the end of phase $i + 1$, i.e., α*-`MultApprox` *has asked z questions, while* `Other` *has asked $z' > z$ questions in total. Then, notice that from the state corresponding to y questions that α*-`MultApprox` *is currently at, in phase $i+1$, there is an input instance for which $x_m + 1$ questions results in a termination state, and $z = \alpha \times (x_m + y + 1)$. This same state is reachable from the state that* `Other` *is at, corresponding to y' questions. For this input instance, corresponding to $C_{opt} = x_m + y + 1$,* `Other` *asks a total of z' questions, where $z' > z = \alpha \times C_{opt}$. Thus,* `Other` *violates (a). Hence proved.*

5.3.3 Problem 5.2.4: Additive Approximation

The algorithm, called α-`AddApprox`, proceeds as follows: Let the current state of knowledge be (a, b), and the current phase be y. Then, the algorithm asks $x_m + \alpha + 1$ questions corresponding to the largest x_m satisfying equation 5.1. Thus, the algorithm asks up to α additional questions beyond what `OptCost` asks.

We have the following theorem, whose proof is similar to that of Theorem 5.3.4:

Theorem 5.3.6 *Algorithm α*-`AddApprox` *solves Problem 5.2.4.*

5.3.4 Discussion

The previous subsections gave us optimal online algorithms for the deterministic variant of the finding problem, given α, and whether we would like to use additive or multiplicative approximations. For a given value of α, the additive approximation algorithm uses the same "aggressiveness", i.e., it asks up to α additional questions in a given phase, independendent of how many questions have been asked so far. On the other hand, the multiplicative approximation algorithm becomes progressively more aggressive as more questions are asked. For instance, for $\alpha = 2$, if 3 questions are required to satisfy the output condition, the multiplicative algorithm asks up to 6 questions, but if 1000 questions are required to satisfy the output condition, the algorithm may ask up to 2000 questions.

An important hallmark of our algorithms is that as long as the output condition is not satisfied, we may switch from an additive to a multiplicative approximation or vice-versa. For instance, we may employ multiplicative approximation until a certain phase, and then switch to an additive approximation (thereby allowing the user of the algorithm to force the algorithm to be more conservative.) The reason why switching is possible is that we do not really "commit" to a specific approximation until the last phase when the output condition is eventually met. Until that phase, any questions used count towards satisfying the output condition, rather than being extra questions over and above the optimal number of questions. However, note that we may no longer have latency guarantees if we switch between strategies. The only guarantee we have is that the latency would be at most the sum of the latencies if we used only additive approximations, and if we used only multiplicative approximations.

5.4 Uncertain Human Answers

This section considers the case in which humans may make errors. We formally define the uncertainty setting in Section 5.4.1. Under the uncertainty model, even the best sequential algorithm is not obvious; we introduce the main sequential algorithm in Section 5.4.2. We also consider finding only Pass items in Section 5.4.2 (or only Fail items). We also show that adaptations of filtering to finding are arbitrarily worse than our algorithms. We present a brief description of extending to the case when we desire both Pass and Fail items in Section 5.4.3.

5.4.1 Model of Uncertainty

Setting: In the uncertain setting, as before, we have a single predicate or filter f, and each item may or may not satisfy f. In this case, however, on being asked a question on an item, the human response may not always be correct. As in Chapter 3, we assume that we know the selectivity s of f, as well as the error rates of the humans e_0, e_1, i.e., the false positive and false negative error rates of individual human responses (using prior history or by sampling a few items).

With each item receiving a set of x YES and y NO responses, we use a strategy $\mathcal{F}$ (as described in Chapter 3) to infer the Pass/Fail value for each item based on the set of human responses (x, y). Unlike in the deterministic setting, the strategy $\mathcal{F}$ may require x and y to both be greater than 1. Chapter 3 described many strategies (and algorithms to find optimal strategies) for inferring Pass/Failfor each item, and our techniques apply to all strategies. It remains to be seen if we can get significant improvements on using generalized filtering strategies from Chapter 3.

Our output condition is, as before, specified by a pair (k_1, k_0): Given a set $\mathcal{I}$ of items, our goal is to find $\mathcal{I}_1, \mathcal{I}_0 \subseteq \mathcal{I}$ such that: (1) all items in $\mathcal{I}_1$ are inferred to be Pass items, all items in $\mathcal{I}_0$ are inferred to be Fail items (based on a chosen strategy); (2) $|\mathcal{I}_1| \geq k_1$, $|\mathcal{I}_0| \geq k_0$.

The key ideas of our algorithms are best presented by assuming $k_0 = 0$; i.e., assuming we are only looking for Pass items. So for most of the rest of the section we assume $k_0 = 0$, and consider extension to $k_0 > 0$ in Section 5.4.3. Additionally, to begin with, we make the assumption that the number of items $n = |\mathcal{I}|$ is large (in comparison with k_0 or k_1); our algorithms generalize to the case when there is a bounded set of items. We describe this generalization in Section 5.4.5.

Expected Cost Computation: Given a set of x YES responses and y NO responses on an item I, we define $C_{next}(x, y)$ to be the *expected cost to finding the next item that satisfies the predicate*; i.e., $C_{next}(x, y)$ includes two cases:

- **Item I becomes a** Pass **item:** Let the probability of this event be PR_1. In this case, we want the expected number of questions required to declare I to be a Pass item; let this number of questions be denoted N_1.

- **Item I becomes worse than pursuing a new item:** Let the probability of this event be PR_2. In this case, we want $C_{next}(0, 0)$ plus the the number of questions required to make the cost higher than $C_{next}(0, 0)$. Let the number of questions required to make the cost higher than $C_{next}(0, 0)$ be denoted N_2.

The following expression combines the two cases above for $C_{next}(x, y)$:

$$C_{next}(x, y) = PR_1 \times N_1 + PR_2 \times (C_{next}(0, 0) + N_2)$$

Given a strategy, we describe how to compute C_{next} for each grid point in the strategy in Section 5.4.4.

Algorithm 2: Algorithm `OptSeq`, a cost-optimal sequential algorithm for the $(k_1, 0)$ uncertainty problem.

```
**Data**: $\mathcal{I}, k_1, s, e_0, e_1$, strategy $\mathcal{F}$
**Result**: Set $\mathcal{L}$ of $k_1$ Pass items
$\mathcal{L} \leftarrow \varnothing$;
$\mathcal{U} \leftarrow \mathcal{I}$;
Compute $C_{next}$ for each point in the strategy $\mathcal{F}$;
while $|\mathcal{L}| < k_1$ do
    Pick $I_j \in \mathcal{U}$ with lowest $C_{next}(x_j, y_j)$;
    Ask a question on $I_j$ to the crowdsourcing service;
    Add answer to $(x_j, y_j)$;
    if $\mathcal{F}(x_j, y_j)$ = Pass then
        Remove $I$ from $\mathcal{U}$ and add it to $\mathcal{L}$;
    if $\mathcal{F}(x_j, y_j)$ = Fail then
        Remove $I$ from $\mathcal{U}$;
```

5.4.2 Algorithms for $k_0 = 0$

We start by studying the case where we are only required to find k_1 Pass items, and $k_0 = 0$ Fail items; the converse case of $k_1 = 0$ is solved identically.

Problem 5.2.1: Sequential Algorithm

Consider the simple cost-optimal algorithm `OptSeq` shown in Algorithm 2 for any strategy. As the first step, we precompute $C_{next}(x, y)$ for each reachable point in the strategy. The algorithm simply picks an "undecided" item I_j (from set $\mathcal{U}$) with the lowest cost $C_{next}(R(I))$ at each phase and asks a human a question on that item. We may maintain a priority queue of unprocessed items and their cost, in order to pick the best candidate at each phase. Notice that all items have the same value $C_{next}(0, 0)$ to begin with, so we only need to sort the C_{next} values of the items for which at least one question has been asked, rather than all items in $\mathcal{U}$. Whenever an item is inferred to be a Pass or a Fail (according to the strategy $\mathcal{F}$), it is removed from consideration (i.e., removed from set $\mathcal{U}$).

Algorithm 3: Algorithm `UncOptCost`, a cost-optimal, phase-optimal algorithm for the $(k_1, 0)$ uncertainty problem.

```
Data: $\mathcal{I}, k_1, s, e_0, e_1$, strategy $\mathcal{F}$
Result: Set $\mathcal{L}$ of $k_1$ Pass items
$\mathcal{L} \leftarrow \emptyset$;
$\mathcal{U} = \mathcal{I}$;
Compute $C_{next}$ for each point in the strategy $\mathcal{F}$;
while $|\mathcal{L}| < k_1$ do
    Pick $\mathcal{I}' \subseteq \mathcal{U}, |\mathcal{I}'| = (k_1 - |\mathcal{L}|)$ items with the lowest $C_{next}(x, y)$;
    $\mathcal{Q} \leftarrow \{\}$;
    for each $I_j \in \mathcal{I}'$ do
        Let $a$ be the fewest YES responses required to make $C_{next}(x_j + a, y_j) = 0$;
        Let $b$ be the fewest NO responses required to make $C_{next}(x_j, y_j + b) = C_{next}(0, 0)$;
        Add $\min\{a, b\}$ questions on $I_j$ to $\mathcal{Q}$;
    Ask $\mathcal{Q}$ in parallel to the crowdsourcing service;
    Update $(x_j, y_j)$ for each $I_j \in \mathcal{Q}$ based on answers;
    for each $I_j \in \mathcal{Q}$ do
        if $\mathcal{F}(x_j, y_j)$ = Pass then
            Remove $I_j$ from $\mathcal{U}$ and add it to $\mathcal{L}$;
        if $\mathcal{F}(x_j, y_j)$ = Fail then
            Remove $I_j$ from $\mathcal{U}$;
```

Problem 5.2.2: Min-Cost Phase Optimal ($\alpha = 1$)

Next, we present a cost-optimal and phase-optimal algorithm `UncOptCost`, which uses an idea similar to Algorithm 2, but combines as many questions into a phase as will be definitely asked in Algorithm 2. (This enables us to give the guarantee of low number of phases while keeping the cost the same as Algorithm 2.) The pseudocode of `UncOptCost` is presented in Algorithm 3. Intuitively, the algorithm asks questions on at most k_1 items in each phase; these are the items with the lowest expected cost of a Pass decision based on the current number of YES and NO responses. To ensure that no unnecessary question is asked on any of these items, we ask the minimum number of questions that may: (a) either have the strategy $\mathcal{F}$ confirm the item as a Pass item, or (b) reduce the expected cost $C_{next}(x, y)$ below the next highest item, removing it from the set of top items being considered. Based on the criteria for the set of items and number of questions asked at each phase, we can show that `UncOptCost` is a min-cost phase-optimal algorithm.

Theorem 5.4.1 • `UncOptCost` *asks all and only questions asked by* `OptSeq`. *Therefore,* `UncOptCost` *is a min-cost solution to the* $(k_1, 0)$ *uncertainty problem.*

- `UncOptCost` *is phase-optimal.*

Proof 5.4.2 *(Sketch) For any algorithm that asks more questions than* `UncOptCost` *at any phase, it is easy to construct an input for which that algorithm asks strictly more questions than* `OptSeq`. □

Problem 5.2.3: α-cost approximation ($\alpha > 1$)

Next we consider α-multiplicative cost approximate algorithms. For any $\alpha > 1$, intuitively we are allowed to ask more questions in each phase than asked by `UncOptCost`. Broadly there are two ways to increase the number of questions we ask: We could expand the set of items and ask a similar number of questions on each item, or we could ask more questions on the set of items asked by `UncOptCost`. In the following, we consider both these approaches to asking more questions. For ease of presentation, we shall assume that α is an integer. In practice, all our algorithms can be extended to non-integer α; a trivial (and non-optimal) way to do this is by simply considering $\lfloor \alpha \rfloor$. Further, our ideas may be adapted for α-additive cost approximations, but details are omitted from this study.

Expand Set of Items Considered: We present Algorithm α`-Expand`, which is an α-cost approximate algorithm for the $(k_1, 0)$ uncertainty problem: Given an $\alpha > 1$, α`-Expand` runs α simultaneous instances of `UncOptCost` as follows. The first instance, a "master instance", of `UncOptCost` proceeds identically to the $\alpha = 1$ case, except that it keeps track of the total number of Pass items that have been found across all simultaneous instances of the algorithm. Therefore, every instance stops as soon as a total of k_1 Pass items have been found. Simultaneously, we have a set of $(\alpha - 1)$ "slave instances" that mimic the master instance: Each slave instance maintains a running set of items on which to ask questions at each stage. The total number of items asked in each stage is identical to the master instance, and there is a one-to-one correspondence in these sets of items: If the master instance asks a_i questions on the i'th item, even the slave instance asks exactly a_i items on the i'th item. At the end of each phase, the total number of Pass and Fail items are computed, and removed from the working set of items.

Our next theorem summarizes the result of the α`-Expand` algorithm. Intuitively, the guarantee that we aim to get is that each of the instances contribute $\frac{k_1}{\alpha}$ Pass items. If we were to run the slave instances independent of the master, this will allow us to get to k_1 Pass items in the same number of phases it takes for one slave to get $\frac{k_1}{\alpha}$ Pass items. However, since the slaves mimic the master exactly (in order to ensure that the algorithm is a α-multiplicative cost approximation), we multiply the number of phases by a "delaying factor", as can be seen in the following.

Theorem 5.4.3 *Assuming an infinite set of input items, where each item is drawn independently and uniformly at random from some distribution, let the expected number of phases required by Algorithm* `UncOptCost` *to find k_1 items by $ET_{\text{opt}}(k_1)$, and let the expected number of phases required by Algorithm α*`-Expand` *be $ET_{\alpha\text{-Expand}}(k_1)$. Suppose a^{max} is the maximum number of questions asked on any slave instance in one phase, and a^{min} is the minimum number of questions asked in any phase in the master instance, we have:*

- α`-Expand` *is an α-multiplicative cost approximation.*
- $ET_{\alpha\text{-Expand}}(k_1) \leq \min\{\frac{a^{max} ET_{\text{opt}}(\frac{k_1}{\alpha})}{a^{min}}, ET_{\text{opt}}(k_1)\}$

Proof 5.4.4 *Since α*`-Expand` *maintains a one-to-one correspondence between the set of items in the master instance, and items in each slave instance, the total number of questions asked is α times the number of questions asked in the master instance. Since the master instance mimics* `UncOptCost`*, we have that α*`-Expand` *is an α-multiplicative approximation.*

Let the random variable denoting the number of items obtained by α`-Expand` *after T phases be $X_\alpha(T)$. α*`-Expand` *runs α copies of* `UncOptCost`*, and let the random variable denoting number of items obtained by each of these instances be $X_i(T)$, $i = 1..\alpha$. We have:*

$$X_\alpha(T) = \sum_{i=1}^{\alpha} X_i(T)$$

Therefore:

$$E[X_\alpha(T)] = E[\sum_{i=1}^{\alpha} X_i(T)] = \sum_{i=1}^{\alpha} E[X_i(T)]$$

Let us set $T = ET_{\text{opt}}(\frac{k_1}{\alpha})$. Further, since each slave phase runs exactly the same number of questions as the master, for each item, each slave may require up to a multiplicative factor of $\frac{a^{max}}{a^{min}}$ more questions. Therefore, we have that:

$$E[X_\alpha(\frac{a^{max}}{a^{min}} ET_{\text{opt}}(\frac{k_1}{\alpha}))] \geq \sum_{i=1}^{\alpha} E[X_i(ET_{\text{opt}}(\frac{k_1}{\alpha}))]$$

$$= \sum_{i=1}^{\alpha} \frac{k_1}{\alpha} = k_1 = E[X_\alpha(ET_{\alpha\text{-Expand}}(k_1))]$$

$$\Rightarrow ET_{\alpha\text{-Expand}}(k_1) \leq \frac{a^{max}}{a^{min}} ET_{\text{opt}}(\frac{k_1}{\alpha})$$

□

Multiply Number of Questions: Our next approach, Algorithm α-`Multiply` proceeds by asking questions on exactly the same set of items as `UncOptCost`, but asking α-times as many questions on each item. If `UncOptCost` asked a_i questions in the i'th phase on a particular item I, then α-`Multiply` asks $\alpha \times a_i$ questions on I in the same phase. (Note that `UncOptCost` picks at most k_1 items based on probability order at the beginning of each phase; the ordering of items may be different for α-`Multiply`, but we still pick exactly the same set of items as `UncOptCost` to ensure α-cost approximation.)

Intuitively, our guarantee states that we may speed up the processing of items by a factor of α, while processing items in the same order. Once again, we need to add a "delaying factor" to account for the fact that we may end up asking extra unnecessary questions.

Theorem 5.4.5 *Given an input set $\mathcal{I}$ of items, and k_1, let T_{opt} be the number of phases required by* `UncOptCost`*; further, for each output item I_i ($i = 1..k_1$), let a^{max} and a^{min} be the maximum and minimum number of questions asked by* `UncOptCost` *in any phase. Let $\rho = \min\{\max_{i=1..k_1} \frac{a_i^{max}}{a_i^{min}}, \alpha\}$. We have that:*

- α-`Multiply` *is an α-multiplicative cost approximation.*
- α-`Multiply` *takes at most $\frac{\rho T_{\text{opt}}}{\alpha} + \frac{a^{max}}{a^{min}}$ phases to solve the $(k_1, 0)$ problem on $\mathcal{I}$.*

Proof 5.4.6 *Since each phase of α-*`Multiply` *asks at most α times as many questions as* `UncOptCost`*, the algorithm is an α-multiplicative approximation.*

Let T be the number of phases taken by α-`Multiply`*. Let $T_{\text{opt}}(i)$ be the number of phases for which I_i is active in* `UncOptCost`*, and let $T(i)$ be the number of phases I_i is active in α-*`Multiply`*. For each item I_i,* `UncOptCost` *asks at most $a_i^{max} T_{\text{opt}}(i)$ questions, and α-Multiply asks at least $\alpha T(i) a_i^{min}$ questions.*

Since the total number of questions required to resolve each item is independent of the specific algorithm, we have that the number of questions asked by α-Multiply is at most αa_i^{max} more than that asked by the sequential algorithm. Therefore, we have that

$$\alpha T(i) a_i^{min} \leq a_i^{max} T_{\text{opt}}(i) + \alpha a_i^{max}.$$

Therefore,

$$T(i) \leq \frac{\rho T_{\text{opt}}(i)}{\alpha} + \frac{a_i^{max}}{a_i^{min}}.$$

Since this holds for each item I_i, we have: $T \leq \frac{\rho T_{\text{opt}}}{\alpha} + \frac{a^{max}}{a^{min}}$

Comparison with Filtering

Next, we formally show that adaptations of filtering algorithms (as discussed in Chapter 3) to the finding problem have provably (a) much higher cost (b) much higher latency than the finding algorithms. For simplicity, we consider the case in which the output condition requires just k_1 items satisfying the predicate. Our discussion also applies to the general (k_1, k_0) variant.

One obvious way of adapting filtering to finding is to filter all $|\mathcal{I}| = n$ items simultaneously; naturally, this algorithm has arbitrarily high $O(n)$ cost compared to our algorithms, all of which use cost proportional to $O(k_1/s)$, independent of n.

The other way of using filtering for our problem is to filter in sequence; asking questions on one item until it is resolved to be a Pass or a Fail. Once an item is resolved, we operate on the next item, and so on, until k_1 items are resolved to Pass, thereby satisfying the output condition. We call this algorithm `OptSeqFilter`.

We first present a comparison of our sequential algorithm `OptSeq` with the sequential filtering algorithm `OptSeqFilter`. Note that the key difference between `OptSeq` and `OptSeqFilter` is that `OptSeq` may abandon an item even before it is resolved to a Fail or Pass (if there is another item with a better chance of being resolved to Pass, at a lower cost) — while the filtering algorithm always continues filtering until we resolve to a Fail or Pass. The following result establishes that: (1) The expected cost of `OptSeqFilter` is at least as much as `OptSeq`, (2) `OptSeqFilter` may incur arbitrarily higher cost compared to `OptSeq`.

Lemma 5.4.7

- *Given a set of items $\mathcal{I}$, k_1, and a filtering strategy $\mathcal{F}$, the expected cost of finding a solution using* `OptSeq` *is at most as much as the expected cost of finding a solution using* `OptSeqFilter`.
- *Given any $a > 0$, we can construct a finding problem instance such that the expected cost of finding a solution with* `OptSeqFilter` *is $\Omega(a)$ and that of finding a solution with* `OptSeq` *is $\mathcal{O}(1)$.*

Proof 5.4.8 *The proof of the first statement directly follows from the fact that* `OptSeq` *always picks an item with minimum expected cost to satisfy a predicate (recall the definition of $C_{next}(x, y)$ from*

Section 5.4.1).

We prove the second statement by constructing an input instance on which we can make the expected cost of `OptSeqFilter` *to be* $\Omega(a)$, *for any parameter* a, *but the expected cost of* `OptSeq` *is* $\mathcal{O}(1)$. *Consider constructing the* $(k_1 = 1, k_0 = 0)$ *problem over a set of two items* I_1, I_2, *with* I_1 *being a* Pass *item and* I_2 *being a* Fail *item. An instance of the problem orders items randomly, so either* I_1 *may be presented first, or* I_2. *Further, suppose that whenever a question is asked on a* Fail *item, a human always returns NO. And whenever a question is asked on a* Pass *item, with very high likelihood we get YES, but with a non-zero probability we get NO. Based on this, consider the strategy* $\mathcal{F}$ *where an item is declared to be* Pass *as soon as it gets one YES response, but an item is declared to be* Fail *only when the number of NO responses exceeds the number of YES responses by* M. *The expected cost of solving this input instance using* `OptSeq` *is* $\mathcal{O}(1)$ *since with probability half* I_1 *will be picked, which has an expected cost of* $\mathcal{O}(1)$ *before a YES response is obtained. (Even if* I_2 *is first in the order,* `OptSeq` *will switch to* I_1 *whenever the number of NO responses is more for* I_2, *still resulting in an* $\mathcal{O}(1)$ *expected cost.) On the other hand, for* `OptSeqFilter`, *with probability half* I_2 *will be first in the order, in which case the expected cost is at least* a, *therefore the expected cost of* `OptSeqFilter` *over the two possible orderings is also* $\Omega(a)$.

The previous lemma demonstrates that `OptSeqFilter` can have much higher expected cost than the cost optimal algorithm `OptSeq`. We next show how `OptSeqFilter` can have much higher expected latency as well as expected cost than our phase-optimal and cost-optimal algorithm `UncOptSeq`. Basically, the lemma states that our method of parallelizing as much as possible while retaining the same cost can result in significant gains in latency.

Lemma 5.4.9

- *Given a set of items* $\mathcal{I}$, k_1, *and a filtering strategy* $\mathcal{F}$, *the expected cost and latency of finding a solution using* `UncOptCost` *is at most as much as the expected cost and latency of finding a solution using* `OptSeqFilter`.
- *We can construct a finding problem with input instance such that the expected latency of finding a solution with* `OptSeqFilter` *is* $\Omega(k_1)$, *and that of finding a solution with* `UncOptCost` *is* $\mathcal{O}(1)$.

Proof 5.4.10 *The first statement follows directly from the fact that* `UncOptCost` *asks precisely as many questions as* `OptCost`, *and* `OptSeqFilter` *has higher cost than* `OptCost` *(as seen from Lemma 5.4.7). Moreover,* `OptSeqFilter` *examines items one at a time, resulting in a higher latency than* `UncOptCost`, *which parallelizes questions as much as possible.*

We prove the second statement by constructing an instance of the problem on which we can make the expected latency of `UncOptCost` *be $O(1)$. We use the majority strategy where one question determines whether or not the item satisfies the filter, and we set s to be arbitrarily high. In this case,* `UncOptCost` *will ask k_1 items one question in one phase, and with high probability, will find k_1 items satisfying the predicate. On the other hand,* `OptSeqFilter` *will take at least $\Omega(k_1)$ to find k_1 items satisfying the predicate.*

5.4.3 Pass and Fail Items

So far we only considered algorithms where the goal is to find k_1 Pass items and zero Fail items. We show that for the general (k_1, k_0) problem of finding k_1 Pass items and k_0 Fail items, there is a simple 2-approximation algorithm in the number of phases. Consider Algorithm `2Step` that operates as follows: (1) In the first step it solves the $(k_1, 0)$ problem using techniques destribed above, (2) In the second step it solves the $(0, k_0)$ problem, again using techniques described above, then combines the items obtained. It can be seen easily that `2Step` is a 2-approximation to the latency of an optimal α-approximation algorithm $\mathcal{A}^*$.

Lemma 5.4.11 *We are given an input instance $\mathcal{I}$ for the (k_1, k_0) problem, and input α denoting the required cost-approximation. Let $\mathcal{A}^*$ be an optimal α-cost multiplicative approximation algorithm for an output condition that can be expressed as (k_1, k_0). We have that* `2Step` *that applies the optimal α-cost approximation techniques on $(k_1, 0)$ and $(0, k_0)$ problems respectively is a 2-approximation of $\mathcal{A}^*$.*

Proof 5.4.12 *Let $T(A)$ be the number of phases required by algorithm $\mathcal{A}$. Let the two stages of* `2Step` *be $\mathcal{A}$ and $\mathcal{A}$. Since $\mathcal{A}$ and $\mathcal{A}$ are phase-optimal for the $(k_1, 0)$ and $(0, k_0)$ problems respectively, we have that:*

$$T(\mathcal{A}^*(k_1, k_0)) \geq T(\mathcal{A}^*(k_1, 0)) \geq T(\mathcal{A}_1(k_1, 0))$$

$$T(\mathcal{A}^*(k_1, k_0)) \geq T(\mathcal{A}^*(0, k_0)) \geq T(\mathcal{A}_2(0, k_0))$$

Adding the two equations gives our result:

$$T(\texttt{2Step}(k_1, k_0)) = T(A_1(k_1, 0)) + T(A_2(0, k_0)) \leq 2T(\mathcal{A}^*(k_1, k_0))$$

Note that the 2-approximation above applies to either a min-cost phase optimal algorithm $\mathcal{A}^*$, or to any α-cost phase optimal algorithm. Effectively, given any algorithm $\mathcal{A}^*$, we have that `2Step` achieves the same result as $\mathcal{A}^*$ in at most twice the cost.

5.4.4 Cost Computation

We now discuss how to compute the expected cost $C_{next}(x, y)$ given a strategy. As in Chapter 3, a strategy can be represented as a *closed* region in the X-Y plane, i.e., there is a certain value m such that no item ever reaches $x + y = m$ (where x, y are the number of YES/NO answers from humans) during processing. Note that for most commonly used strategies, even a very small m (say $m = 5$) will suffice.

Given a strategy, there is a recursive dynamic programming algorithm that computes $C_{next}(x, y)$ for all points within the strategy, starting at the boundary of the strategy, and proceeding towards $(0, 0)$. Consider a point (x, y). Let the probability of getting a YES answer at (x, y) be $p_{\text{YES}}(x, y)$ and the probability of getting a negative answer at (x, y) be $p_{\text{NO}}(x, y)$. We have:

$$p_{\text{NO}}(x, y) = \Pr(V = 1|(x, y)) \times \Pr(\text{NO}|V = 1) + \Pr(V = 0|(x, y)) \times \Pr(\text{NO}|V = 0),$$

where $V = 0/1$ is a random variable indicating whether the item satisfies the predicate/filter or not. (There is a similar expression for $p_{\text{YES}}(x, y)$.) These expressions do not depend on the strategy, but instead depend on x and y, the selectivity of the predicate, and the error rates of human workers which depend on the item being $V = 0/1$.

At each point (x, y), we either have the option of starting afresh (i.e., starting with a new item), or proceeding with the current item by asking an additional question. If we ask an additional question and end up at $(x, y + 1)$, then our expected cost from that point on is $C_{next}(x, y + 1)$, while if we end up at $(x + 1, y)$, our expected cost from that point on is $C_{next}(x + 1, y)$. We have the following:

$$C_{next}(x, y) = \min \begin{cases} C_{next}(0, 0) \\ p_{\text{YES}}(x, y) C_{next}(x + 1, y) + p_{\text{NO}}(x, y) C_{next}(x, y + 1) + 1 \end{cases}$$

If the first case is used, then we return to $(0, 0)$ (and start with a new item) if we reach (x, y), and in the second case, we ask an additional question. Our corner cases are the following: At all points of the strategy (x, y) where Fail is returned, we set $C_{next}(x, y) = C_{next}(0, 0)$, while at all points of the strategy (x, y) at which Pass is returned, we set $C_{next}(x, y) = 0$. To compute $C_{next}(x, y)$ for any pair of values x, y, we recursively unfold the equation above, until the base case of $(0, 0)$. Note, however, that the value of $C_{next}(0, 0)$ is unknown, thus the expression keeps increasing in size as we move towards the origin. More precisely, the size of the expression could be as large as the number of points in the strategy (or exponential in m).

However, we may use the following theorem to guide our search for the right value of $C_{next}(0,0)$:

Theorem 5.4.13 *The values of $C_{next}(x,y)$ for all x,y (not both 0) monotonically decreases as $C_{next}(0,0)$ decreases. Furthermore, if we substitute α for $C_{next}(0,0)$ and compute $C_{next}(x,y)$ for all points (x,y) (not both 0), then $C_{next}(0,0) \geq \alpha$ iff*

$$\alpha \leq p_{\text{YES}}(0,0)C_{next}(1,0) + p_{\text{NO}}(0,0)C_{next}(0,1) + 1$$

(and vice versa.)

Thus, our approximate decision procedure is the following: We use a binary search for α between 0 and m. We start with a value of $C_{next}(0,0) = \alpha$, and use the equations above to compute the values of $C_{next}(x,y)$ for all points where x,y are not both zero, then compare α and $p_{\text{YES}}(0,0)C_{next}(1,0) + p_{\text{NO}}(0,0)C_{next}(0,1) + 1$. If α is larger, then $C_{next}(0,0) < \alpha$ (and so α must be reduced), while if α is smaller, then $C_{next}(0,0) > \alpha$ (thus α must be increased.) We stop when $C_{next}(0,0) \approx \alpha$. Each iteration of the search procedure would take $O(m^2)$.

5.4.5 Discussion

So far, we assumed that $\mathcal{I}$ has an infinite number of items. Our algorithms continue to have the same worst-case guarantees as the infinite case, when $\mathcal{I}$ has a bounded number of items, as long as we can satisfy the output condition without examining all items (which is certainly true when $|\mathcal{I}|$ is much larger than k_1 or k_0.)

Once we end up examining all items, then our main issue is in the cost computation, where $C_{next}(0,0)$ is no longer the expected cost of a new item satisfying the predicate. (Specifically, $C_{next}(0,0)$ will be larger, and as a result, C_{next} values for all items will increase.) We run our algorithms as before until we examine all items. At that point, we set $C_{next}(0,0)$ to be ∞, and recompute $C_{next}(x,y)$. (This recomputation has the effect of setting C_{next} for each item to be the cost of confirming that item to be a Pass or a Fail item.) Then, we continue execution of our algorithms. While approximate, this approach provides a practical solution when we end up examining all items.

In this section, we developed two approximation algorithms (one with expanding the set of items under consideration α-`Expand`, and one with a look-ahead for the same set of items α-`Multiply`). One could envision designing a hybrid algorithm that does both expansion as well as lookahead;

we leave deriving bounds on performance of such an algorithm as future work. However, we may certainly use the worst case bounds of the two algorithms to select, for a given finding problem, the algorithm which has better bounds and therefore will perform better.

Lastly, while we considered multiplicative approximation in this section, one could imagine an analog of the α-`Multiply` algorithm for additive approximation. We omit details of the derivation of performance bounds for such an algorithm.

5.5 Problem 5: Expected Cost and Latency

In this chapter, so far, we focused on problems that provide *instance-specific guarantees*, i.e., we designed algorithms with approximation guarantees *for each instance*, relative to the sequential algorithm for the same input instance. In this section, we derive *expected* cost and latency across all input instances, for the algorithms that we devised. Deriving such guarantees will enable us to directly compare algorithms on cost and time.

We make two simplifying assumptions in this section: we assume that our output condition asks for k_1 items that satisfy the filter, and that humans do not make mistakes.

We begin by introducing some notation: To distinguish the expected cost and latency from the instance-specific cost and latency, we denote the expected monetary cost of an algorithm as EC and the expected latency or expected time as ET. As before n denotes the total number of items $|\mathcal{I}|$. Next we analyze EC and ET of completely sequential, completely parallel, and intermediate algorithms.

Sequential and Parallel Extremes: As in Section 5.3, in the sequential case, we can ask one question at a time, and stop once we have k_1 items satisfying the predicate. This approach has $EC = ET = \frac{k_1}{s}$ (recall that s is the selectivity of the filter), since we examine k_1/s items before we find k_1 that satisfy the predicate.

The straightforward parallel algorithm that asks all items in a single phase is guaranteed to find k_1 items that satisfy the predicate. This approach has $EC = n, ET = 1$.

Intermediate Solutions: We may now design algorithms to "parallelize" the sequential algorithm and potentially speed it up. For instance, we simply ask k_1 questions in the first phase, ask only as many as necessary in the second phase (i.e., only as many items as strictly necessary to satisfy the output condition), and so on. To analyze algorithms of this type, let us first assume that when we ask questions, we get precisely what we expect. Subsequently, we will show how to ensure that with high probability we get what we expect.

In this setting, if we ask k_1 questions in the first phase, the number of items that satisfy the predicate would be precisely sk_1. Then, we may ask $k_1 - sk_1$ in the next phase, and we get $s(k_1 - sk_1)$ items that satisfy the predicate in the second phase. In the ith phase, generalizing this computation, it can be shown that we ask $k_1(1-s)^{i-1}$ questions. We would like this number to be < 1, which guarantees that we have found enough items. That is,

$$i > 1 - \frac{\log k_1}{\log(1-s)}$$

Moreover, the total number of questions asked can be computed as follows:

$$\sum_{j=1}^{j=i} k_1(1-s)^{j-1} = k_1 \frac{1-(1-s)^i}{s}$$

We can generalize the algorithm above by asking α times the remaining number of items required at each phase. That is, we begin by asking αk_1 questions, then, if $k_1' \leq k_1$ items are still to be found by the second phase, we ask $\alpha k_1'$, and so on. Using similar ideas to the computation in the previous algorithm, we have:

$$i > 1 - \frac{\log(\alpha k_1)}{\log(1-\alpha s)}$$

Moreover, the total number of questions asked can be computed as follows:

$$\sum_{j=1}^{j=i} \alpha k_1(1-\alpha s)^{j-1} = k_1 \frac{1-(1-\alpha s)^i}{s}$$

However, note that in a given phase, we may not exactly obtain the number of items that we expect to satisfy the predicate. To ensure that with high probability we get *at least* the number of items that we expect in each round, we scale up the number of questions asked at each phase by a factor β, and we are guaranteed to get expected monetary cost and latency lower than those computed above:

$$ET \leq 2 - \frac{\log(\alpha k_1)}{\log(1-\alpha s)} \quad EC \leq \beta k_1 \frac{1-(1-\alpha s)^i}{s}$$

The following theorem formalizes this result.

Theorem 5.5.1 *Given the set $\mathcal{I}$ of items, each with an independent probability s of satisfying the input predicate, and a required number k_1 of items to be found, the parallel algorithm of asking $\beta\alpha k_i$ items in the ith phase ($\alpha \leq 1/s$), where k_i is the remaining number of items to be found has the following* *guarantees with* $\beta = \lceil 2 + \frac{a^2(1-s)}{k_1} \rceil$*:*

$$\Pr(C \geq \beta k_1 \frac{1-(1-\alpha s)^i}{s}) \leq \frac{T}{a^2}$$

where T *is the actual number of phases, whose expectation is:*

$$ET \leq 2 - \frac{\log(\alpha k_1)}{\log(1-\alpha s)}$$

Proof 5.5.2 *Suppose the number of questions asked in some* i*th phase is* βq*. Then the expected number of items obtained are* $s\beta q$*, and the variance is* $s(1-s)\beta q$*. Therefore, by application of Chebyshev's inequality, we obtain the following probabilistic bound on the number of items* X_i *that are returned:*

$$\Pr(X_i \leq (s\beta q - a\sqrt{s(1-s)\beta q})) \leq \frac{1}{a^2}$$

To obtain at least sq *items, we require*

$$qs\beta - a\sqrt{s(1-s)\beta q}) \geq sq$$
$$\frac{\beta - 1}{\sqrt{\beta}} \geq \frac{a\sqrt{(1-s)}}{\sqrt{qs}}$$

Since the most number of items asked will be in the first phase, we conservatively substitute $q = \frac{k_1}{s}$*, since* $\alpha \leq \frac{1}{s}$*, obtaining:*

$$\sqrt{\beta} - \frac{1}{\sqrt{\beta}} \geq a\sqrt{\frac{(1-s)}{k_1}}$$

Squaring the equation and simplifying gives us:

$$\beta \geq 2 + \frac{a^2(1-s)}{k_1}$$

Finally, the result above gives a bound for a single phase. Suppose the algorithm is applied for T *phases, using the union bound, we obtain that the probability of obtaining required number of items in each phase is at least* $\frac{T}{a^2}$*, giving us the desired result.*

2D Plane of Algorithms: In figure 5.2, we depict the sequential and parallel extremes, along with the intermediate solutions on varying α for a scenario where $n = 500, k_1 = 30, s = 0.3, m^2 = 100$, and

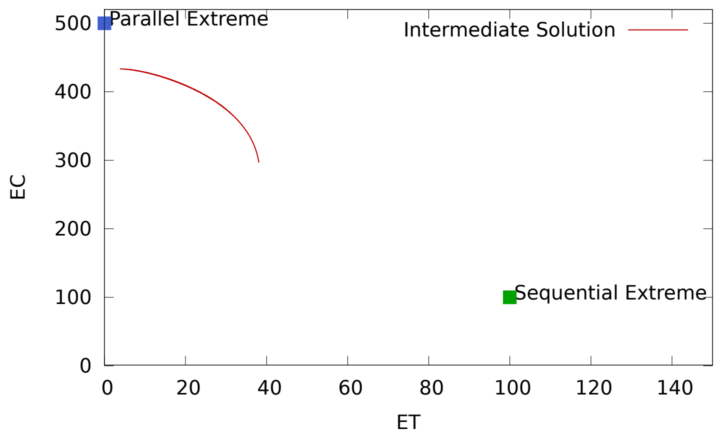


Figure 5.2: Comparing algorithms on ET and EC for $k_1 = 30, s = 0.3, N = 500$

$\beta = \lceil 2 + \frac{m^2(1-s)}{k_1} \rceil$. As can be seen in the figure, the intermediate solutions provide valuable alternatives to the sequential or parallel extremes (with at least one of EC or ET lower).

5.6 Related Work

We now briefly describe work related to the finding problem.

In [192], the goal is to use humans to assist in finding one image relevant to an image search query, which maps to an instance of our problem. The work develops a machine-learned model of the delay and accuracy of human workers, and uses it to heuristically determine whether to pursue the given image. Our work is more general in that it considers not only a broader class of problems, but also develops algorithms to determine all solutions that lie on the skyline of cost and latency.

