2020

A new solution for Markov Decision Processes and its aerospace applications

Joshua R. Bertram
Iowa State University

A new solution for Markov Decision Processes and its aerospace applications

by

Joshua Bertram

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Peng Wei, Co-major Professor
Joseph Zambreno, Co-major Professor
Phillip Jones

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

DEDICATION

I would like to dedicate this thesis to my wife Gretchen, son Kaiden, and daughter Reyna without whose support I would not have been able to complete this work.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Peng Wei for his supportive yet challenging guidance through the masters process. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Joseph Zambreno and Dr. Phillip Jones. I would also like to acknowledge the collaboration with Xuxi Yang, Marc Brittain, and Dr. Guodong Zhu during our studies. As we completed our degrees our discussions, brainstorming, and willingness to share our knowledge helped to lift all of us up together by honing our understanding. May you all find great success and may you all remember me when you are famous!

In particular, I would like to pay special acknowledgement to Xuxi Yang for his contributions to the single peak proof in Chapter 2 and his review of the remaining proofs in that chapter. Xuxi's mentoring was instructive and instrumental in that process and I would not have been able to complete it without his help!

ABSTRACT

Markov Decision Processes (MDPs) are a powerful technique for modelling sequential decision making problems which have been used over many decades to solve problems including robotics, finance, and aerospace domains. However, MDPs are also known to be difficult to solve due to explosion in the size of the state space which makes finding their solution intractable for many practical problems. The traditional approaches such as value iteration required that each state in the state space is represented as an element in an array, which eventually will exhaust the available memory of any computer. It is not unusual to find practical problems in which the number of states is so large that it will never conceivably be tractable on any computer (e.g., the number of states is of the order of the number of atoms in the universe.) Historically, this issue has been mitigated by various means, but primarily by approximation (under the umbrella of Approximate Dynamic Programmming) where the solution of the MDP (the value function) is modelled via an approximation function. Many linear function approximation methods have been proposed since Markov Decision Processes were proposed nearly 70 years ago. More recently non-linear (e.g. deep neural net) function approximation methods have also been proposed to obtain a higher quality estimate of the value function. While these methods help, they come with disadvantages including loss of accuracy caused by the approximation, and a training or fitting phase which may take a long time to converge.

This thesis makes two main contributions in the area of Markov Decision Processes: (1) a novel alternative theoretical understanding of the nature of Markov Decision Processes and their solutions, and (2) a new series of algorithms that can solve a subset of MDPs extremely quickly compared to the historical methods described above. We provide both an intuitive and mathematical description of the method. We describe a progression of algorithms that demonstrate the utility of the approach in aerospace applications including guidance to goals, collision avoidance, and pursuit evasion. We

start in 2D environments with simple aircraft models and end with 3D team-based pursuit evasion where the aircraft perform rolls and loops in a highly dynamic environment. We close by providing discussion and describing future research.

CHAPTER 1. INTRODUCTION

This thesis presents observations and an alternative analysis of Markov Decision Processes (MDPs) which lead to a novel approach to solving certain MDPs much more efficiently than traditional methods. First we provide some introduction and context to MDPs and how to solve them so that the contributions of this thesis may be made clear.

1.1 Markov Decision Process Background

Markov Decision Processes (MDPs) are a framework for decision making with broad applications to finance, robotics, operations research and many other domains Bellman (1957); Bertsekas (1995); Powell (2007); Kochenderfer (2015); Sutton and Barto (1998). MDPs are formulated as the tuple (s_t, a_t, r_t, t) where $s_t \in S$ is the state at a given time t, $a_t \in A$ is the action taken by the agent at time t as a result of the decision process, r_t is the reward received by the agent as a result of taking the action a_t from s_t and arriving at s_{t+1}, and $T(s_t, a, s_{t+1})$ is a transition function that describes the dynamics of the environment and capture the probability $p(s_{t+1}|s_t, a_t)$ of transitioning to a state s_{t+1} given the action a_t taken from state s_t.

A policy π can be defined that maps each state $s \in S$ to an action $a \in A$. From a given policy $\pi \in \Pi$ a value function $V^\pi(s)$ can be computed that computes the expected return that will be obtained within the environment by following the policy π.

The solution of an MDP is termed the optimal policy π^*, which defines the optimal action $a^* \in A$ that can be taken from each state $s \in S$ to maximize the expected return. From this optimal policy π^* the optimal value function $V^*(s)$ can be computed which describes the maximum expected value that can be obtained from each state $s \in S$. And from the optimal value function $V^*(s)$, the optimal policy π^* can also easily be recovered. It can be shown that for a given MDP, both the optimal

policy π^* and the optimal value function V^* are unique. MDPs are interesting because their solution provides the optimal action a^* to perform from any starting state.

Though normally not used in the literature, we refer to the path taken through the state space as a result of following the optimal policy as the *optimal trajectory*.

1.2 Value Iteration

Here we briefly describe value iteration and provide a survey of the literature which describes the major approaches to working around the limitations of value iteration.

One of the most fundamental approaches to solving an MDP is *value iteration*. Value iteration is a dynamic programming approach to solving an MDP which iteratively determines the value of each state. In an infinite horizon problem with a discount factor of $0.0 < \gamma < 1.0$ the expected cumulative reward (or value) at time step t associated with a sequence of immediate rewards r_t is:

$$V(s) = \sum_{t=0}^{\infty} \gamma^t r_t(s_t, a_t) \tag{1.1}$$

More generally, this is expressed recursively using the Bellman equation Bellman (1957).

$$V_k(s_t) = \max_a \Big[r(s_t, a_t) + \gamma \sum_{s_{t+1}} T(s_{t+1}|s_t, a_t) V_{k-1}(s_{t+1}) \Big], \tag{1.2}$$

where k is the current iteration of the value iteration algorithm. Value iteration obtains the optimal value when the policy and value function become stationary with respect to the Bellman operator L satisfying the equation $V^* = LV^*$. (See Chapter 1 of Sigaud and Buffet (2013) for more information on this important topic as well as Bellman's original treatment Bellman (1957).)

Examining the run time complexity of value iteration, from Sigaud and Buffet (2013); Papadimitriou and Tsitsiklis (1987), every iteration of the value iteration algorithm takes $O(|A| \times |S|^2)$, and the overall maximum number of iterations needed by the algorithm is polynomial in $|S|$, $|A|$, and $\frac{1}{1-\gamma} \log \frac{1}{1-\gamma}$. However, in many problems the size of the state space $|S|$ itself or action space $|A|$ (known together as the state-action space) can explode exponentially either by attempting to

describe a more complex environment or by a desire to have a finer granularity of a discretized underlying continuous state-action which is represented by the state space S. In either case, as value iteration must represent each state-action with an entry in a table, as the size of the state space grows towards infinity, eventually the memory available in a computer will be exhausted rendering the problem intractable.