Our work (especially when humans are assumed to not make mistakes) is also related to the vast field of parallel computation [42, 64, 114, 179] wherein several models of parallel computation have been proposed, including PRAMs [114], Bulk-Synchronous Parallel processing [179] and LogP [64], and more recently, ones based on MapReduce [27, 113, 121]. There are a couple of key differences between this field and our work: First, this field typically assumes a fixed number of processors operating in parallel (in each phase)–while there are practical implementations that could vary the number of processors. On the other hand, we can dynamically vary the number of humans working on our tasks at any point. There is direct no notion of monetary cost in their setting, while in our setting the total number of human operations or questions is the total cost. Second, the main consideration in this field is to model and understand the tradeoffs between local computation and data storage at each processor and communication between processors. In our setting we have a central coordinator which does the computation between phases (assumed negligible in comparison with the latency of crowdsourcing), and leveraging human processors on demand to do simple tasks (Thus, the human processors don't communicate with each other.) Of course, parallel computation has no counterparts for the case when humans make mistakes.

In our work, we make the simplifying assumption that humans are equally likely to make errors, like in Chapter 3. Recent work on crowdsourcing has tried to identify which workers to ask which questions [81, 188], as well as learning characteristics of workers while asking questions [74, 112, 131, 189]. It remains to be seen if fine-grained error models improve the performance of finding algorithms.

5.7 Conclusions

In this chapter, we studied the fundamental crowd-powered finding problem, relevant in many crowdsourcing applications. We developed optimal solutions that lies on the skyline of cost and latency for two settings: when humans answer correctly, and when they may make errors. Unlike Chapter 3, here, we focused primarily on instance-optimal guarantees. The reason why instance optimal guarantees make more sense in the finding problem is because the actual cost and error depends a lot on the order in which the items are considered: for instance, if the items are so ordered that a lot of "unhelpful" items are at the start, then our cost and latency both suffer. Therefore, we would like to offer guarantees that hold irrespective of the order in which the items are received by the algorithm.

In the next chapter, we move to the crowd-powered maximum problem, wherein we look at questions asked to humans that consider pairs of items.

Chapter 6

Algorithm 3: Maximum

6.1 Introduction

In this chapter, we design algorithms for the crowd-powered *Max (Maximum)* problem[1]. We have a set of items (e.g., maps, photographs, Facebook profiles), where conceptually each item has an intrinsic "quality" measure (e.g., how useful is a map for a specific humanitarian mission, how well does a photo describe a given restaurant, how likely is it that a given Facebook profile is the actual profile of Lady Gaga). Of the set of items, we want to find the one with the largest quality measure. While there are many possible underlying types of human questions that may be used for the max problem, in this chapter we focus on a pairwise question: a human is asked to compare two items and returns the one item he/she thinks is of higher quality. We call this type of pairwise comparison a *vote*. Unlike the previous chapters, where there are at most $|\mathcal{I}| = n$ distinct questions (one question for each item), here there are $\binom{n}{2}$ distinct questions, corresponding to a pairwise comparison or vote for every pair of items.

If we ask two humans to compare the same pair of items they may give us different answers, either because they makes mistakes or because their notion of quality is subjective. Either way, the algorithm may need to submit the same vote to multiple humans to increase the likelihood that its final answer is correct (i.e. that the reported max is indeed the item with the highest quality measure). Of course, executing more votes increases the cost of the algorithm, either in running time and/or in monetary compensation given to the humans for their work.

There are two types of algorithms for the Max Problem: structured and unstructured. With a

[1]This chapter is adapted from our paper [92], published at SIGMOD 2012, written jointly with Stephen Guo and Hector Garcia-Molina.

structured approach, a regular pattern of votes is set up in advance, as in a tournament. For example, if we have 8 items to consider, we can first compare 1 to 2, 3 to 4, 5 to 6 and 7 to 8. After we get all results, we compare the 1-2 winner to the 3-4 winner and the 5-6 winner to the 7-8 winner. In the third stage, we compare the two winners to obtain the overall winner, which is declared the max. If we are concerned about voting errors, we can repeat each vote an odd number of times and use the consensus result. For instance, three humans can be asked to do the 1-2 comparison, and the winner of this comparison is the item that wins in 2 or 3 of the individual votes.

While structured approaches are very effective in predictable environments (such as in a sports tournament), they are much harder to implement in a crowd-powered system, where humans may simply not respond to a vote, or may take an unacceptably long time to respond. In our 8-item example, for instance, after asking for the first 4 votes, and waiting for 10 minutes, we may have only the answers to the 1-2 and 5-6 comparisons. We could then re-issue the 3-4 and 7-8 comparisons and just wait, but perhaps we should also try comparing the winner of 1-2 with the winner of 5-6 (which was not in our original plan).

The point is that even if we start with a structured plan in mind, because of incomplete votes we will likely be faced with an unstructured scenario: some subset of the possible votes have completed (some with varying numbers of repetitions), and we have to answer one or both of the following questions:

- *Judgment Problem*: what is our current best estimate for the overall max winner?
- *Next Votes Problem*: if we want to invoke more votes, which are the most effective ones to invoke, given the current standing of results?

In this chapter, we focus precisely on these two problems, in an unstructured setting that is much more likely to occur in a crowd-powered system or application. Both of these problems are quite challenging because there may be many items in the data set, and because there are many possible votes to invoke. An additional challenge is contradictory evidence. For instance, say we have three items, and one vote told us 1 had higher quality than 2, another vote told us that 2 had higher quality than 3, and a third one told us that 3 had higher quality than 1. What is the most likely max in a scenario like this one where evidence is in conflict? Should we just ignore "conflicting" evidence, but how exactly do we do this? Yet another challenge is the lack of evidence for some items. For example, say our evidence is that 1 is of higher quality than items 2, 3 and 4. However, there are two additional items, 5 and 6, for which there is no data. If we can invoke one more vote, should we compare the current favorite, item 1, against another item to verify that it is the max, or should we at least try comparing 5 and 6, for which we have no information?

The Judgment Problem draws its roots from the historical *paired comparisons* problem, wherein the goal is to find the best ranking of items when noisy evidence is provided [70,116,172]. The problem is also related to the *Winner Determination* problem in the economic and social choice literature [57], wherein the goal is to find the best item via a *voting rule*: either by finding a "good" ranking of items and then returning the best item(s) in that ranking, or by scoring each item and returning the best scoring item(s). As we will see in Section 6.2, our solution to the Judgment Problem differs from both of these approaches. As far as we know, no counterpart of the Next Votes problem exists in the literature. (The reason, we believe, is that before crowdsourcing, say in sporting events and user evaluations, votes had to be set up in advance, without the possibility of dynamically issuing new votes as results came in.) We survey work related to this chapter in more detail in Section 6.4.

6.1.1 Outline of Chapter

Here is the outline for the rest of the chapter:

- We study the following aspects of the Judgment problem (Section 6.2):
  - We describe the Judgment problem formally. (Section 6.2.1)
  - We propose a Maximum Likelihood (ML) formulation of the Judgment Problem, which finds the item that is probabilistically the most likely to be the maximum. (Section 6.2.2)
  - We show that computing the Maximum Likelihood item is NP-Hard, while computation of the probabilities involved is #P-Hard. (Section 6.2.3)
  - We propose and evaluate four different heuristics for the Judgment Problem, some of which are adapted from solutions for sorting with noisy comparisons. (Section 6.2.4)
  - For small problem settings, we compare the heuristic solutions to those provided by ML. When there is only a small number of votes available, we show that one of our methods, a novel algorithm based on PageRank, is the best heuristic. (Section 6.2.5)

  To the best of our knowledge, our ML formulation provides the first formal definition and analysis of the Judgment Problem.

- We study the following aspects of the Next Votes Problem (Section 6.3)

– We provide the first formal definition of the Next Votes Problem, and again, propose a formulation based on ML. (Section 6.3.1)

– We show that selecting optimal additional votes is NP-Hard, while computation of the probabilities involved is #P-Hard. (Section 6.3.2)

– We propose four novel heuristics for the Next Votes Problem. We experimentally evaluate the heuristics, and when feasible, compare them to the ML formulation. (Section 6.3.3)

6.2 Judgment Problem

6.2.1 Problem Setup

Items and Permutations: We are given a set $\mathcal{I}$ of n items $\{I_1, ..., I_n\}$, where each item I_i is associated with a latent *quality* c_i, with no two c's being the same. If $c_i > c_j$, we say that I_i is *greater* than I_j. Let π denote a permutation function, e.g., a bijection from N to N, where $N = \{1, ..., n\}$. We use $\pi(i)$ to denote the *rank*, or index, of item I_i in permutation π, and $\pi^{-1}(i)$ to denote the item index of the ith position in permutation π. If $\pi(i) < \pi(j)$, we say that I_i is ranked *higher* than I_j in permutation π. Since no two items have the same quality, there exists a *true* permutation π^* such that for any pair (i, j), if $\pi^*(i) < \pi^*(j)$, then $c_{\pi^*(i)} > c_{\pi^*(j)}$. Note that throughout this chapter, we use the terms *permutation* and *ranking* interchangeably.

Voting: We wish to develop an *algorithm* to find the maximum (greatest) item in set $\mathcal{I}$, i.e., to find $\pi^{*-1}(1)$. The only type of operation or information available to an algorithm is a pairwise *vote*: in a vote, a human worker is shown two items I_i and I_j, and is asked to indicate the greater item. We assume that every worker votes correctly with probability $1 - e$, $(0 \le e \le 0.5)$, where e is the average worker error probability. Like in Chapter 3 and Chapter 5, we assume that each worker answers questions (here, votes) independently and with the same error probability. Recall that in Chapter 3 and Chapter 5, we had two types of error probabilities — e_0, e_1, denoting false positive and false negative errors. Here, we have a single error probability e that is unaffected by the pair of items being compared. (We are, however, studying a simpler setting than Chapter 4, since we do not take individual worker accuracies into account.) As a result, each vote can be viewed as an independent Bernoulli trial with failure probability e. In general, the value e is not available to the algorithm, but may be used for algorithm evaluation. However, for reference we do study two algorithms where e is known (possibly by using sampling on a few items or by using prior history).

Goals: No matter how the algorithm decides to issue vote requests to workers, at the end it must

$$W = \begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

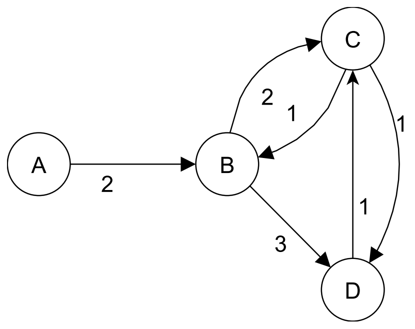


Figure 6.1: How should these items be ranked? Vote matrix (left) and equivalent graph representation (right). Arc weights indicate number of votes.

select what it thinks is the maximum item based on the evidence, i.e., based on the votes completed so far. We start by focusing on this *Judgment Problem*, which we define as follows:

Problem 6.2.1 (Judgment) *Given W, predict the maximum item in $\mathcal{I}$, $\pi^{*-1}(1)$.*

In Section 6.3, we then address the other important problem, i.e., how to request additional votes (in case the algorithm decides it is not done yet). In general, a solution to the Judgment Problem is based upon a *scoring function* $score(\cdot)$. The scoring function first computes a *score* for each item I_i, with $score(i)$ representing the "confidence" that item I_i is the true maximum. As we will see, for some strategies scores are actual probabilities, for others they are heuristic estimates. Then, the strategy selects the item with the largest score as its answer.

Representation: We represent the evidence obtained as an $n \times n$ vote matrix W, with w_{ij} being the number of votes for I_j being greater than I_i. Note that $w_{ii} = 0$ for all i. No other assumptions are made about the structure of matrix W. The evidence can also be viewed as a directed weighted graph $G_v = (V, A)$, with the vertices being the items and the arcs representing the vote outcomes. For each pair (i, j), if $w_{ij} > 0$, arc (i, j) with weight w_{ij} is present in A. For example, Figure 6.1 displays a sample vote matrix and equivalent graph representation. In this example, item 1 is called A, item 2 is B, and so on. For instance, there is an arc from vertex (item) B to vertex C with weight 2 because $w_{2,3} = 2$, and there is a reverse arc from C to B with weight 1 because $w_{3,2} = 1$. If there are no votes ($w_{ij} = 0$, as from B to A), we can either say that there is no arc, or that the arc has weight 0.

6.2.2 Maximum Likelihood

Preliminaries: We first present a Maximum Likelihood (ML) formulation of the Judgment Problem. We directly compute the item that has the highest probability of being the maximum item in $\mathcal{I}$, given vote matrix W. Assuming that average worker error probability e is known, the ML formulation we present is the optimal feasible solution to the Judgment Problem.

Let $\boldsymbol{\pi}$ be a random variable over the set of all $n!$ possible permutations, where we assume a-priori that each permutation is equally likely to be observed. We denote the probability of a given permutation π_d given the vote matrix W as $\Pr(\boldsymbol{\pi} = \pi_d|W)$. For the ease of exposition, we adopt the shorthand $\Pr(\pi_d|W)$ instead of writing $\Pr(\boldsymbol{\pi} = \pi_d|W)$. To derive the formula for $\Pr(\pi_d|W)$, we first apply Bayes' theorem,

$$\Pr(\pi_d|W) = \frac{\Pr(W|\pi_d)\Pr(\pi_d)}{\Pr(W)} = \frac{\Pr(W|\pi_d)\Pr(\pi_d)}{\sum_j \Pr(W|\pi_j)\Pr(\pi_j)} \tag{6.1}$$

From our assumption that the prior probabilities of all permutations are equal, $\Pr(\pi_d) = \frac{1}{n!}$.

Now consider $\Pr(W|\pi_d)$. Given a permutation π_d, for each unordered pair $\{i, j\}$, the probability $p_{\pi_d}(i, j)$ of observing w_{ij} and w_{ji} is the binomial distribution probability mass function (p.m.f.):

$$p_{\pi_d}(i,j) = \begin{cases} (1-e)^{w_{ji}} e^{w_{ij}} & \text{if } \pi_d(i) < \pi_d(j) \\ (1-e)^{w_{ij}} e^{w_{ji}} & \text{if } \pi_d(j) < \pi_d(i) \end{cases}$$

If both w_{ij} and w_{ji} are equal to 0, then $p_{\pi_d}(i, j) = 1$. Now, given a permutation π_d, observing the votes involving an unordered pair $\{i, j\}$ is conditionally independent of observing the votes involving any other unordered pair. Using this fact, $\Pr(W|\pi_d)$, the probability of observing all votes given a permutation π_d is simply:

$$\Pr(W|\pi_d) = \prod_{i,j:i<j} p_{\pi_d}(i,j) \tag{6.2}$$

Since we know the values of both e and W, we can derive a formula for $\Pr(\pi_d|W)$ in Equation 6.1. The most likely permutation(s), is simply:

$$\arg\max_d \Pr(\pi_d|W) \tag{6.3}$$

The permutations optimizing Equation 6.3 are also known as *Kemeny permutations* or *Kemeny rankings* [60].

For example, consider the matrix W of Figure 6.1. We do not show the computations here, but it turns out that the two most probable permutations of the items are (D, C, B, A) and (C, D, B, A), with all other permutations having lower probability. This result roughly matches our intuition, since item A was never voted to be greater than any of the other items, and C and D have more votes in favor over B.

We can derive the formula for the probability that a given item I_j has a given rank k. Let $\boldsymbol{\pi}^{-1}(i)$ denote the position of item i in the permutation associated with random variable $\boldsymbol{\pi}$. We are interested in the probability $\Pr(\boldsymbol{\pi}^{-1}(k) = j|W)$. Since the event $(\boldsymbol{\pi} = \pi_d)$ is disjoint for different permutations π_d, we have:

$$\Pr(\boldsymbol{\pi}^{-1}(k) = j|W) = \sum_{d:\pi_d^{-1}(k)=j} \Pr(\pi_d|W)$$

Substituting for $\Pr(\pi_d|W)$ using Equation 6.1 and simplifying, we have:

$$\Pr(\boldsymbol{\pi}^{-1}(k) = j|W) = \frac{\sum\limits_{d:\pi_d^{-1}(k)=j} \Pr(W|\pi_d)}{\sum\limits_{l} \Pr(W|\pi_l)} \tag{6.4}$$

Since we are interested in the item I_j with the highest probability of being rank 1, e.g., $\Pr(\boldsymbol{\pi}^{-1}(1) = j|W)$, we now have the Maximum Likelihood formulation to the Judgment Problem:

ML Formulation 1 (Judgment) *Given W and e, determine:* $\arg\max_j \Pr(\boldsymbol{\pi}^{-1}(1) = j|W)$.

In the example graph of Figure 6.1, while C and D both have Kemeny permutations where they are the greatest items, D is the more likely max over a large range of e values. For instance, for $e = 0.25$, $\Pr(\boldsymbol{\pi}^{-1}(1) = C|W) = 0.36$ while $\Pr(\boldsymbol{\pi}^{-1}(1) = D|W) = 0.54$. This also matches our intuition, since C has one vote where it is less than B, while D is never voted to be less than either A or B.

Maximum Likelihood Strategy: Equation 6.4 implies that we only need to compute $\Pr(W|\pi_d)$ for each possible permutation π_d, using Equation 6.2, in order to determine $\Pr(\boldsymbol{\pi}^{-1}(k) = j|W)$ for all values j and k. In other words, by doing a single pass through all permutations, we can compute the probability that any item I_j has a rank k, given the vote matrix W.

We call this exhaustive computation of probabilities the *Maximum Likelihood Strategy* and use it as a baseline in our experiments. Note that the ML strategy is the optimal feasible solution to the

Judgment Problem. The strategy utilizes a ML scoring function, which computes the score of each item as $score(j) = \Pr(\pi^{-1}(1) = j|W)$. The predicted max is then the item with the highest score. The strategy can be easily adapted to compute the maximum likelihood of arbitrary ranks (not just first) over the set of all possible permutations.

6.2.3 Computational Complexity

Hardness of the Judgment Problem: In Section 6.2.2, we presented a formulation for the Judgment Problem based on ML for finding the item most likely to be the max (maximum) item in $\mathcal{I}$. Unfortunately, the strategy based on that formulation was computationally infeasible, as it required computation across all $n!$ permutations of the items in $\mathcal{I}$. We now show that the optimal solution to the problem of finding the maximum item is in fact NP-Hard using a reduction from the problem of *determining Kemeny winners* [100]. (Hudry et al. [100] actually show that *determining Slater winners in tournaments* is NP-Hard, but their proof also holds for *Kemeny winners*. We will describe the *Kemeny winner* problem below.) Our results and proof are novel.

Theorem 6.2.2 *(Hardness of the Judgment Problem) Finding the maximum item given evidence is* NP-*Hard.*

Proof 6.2.3 *We first describe the Kemeny winner problem. In this proof, we use an alternate (but equivalent) view of a directed weighted graph like Figure 6.1. In particular, we view weighted arcs as multiple arcs. For instance, if there is an arc from vertex A to B with weight 3, we can instead view it as 3 separate arcs from A to B. We use this alternate representation in our proof.*

An arc $i \to j$ respects a permutation if the permutation has I_j ranked higher than I_i (and does not if the permutation has I_i ranked higher than I_j). A Kemeny permutation *is simply a permutation of the vertices (items), such that the number of arcs that do not respect the permutation is minimum. There may be many such permutations, but there always is at least one such permutation. The starting vertex (rank 1 item) in any of these permutations is a* Kemeny winner. *It can be shown that finding a Kemeny winner is* NP-*Hard (using a reduction from the feedback arc set problem, similar to the proof in Hudry et al. [100]).*

We now reduce the Kemeny winner determination problem to one of finding the maximum item. Consider a directed weighted graph G, where we wish to find a Kemeny winner. We show that with a suitable probability e, which we set, the maximum item (i.e., the solution to the Judgment Problem) in G is a Kemeny winner. As before, the probability that a certain item I_j is the maximum item is the right *hand side of Equation 6.4 with k set to 1. The denominator can be ignored since it is a constant for all j. We set worker error probability e to be very close to 0. In particular, we choose a value e such that $\frac{e}{(1-e)} < \frac{1}{n!}$.*

Now, consider all permutations π_d that are not Kemeny permutations. In this case, it can be shown that

$$\sum_{d:\pi_d \text{ is not Kemeny}} \Pr(W|\pi_d) < \Pr(W|\pi_s)$$

for any Kemeny permutation π_s. Thus, the item I_j that maximizes Equation 6.4 (for $k = 1$) has to be one that is a Kemeny winner.

To see why

$$\sum_{d:\pi_d \text{ is not Kemeny}} \Pr(W|\pi_d) < \Pr(W|\pi_s)$$

for a Kemeny permutation π_s, notice that the left hand side is at most $n! \times \Pr(W|\pi'_d)$ where π'_d is the permutation (not Kemeny) that has the least number of arcs that do not respect the permutation. Note that $\Pr(W|\pi'_d)$ is at most $\Pr(W|\pi_s) \times \frac{e}{(1-e)}$, since this permutation has at least one more mistake as compared to any Kemeny permutation.

Therefore, we have shown that, for a suitable e, the maximum item in G is a Kemeny winner. Thus, we have a reduction from the Kemeny winner problem to the Judgement problem. Since finding a Kemeny winner is NP*-Hard, this implies that finding the maximum item in G is* NP*-Hard.*

#P-Hardness of Probability Computations: In addition to being NP-Hard to find the max item, we can show that evaluating the numerator of the right hand side of Equation 6.4 (with $k = 1$) is #P-Hard, in other words: computing $\Pr(\boldsymbol{\pi}^{-1}(1) = j, W)$ is #P-Hard.

We use a reduction from the problem of counting the number of linear extensions in a directed acyclic graph (DAG), which is known to be #P-Hard.

Theorem 6.2.4 *(#P-Hardness of Probability Computation) Computing $\Pr(\boldsymbol{\pi}^{-1}(1) = j, W)$ is #P-Hard.*

Proof 6.2.5 *A linear extension is a permutation of the vertices, such that all arcs in the graph respect the permutation (i.e., a linear extension is the same as a Kemeny permutation for a DAG).*

Consider a DAG $G = (V, A)$. We add an additional vertex x such that there is an arc from each of the vertices in G to x, giving a new graph $G' = (V', A')$. We now show that computing $\Pr(\boldsymbol{\pi}^{-1}(1) =$

$x, W)$ in G' can be used to compute the number of linear extensions in G. Notice that:

$$\begin{aligned}\Pr(\boldsymbol{\pi}^{-1}(1) = x, W) &= \sum_{i=0}^{|A'|} a_i (1-e)^i e^{|A'|-i} \\ &= (1-e)^{|A'|} \times \sum_{i=0}^{|A'|} a_i \left(\frac{e}{1-e}\right)^{|A'|-i} \qquad (6.5)\end{aligned}$$

where a_i is the number of permutations where there are i arcs that respect the permutation. Clearly, the number that we wish to determine is $a_{|A'|}$, since that is the number of permutations that correspond to linear extensions. Equation 6.5 is a polynomial of degree $|A'|$ in $\frac{e}{(1-e)}$, thus, we may simply choose $|A'|+1$ different values of $\frac{e}{1-e}$, generate $|A'|+1$ different graphs G', and use the probability computation in Equation 6.5 to create a set of $|A'|+1$ equations involving the a_i coefficients. We may then derive the value of $a_{|A|}$ using Lagrange's interpolation formula.

Since vertex x is the only maximum vertex in G', by computing $\Pr(\boldsymbol{\pi}^{-1}(1) = x, W)$ in G', we count the number of linear extensions in DAG G. Since counting the number of linear extensions in a DAG is #P-Hard, this implies that the computation of $\Pr(\boldsymbol{\pi}^{-1}(1) = x, W)$ in G' is #P-Hard, which implies that the computation of $\Pr(\boldsymbol{\pi}^{-1}(1) = j, W)$ for directed graph G_v (associated with vote matrix W) is #P-Hard.

6.2.4 Heuristic Strategies

The ML scoring function is computationally inefficient and also requires prior knowledge of e, the average worker error probability, which is not available to us in real-world scenarios. We next investigate the performance and efficiency of four heuristic strategies, each of which runs in polynomial time. The heuristics we present, excluding the Indegree heuristic, do not require explicit knowledge of the worker error probability.

Indegree Strategy: The first heuristic we consider is an Indegree scoring function proposed by Coppersmith et al. [63] to approximate the optimal feedback arc set in a directed weighted graph where arc weights l_{ij}, l_{ji} satisfy $l_{ij} + l_{ji} = 1$ for each pair of vertices i and j.

We can transform the vote matrix W to a graph where this Indegree scoring function can be directly applied. The idea is to construct a complete graph between all items where arc weights l_{ji} are equal to $\Pr(\boldsymbol{\pi}(i) < \boldsymbol{\pi}(j) | w_{ij}, w_{ji})$, where l_{ji} reflects the probability that I_i is greater than I_j given the *local* evidence w_{ij} and w_{ji}.

The Indegree scoring function computes the score of item I_j as: $score(j) = \sum_i l_{ij}$. Intuitively, vertices with higher scores correspond to items which have compared favorably to other items, and hence should be ranked higher. The predicted ranking has been shown to be a constant factor approximation to the feedback arc set for directed graphs where all arcs (i, j) are present and $l_{ij} + l_{ji} = 1$ [63]. The running time of this heuristic is dominated by the time to do the final sort of the scores.

Let us walk through the example graph in Figure 6.1. First, for those pairs of vertices that do not have any votes between them, we have $l_{AC} = 0.5, l_{CA} = 0.5, l_{AD} = 0.5$, and $l_{DA} = 0.5$. By symmetry, $l_{CD} = 0.5$ and $l_{DC} = 0.5$. For $e = 0.45$, we have $l_{AB} = 0.599, l_{BA} = 0.401, l_{BC} = 0.55, l_{CB} = 0.45, l_{BD} = 0.646$, and $l_{DB} = 0.354$. With these computed arc weights, we obtain the scores: $score(A) = 1.401, score(B) = 1.403, score(C) = 1.55$, and $score(D) = 1.65$, generating a predicted ranking of (D, C, B, A), with item D being the predicted maximum item. Note that if e is smaller, e.g. $e = 0.05$, the Indegree heuristic predicts the same ranking.

Local Strategy: The Indegree heuristic is simple to compute, but only takes into account local evidence. That is, the score of item I_i only depends on the votes that include I_i directly. We now consider a Local scoring function, adapted from a heuristic proposed by David [71], which considers evidence two steps away from I_i. This method was originally proposed to rank items in incomplete tournaments with ties. We adapted the scoring function to our setting, where there can be multiple comparisons between items, and there are no ties in comparisons.

This heuristic is based on the notion of wins and losses, defined as follows: $wins(i) = \sum_j w_{ji}$ and $losses(i) = \sum_i w_{ij}$. For instance, in Figure 6.1, vertex B has 3 wins and 5 losses.

The score $score(i)$ has three components. The first is simply $wins(i) - losses(i)$, reflecting the net number of votes in favor of I_i. For vertex B, this first component would be $3 - 5 = -2$. Since this first component does not reflect the "strength" of the items I_i was compared against, we next add a "reward": for each I_j such that $w_{ji} > w_{ij}$ (i has net wins over j), we add $wins(j)$ to the score of I_i. In our example, B only has net wins over A, so we reward B with $wins(A)$ (which in this case is zero). On the other hand, since C beat out B, then C gets a reward of $wins(B) = 3$ added to its score. Finally, we "penalize" $score(i)$ by subtracting $losses(j)$ for each I_j that overall beat I_i. In our example, we subtract from $score(B)$ both $losses(C) = 2$ and $losses(D) = 1$. Thus, the final score $score(B)$ is -2 plus the reward minus the penalty, i.e., $score(B) = -2 + 0 - 3 = -5$.

More formally, score $score(i)$ is defined as follows:

$$score(i) = wins(i) - losses(i) + \sum_j \left[\mathbf{1}(w_{ji} > w_{ij}) wins(j)\right] - \sum_j \left[\mathbf{1}(w_{ij} > w_{ji}) losses(j)\right]$$

Having computed $score(\cdot)$, we sort all items by decreasing order of $score$. The resulting permutation is our predicted ranking, with the vertex having largest $score$ being our predicted maximum item.

To complete the example of Figure 6.1, The strategy above computes the following scores: $score(A) = 0 - 2 - 5 = -7$, $score(B) = 3 - 5 - 3 = -5$, $score(C) = 3 - 2 + 3 = 4$, and $score(D) = 4 - 1 + 3 = 6$. The predicted ranking is then (D, C, B, A), with item D being the predicted maximum item.

PageRank Strategy: Both the Indegree and Local heuristics use only information one or two steps away to make inferences about the items of $\mathcal{I}$. We next consider a global heuristic scoring function inspired by the PageRank [148] algorithm. The general idea behind using a PageRank-like procedure is to utilize the votes in W as a way for items to transfer "strength" between each other. We design a modified PageRank algorithm to predict the maximum item in $\mathcal{I}$, which in particular, can handle directed cycles in the directed graph representing W.

Consider again the directed graph G_v representing the votes of W (Figure 6.1 is an example). Let $d^+(i)$ to denote the outdegree of vertex i in G_v, e.g. $d^+(i) = \sum_j w_{ij}$. If $d^+(i) = 0$, we say that i is a *sink* vertex. Let $pagerank_t(i)$ represent the PageRank of vertex i in iteration t. We initialize each vertex to have the same initial PageRank, e.g., $pagerank_0(\cdot) = \frac{1}{n}$. In each iteration $t+1$, we apply the following update equation to each vertex i:

$$pagerank_{t+1}(i) = \sum_j \frac{w_{ji}}{d^+(j)} pagerank_t(j) \tag{6.6}$$

For each iteration, each vertex j transfers all its PageRank (from the previous iteration) proportionally to the other vertices i whom workers have indicated may be greater than j, where the proportion of j's PageRank transferred to i is equal to $\frac{w_{ji}}{d^+(j)}$. Intuitively, $pagerank_t(i)$ can be thought as a proxy for the probability that item I_i is the maximum item in $\mathcal{I}$ (during iteration t).

What happens to the PageRank vector after performing many update iterations using Equation 6.6? Considering the strongly connected components (SCCs) of G_v, let us define a *terminal* SCC to be a SCC whose vertices do not have arcs transitioning out of the SCC. After a sufficient number of iterations, the PageRank probability mass in G_v becomes concentrated in the terminal SCCs of G_v, with all other vertices outside of these SCCs having zero PageRank [48]. In the context of our problem, these terminal SCCs can be thought of as sets of items which are ambiguous to order.

Our proposed PageRank algorithm is described in Strategy 4. We now describe how our strategy is different from the standard PageRank algorithm. The original PageRank update equation is:

$$pagerank_{t+1}(i) = \frac{1-\gamma}{n} + \gamma \sum_j \frac{w_{ji}}{d^+(j)} pagerank_t(j)$$

Algorithm 4: PageRank Maximum Strategy

```
Data: $n$ items, vote matrix $W$, $\alpha$ iterations
Result: $ans$ = predicted maximum item
begin
    construct $G_v = (V, A)$ from $W$;
    // compute all outdegrees
    compute $d^+[\cdot]$ for each vertex;
    forall the $i : 1 \ldots n$ do
        if $d^+[i] == 0$ then
            $w_{ii} \leftarrow 1$;
    // $pagerank_0$ is the PageRank vector in iteration 0
    $pagerank_0[\cdot] \leftarrow \frac{1}{n}$;
    for $k : 1 \ldots \alpha$ do
        for $i : 1 \ldots n$ do
            for $j : 1 \ldots n, j \neq i$ do
                $pagerank_k[i] \leftarrow pagerank_k[i] + \frac{w_{ji}}{d^+[j]} pagerank_{k-1}[j]$;
    compute $period[\cdot]$ of each vertex using final iterations of $pagerank[\cdot]$;
    for $i : 1 \ldots n$ do
        // $score[\cdot]$ is a vector storing average PageRank
        $score[i] \leftarrow 0$;
        for $j : 0 \ldots period[i] - 1$ do
            $score[i] \leftarrow score[i] + pagerank_{\alpha - j}[i]$;
        $score[i] \leftarrow \frac{score[i]}{period[i]}$;
    $ans \leftarrow \text{argmax}_i\, score[i]$;
```

Comparing the original equation and Equation 6.6, the primary difference is that we use a damping factor $\gamma = 1$, e.g. we remove jump probabilities. PageRank was designed to model the behavior of a random surfer traversing the web, while for the problem of ranking items, we do not need to model a random jump vector.

A second difference between our modified PageRank and the original PageRank is that prior to performing any update iterations, for each sink vertex i, we set w_{ii} equal to 1 in W. In our setting, sinks correspond to items which may be the maximum item (e.g., no worker voted that I_i is less than another item). By setting w_{ii} to 1 initially, from one iteration to the next, the PageRank in sink i remains in sink i. This allows PageRank to accumulate in sinks. Contrast this with the standard PageRank methodology, where when a random surfer reaches a sink, it is assumed that (s)he transitions to all other vertices with equal probability.

Finally, a caveat to our PageRank strategy is that the PageRank vector ($pagerank(\cdot)$ in Strategy 4) may not converge for some vertices in terminal SCCs. To handle the oscillating PageRank in terminal SCCs, we execute our PageRank update equation (Equation 6.6) for a large number of iterations, denoted as α in Strategy 4. Then, we examine the final iterations, say final 10%, of the PageRank vector to empirically determine the *period* of each vertex, where we define the period as the number of iterations for the PageRank value of a vertex to return to its current value. In practice, we find that running PageRank for α iterations, where $\alpha = O(n)$, is sufficient to detect the period of nearly all vertices in terminal SCCs. For example, consider a graph among 3 items A, B, C with 3 arcs: (A, B), (B, C), and (C, B). All vertices initially have $\frac{1}{3}$ PageRank probability. After 1 iteration, the PageRank vector is $(0, \frac{2}{3}, \frac{1}{3})$. After 2 iterations, the PageRank vector is $(0, \frac{1}{3}, \frac{2}{3})$. And so on. In this example, item B and C each have periods of 2.

With the periods computed for each vertex, we compute an *average* PageRank value for each vertex over its period. This average PageRank is used as the scoring function $score(\cdot)$ for this strategy. After the termination of PageRank, we sort the vertices by decreasing order of $score(\cdot)$, and predict that the vertex with maximum average PageRank corresponds to the maximum item in $\mathcal{I}$. Note that our PageRank heuristic is primarily intended to predict a maximum item, not to predict a ranking of all items (as many items will end up with no PageRank). The details of our implementation are displayed in Strategy 4.

To illustrate our PageRank heuristic, consider again the example in Figure 6.1. There are 2 SCCs in the graph: (A) and (B, C, D), with (B, C, D) being a terminal SCC. Each of the 4 vertices is initialized with 0.25 PageRank. After the first iteration, the PageRank vector is $(0, 0.375, 0.35, 0.275)$. After the second iteration, the PageRank vector is $(0, 0.375, 0.35, 0.275)$. After ~20 iterations, the PageRank

Heuristic	Prediction of Max
ML	D
Indegree	D
Local	D
PageRank	C
Iterative	C or D

Table 6.1: Predictions using each heuristic for Figure 6.1.

vector oscillates around $(0, 0.217, 0.435, 0.348)$. With a sufficiently large number of iterations and an appropriately chosen convergence threshold, the heuristic determines a period of 1 for both SCCs and computes an average PageRank vector of $(0, 0.217, 0.435, 0.348)$. The PageRank heuristic then predicts item C to be the maximum item in $\mathcal{I}$.

Iterative Strategy: We next propose an Iterative heuristic strategy to determine the maximum item in $\mathcal{I}$. The general framework is the following:

1. Place all items in a set.
2. Rank the items in the set by a scoring metric.
3. Remove the lower ranked items from the set.
4. Repeat steps 3 and 4 until only one item remains.

There are two parameters we can vary in this framework: the scoring metric and the number of items eliminated each iteration. Let us define the $dif(i)$ metric of item I_i to be equal to $wins(i) - losses(i)$. An implementation of the Iterative strategy using the dif metric is displayed in Strategy 5. In our particular implementation, we emphasize computational efficiency and remove half of the remaining items each iteration. The Iterative strategy relies upon the elimination of lower ranked items before re-ranking higher ranked items. With each iteration, as more items are removed, the difs of the higher ranked items separate from the difs of the lower ranked items. Basically, by removing lower ranked items, the strategy is able to more accurately rank the remaining set of items. The strategy can be thought of as iteratively narrowing in on the maximum item.

It is important to note that other scoring metrics can be used with this Iterative strategy as well. For example, by iteratively ranking with the Local heuristic, we were able to achieve (slightly) better performance than the simple dif metric. Our method is similar to the Greedy Order algorithm proposed by Cohen et al. [58], who considered a problem related to feedback arc set. Our strategy differs in that it is more general (e.g., it can utilize multiple metrics), and our strategy can be optimized (e.g., if we eliminate half of the items each iteration, we require only a logarithmic number of sorts, as opposed to a linear number).

The Iterative strategy can also be viewed as a scoring function $score(\cdot)$, like the prior heuristics we have examined. Denoted as $score[\cdot]$ in Strategy 5, we can assign each item a score equal to the iteration number in which it was removed from set T. Using this scoring function $score(\cdot)$, the predicted maximum item is then simply $\text{argmax}_i\, score(i)$.

Returning to the example graph in Figure 6.1, the Iterative heuristic first computes the dif metric for each item: $dif(A) = -2$, $dif(B) = -2$, $dif(C) = 1$ and $dif(D) = 3$. The items are then placed in a set and sorted by dif. In the first iteration, items A and B are assigned ranks 3 and 4 and removed from the set. Then, dif is recomputed among all remaining items in the set, $dif(C) = 0$ and $dif(D) = 0$. In the second iteration, either item C or D is removed and assigned rank 2. In the third iteration, the remaining item is removed and assigned rank 1. Therefore, the predicted ranking of the Iterative heuristic is equally likely to be (C, D, B, A), (C, D, A, B), (D, C, B, A), or (D, C, A, B), with the predicted maximum item of the heuristic being item C or D with equal probability.

To summarize, Table 6.1 displays the predictions for ML and our four heuristics for the example displayed in Figure 6.1. Next, we evaluate the heuristic strategies via experiments.

6.2.5 Experiments

In this section, we experimentally compare our heuristic strategies: Indegree (DEG), Local (LOC), PageRank, (PR), and Iterative (ITR). We also compare them with the Maximum Likelihood (ML) Strategy, which we consider the best possible way to select the maximum. However, since ML is computationally very expensive, we only do this comparison on a small scenario. For our experiments, we synthetically generate problem instances, varying : n (the number of items in $\mathcal{I}$), $votes$ (the number of votes we sample for W), and e (average worker error probability). We prefer to use synthetic data, since it lets us study a wide spectrum of scenarios, with highly reliable or unreliable workers, and with many or few votes.

In our base experiments, we vary the number of sampled votes $votes$, from 0 to $5n(n-1)$ and vary worker accuracy $(1-e)$ from 0.55 to 0.95. As a point of reference, we refer to $\frac{n(n-1)}{2}$ votes as $votes$ = 1x *Edge Coverage*, e.g. each pair of items is sampled approximately once. So $5n(n-1)$ votes is equivalent to $votes$ = 10x Edge Coverage in our experiments.

Each data point (given n, $(1-e)$, $votes$ values) in our results graphs is obtained from 5,000 *runs*. Each run proceeds as follows: We initialize W as an $n \times n$ null matrix and begin with an arbitrary *true*

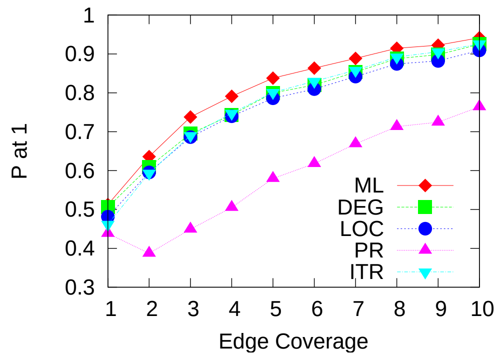
Algorithm 5: Iterative Strategy

```
Data: $n$ items, vote matrix $W$
Result: $ans$ = predicted maximum item
// $dif[\cdot]$ is the scoring metric
$dif[\cdot] \leftarrow 0$;
for $i : 1 \ldots n$ do
    for $j : 1 \ldots n, j \neq i$ do
        $dif[j] \leftarrow dif[j] + w_{ij}$;
        $dif[i] \leftarrow dif[i] - w_{ij}$;
initialize set $L$;
for $i : 1 \ldots n$ do
    $L \leftarrow L \cup i$;
while $|L| > 1$ do
    sort items in $L$ by $dif[\cdot]$;
    for $r : (\frac{|L|}{2} + 1) \ldots |L|$ do
        remove item $i$ (with rank $r$) from $L$;
        for $j : j \in L$ do
            if $w_{ij} > 0$ then
                $dif[j] \leftarrow dif[j] - w_{ij}$;
                $dif[i] \leftarrow dif[i] + w_{ij}$;
            if $w_{ji} > 0$ then
                $dif[i] \leftarrow dif[i] - w_{ji}$;
                $dif[j] \leftarrow dif[j] + w_{ji}$;
// $L[1]$ is the final item in $L$
$ans \leftarrow L[1]$;
```

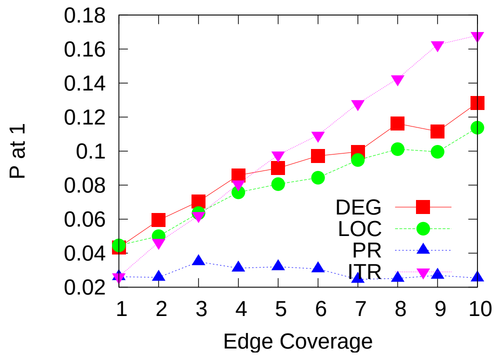
permutation π^* of the items in $\mathcal{I}$. Let U denote the set of all tuples (i, j) where $i \neq j$. We randomly sample $votes$ tuples from U with replacement. After sampling a tuple (i, j), we simulate the human worker's comparison of items I_i and I_j. If $\pi^*(i) < \pi^*(j)$, with probability $(1 - e)$, we increment w_{ji}, and with probability e, we increment w_{ij}. If $\pi^*(j) < \pi^*(i)$, with probability $(1 - e)$, we increment w_{ij}, and with probability e, we increment w_{ji}.

For each generated matrix W in a run, we apply each of our heuristic strategies to obtain predicted rankings of the items in $\mathcal{I}$.

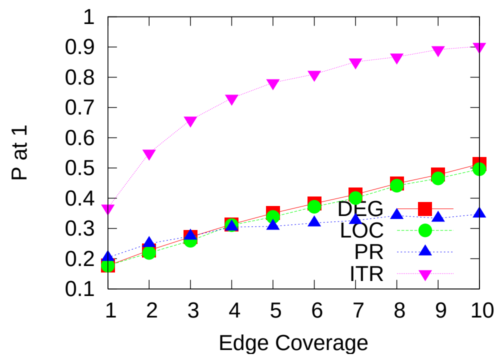
Comparing the predicted ranking with π^* we record a "yes" if the predicted maximum agrees with the true maximum. After all runs have completed, we compute Precision at 1 (P@1), the fraction of "yes" cases over the number of runs. Similar results were observed for other evaluation metrics.



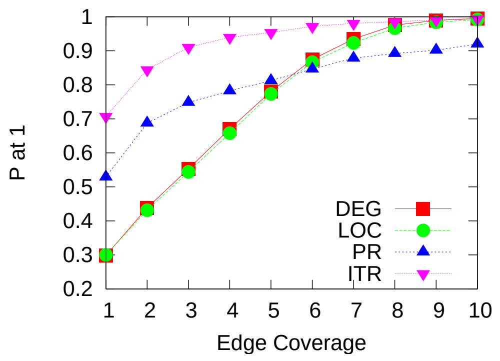
(a) e=0.25, 5 OBJ



(b) e=0.45, 100 OBJ



(c) e=0.25, 100 OBJ



(d) e=0.05, 100 OBJ

Figure 6.2: Precision at 1 (P@1) versus Edge Coverage.

As a first experiment, we consider the prediction performance of Maximum Likelihood (ML) and the four heuristics for a set of 5 items with $e = 0.25$, displayed in Figure 6.2(a). We choose a small set of items, so that ML can be computed. We find that as the number of votes sampled increases, the P@1 of all heuristics (excluding PageRank) increase in a concave manner, approaching a value of 0.9 for 10x Edge Coverage (e.g., if $5n(n-1)$ votes are uniformly sampled, the heuristics can predict the maximum item 90% of the time, even though average worker accuracy is 0.75).

ML has better performance than all the four heuristics.

As expected, ML performs the best in Figure 6.2(a), but recall that ML requires explicit knowledge of e, and it is computationally very expensive. Still, the ML curve is useful, since it tells us how far the heuristics are from the optimal feasible solution (ML). Also, note that PageRank (PR) performs poorly, indicating that PageRank is poor when the number of items is small.

Iterative is the best of the four heuristics when the number of votes sampled is $\frac{n(n-1)}{2}$, e.g. 1x Edge Coverage.

For a larger experiment, we consider the problem of prediction for $n = 100$ items in Figure 6.2(b), (c), and (d). ML is necessarily omitted from this experiment. Looking at the graphs, we first note that the Iterative (ITR) heuristic performs significantly better than the other heuristics, particularly when $e = 0.45$ or $e = 0.25$. This is best demonstrated by Figure 6.2(c), which shows that for $e = 0.25$ and 10x Edge Coverage, the Iterative heuristic has a P@1 of over 0.9, whereas the second best heuristic, Indegree (DEG), only has a P@1 of approximately 0.5. Looking at the middle graph again, note how the performance gap between the Iterative heuristic and the other heuristics widens as the Edge Coverage increases from 1x to 5x. The strength of the Iterative strategy comes from its ability to leverage the large number of redundant votes, in order to iteratively prune out lower-ranked items until there is a predicted maximum. The strategy is robust even when worker accuracy is low. When average worker accuracy is high, Figure 6.2(d), the Iterative heuristic still is the heuristic of choice, although the performance gap between the Iterative and Indegree or Local (LOC) heuristics decreases to a minimal amount, as the number of votes sampled becomes very large.

PageRank is a poor heuristic when worker accuracy is low. However, when worker accuracy is reasonable, PageRank is quite effective, even when the number of votes is low.

We next focus upon the performance of the PageRank (PR) heuristic. For $e = 0.25$ and $e = 0.05$, the PageRank heuristic's prediction curve crosses the prediction curves for the Indegree (DEG) and Local (LOC) heuristics. This is an indication that the PageRank heuristic is quite effective when the number of votes is low, but is unable to utilize the information from additional votes when the number of votes is large. We also observe the poor performance of PageRank when $e = 0.45$, in Figure 6.2(b), indicating that PageRank is not a suitable heuristic when worker accuracy is low.

Over various worker accuracies, Iterative is the best heuristic, followed by PageRank, Local and Indegree.

From the prior experiments, we see that prediction performance for each strategy varies greatly with respect to the average worker accuracy $(1-e)$. We next directly investigate prediction performance versus worker accuracy for a fixed 1x Edge Coverage in Figure 6.3(left). We find that for this

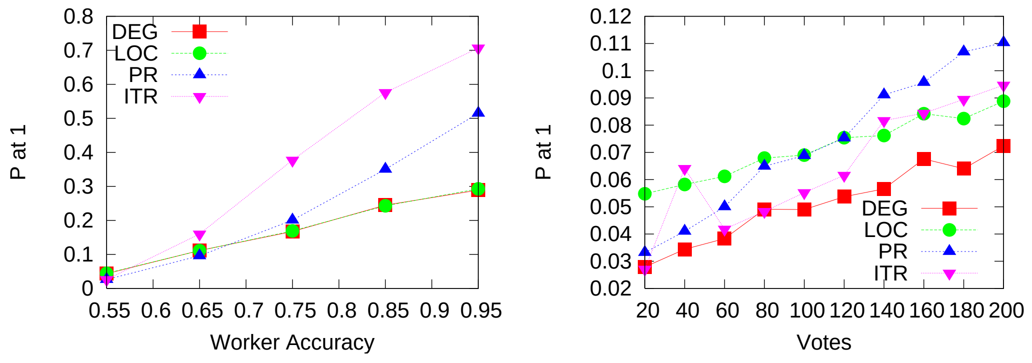


Figure 6.3: P@1 versus worker accuracy (left), 1x Edge Coverage. P@1 versus number of votes (right), e=0.05. 100 items.

fixed Edge Coverage, the Iterative (ITR) strategy performs the best, followed by PageRank (PR), then the Local (LOC) and Indegree (DEG) heuristics. As expected, prediction performance increases with worker accuracy across all strategies. In particular, note the large slope of the Iterative and PageRank prediction curves, as compared to the Local and Indegree prediction curves, which are near identical.

PageRank is the best of the four heuristics when there are few votes and worker accuracy is high.

All experiments considered thus far examine prediction when the number of votes is an order of magnitude larger than the number of items. For a more difficult scenario, we examine prediction performance when the number of votes is approximately the same as the number of items. Figure 6.3(right) displays P@1 for 100 items when the number of votes is varied from 20 to 200 and $e = 0.05$. We observe that PageRank (PR) has the highest prediction performance among the four heuristics. Conducting several other experiments, we find that, so long as worker accuracy is high, PageRank facilitates good prediction, even when the number of votes is low relative to the number of items. This fact will prove useful when we consider the problem of selecting which additional votes to request, given an initial sparse vote graph.

From our experiments, we conclude that Iterative (ITR) is the strategy of choice when evaluating a large number of votes (relative to the number of items), whereas PageRank is the preferred heuristic when evaluating a small number of votes.

6.3 Next Votes Problem

We now consider the second half of the Max Problem, the Next Votes Problem. Beginning with an initial vote matrix W, if we wish to submit additional vote requests to a crowdsourcing marketplace, which additional votes (i.e., comparisons between pairs of items) should be requested to augment our existing vote matrix W, and improve our prediction of the maximum item? In particular, we assume that we are given a vote budget of b additional votes that may be requested. There are two ways in which we can use this vote budget: (a) an adaptive strategy, where we submit some initial votes, get some responses, then submit some more, get more responses, and so on, or (b) a one-shot strategy, where we submit all votes at once. In this chapter, we consider a one-shot strategy with a vote budget of b. This strategy is more relevant in a crowdsourcing setting since the latency of crowdsourcing is high. Once the responses for these vote requests are received, we assume that the entire evidence thus far is our new vote matrix W'. Note that we can iteratively submit batches of votes to improve our prediction of the maximum item. As before, we assume that the response to each vote is independently correct with probability $(1 - e)$. We define the Next Votes Problem as follows:

Problem 6.3.1 (Next Votes) *Given b, W, select b additional votes and predict the maximum item in $\mathcal{I}$, $\pi^{*-1}(1)$.*

6.3.1 Maximum Likelihood

We first present a Maximum Likelihood (ML) formulation of the selection of votes for the Next Votes Problem; we directly compute the multiset of votes which most improves the prediction of the maximum item in $\mathcal{I}$. Assuming that average worker error probability e is known, the ML vote selection formulation we present is the optimal feasible solution to the Next Votes Problem. Beforing presenting the ML formulation, we first provide some definitions needed for the Next Votes Problem.

Vote and Answer Multisets: We represent a potential vote (comparison) between items I_i and I_j as a unordered pair $\{I_i, I_j\}$. Given a vote budget b, all possible multisets $\mathcal{Q}$ of b votes are allowed (note that repetition of votes is allowed). For a potential vote $\{I_i, I_j\}$, we define an answer to be a tuple $(\{I_i, I_j\}, I_x)$, where the first element of the tuple is an unordered pair, and the second element is one of the items in the pair indicating the human worker's answer (e.g., $x = i$ if the worker states that I_i

is greater than I_j, or $x = j$ otherwise).

For each vote multiset $\mathcal{Q}$, we define an answer multiset a of $\mathcal{Q}$ to be a multiset of answer tuples, where there is a one-to-one mapping from each unordered pair in $\mathcal{Q}$ to an answer tuple in a. Each vote is answered (independently) with probability of error e. As an example, if $\mathcal{Q} = \{\{I_i, I_j\}, \{I_k, I_l\}\}$, a possible answer multiset a that could be received from the workers is $\{(\{I_i, I_j\}, I_i), (\{I_k, I_l\}, I_k)\}$. For a multiset of b votes, there are 2^b possible answer multisets. Let $A(\mathcal{Q})$ denote the multiset of all possible answer multisets of $\mathcal{Q}$.

Having defined vote and answer multisets, we next consider the probability of receiving an answer multiset given W, then explain how to compute the confidence of the maximum item having received an answer multiset, before finally presenting the ML vote selection strategy.

Probabilities of Multisets and Confidences: Suppose that we submitted vote multiset $\mathcal{Q}$ and received answer multiset a from the crowdsourcing marketplace. Let $\Pr(a|W)$ denote the probability of observing an answer multiset a for $\mathcal{Q}$, given initial vote matrix W. We have the following:

$$\Pr(a|W) = \frac{\Pr(a \wedge W)}{\Pr(W)} \tag{6.7}$$

where $a \wedge W$ is the new vote matrix formed by combining the votes of a and W.

Our estimate for how well we are able to predict the maximum item in $\mathcal{I}$ is then the probability of the maximum item, given the votes of our new vote matrix, i.e., $a \wedge W$. We denote this value by $P_{max}(a \wedge W)$, i.e., this value is our confidence in the maximum item. The computation, based upon Equation 6.4, is the following:

$$P_{max}(a \wedge W) = \max_i \Pr(\boldsymbol{\pi}^{-1}(1) = i | a \wedge W)$$

This simplifies to give:

$$P_{max}(a \wedge W) = \frac{\max_i \Pr(\boldsymbol{\pi}^{-1}(1) = i, a \wedge W)}{\Pr(a \wedge W)} \tag{6.8}$$

Maximum Likelihood Strategy: We can now define the Maximum Likelihood formulation of the Next Votes Problem. We wish to find the multiset $\mathcal{Q}$ of b votes such that, on average over all possible answer multisets for $\mathcal{Q}$ (and weighted by the probability of those answer multisets), our confidence in the prediction of the maximum item is greatest.

In other words, we want to find the multiset that maximizes:

$$\sum_{a \in A(\mathcal{Q})} \Pr(a|W) \times P_{max}(a \wedge W)$$

which, on using Equations 6.7 and 6.8, simplifies to:

$$\frac{1}{\Pr(W)} \times \sum_{a \in A(\mathcal{Q})} \max_i \Pr(\boldsymbol{\pi}^{-1}(1) = i, a \wedge W)$$

Since $\Pr(W)$ is a constant, independent of $\mathcal{Q}$, we have:

ML Formulation 2 (Next Votes) *Given b, W, find the vote multiset $\mathcal{Q}$, $|\mathcal{Q}| = b$, that maximizes*

$$\sum_{a \in A(\mathcal{Q})} \max_i \Pr(\boldsymbol{\pi}^{-1}(1) = i, a \wedge W) \tag{6.9}$$

Let $score(\mathcal{Q})$ be the value in Equation 6.9. We now have an exhaustive strategy to determine the best multiset $\mathcal{Q}$: compute $score(\cdot)$ for all possible multisets of size b, and then choose the multiset with the highest score. Although this strategy is the optimal feasible solution to the Next Votes Problem, it is also computationally infeasible, since a single iteration of ML itself requires enumeration of all $n!$ permutations of the items in $\mathcal{I}$. Additionally, knowledge of worker error probability e is required for ML vote selection. This leads us to develop our own vote selection and evaluation framework enabling more efficient heuristics.

6.3.2 Computational Complexity

As in the Judgment Problem, the Next Votes Problem also turns out to be NP-Hard, while the computation of the probabilities involved also turns out to be #P-Hard. While the proofs use reductions from similar problems, the details are quite different.

Hardness of the Next Votes Problem: We first show that the ML formulation for the Next Votes Problem is NP-Hard, implying that finding the optimal set of next votes to request is intractable.

Theorem 6.3.2 (Hardness of Next Votes) *Finding the vote multiset Q that maximizes*

$$\sum_{a \in A(Q)} \max_i \Pr(\boldsymbol{\pi}^{-1}(1) = i, a \wedge W)$$

is NP*-Hard, even for a single vote.*

Proof 6.3.3 *(Sketch) Our proof for the Next Votes problem uses a reduction from the same* NP*-Hard problem described in Section 6.2.3, i.e., determining Kemeny winners.*

We are given a graph G where we wish to find a Kemeny winner. We add an extra vertex $\mathtt{votes}$ to this graph to create a new graph, G', where $\mathtt{votes}$ does not have any incoming or outgoing arcs. By definition, $\mathtt{votes}$ is a Kemeny winner in G', since trivially, $\mathtt{votes}$ can be placed anywhere in the permutation without changing the number of arcs that are respected. Therefore, there are at least two Kemeny winners in G'. Recall, however, that our goal is to return a Kemeny winner in G', not in G.

Now, consider the solution to the Next Votes problem on G', where an additional vote is requested. As in the proof of Theorem 6.2.2, we set e to be very close to 0. It can be shown that the solution to the Next Votes problem on G' consists of two vertices, such that they are both Kemeny winners on G'. (Note that $\mathtt{votes}$ may be one of the vertices, but at least one more vertex is returned.) These vertices (if they are not $\mathtt{votes}$) are also Kemeny winners in G. Thus, the Kemeny winner determination problem on G can be reduced to the Next Votes (with one vote) problem on G'.

To complete the proof, we need to show that the two vertices returned by the Next Votes problem are both Kemeny winners. Let the two vertices be x, y. As before, recall that

$$\Pr(\boldsymbol{\pi}^{-1}(1) = i, a \wedge W) = \sum_{\pi : i\ wins} \Pr(\pi) \Pr(W \wedge a | \pi)$$

Ignoring $\Pr(\pi)$, which is a constant, we have two terms:

$$F = \max_i \sum_{\pi : i\ wins} \Pr(W \wedge x > y | \pi) + \max_i \sum_{\pi : i\ wins} \Pr(W \wedge y > x | \pi)$$

Now consider Kemeny permutations of W. Let the set of Kemeny winners be S, and let the number of Kemeny permutations beginning with each of the winners be $s_1 \geq s_2 \geq \ldots s_n$. We also let the probability e be very close to 0 so that only Kemeny permutations form part of F. If we choose two Kemeny winners as x and y, the expression F can be as large as $(s_1 + s_2) \times P$, where P is the probability corresponding to one Kemeny permutation. On the other hand, if both of x and y are not Kemeny winners, then we can show that $F < (s_1 + s_2) \times P$ (since the constraint of $x < y$ and $y > x$ eliminates some non-zero number *of permutations from the right hand side of the expression.) Now it remains to be seen if x may be a Kemeny winner while y is not. Clearly, the first term can be as big as $s_1 P$. It remains to be seen if the second term can be $s_2 P$. Since x is Kemeny, enforcing that $y > x$ is going to discount all permutations where $max > x > y$. Thus the second term cannot be as big as $s_2 P$. Thus both the vertices returned by the Next Votes problem are Kemeny winners.*

#P-Hardness of Probability Computations: We next show that computing Equation 6.9 is #P-Hard.

Theorem 6.3.4 (#P-Hardness of Next Votes) *Computing*

$$\sum_{a \in A(Q)} \max_i \Pr(\pi^{-1}(1) = i, a \wedge W)$$

is #P-Hard, even for a Q with a single vote.

Proof 6.3.5 *(Sketch) Our proof uses a reduction from the #P-Hard problem of counting linear extensions in a DAG. Consider a DAG $G = (V, A)$. We now add two additional vertices, x and y, such that there is an arc from each of the vertices in G to x and to y giving a new graph $G' = (V', A')$.*

Consider the computation of

$$\sum_{a \in A(Q)} \max_i \Pr(\pi^{-1}(1) = i, a \wedge W)$$

for $Q = \{\{x, y\}\}$ for G', which simplifies to:

$$\max_i \sum_{\pi : i \ wins} \Pr(W \wedge (x > y)|\pi) + \max_i \sum_{\pi : i \ wins} \Pr(W \wedge (y > x)|\pi).$$

The first of these two terms is maximized when x is the maximum, and the second term is maximized when y is the maximum. Both terms are identical, since x and y are identical, so we focus on only one of the terms. Let $F(e) = \sum_{\pi : x \ wins} \Pr(W \wedge x > y|\pi)$. Using a calculation similar to that used to derive Equation 6.5, we have:

$$F(e) = (1 - e)^{|A'|} \times \sum_{i=0}^{|A'|} a_i \left(\frac{e}{1 - e}\right)^{|A'| - i}.$$

We are interested in $a_{|A'|}$, the number of permutations that correspond to linear extensions. Once again, by repeating the trick in Theorem 6.2.4, we may use multiple values for e to generate different graphs G', *and use the probability computation to derive many equations $F(e)$ corresponding to different e, and then derive the value of $a_{|A'|}$ using Lagrange's interpolation.*

Therefore, counting the number of linear extensions in G can be reduced to a polynomial number of instances of computing the probability expression corresponding to the Next Votes problem.

6.3.3 Selection and Evaluation of Additional Votes

We next present a general framework to select and evaluate additional votes for the Next Votes Problem. Our approach is the following:

1. score all items with a scoring function $score(\cdot)$ using initial vote matrix W
2. select a batch of b votes to request
3. evaluate the new matrix $W^{'}$ (initial votes in W and additional b votes) with a scoring function $final$ to predict the maximum item in $\mathcal{I}$.

This framework is displayed in more detail in Algorithm 6. In Step 1, we use a scoring function $score(\cdot)$ to score each item, and in Step 3, we use a scoring function $final(\cdot)$ to evaluate the new matrix $W^{'}$ to predict the maximum item in $\mathcal{I}$. We briefly discuss the choice of these scoring functions when presenting experimental results later in Section 6.3.4. For now, we assume the use of a scoring function in Step 1 which scores items proportional to the probability that they are the maximum item in $\mathcal{I}$. It is important to note that our general framework assumes no knowledge of worker accuracy, unlike in ML vote selection. We next focus our attention upon how to select b additional votes (Step 2).

Heuristic Vote Selection Strategies: How should we select pairs of items for human workers to compare, when given a vote budget of b votes? Since ML vote selection is computationally infeasible, we consider four efficient polynomial-time vote selection strategies: Paired, Max, Greedy, and Complete Tournament strategies. For ease of explanation, we use the graph in Figure 6.4 as an example. Before executing a vote selection strategy, we assume that each item has been scored by a scoring function in Step 1 of the framework, denoted by $score[\cdot]$ in Algorithm 6. As a running example to explain our strategies, we assume that our PageRank heuristic (Section 6.2.4) is used as the scoring function in Step 1: item A has score 0.5, items B and E each have score 0.25, and items C, D, and F have score 0. Without loss of generality, assume that the final rank order of the items, before next vote selection, is (A, B, E, C, D, F).

```
Algorithm 6: General Vote Selection Framework
    Data: $n$ items, vote matrix $W$, budget $b$
    Result: $ans$ = predicted maximum item
    // Step 1
    compute score $score[\cdot]$ for all items using function $score$;
    initialize multiset $\mathcal{Q}$ ;
    sort all items by $score[\cdot]$, store item indices in $index[\cdot]$ ;
    // Step 2
    select $b$ votes for $\mathcal{Q}$ using a vote selection strategy;
    submit batch $\mathcal{Q}$;
    update $W$ with new votes from workers;
    // Step 3
    compute final score $final[\cdot]$ for all items using function $final$;
    $ans \leftarrow \text{argmax}_i\, final[i]$
```

The first strategy we consider is Paired vote selection (PAIR). In this strategy, pairs of items are selected greedily, such that no item is included in more than one of the selected pairs. For example, with a budget of $b = 2$, the strategy asks human workers to compare the rank 1 and rank 2 items, and the rank 3 and rank 4 items, where rank is determined by the scoring function from Step 1 in Algorithm 6. The idea behind this strategy is to restrict each item to be involved in at most one of the additional votes, thus distributing the b votes among the largest possible set of items. This can be anticipated to perform well when there are many items with similar scores, e.g., when there are many items in the initial vote graph G_v which have equally high chances of being the maximum item. Considering the example in Figure 6.4, for $b = 2$, this strategy requests the votes (A, B) and (E, C).

The second strategy we consider is Max vote selection (MAX). In this strategy, human workers are asked to compare the top-ranked item against other items greedily. For example, with a budget of $b = 2$, this strategy asks human workers to compare the rank 1 and rank 2 items, and the rank 1 and rank 3 items, where rank is determined by the scoring function in Step 1 in Algorithm 6. Considering again the example in Figure 6.4, for $b = 2$, this strategy requests the votes (A, B) and (A, E).

The third strategy we consider is Greedy vote selection (GREEDY). In this strategy, all possible comparisons (unordered item pairs) are weighted by the product of the scores of the two items, where the scores are determined in Step 1 of Algorithm 6. In other words, a distribution is constructed across all possible item pairs, with higher weights assigned to item pairs involving high scoring items (which are more likely to be the maximum item in $\mathcal{I}$). After weighting all possible item pairs, this strategy submits the b highest weight pairs for human comparison. Considering the example in Figure 6.4,

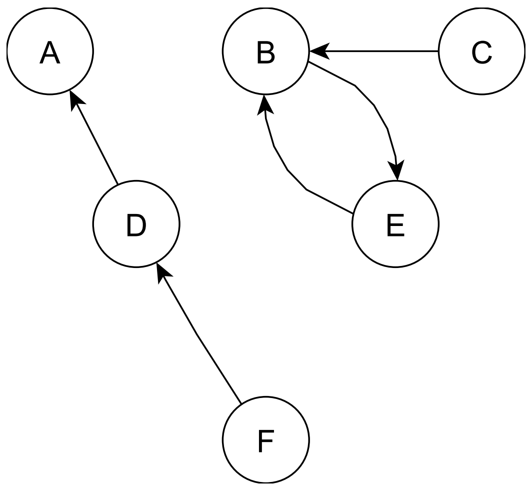


Figure 6.4: How should we select additional votes to request?

item pairs (A, B) and (A, E) has weight 0.125, (B, E) has weight 0.0625, and all other pairs have weight 0. For a budget $b = 2$, this strategy requests the votes (A, B) and (A, E).

The fourth strategy we consider is Complete Tournament vote selection (COMPLETE). In this strategy, we construct a single round-robin tournament among the K items with the highest scores from Step 1 of Algorithm 6, where K is the largest number such that $\frac{K*(K+1)}{2} \leq b$. In a single round-robin tournament, each of the K items is compared against every other exactly once. For the remaining $r = b - \frac{K*(K+1)}{2}$ votes, we consider all item pairs containing the $(K+1)st$ (largest scoring) item and one of the first K items, and weight each of these K item pairs by the product of the scores of the two items (as we did with Greedy vote selection). We then select the r item pairs with highest weight.

The idea behind the Complete Tournament strategy is that a round-robin tournament will likely determine the largest item among the set of K items. If the set of K items contains the true max, this strategy can be anticipated to perform well. Regarding the selection of the remaining votes, the strategy can be thought of as augmenting the K item tournament to become an incomplete $K+1$ item tournament, where the remaining votes are selected greedily to best determine if the $(K+1)st$ item can possibly be the maximum item in $\mathcal{I}$. Considering the example in Figure 6.4, for $b = 2$, there is a 2-item tournament among items A and B and vote (A, B) is requested. Then, for the remaining vote, the strategy greedily scores item pairs which contain both the next highest ranked item not in the tournament, item E, and one of the initial 2 items. item pair (A, E) will be scored 0.125 and (B, E) will be scored 0.0625, so the second vote requested is (A, E).

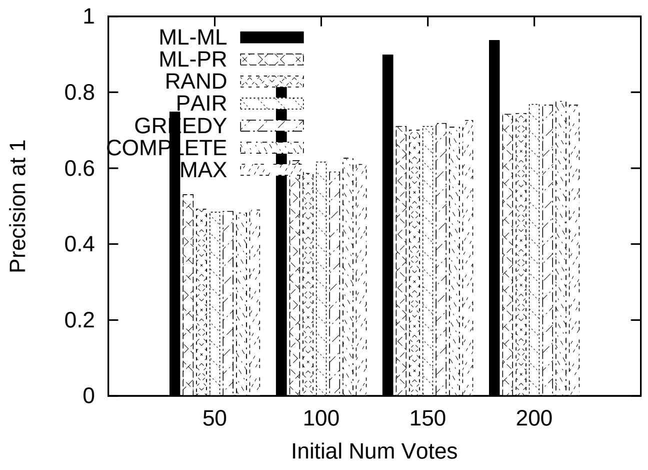


Figure 6.5: Precision at 1 versus number of initial votes. 1 additional vote, 7 items, e=0.25.

6.3.4 Experiments

Which of our four vote selection heuristics (PAIR, MAX, GREEDY, or COMPLETE) is the best strategy? We now describe a set of experiments measuring the prediction performance of our heuristics for various sets of parameters. When evaluating our vote selection strategies, we utilized a uniform vote sampling procedure, described previously in Section 6.2.5, to generate an initial vote matrix W. Then, in Step 1 of our vote selection framework (Algorithm 6), we adopted our PageRank heuristic (Section 6.2.4) as our scoring function $score(\cdot)$ to score each item in $\mathcal{I}$. In Step 2, we executed each of our vote selection strategies using these scores. In Step 3, we used our PageRank heuristic as our scoring function $final(\cdot)$ to score each item in the new matrix W' (composed of both the initial votes in vote matrix W and the b requested additional votes), and generate final predictions for the maximum item in $\mathcal{I}$. We performed several experiments contrasting prediction performance of PageRank versus other possible scoring functions and found PageRank to be superior to the other functions. Hence, we selected PageRank as the scoring function for both Step 1 and Step 3 of our vote selection framework.

- ML vote selection outperforms heuristic strategies when results are evaluated with ML scoring.
- However, when ML vote selection is evaluated with PageRank (e.g., like the heuristics), prediction performances of all methods are similar.

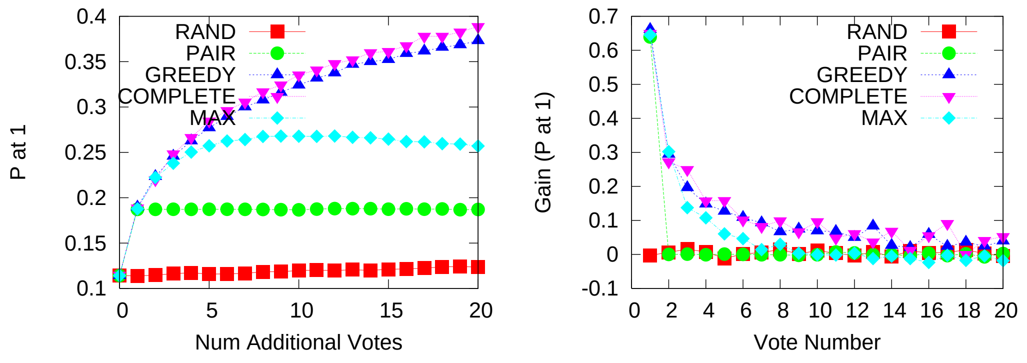


Figure 6.6: Precision at 1 versus number of additional votes (left). Incremental Gain (P@1) of each vote relative to a 0 additional votes baseline (right). 100 items, e=0.05, 200 initial votes.

For a first experiment, we compare the prediction performance (Precision at 1) of our four vote selection heuristics (and random initial vote selection (RAND)) against the "optimal" strategy, i.e., the Maximum Likelihood (ML) vote selection procedure described in Section 6.3.1. Recall that ML can be used in two places: when selecting additional votes (as in Section 6.3.1), and when predicting the max given the initial plus additional votes (e.g., ML evaluation in Section 6.2.2). We use ML-ML to refer to using ML for both tasks, this gives the best possible strategy. To gain additional insights, we also consider ML-PR, a strategy where ML is used to select the additional votes, and PageRank is used to select the winner. Since ML is computationally very expensive, for this experiment we consider a small problem: select *one* additional vote given a set of 50 (2.5x Edge Coverage) to 200 initial votes (10x Edge Coverage) among a set of 7 items, $e = 0.25$.

Our experimental results are displayed in Figure 6.5. First, as expected, ML-ML has the best performance. Clearly, ML-ML is doing a better job at selecting the additional vote and in selecting the winner. Of course, keep in mind that ML-ML is not feasible in most scenarios, and it also requires knowledge of the worker error rate e. Nevertheless, the gap between ML-ML and the other strategies indicates there is potential room for future improvement beyond the heuristics we have developed.

Second, we observe in Figure 6.5 that all other strategies, including ML-PR, perform similarly. The relative performance of ML-PR indicates that the gain achieved by ML-ML is due to its better prediction of the winner, as opposed to its choice for the next vote. In hindsight, this result is not surprising, since the selection of a single vote cannot be expected to have a large impact. (We will observe larger impacts when we select multiple additional votes.) The results also demonstrate that our vote selection heuristics show promise, since they seem to be doing equally well as ML, and since they often perform slightly better than RAND, at least for the selection of a single next vote.

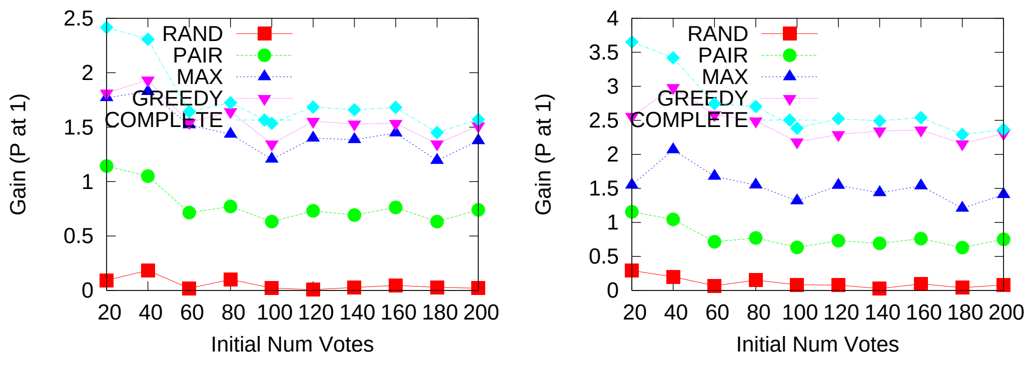


Figure 6.7: Gain (P@1) relative to a 0 additional votes baseline vs number of initial votes. 100 items, e=0.05. 5 add. votes (left), 15 add. votes (right).

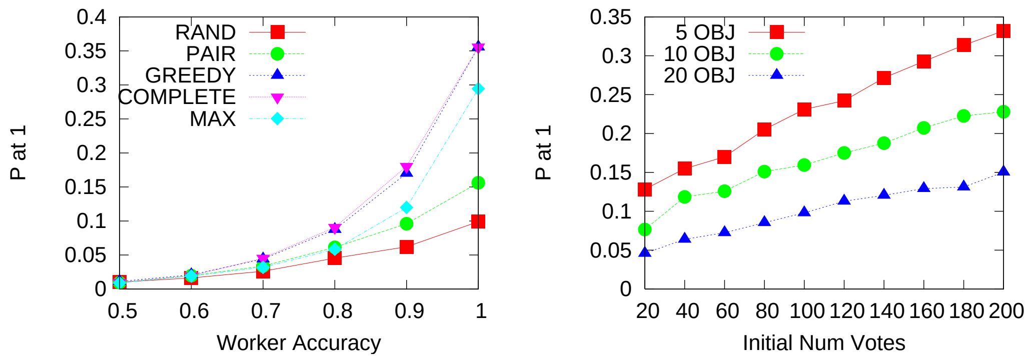


Figure 6.8: Precision at 1 versus worker accuracy (left). 100 items, 100 initial votes, 15 additional votes. Variations of the Complete Tournament strategy, 10 additional votes (right). 5 OBJ = compare 5 items 4 times each. 10 OBJ = compare 10 items 2 times each. 20 OBJ = compare 20 items 1 time each. 100 items, e=0.05.

To evaluate our heuristics in larger scenarios, we conducted a series of experiments, and the results of some of those are summarized in Figures 6.6, 6.7, and 6.8(left). To begin, we summarize some of the general trends that can be observed in these figures.

General observations regarding all strategies:

- As the number of additional votes increases, prediction performance increases.
- As the number of additional votes increases, the gain from additional votes decreases (though the decrease is not very dramatic).
- As worker accuracy increases, prediction performance increases.

- As worker accuracy increases, the gain from additional votes increases.

We only explain the graph in Figure 6.6(right), since the others are self-explanatory. In this graph, the vertical axis shows the incremental P@1 gain at b additional votes, defined as (P@1 with b additional votes - P@1 with $b - 1$ additional votes) / (P@1 with 0 additional votes). As we can see, the information provided by additional votes is more valuable when there are fewer initial votes (second bullet above).

The Complete Tournament and Greedy strategies are significantly better than the Max and Paired strategies.

We can also use Figures 6.6, 6.7 and 6.8(left) to compare our heuristics. First, notice that the difference between heuristics can be very significant. For instance, in Figure 6.7(left) we see that the Paired (PAIR) strategy provides a 0.7x P@1 gain for 5 additional votes (100 initial votes, $e = 0.05$), while the Complete Tournament (COMPLETE) strategy provides a 1.5x P@1 gain, where we measure P@1 gain as (P@1 with b votes - P@1 with 0 votes) / (P@1 with 0 votes). Second, we observe that the Complete Tournament and Greedy (GREEDY) vote selection strategies consistently outperform the Max (MAX) and Paired strategies in all scenarios. In particular, in Figure 6.6(left), we observe that the prediction performances of the Complete Tournament and Greedy strategies steadily improve with additional votes, while the Max and Paired strategies taper off. This indicates that when a larger vote budget b is available for additional votes, the additional votes will be better utilized by the more sophisticated strategies (Complete Tournament and Greedy) as compared to the simpler strategies (Max and Paired).

Given only votes between items of the same type:

- The value of additional votes is greater when it is more difficult to predict the maximum item.
- The Complete Tournament strategy is the best strategy.

In our scenarios so far, the Complete Tournament and Greedy strategies perform similarly. To differentiate between the two, we explored different ways in which the initial votes could be generated.