1.3 Markov Decision Process Related Work

Markov Decision Processes and Dynamic Programming have been researched since the 1950s and there is a tremendous amount of literature on the topic. Here we provide a view into literature related to improving value iteration with a focus on using MDPs to solve practical problems. (This by no means is an exhaustive list of available literature on the topic.)

Some algorithms address the explosion in state-action space size by taking advantage of structure of the problem to more compactly represent the MDP Schuurmans and Patrascu (2002), Guestrin et al. (2003) which can lead to performance improvements. The other major thrust is to create methods that use value function approximations or estimation techniques to avoid having to maintain a table in memory of each state Powell (2007), known collectively as Approximate Dynamic Programming (ADP). Attempts to scale to high dimensional state spaces with neural nets Mnih et al. (2013) and Monte-Carlo Tree Search (MCTS) Kocsis and Szepesvári (2006) represent recent attempts to deal with state space explosion.

Attempts have been made to improve the performance of value iteration itself. Asynchronous value iteration is a variation of value iteration that processes only certain states during each iteration which improves memory usage and can converge more quickly than value iteration in some cases Kochenderfer (2015). Prioritized sweeping is another strategy that orders the processing of the states via some metric and after updating backpropagates to predecessors of the processed state Moore and Atkeson (1993); Wingate and Seppi (2005). An excellent summary of research in this area is provided in de Guadalupe Garcia-Hernandez et al. (2012). Of particular interest, McMahan and Gordon (2005) examines stochastic shortest path problems using an approach based off Dijkstra's

algorithm (also discussing deterministic MDPs), building on work in Bertsekas (1995), where they show that deterministic MDPs can reduce to Dijkstra's algorithm which has some performance benefits over value iteration. In Dai and Hansen (2007), methods are discussed that eliminate a priority queue typically required. Of particular note is a backwards value iteration algorithm which computes value iteration from a terminating goal state, considering the problem in terms of states in which the goal state is reachable and working backwards from there. While they do eliminate the overhead of a priority queue, they retain a first-in-first-out (FIFO) queue. They similarly propose a forward value iteration algorithm that considers states that are reachable from the initial state and work forward from there, which they point out is equivalent to the LAO* algorithm in Hansen and Zilberstein (2001).

Normally an MDP is valid only for a stationary environment, in which the transition function T does not vary with time. Allowing for non-stationary environments has been studied in Szita et al. (2002) with the restriction that the changes to the transition function T are bounded by some small value ϵ. In Yu et al. (2008), reward is allowed to vary arbitrarily between time steps in a regret-based framework that relies on solving a linear program at each step. Both the environment and rewards are varied in Yu and Mannor (2009) using a robust dynamic programming method which also ultimately relies on linear programming at each time step. In Even-Dar et al. (2005), reward is allowed to change arbitrarily at each time step (possibly in an adversarial manner) in a stochastic setting where N black-box "experts" are provided; convergence bounds with respect to a fixed horizon and expected regret are provided on resulting policy changes, and performance is shown to be polynomial with respect to the N experts and $|A|$ actions, though it relies on the existence of the expert algorithms (which are not within the scope of the paper itself). In Van Seijen et al. (2017), reward functions are decomposed into simpler MDPs, each are solved with a neural net based approach similar to DQN, and the results of the simpler MDP Q-value functions are aggregated into a resulting global Q-value function, but the approach does not lead to a more fundamental understanding of how the value function is composed from the smaller MDPs. Time-dependent MDPs (TMDPs) in Boyan and Littman (2001) are used to calculate MDPs with a continuous time

dimension, claiming an exact solution in terms of piece-wise linear time steps but still relies on value iteration to approximate the true (exact) solution.

1.4 Nature and Structure of Value Function

To better illustrate the operation and result of value iteration, we will provide some examples for simple problems. The intuition we develop from these visualizations will be used as a backdrop to explain the new method.

First, in value iteration, the array representing the value function is initialized to an initial value (normally all zeros); we will assume an initial value of zero in this explanation for improved clarity. During the first iteration of the value iteration algorithm, any immediate rewards present in the environment cause the value function at the location of those rewards to be set to the immediate reward, but all states without any immediate reward remain at zero. On the second iteration, states which are one action away from states with non-zero reward obtain a non-zero value due to the Bellman operator, though the value is reduced by the discount factor γ. This effectively has caused value to "spread" from states with reward to nearby states. This process continues on subsequent states, where we observe that value from the rewards appears to diffuse through the state space. At first the spread of this reward is very dramatic with large changes occurring at each iteration of the algorithm, but as the reward spreads through the space it appears to reach an equilibrium where the changes from iteration to iteration are less noticeable and are finally nearly undetectable. In fact, the changes do in fact reduce from iteration to iteration and a small threshold for change known as a Bellman residual is used as a terminal condition for the iteration. When discussing this process, the literature refers to reaching this terminal condition as "convergence". Proofs are available that describe how value iteration is guaranteed to converge. The Bellman update is shown to be a contraction operator leading to monotonic updates toward the true value function, which in the limit leads to the optimal value function.

This spread of value through the state space is similar to the classic thermodynamics problem where a metal plate has hot locations and an iterative process is used to determine how heat spreads

through the material. Though the equations that describe heat flow are different than the Bellman equation, the evaluation process is very similar. For each grid point being evaluated for the thermal problem, the current temperature of neighboring points is examined to determine a new value for the current grid point. If neighboring points have a higher temperature, this will tend to drive the temperature of the current grid point higher. The thermodynamics problem, too, tends to produce smaller changes at each iteration as the temperatures determined by the algorithm converge on their true answer and a small threshold is used to terminate the temperature flow iterations.

Our first insight is that value iteration is a kind of diffusion process that is governed by the Bellman equation (variously referred to in the literature as the Bellman update or Bellman backup depending on the perspective of the author). We also know that for a given set of rewards, the value iteration process always converges to the optimal value function V^* and that this value function is unique for that MDP with that set of rewards. This implies that the value function is in some way predictable – if we knew what the value function would look like at the end of value iteration based off the set of rewards that are inputs to the value iteration process, we could conceivably just describe the resulting value function surface through some shortcut method if one were available.

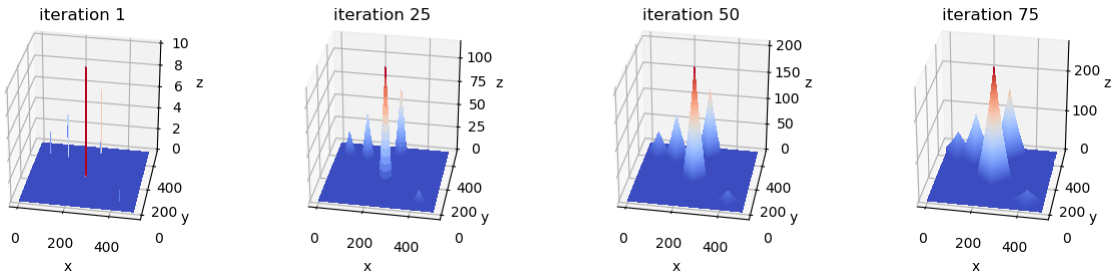

Figure 1.2: Illustration of value function over multiple iterations of value iteration algorithm for a MDP with non-terminal rewards demonstrating that value grows outward from states which contain reward. This can be considered a kind of diffusion process that eventually reaches a steady state equilibrium which is known as convergence.