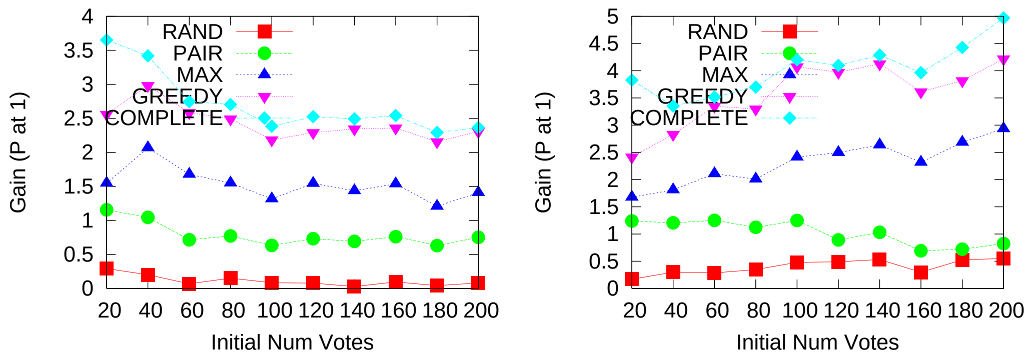


Figure 6.9: items are divided into t initial item types. Gain (P@1) relative to a 0 additional votes baseline vs number of initial votes. 100 items, e=0.05, 15 additional votes. 1 type (left), 20 types (right).

(Recall that up to this point we have been randomly selecting the pairs of items that are compared by the initial votes.) We next discuss one of these possible different vote generation schemes. Suppose that our items are of different types (e.g., softcover books, hardcover books, e-books, etc.), and for some reason initial votes between items of the same type are much more likely than across types. For example, it is more likely that two e-books have been compared, rather than one e-book and one hard-cover book. (The situation is analogous to sporting events, where intra-league games are more likely than inter-league games.)

For our experiment, we consider an extreme instance where there are *no* initial votes involving items of different types. In particular, we divide our set $\mathcal{I}$ of n items into t disjoint item types. When votes are sampled for the initial vote matrix W, sampling of votes is only permitted between items of the same type. Keep in mind that predicting the maximum item in $\mathcal{I}$ is more difficult when there are more item types because each item type will likely have a leader (greatest item), each of these leaders will have on average similar probabilities of being the maximum item (since item type groups are likely of similar size), and the initial vote matrix W provides no information regarding comparisons between these leaders.

We perform experiments for different values of t (e.g., different numbers of initial item types), Figure 6.9 displays Precision at 1 gain relative to a 0 additional votes baseline for $t = 1$ and $t = 20$. We observe that the P@1 gain increases for the Complete Tournament and Greedy strategies as t increases, implying that the value of additional votes is greater when it is more difficult to predict the maximum item from the initial vote matrix. That is, in the harder problem instances (larger t), the additional votes play a more critical role in comparing the item type leaders. More importantly, the Complete Tournament strategy outperforms the Greedy strategy (and the others too) in this more challenging scenario.

Finally, we conduct a more in-depth study of the Complete Tournament vote selection strategy and examine the benefit of vote redundancy. Given a limited budget, should the Complete Tournament strategy select fewer top items and propose more redundant votes, or should it select more items and ask fewer votes per pair? For instance, the Complete Tournmament strategy could select the top three items and submit four votes for each pair, for a total of 12 additional votes. Or it could select the top 4 items, and for each of the possible 6 comparisons, request 2 votes (for the same 12 total additional votes). What is the best approach?

Figure 6.8(right) displays the Precision at 1 of the Complete Tournament strategy for 10 additional votes, where the votes are uniformly and randomly distributed among the 5, 10, or 20 items in $\mathcal{I}$ with highest score (as provided in Step 1 of Algorithm 6). We find that distributing the 10 votes among 5 items, where each item is compared against every other, leads to the best prediction performance. That is, we do not observe any benefit for distributing votes among a larger set of items when using the Complete Tournament strategy. The strategy performs well only when additional votes provide the ability to rank the items in a set. Assuming that votes are distributed randomly among item pairs, the Complete Tournament strategy is able to order the set only when most items in the set are compared against each other. Note that Figure 6.8(right) is only an illustration of the interaction between the number of top items selected, and the redundancy of votes. The results will vary depending upon worker accuracy and the vote budget b.

6.4 Related Work

As far as we know, we are the first to address the Next Votes Problem, and there is no relevant literature that directly addresses this problem. Thus, in this section, we review work related to the Judgment Problem. The algorithms and heuristics we presented for the Judgment Problem are primarily drawn from three diverse topic areas: paired comparisons, social choice, and ranking.

The Judgment Problem has its roots in the *paired comparisons* problem, first studied by statisticians decades ago [70, 116]. In the paired comparisons problem, given a set of pairwise observations regarding a set of items, it is desired to obtain a ranking of the items. In contrast, in the Judgment Problem, we are interested in predicting the maximum item.

The Judgment Problem also draws upon classical work in the economic and social choice literature regarding *Winner Determination* in elections [146, 193]. Numerous voting rules have been used (Borda, Condorcet, Dodgson, etc.) to determine winners in elections [167]. The voting rules most closely related to our work are the Kemeny rule [115] and Slater rule [172]. A *Kemeny permutation* minimizes the total number of pairwise inconsistencies among all votes, whereas a *Slater permutation* minimizes the total number of pairwise inconsistencies in the majority-vote graph [57]. An item is considered a *Kemeny winner* or *Slater winner* if it is the greatest item in a Kemeny permutation or Slater permutation.

We believe our ML formulation is more principled than these voting rules, since ML aggregates information across all possible permutations. For example, in the graph of Figure 6.1, while C and D are both admissible solutions for the Kemeny rule, ML returns D as an answer, since D has almost one and a half more times the probability of being the maximum item compared to C. No prior work about the Judgment Problem, to our knowledge, uses the same approach as our ML formulation.

In the recent social choice literature, the research most closely related to ours has been work by Conitzer et al. regarding Kemeny permutations [60]. Conitzer has studied various voting rules and determined for which of them there exist voter error models where the rules are ML estimators [61]. In our study, we focused upon the opposite question: for a specific voter error model, we presented both Maximum Likelihood, as well as heuristic solutions, to predict the winner.

Our work is also related to research in the theory community regarding ranking in the presence of errors [29, 117] and noisy computation [30, 84]. Both Kenyon et al. and Ailon et al. present randomized polynomial-time algorithms for feedback arc set in tournament graphs. Their algorithms are intended to approximate the optimal permutation, whereas we seek to predict the optimal winner. Feige et al. and Ajtai et al. present algorithms to solve a variety of problems, including the Max Problem, but their scenarios involve different comparison models or error models than ours.

6.5 Conclusion

In this chapter, we studied the problem of identifying the maximum item in a data set using humans. Using humans to identify the maximum can be quite challenging, and there are many issues to consider. The main reason for the complexity, as we have seen, is that our underlying comparison operation may give an incorrect answer, or it may even not complete. Thus, we need to decide which is the "most likely" max (Judgment Problem), and which additional votes to request to improve our answer (Next Votes Problem).

Our results show that solving either one of these problems optimally is very hard, but fortunately we have proposed effective heuristics that do well. Among the heuristics, we observed significant differences in their predictive ability, indicating that it is very important to carefully select a good heuristic. Our results indicate that in many cases (but not all) our proposed PageRank heuristic is the best for the Judgment Problem, while the Complete Tournament heuristic is the best for the Next Votes Problem.

Our results are based on a relatively simple model where item comparisons are pairwise, and worker errors are independent. Of course, in a real crowdsourcing system these assumptions may not hold. Yet we believe it is important to know that even with the simple model, the optimal strategies for the Judgment Problem and Next Votes Problem are NP-Hard. Furthermore, our heuristics can be used even in more complex scenarios, since they do not depend on the evaluation model.

In the next chapter, we design algorithms for crowd-powered categorization. Unlike the previous chapters, we find that categorization is challenging *even when* workers do not make mistakes.

Chapter 7

Algorithm 4: Categorization

7.1 Introduction

In this chapter, we design algorithms for crowd-powered categorization[1]. Given a data set of items, we want to to categorize each item into classes in a hierarchical taxonomy. As an example, we may have a data set of images, and we may want to categorize each image into the taxonomy shown in Figure 7.1. If the image is that of a Nissan car, but the model is not identifiable, the most suitable category is "**Nissan**". If the model is identifiable as well, say, Sentra, then the most suitable category would be "**Sentra**".

Our goal is to ascertain the most suitable category (which could be anywhere in the taxonomy) by asking the minimum number of categorization questions to humans. The questions we ask are of the form "Is this a/an **X**?", where **X** is a node in the taxonomy. As an example, we may pick **X** to be "**Vehicle**", and we can ask a human for a specific item, "Is this a **Vehicle**?"

Furthermore, in this chapter, we make the assumption that humans do not make mistakes while answering questions. While this is a major assumption, we note that we can leverage optimized filtering strategies (Chapters 3 and 4) to ask multiple humans the same question to ensure a certain degree of accuracy, while not incurring a high cost. Additionally, as we show in this chapter, categorization is computationally challenging even without considering human mistakes.

Since humans do not make mistakes while answering categorization questions, in the Nissan car example above, receiving a YES answer to "Is this a **Car**?" says that the most suitable category, in this

[1]This chapter is a simplified version of our paper [155], published at VLDB 2011, written jointly with Anish Das Sarma, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. The paper, studies human-assisted graph search, a generalization of categorization.

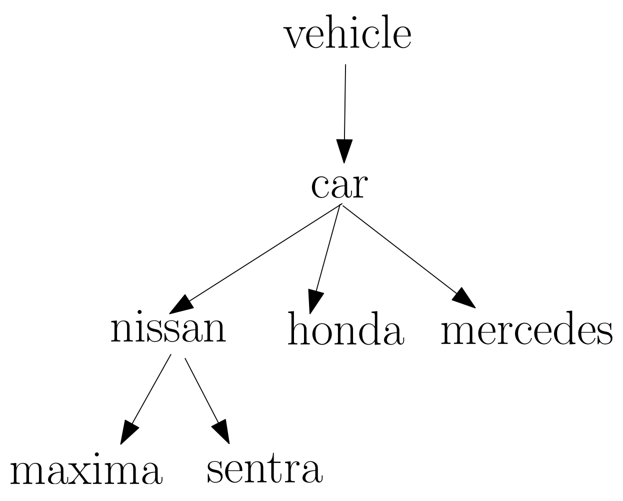


Figure 7.1: Categorization

case **Nissan**, is "reachable" from the category **Car** via a directed path in the taxonomy of Figure 7.1. Also, asking a question at the root, **Vehicle**, (a general question) gives a YES answer while asking a question at a leaf, **Maxima** (a specific question), gives a NO answer. If the image is that of a car, but the model is not identifiable, then asking a question at **Vehicle** and **Car** will yield YES, while all other questions will receive a NO answer.

There are several interesting properties that make categorization a nontrivial problem. First, the answers to different categorization questions (i.e., questions of the form "Is this an **X**?") may be correlated, e.g., if the answer to the categorization question corresponding to (or at) a node in the taxonomy is YES, then the answer to a categorization question corresponding to an ancestor of that node will be YES as well. Therefore, it is possible to identify the target categories without asking categorization questions for all nodes in the graph. Second, the location of a node affects the amount of information that can be obtained from the corresponding categorization question. Asking categorization questions at nodes close to leaves (very *specific* questions) are more likely to receive a negative answer, while asking questions at nodes close to roots (very *general* questions) are more likely to receive positive answers. Asking categorization questions at the "middle" nodes may give more information. In this sense, the categorization problem is similar to the 20 questions game[2], where very specific or very general questions do not help.

An additional challenge stems from how we use humans. Ideally, we would like to issue one question at a time, selecting the next categorization question based on the answers to previous questions.

[2] 20 questions is a two player game, where one player thinks of an object, person or place and the other player has to guess the identity of that item by asking the first player up to 20 YES/NO questions.

However, since latencies of crowdsourcing marketplaces can be large, we may prefer to issue several questions in parallel in one phase, whose answers are then combined to solve the task at hand. The challenge, therefore, is to reason about the possible answers for categorization questions at different nodes in the graph, and to select the set of questions in order to infer as much information as possible about the target categories *across all possibilities*. In this chapter, we focus on optimizing the set of questions asked in a single phase. In Section 7.6 (Experiments) we briefly look at a hybrid approach, where we ask some questions in one phase, examine the results, and then ask additional questions in subsequent phases.

Note that in Chapter 3, we discussed the generalization of filtering to multiple categories (beyond boolean), e.g., filtering an item based on colors—red, blue, green. Here, unlike that generalization, the categories are inter-related, i.e., finding out that an item falls under the Nissan car category automatically implies that the item falls under the Vehicle category. We leverage these inter-relationships between categorization questions to select questions that give us the maximum amount of "information" possible. Furthermore, unlike that generalization, the number of categories in this case can be very large, even as many as millions of nodes.

7.1.1 Outline of Chapter

Here is the outline for this chapter:

- We delineate three orthogonal problem dimensions of the categorization problem. (Section 7.2)
- We formally define the categorization problem. (Section 7.3)
- We present algorithms and complexity results for the problem for the three dimensions identified in Section 7.2. We show that while the general problem is computationally hard, the more constrained variants are tractable. (Sections 7.4 and 7.5)
- We study the performance of our algorithms versus others for webpage categorization. (Section 7.6)

7.2 Dimensions

We will study variants of the categorization problem along three orthogonal dimensions.

Dimension 1: Single/Multi.

The first dimension controls the characteristics of the size of the target set of categories. If it is known that there is a single (best) target category, then we will use the Single variant, that constraints the target set to be a single node (i.e., a single category), else we will have to use the Multi variant.

For instance, an image that contains a Honda car as well as a Nissan car, may have to be categorized under both **Nissan** and **Honda**.

Dimension 2: Bounded/Unlimited.

The second dimension controls the number of questions that can be asked in parallel in one phase. In the Bounded case, we are given a budget b for the total number of questions that can be asked and we want to compute a node set $\mathcal{Q}$, $|\mathcal{Q}|\leq b$, at which to ask categorization questions such that we narrow down the candidates for the target categories as much as possible. The Unlimited case does not put a bound on the number of questions. In this case, we want to compute the minimal set of nodes to ask questions such that we precisely identify the target categories, all in one phase.

The Bounded case is relevant when have a cost constraint for amount of questions asked in one phase, and we want to minimize the uncertainty in the identities of the target categories, while the Unlimited case is relevant when cost is not as important as latency (that is, we must complete categorization in one phase).

Dimension 3: DAG/Downward-Forest.

The third dimension controls the type of taxonomy on which we perform the search for the target categories. We consider a general DAG, as well as a “downward forest” structure, where there are several trees with edges directed from parents to children.

Note that categorization is, in fact, an instance of a more general problem that we call HumanGS, which has many more applications beyond categorization. These applications, as well as consideration of additional dimensions beyond those relevant for categorization may be found in [155]. All of these applications involve humans in the loop in some form (either as crowd workers, or as users interacting with the system).

7.2.1 Summary of Results and Outline

Table 7.1 summarizes our results on the complexity of the examined categorization variants. The details of the analysis and the corresponding algorithms are given in the following sections. The presentation is organized in two sections based on Dimension 1: Single is covered in Section 4, and Multi is covered in Section 5. Each row in the table corresponds to a subsection in the corresponding section. In each case, we first provide a formal definition of the corresponding categorization problem, followed by the complexity analysis for the two graph structures.

		DAG	**Downward-Forest**
Single	Bounded	NP-Complete(n, b), $O((2n)^b n^2 b)$	$O(n \log n)$
	Unlimited	$\min\{\log^2 o, \log n\} \times$ Single-Bounded	$O(1)$
Multi	Bounded	NP-Hard(n, b), $\Sigma_2^P(n, b)$	$O(lb^2 n^6)$
	Unlimited	$O(1)$	

Table 7.1: Summary of Results: b is the budget of questions, $n = |V|$, l is the arity of the tree or forest, and o is the size of the optimal $\mathcal{Q}$.

7.3 The Categorization Problem

In categorization, like in other crowd-powered algorithms, we select a set $\mathcal{Q}$ of questions to ask humans, all in one phase. After human workers provide the answers to these questions, we will be able to infer either the target node(s) (i.e., categories) or a superset of the target nodes. We may then ask additional questions in subsequent phases until the target nodes are found. We focus on optimizing a single phase. We now describe the problem more formally.

We are given the taxonomy as a directed acyclic graph $G = (V, E)$. We use $n \equiv |V|$ to denote the number of nodes (i.e., categories) in the graph. Note that we redefine n here to mean the number of nodes in the taxonomy rather than the number of items, as we have used in previous sections. A node $v \in V$ is *reachable* from another node $u \in V$ if there exists a directed path from u to v. The *reachable set* of u, denoted $\mathsf{rset}(u)$, contains all nodes that are reachable from u, including u. For instance, the reachable set of Nissan in Figure 7.1 is $\{$Nissan, Maxima, Sentra$\}$. The *preceding set* of u, denoted $\mathsf{pset}(u)$, contains all $v \neq u$ such that $u \in \mathsf{rset}(v)$. For instance, the preceding set of Nissan in Figure 7.1 is $\{$Vehicle, Car$\}$. We say that u and v are *unrelated* if there is no directed path between them, i.e., $u \notin \mathsf{pset}(v) \cup \mathsf{rset}(v)$.

We assume that there is a node-set $U^* \subseteq V$, termed the *target set of categories* or simply *target set*, that comprises the *target nodes* (i.e., the individual target categories). The target set must satisfy the following property:

Independence Property: No two nodes in U^* are related.

This property holds in categorization: intuitively, if there are two nodes u and $v \in U^*$ such that $u \neq v$ and $u \in \mathsf{rset}(v)$, then v can be discarded because u "subsumes" v. For instance, in Figure 7.1, if an image falls under the 'Vehicle' category as well as the 'Nissan' category, then we would prefer to retain just the 'Nissan' category in U^* instead of 'Vehicle' because 'Nissan' subsumes 'Vehicle'.

We can informally describe the categorization problem as computing a set of nodes $\{u_1, \ldots, u_b\}$ such that the answers to the corresponding *categorization questions* at the set of nodes lead to the identification of U^*. (Recall that a categorization question at a node u corresponds to the question "Is this item of type u?".) Each categorization question corresponds to a node in the graph, and hence we interchangeably use "asking a question", "asking a question at a node" and "asking a node". Asking a question is defined formally as follows.

Definition 7.3.1 (Asking a Question $\mathsf{q}(u, U^*)$) *Asking a* question *at node $u \in V$, denoted as* $\mathsf{q}(u, U^*)$, *returns YES if $\mathsf{rset}(u) \cap U^* \neq \emptyset$, and NO otherwise.*

In other words, $\mathsf{q}(u, U^*)$ returns YES iff a directed path starting at u reaches at least one node in U^*. Note that u may itself be in the target set. Also note that for any U_1^* and U_2^*, $U_1^* \neq U_2^*$, both of which satisfy the independence property, there is some node at which asking a question would give different answers. (Consider u such that $u \in U_1^*$ and $u \notin U_2^*$. Either there is no $v \in \mathsf{rset}(u)$ present in U_2^*, in which case asking a question at u would give different answers. Or, there is such a $v \neq u$, in which case asking a question at v would give different answers.)

A solution to the categorization problem always exists, as we can identify U^* by asking questions at every node in V. However, it is not necessary to ask questions at every node, as the following trivial lemma illustrates.

Lemma 7.3.2 (DAG Property) *If $\mathsf{q}(u, U^*)$ is YES, then $\mathsf{q}(v, U^*)$ is YES for every v in $\mathsf{pset}(u)$. Conversely, if $\mathsf{q}(u, U^*)$ is NO then $\mathsf{q}(v, U^*)$ is NO for every node v in $\mathsf{rset}(u)$.*

Let $\mathcal{Q} \subseteq V$ be some set of nodes at which we ask categorization questions. In general, the answers to these questions may not be sufficient to precisely identify U^*, since there may be other nodes (not in U^*) for which the answers to the categorization questions asked would be the same even if those nodes *were* in U^*. We introduce the notion of a *candidate set* to capture the possibilities for U^* based on questions on a node-set $\mathcal{Q}$. The candidate set, denoted as $\mathsf{cand}(\mathcal{Q}, U^*)$, is the maximal set of nodes that we cannot distinguish from U^* based solely on the answers to questions at the nodes in $\mathcal{Q}$. In other words, the nodes in $\mathsf{cand}(\mathcal{Q}, U^*)$ have the same reachability properties (with respect to $\mathcal{Q}$) as the nodes in U^*. For $|\mathcal{Q}| = 0$, we have the trivial result that $\mathsf{cand}(\mathcal{Q}, U^*) = V$. We first consider how asking a single question (i.e., $|\mathcal{Q}| = 1$) allows us to restrict the contents of the candidate set beyond V.

Theorem 7.3.3 (One Question Pruning) *Assume that we ask a single question at node u. The candidate set is computed as follows, based on the answer and the variant of Single/Multi that the categorization instance falls under.*

$$cand(\{u\}, U^*) = \begin{cases} V - rset(u) & q(\{u\}, U^*) = \text{NO} \\ V - pset(u) & q(\{u\}, U^*) = \text{YES} \wedge Multi \\ rset(u) & q(\{u\}, U^*) = \text{YES} \wedge Single \end{cases}$$

Proof 7.3.4 *If we get a NO on asking a question at a node u, then none of $rset(u)$ can be present in $cand(\{u\}, U^*)$, but any other node could be a target node, therefore $cand(\{u\}, U^*) = V - rset(u)$. Suppose we get YES, so a target node exists in $rset(u)$. If there are one or more target nodes ($|U^*| \geq 1$, as in Multi), the independence property implies that no target node can be present in $pset(u)$. Hence, $cand(\{u\}, U^*) = V - pset(u)$. Furthermore, if we know that there exists a single target node, i.e., $|U^*| = 1$, as in Single, then the candidate set is simply $rset(u)$.*

Given this base case of one question, we can compute $\text{cand}(\mathcal{Q}, U^*)$ for a general node-set $\mathcal{Q}$ as the intersection of the candidate sets resulting from individual questions.

Theorem 7.3.5 *After asking questions at all nodes in a node-set $\mathcal{Q}$, we have:*

$$cand(\mathcal{Q}, U^*) = \bigcap_{u \in \mathcal{Q}} cand(\{u\}, U^*)$$

.

Proof 7.3.6 *Let $C = \bigcap_{u \in \mathcal{Q}} cand(\{u\}, U^*)$. If $x \notin cand(\{u\}, U^*)$ for nodes $x \in V$ and $u \in \mathcal{Q}$, then it is easy to see that x cannot be present in $cand(\mathcal{Q}, U^*)$. Now consider a node $x \in C$. For the Multi variant, consider a set U that contains x, no ancestors or descendants of x, as well as all other nodes in C that do not have descendants in C. Note that U satisfies the independence property. If U were the target set, it would give rise to the same answers to questions at $\mathcal{Q}$ as U^*. Thus, x could be in U^* and has to be present in $cand(\mathcal{Q}, U^*)$. For the Single variant, consider a set $U = \{x\}$. U would give rise to the same answers to questions at $\mathcal{Q}$ as U^*. Thus, x could be in U^* and has to be present in $cand(\mathcal{Q}, U^*)$.*

Thus, each question may enable some additional pruning of the candidate set, and the order in which questions are asked does not affect the final result. As an example, let us consider again the categorization problem illustrated in Figure 7.1. Suppose that the single target node is *Maxima*, and assume that we ask questions at $\mathcal{Q} = \{Car, Nissan, Mercedes\}$. Clearly, the questions at *Car* and *Nissan* yield YES, whereas the question at *Mercedes* yields NO. Based on these answers, we can assert that $\mathsf{cand}(\mathcal{Q}, U^*) = \{Nissan, Maxima, Sentra\}$. The candidate set contains the target node as well as two "false positives" (*Nissan* and *Sentra*). Picking $\mathcal{Q}$ so as to minimize the number of false positives is the goal of the algorithms that we present later.

Since we are operating in a single phase setting, we are interested in minimizing the size of the candidate set in the worst case. Given that U^* is unknown, we may use the maximum size of $\mathsf{cand}(\mathcal{Q}, U^*)$ (under all admissible possibilities for U^*) as an indication of the worst-case uncertainty that remains after asking the questions in $\mathcal{Q}$. We use $\mathsf{wcase}(\mathcal{Q})$ to denote this worst-case size. A natural objective is to select $\mathcal{Q}$ so that $\mathsf{wcase}(\mathcal{Q})$ is minimized. We define wcase formally in Section 7.4 for Single and in Section 7.5 for Multi.

7.4 Single Target Category

In the Single problem, we have the constraint that there is a single target node, i.e., $|U^*| = 1$. Let this node be u^*. To simplify notation for Single, we use $\mathsf{cand}(\mathcal{Q}, u^*)$, instead of $\mathsf{cand}(\mathcal{Q}, \{u^*\})$, to denote the candidate set after questions have been asked at the node set $\mathcal{Q}$, and we use $\mathsf{q}(u, u^*)$, instead of $\mathsf{q}(u, \{u^*\})$, to denote the answer to asking a question at u. Recall that as in Theorem 7.3.3, asking a question at u for the Single problem tells us whether the candidate set $\mathsf{cand}(\{u\}, u^*)$ is $\mathsf{rset}(u)$ (if the answer is YES) or $V - \mathsf{rset}(u)$ (if the answer is NO).

Given a node set $\mathcal{Q}$ at which we ask categorization questions, we define the *worst-case candidate set size* as

$$\mathsf{wcase}(\mathcal{Q}) = \max_{u_i \in V} |\mathsf{cand}(\mathcal{Q}, u_i)| \tag{7.1}$$

In other words, wcase computes the size of the largest candidate set when the target node could be any node in V.

7.4.1 Single-Bounded

In the Single-Bounded variant we have a fixed budget b on the number of questions that may be asked, i.e., the size of $\mathcal{Q}$ cannot exceed b. The goal is to pick the set of nodes $\mathcal{Q}$ such that the worst-case candidate set size is minimized.

Definition 7.4.1 (Single-Bounded) *(Bounded Search for a Single target node.) Given a parameter b and the restriction that $|U^*| = 1$, find a set $\mathcal{Q}$ of nodes $\mathcal{Q} \subseteq V$ to ask questions such that $|\mathcal{Q}| = b$ and* *wcase*$(\mathcal{Q})$ *is minimized.*

The following subsections examine the complexity of the problem under the different possibilities for the structure of G, i.e., general DAG and downward-forest.

Single-Bounded: DAG

We begin with the auxiliary result that wcase$(\mathcal{Q})$ can be computed in time polynomial in the number of nodes in the graph. This result is used later to bound the complexity of Single-Bounded.

Theorem 7.4.2 (Computation of Worst Case) *Given a node set $\mathcal{Q}$ at which questions are asked, wcase$(\mathcal{Q})$ can be computed in $O(n^2 \cdot b)$, where $n = |V|$.*

Proof 7.4.3 *In time $O(n^2)$, for all pairs of nodes a and b in G, we can compute and maintain whether or not $a \in$ rset(b). Let u^* be the target node in the graph. In time $O(|\mathcal{Q}|)$, we can compute the answers to each of the questions asked at the nodes in $\mathcal{Q}$. Subsequently, we can compute cand$(\mathcal{Q}, u^*)$ in $O(n \cdot |\mathcal{Q}|)$ by checking the following for every node u: if there is a node in $u' \in$ rset$(u), u \neq u'$ such that $u' \in \mathcal{Q}$ and u' returned YES, or if there is a node $u \in$ pset(u), such that $u \in \mathcal{Q}$ and u' returned NO, then $u \notin$ cand$(\mathcal{Q}, u^*)$, else $u \in$ cand$(\mathcal{Q}, u^*)$. We can compute wcase by repeating the above procedure for every possible u^*, taking a total of $O(n^2 \cdot |\mathcal{Q}|)$ time.*

The main idea above is to first compute all pairs (a, b) in $V \times V$ such that $b \in$ rset(a) and then use this information to compute cand$(\mathcal{Q}, u^*)$ for every possibility of u^*. Using the result above, we can define a brute-force approach to solving Single-Bounded, by considering all possible combinations of $\mathcal{Q}$ with size at most b.

Lemma 7.4.4 (Brute-force Solution) *The optimal solution of Single-Bounded for any DAG can be found in $O(n^b \cdot n^2 \cdot b)$, where n is the number of nodes in V.*

Proof 7.4.5 *We find wcase for each choice of $\binom{n}{b}$ nodes at which questions are asked. The choice for which the worst-case candidate set is the smallest is the optimal solution.*

Clearly, we can solve Single-Bounded optimally in PTIME if b is bounded by a constant. However, the appearance of b in the exponent hints at the hardness of the problem in the general case. Indeed, the following result shows that Single-Bounded is computationally hard. The proof shows a reduction to Single-Bounded from the NP-hard max-cover problem [133].

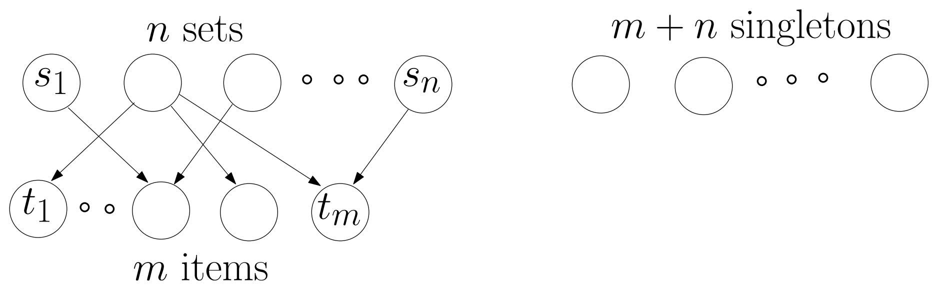


Figure 7.2: Hardness Proof for Single-Bounded

Theorem 7.4.6 *Single-Bounded cannot be solved in polynomial time unless $P = NP$.*

Proof 7.4.7 *We prove that the decision version of the problem is NP-complete, stated as follows: Given a budget b and a positive integer m, is there a node-set Q to ask questions such that wcase$(Q) \leq m$? We refer to this problem as Single-Bounded-Decision.*

We use a reduction from the NP-complete max-cover problem *[133]. In this problem, the objective is to pick a certain number of sets such that they cover as many items as possible. (A set containing a given item is said to cover that item.) Let there be m items and n sets in the max cover problem. We need to select b sets such that the maximum number of items are covered.*

We reduce the max-cover problem to Single-Bounded-Decision with the following directed acyclic graph. (An instance of the graph is shown in Figure 7.2.) Consider nodes arranged in two layers. In the first layer, we have one node corresponding to each set s_i in the max cover problem. In the second layer, we have a node corresponding to each item t_i in the max-cover problem. There is a directed edge from the node corresponding to set s_i to the node corresponding to the item t_j iff item t_j is present in set s_i. In addition, we include $n + m$ singleton nodes (nodes with no incoming or outgoing edges) in the DAG. Subsequently, we call Single-Bounded on this DAG with a budget of b questions. The worst-case candidate set corresponds to each of the questions in Q receiving NO answers. To see this, note that if we get a YES answer to a question asked at any of the nodes corresponding to sets, the candidate set is $\leq m$. Receiving a YES answer to any question asked at the nodes corresponding to items or singletons will give a candidate set of 1. However, if we get all NO answers, the number of nodes remaining in the worst-case candidate set is at least $m + n$, even if all the nodes corresponding to items as well as b from the nodes corresponding to sets and singletons are eliminated from the candidate set. Additionally, the solution of the Single-Bounded problem, i.e., Q, will only contain nodes corresponding to sets, because those nodes exclude the maximum number of nodes from the candidate set, in the worst case. Thus, Single-Bounded picks nodes corresponding to sets such that the maximum number of nodes corresponding to *items are covered. Therefore, the solution to Single-Bounded corresponds to a max-cover. Conversely, every solution of the max-cover problem can be written as a solution for Single-Bounded.*

In addition, given that we can compute wcase(Q) in PTIME (see Theorem 7.4.2), a solution for Single-Bounded-Decision can be verified in PTIME. Thus, Single-Bounded-Decision on DAGs is in NP and is NP-Complete.

Even though the general problem is intractable, it may be possible to find efficient solutions by leveraging specific characteristics of the input, and in particular of the DAG G. The following subsections examine this hypothesis for downward-forests.

Single-Bounded: Downward-Forest

In the *downward-forest* case, G is a forest of directed trees with edges from parents to children nodes (see also Section 7.2.)

We begin by showing that Single-Bounded on a downward-forest can be reduced to Single-Bounded on a downward-tree (a tree with edges from parents to children), by attaching a virtual root node that links all the trees in the forest. The following theorem is proved by showing that the optimal solution for the resulting tree gives a solution to the forest with wcase at most one more than optimal.

Theorem 7.4.8 (Downward-Forest ⇒ Tree) *Given a downward-forest G_F, there exists a downward-tree G_T, such that a solution Q_T to categorization on G_T gives a node set Q'_T of G_F such that wcase(Q'_T) $\leq$ wcase(Q) + 1, for any node set Q of G_F where $|Q|, |Q'_T| \leq b$.*

Proof 7.4.9 *We augment the downward-forest with a single root node such that there exists an edge from the new root node to the root of each of the trees in the directed forest. Let the original forest be G_F and the new augmented tree (a downward-tree) be G_T.*

Let Q_T be an optimal selection of b nodes from G_T at which questions are asked. We first convert Q_T into Q'_T such that there is no question asked at the root. (We delete the root from Q_T, if present.) Since a question at the root has to return a YES answer, the worst-case candidate set at G_T, denoted by wcase$_T$, will be unchanged. We therefore have wcase$_T(Q'_T)$ = wcase$_T(Q_T)$.

Let Q_F be the optimal solution on G_F. If we select set Q_F as the questions to be asked at G_T, we get wcase$_T(Q_T) \leq$ wcase$_T(Q_F)$, since Q_T gives the optimal worst-case candidate set for G_T. Also, since there is an addition of at most one node to the worst-case candidate set when Q_F is used on G_T instead of G_F, wcase$_T(Q_F) \leq$ wcase$_F(Q_F)+1$, where wcase$_F$ is the worst case candidate set when the questions

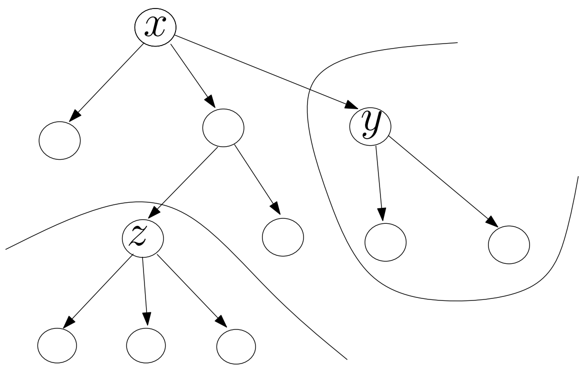


Figure 7.3: Partition Example

are asked at the nodes in G_F. Now, apply Q'_T on G_F. We then have $\mathsf{wcase}_F(Q'_T) \le \mathsf{wcase}_T(Q'_T) = \mathsf{wcase}_T(Q_T) \le \mathsf{wcase}_F(Q_F) + 1$. Thus, choosing the questions to minimize worst-case candidate set in the tree gives the optimal worst-case candidate set for the forest plus at most one more node. We ignore the additive factor of 1 in our calculations.

We can therefore focus on solving the Single-Bounded problem for a single downward-tree, instead of a downward-forest, ignoring the additive constant of ≤ 1. We show that this problem is equivalent to the *partition problem* [122], which admits efficient solutions.

Definition 7.4.10 (Partition Problem) *Given an undirected tree, find b edges such that their deletion minimizes the size of the largest connected component.*

To show the equivalence, we first define how a chosen set Q induces a partition of the tree into connected components.

Definition 7.4.11 (Partition on a Node Set) *In a downward-tree, we recursively define the partitions on a node set Q, denoted $P(Q)$, as the following:*

- *If $x \in Q$ and none of x's descendants are in Q, then the subtree under x (including x) is a partition. (We call this partition the partition of x.)*
- *If $x \in Q$ and some of x's descendants are in Q, then the subtree under x (including x) excluding all of the partitions of x's descendants is a partition. (We call this partition the partition of x.)*
- *Whatever is left after all the partitions are formed is a partition. (If x is the root of the remainder of the tree, then the partition is called the partition of x.)*

We have exactly $|Q|$ or $|Q|+1$ partitions. As an example, consider Figure 7.3. Here there are three partitions corresponding the node set $Q = \{y, z\}$, i.e., the partition of x, y and z.

Lemma 7.4.12 (Candidate Set Partition) *Given questions asked at a node set Q, the candidate set $\mathsf{cand}(Q, u^*)$ for any u^* corresponds to one of the partitions from $P(Q)$.*

Proof 7.4.13 *If $u^* \in$ partition p of x, then we prove that $\mathsf{cand}(Q, u^*) = p$. First, we consider the case when $x \in Q$. In this case, notice that the question at x returns a YES answer. Using Theorem 7.3.3, the candidate set can be restricted to the subtree under x. The only nodes $\in Q$ at which the answer is YES are those that are ancestors of x, but they do not change the candidate set. Those nodes $\in Q$ that are descendants of x or unrelated to x all return NO, since there is no path from those nodes to the target node u^*. Since all those nodes return NO, the subtrees under those nodes can be removed from the candidate set. Thus, the candidate set is precisely p.*

If $x \notin Q$, then x is the root of the directed tree. Here, once again, if we asked a question at x, we would obtain a YES answer. Those nodes $\in Q$ that are descendants of x all return NO, since there is no path from those nodes to the target node u^*. Since all those nodes return NO, the subtrees under those nodes can be removed from the candidate set. Thus, the candidate set is once again p.

The previous result essentially establishes the equivalence between the two problems, as our goal is to minimize $\mathsf{wcase}(Q)$, which is equivalent to the size of the largest partition that can be induced by Q. Since the partition problem can be solved in PTIME, it follows directly that the same holds for Single-Bounded on a downward-tree. The following theorems formalize these observations.

Theorem 7.4.14 (Partition Problem Equivalence) *The problem of Single-Bounded on downward-trees is equivalent to the partition problem.*

Proof 7.4.15 *As seen in Lemma 7.4.12, the candidate set on asking a question at any node corresponds to one of the partitions induced by asking questions. Thus, in order to minimize wcase, it is sufficient to solve the partition problem on the downward-tree. The size of the largest partition is the size of the worst possible candidate set. Conversely, every instance of the partition problem can be cast as an instance of the Single-Bounded problem for a downward-tree. Thus, the two problems are equivalent.*

Using a dynamic programming algorithm from [122] for the partition problem, we obtain the following result for Single-Bounded:

Theorem 7.4.16 (Single-Bounded) *There exists an algorithm with complexity $O(n \log n)$ that solves Single-Bounded on a downward-forest.*

Proof 7.4.17 *The algorithm solves the equivalent partition problem on the downward-tree formed from the downward-forest augmented with a root node. Reference [122] gives a dynamic programming algorithm that finds the minimal number of edges that need to be cut in order to achieve a certain partition size in $O(n)$. We run binary search over partition sizes in order to find the smallest partition size for which the minimal number of edges to be cut is $\leq b$.*

7.4.2 Single-Unlimited

In the Single-Unlimited problem, we do not have a strict budget on the number of questions that can be asked; instead, we want to find the smallest set of questions $\mathcal{Q}$ such that the target node is uniquely determined in the every case.

Definition 7.4.18 (Single-Unlimited) *(Unlimited Search for a single target node) Given that the target set $|U^*| = 1$, find the smallest set $\mathcal{Q} \subseteq V$ to ask questions such that $\mathsf{wcase}(\mathcal{Q}) = 1$.*

First, we illustrate in the following example that the number of questions that are required to ensure that $\mathsf{wcase}(\mathcal{Q}) = 1$ can vary widely depending on the structure of the underlying graph G. Subsequently, we study various structures of G and explore algorithms to solve Single-Unlimited.

Example 7.4.19 *There exist connected graphs for which the smallest node-set $\mathcal{Q}$ is almost all the nodes. Consider a downward-tree with one parent and n immediate children. Suppose that we leave two of the n children unasked, and that one of these is the target node. Then the unasked node will still be present in the candidate set. Hence, we need to ask $n - 1$ nodes in this scenario.*

Also, there are graphs for which $|\mathcal{Q}|$ is $O(\log n)$. Consider a graph with nodes in two levels, r nodes in the first level, and 2^r nodes in the second level. We add directed edges from the nodes in the first level to those in the second level, as follows: There is an edge from the j-th node in the first level to the i-th node in the second level (always counting from the left) if and only if there is a 1 in the j-th position of the binary encoding of i.

In this case, if we ask all r nodes in the first level, we only need to ask at most r additional nodes in the second level (precisely those corresponding to $1, 2, 4, \ldots, 2^{r-1}$, since each of these r additional nodes in the second level has a single parent, and so if the parent returns YES, then it is unclear if the node or its parent is the target node). All other nodes in the second level have a unique encoding with respect to the r nodes in the first level. Thus the answers returned by the r parent nodes are sufficient to infer whether or not one of these nodes in the second level is the target node. Thus, the size of $|\mathcal{Q}| = O(\log n)$ for this graph.

Single-Unlimited: DAG

Our first result is a general scheme to solve Single-Unlimited for any G using our results from Single-Bounded: By repeating Single-Bounded with various values of b, it is possible to derive a solution for Single-Unlimited, formalized in the theorem below.

Theorem 7.4.20 (Repeating Single-Bounded) *If there is an algorithm that solves Single-Bounded on a DAG in time T, then Single-Unlimited on the same DAG can be solved in $O(K \times T)$, with $K = \min\{\log^2 o, \log n\}$, where o = size of the optimal $\mathcal{Q}$.*

Proof 7.4.21 *The proof involves repeating runs of Single-Bounded to find the smallest $\mathcal{Q}$ such that the worst-case candidate set is of size 1. Below we show two approaches, that are combined by running one step of each until one of them terminates.*

The first approach involves binary search. Here, we let $|\mathcal{Q}|$ vary between 1 and n, run Single-Bounded and stop once the worst case candidate set size is 1 for $|\mathcal{Q}| = b$, but not when $|\mathcal{Q}| = b - 1$. This process takes $O(\log n)$ time.

The second approach involves doubling $|\mathcal{Q}|$ each time starting from 1, i.e., $|\mathcal{Q}| = 1, 2, 4, 8, \ldots$, and running Single-Bounded every time. Once we find a b such that Single-Bounded when $|\mathcal{Q}| = b$ has worst case candidate set 1, then we repeat the procedure of doubling from $|\mathcal{Q}| = b/2$, i.e.,$|\mathcal{Q}| = b/2 + 1, b/2 + 2, b/2 + 4, \ldots$, recursively, until we find two consecutive $|\mathcal{Q}|$ such that one of them has worst case candidate set of 1 and the other one does not. Thus, we will do $\log o$ steps every time, and we reduce the space of search by 1/2 every time, leading to $\log o$ such steps. Thus, we have a complexity of $\log^2 o$.

The approach is to run Single-Bounded repeatedly using either binary search on n, or by doubling b until we locate the smallest $\mathcal{Q}$ for which wcase = 1.

Single-Unlimited: Downward-Forest

On a downward-forest, Single-Unlimited can be solved in $O(n)$:

Theorem 7.4.22 (Downward-Forest) *On a downward-forest, we need to ask almost all nodes (except at most one) in order to solve Single-Unlimited.*

Proof 7.4.23 *Consider all leaf nodes across all trees in the forest. Each leaf node has a single parent. Firstly, it is easy to see that we need to ask all leaves. If not, consider a leaf node a. Let its parent be b. If we do not ask a, but b returns YES, (but no other child of b returns YES) then it is not clear if the target node is a or b. Therefore, we need to ask a. The same argument can be used for all leaves. Now* *assume all questions at leaves return a NO answer (effectively, leaves can be removed from the tree.) The argument can be repeated for each parent of the leaves as well. We then arrive at the roots of each of the trees in the forest and their children. We cannot leave two of the roots unasked because we cannot distinguish between them without asking a question at both of them. However, we can avoid asking a question at one of the roots because if we get a NO response from all other trees as well as children of that node, then that node has to be the target node.*

The main insight in the proof is that if we leave a node unasked, then on getting a NO answer from all the children of the node and a YES answer from the parent of the node, we have wcase > 1 because we cannot distinguish between the node and its parent. However, in the general case of several trees in the forest, one of the roots need not be asked. This is because there is a single target node, and if all children of this root as well as everything else returns NO, then this root has to be the target node.

7.5 Multiple Target Nodes

In the Multi version of the problem, there exists a *target set* $U^* \subseteq V$, which denotes the unknown set of nodes we wish to discover by asking questions. Unlike the Single case, the size of the target set can be any $|U^*| \geq 1$ and is unknown. The only constraint we are given is that the target nodes satisfy the independence property, i.e., no two nodes in U^* are related.

Computation of the candidate set for the Multi problem can be found in Section 7.3. To recap, a single question at u lets us exclude from the candidate set either $\mathsf{pset}(u)$ (if the answer is YES) or $\mathsf{rset}(u)$ (if the answer is NO).

To incorporate the independence property in defining a worst-case candidate set, we define the function ip on a set of nodes to return *true* if and only if the set of nodes satisfy the independence property. Given a set Q of nodes, we redefine the *Worst-Case Candidate Set* to be:

$$\mathsf{wcase}(Q) = \max_{U \subseteq V,\ ip(U)=\mathsf{true}} |\mathsf{cand}(Q, U)|$$

Next we study the Bounded (Section 7.5.1) and Unlimited (Section 7.5.2) versions of the categorization problem for the Multi case.

7.5.1 Multi-Bounded

In this section we consider the bounded categorization problem for a target set of nodes, formally stated below.

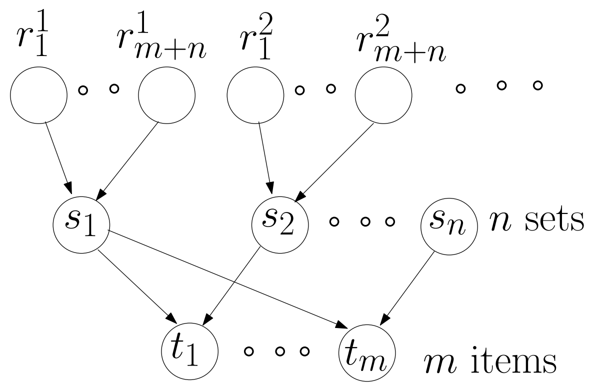


Figure 7.4: Hardness Proof for Multi-Bounded

Definition 7.5.1 (Multi-Bounded) *(Bounded Search for a target set.) Given a parameter b, find a set Q of nodes $Q \subseteq V, |Q| = b$ to ask questions such that $\mathsf{wcase}(Q)$ is minimized.*

We present algorithms and complexity results for Multi-Bounded, examining various properties of the structure of the graph G.

Multi-Bounded: DAG

In this section, we establish the overall complexity of Multi-Bounded for an arbitrary DAG. We first show an NP-hard lower bound, and follow it with an upper bound of Σ_2^P. [3] Bridging the gap between our lower and upper bounds is an open problem.

The following theorem establishes the NP-hardness of Multi-Bounded. As in Single-Bounded, we use the max-cover problem to prove NP-hardness, although the details of our reduction need to be modified for the Multi-Bounded problem.

Theorem 7.5.2 (Lowerbound) *The Multi-Bounded problem is NP-Hard in n and b.*

Proof 7.5.3 *We give a reduction from the NP-hard max-cover problem: Given m items, n sets, and an integer b, the goal is to choose b sets that cover the most number of items.*

Given an instance of the max-cover problem, we construct an instance of Multi-Bounded with the following DAG containing three layers of nodes, as depicted in Figure 7.4. The second and third layers of nodes are identical to the first and second layers in the proof for Theorem 7.4.6: The second layer has one

[3] Σ_2^P corresponds to a class of problems higher than P and NP.

node corresponding to each set and the third layer has a node corresponding to each item. There is an edge from node s_i to the node t_j iff item $t_j \in s_i$ in the max-cover instance. For each node s_i in the second layer, we add $m + n$ unique parents in the first layer, $r^i_1, r^i_2, \dots, r^i_{m+n}$, with an edge to s_i. We want to solve Multi-Bounded *on this constructed DAG for b questions. To solve the* Multi-Bounded *problem, we will always pick nodes corresponding to sets, because they let us eliminate the maximum number of nodes corresponding to items, in the worst case.*

If we pick a node from the first or the third layer, we will get a YES or NO response respectively in the worst case, allowing us to eliminate only one node from the candidate set. On the other hand, a node corresponding to a set allows us to eliminate either $m + n$ nodes (on YES) or the number of nodes corresponding to items covered by that set (on NO). Thus, we only pick nodes corresponding to sets. In the worst case, none of the answers returned are YES. If any of them were, then we would eliminate $m+n$ nodes per YES answer from the candidate set. On the other hand, if those answers were NO, we would eliminate less than m nodes corresponding to items. Thus, the worst-case candidate set occurs when all answers are YES. In order to improve the worst case, we would like to cover as many nodes from the m nodes corresponding to items by picking nodes corresponding to sets.

Thus the nodes corresponding to sets that are picked in Multi-Bounded *precisely correspond to the sets that are picked in the max cover problem.*

The following theorem establishes the upperbound on the complexity of Multi-Bounded.

Theorem 7.5.4 (Upperbound) *The decision version of* Multi-Bounded[4] *is in Σ_2^P.*

Proof 7.5.5 *Given an instance of the decision version of the* Multi-Bounded *problem, we can express it as an instance of Σ_2^P in the following way:*

$$\exists y_1 \forall y_2 [L(y_2) \vee (R(y_1, y_2) < X)],$$

where y_1 corresponds to a set of nodes at which questions are asked, y_2 corresponds to all possible instances of the target set. $L(y_2)$ checks in PTIME whether y_2 contains two nodes with a path from one to the other: If so, it returns YES. $R(y_1, y_2)$ evaluates the candidate set given y_1 and y_2.

[4]The decision version of Multi-Bounded is defined as follows: We want to test whether wcase($\mathcal{Q}$) is reduced to a size below some τ.

Multi-Bounded: Downward-Forest

Next we consider Multi-Bounded for forests of arbitrary trees. We can extend Theorem 7.4.8 to the case of Multi-Bounded, which enables us to focus our attention on trees instead of forests. We then present the main result of this section, providing a PTIME dynamic programming algorithm (Algorithm 7) that solves Multi-Bounded for downward-trees.

The main insight in the algorithm is that if we solve the sub-problem of finding the worst-possible contributions to the overall candidate set while allocating b questions to a node, then these contributions can be combined bottom-up to solve the problem at the parent.

For each node x, for each $i \in \{0, \ldots, b\}$, we generate and populate array $T_x[i]$, which contains a set of worst-case contributions corresponding to each allocation of i questions in total to x and the two children y and z. Each scenario is a 4-tuple, $((b_1, b_2), p_1, p_2, n)$ where the first entry denotes that this 4-tuple arises from an allocation of b_1 questions to the sub-tree under y and b_2 questions to the sub-tree under z. For this particular allocation of questions, p_1 is the contribution to the candidate set such that (a) it contains x and (b) there is a target node below x, p_2 is the contribution such that (a) it does not contain x and (b) there is a target node below x, and n is the contribution if there is no target node below x. Note that there is no r value in each tuple (unlike Single-Bounded) since the presence (or not) of a target node above x in the tree will not change the answers to questions in x's subtree if we already know that x's subtree does/does not contain a target.

Theorem 7.5.6 (DP Algorithm) *There exists an algorithm that solves the Multi-Bounded problem for forests in $O(b^2 \cdot n^6)$.*

Details of Algorithm for Downward Forests:

We now describe the algorithm for solving Multi-Bounded on Downward forests in detail. We describe the algorithm for a tree with arity 2, however our approach generalizes to trees with different arities.

We set $T_x[0] = ((0,0), size(x), 0, size(x))$, since the worst-case contribution to the candidate set when no questions are asked is always the entire tree under x, except 0 in the case of p_2, since the root can never be eliminated when no questions are asked. We define the subtree under x to be $\text{rset}(x)$.

Now, we generate rules to generate a triple (p_a, p_b, n_a) by combining triples from $T_y[b_1] : (p_1, p_2, n)$ and $T_z[b_2] : (p'_1, p'_2, n')$.

Algorithm 7: Multi-Bounded Downward-Forest

```
Data: $\mathcal{G}$ =downward-tree, $b$ = budget
Result: optimal set of $b$ nodes to ask questions to
for all nodes $x$ in $\mathcal{G}$, bottom-up do
    $T_x := \varnothing$;
    $T_x[0] := \{((0,0), size(x), 0, size(x))\}$;
    if $x$ has 1 or 2 children then
        $y$ := left sub-child of $x$;
        $z$ := right sub-child of $x$;
        for $i : 0 \ldots b$ do
            for all $b_1, b_2 : b_1 + b_2 = i$ do
                for all $((*,*), p_1, p_2, n)$ in $T_y[b_1]$ and all $((*,*), p'_1, p'_2, n')$ in $T_z[b_2]$ do
                    $p_a := \max\{p_1 + n', p'_1 + n, p_1 + p'_1, n + n'\} + 1$;
                    $p_b := \max\{p_2 + p'_2, p_2 + p'_1, p'_2 + p_1, p_2 + n', p'_2 + n\}$;
                    $n_a := n + n' + 1$;
                    if $b_1 == 0 \wedge b_2 \neq 0$ then
                        $p_b := p'_2 + n$;
                    if $b_2 == 0 \wedge b_1 \neq 0$ then
                        $p_b := p_2 + n'$;
                    $T_x[i+1] := T_x[i+1] \cup \{((b_1, b_2), p_a, p_b, 0)\}$;
                    if $i \neq 0$ then
                        $T_x[i] := T_x[i] \cup \{((b_1, b_2), p_a, p_b, n_a)\}$;
                compress $T_x[i]$;
    else
        for $i : 1 \ldots b$ do
            $T_x[i] := \{((b, 0), 1, 0, 0)\}$;
$r$ := root of tree;
$t$ := tuple in $T_r[b]$ that has smallest $p_1$;
trace the origin of $t$ until the leaves of the tree;
output the questions;
```

Case 1: None at root, b_1, b_2 nonzero. In this situation, for p_a, none of the answers in any of the sub-trees can be YES. If any answer in either sub-tree is YES, then the root node x is automatically excluded. Therefore, none of the answers is YES, and target nodes can be in the subtree under y or z or both or be x itself.

If there are target nodes under y but none under z, and if none of the answers are YES, we have a contribution of p_1 towards the candidate set from the sub-tree under y. Additionally, the sub-tree under z will make a contribution of n' towards the candidate set (since there are no target nodes under z). Therefore, we have a overall contribution of $p_1 + n' + 1$. Similarly, if there are target nodes under z but not y, then the contribution is $p'_1 + n + 1$. If the target node is x itself, then we get a contribution to candidate set of size $n + n' + 1$, since the target nodes are not present in the sub-trees under y or z.

If there are target nodes under both y and z, then we have a contribution of $p_1 + p'_1 + 1$. Therefore, in total, we have $p_a = \max\{p_1 + n' + 1, p'_1 + n + 1, p_1 + p'_1 + 1, n + n' + 1\}$.

For p_b, at least one of the answers has to be a YES, only then will the root x be eliminated. Thus, the cases that we have are either that the YES came from the sub-tree under y, or the YES came from the sub-tree under z or both. Therefore, at least one component of the contribution to the candidate set must be either p_2 or p'_2. Thus, the contribution could be: $p_2 + p'_2$ (when there are YES answers in both sub-trees), $p_2 + p'_1$, $p'_2 + p_1$, $p_2 + n'$, $p'_2 + n$ (when there are YES answers from only one sub-tree). For n_a, we simply have worst-case contribution to be $n + n' + 1$.

Case 2: None at root, b_1 zero, b_2 nonzero. For p_a, the root will remain iff there is no YES answer to a question in the sub-tree under z. Since we have $p_1 = n = size(y)$, the size of the sub-tree under y, the same equation applies, i.e., the sub-tree under y contributes p_1 or n, while the sub-tree under z contributes p'_1 or n', and x contributes 1.

For p_b, at least one of the answers has to be a YES, only then will the root be eliminated. Thus, a YES has to come from the sub-tree under z. Therefore, z has to contribute p'_2, while y contributes $size(y)$ to the candidate set. For n_a, we have worst-case contribution to be $n + n' + 1$.

Case 3: root, b_1, b_2 nonzero. For p_a, the root will remain in the candidate set iff there is no YES answer to a question under either of the sub-trees. Therefore, the target node can either be the root, or under y or z or both. Thus, we have, as before, $p_a = \max\{p_1 + n' + 1, p'_1 + n + 1, p_1 + p'_1 + 1, n + n' + 1\}$.

For p_b, the answer from the root has to be a YES. Additionally, since the root is eliminated, at least one other answer is a YES. Thus, we have the same potential worst-case contributions: $p_2 + p'_2$ (when there are YES answers in both sub-trees), $p_2 + p'_1$, $p'_2 + p_1$, $p_2 + n'$, $p'_2 + n$ (when there are YES answers from only one sub-tree). For n_a, since we get a NO answer from the root, $n_a = 0$.

Case 4: root, b_1 zero, b_2 nonzero. For p_a, the root will remain in the candidate set iff there is no YES answer to a question under either of the sub-trees. Therefore, the target node can either be the root, or under y or z or both. Thus, we have, as before, $p_a = \max\{p_1 + n' + 1, p'_1 + n + 1, p_1 + p'_1 + 1, n + n' + 1\}$. (The same equation works since $p_1 = n = size(y)$.)

For p_b, the answer from the root has to be a YES. Additionally, since the root is eliminated, at least one other answer is a YES. Thus, we have a contribution of p'_2 from z and a contribution of

$size(y)$ from y. For n_a, since we get a NO answer from the root, $n_a = 0$.

Case 5: root, b_1, b_2 zero For p_a, the root will remain in the candidate set. However, the same equation holds, i.e., $p_a = \max\{p_1 + n' + 1, p'_1 + n + 1, p_1 + p'_1 + 1, n + n' + 1\}$, since $p_1 = n = size(y)$ and $p'_1 = n' = size(z)$.

For p_b, the answer from the root has to be a YES. Since there is no way the root is getting eliminated, we get $p_b = 0$. For n_a, since we get a NO from the root, $n_a = 0$.

For the case when x has a single child y, if we create a dummy node z with a single tuple $T_z[0] = ((0,0),0,0,0)$, then the same equations given above hold for each of the cases.

The approach above can be generalized to l-ary trees as well.

7.5.2 Multi-Unlimited

Finally, we address the problem of unlimited categorization:

Definition 7.5.7 (Multi-Unlimited) *(Unlimited Search in a DAG for a target set) Find the smallest set of nodes $Q \subseteq V$ to ask questions such that $\forall U^* \subseteq V$ satisfying $ip(U^*) = 1$, we have $|cand(Q, U^*)| = |U^*|$.*

The following theorem shows an interesting result that the Multi-Unlimited problem is "trivialized" by the fact that questions need to be asked at all nodes to ensure that no extraneous nodes remain in the candidate set.

Theorem 7.5.8 (Triviality) *The optimal solution to an instance of Multi-Unlimited is $Q = V$, i.e., we need to ask a question at every node in the graph.*

Proof 7.5.9 *Consider Figure 7.5, abstractly representing a connected component of the input graph, focusing on any node n. We prove that we need to ask a question at n in order to ascertain if $n \in U^*$. Suppose we don't ask a question at n. Let the questions asked at all of the ancestors of n, i.e., A, return YES, while questions at all of the descendants of n, i.e., D, return NO. In this case, it is not clear if n is in U^* or not. It is possible that n is in U^*, in which case none of the nodes in A form part of U^*. Otherwise, if $n \notin U^*$, then there may be many nodes from A which are part of U^*.*

Intuitively, we need to ask a question at every node x because if we get a YES response from all of x's ancestors, and a NO response from all of x's descendants, we cannot be sure if x is in the target set or not.

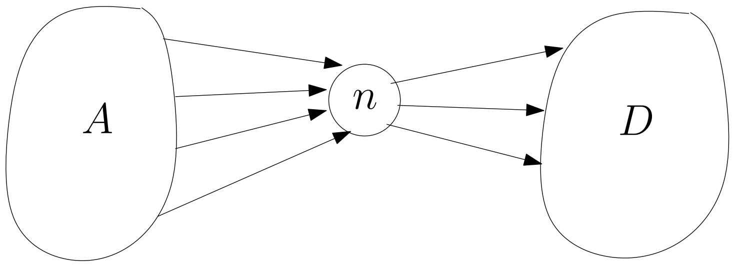


Figure 7.5: Triviality of Multi-Unlimited

7.6 Experimental Study

We conducted an experimental study of our categorization algorithms using a webpage categorization task on the real-world DMOZ concept hierarchy (`http://dmoz.org`). DMOZ is a human-curated internet directory based on a downward tree of categories. The goal is to assign websites of interest to nodes in this downward tree. In general, human judgment is needed for this assignment.

To expedite the process of assigning new web-pages to categories, we use the categorization algorithms to select which questions to ask humans. A question at the node corresponding to category X would be of the form: does this new webpage fall under X? Questions for a given web-page can be asked concurrently to independent humans, and the answers can then be combined to determine the candidate set of categories.

Experimental Objectives. Our experimental study has two important features that complement the analysis presented in previous sections. First, we study how our algorithms behave on average (rather than worst case) for a specific problem over real data. Second, since the algorithm for the Single-Unlimited problem for a downward tree is impractical, we study the performance when we use many iterations of the Single-Bounded algorithm. We now describe these features in more detail.

Recall that our algorithms are provably optimal in terms of minimizing the worst-case size of the candidate set. In our experiments, we would like to complement that theoretical analysis with measurements of the actual size of the candidate set after obtaining answers to the questions selected by the algorithm. By measuring the actual size, our goal is to examine whether the worst-case objective function also corresponds to good *average-case* performance.

One algorithm we could use for the webpage categorization task is Single-Unlimited. This algorithm selects questions to guarantee a wcase of size 1, but requires asking questions at almost every node, which is clearly impractical. Instead, since the average case may not correspond to the worst case, we would like to see how much the candidate set can be reduced by asking a bounded number

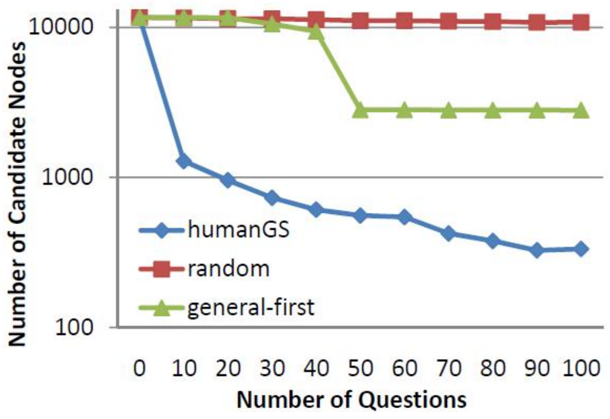


Figure 7.6: Experiment on Varying the Number of Questions Asked

of questions (i.e., using the algorithms for Single-Bounded). Using the answers to these questions, we may be able to precisely determine the target node or category. If the target node is not precisely determined, we may need to invoke the algorithm once again to issue another bounded set of categorization questions to the crowdsourcing marketplace. Thus, a practical method of using our algorithm is in *phases* where in each phase we invoke the Single-Bounded algorithm in order to select b additional questions by taking the current candidate set as the input downward tree. The net effect is that each phase shrinks the candidate set further, until the target node is identified or the candidate set is small enough to assign to a human worker for the final solution. This approach is a hybrid approach between a completely offline approach and a completely online approach (when we issue a single question at a time).

We present experiments to evaluate the average-case performance of our algorithms, focusing on the following questions:

- How does the candidate set shrink
    - as we vary the number of questions asked?
    - as we vary the size of the input downward tree?
    - when multiple phases are used?
- What is the relationship between the number of questions per phase and the total number of phases in order to complete categorization?

7.6.1 Methodology

Task Specifics. We evaluate our algorithms with a webpage categorization task on the science subtree of the DMOZ hierarchy (containing over 11,600 nodes). Each task is the placement of a web page into the hierarchy. We simulate each task by picking a node in the hierarchy where the web page would go. Then we simulate the crowdsourcing marketplace by answering the questions asked by our algorithms truthfully. If an actual crowdsourcing marketplace is used, correctness may be ensured by having multiple humans attempt the same question (as discussed in Chapter 3).

Tested Algorithms. We implemented our algorithm for Single-Bounded for a downward tree. We henceforth refer to this algorithm as *humanGS* (for Human-assisted Graph Search, which our algorithms are an instance of). We compare *humanGS* against two baseline algorithms. The first algorithm, *random*, simply picks b random nodes from the downward tree. The second algorithm, termed *general-first*, asks questions at the first b nodes encountered in a breadth-first traversal starting from (but excluding) the root.

Recall that each algorithm is executed in phases, where each phase receives as input the graph formed from the candidate set computed using all previous phases.

Metrics. We measure the performance of an algorithm as the actual size of the candidate set after asking the questions selected by the algorithm. To ensure statistical robustness, we test each algorithm on a set of 100 random tasks, each generated by sampling the target node uniformly at random from the nodes of the input tree, and we report the average size of the candidate set over all tasks in the test set. For algorithm *random*, we perform an additional averaging step over 10 runs of the algorithm per test task, in order to mitigate the effects of random question sampling.

7.6.2 Results

Effect of Number of Questions. This experiment examines how the candidate set shrinks on increasing the number of asked questions in one phase. We use b to denote the number of questions, and focus on the first phase where the input is the complete hierarchy.

Figure 7.6 shows the average number of candidate nodes as we vary b. Note that the y-axis is in log scale. As shown, our *humanGS* algorithm outperforms the baseline algorithms by an order of magnitude for all tested values of b. In particular, *humanGS* reduces the candidate set to around 1000 nodes with just 10 questions (a 10-fold decrease), whereas *random* and *general-first* flatten out well above 1000 nodes even after 100 questions. The steep drop for *general-first* at around 45 questions occurs because in our dataset there is a node close to the root with a large number of descendants.

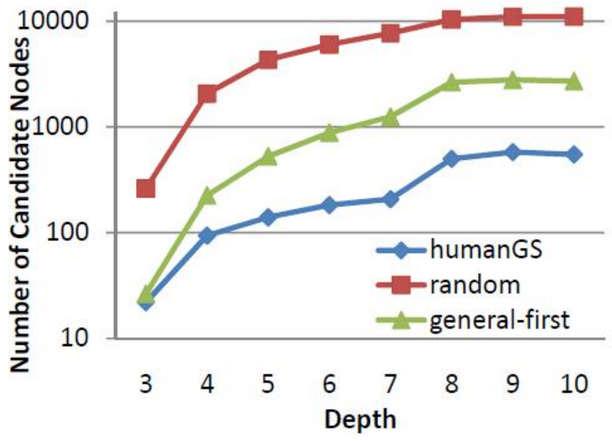


Figure 7.7: Experiment on Varying the Size of the Tree

When we select that node, a large part of the hierarchy is pruned.

Effect of Tree Size. The second experiment examines the performance of the three algorithms as we vary the size of the input tree. We varied the size of the downward tree by restricting the depth of the tree, i.e., any node at a greater depth is removed. The target node is sampled from the remaining nodes. We set $b = 50$ and focus again on a single phase.

Figure 7.7 depicts the average candidate size (again in log scale on the y-axis) against the depth of the tree. Algorithm *humanGS* continues to outperform its competitors, with an increasing margin as we approach the actual size of the tree. Its performance is matched by *general-first* only for a "trivial" depth of three, which corresponds to an unrealistically small task.

Phase-based Operation. The final experiment evaluates the overall performance of the phase-based approach. Our goal is to examine how rapidly each algorithm identifies the single target node of the specific categorization instance.

Figure 7.8 depicts the average candidate-set size for the three algorithms as we increase the number of phases, when $b = 100$ for each phase, once again with the y-axis on a log scale. The three plots corresponding to 100 questions per phase are denoted *humanGS (100)*, *random (100)* and *general-first (100)* in the figure. (We also plot the curve for *humanGS* when $b = 50$ and we discuss it in the next paragraph – depicted as *humanGS (50)* in the figure.) We observe that *humanGS* yields the fastest decrease among the three algorithms. For instance, *humanGS* is able to identify the target node after 5 phases on average. This compares favorably to the eight phases required by *general-first*, whereas *random* was not able to decrease the candidate set below 10000 nodes for the whole experiment. Additionally, the candidate-set size of *humanGS* is below 20 after only two phases. In contrast, the

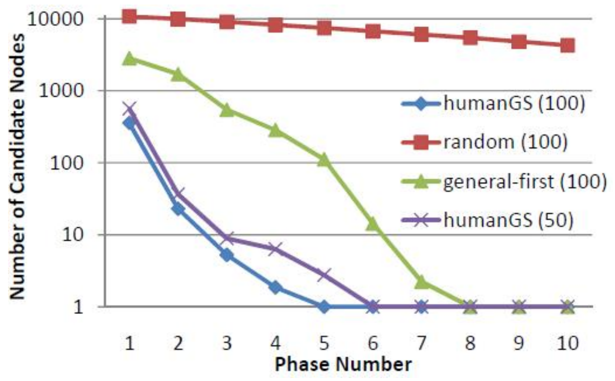


Figure 7.8: Experiment on Varying the Number of Phases

corresponding sizes for *general-first* and *random* are close to 1000 and 10000 respectively. Note that when the candidate set size is small, we may also consider giving the candidate nodes to a human worker in order to make the categorization.

Focusing on the two curves for *humanGS*, we observe a small degradation in performance from $b = 100$ to $b = 50$ for the same number of phases. However, $b = 50$ yields much better overall performance if we consider the total number of questions asked. For instance, with two phases and $b = 50$ (a total of 100 questions), *humanGS* performs an order of magnitude better than having a single phase with $b = 100$. Furthermore, *humanGS* requires 6 phases to discover the single target node with $b = 50$ (a total of 300 questions), compared to 5 phases for $b = 100$ (a total of 500 questions). These results indicate that increasing the number of phases is more beneficial compared to increasing the number of questions per phase. The trade-off of course is in latency, since the extra phases mean additional round-trips to the crowd-sourcing service, and reduced parallelism. Examining this trade-off in more detail is an interesting direction for future work.

7.7 Related Work

Unlike previous chapters, where there are a number of related papers studying similar problems, here, we are not aware of any prior work that studies the same or similar problems. However, we do briefly mention three topics studied in related work that are similar in spirit:

Active learning [169], also mentioned in earlier chapters, also studies the problem of requesting input from experts with the maximal "information content", similar in spirit to our problem. This input is only used to generate training data for machine learning tasks (especially when the current data is insufficient or not informative). Additionally, most work in active learning does not ask questions to humans in parallel, and thus does not leverage the inherent parallelism in crowd-sourcing systems.

Our problem is similar to that of decision trees [46]. In decision trees, we wish to classify an item as belonging to one of many classes (in our case, as being one of the nodes in the DAG). Unlike decision trees, we do not have a training set (or statistics of various classes) and there are no attributes on which a classifier can be built. The only questions that we can ask are those that involve reachability, and the optimization issues that arise are very different in our case.

There are other domains with optimization problems that involve searching with a budget, e.g., searching in a 2-dimensional area for oil [140], where the aim is to find oil as quickly as possible by selecting areas to drill and the depths of drilling. There has also been work on multi-armed bandits [180] and similar stochastic models for searching for an optimal solution that yields most revenue in computation advertising and machine learning settings. These models analyze the trade-offs between *exploration vs. exploitation*. The searching model in these domains is very different from ours, and hence the solutions proposed are different as well.

7.8 Conclusions

In this chapter, we focused on categorization. We explored the problem space via three orthogonal axes: Single / Multi, Bounded / Unlimited and DAG / Downward-Forest, and developed algorithms for all combinations. Our algorithms generate the optimal set of questions that can be asked to humans: that is, for a cost bound (b questions) and latency bound (one phase), we minimize worst-case uncertainty.

This chapter concludes our treatment of crowd-powered algorithms. Next, we study systems and applications wherein our crowd-powered algorithms may be used.

Chapter 8

Application 1: Datasift

In this chapter, we describe our first crowd-powered system, DataSift[1]. DataSift is a crowd-powered search toolkit that enables users to leverage crowdsourcing to get better search results. DataSift demonstrates the usefulness of our crowd-powered algorithms, specifically, filtering (Chapter 3). This chapter will illustrate that there are many additional challenges, beyond core algorithms, in designing and building crowd-powered systems and applications.

8.1 Introduction

While information retrieval systems have come a long way in the last two decades, modern search engines still have quite limited functionality. For example, they have difficulty with:

1. Non-textual queries or queries containing both text and non-textual fragments: For instance, a query *"cables that plug into <IMAGE>"*, where *<IMAGE>* is a photo of a socket, cannot be handled by any current search engine.

2. Queries over non-textual corpora: For instance, a query *"funny pictures of cats wearing hats, with captions"* cannot be handled adequately by any image search engine. Search engines cannot accurately identify if a given image satisfies the query; typically, image search engines perform keyword search over image tags, which may not be sufficient to identify if the image satisfies the query.

[1] This chapter is adapted from our paper [150], published at HCOMP 2013, written jointly with Ming Han Teh, Hector Garcia-Molina, and Jennifer Widom.

3. Long queries: For instance, a query *"find noise canceling headsets where the battery life is more than 24 hours"* cannot be handled adequately by a product search engine. Search results are often very noisy for queries containing more than 3-4 keywords. Most search engines require users to employ tricks or heuristics to craft short queries and thereby obtain meaningful results [16].

4. Queries involving human judgment: For instance, a query *"apartments that are in a nice area near Somerville"* cannot be handled adequately by an apartment search engine.

5. Ambiguous queries: For instance, a query *"fast jaguar images"* cannot be handled adequately by an image search engine. Search engines cannot tease apart queries which have multiple or ambiguous interpretations, e.g., the car vs. the animal.

For all of these types of queries, currently the burden is on the user to attempt to express the query using the search interfaces provided. Typically, the user will try to express his or her query in as few textual keywords as possible, try out many possible reformulations of the query, and pore over hundreds or thousands of search results for each reformulation. For some queries, e.g., *"buildings that look like <IMAGE>"*, identifying a formulation based solely on text is next to impossible.

Additionally, there are cases where the user does not possess the necessary knowledge to come up with query reformulations. For instance, for the query *"cables that plug into <IMAGE>"*, a particular user may not be able to identify that the socket is indeed a USB 2.0 socket.

To reduce the burden on the user, both in terms of labor (e.g., in finding reformulations and going through results) and in terms of knowledge (e.g., in identifying that a socket is indeed a USB 2.0 socket), we turn to crowdsourcing for assistance. In this chapter, we present DataSift, a powerful general-purpose search toolkit that uses human workers to assist in the retrieval process. Our toolkit can be connected to any traditional corpus with a basic keyword search API. DataSift then automatically enables rich queries over that corpus. Additionally, DataSift produces better results by harnessing human computation to filter answers from the corpus.

Figure 8.1 shows a high-level overview of DataSift: The user provides a rich search query Q of any length, that may include textual and/or non-textual fragments. DataSift uses an internal pipeline that makes repeated calls to a crowdsourcing marketplace—specifically, Mechanical Turk [14]—as well as to the keyword search interface to the corpus. When finished, a ranked list of results are presented back to the user, like in a traditional search engine. As an example, Figure 8.2 depicts the ranked list of results for the query Q = *"type of cable that connects to <IMAGE: USB B-Female socket of a*

printer>" over the Amazon products corpus [1]. The ranked results provide relevant USB 2.0 cables with a B-Male connector.

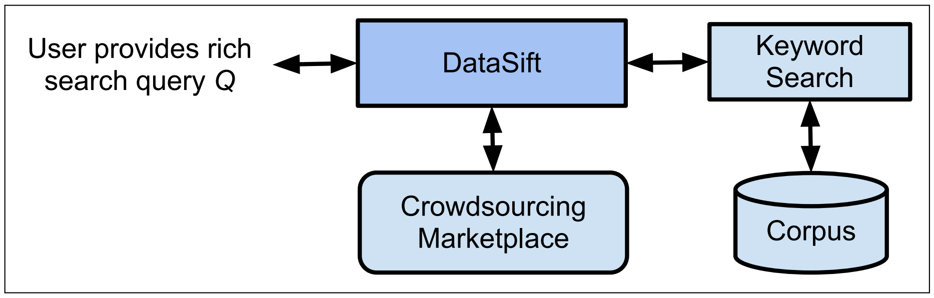
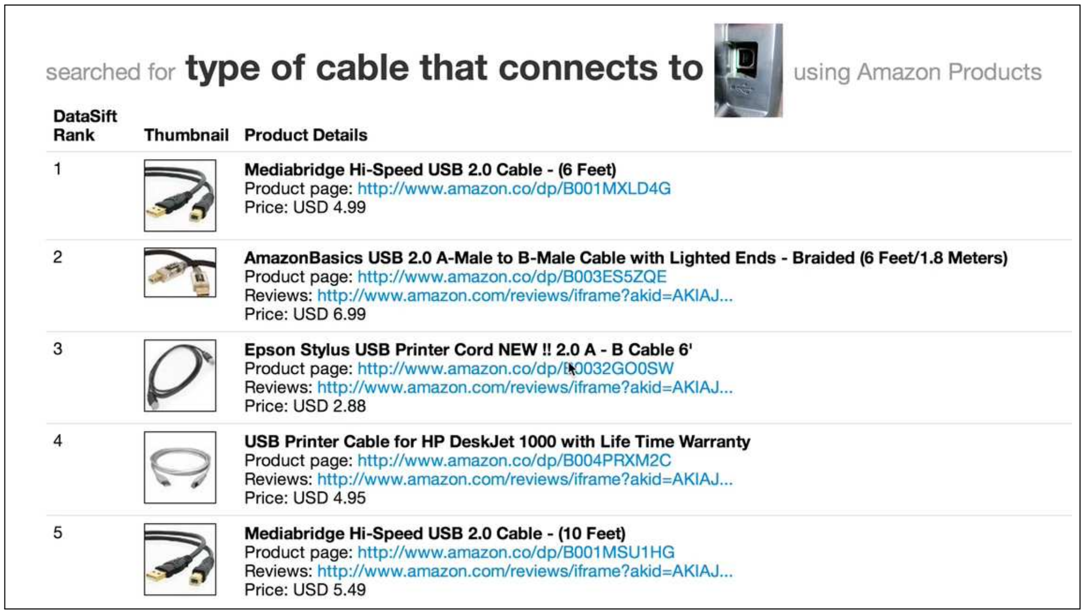


Figure 8.1: DataSift Overview



searched for **type of cable that connects to** using Amazon Products

DataSift Rank	Thumbnail	Product Details
1		**Mediabridge Hi-Speed USB 2.0 Cable - (6 Feet)** Product page: http://www.amazon.co/dp/B001MXLD4G Price: USD 4.99
2		**AmazonBasics USB 2.0 A-Male to B-Male Cable with Lighted Ends - Braided (6 Feet/1.8 Meters)** Product page: http://www.amazon.co/dp/B003ES5ZQE Reviews: http://www.amazon.com/reviews/iframe?akid=AKIAJ... Price: USD 6.99
3		**Epson Stylus USB Printer Cord NEW !! 2.0 A - B Cable 6'** Product page: http://www.amazon.co/dp/B0032GO0SW Reviews: http://www.amazon.com/reviews/iframe?akid=AKIAJ... Price: USD 2.88
4		**USB Printer Cable for HP DeskJet 1000 with Life Time Warranty** Product page: http://www.amazon.co/dp/B004PRXM2C Reviews: http://www.amazon.com/reviews/iframe?akid=AKIAJ... Price: USD 4.95
5		**Mediabridge Hi-Speed USB 2.0 Cable - (10 Feet)** Product page: http://www.amazon.co/dp/B001MSU1HG Reviews: http://www.amazon.com/reviews/iframe?akid=AKIAJ... Price: USD 5.49

Figure 8.2: DataSift Example

A disadvantage of our approach is that the response time will be substantially larger than with a traditional search engine. Thus, our approach is only applicable when the user is willing to wait for higher quality results, or when he is not willing or capable of putting in the effort to find items that satisfy his query. Our experience so far is that the wait times are of the order of 20 minutes to an hour. (Note that users can see partial results as they come in.)