If we repeatedly observe the value iteration process for randomly generated rewards, we begin to see patterns emerge. Qualitatively, we observe that positive rewards seem to form "peaks" in the value function surface. We also observe that depending on the magnitude and location of the

rewards with respect to each other, it is possible for the peaks to appear to disappear when other peaks are nearby and of large magnitude. Chapter 2 explores this idea in detail and results in a mathematical description of the value function surface which results from positive rewards in a Markov Decision Process and describes an algorithm that is polynomial in the number of rewards (but is still linear in storage with respect to the state space – an improvement over value iteration, but still not remarkable.) Chapter 3 describes a refinement which is polynomial in the number of rewards but has no dependence on the size of the state space. This is an important contribution because for many practical problems the size of the state space is too large to be represented as a table in memory.

Chapter 4 demonstrates that this new approach provides an increased level of explainability of the actions of the optimal policy. We are able to explain the actions in terms of one reward dominating the others and can quantify that level of dominance. This allows regions of dominance to be found within the MDP and makes the operation of MDPs less opaque. Additionally, an alternative view of MDP optimal policies termed the "Principle of Opportunity" is presented which captures the intuition built up through the thesis.

Chapter 5 describes how negative rewards representing a "risk well" can be incorporated into the formulation in an efficient way. Additional optimizations to the `Memoryless` algorithm are presented with results in the `FastMDP` algorithm which is linear in the number of rewards and retains independence from the size of the state space.

Chapter 6 shows the first application of these ideas to an aerospace related problem. The algorithm is used to allow a Unmanned Aerial Vehicle (UAV) to navigate through an airspace to a goal while avoiding collisions with other aircraft in the airspace. This application uses a 2D discretized state space and shows promising performance.

Chapter 7 demonstrates the algorithms ability to scale to highly dynamic environments to obtain complex behavior very efficiently. An aircraft pursuit evasion game (e.g. teamed dogfighting) is used to show two teams of aircraft trying to pursue opponents while also trying to simultaneously evade pursuit by their opponents. This requires balancing a desire to capture an opponent with

avoiding being captured, but also requires that collisions are avoided with teammates. Each frame of the simulation results in a new MDP being formulated and solved, and is therefore an excellent demonstration of the power of the approach described in this thesis. This chapter also describes some aspects of parallelism that are present in the algorithm and uses them to obtain higher performance.

CHAPTER 2. POSITIVE REWARDS: EXACT ALGORITHM

2.1 Introduction

This chapter based on Bertram et al. (2018) provides the fundamental algorithm and analysis that underpins the approach presented in this thesis. The method is shown to be exactly equivalent to Value Iteration with both proofs and extensive experimental validation. Later chapters build on this foundation by optimizing or extending the algorithm to improve performance when applied to a specific problem.

Note that in this chapter, we will use the following convention to differentiate between the state at time t with $s^{(t)}$ with a superscript and parentheses and a particular state $s_k \in S$ with a subscript. Thus, the state s_k at time t would be denoted $s_k^{(t)}$. Similarly, an action a_k and reward r_k at time t would be denoted as $a_k^{(t)}$ and $r_k^{(t)}$ respectively. A superscript by itself indicates raising to a power, as in the discount factor γ being raised to the power of t in γ^t. A state s may refer to either a state $s \in S$ or a "current" state, where a state s' always refers to a next state.

2.2 Methodology

Normally MDPs are usually considered in the literature as trees in which the current state leads to future states through available actions. We however in this chapter will describe an MDP in terms of a directed graph which is potentially cyclic. A similar description was provided in Papadimitriou and Tsitsiklis (1987) for deterministic MDPs.

As will become clear later, we adopt this convention in order to take advantage of properties that will emerge to arrive at a method to calculate the exact solution to MDPs with reward functions $R(s)$ that depend only on state s.

2.2.1 MDP Transition Graphs

An MDP can theoretically allow a transition from any state s to any other state s' by action a, which is defined by the transition function $T(s, a, s')$. A zero value for a given s, a, s' means no transition is possible, otherwise a probability from $(0, 1]$ is given and the state s and s' is defined in this chapter as *connected*. The probabilities of transition from any state s to all possible next states (including state s itself) must total 1.0.

A *transition graph* for a deterministic MDP can be defined where each node of the graph is a state s and each edge of the graph is a possible action a. The transition graph is a directed graph which may be cyclic. (Note that this is just a graphical representation of the information contained in the transition function T.)

A sample transition matrix for a 4-state, 2-action deterministic problem might be:

	action	s_0	s_1	s_2	s_3
s_0	a_0	0.0	1.0	0.0	0.0
s_1	a_0	0.0	0.0	1.0	0.0
s_2	a_0	0.0	0.0	0.0	1.0
s_3	a_0	1.0	0.0	0.0	0.0
s_0	a_1	0.0	0.0	0.0	1.0
s_1	a_1	1.0	0.0	0.0	0.0
s_2	a_1	0.0	1.0	0.0	0.0
s_3	a_1	0.0	0.0	1.0	0.0

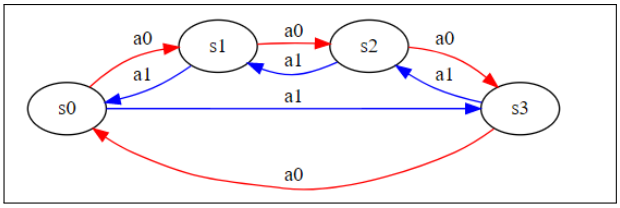

Figure 2.1: MDP states graph for a small, deterministic MDP

From this we can infer that action a_0 causes the state number to increment, and a_1 causes the state number to decrement. It is deterministic because the specified action always works 100% of the time. For a given state and action (a row of the table), the probabilities sum to 1.0. The corresponding MDP states graph is shown in Figure 2.1.

For a deterministic transition graph, the *distance* is defined as the minimum positive number of actions (or transitions) needed to move from a given state s_0 to a desired state s_k.

Formally, suppose an MDP has n states $\mathcal{S} = \{s_1, s_2, \cdots, s_n\}$. At each state, there are m actions to choose: $\mathcal{A} = \{a_1, a_2, \cdots, a_m\}$. At time t, the state is denoted $s^{(t)} \in \mathcal{S}$ and action $a^{(t)} \in \mathcal{A}$. Since this MDP is deterministic, the next state given current state and current action can be denoted as $s^{(t+1)} = T(s^{(t)}, a^{(t)})$, where $s^{(t)}$ and $a^{(t)}$ are the current state and current action, and the mapping $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the next state $s^{(t+1)}$ according to current state and action.