We now present some challenges in building DataSift, using our earlier example: Q = *"type of cable that connects to <IMAGE>"*. We assume that we have a product corpus (e.g., Amazon products) with a keyword search API. Consider the following three (among others) possible configurations for DataSift:

- **Gather:** Provide Q as is to a number of human workers and ask them for one or more reformulated textual keyword search queries, e.g., *"USB 2.0 Cable"* or *"printer cable"*. Then retrieve products using the keyword search API for the reformulated keyword search queries and present the results to the user.
- **Gather-Filter:** The same configuration as **Gather**, but in addition ask human workers to filter the retrieved products for relevance to the query Q, e.g., whether they are cables that plug into the desired socket, before presenting the results to the user.
- **Iterative Gather-Filter:** The same configuration as **Gather**, but in addition first ask human workers to filter a small sample of retrieved products from each reformulated textual query for relevance to Q, allowing us to identify which reformulations produce better results. Then, retrieve more from the better reformulations, e.g., more from *"USB 2.0 printer cable"* instead of *"electronic cable"*. Finally, ask human workers to filter the retrieved results before presenting the results to the user.

In addition to determining which configuration we want to use, each of the configurations above has many parameters that need to be tuned. For instance, for the last configuration, DataSift needs to make a number of decisions, including:

- How many human workers should be asked for reformulated keyword search queries? How many keyword search queries should each worker provide?
- How many items should be retrieved initially for each reformulation? How many should be retrieved later (once we identify which reformulations produce better results)?
- How do we decide if a reformulation is better than a different one?
- How many human workers should be used to filter each product for relevance to Q? Here, we may certainly reuse results from Chapter 3.
- How should the cost be divided between the steps?

Our current implementation of DataSift is powerful enough to be configured to match all of the configurations we have described, plus others. We achieve this flexibility by structuring the toolkit as six plug-and-play components that can be assembled in various ways, described in detail in the next section. In this chapter, we present and evaluate a number of alternative configurations for DataSift, and identify good choices for the parameters in each configuration.

8.1.1 Outline of Chapter

Here is an outline for the rest of the chapter:

1. We describe a number of plug-and-play components — automated and crowdsourced — that form the core of DataSift (Section 8.2).
2. We identify a number of configurations for DataSift using the plug-and-play components (Section 8.3).
3. We present the current implementation of DataSift, which supports all the described configurations (Section 8.4).
4. We perform a performance evaluation of these configurations. We show that configurations that use the crowd can yield 100% more precision than traditional retrieval approaches, and those that ask the crowd for reformulations can improve precision by an additional 100% (Section 8.5).
5. We optimize the selected configurations, identifying good values for individual parameters (Section 8.6).

8.2 Preliminaries and Components

A user enters a query Q into DataSift, which could contain textual and non-textual fragments. Fully textual queries or fully textual reformulations are denoted with the upper case letter TQ (denoting text). The corpus of items $\mathcal{I}$ (products, images, or videos) over which DataSift is implemented has a keyword search API: it accepts a textual keyword search query and a number k, and returns the top k items (products, images, or videos) along with their ranks.DataSift makes repeated calls to both $\mathcal{I}$ and the crowdsourcing marketplace, and then eventually provides the user with n items in ranked order. (In fact, DataSift is flexible enough to provide the user with a dynamically updated ranking of items that is kept up-to-date as DataSift evaluates the items.)

Next, we describe the components internal to DataSift. Components are categorized into: (1) Crowdsourced Components: components that interact with the crowdsourcing marketplace, and (2) Automated Components: components that function independent of the crowdsourcing marketplace. The function signatures of the components are provided in Table 8.1. Note that the query Q and the corpus of items $\mathcal{I}$ are implicit input arguments in the signatures for all these components.

8.2.1 Crowdsourced Components

- (G) **Gather Component**: $h, s \rightarrow \{TQ\}$

  The Gather Component G asks human workers for fully textual reformulations for Q, providing DataSift a mechanism to retrieve items (using the keyword search API) for Q (recall that Q may contain non-textual fragments).

  Given a query Q, G uses the marketplace to ask h human workers for s distinct textual reformulations of Q each, giving a total of $h \times s$ textual queries. Specifically, human workers are asked to respond to the following task: "Please provide s reformulated keyword search queries for the following query:" Q. The human workers are also able to run the reformulated query on the corpus $\mathcal{I}$ to see if the results they get are desirable.

- (F) **Filter Component**:

  $\{(\mathcal{I}, TQ, rank)\}, t \rightarrow \{(\mathcal{I}, TQ, x, y)\}$

  The input to the Filter Component F is a set of items. We will ignore TQ and $rank$ for now, these parameters are not used by F. For each item i in the set of items, F determines whether the item satisfies the query Q or not. The component does this by asking human workers to respond to the following task: "Does the item i satisfy query Q: (YES/NO)". Since human workers may be unreliable, multiple workers may be asked to respond to the same task on the same item i. The number of humans asked is determined by designing the optimal filtering strategy (from Chapter 3) using the overall accuracy threshold τ (set by the application designer)—we provide details in the section on implementation. The number of positive responses for each item is denoted x, while the number of negative responses is denoted y.

  In the input, each item i is annotated with TQ and $rank$: TQ is the textual query TQ whose keyword search result set item i is a part of. $rank$ is the rank of i in the keyword search results for TQ. Both these annotations are provided as part of the output of the keyword search API call – see component R). $TQ, rank$ are part of the input for compatibility with the calling component, and TQ is part of the output for compatibility with the called component.

8.2.2 Automated Components

- (R) **Retrieve Component**:

  $\{T\} \mid \{(TQ, w)\},\ k\ \rightarrow\ \{(\mathcal{I}, TQ, rank)\}$

  The Retrieve Component R uses the keyword search API to retrieve items for multiple textual queries TQ from the corpus. For each textual query TQ, items are retrieved along with their keyword search result ranks for TQ (as assigned in the output of the keyword search API call). Specifically, given a set of textual queries TQ_i along with weights w_i, R retrieves k items in total matching the set of queries in proportion to their weights, using the keyword search API. In other words, for query TQ_i, the top $k \times \frac{w_i}{\sum_j w_j}$ items are retrieved along with their ranks for query TQ_i. If the weights are not provided, they are all assumed to be 1. We ignore for now the issue of duplicate items arising from different textual queries; if duplicate items arise, we simply retrieve additional items from each TQ_i in proportion to w_i to make up for the duplicate items.

- (S) **Sort Component**:
  The Sort Component S has two implementations, depending on which component it is preceded by. Overall, S merges rankings, providing a rank for every item based on how well it addresses Q.
  $\{(\mathcal{I}, TQ, x, y)\} \rightarrow \{(\mathcal{I}, rank)\}$
  If preceded by the F component, then S receives as input items along with their textual query TQ, as well as x and y, the number of YES and NO votes for the item. Component S returns a rank for every item based on the difference between x and y (higher $(x - y)$ gets a higher rank); ties are broken arbitrarily. The input argument corresponding to the textual query TQ that generated the item is ignored. NOte that while we could adapt ideas from Chapter 6 (Maximum) for sorting, we use a simpler approach here.
  $\{\mathcal{I}, TQ, rank\} \rightarrow \{(\mathcal{I}, rank)\}$
  If preceded by the R component, then S receives as input items along with their textual query TQ, as well as $rank$, the rank of i in the result set of TQ. Component S simply ignores the input argument corresponding to TQ, and merges the ranks; ties are broken arbitrarily. For example, if $(a, TQ_1, 1), (b, TQ_1, 2), (c, TQ_2, 1), (d, TQ_2, 2)$ form the input, then one possible output is: $(a, 1), (b, 3), (c, 2), (d, 4)$; yet another one is: $(a, 1), (b, 4), (c, 2), (d, 3)$.
- (W) **Weighting Component**:$\{(\mathcal{I}, TQ, x, y)\} \rightarrow \{(TQ, w)\}$
  For Iterative Gather-Filter (Section 8.1), the weighting component is the component that actually evaluates reformulations. The component always follows F, using the results from F to compute weights corresponding to how good different reformulations are in producing items that address Q.
  Component W receives as input items from the Filter Component F, annotated with x and y (the number of YES and NO votes), and the textual query TQ that generated the items. For each textual query TQ, given the output of the filtering component F, the weighting component returns a weight based on how useful the textual query is in answering Q.

	Signature	Followed by
G	$h, s \rightarrow \{TQ\}$	R
F	$\{(\mathcal{I}, TQ, rank)\},\ t \rightarrow \{(\mathcal{I}, TQ, p, n)\}$	W, S
R	$\{T\} \mid \{(TQ, w)\},\ k \rightarrow \{(\mathcal{I}, TQ, rank)\}$	F, S
S	$\{(\mathcal{I}, TQ, p, n)\} \mid \{\mathcal{I}, TQ, rank\} \rightarrow \{(\mathcal{I}, rank)\}$	—
W	$\{(\mathcal{I}, TQ, p, n)\} \rightarrow \{(TQ, w)\}$	R

Table 8.1: Components, their function signatures (Q and $\mathcal{I}$ are implicit input parameters in all of these functions), and other components that can follow them.

There are three variants of W that we consider: W_1, W_2, and W_3, corresponding to three different ways in which weights w_i are assigned to TQ_i. For describing these variants, for convenience, we introduce two new definitions for the output of F: for a given item, if $x > y$, then we say that the item belongs to the *pass set*, while if $y \geq x$, then we say that the item belongs to the *fail set*.

- W_1: For each textual reformulation TQ_i, we set w_i to be the number of items (from that reformulation) in the pass set.
- W_2: Unlike W_1, which accords non-zero weight to every reformulation with items in the pass set, W_2, preferentially weights only the best reformulation(s). Let the size of the pass set for TQ_i be x_i, and let $X = \max_i(x_i)$. For each reformulation TQ_i that has $x_i = X$, we assign the weight $w_i = 1$. Otherwise, we assign the weight $w_i = 0$.
- W_3: Each reformulation is weighted on how much agreement it has with other reformulations based on the results of F. For instance, if reformulation TQ_1 has items $\{a, b\}$, TQ_2 has $\{b, c\}$, and TQ_3 has $\{a, d\}$ as ranks 1 and 2 respectively, then TQ_1 is better than TQ_2 and TQ_3 since both items a and b have support from other reformulations.
  For the i-th reformulation, we set w_i to be the sum, across all items (from that reformulation), the number of other reformulations that have that particular item. Thus: ($\mathcal{J}$ stands for the indicator function)

$$w_i = \sum_{\forall a \text{ from } TQ_i} \sum_{j \neq i} \mathcal{J}(a \text{ is in } TQ_j\text{'s results}),$$

8.3 Configurations

We now describe the DataSift configurations that we evaluate in this chapter. The goal of each configuration is to retrieve n items in ranked order matching query Q. Some configurations may retrieve $n' \geq n$, and return the top n items.

Given that the components described in the previous section are plug-and-play, there is a large number of configurations that we could come up with; however, we focus our attention on a few that we have found are the most interesting and important:

- RS: (Only possible if Q is textual) This configuration refers to the traditional information retrieval approach: component R uses the query Q to directly retrieve the top n items with ranks using the keyword search API. In this case, component S does nothing, simply returning the same items along with the ranks.
- RFS: (Only possible if Q is textual) From the $n' \geq n$ items retrieved by component R, component F uses humans to better identify which items are actually relevant to the query Q. Component S then uses the output of F to sort the items in the order of the difference in the number of YES and NO votes for an item as obtained by F, and returns n items along with their ranks.
- GRS: (Gather from Section 8.1) Component G gathers textual reformulations for Q, asking h human workers for s reformulations each. Subsequently, R retrieves the top $n/(hs)$ items along with ranks for each of these $h \times s$ reformulations. Then, S sorts the items by simply merging the ranks across the $h \times s$ reformulations, with ties being broken arbitrarily. Items are returned along with their ranks.
- GRFS: (Gather-Filter from Section 8.1) Component G gathers $h \times s$ textual reformulations, after which component R retrieves $n'/(hs)$ items with ranks for each of the reformulations. Then, component F filters the n' items using human workers. Subsequently, the n' items are sorted by component S based on the difference in the number of YES and NO votes for each item, and the top n are returned along with their ranks; ties are broken arbitrarily (the input argument corresponding to the textual reformulation is ignored).
- GRFW$_i$RFS for $i = 1, 2, 3$: (Iterative Gather-Filter from Section 8.1) Component G gathers $h \times s$ textual reformulations, after which component R retrieves δ items from each of the reformulations (δ is a small sample of results from each reformulation, typically much smaller than n). Component F then filters the set of $\delta \times h \times s$ items. The output of F provides us with an initial estimate as to how useful each reformulation is in answering the query Q.

Subsequently, component W (either W_1, W_2, or W_3) computes a weight for each of the textual reformulations based on the results from F. These weights are then used by component R to preferentially retrieve $n' - \delta \times h \times s$ items in total across reformulations in proportion to the weight. Component F filters the retrieved items once again. Eventually, the component S sorts the items in the order of the difference between the number of YES and NO votes (ignoring the input argument corresponding to the reformulation, and breaking ties arbitrarily), and returns the items along with their ranks.

For now, we consider only GRFW_1RFS (and not W_2 or W_3), which we refer to as GRFWRFS. We will consider other variants of W in Section 8.6.

8.4 Implementation

We provide a very brief overview of the DataSift implementation followed by details regarding the crowdsourced components.

DataSift is implemented in Python 2.7.3 using Django, the popular web application development library. We use Amazon's Mechanical Turk [14] as our marketplace. We leverage the Boto library [2] to connect to Mechanical Turk, and the Bootstrap library [22] for front-end web templates. A complete trace of activity from previous queries on DataSift, along with the results, are stored in a MySQL 5 database. The current version of DataSift connects to four corpora: Google Images [10], YouTube Videos [25], Amazon Products [1], and Shutterstock Images [19].

We now provide specifics regarding the implementation of the crowdsourced components:

- (G) **Gather Component**:
  Using the search query Q provided, component G issues Amazon Mechanical Turk microtasks, also called Human Intelligence Tasks (HITs) that solicits textual reformulations from human workers. Here, we discuss some of the design features of the HIT seen by a human worker.
  Humans are allowed to refine their reformulations before submission by using a "test search" feedback loop to probe their reformulations against the corpus. The top ten results from the corpus for the corresponding reformulation are displayed to the human worker so that he or she can decide if the results are satisfactory.
  Initially, we discovered that instructions were unclear, due to which humans were providing reformulations that were simple paraphrasing of the search predicate. To gather better reformulations, we added examples to the instructions to allow human workers to provide reformulations based on contextual knowledge. One such example we provided hinted that a possible

reformulation for Q = *"SF bridge; night scene"* might be *"golden gate bridge night"*. In addition, we extracted and included tips from Google's "Basic Search Help" [13] to enable better reformulations.

The gather component prevents human workers from submitting duplicate reformulations for the same query Q, and distinct humans from submitting identical reformulations. While it might be possible to make inferences about the relevance of a reformulation given duplicate reformulations, we hypothesized that it is best to not have duplicates to ensure diversity and maximize utility, and then have downstream components eliminate the less-relevant reformulations.

- (F) **Filter Component**:

  Recall that the filter component F takes as input a set of items $\mathcal{I}$, and uses human workers to check if items satisfy query Q. As a first step, for every item, F asks $h = 3$ human workers to verify if the item satisfies query Q. Subsequently, these $|\mathcal{I}| \times 3$ answers are used to learn the average probabilities of human error (i.e., the probability that an item satisfies Q, but a human answers otherwise, and vice versa), and the a-priori probability of an item satisfying Q. Unfortunately, since DataSift is completely unsupervised, we do not know the true values for items. Instead, we use a simple heuristic to infer true values of items: if the number of YES answers is greater than the number of NO answers, then we assume that the item satisfies Q, and that the item does not satisfy Q otherwise. Using the true values, we can learn the probabilities of human error and a-priori probability. These quantities are then input to the strategy computation algorithm from Chapter 3, which outputs a strategy that determines how many additional answers are needed (beyond the three initial ones) for each item. For instance, the strategy might output that if 3 YES answers have been obtained for an item, then no additional answers are needed, but if 2 YES and 1 NO answers have been obtained, then one more question needs to be asked. Note that we do not use the generalized filtering strategies from Chapter 4 since we do not have accurate estimates of error rates of individual workers.

  To reduce cost and improve accuracy, to each human worker, we provide a batch of items to be evaluated at the same time (i.e., whether or not they satisfy Q) as one HIT on Mechanical Turk. We set our batch size to be 50 to provide workers the ability to have enough items to evaluate simultaneously, and yet not be fatigued by the task. To reduce bias, we randomized the assignment of items to batches, as well as the ordering of items within a batch. Our batches look similar to the example task in Figure 2.2.

Easy Queries (5)
funny photo of barack obama eating things
bill clinton waving to crowd
matrix digital rain
eiffel tower, paris
5 x 5 rubix cube

Hard Queries (5)
tool to clean laptop air vents
cat on computer keyboard with caption
handheld thing for finding directions
the windy city in winter, showing the bean
Mitt Romney, sad, US flag

Selected Others
funny photos of cats wearing hats, with captions
the steel city in snow
stanford computer science building
database textbook

Table 8.2: List of Textual Queries for Initial Evaluation

8.5 Initial Evaluation on Textual Queries

We perform an initial evaluation of the configurations described in Section 8.3. Specifically, we assess how much benefit we can get from using various crowd-powered configurations over the traditional fully-automated retrieval approach (RS). Since rich media queries are simply not supported by traditional retrieval approaches, for our initial comparison, we focus on fully textual queries. (We consider rich media queries in the next section.)

Setup: We hand-crafted a set of 20 diverse textual queries (some shown in Table 8.2). We executed these 20 queries using each of four configurations RS, RFS, GRS, GRFWRFS on the Google Images corpus. For each of the configurations, we set n', i.e., the total number of items retrieved, to be 50. For both GRS and GRFWRFS, we used $h = w = 3$ and for GRFWRFS, we used $\delta = 3$.

Evaluation: To evaluate the quality of the ranked results, we measure the fraction of true positives in the top-n items, i.e., the number of items in the top n satisfying Q divided by n. Note that this quantity is precisely *precision@n*. To determine the number of true positives, we manually inspected the results, carefully checking if each item returned satisfies the query Q or not.

Basic Findings: Our results can be found in Figure 8.3. We plot the fraction of true positives in

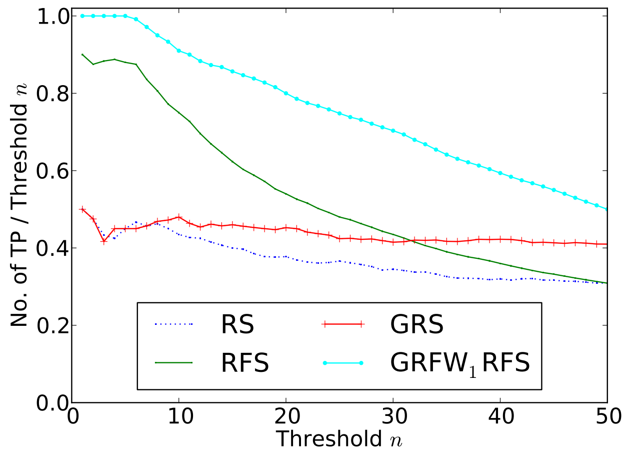


Figure 8.3: Precision curve

the top-n result set for each of the configurations, on varying the threshold n. As an example, for threshold $n = 30$, GRS and RFS have precision 0.4 (i.e., 0.4 * 30 = 12 items satisfy Q on average from the top 30), while RS has precision 0.35, and GRFWRFS has precision 0.7, 100% higher than the precision of RS. Therefore, sophisticated configurations combining the benefits of the crowdsourced components F and G perform much better than those with just one of those components, and perform significantly better than fully automated schemes.

Notice that the configuration RFS is better than GRS for smaller n. Configuration RFS retrieves the same set of items as RS, but the additional crowdsourced filter F component ensures that the items are ranked by how well they actually satisfy Q. Configuration GRS on the other hand, gathers a number of reformulations, ensuring a diverse set of retrieved items. However, the items may not be ranked by how well they actually satisfy Q – the good items may in fact be lower ranked. As a result, for smaller n, RFS does better, but GRS does better for larger n.

In addition, we plotted the precision curve with error bars in Figure 8.4. Unlike the other configurations, the error bars for GRFWRFS are initially zero and increase with n, indicating that GRFWRFS produces consistently relevant results for the top 5 items. GRS has initial error bars that are higher than the rest because the top results might be relevant to one of the reformulations but not to Q. However, as more results are considered, its easier to find items that are relevant to the original query Q.

Summary: Crowd-powered configurations RFS, GRS, *and* GRFWRFS *outperform* RS. GRFWRFS *clearly does the best, with 50-200% higher precision than* RS *on average, followed by* GRS. RFS *is better*

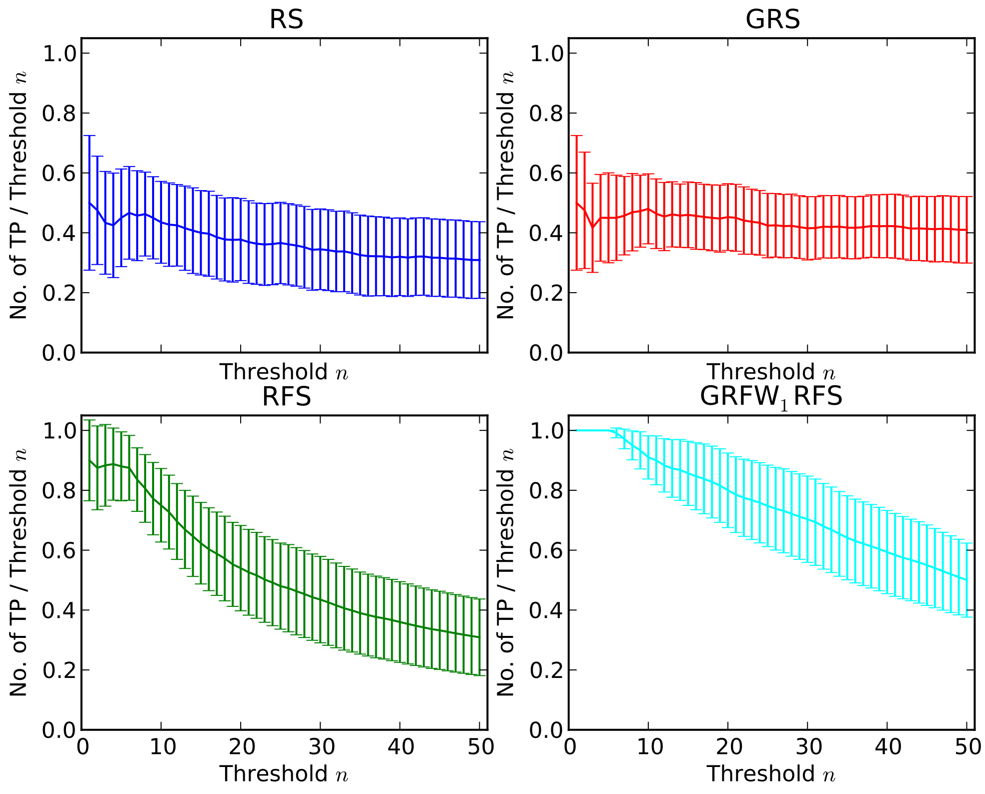


Figure 8.4: Precision curve with error bars showing 95% confidence interval

than GRS *for smaller n due to* F*, but* GRS *does better for larger n.*

Query Difficulty: To study the impact of query difficulty on results, we ordered our queries based on the number of true positive results in the top 10 results using the traditional retrieval approach. We designated the top 5 and the bottom 5 queries as the *easy* and the *hard* queries respectively — see Table 8.2 for the list of queries in each category. We then plotted the fraction of true positives on varying n for each category. We depict the results in Figure 8.5. The general trend is consistent with Figure 8.3 except that for easy queries, RFS and RS outperforms GRS. This somewhat counterintuitive result makes sense because for easy queries, most of the results from the traditional retrieval approach are already good, and therefore it is more beneficial to use the filter component rather than the gather component. In fact, the gather component may actually hurt performance because the reformulations may actually be worse than the original query Q. On the hard queries, GRFWRFS performs significantly better than the other configurations, getting gains of up to 500% on precision for small n.

Summary: Crowd-powered configurations RFS *and* GRFWRFS *outperform* RS *even when restricted to very hard or easy queries. However, the benefits from using crowd-powered configurations is more evident on the hard queries.*

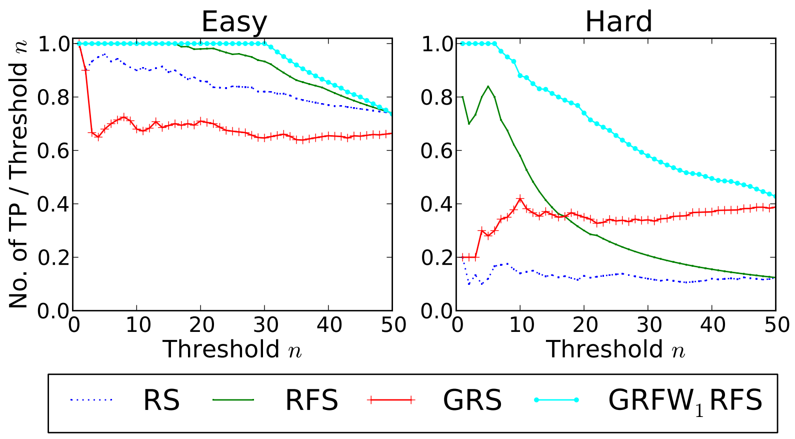


Figure 8.5: Precision curves for easy vs hard queries

8.6 Rich Queries and Parameter Tuning

We now describe our results on running the sophisticated configurations on rich media queries, and also describe our experiments on choosing appropriate values for parameters for the sophisticated

Rich Queries (5)
buildings around <IMAGE: UC Berkeley's Sather Tower>
device that reads from <IMAGE: Iomega 100MB Zip Disk>
where to have fun at <IMAGE: Infinity Pool at Marina Bay Sands hotel in Singapore>
tool/device that allows me to do hand gestures such as in: <VIDEO: motion sensing demonstration using fingers >
type of cable that connects to <IMAGE: USB B-Female socket of a printer>

Table 8.3: List of Rich Queries

configurations. For both these objectives, we generated a test data-set in the following manner:

Data Collection: We constructed 10 queries: 5 (new) fully textual queries and 5 queries containing non-textual fragments — that we call *rich* queries. (See Table 8.3 for the list of rich queries.) For each query, we gathered 25 reformulations (5 human workers × 5 reformulations per worker), then retrieved a large (> 100) number of items for each reformulation, and filtered all the items retrieved using crowdsourced filter component F. This process actually provided us with enough data to simulate executions of all configurations (described in Section 8.3) on any parameters $h, s \leq 5, n' \leq 100$. Moreover, by randomizing the order of human participation in G, we can get multiple executions for a fixed configuration with fixed parameters. That is, if we have $h = 3$, then we get $\binom{5}{3}$ simulated executions by allowing G to get reformulations from any 3 workers out of 5.

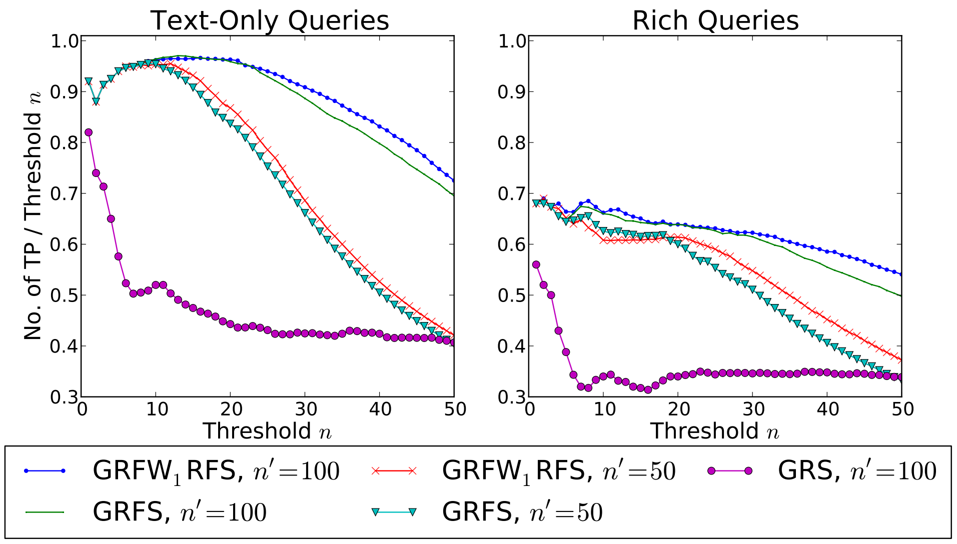


Figure 8.6: Curves for: (a) textual queries (b) rich queries

Monetary Costs: So far, while we have compared the configurations against each other on precision, these configurations actually have different costs. We tabulate the costs for each configuration in symbolic form in Table 8.4. In addition to precision, we will use the costs described above to compare configurations in subsequent experiments.

Basic Findings: We first study the differences in performance of DataSift configurations on rich queries and on textual queries. We set $h = s = 3$, $\delta = 1$, and simulated the execution of configurations GRS (for $n' = 50$), GRFS (for $n' = 50, 100$), GRFWRFS (for $n' = 50, 100$). We plot the the average fraction of true positives in the top n, divided by n, on varying n from $1 - 50$, for textual queries in Figure 8.6(a) and for rich queries in Figure 8.6(b). As can be seen in the two figures, the relative positions of the five configurations are similar in both figures.

We focus on the rich queries first (Figure 8.6(b)). As can be seen in the figure, for $n' = 50$, GRFWRFS has higher precision than GRFS (with the differences becoming more pronounced for larger n), and much higher precision than GRS. For instance, for $n = 50$, GRFWRFS has 15% higher precision than GRS and GRFS — the latter two converge at $n = 50$ because the same set of $n' = 50$ items are retrieved in both configurations. For $n' = 100$, GRFWRFS has higher precision than GRFS and GRS, as well as the plots for $n' = 50$. For instance, for $n = 50$, GRFWRFS with $n' = 100$ has close to 100% higher precision than GRS, and close to 50% higher precision than GRFWRFS with $n' = 50$. This is not surprising because retrieving more items and filtering them enables us to have a better shot at finding items that satisfy Q (along with ordering them such that these items are early on in the result set). We study the behavior relative to n' in more detail later on. GRS continues to perform similarly independent of the items n' retrieved since only the top n items are considered, and since $n' \geq n$.

Recall that GRFWRFS has the same cost as GRFS (Table 8.4). Thus, GRFWRFS strictly dominates GRFS in terms of both cost and precision. On the other hand, GRFWRFS may have higher cost than GRS, but has higher precision.

We now move back to comparing text and rich queries. As can be seen in the two figures, the gains in precision for textual queries from using more sophisticated configurations are smaller than the gains for rich queries. Moreover, the overall precision for the rich queries (for similar configurations) is on average much lower than that for text-only queries; not surprising given that the rich queries require deeper semantic understanding and more domain expertise.

Summary: On average, the relative performance of DataSift *configurations is similar for both textual and rich queries, with lower precision overall for rich queries, but higher gains in precision on using sophisticated configurations. For both textual and rich queries, on fixing the total number of items*

Configuration	Cost
RS	Free
RFS	$n' \times \tau \times C_1$
GRS	$h \times s \times C_2$
GRFS	$h \times s \times C_2 + n' \times \tau \times C_1$
GRFWRFS	$h \times s \times C_2 + n' \times \tau \times C_1$

Table 8.4: Breakdown of monetary costs associated with each configuration. τ is the expected number of human workers used to filter the item. C_1 is the cost of asking for a reformulation and C_2 is the cost of getting a human worker to filter a single item. Typical values are $C_1 = \$0.003$ (for images), $C_2 = \$0.10$, $\tau = 4$.

retrieved n' and the number of reformulations, GRFWRFS *does slightly better than* GRFS*, and does significantly better than* GRS*. For individual queries, the gains from using* GRFWRFS *may be even higher. On increasing the number of items retrieved,* GRS *continues to performs similarly, while* GRFS *and* GRFWRFS *both do even better.*

Optimizing GRFWRFS**:** Previously, we have found that of the configurations considered so far, GR-FWRFS provides the best precision. We now focus our attention on optimizing the parameters of GRFWRFS for even better precision. Specifically, we try to answer the following questions:

1. How do the variations of W_i, $i = 1, 2, 3$ perform against each other?
2. How do the number of human workers (h) and number of reformulations per worker (s) affect the results?
3. How should the sample size δ (used to evaluate the reformulations) be determined?
4. How does the number of target items n' affect precision?

Questions 1 and 2: Varying h, s and Varying W_{1-3}**:** We simulate the five configurations: GRS, GRFS, GRFW$_{1-3}$RFS on the 10 textual and rich queries, for $n' = 100$, $\delta = 3$. (Similar results are seen for other parameter settings.) We depict the fraction of true positives in the top-50 on varying h, s, as a heat map in Figure 8.7. In general, GRFW$_{1-3}$RFS has a higher number of true positives than GRFS, and GRFS has a higher number of true positives than GRS. We see a clear trend across rows and across columns: fixing one dimension while increasing the other increases the fraction of true positive results. For the 3 GRFW$_i$RFS configurations, having 1 worker with 5 reformulations outperforms 5 workers with 1 reformulation each; additionally, recreating the benefits of two workers with four reformulations each (a total of 8) requires at least five workers with three or more reformulations each

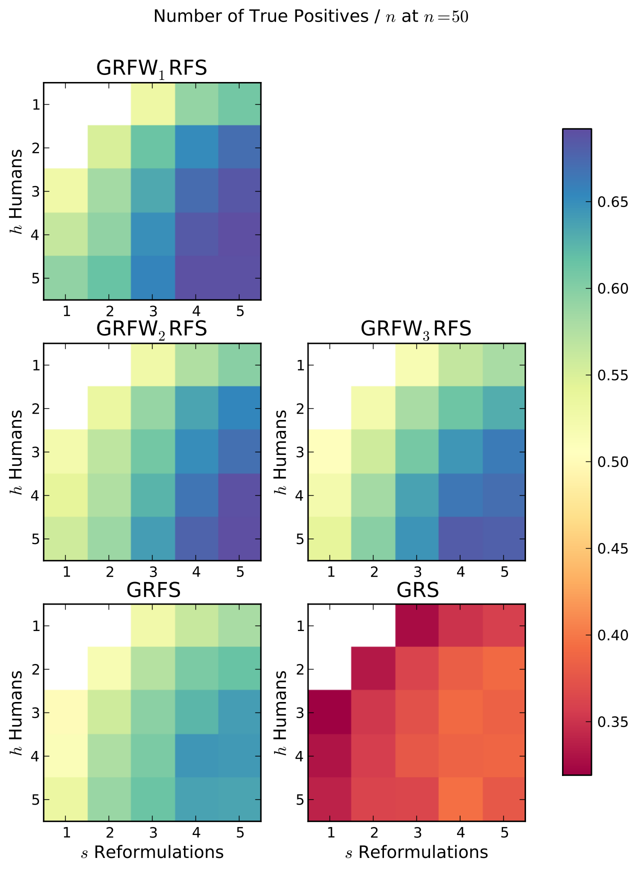


Figure 8.7: Heat map of no. of true positives for the top 50 items. Each configuration uses $\delta = 3$, $n' = 100$. The 3 white-colored cells on the top left in each grid are masked due to insufficient data. *Note: view this figure in color!*

(a total of 15). These results indicate that forcing more reformulations from a human workers prompts them to think deeper about Q, and provide more useful reformulations overall. We see diminishing returns beyond three workers providing five reformulations each.

Summary: W$_1$ *performs marginally better than* W_2 *and* W_3. *The precision improves as we increase* h *and* s *for all configurations, however having fewer human workers providing more reformulations each is better than more human workers providing fewer reformulations each.*

Question 3: Varying δ (size of retrieved sample) in GRFW$_{1-3}$RFS**:** We fixed $n' = 100$, and plotted the number of true positives in the top 50 items as a function of the number of the number of items sampled δ. The results are displayed for $h = s = 5$ in Figure 8.8, and for $h = s = 3$ in Figure 8.9.

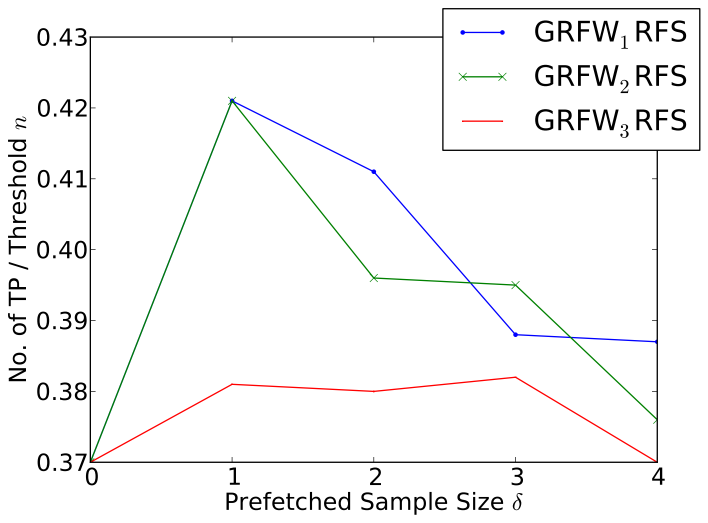


Figure 8.8: Effect of varying sampled items δ in GRFW$_{1-3}$RFS. Using $n' = 100$, $n = 100$, $h = s = 5$

We focus first on $h = s = 5$. Since the total number of items retrieved n' is fixed, there is a trade-off between *exploration* and *exploitation*: If $\delta = 1$, then a total of $h \times s \times \delta = 25$ items are sampled and evaluated, leaving us $n' - 25 = 75$ items for the second phase of retrieval. On the other hand, if $\delta = 3$, then a total of $h \times s \times \delta = 75$ items are sampled and evaluated — giving us a better estimate of which reformulations are good, however, we are left with only $n' - 75 = 25$ items to retrieve from the good reformulations. With $\delta = 1$, we do very little exploration, and have more budget for exploitation, while with $\delta = 3$, we do a lot of exploration, and as a result, have less budget for exploitation.