Suppose the initial state is $s^{(0)}$, after taking action $a^{(0)}$, the next state is $s^{(1)} = T(s^{(0)}, a^{(0)})$. After taking another action $a^{(1)}$, the third state will be $T(T(s^{(0)}, a^{(0)}), a^{(1)})$. For convenience, we denote this as $T(s^{(0)}, a^{(0)}, a^{(1)})$. More generally, if the initial state is $s^{(0)}$, after taking sequential actions $a^{(0)}, a^{(1)}, a^{(2)}, \cdots, a^{(t)}$, the agent will be at state $T(s^{(0)}, a^{(0)}, a^{(1)}, a^{(2)}, \cdots, a^{(t)})$.

Definition 1. *For a deterministic MDP with finite states, if from state s_i, after taking finite actions, the agent can reach state s_k, then the distance from s_i to s_k is defined as:*

$$\delta(s_i, s_k) = \min_t \{t | T(s_i, a^{(1)}, a^{(2)}, \cdots, a^{(t)}) = s_k\} \tag{2.1}$$

If from state s_i, no matter what sequence of actions the agent takes, it cannot reach state s_k, then the distance from s_i to s_k is defined to be:

$$\delta(s_i, s_k) = \infty \tag{2.2}$$

Finally, we define the distance from a state to itself $\delta(s, s) = 0$ for any $s \in \mathcal{S}$.

Note that for a two dimensional grid world MDP, the distance from one state to another state is just the Manhattan distance with respect to the Cartesian coordinate of the grid cells.

Definition 2. *An MDP problem is said to be a fully connected MDP if all states can be reached from all other states:* $\forall s, s' \in S, \delta(s, s') < \infty$.

By the definition of fully connected MDP, we wish to examine MDPs in which the agent can arrive at any state from any given initial state (that is, all states are potentially reachable through some set of actions.)

2.2.2 Exact Solutions for a Single Reward Source

Given the definition of MDP transition graph, we now describe the exact solution to deterministic non-terminating MDPs for a single reward source.

We define a single reward source as a reward function of the following form:

$$r(s) = \begin{cases} r_g > 0 & \text{if } s = s_g \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

where $s, s_g \in S$ is the state where reward r_g is collected.

We define the concept of a cycle which occurs in non-terminating MDPs and derive the exact value function.

Definition 3. *The* cycle *of a state* s*, which denoted as* $\mathcal{C}(s)$*, is an ordered sequence of states:* $s^{(1)}, s^{(2)}, \cdots, s^{(t)}$ *where the states in this sequence satisfy the following condition:*

There exists a sequence of action $a^{(1)}, a^{(2)}, \cdots, a^{(t+1)}$ *such that:*

$$T(s, a^{(1)}) = s^{(1)}$$

$$T(s, a^{(1)}, a^{(2)}) = s^{(2)}$$

$$\cdots$$

$$T(s, a^{(1)}, a^{(2)}, \cdots, a^{(t)}, a^{(t+1)}) = s$$

The length of the cycle, $d(\mathcal{C}(s))$ is the number of actions in the sequence $(t+1)$ that causes a return to s.

Note that if a state s has more than one cycle there always exists one cycle with finite distance. If there exists an action $a \in \mathcal{A}$ such that $T(s,a) = s$, this is also a cycle with distance 1. Note that a state s may have no cycle. Note also that a state s can have more than one cycles and that the states in these cycles do not need to be distinct. (Some states of a given cycle may be shared with other cycles.)

Definition 4. *Suppose a state s has p cycles $\mathcal{C}^1(s), \cdots, \mathcal{C}^p(s)$, where p can be finite or infinite. The* minimum cycle *of state s, which denoted as $\mathcal{C}^*(s)$, is a cycle with minimum distance:*

$$\mathcal{C}^* = \{\mathcal{C}^i | d(\mathcal{C}^i) \leq d(\mathcal{C}^j), \forall j \in \{1, \cdots, p\}\} \tag{2.4}$$

Note that a state s can have no minimum cycle, if and only if the state s has no cycles. And a state s can also have more than one minimum cycle when there are more than one cycles having same minimum distance among all the cycles.

We denote the distance of the minimum cycle of state s as $\phi(s)$.

We now describe how to calculate the value function given this definition of a minimum cycle.

Theorem 1. *Every deterministic non-terminating fully connected MDP with a single reward source has a minimum cycle.*

Proof. Assume that we reach the goal state s_g whereupon we obtain reward r_g. We must then choose an action $a \in \mathcal{A}$ which will select our next state. We know from the definition of the reward function in Equation 2.3 that immediate reward is 0 in all states other than s_g; therefore, the only way to accumulate any new reward is to take a set of actions that result in a return to state s_g, which we termed a cycle which we can denote here as D_c. We observe that since reward can only be collected at s_g, that the reward per cycle that is collected is $R_c = \gamma^{D_c} \times r_g$. We observe that R_c grows as D_c decreases, with the max occurring at $D_{c_{max}} = 1$. Thus, we prove that a cycle must exist for a single reward source, and that cycles with shorter length are preferred.

We must consider two types of transition functions, T:

Case 1: Those that allow self transitions ($s_g \rightarrow s_g$ takes one action).

For MDPs which have a transition matrix T that allow for an action to stay in the same state, it is possible for our action a above to transition from the assumed start state of s_g to a next state of $s' = s_g$. (This transition is not possible in the 2D grid world we use for illustration, but is possible for a general deterministic non-terminating MDP, so we include this case for consideration.) In this case, we say that the minimum cycle distance $\phi(s_g) = 1$ because it takes one action to go from s_g to itself.

Case 2: Those that do not allow self transitions ($s_g \rightarrow s_g$ cannot be accomplished in one action, but instead leads to some next state s' which is distinct from s_g).

From s' we can obtain the distance back to the goal state s_g with $\delta(s', s_g)$. We now consider the following possible cases of $\delta(s', s_g)$, which is 1, or a finite k. Note that our assumption of a fully connected MDP by definition means that all states are connected, meaning for all s', $\delta(s', s_g) < \infty$.

Case 2.a: $\delta(s', s_g) = 1$:

For $\delta(s', s_g) = 1$, we can then conclude that to return to the goal state s_g, we would simply take action $a^* \in \mathcal{A} | f(s', a^*) = s_g$, thus establishing a minimum cycle with distance $\phi(s_g) = 2$ from s_g back to itself.

Case 2.b: $\delta(s', s_g) = k$, where $1 < k < \infty$:

For $\delta(s', s_g) = k$ where $1 < k < \infty$, we can similarly conclude that to return to the goal state s_g, we would simply take a sequence of actions $a^{(i)} \in \mathcal{A} | f(s', a^{(1)}, a^{(2)}, ...a^{(k)}) = s_g$, thus establishing a minimum cycle distance $\phi(s_g) = k + 1$ from s_g back to itself.

Thus we have established that for a non-terminating deterministic fully connected MDP, a minimum cycle must exist. □

Theorem 2. *For a deterministic non-terminating fully connected MDP with a single reward source r_g at state s_g, the value at s_g is equal to:*

$$V(s_g) = \frac{r_g}{1 - \gamma^{\phi(s_g)}}, \tag{2.5}$$

where $\phi(s_g)$ is the minimum cycle distance for the MDP.

Proof. From Theorem 1, we know that a minimum cycle must exist for a deterministic non-terminating fully connected MDP. To determine the value at s_g, we again consider taking an action $a \in \mathcal{A}$ from s_g which leads to state s', where the distance from s' back to s_g is 0, 1, or a finite k. Note again that our assumption of a fully connected MDP by definition means that all states are connected, meaning for all s', $\delta(s', s_g) < \infty$.

Case 1: $\delta(s', s_g) = 0$:

For MDPs which have a transition matrix T that allow for an action to stay in the same state, it is possible for our action a above to transition from the assumed start state of s_g to a next state of $s' = s_g$. (This transition is not possible in the 2D grid world we use for illustration, but is possible for a general deterministic non-terminating MDP, so we again include this case for consideration.)

Starting from s_g and taking action $a^* \in \mathcal{A}^*$ as defined in case 1 of Theorem 1, we obtain an immediate reward of r_g and find that our next state $s' = s_g$. We then again take action a^* and receive immediate reward r_g, making the cumulative reward $R = r_g + \gamma \times r_g$. As we continue to take take action a^*, we find that in the limit the cumulative reward is $R = r_g + \gamma \times r_g + \gamma^2 \times r_g + \gamma^3 \times r_g ....$ Note that as $0 < \gamma < 1.0$, this is a convergent geometric

series with a limit of $\frac{r_g}{1-\gamma}$. As in Theorem 1 we have shown that the minimum cycle distance for this case $\phi(s_g) = 1$, the value at s_g can be expressed equivalently as:

$$V(s_g) = \frac{r_g}{1 - \gamma^{\phi(s_g)}} \tag{2.6}$$

Case 2: $\delta(s', s_g) = 1$:

For $\delta(s', s_g) = 1$, we know by definition that at least one action $a^* \in \mathcal{A} | f(s', a^*) = s_g$ exists that will lead back to s_g, and we define all other actions $a^- = \mathcal{A} \setminus a^*$. We know from our reward function that the only state in which reward is non-zero is s_g, thus taking an action a^* will result in reward r_g and taking an action a^- will result in no reward, thus action a^* is optimal. We may also concluded that taking a^* will result in a minimum cycle distance of $\phi(s_g) = 2$, yielding total reward of $R = r_g + \gamma \times \gamma \times r_g = r_g + \gamma^2 r_g$. If we repeat this procedure, we then obtain reward two steps later yielding total reward of $r_g + \gamma^2 r_g + \gamma^4 r_g$. In the limit, the cumulative reward (in other words, the value) is a geometric series:

$$\begin{aligned} V(s_g) &= r_g + \gamma^2 r_g + \gamma^4 r_g + ... \\ &= \frac{r_g}{1 - \gamma^2} \\ &= \frac{r_g}{1 - \gamma^{\phi(s_g)}} \end{aligned} \tag{2.7}$$

Case 3: $\delta(s', s_g) = k$, where $1 < k < \infty$:

For $\delta(s', s_g) = k$ where $1 < k < \infty$, we can similarly conclude that to return to the goal state s_g, we would simply take a sequence of actions $a^{(i)} \in \mathcal{A} | f(s', a^{(1)}, a^{(2)}, ...a^{(k)}) = s_g$, yielding total reward of $r_g + \gamma^{k+1} r_g$ and a minimum cycle distance $\phi(s_g) = k + 1$. If we repeat this procedure,

we then obtain reward $k+1$ steps later yielding total reward of $r_g + \gamma^{k+1} r_g + \gamma^{2(k+1)} r_g$, and so on. In the limit, the reward is:

$$\begin{aligned} V(s_g) &= r_g + \gamma^{k+1} r_g + \gamma^{2(k+1)} r_g + \gamma^{4(k+1)} r_g + ... \\ &= \frac{r_g}{1 - \gamma^{k+1}} \\ &= \frac{r_g}{1 - \gamma^{\phi(s_g)}} \end{aligned} \tag{2.8}$$

Thus we have established the value at the state s_g where reward r_g is collected. □

Theorem 3. *The value function for a deterministic non-terminating fully connected MDP with a single reward source with discount factor* $0 < \gamma < 1$ *has the form:*

$$V(s) = \gamma^{\delta(s, s_g)} \times V(s_g) \tag{2.9}$$

where $s \in \mathcal{S}$*, and where* $V(s_g) = \frac{r_g}{1-\gamma^{\phi(s_g)}}$ *and* $\phi(s_g)$ *is the minimum cycle distance for the MDP, as established in Theorem 2.*

Proof. Given then, that we now know that a minimum cycle exists, and that we know the value at the state s_g where reward r_g is collected, we turn to examine the value at all other states. Given our definition of the reward function for a single source MDP, we note here that in all states other than s_g, no reward is collected. We will prove by induction.

Let us now assume that we start not at state s_g, but at some state one action away from s_g, which we will denote as $s^{(1)} | \delta(s^{(1)}, s_g) = 1$. Note that we already know from Theorem 2 the value we will obtain once we reach state s_g, which we refer to here as $V(s_g)$. We therefore know that since we must take one step to obtain this value $V(s_g)$, then the future discounted reward is then $\gamma \times V(s_g)$.

As no immediate reward is collected at state $s^{(1)}$ we know that the expected value at state $s^{(1)}$ is then simply the future discounted reward: $V(s^{(1)}) = \gamma \times V(s_g)$.

$$\begin{aligned} V(s^{(1)}) &= \gamma \times V(s_g) \\ &= \gamma^{\delta(s^{(1)}, s_g)} \times V(s_g) \end{aligned} \tag{2.10}$$

We now consider the the case where we have a minimum cycle distance of $s^{(n)}|\delta(s^{(n)}, s_g) = n$ and $s^{(n+1)}|\delta(s^{(n+1)}, s_g) = n+1$. We can see clearly from the definition of the cycle distance that for any state $s^{(n+1)}$ there exists an action $a^* \in \mathcal{A}|f(s^{(n+1)}, a^*) = s^{(n)}$. Also given that reward is only collected at s_g, we again have no immediate reward when transitioning from state $s^{(n+1)}$ to state $s^{(n)}$ and need only consider future discounted reward. Thus:

$$V(s^{(n+1)}) = \gamma \times V(s^{(n)}) \tag{2.11}$$

This means that:

$$\begin{aligned} V(s^{(2)}) &= \gamma \times V(s^{(1)}) \\ &= \gamma \times \gamma^{\delta(s^{(1)}, s_g)} \times V(s_g) \\ &= \gamma \times \gamma \times V(s_g) \\ &= \gamma^{\delta(s^{(2)}, s_g)} \times V(s_g) \end{aligned} \tag{2.12}$$

Then by induction, we see that the value of any state s is as follows, completing the proof:

$$V(s) = \gamma^{\delta(s, s_g)} \times V(s_g) \tag{2.13}$$

□

2.2.3 Exact Solution for Multiple Reward Sources

We move now to discuss how to find the value function when multiple reward sources are present. First, we define multiple reward sources as having $N > 0$ positive rewards $R = \{r_1, ...r_N\}$, where we will refer to the number of rewards as $|R|$. We introduce some terms to help us describe the algorithm. Informally, we describe a state where the value function increases due to acquiring a reward as a *peak*, and each reward will generate a peak.