Figure 8.8 depicts the effects of exploration versus exploitation: the number of true positives for all three plots increases as δ is increased, and then decreases as δ goes beyond a certain value. When $\delta = 0$, the configurations are identical to one another and have the same effect as GRFS. Increasing δ by 1 gives a ≈15% improvement in precision of results with the exception of GRFW$_3$RFS. GRFW$_3$RFS (which uses a weighting component based on the agreement across reformulations) shows a dome-shaped curve which peaks at 1-3 items. As δ is increased further, the number of true positives decreases as n' is wasted on exploration rather than exploitation.

The results in Figure 8.9 are similar, however, GRFW$_2$RFS's trend is erratic. This is because taking the single best-looking reformulation may not be a robust strategy when using smaller h and s. For $\delta = 0$ and 1, for both figures, the number of true positives for GRFW$_2$RFS is similar to GRFW$_1$RFS. This is expected since the weighting approach used is similar in practice for the two corner cases.

Summary: On fixing the total number of items retrieved n', retrieving and filtering a sample of $\delta = 1$

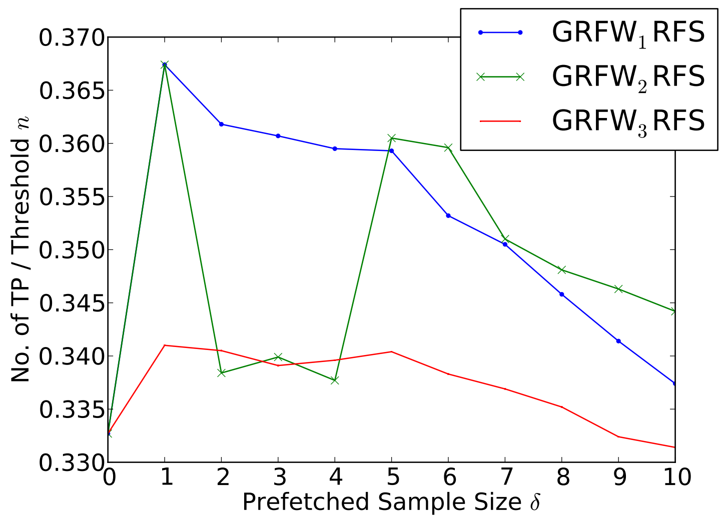


Figure 8.9: Effect of varying sampled items δ in GRFW$_{1-3}$RFS. Using $n' = 100$, $n = 100$, $h = s = 3$

items from each reformulated query is adequate to find the best queries from which to retrieve additional items.

Question 4: Varying Target Number of Items n': Figure 8.10 shows the effect of varying the number of retrieved items n' on the number of true positives in the top 50 items. We use $h = s = 4$ for each configuration, and $\delta = 3$ for GRFWRFS. As is evident from the plot, GRS is unable to effectively utilize the additional items retrieved when n' is increased. On the other hand, we see a positive trend with the other two configurations, with diminishing returns as n' increases. Note that for GRFS and GRFWRFS cost is directly proportional to n' (ignoring a fixed cost of gathering reformulations) — see Table 8.4 — so the figure still holds true if we replace the horizontal axis with cost.
Summary: The fraction of true positives increases as n' increases, with diminishing returns.

8.7 Related Work

To the best of our knowledge, we are the first in addressing the problem of designing a rich general-purpose search toolkit augmented with the power of human computation. By themselves, traditional information retrieval techniques are insufficient for our human-assisted retrieval task. On the other hand, existing crowd-powered systems, including Soylent [41], CrowdPlan [126], Clowder [65], and Turkit [130] do not address the problem of improving information retrieval.

Unlike social or collaborative search, e.g., [26,97,143,144], we do not leverage the social network,

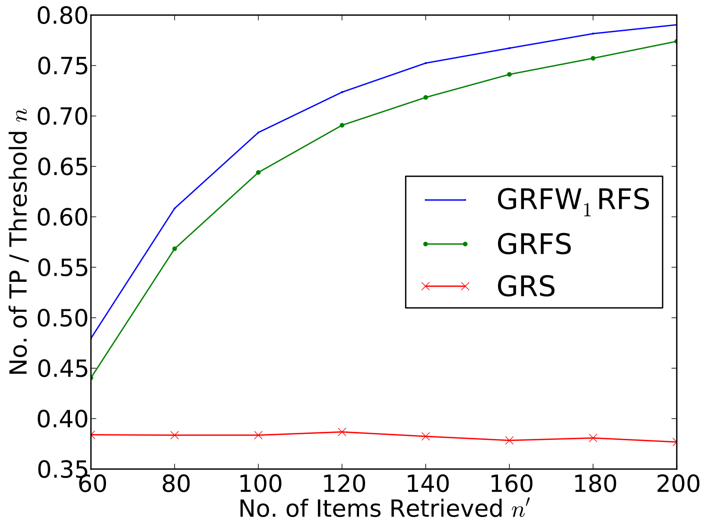


Figure 8.10: Effect of varying target number of items n'

and the system moderates the interaction using reformulations and filtering to ensure high quality results.

8.8 Conclusion

We presented DataSift, a crowd-powered search toolkit that can be instrumented easily over traditional search engines on any corpora. DataSift is targeted at queries that are hard for fully automated systems to deal with: rich, long, or ambiguous queries, or semantically-rich queries on non-textual corpora. We presented a variety of configurations for this toolkit, and experimentally demonstrated that they produce accurate results — with gains in precision of 100-150% — for textual and non-textual queries in comparison with traditional retrieval schemes. We identified that the best configuration is GRFW$_1$RFS, and identified appropriate choices for its parameters. We leveraged our optimized crowd-powered algorithms—specifically, filtering (Chapter 3)—in designing the components of DataSift.

Chapter 9

Application 2: Peer Evaluation in MOOCs

In this chapter, we introduce our second crowd-powered application: peer grading in online courses. We demonstrate that peer grading is in fact an instance of crowd-powered filtering, and we show that algorithms from Chapters 3 and 4 can be profitably employed for peer grading.

9.1 Introduction

MOOCs (Massive Open Online Courses) [36] are revolutionizing the world of education. There are hundreds of courses being offered by institutions or organizations such as Coursera [4], Udacity [23], and EdX [8], and each of these courses are being taken by thousands of students worldwide. It is estimated that (as of 2013) close to 2 Million students have signed up for accounts with Coursera, one of the primary MOOC providers [9].

Evaluating students in many of these MOOCs requires human expertise: for instance, it is impossible to grade an essay or a mathematical proof completely automatically. Further, most courses in the humanities, e.g., poetry, sociology, and literature, evaluate student learning and comprehension on more than just multiple-choice questions, requiring human expertise in evaluations.

Given that thousands of students are taking each of these courses, it is prohibitively expensive for the MOOC providers to pay graders to evaluate student submissions. As a result, these providers have turned to peer evaluations, i.e., having students evaluate each other's work, as the primary mechanism for grading.

Peer evaluation is a large scale application of crowd-powered filtering: for each student submission, the peer evaluation system needs to decide how many student graders would need to evaluate that submission in order to correctly determine the true grade for that submission. Student graders may make mistakes while evaluating submissions, and therefore, we may need multiple student graders to evaluate each submission. Since student grader time is a limited resource, we would rather have graders evaluate student submissions for which there is more uncertainty regarding the true grade, instead of submissions for which the true grade is fairly certain.

The goal this chapter is two-fold: first, we demonstrate that the crowd-powered filtering algorithms we developed in Chapters 3 and 4 are useful in the peer evaluation application and provide higher quality results than standard heuristics currently in use in the peer evaluation system; second, we evaluate our generalized algorithms in Chapter 4 and compare it to the simpler algorithms in Chapter 3.

9.2 Peer Evaluation Experiments

We describe the dataset and the setting first, followed by our experimental methodology.

Dataset Description: We validate our algorithms on a real MOOC course dataset — the Human Computer Interaction (HCI) course offered during the Fall 2012-13 quarter at Stanford. The HCI course involved around 1000 students, who were evaluated on five assignments, each containing five questions, for a total of 25 questions. Thus, the total number of student submissions (across all questions) was 25000. The course relied entirely on evaluation by peer graders to judge the quality of the student submissions for each question. Each submission was graded independently by 10 (randomly selected) student graders on average, each grader providing a score between 0–5, both inclusive, i.e., one of six scores. This dataset is ordered, that is, for every submission, the scores provided by the ten graders are listed in the order in which they were received. For each score assigned to a submission, the identity of the grader who provided the score is also recorded as part of the dataset.

Mapping to Filtering: We treat each student submission on a question as an item to be scored on a scale from 0—5 (both inclusive). Thus, we are operating under the scoring generalization described in Section 4.4.6 in Chapter 4, instead of filtering items as being YES/NO (which was the focus of Chapter 3). Since there are 25 questions, each answered by 1000 students, we have a total of 25000 items to be scored.

Grader Evaluation: The dataset also contains a set of 250 "test" submissions that were graded by all 1000 graders, as well as the course staff (instructors or TAs). These test submissions allows the peer evaluation system to calibrate the error rates of each grader prior to peer evaluation.

Since we are scoring items rather than performing binary filtering, the error rates or accuracies for each grader (or worker) w_k are of the following form:

$$p_{(i,j)}(w_k) = \Pr[\text{Score assigned by } w_k \text{ is } i | V = j]$$

for all $i, j \in 0 \dots 5$. Thus, $p_{(i,j)}$ represents the probability that a worker examines an item with staff score j, and assigns it a score of i. Since we have 6 possible scores, each grader's error rate is therefore defined by a set of 36 $p_{(i,j)}$ values. We set the grader's error rate based on his or her performance on the 250 test submissions. Our estimate of $p_{(i,j)}(w_k)$ is simply: the fraction of items whose staff scores are j that the worker w_k judged to be i instead, over the total number of items with a staff score of j.

Maximum Likelihood Score: We assume that an item can only have a score $j \in 0 \dots 5$, and that graders evaluate items with accuracies (or error rates) corresponding to $p_{(i,j)}$. Then, the *maximum likelihood score* for an item a is precisely the score $j \in 0 \dots 5$ that the item is most likely to be, based on all existing information about a (that is, all grader evaluations of a). We define this concept more formally below.

We define $L(j, a), j \in 0 \dots 5$ to be the probability that the score of item a is j, given the evidence we have. That is,

$$L(j, a) = \prod_{w_k\text{'s score for item } a = i} p_{(i,j)}(w_k)$$

Therefore, $L(j, a)$ encodes the product of the probabilities $p_{(i,j)}$ for all workers who looked at the item, and gave it a score of i, for some i. For instance, if worker w_1 gave an item a a score of 3, and worker w_3 gave a a score of 5, then:

$$L(j, a) = p_{(3,j)}(w_1) \cdot p_{(5,j)}(w_3)$$

Now, the *maximum likelihood score* of an item a, $V(a)$ is defined as the score j that maximizes $L(j, a)$:

$$V(a) = \operatorname*{argmax}_{j \in 0 \dots 5} L(j, a)$$

Thus, the maximum likelihood score of an item is the score that maximizes the product of the probabilities of the individual grader scores, across all graders who have provided scores for the item. Thus, we use the entire dataset to assign a maximum likelihood score to each item. Note that we are overloading V to mean both the "true value" of items, and the maximum likelihood score: this is because for all practical purposes, the maximum likelihood score is our best estimate of the true value of items given the entire dataset.

Overall Goal and Methodology: The goal of our experiments is to study the trade-off between expected cost and expected error for the filtering strategies output by our algorithms. Our methodology to compare the algorithms is to repeat the following for each algorithm:

- For each error threshold $\tau \in [0, 1]$, we execute the algorithm to generate a filtering strategy that obeys the expected error threshold, and is optimized for minimum expected cost.
- We then simulate a run of the generated filtering strategy on each item (i.e., each student submission) in the dataset. When the filtering strategy requests an additional human grader score while processing an item, then this score as well as the identity of the human grader who provided the score is retrieved from the dataset. That is, when the filtering strategy requests an additional grader score, it is allowed to "see" another score for the item from the dataset (in the order in which the scores were assigned by graders in the first place).
- When the simulation of the strategy on each item terminates, we record both the empirical cost (the number of scores requested for that item), and the empirical error (the difference between the maximum likelihood score—as assigned above—and the score output by the strategy for the item).
- We then measure the average empirical cost (i.e., total number of scores requested by the strategy, as a fraction of the total number of scores available in the dataset across all items), and the average empirical error (i.e., the average difference between the score assigned to an item and its maximum likelihood score, across all items).
- We repeat the procedure above for a range of τ, recording the average empirical cost and error, giving us a cost-error curve. These cost-error curves allow us to pictorially compare between algorithms over a range of cost and error values.

Algorithms: We evaluate four filtering algorithms adapted from Chapters 3 and 4. For all algorithms that we study, we use the posterior-based representation (as described in Chapter 4), wherein the state of processing is recorded using two quantities: the probability p that the item passes the filter, given answers seen so far, and cost spent so far, c. Since we wish to score items from $0 \ldots 5$, instead of performing binary filtering, the probability p is replaced with 5 probabilities $p_0, p_1, \ldots, p_4$, i.e., the probability that the item has score i, $i \in 0 \ldots 4$, given the grader evaluations seen so far. (The probability p_5 that the item has score 5 given the scores seen so far can be inferred from the remaining 5 values, and therefore need not be recorded.) Since we use the posterior-based representation, all the algorithms we study have a discretization factor δ, representing the number of intervals we divide the probability coordinate into. As we saw in Chapter 4, the larger the δ, the more fine-grained our probability estimates are, but the more time it takes to compute the strategy.

The first two algorithms we study are direct adaptations of those in Chapters 3 and 4 :

- *Single(δ):* This algorithm, for each threshold τ, generates the cost-optimal strategy assuming all workers are have the same error rate, using techniques from Chapter 3.
- *Complete(δ):* This generalized algorithm, for each threshold τ, generates the cost-optimal strategy assuming worker abilities are all distinct, using techniques from Chapter 4.

The Single algorithm has just one worker "group" — that is, all workers are assumed to have the same error rates, while the Complete algorithm has as many groups of workers as the number of workers. Here, for illustration, we will use 1000 as the number of workers. Recall from Chapter 4 that, for the approximate posterior-based representation, the complexity of strategy computation is proportional to the number of worker groups or classes. We wish to study the performance of cost and error of intermediate algorithms, where the number of worker groups is in between 1 and 1000: can we, by grouping workers together, derive the accuracy benefits of having 1000 groups, but at a lower computational cost?

To study intermediate algorithms, where the number of groups is between 1 and 1000, we use two simple schemes to cluster workers into $k, 1 \leq k \leq 1000$ groups, following which, we assume that all workers within a group have the same error rate. The two schemes (described below) come up with a single numerical value to capture the 6×6 matrix of probabilities $p_{(i,j)}$, and then use that value to cluster workers into groups:

- *Var(k, δ):* We define variance as the average difference between the scores provided by the grader and the staff scores, as observed during the testing period (as described above). This algorithm, for each value k, first places graders into k equal-sized intervals based on their variance: that is, graders are sorted based on their variance, and then we partition variance into k intervals such that the same number of graders are in each interval. This algorithm, then, for each error threshold τ, generates the cost-optimal strategy, assuming that workers within each interval have the same error rate. That is, workers in an interval are assumed to be equally capable at evaluating items.

  Note that when $k = 1000$, i.e., equal to the total number of graders, then this algorithm is identical to the Complete algorithm, since in that case, each grader will be in a partition all by himself or herself. Additionally, when $k = 1$, then this algorithm is identical to the Single algorithm, since in that case all graders will be in the same partition. Thus, this algorithm can be viewed as an algorithm that generalizes both Single and Complete, as is the Bias algorithm described next.

- *Bias(k, δ):* This generalized algorithm is the same as the previous one, except that we partition graders based on bias; we define bias as the average *signed* difference between the scores provided by the grader and the staff scores, as observed during the testing period.

We compare the four algorithms above to the Median heuristic:

- *Median:* This heuristic is currently being used in the Coursera system for peer evaluation. For each submission, the scores given by the randomly selected student graders are combined using the *median* heuristic: that is, the median of the scores for each submission is the final grade assigned to the submission.

  We can generate a cost-error curve for the median algorithm by constraining the cost to be some fraction γ of the maximum possible cost—that is, with probability γ, we include each score assigned to an item while computing the median—and then we measure the error of the median scores assigned (i.e., the difference between the median score and the maximum likelihood score, on average across all items). We repeat this for multiple γ to give us a cost-error curve.

We implemented all of our algorithms in Python. Our experiments were all conducted on a large memory (100 GB) 25 processor Ubuntu server.

Experiment 1: How much benefit do we get from optimized crowd-powered algorithms as compared to simple heuristics, and how much benefit do we get from considering worker abilities?

On comparing Single, Complete, and Median on cost and error, we find that for the same error, Complete has significantly lower cost than Single, which has significantly lower cost than Median. Additionally, on fixing cost, we find that Complete has significantly lower error than Single, which has significantly lower cost than Median. For most costs, Complete has 60% the error of Single, and 50% the error of Median.

We use the methodology described above to trace the cost-error curve for Single, Complete (both for $\delta = 10$) — denoted Single-Factor10 and Complete-Factor10 respectively, and Median. The results

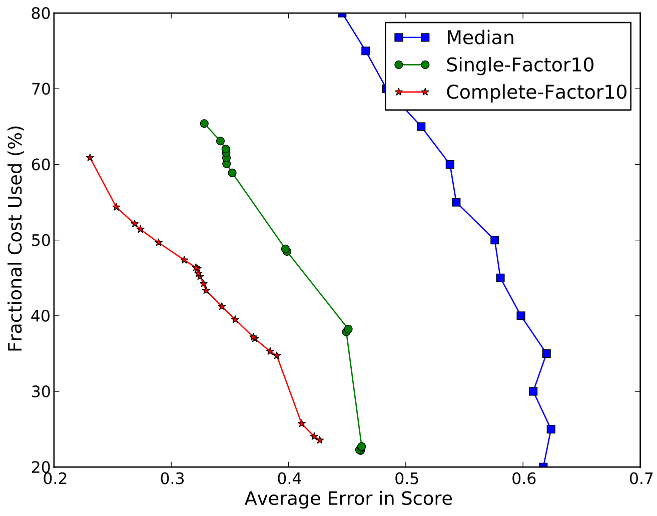


Figure 9.1: Basic Comparison

are displayed in Figure 9.1. The figure shows the fraction of the dataset that is "seen" or "consumed" by each of the algorithms (i.e., the total empirical cost) on the y-axis, versus the average difference between the maximum likelihood score and the estimated score (i.e., the average empirical error) on the x-axis. As can be seen in the figure, Complete has much lower cost and error than Single, which has much lower cost and error than Median. For instance, on fixing cost, say at 40%, which means that each of the algorithms requests 4 scores on average for each submission, Median has an error of 0.6, i.e., on average, the actual score assigned to a student is 0.6 away from the maximum likelihood score. Single, has an error of 0.45, just 75% of the error of Median. Complete, on the other hand, has an error of 0.4, just 66% of the error of Median.

Thus, both our optimized crowd-powered algorithms—Complete and Single—provide significant benefits in both cost and error over the algorithm currently used in the peer evaluation system. This is because our algorithms find strategies that are optimized to make the right decision at every possible intermediate state of processing. Further, we find that Complete does better than single; thus there are significant benefits to tracking individual grader abilities rather than assuming that all graders have the same error rate. Since Complete takes into account individual grader abilities, it can appropriately "weigh" differently the same answer coming from two different graders with different abilities. The algorithm Single, on the other hand, is not able to take this information into account. We explore this aspect in more detail, next.

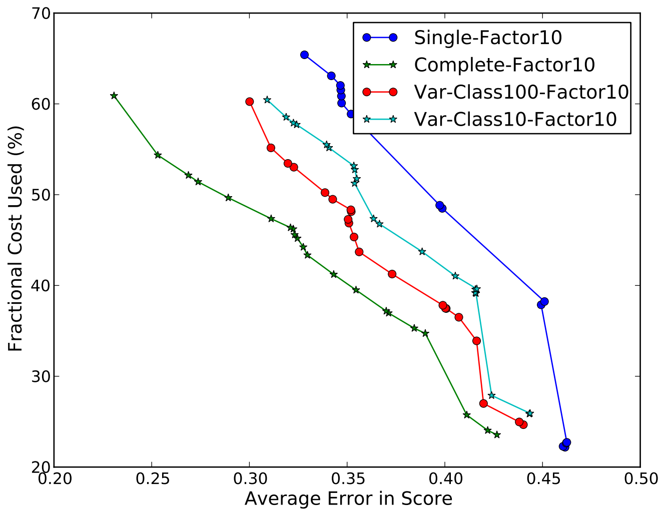


Figure 9.2: Varying Class Size

Experiment 2: How much does taking worker abilities into account impact performance? That is, how fine-grained should our worker ability partitions be?

On keeping δ fixed, increasing the number of worker partitions has the effect of reducing error for fixed cost, or vice versa. However, the impact of the number of partitions is more pronounced early on (for a small number of partitions), than later on, when the number of partitions is already large. Thus, increasing the number of partitions yields significant savings in cost even though it leads to higher computational cost while computing the strategy.

We next study how our hybrid algorithm for Variance performs in comparison with Single and Complete, on varying k, the number of worker partitions. We fix the discretization factor to be δ, and vary the number of partitions from 1 (i.e., Single), to 10, 100, and then finally to 1000 (i.e., Complete). Figure 9.2 depicts the cost-error curves for each of these four algorithms (the Variance curves are denoted Var-Class-k-Factor10 in the figure.) As can be seen in the figure, there are significant gains to be had in terms of cost and error in increasing the number of grader partitions from 1 to 10, from 10 to 100, and from 100 to 1000. For instance, if we fix the error to be around 0.35, Complete gives us a cost of 40%, Var-Class100-Factor10 (i.e., Variance with $k = 100$) has a cost of around 50%, Var-Class10-Factor10 (i.e., Variance with $k = 10$) has a cost of around 55%, and Single has a cost of around 60%.

As can be also seen in the figure, small changes in k are more likely to impact the cost-error curve when k is small, rather than when k is already large: for instance, the impact of changing k from 1 to 10 is as pronounced as the impact of changing k from 100 to 1000.

Thus, if the computing the strategy is feasible for large k, this figure shows that it is preferable to do so in order to take advantage of the additional cost savings to be had on increasing k. We consider computational cost on varying k later on.

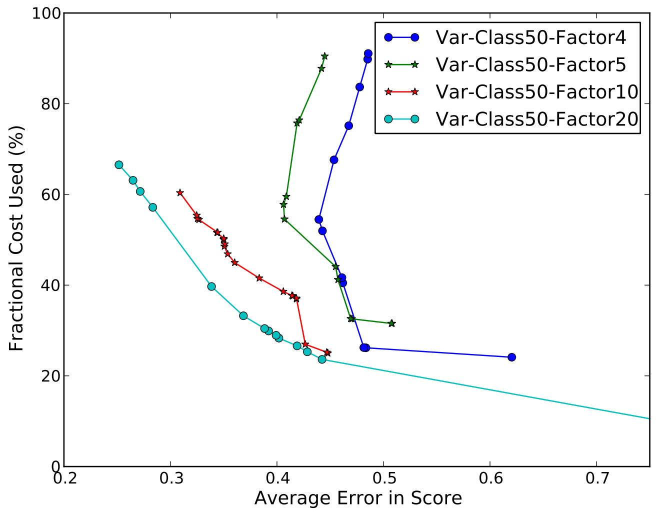


Figure 9.3: Variance with Factor

Experiment 3: How finely should we discretize our probabilities δ?

On keeping k fixed at 50, increasing the discretization factor has a significant impact on performance: that is, it has the effect of reducing error for fixed cost, or vice versa. Thus, increasing the discretization factor yields significant savings in monetary cost even though it leads to higher computational cost while computing the strategy.

For this experiment, we fix $k = 50$, and let δ be $4, 5, 10$, or 20. (These values of δ were chosen because each of these values are divisors of the number 100.) We then plot the cost-error curves for Variance for these values of δ in Figure 9.3, and for Bias for these values of δ in Figure 9.4. As can be seen in the figure, the cost-error curves for $\delta = 4$ or 5 are not as smooth as the ones for $\delta = 10$ or 20: this is because when the probability discretization is so coarse-grained, then there is a lot more

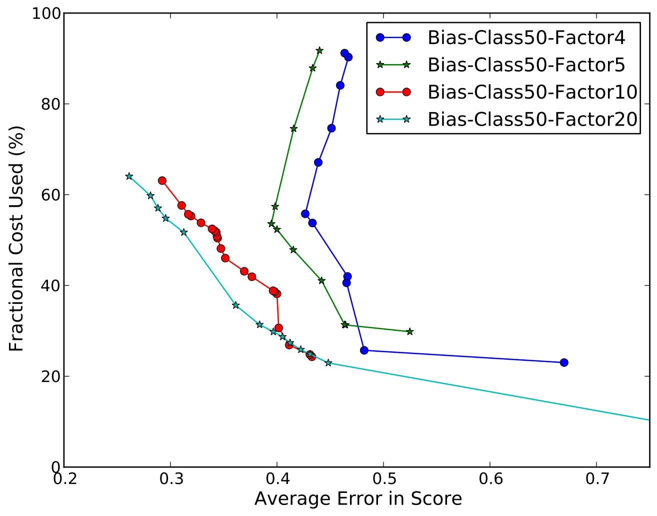


Figure 9.4: Bias with Factor

noise, and the trade-off between cost and error is less predictable.

Further, as we can see here, as we increase δ, there are significant gains in both cost and error. For instance, in Figure 9.3, for error being equal to 0.35 the cost for $\delta = 20$ is 35%, while the cost for $\delta = 10$ is 45%, an almost 30% increase. The cost-error curves for 4 or 5 never manage to achieve error 0.35.

Thus, these set of results dictate that we should use as high a δ as possible, to profit from the gains in both monetary cost and error. However, increasing δ leads to much higher computational and storage cost. In fact, in our experiments, we were not able to compute the strategy for $\delta = 25$: this is because even storing the strategy (in a memoized form) would require an array of would be $10 \times 25^5 \times 50 \times 6 \approx 30$ Billion entries, which is more than we could manage on our Ubuntu server. We will study this aspect in more detail later.

Experiment 4: How should we partition graders?

Partitioning graders on Bias or Variance leads to similar results.

In Figure 9.5, we study the difference between using Bias or Variance to partition graders. We let $k = 50$, and plot the cost-error curves for both Bias and Variance for $\delta = 10$ and 20. As can be seen in the figure, Bias and Variance perform similarly: while it seems like Variance is better for higher δ

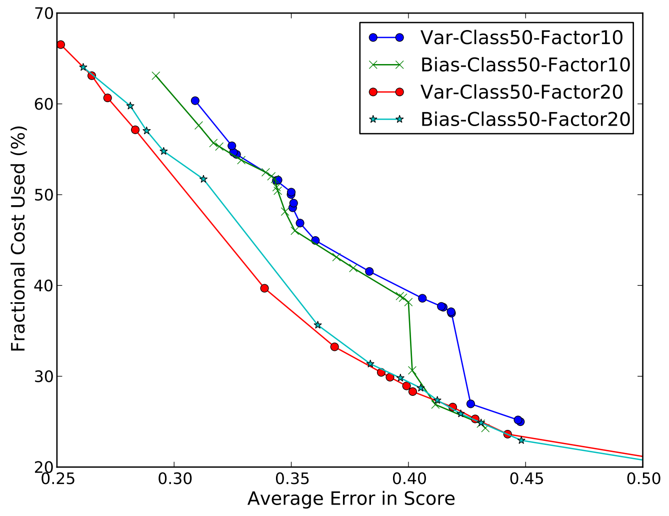


Figure 9.5: Bias Vs. Variance

and Bias is better for lower δ, these changes may be attributed to experimental noise, rather than to some systematic variation. Overall, using Bias to partition graders is just as good as using Variance.

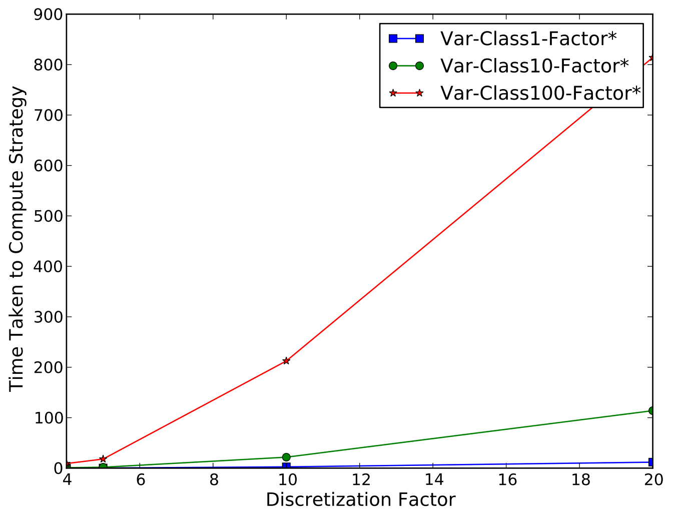


Figure 9.6: Computational Cost of Varying Class Size

Experiment 5: How does the computational cost of computing a strategy vary with k or δ?

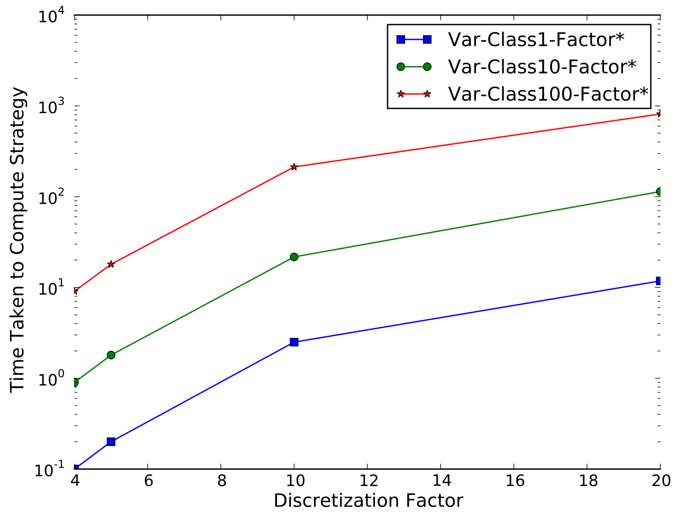


Figure 9.7: Computational Cost of Varying Class Size

The cost of computing a strategy grows linearly with k and polynomially with δ.

We focus on the Variance worker partition scheme, and plot the cost of computing the strategy in minutes versus δ for different values of k: 1 (same as Single), 10, and 100, shown in Figure 9.6 and 9.7 (Figure 9.7 is the same as Figure 9.6, but with the y-axis in log scale). As you can see in Figure 9.6, the time to compute the strategy increases very rapidly with δ: for instance, for $k = 100$, the time varies from less than 10 minutes for $\delta = 4$, to three hours for $\delta = 10$, to half a day for $\delta = 20$. The growth curve is convex (i.e., the rate of change increases as we increase δ) for each of the three plots corresponding to different k. Recall that in our analysis of the posterior-based representation for the multiple scores case (Section 4.4.6 in Chapter 4), we mentioned that the complexity of representing the strategy itself (and computing it) is proportional to a large polynomial of δ, thus the experimental results confirm the theoretical analysis.

Then, in Figure 9.7, the trend on increasing k is very clear: for each value of δ, the difference between the log of the computation time for $k = 100$ and $k = 10$ is the same as the difference between that for $k = 10$ and $k = 1$ (for all δ): Thus, (a) the ratio between the time to compute the strategy is proportional to the ratio of the k values (b) this ratio stays the same independent of δ. Thus, as predicted by theoretical analysis in Chapter 4, the time to compute the strategy is linearly proportional to k.

Overall, the cost of computing the strategy increases polynomially with δ, and linearly with k. On the other hand, the cost of storing the strategy still increases polynomially with δ, but is not dependent on k.

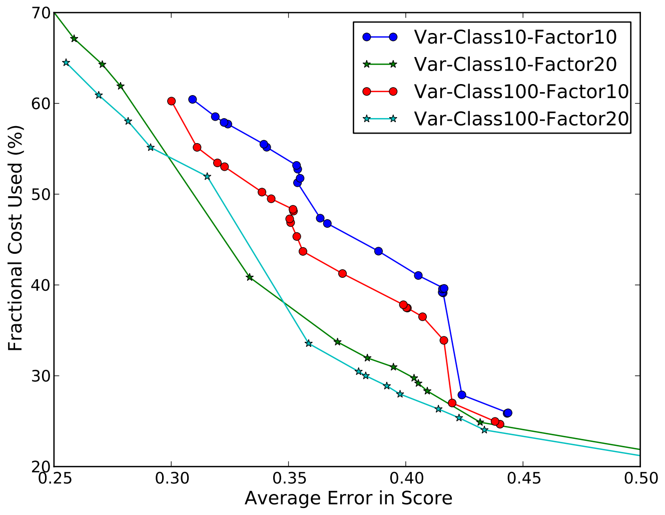


Figure 9.8: Relative Impacts of k and δ

Experiment 6: Should we increase k or δ?

Both k and δ affect cost and error significantly; however, it may be more beneficial to increase k first, since it increases the complexity linearly rather than polynomially (as in the case of δ).

We focus on the Variance worker partition scheme, and consider two values each of k and δ: $k = 10, 100$, and $\delta = 10, 20$: we plot the cost-error curves for these four algorithms in Figure 9.8. We find that the two curves for $\delta = 20$, and the two curves for $\delta = 10$ perform similarly, with the curve for $k = 100$ performing better than the curve for $k = 10$ in both cases. However, the curves for $\delta = 10$ perform worse than $\delta = 20$. Thus, δ has a larger impact on cost and error than k. This impact comes at a price: the computational complexity is proportional to a large polynomial of δ, while being linearly proportional to k. And since the number of k values is not likely to be very large (in the hundreds or thousands, rather than the millions), it may be preferable to increase k first before δ.

9.3 Conclusion

In this chapter, we demonstrated the use of our crowd-powered algorithms in yet another application: peer evaluation in MOOCs. Our algorithms are able to provide significant improvements in both cost and error (as high as 30-50% savings in cost, and 30-50% improvement in accuracy) over schemes used in practice. Furthermore, we demonstrated that our generalized algorithms from Chapter 4 perform significantly better than the simpler algorithms from Chapter 3.

However, as we showed in this chapter, there are significant computational costs in running the algorithm to derive the optimized strategy, as well as significant costs in representing and storing the optimized strategy. The computational costs are linearly dependent on k (the number of worker partitions based on ability) and polynomially dependent on δ (the strategy discretization factor). We demonstrated that there are significant benefits to increasing the strategy parameters k and δ; we should increase both k and δ as much as the computational capability allows. Since k is likely to be no more than a few hundred or a few thousand in our peer evaluation system, we prefer to increase k first, before δ. Of course, in other systems or other applications, we may wish to increase both k and δ simultaneously.

Discipline	Perspectives
Human Computer Interaction (Section 10.2)	Better interface design; novel interaction mechanisms
Machine Learning (Section 10.3)	Using humans for training data; improving crowdsourcing
Social Science (Section 10.4)	Behavioral experiments; motivations; demographic studies
Game Theory (Section 10.5)	Pricing; incentives; game design
Algorithms and Databases (Sections 10.6,10.7)	Using humans as data processors; optimization

Table 10.1: Summary of Perspectives

Chapter 10

Related Work

We organize crowdsourcing-related work in terms of research communities, the way we did in Section 1.2. We reproduce the table we had in that section in Table 10.1, listed along with the subsection of this chapter in which we discuss the work done in that discipline. Before moving to the disciplines, we describe survey articles.

10.1 Surveys

There are a number of recent surveys describing various aspects of crowdsourcing research. The article that coined the term "crowdsourcing" first appeared in Wired [38]; Quinn et al. [159] present a taxonomy of the various terms (e.g., crowdsourcing, social computation, human computation) used to describe the different ways humans may participate in computer algorithms and systems. Perhaps the most comprehensive survey on crowdsourcing is the monograph by Edith and Von Ahn [125].

Doan et al. [80] is another, albeit shorter, survey on crowdsourcing technologies.

10.2 Human Computer Interaction

The Human Computer Interaction (HCI) community has been studying two aspects of crowdsourcing: (a) development of novel crowd interfaces, along with their usage in applications, as well as novel interfaces for users or application developers to supervise crowdsourcing (b) development of games as a mechanism to interact with human workers.

10.2.1 Novel Worker and User Interfaces

The novel interfaces considered by the HCI community include: an interface for collaborative constraint satisfaction for trip planning [196], a bounding-box interface to capture dietary information in photographs of food [147], an interface that keeps workers on standby for tasks requiring low latency such as photography [40] or assistance for blind people [45], and an interface that allows collaborative editing for shortening or improving a Word document [41].

There is also work from the HCI community on improved interfaces for application developers to supervise and evaluate workers, including visualizing worker behavior [166], supervising their work [82], performing analytics on worker retention and fatigue [96], and relinquishing control of GUIs to remote workers [124].

10.2.2 Games With a Purpose

Work by Von Ahn's group has studied the design of game-based interfaces to extract useful data from human players while enticing them to continue playing [181, 182, 183, 184]. As an example, the ESP game has players collaboratively guessing tags for images, while Peekaboom has players identifying which portions of an image are the most "evocative" for a specific tag. Creating enticing games for the purpose of extracting useful data has been applied in other fields as well — FoldIt [62] has humans identifying stable 3-D configurations for proteins, while Duolingo [7] has humans translating sentences while learning a new language.

As we mentioned in Chapter 2, our optimization techniques could also prove helpful in domains where workers are not compensated monetarily for their contributions, but are instead enticed to continue contributing via "gamification", i.e., making it fun for the workers.

10.3 Machine Learning and Artificial Intelligence

The Machine Learning (ML) and Artificial Intelligence (AI) communities have been studying how crowds may be used to get better training data, and how machine learning algorithms may be used to improve crowdsourcing.

10.3.1 Active Learning

The field of Active Learning studies the problem of adaptively selecting training data to be labeled to improve the performance of machine learning algorithms. The survey by Settles et al. [169] provides a good overview of the area. Most papers in Active Learning do not assume that the labels for training data may be human-provided (or at least that they are provided by experts rather than error-prone workers) and therefore may contain mistakes, although a small fraction of papers do take mistakes into account.