If we assume that there is a peak value v_g at state s_g, then we will term the operation of calculating the whole value function from a peak as *propagating reward* and will denote this for reward r_g at state s_g generally as:

$$\mathcal{P}_g(s) = \gamma^{\delta(s,s_g)} \times v_g \tag{2.14}$$

where $\delta(s, s_g)$ is the distance from s to s_g. Note that this simply corresponds to the discounted future reward from state s with respect to reward r_g, but is a convenient notational shorthand.

Definition 5. *We use the term* baseline peak*, $\mathcal{B}^i$, to describe the a single reward source r_i located at state s_i which is collected infinitely but has no other rewards in its minimum cycle. When the context is clear or we are speaking generally of a baseline peak, we may drop the i and denote the baseline peak as $\mathcal{B}$. The value at the baseline peak is:*

$$\mathcal{B}^i = \frac{r_i}{1 - \gamma^{\phi(s_i)}}$$

The value function for the baseline peak $\mathcal{B}^i$ is a mapping $\mathcal{P} : \mathcal{S} \to \mathbb{R}$:

$$\mathcal{P}_{\mathcal{B}^i}(s) = \gamma^{\delta(s,s_i)} \times \frac{r_i}{1 - \gamma^{\phi(s_i)}} \tag{2.15}$$

Definition 6. *We use the term* combined peak *to describe a primary reward source at state s_p which is collected infinitely and has a secondary reward source at state s_s within the primary state's minimum cycle. We denote the value of the combined peak at s_p as $\Gamma^{p,s}$ (or Γ^p or even Γ when the context is clear.) The value function for a combined peak is*

$$\mathcal{P}_{\Gamma^{p,s}}(s) = \mathcal{P}_{\mathcal{B}^p}(s) + \mathcal{P}_{\mathcal{B}^s}(s) \tag{2.16}$$

For rewards that are collected just once, we refer to the increase to the value function of collecting this reward as a *delta peak*. This represents a "bump" in the value function at the state where the reward is collected, which is propagated outward.

Definition 7. *A delta peak for a reward r_i is calculated by adding the reward r_i at state s_i to some pre-existing value function $\mathcal{P}^j(s)$ formed by propagation. At s_i, the value of the delta peak is $\Delta^i = r_i + \mathcal{P}^j(s_i)$. The value function for the delta peak is formed by propagation:*

$$\mathcal{P}_{\Delta^i}(s) = \gamma^{\delta(s,s_i)} \times \Delta^i \tag{2.17}$$

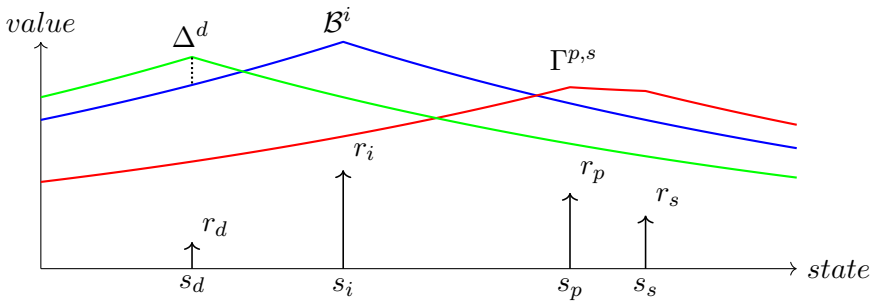

Figure 2.2: Illustration of baseline (blue), combined baseline (red), and delta baseline (green)

2.2.4 Algorithm

The algorithm, which we have named `Exact` , is designed to maintain a sorted list of all valid possible peaks at any time. Each iteration, it selects the maximum peak from the list and this peak

is considered *processed*. The processed peak has at least one affected reward (combined peaks have more than one); all peaks that are composed from the affected rewards are removed from the list.

Baseline peaks and combined peaks rely only on the value of the reward, and are therefore pre-calculated and added to the reward list before the first iteration. Delta rewards however depend on the value function at each iteration and are therefore calculated at the beginning of each iteration. Processing continues until the list of possible peaks is empty.

In addition to the proof provided in the appendix, the algorithm was additionally verified by generating test scenarios where randomly sized grid worlds with randomly generated rewards. The number of rewards varied between 1 and $|S|$. The MDP was solved with value iteration, and then the result was used to verify the operation of our algorithm. Hundreds of thousands of these randomly generated scenarios were used to ensure no corner cases were missed.

Algorithm 1 `Exact`

```
1:  procedure EXACT (rewardSources)
2:      valueFunction ← preallocate array of zeros
3:      processedPeaks ← empty list
4:      sortedPeaks ← PrecomputePeaks( rewardSources )
5:      while sortedPeaks is not empty do
6:          deltaPeaks ← ComputeDeltas( valueFunction )
7:          sortedPeaks ← PruneInvalidPeaks()
8:          maxPeak ← max( [ sortedPeaks, deltaPeaks ] )
9:          sortedPeaks ← RemoveAffectedPeaks( maxPeak )
10:         valueFunction ← UpdateValueFunction()
        return valueFunction
```

Line 2 initializes memory to hold the value function. Line 3 initializes an empty list to track which peaks have been processed by the algorithm. Line 4 pre-computes baseline peaks and combined peaks based off a list of reward sources and stores them in the form of a sorted list, sorted by value of each peak. Lines 5-10 continue until we have exhausted the potential peaks and each iteration of the loop whittles away at the list of possible peaks. Line 6 computes delta peaks for any remaining reward sources utilizing the value function that has been computed so far. Line 7 removes any peaks that have become invalid. Line 8 selects the peak with maximum value. Line 9 removes any other

potential peaks in the list that are affected by selecting the peak with maximum value. (For example, a combined peak with states 3 and 4 are selected. The baselines for states 3 and 4 would then be removed.) Line 10 then updates the value function based off the newly selected peak with maximum value.

```
1: procedure PrecomputePeaks(rewardSources)
2:     list ← empty SortedList
3:     for all rewardSources do
4:         list.add( baseline peak for reward source )
5:     for all rewardSources do
6:         nbr ← find neighboring state with highest reward
7:         if nbr is not empty then
8:             list.add( cycle peak for reward source )
       return list
```

Line 2 initializes a sorted list that is sorted by value of the peaks. In Lines 3-4, a baseline peak is computed for each reward sourcea. In lines 5-8, if any reward sources are next to each other, their combined peaks are computed.

PrecomputePeaks() is a $O(|R| \times |A|)$ algorithm that is done one time at the beginning of the algorithm and yields a list with worst case length of $O(|R| \times |A|)$ entries (but only if the reward sources are all adjacent to each other).

```
1: procedure ComputeDeltas
2:     list ← empty SortedList
3:     for all reward sources do
4:         compute delta of reward and value function
5:         nbr ← find neighboring state with highest value
6:         list.add(max(deltapeak, neighborvalue))
```

Line 2 initializes a sorted list that is sorted by value of the peaks. Lines 3-6 compute delta peak for any reward sources that remain. Line 5-6 properly sort the delta with respect to neighboring states.

ComputeDeltas(valueFunction) is a $O(|R| \times |A|)$ algorithm that is done for each pass of the loop.

```
1: procedure PruneInvalidPeaks
2:     for all remaining peaks do
3:         nbr ← find neighboring state with highest value
4:         if nbr > peak then
5:             list.remove( peak )
```

Lines 2-5 remove any peaks that have become invalid.

PruneInvalidPeaks() is a $O(|R| \times |A|)$ algorithm that is done for each pass of the loop, but this also shrinks by $O(|A|)$ entries each pass.

```
1: procedure RemoveAffectedPeaks(list, state)
2:     for all remaining peaks do
3:         if peak is affected by state then
4:             list.remove( peak )
```

Lines 2-4 remove any peaks that have been eliminated by the choice of the peak with maximum value.

RemoveAffectedPeaks operates over the $O(|R| \times |A|)$ *sortedPeaks* list, but this also shrinks by $O(|A|)$ entries each pass.

```
1: procedure UpdateValueFunction(value_function, peak)
2:     interim ← Propagate(peak)
3:     valueFunction ← element-wise-max( interim, valueFunction)
```

Line 2 propagates the peak outward to compute an interim value function for the reward source selected during this iteration. Line 3 then performs an element-wise max operation over the value function computed during the previous iteration and the interim value function, resulting in the value function for this iteration.

UpdateValueFunction is a $O(|S|)$ operation due to the call to *Propagate*.

```
1: procedure Propagate(peak)
2:     valueFunc ← allocate empty array of zeros
3:     for all states do
4:         valueFunct[state] ← peak × discount^ConnDist
       return valueFunc
```

Line 2 creates a value function with all zeros. Lines 3-4 compute the value function based off the peak's value, the distance through the transition graph, and the discount factor.

Propagate() is a $O(|S|)$ operation.

```
1: procedure ConnDist
2:     dist ← manhattan distance between start and end state
3: return dist
```

Line 2 calculates the distance through the transition graph. Note that because our 2D grid world has a known structure, we can take advantage of this knowledge to perform our distance calculation in constant time rather than having to perform a shortest path search through the graph. This algorithm will receive an important performance boost whenever this is possible. (The overhead of performing a graph search through the transition graph for a general MDP may outweigh the benefits of this algorithm. That is an open question for future work.)

ConnDist() for this 2D grid world is a $O(1)$ constant operation.

2.2.4.1 Time Complexity

Because this algorithm still requires the full value function to be computed, this drives an underlying $O(|S|)$ time complexity for creating the data structure and updating the value function at the end of each pass.

In general the time complexity of this algorithm is $O(|R|^2 \times |A|^2 \times |S|)$, where $|R|$ is the number of reward sources, $|A|$ is the number of actions, and $|S|$ is the number of states.

For environments where the connected distance is not easily determined (arbitrary transition graph), then the complexity to determine the distance between states must be taken into consideration. However, it is assumed that this can be precomputed offline because T is assumed to be stationary.

For environments like the 2D grid world where the structure of the space is known, determining the connected distance between states is a $O(1)$ calculation.

2.2.4.2 Memory Complexity

Memory complexity for the algorithm is $O(|S| + |R| \times |A|)$

2.2.5 Proof of Algorithm Correctness

We remind the reader that the proofs for the algorithm in this section are claimed to only apply to a narrow subset of MDPs:

1. Deterministic non-terminating MDPs
2. Reward function based only on state (not action)
3. Only positive, real rewards (no negative rewards)

While we will explore in future papers whether this method can be applied to a larger class of MDPs, we will start with this narrow definition. To prove that our algorithm results in an optimal value function, we must prove the resulting value function satisfies Bellman optimality equation $V^* = LV^*$. This is a complex, multi-step proof. In Part 1, we establish that optimal value function is reached when the maximum possible value is found at each state. In Part 2, we show that our algorithm's effect is to determine the maximum possible value at each state, thereby satisfying the Bellman optimality equation.

2.2.6 Part 1: Bellman optimality and maximum value

The Bellman equation and Bellman operator have been well studied by many sources. For completeness, we repeat proofs available elsewhere such as Sigaud and Buffet (2013) regarding monotonicity, contraction over the max norm, and the uniqueness of the fixed point solution V^*.

Monotonicity: First, we repeat the well known property that the Bellman operator L satisfies the property of monotonicity, which means that for any two value functions V and V' and given $V \leq V'$, then $LV \leq LV'$.

Proof. Note that the $\leq$ operator here is an element-wise operator: $V \leq V' \rightarrow \forall s, V(s) \leq V'(s)$.

We then translate the inequality to a more convenient equivalent form:

$$LV \leq LV'$$

$$LV - LV' \leq 0$$

$$L[V(s)] - L[V'(s)] \leq 0, \forall s$$

Then, for all $s \in S$:

$$\begin{aligned} L[V(s)] - L[V'(s)] &\leq 0 \\ R(s) + \max_a \left[\gamma \sum_{s'} T(s,a,s')V(s')\right] - R(s) - \max_a \left[\gamma \sum_{s'} T(s,a,s')V'(s')\right] &\leq 0 \\ \max_a \left[\gamma \sum_{s'} T(s,a,s')V(s')\right] - \max_a \left[\gamma \sum_{s'} T(s,a,s')V'(s')\right] &\leq 0 \end{aligned}$$

Since we know that $V \leq V'$, we then know that:

$$\begin{aligned} \max_a \left[\gamma \sum_{s'} T(s,a,s')V(s')\right] - \max_a \left[\gamma \sum_{s'} T(s,a,s')V'(s')\right] \leq \\ \max_a \left[\gamma \sum_{s'} T(s,a,s')V(s')\right] - \max_a \left[\gamma \sum_{s'} T(s,a,s')V(s')\right] \\ \max_a \left[\gamma \sum_{s'} T(s,a,s')V(s')\right] - \max_a \left[\gamma \sum_{s'} T(s,a,s')V'(s')\right] \leq 0 \end{aligned}$$

□

Contraction mapping: We then recall that the Bellman operator is a contraction over the max norm $|\cdot|_\infty$.

Proof. A contraction operator means that for any two functions f and g,

$$|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|,$$

where, again, the $\leq$ operator is taken to be element-wise and is true for all a.