Papers focusing on the theory of active learning [33,35,43,44,59,68,69,94,111,142,194] try to show that the adaptive selection strategies proposed (i.e, the procedures that select, at every point, a training example to be labeled by a human worker) provably converge to the optimal machine learning model, under some assumptions on the selection of training data to be labeled, as well as the noise in the underlying model. Some of these schemes suffer from a severe computational barrier: they explictly maintain all models that are still under consideration at each point during adaptive selection of training data points. Recent approaches, such as Importance Weighted Active Learning [44] try to eliminate this computational barrier while providing comparable guarantees.

Other papers have suggested many adaptive selection schemes that work well in practice (but provide limited to no theoretical guarantees), including, but not limited to, uncertainty sampling (picking the training data point that the current model is least certain about) [127,171], query by committee (picking the training data point that a "committee" of current models disagree about) [170], or error reduction (picking the training data point that is most likely to reduce the error).

Our work on optimized crowd-powered algorithms can also be used to generate correctly labeled training data efficiently for machine or active learning algorithms.

10.3.2 Quality Estimation

Expectation Maximization, or EM [73, 76] has been studied and used by the statistics and machine learning communities for several decades now, with many textbooks and surveys on the topic [93, 141, 185]. Expectation Maximization provides maximum likelihood estimates for hidden model parameters based on a sequence of E and M steps that converges to a locally optimal estimates for the hidden model parameters.

There have been a number of recent papers that study the use of EM to simultaneously estimate the answers to tasks and error rates of workers. These papers consider increasingly expressive models for this estimation problem, including difficulty of tasks and worker expertise [163, 189], adversarial behavior [162], and online evaluation of workers [131,188]. There has also been some work on selecting which items to get evaluated by which workers in order to reduce overall error rate [81, 112, 171]. Our work in this space has been on automatically identifying good workers [161], and getting guaranteed confidence intervals on worker abilities [108].

Our crowd-powered algorithms and systems could certainly benefit from using some of these techniques to better assess the quality of the work provided by workers. For instance, our generalized filtering algorithms in Chapter 4 assume that worker error rates are provided. These error rates could certainly come from one of the techniques listed above.

10.3.3 Decision Theory

Recent work has leveraged decision theory for improving cost and quality in crowdsourcing workflows: Dan Weld's group has used POMDPs (Partially Observable Markov Decision Processes) to design optimized workflows [52, 65, 128, 129]. In particular, they model worker behavior, task difficulty, and output quality to dynamically choose the best decision to make at any step in the workflow (refine, improve, vote, or stop), and also to dynamically switch between workflows to improve the overall "utility".

Kamar et al. [109] use POMDPs to study how to best utilize participation in voluntary crowdsourcing systems, specifically, Galaxy Zoo, an astronomical data set verified by human workers.

Our filtering strategies in Chapters 3 and 4 also use decision theory, specifically, MDPs (Markov Decision Processes); however, unlike the papers listed above, our models are simpler, enabling us to get guarantees for optimality for our filtering strategies, while performing exceptionally well in practice (as seen in Chapter 9). The papers mentioned above do not provide theoretical guarantees of any kind.

10.4 Social Science

There is a wealth of work on exploring social and behavioral aspects of crowdsourcing, typically by running experiments on crowdsourcing marketplaces (primarily Mechanical Turk [14]). These include studies on: how honest workers are [174], what kinds of tasks workers enjoy [99,177], whether crowdsourcing marketplaces are a good testbed for user studies [118,120], how pricing impacts worker behavior [98, 138], and how often spam or bias occurs [102, 145].

10.5 Game Theory and Pricing

The algorithmic game theory community has been addressing economic issues in crowdsourcing: for instance, ensuring that the marketplace is "efficient" and that workers are compensated justly, i.e, "fairness", and are incentivized to put in their best effort, i.e., "truthfulness". In particular, the community has studied incentive structures in crowdsourcing marketplaces [55, 98]; they have also studied how to improve the efficiency of crowdsourcing [110], games with a purpose [104, 105], crowdsourcing contests [53], Question-Answer (QA) forums [103], and user-generated content [88]. A recent survey [87] summarizes the recent developments in this field.

While our focus has been on minimizing the number of questions asked to human workers in designing crowd-powered algorithms and systems, we may certainly leverage research results from this community to ensure that workers are paid a fair price for their work, and are incentivized to answer truthfully.

10.6 Systems

We now describe three types of crowd-powered systems (note that we also covered some crowd-powered systems with novel interfaces within Section 10.2): full-fledged database systems that support crowdsourcing, specialized crowd-powered toolkits targeted at specific application domains, and generic programming toolkits that allow users or application developers to use crowdsourcing within programs. A tutorial on crowd-powered systems can be found in [78].

10.6.1 Crowd-Powered Database Systems

Several database groups have been building crowd-powered database systems. In nearly all of these systems, humans are regarded as a data source, and the goal is to be able to answer declarative queries by seamlessly leveraging data collected from humans as well as data stored in the system. The three primary systems are: CrowdDB [85, 86], Qurk [134, 136], and our system, Deco [153, 154, 157, 158]. These systems face the same trade-offs among cost, accuracy, and accuracy that we have encountered in the rest of the thesis. For instance, one challenge in using humans as a data source is that unlike stored data, where the data is provided immediately, humans take time to respond. Thus, all of these systems adopt some form of asynchronous processing to ensure that processing can continue instead of waiting for outstanding human-provided data. Of the three, Deco and CrowdDB are perhaps the most similar to each other. Overall, our system Deco opts for more flexibility and generality (in terms of user interfaces and uncertainty resolution), while CrowdDB makes some fixed choices at the outset to enable a simpler and easier to understand design. Qurk is a workflow system that implements declarative crowdsourcing, unlike Deco and CrowdDB which are database systems.

All of these systems can benefit from the optimized crowd-powered algorithms that we have described in this thesis for reducing cost and latency while obtaining data from the crowd, and, at the same time, ensuring the same accuracy.

10.6.2 Domain-Specific Toolkits

There are other domain-specific systems that gather data from crowds:

- Reference [54] leverages crowdsourcing for feedback in information integration pipelines. Our work [67, 151] tackles a similar problem in information extraction pipelines.
- CrowdSearcher [49, 50, 51] provides a declarative platform to leverage the user's social network as well as QA forums to solve user tasks. In recent work, CrowdSearcher has been enhanced with active rules that enable better user-driven control of crowds.
- DataMasster provides a declarative approach to leverage humans for data cleaning [77].

Our own system, DataSift (Chapter 8), focuses on search or information retrieval as an application domain.

10.6.3 Generic Toolkits

There are many generic toolkits that enable application developers to easily leverage crowdsourcing within various applications. However, none of these systems provide the functionality of optimization. In effect, it is up to the programmer or application designer to manually optimize the workflow while using these programming toolkits.

- Turkit [130], perhaps the first attempt at building a crowdsourcing toolkit, focuses on a workflow type called *iterative improvement*, applicable to one-item-at-a-time tasks like text editing or design.
- Jabberwocky [28] and CrowdForge [119] enable application developers to write parallel data processing workflows using humans.
- Automan [37] enables application developers to leverage crowdsourcing via subroutines in regular programs.

All of these toolkits could benefit from using our optimized crowd-powered algorithms.

10.7 Algorithms

Other groups have also been studying crowd-powered algorithms:

10.7.1 Sorting, Max, and Top-K

Marcus et al. [135] study different interfaces for crowd-powered sorting: they empirically find that a hybrid algorithm that uses ratings to get a rough idea of how "good" items are, and then pairwise comparisons between items in the same rating class, needs much less cost than an algorithm that uses ratings alone or pairwise comparisons alone. It remains to be seen if similar benefits can be leveraged by using different interfaces for crowd-powered maximum as well.

Davidson et al. [72] design a completely offline pairwise-comparison-based structured tournament for crowd-powered top-k and maximum problems. In Chapter 6, we designed online algorithms to select the best questions in an unstructured online setting. It may be interesting to study the benefits when the two algorithms are combined.

10.7.2 Entity Resolution and Clustering

CrowdER [186] develops algorithms for crowd-powered Entity Resolution (ER) [83]. They select clusters of entities to show to humans such that every pair of entities that are significantly similar appear together in a cluster. Our own work on active sampling [39] addresses a similar problem, but makes use of a classifier that is learned using human input. Crowd-clustering [89] tackles the question of what tasks humans should be used for when clustering a large set of items.

10.7.3 Other Algorithms

Amsterdamer et al. [32] use humans to verify data mining "association rules", Lotosh et al. [132] use humans to generate optimized plans for workflows, and Demartini et al. [75] use crowds to verify entities and relationships to enable better pattern matching for search queries. Getting it all [178] studies the problem of recovering a set of related items using the crowd (e.g., ice cream flavors, countries, restaurants). They use statistical techniques (inspired by the coupon collector problem) to estimate exactly how many times humans must be asked to provide items before the entire set is collected.

Chapter 11

Conclusions

11.1 Summary of Contributions

The focus of this thesis, as stated in the introduction, was:

> *to develop a formalism for reasoning about human-powered data processing, and use this formalism to design: (a) a toolbox of basic data processing algorithms, optimized for cost, latency, and accuracy, and (b) practical data management systems and applications that use these algorithms.*

The crowd-powered data processing algorithms that we have designed in this thesis are general-purpose, and can be deployed within crowd-powered databases or systems, as well as within stand-alone applications. These algorithms provide significant gains in latency, cost, and quality.

Our crowd-powered data management applications: namely DataSift, MOOC peer evaluation, and Deco, serve as a testbed for these algorithms, and additionally provide novel application scenarios for crowdsourcing in practice — information retrieval, database systems, and online courses.

11.2 Future Work

We now discuss open research directions. We begin with medium-term research directions, i.e., research problems that may be solved in the next five years, and then describe long-term directions, i.e., research problems that may require a longer time, but have higher potential for impact.

11.2.1 Medium-Term Research Agenda

While crowdsourcing is now "mainstream", there are still many common complaints originating both from users of crowdsourcing in practice, and from workers. We list these complaints, as well as initial approaches that may be used to solve them.

A. *User Complaint: "Crowdsourcing takes too long!"*
To deal with high human latency, we need to design systems and algorithms that provide partial results while they do their computation. However, our crowd-powered algorithms are currently optimized for reducing the latency of the eventual result rather than the latency of partial results. Prioritizing for quick partial results (e.g., minimizing the time to generate the first of many results, or providing approximate results that slowly improve over time) requires us to carefully revisit the design of algorithms and systems. We could certainly leverage principles from prior work in approximate query processing, including online aggregation [95].
Furthermore, the latency of crowdsourcing applications can be significantly reduced by more user supervision. For instance, in DataSift, the latency of the eventual outcome can be reduced significantly by having the user supervise the execution: by removing poor query reformulations, or by forcing the system to focus on some reformulations over others.

B. *Worker Complaint: "This task is badly specified!"*
Even though crowdsourcing application developers try to be as clear as possible while specifying instructions in crowdsourcing tasks, a common complaint among workers is that the tasks are ill-specified or vague. One way to deal with ill-specified or abstractly specified tasks is to use humans to break the abstract task up into smaller well-defined unit tasks, following which other humans work on the individual well-defined unit tasks. For instance, if we wish to generalize DataSift to handle queries like "find me all movies that star Nicholas Cage", or "find me a discount holiday package visiting at least two exotic locations close to each other", then, we can have humans break the query up into a workflow comprising smaller tasks. As an example, humans may specify that for the first query, the following workflow will give good results: some humans provide websites containing lists of Nicholas Cage movies, other human workers vote on them, and then other workers extract movies from the highest voted website. The workflow will need to be specified in a (possibly relational) language that is easy for the human workers to understand, generate and manipulate. Additionally, we may need to take into account input from many workers to ensure that this intermediate workflow is indeed correct.

C. *User Complaint: "My workers are terrible!"*

Many human workers view crowdsourcing platforms as easy ways to make money without effort. They spend as little time on tasks as possible, giving answers to questions without careful thought. There are various means of dealing with such workers, however, there has yet to be a principled study of cost-effective worker quality management. Certainly, the system must monitor human worker performance as time goes on, and try to bar workers from future work if their performance goes below a certain quality threshold. We have performed an initial experimental study on heuristic eviction schemes [161], but further work remains to be done.
Further, many crowdsourcing systems spend some time "training" humans, by having humans work on training tasks, after which they are provided feedback on how well they did. Then, they are allowed to work on the "real" tasks. While users of crowdsourcing seem to believe that training helps, nobody knows exactly how much training is required. For instance, having human workers train on 1000 tasks, then perform one real task, is clearly not cost effective. On the other hand, having human workers train on one task, and then perform a million tasks, may not give the best accuracy. If we view human ability as nodes in a Markov chain [187], then training can help probabilistically transport a human worker from one ability state to another ability state. A principled analysis of the Markov chain of abilities may be very useful to users of crowdsourcing.

D. ***User Complaint: "Crowdsourcing costs too much!"***
A common complaint from users is that crowdsourcing costs too much. If users wanted to filter a million items with a fairly high accuracy requirement, then the users may have to spend several tens of thousands of dollars on filtering. One way of reducing cost is by using Machine Learning (ML) algorithms in conjunction with crowdsourcing. Then, crowdsourcing is only used on items that the ML algorithm is truly uncertain about. While we have accomplished this integration with ML algorithms for crowd-powered filtering (see Chapter 4), it remains to be seen if we can do it for other crowd-powered algorithms.

11.2.2 Long-Term Research Agenda

We now describe "pie in the sky" ideas for crowd-powered data management systems that are yet to be built. Naturally, designing, optimizing, and building these systems would give rise to a host of additional algorithmic challenges as well.

A. Interactive Analytics: Interactive data analytics is an area that seems ripe for crowdsourcing. There are many users who have access to data; for instance, most local eateries have a web site, and log visitor accesses—thus, restaurant proprietors have access to data that is potentially useful to them.

However, most users (in this scenario, restaurant owners) are not aware of how to use or query this data, nor do they know how to use database management systems. One way to help such users get value out of their data is by having human workers assist in transforming the data into a structured form amenable for analysis, in formulating queries based on the what the user wants to study, and helping them visualize query results. In other words, having a human data analyst in the loop can significantly improve data analysis.

B. News Recommendations: Current automated news web sites such as Google News [12] suffer from a lack of human input in the newspaper generation process, resulting in newspapers where not all the articles are very well-written, and there is little, if any, diversity of content. On the other hand, standard newspaper sites such as the New York Times [21] have very well-written articles, but are lacking in personalization. We could combine the benefits of editorial input with automation to provide a crowd-powered personalized newspaper, where the day's top stories (personalized to the user) are voted on, categorized, and organized by human workers. Naturally, humans work in concert with automated machine learning algorithms that provide initial judgments as to whether the article may be of interest at all to the user.

C. Data Integration: While there has been a lot of progress in data integration in the last several years (see [79] for a survey), the problem is still acknowledged to be widely open. Data integration is another area that could significantly benefit from crowdsourcing. For instance, we could ask human workers questions at the schema level "do these two attributes mean the same thing?", or at the row level "do these two rows refer to the same entity?". Of course, once again, we will need to ask as few questions as possible but still be able to resolve as many disparate entities or relations as possible.

11.3 Outlook

While crowdsourcing is still very much a nascent area, it is rapidly gaining ground [20], and is expected to grow even more rapidly in the next decade. It is certainly likely that at some point in the future, most organizations will have a small core group of employees, along with a large number of crowdsourced workers available on demand. Thus, the principles and techniques developed in this thesis — saving users millions of dollars, or days of labor, while still ensuring high-quality results — could prove to be even more valuable in the days to come.

Bibliography

[1] Amazon Inc. (Retrieved 22 July 2013). *http://www.amazon.com*.

[2] Boto Web Services Library (Retrieved 22 July 2013). *https://github.com/boto/boto*.

[3] ClickWorker (Retrieved 22 July 2013). *http://clickworker.com*.

[4] Coursera Inc. (Retrieved 14 August 2013). *http://www.coursera.com*.

[5] CrowdFlower Content Moderation Platform (Retrieved 14 August 2013). *http://crowdflower.com/type-content-moderation*.

[6] CrowdFlower (Retrieved 22 July 2013). *http://crowdflower.com*.

[7] Duolingo Inc. (Retrieved 14 August 2013). *http://www.duolingo.com*.

[8] EdX (Retrieved 14 August 2013). *http://edx.edu*.

[9] Enlightening Statistics about MOOCS (Retrieved 14 August 2013). *http://www.edtechmagazine.com/higher/article/2013/02/11-enlightening-statistics-about-massive-open-online-courses*.

[10] Google Images (Retrieved 22 July 2013). *http://images.google.com*.

[11] Google, Inc. (Retrieved 14 August 2013). *http://www.google.com*.

[12] Google News (Retrieved 22 July 2013). *http://news.google.com*.

[13] Google's Basic Search Help (Retrieved 22 July 2013). *https://support.google.com/websearch/answer/134479?hl=en*.

[14] Mechanical Turk (Retrieved 22 July 2013). *http://www.mturk.com*.

[15] Microsoft Bing (Retrieved 14 August 2013). *http://www.bing.com*.

[16] MOOC on Searching the Web, Google Inc. (Retrieved 22 July 2013). *http://www.google.com/insidesearch/landing/powersearching.html*.

[17] ODesk (Retrieved 22 July 2013). *http://odesk.com*.

[18] Samasource (Retrieved 22 July 2013). *http://samasource.com*.

[19] Shutterstock Inc. (Retrieved 22 July 2013). *http://www.shutterstock.com*.

[20] Small firms, Start-ups Drive Crowdsourcing Growth, The Wall Street Journal, February 28, 2012 (Retrieved August 18, 2013). *http://online.wsj.com/article/SB10001424052970204653604577251293100111420.html*.

[21] The New York Times (Retrieved 22 July 2013). *http://nytimes.com*.

[22] Twitter Bootstrap Library (Retrieved 22 July 2013). *http://twitter.github.com/bootstrap/*.

[23] Udacity Inc. (Retrieved 14 August 2013) . *http://www.udacity.edu*.

[24] Yahoo! Inc. (Retrieved 14 August 2013). *http://www.yahoo.com*.

[25] Youtube (Retrieved 22 July 2013). *http://www.youtube.com*.

[26] L. A. Adamic, J. Zhang, E. Bakshy, and M. S. Ackerman. Knowledge sharing and yahoo answers: everyone knows something. In *WWW*, pages 665–674, 2008.

[27] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, pages 498–509, 2012. Online: http://doi.ieeecomputersociety.org/10.1109/ICDE.2012.66.

[28] S. Ahmad, A. Battle, Z. Malkani, and S. D. Kamvar. The jabberwocky programming environment for structured social computing. In *UIST*, pages 53–64, 2011.

[29] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: Ranking and clustering. *J. ACM*, 55(5), 2008.

[30] M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. In *ICALP (1)*, pages 37–48, 2009.

[31] O. Alonso, D. E. Rose, and B. Stewart. Crowdsourcing for relevance evaluation. *SIGIR Forum*, 42(2):9–15, 2008.

[32] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD Conference*, pages 241–252, 2013.

[33] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD Conference*, pages 783–794, 2010.

[34] E. Bakshy, J. M. Hofman, W. A. Mason, and D. J. Watts. Everyone's an influencer: quantifying influence on twitter. In *WSDM*, pages 65–74, 2011.

[35] M.-F. Balcan, A. Beygelzimer, and J. Langford. Agnostic active learning. *J. Comput. Syst. Sci.*, 75(1):78–89, 2009.

[36] M. Barber, K. Donnelly, and S. Rizvi. *An Avalanche is Coming: Higher Education and the Revolution Ahead*. Institute for Public Policy Research, March 2013.

[37] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor. Automan: a platform for integrating human-based and digital computation. In *OOPSLA*, pages 639–654, 2012.

[38] J. Barr and L.-F. Cabrera. Ai gets a brain. *ACM Queue*, 4(4):24–29, 2006.

[39] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. In *KDD*, pages 1131–1139, 2012. Online: http://doi.acm.org/10.1145/2339530.2339707.

[40] M. S. Bernstein, J. Brandt, R. C. Miller, and D. R. Karger. Crowds in two seconds: enabling realtime crowd-powered interfaces. In *UIST*, pages 33–42, 2011.

[41] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. In *UIST*, pages 313–322, 2010.

[42] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation*. Prentice Hall, 1989.

[43] A. Beygelzimer, S. Dasgupta, and J. Langford. Importance weighted active learning. In *ICML*, page 7, 2009.

[44] A. Beygelzimer, D. Hsu, J. Langford, and T. Zhang. Agnostic active learning without constraints. In *NIPS*, pages 199–207, 2010.

[45] J. P. Bigham, C. Jayant, H. Ji, G. Little, A. Miller, R. C. Miller, R. Miller, A. Tatarowicz, B. White, S. White, and T. Yeh. Vizwiz: nearly real-time answers to visual questions. In *UIST*, pages 333–342, 2010.

[46] C. M. Bishop and N. M. Nasrabadi. *Pattern Recognition and Machine Learning*, volume 16. 2007.

[47] P. Bohannon, S. Merugu, C. Yu, V. Agarwal, P. DeRose, A. S. Iyer, A. Jain, V. Kakade, M. Muralidharan, R. Ramakrishnan, and W. Shen. Purple sox extraction management system. *SIGMOD Record*, 37(4):21–27, 2008.

[48] P. Boldi, M. Santini, and S. Vigna. Pagerank as a function of the damping factor. In *WWW*, pages 557–566, 2005.

[49] A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *WWW*, pages 1009–1018, 2012.

[50] A. Bozzon, M. Brambilla, S. Ceri, and A. Mauri. Reactive crowdsourcing. In *WWW*, pages 153–164, 2013.

[51] A. Bozzon, M. Brambilla, S. Ceri, M. Silvestri, and G. Vesci. Choosing the right crowd: expert finding in social networks. In *EDBT*, pages 637–648, 2013.

[52] N. Bruno. Minimizing database repros using language grammars. In *EDBT*, pages 382–393, 2010.

[53] R. Cavallo and S. Jain. Efficient crowdsourcing contests. In *AAMAS*, pages 677–686, 2012.

[54] X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD Conference*, pages 87–100, 2009.

[55] D. Chandler and J. J. Horton. Labor allocation in paid crowdsourcing: Experimental evidence on positioning, nudges and prices. In *Human Computation*, 2011.

[56] K.-T. Chen, C.-C. Wu, Y.-C. Chang, and C.-L. Lei. A crowdsourceable qoe evaluation framework for multimedia content. In *ACM Multimedia*, pages 491–500, 2009.

[57] Y. Chevaleyre, U. Endriss, J. Lang, and N. Maudet. A short introduction to computational social choice. In *SOFSEM (1)*, pages 51–69, 2007.

[58] W. W. Cohen, R. E. Schapire, and Y. Singer. Learning to order things. *J. Artif. Intell. Res. (JAIR)*, 10:243–270, 1999.

[59] D. A. Cohn, L. E. Atlas, and R. E. Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, 1994.

[60] V. Conitzer, A. J. Davenport, and J. Kalagnanam. Improved bounds for computing kemeny rankings. In *AAAI*, pages 620–626, 2006.

[61] V. Conitzer and T. Sandholm. Common voting rules as maximum likelihood estimators. In *UAI*, pages 145–152, 2005.

[62] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popović, et al. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, 2010.

[63] D. Coppersmith, L. Fleischer, and A. Rudra. Ordering by weighted number of wins gives a good ranking for weighted tournaments. In *SODA*, pages 776–782, 2006.

[64] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *PPOPP*, pages 1–12, 1993.

[65] P. Dai, Mausam, and D. S. Weld. Decision-theoretic control of crowd-sourced workflows. In *AAAI*, 2010.

[66] N. N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu. A web of concepts. In *PODS*, pages 1–12, 2009.

[67] N. N. Dalvi, A. G. Parameswaran, and V. Rastogi. Minimizing uncertainty in pipelines. In *NIPS*, pages 2951–2959, 2012.

[68] S. Dasgupta, D. Hsu, and C. Monteleoni. A general agnostic active learning algorithm. In *NIPS*, 2007.

[69] S. Dasgupta and J. Langford. Tutorial summary: Active learning. In *ICML*, page 178, 2009.

[70] H. A. David. The method of paired comparisons. 1963.

[71] H. A. David. Ranking from unbalanced paired-comparison data. *Biometrika*, 74(2):432–436, 1987.

[72] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*, pages 225–236, 2013.

[73] A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Applied Statistics*, 28(1):20–28, 1979.

[74] O. Dekel and O. Shamir. Vox populi: Collecting high-quality labels from a crowd. In *COLT*, 2009.

[75] G. Demartini, B. Trushkowsky, T. Kraska, and M. J. Franklin. Crowdq: Crowdsourced query understanding. In *CIDR*, 2013.

[76] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[77] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Using markov chain monte carlo to play trivia. In *ICDE*, pages 1308–1311, 2011.

[78] A. Doan, M. J. Franklin, D. Kossmann, and T. Kraska. Crowdsourcing applications and platforms: A data management perspective. *PVLDB*, 4(12):1508–1509, 2011.

[79] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[80] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, 2011.

[81] P. Donmez, J. G. Carbonell, and J. G. Schneider. Efficiently learning the accuracy of labeling sources for selective sampling. In *KDD*, pages 259–268, 2009.

[82] S. Dow, A. P. Kulkarni, S. R. Klemmer, and B. Hartmann. Shepherding the crowd yields better work. In *CSCW*, pages 1013–1022, 2012.

[83] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[84] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[85] A. Feng, M. J. Franklin, D. Kossmann, T. Kraska, S. Madden, S. Ramesh, A. Wang, and R. Xin. Crowddb: Query processing with the vldb crowd. volume 4, pages 1387–1390, 2011.

[86] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD Conference*, pages 61–72, 2011.

[87] A. Ghosh. Game theory and incentives in human computation systems. *Handbook of Human Computation*, 2013.

[88] A. Ghosh and R. P. McAfee. Crowdsourcing with endogenous entry. In *WWW*, pages 999–1008, 2012.

[89] R. Gomes, P. Welinder, A. Krause, and P. Perona. Crowdclustering. In *NIPS*, pages 558–566, 2011.

[90] P. Gulhane, A. Madaan, R. R. Mehta, J. Ramamirtham, R. Rastogi, S. Satpal, S. H. Sengamedu, A. Tengli, and C. Tiwari. Web-scale information extraction with vertex. In *ICDE*, pages 1209–1220, 2011.

[91] P. Gulhane, R. Rastogi, S. H. Sengamedu, and A. Tengli. Exploiting content redundancy for web information extraction. *PVLDB*, 3(1):578–587, 2010.

[92] S. Guo, A. G. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD Conference*, pages 385–396, 2012. Online: http://doi.acm.org/10.1145/2213836.2213880.

[93] M. R. Gupta and Y. Chen. Theory and use of the em algorithm. *Foundations and Trends in Signal Processing*, 4(3):223–296, 2010.

[94] S. Hanneke. A bound on the label complexity of agnostic active learning. In *ICML*, pages 353–360, 2007.

[95] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997.

[96] P. Heymann and H. Garcia-Molina. Turkalytics: analytics for human computation. In *WWW*, pages 477–486, 2011.

[97] D. Horowitz and S. D. Kamvar. The anatomy of a large-scale social search engine. In *WWW*, pages 431–440, 2010.

[98] J. J. Horton and L. B. Chilton. The labor economics of paid crowdsourcing. *CoRR*, abs/1001.0627, 2010.

[99] E. Huang, H. Zhang, D. C. Parkes, K. Z. Gajos, and Y. Chen. Toward automatic task design: a progress report. In *HCOMP '10*, New York, NY, USA, 2010.

[100] O. Hudry. On the complexity of slater's problems. *European Journal of Operational Research*, 203(1):216–221, 2010.

[101] IBM, P. Zikopoulos, and C. Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.

[102] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *HCOMP '10*, New York, NY, USA, 2010.

[103] S. Jain, Y. Chen, and D. C. Parkes. Designing incentives for online question and answer forums. In *ACM Conference on Electronic Commerce*, pages 129–138, 2009.

[104] S. Jain and D. C. Parkes. A game-theoretic analysis of games with a purpose. In *WINE*, pages 342–350, 2008.

[105] S. Jain and D. C. Parkes. The role of game theory in human computation systems. In *KDD Workshop on Human Computation*, pages 58–61, 2009.

[106] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD Conference*, pages 847–860, 2008.

[107] S. R. Jeffery, L. Sun, M. DeLand, N. Pendar, R. Barber, and A. Galdi. Arnold: Declarative crowd-machine data integration. In *CIDR*, 2013.

[108] M. Joglekar, H. Garcia-Molina, and A. Parameswaran. Evaluating the crowd with confidence. In *KDD*, 2013. Online: http://dl.acm.org/citation.cfm?id=2487575.2487595.

[109] E. Kamar, S. Hacker, and E. Horvitz. Combining human and machine intelligence in large-scale crowdsourcing. In *AAMAS*, pages 467–474, 2012.

[110] E. Kamar and E. Horvitz. Incentives for truthful reporting in crowdsourcing. In *AAMAS*, pages 1329–1330, 2012.

[111] N. Karampatziakis and J. Langford. Online importance weight aware updates. In *UAI*, pages 392–399, 2011.

[112] D. R. Karger, S. Oh, and D. Shah. Budget-optimal task allocation for reliable crowdsourcing systems. In *CoRR*, volume abs/1110.3564, 2011.

[113] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.

[114] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD-88-408, EECS Department, University of California, Berkeley, Mar 1988.

[115] J. Kemeny. Mathematics without numbers. *Daedalus*, 1959.

[116] M. G. Kendall and B. Smith. On the method of paired comparisons. *Biometrika*, 31(3):324–345, 1940.

[117] C. Kenyon-Mathieu and W. Schudy. How to rank with few errors. In *STOC*, pages 95–103, 2007.

[118] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. In *CHI*, pages 453–456, 2008.

[119] A. Kittur, B. Smus, and R. Kraut. Crowdforge: crowdsourcing complex work. In *CHI Extended Abstracts*, pages 1801–1806, 2011.

[120] S. Kochhar, S. Mazzocchi, and P. Paritosh. The anatomy of a large-scale human computation engine. In *HCOMP*, New York, NY, USA, 2010.

[121] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.

[122] S. Kundu and J. Misra. A linear tree partitioning algorithm. *SIAM J. Comput.*, 6(1):151–154, 1977.

[123] R. Kurzweil. *The Singularity Is Near: When Humans Transcend Biology*. Penguin (Non-Classics), 2006.

[124] W. S. Lasecki, K. I. Murray, S. White, R. C. Miller, and J. P. Bigham. Real-time crowd control of existing interfaces. In *UIST*, pages 23–32, 2011.

[125] E. Law and L. von Ahn. *Human Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2011.

[126] E. Law and H. Zhang. Towards large-scale collaborative planning: Answering high-level search queries using human computation. In *AAAI*, 2011.

[127] D. D. Lewis and W. A. Gale. A sequential algorithm for training text classifiers. In *SIGIR*, pages 3–12, 1994.

[128] C. H. Lin, Mausam, and D. S. Weld. Crowdsourcing control: Moving beyond multiple choice. In *UAI*, pages 491–500, 2012.

[129] C. H. Lin, Mausam, and D. S. Weld. Dynamically switching between synergistic workflows for crowdsourcing. In *AAAI*, 2012.

[130] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *KDD Workshop on Human Computation*, pages 29–30, 2009.

[131] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: A crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.

[132] I. Lotosh, T. Milo, and S. Novgorodov. Crowdplanr: Planning made easy with crowd. In *ICDE*, pages 1344–1347, 2013.

[133] M. Garey et. al. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[134] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Demonstration of qurk: a query processor for humanoperators. In *SIGMOD Conference*, pages 1315–1318, 2011.

[135] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

[136] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.

[137] W. Mason and S. Suri. Conducting behavioral research on amazonfbs mechanical turk. *Behavior research methods*, 44(1):1–23, 2012.

[138] W. A. Mason and D. J. Watts. Financial incentives and the ”performance of crowds”. In *KDD Workshop on Human Computation*, pages 77–85, 2009.

[139] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, pages 110–119, 2008.

[140] A. McGregor, K. Onak, and R. Panigrahy. The oil searching problem. In *ESA*, pages 504–515, 2009.

[141] G. J. McLachlan and T. Krishnan. *The EM Algorithm and Extensions (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2 edition, Mar. 2008.

[142] P. Mineiro. Cost-Sensitive Binary Classification and Active Learning, 2003.

[143] M. R. Morris and J. Teevan. Collaborative web search: Who, what, where, when, and why. 2009.

[144] M. R. Morris, J. Teevan, and K. Panovich. What do people ask their social networks, and why?: a survey study of status message q&a behavior. In *CHI*, pages 1739–1748, 2010.

[145] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: Captchas-understanding captcha-solving services in an economic context. In *USENIX Security Symposium*, pages 435–462, 2010.

[146] H. Moulin. Choosing from a tournament. *Social Choice and Welfare*, 3(4):271–291, 1986.

[147] J. Noronha, E. Hysen, H. Zhang, and K. Z. Gajos. Platemate: crowdsourcing nutritional analysis from food photographs. In *UIST*, pages 1–12, 2011.

[148] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[149] A. Parameswaran, S. Boyd, H. Garcia-Molina, A. Gupta, N. Polyzotis, and J. Widom. Optimal crowd-powered rating and filtering algorithms. Technical report, Stanford University, 2013 (In Preparation).

[150] A. Parameswaran, M. H. Teh, H. Garcia-Molina, and J. Widom. Datasift: An expressive and accurate crowd-powered search toolkit. In *HCOMP*, 2013.

[151] A. G. Parameswaran, N. N. Dalvi, H. Garcia-Molina, and R. Rastogi. Optimal schemes for robust web extraction. *PVLDB*, 4(11), 2011. Online: http://www.vldb.org/pvldb/vol4/p980-parameswaran.pdf.

[152] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD Conference*, pages 361–372, 2012. Online: http://doi.acm.org/10.1145/2213836.2213878.

[153] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, pages 1203–1212, 2012. Online: http://dl.acm.org/citation.cfm?id=2396761.2398421.

[154] A. G. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR*, pages 160–166, 2011.

[155] A. G. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: it's okay to ask questions. In *PVLDB*, volume 4, pages 267–278, 2011. Online: http://dl.acm.org/citation.cfm?id=1952376.1952377.

[156] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. An overview of the deco system: Data model and query language; query processing and optimization. *ACM SIGMOD Record*, 41(4), December 2012. Online: http://doi.acm.org/10.1145/2430456.2430462.

[157] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.

[158] H. Park, A. Parameswaran, and J. Widom. Query processing over crowdsourced data. Technical report, Stanford University, September 2012.

[159] A. J. Quinn and B. B. Bederson. Human computation: a survey and taxonomy of a growing field. In *CHI*, pages 1403–1412, 2011.

[160] M. J. Raddick, G. Bracey, P. L. Gay, C. J. Lintott, P. Murray, K. Schawinski, A. S. Szalay, and J. Vandenberg. Galaxy zoo: Exploring the motivations of citizen science volunteers. Sept. 2009. http://arxiv.org/abs/0909.2925.

[161] A. Ramesh, A. Parameswaran, H. Garcia-Molina, and N. Polyzotis. Identifying reliable workers swiftly. Technical report, Stanford University, September 2012.

[162] V. C. Raykar and S. Yu. Eliminating spammers and ranking annotators for crowdsourced labeling tasks. *Journal of Machine Learning Research*, 13:491–518, 2012.

[163] V. C. Raykar, S. Yu, L. H. Zhao, A. K. Jerebko, C. Florin, G. H. Valadez, L. Bogoni, and L. Moy. Supervised learning from multiple experts: whom to trust when everyone lies a bit. In *ICML*, page 112, 2009.

[164] F. P. Ribeiro, D. A. F. Florêncio, and V. H. Nascimento. Crowdsourcing subjective image quality evaluation. In *ICIP*, pages 3097–3100, 2011.

[165] S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[166] J. M. Rzeszotarski and A. Kittur. Crowdscape: interactively visualizing user behavior and output. In *UIST*, pages 55–62, 2012.

[167] D. Saari. Basic geometry of voting. 1995.

[168] A. D. Sarma, A. Parameswaran, H. Garcia-Molina, and A. Halevy. Finding with the crowd. Technical report, Stanford University, June 2013.

[169] B. Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[170] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *COLT*, pages 287–294, 1992.

[171] V. S. Sheng, F. J. Provost, and P. G. Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *KDD*, pages 614–622, 2008.

[172] P. Slater. Inconsistencies in a schedule of paired comparisons. *Biometrika*, 48(3):303–312, 1961.

[173] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and fast - but is it good? evaluating non-expert annotations for natural language tasks. In *EMNLP*, pages 254–263, 2008.

[174] S. Suri, D. G. Goldstein, and W. A. Mason. Honesty in an online labor market. In *Human Computation*, 2011.

[175] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998.

[176] The Economist. The data deluge, February 2010.

[177] M. Toomim, T. Kriplean, C. Pörtner, and J. A. Landay. Utility of human-computer interactions: toward a science of preference measurement. In *CHI*, pages 2275–2284, 2011.

[178] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Getting it all from the crowd. *CoRR*, abs/1202.2335, 2012.

[179] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[180] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *ECML*, pages 437–448, 2005.

[181] L. Von Ahn. *Human computation*. PhD thesis, Pittsburgh, PA, USA, 2005.

[182] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *CHI*, pages 319–326, 2004.

[183] L. von Ahn and L. Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, 2008.

[184] L. von Ahn, R. Liu, and M. Blum. Peekaboom: a game for locating objects in images. In *CHI*, pages 55–64, 2006.

[185] B. Walczak and D. Massart. Dealing with missing data: Part ii. *Chemometrics and Intelligent Laboratory Systems*, 58(1):29 – 42, 2001.

[186] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[187] L. Wasserman. *All of Statistics*. Springer, 2003.

[188] P. Welinder and P. Perona. Online crowdsourcing: rating annotators and obtaining cost-effective labels. In *CVPR*, 2010.

[189] J. Whitehill, P. Ruvolo, T. Wu, J. Bergsma, and J. R. Movellan. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *NIPS*, pages 2035–2043. 2009.

[190] Wikipedia. Citizen science — wikipedia, the free encyclopedia, 2013. [Online; accessed 22-July-2013].

[191] Wikipedia. Unstructured data — Wikipedia, the free encyclopedia, 2013. [Online; accessed 6-July-2013].

[192] T. Yan, V. Kumar, and D. Ganesan. Crowdsearch: exploiting crowds for accurate real-time image search on mobile phones. In *MobiSys*, pages 77–90, 2010.

[193] P. Young. Optimal voting rules. *J. Econ. Perspectives*, 1995.

[194] B. Zadrozny, J. Langford, and N. Abe. Cost-Sensitive Learning by Cost-Proportionate Example Weighting, 2012. http://www.machinedlearnings.com/2012/01/cost-sensitive-binary-classification.html.

[195] O. Zaidan and C. Callison-Burch. Feasibility of human-in-the-loop minimum error rate training. In *EMNLP*, pages 52–61, 2009.

[196] H. Zhang, E. Law, R. Miller, K. Gajos, D. C. Parkes, and E. Horvitz. Human computation tasks with global constraints. In *CHI*, pages 217–226, 2012.

[197] M. Zook, M. Graham, T. Shelton, and S. Gorman. Volunteered geographic information and crowdsourcing disaster relief: a case study of the haitian earthquake. *World Medical & Health Policy*, 2(2):7–33, 2010.