Assume that $\max_a f(a) \geq \max_a g(a)$, that $a^* = argmax_a f(a)$ so that $\max_a f(a) = f(a^*)$. Then,

$$|\max_a f(a) - \max_a g(a)| = f(a^*) - \max_a g(a)$$

Given our assumption that $\max_a f(a) \geq \max_a g(a)$, then $f(a^*) \geq g(a^*)$ and $f(a^*) - g(a^*)$ is positive. Then:

$$|\max_a f(a) - \max_a g(a)| \leq f(a^*) - g(a^*)$$

Then from the definition of absolute value (also given that $f(a^*) - g(a^*)$ is a positive value):

$$\begin{aligned} |\max_a f(a) - \max_a g(a)| &= |f(a^*) - g(a^*)| \\ &= \max_a |f(a) - g(a)| \end{aligned}$$

Then, to prove that the Bellman operator is a contraction mapping, we must prove that:

$$||LV - LV'||_\infty \leq \gamma ||V - V'||_\infty$$

From the definition of the max norm, for all $s \in S$:

$$|LV(s) - LV'(s)| \leq \gamma ||V - V'||_\infty$$

$$|LV(s) - LV'(s)| = |R(s) + \max_a \gamma \sum_{s'} T(s,a,s')V(s') - R(s) - \max_a \gamma \sum_{s'} T(s,a,s')V'(s')|$$
$$= |\max_a \gamma \sum_{s'} T(s,a,s')V(s') - \max_a \gamma \sum_{s'} T(s,a,s')V'(s')|$$

From our contraction mapping proof above, we can then use the result to say:

$$|LV(s) - LV'(s)| = \max_a |\gamma \sum_{s'} T(s,a,s')V(s') - \gamma \sum_{s'} T(s,a,s')V'(s')|$$
$$= \max_a \gamma |\sum_{s'} T(s,a,s')(V(s') - V'(s'))|$$
$$= \max_a \gamma \sum_{s'} T(s,a,s')|V(s') - V'(s')|$$
$$\leq \max_a \gamma \sum_{s'} T(s,a,s')|V(s') - V'(s')|$$
$$\leq \max_a \gamma \sum_{s'} T(s,a,s')||V - V'||_\infty$$

Given that we know $\sum_{s'} T(s,a,s') = 1$ for a given a:

$$|LV(s) - LV'(s)| \leq \max_a \gamma ||V - V'||_\infty$$
$$\leq \gamma ||V - V'||_\infty$$

□

Stationary: Then, given that it is a contraction mapping over the max norm, we repeat the well known property that the Bellman operator has a unique solution V^* and that this optimal solution is fixed (or stationary under L).

Proof. We prove by contradiction. Assume that there are two value functions V, V' that are both fixed points under L, $V = LV$ and $V' = LV'$.

However, from our contraction mapping proof, we know that $||LV - LV'||_\infty \leq \gamma||V - V'||_\infty$. But with our assumption that $LV = V$ and $LV' = V'$, this becomes:

$$||LV - LV'||_\infty = ||V - V'||_\infty$$

Recalling that $0 < \gamma < 1$, then we have the contradiction that:

$$||V - V'||_\infty \leq \gamma||V - V'||_\infty$$

which could only be true if V and V' are all zeros, or if $V = V'$, which in both cases reduces to $V = V'$, proving that there must only be one unique fixed point solution for the Bellman operator L. □

Theorem 4. *Given the optimal policy π^* and the associated value function V^*, the optimal value function has maximum value at each state:*

$$V^* \geq V^\pi, \forall \pi$$

Proof. Given the monotonicity of the Bellman operator L, we know that at each step of value iteration $LV \leq LV'$. Given that the Bellman operator is a contraction, we also know that successive applications of the Bellman operator converge to the optimal policy π^* with a corresponding value

function V^*. And because we know that the optimal value function V^* is a unique, fixed point solution of $V^* = LV^*$, we know that once we reach the optimal solution under the contraction we will never diverge from it. Thus the sequence of value functions is $V_0 \leq V_1 \leq V_2 \cdots \leq V^* \leq V^* \leq V^*...$ and we can then conclude that $\forall s, \pi : V^*(s) \geq V^\pi(s)$. □

Thus, to prove that the algorithm satisfies the Bellman optimality equation, we must show that the algorithm determines the maximum possible value at each state s.

2.2.7 Part 2: Algorithm calculation of maximum value

We turn now to examine the way in which reward is collected and how we can determine whether the algorithm in fact calculates the maximum value at every state.

2.2.7.1 Zero versus Positive Reward

We start first with the observation that the states can be broken into two general categories: those with reward, which we define as $S^+ = s \in S | R(s) > 0$, and those without reward, which we define as $S^Z = s \in S | R(s) = 0$. (Recall that our definition of the reward function permits only positive rewards and that the rewards are based on the state and not on the action. At this time, we do not claim to have solved the problem of rewards based on the action.) Note that S^Z and S^+ may be a collection of disjoint subsets of S.

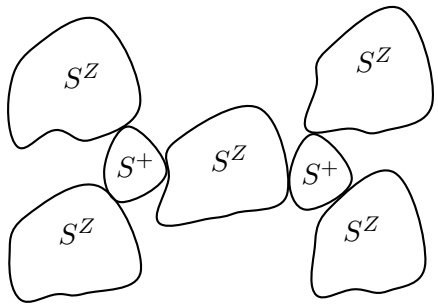

Figure 2.3: Disjoint subsets of S^Z and S^+ that form S.

Theorem 5. *The maximum of the value function cannot occur in states where the reward is 0.*

Proof. If we examine the recursive form of the Bellman equation at the optimal policy π^* with the (stationary) V^*:

$$V^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s')V^*(s'), \tag{2.18}$$

then for $s_z \in S^Z$ the value $V^*(s_z)$ is the discounted future reward. Thus, if $a^* = argmax_a \sum_{s'} T(s_z, a, s')V^*(s')$, then $V^*(s_z) = \gamma \sum_{s'} T(s_z, a^*, s')V^*(s')$. And given that the discount factor $0 < \gamma < 1$, we see that $V^*(s_z) \leq \sum_{s'} T(s_z, a^*, s')V^*(s')$. Furthermore, if $V^*(s') > 0$, then $V^*(s_z) < \sum_{s'} T(s_z, a^*, s')V^*(s')$, which is to say that $V^*(s_z)$ can only be equal to $V^*(s')$ if both are zero. (We do not need to prove it here, but we will state that this can only occur if the value function is zero everywhere.)

Thus, as we know that $V^*(s_z)$ is strictly less than $V^*(s')$ we know that a maximum of the value function cannot occur for $s_z \in S^Z | R(s_z) = 0$. □

The converse then is that if the maximum of the value function V^* cannot occur where $R(s) = 0$ (that is in S^Z), then it must occur where $R(s) > 0$ (that is, in S^+).

Theorem 6. *States with reward of zero, S^Z, are determined from the states with non-zero reward, S^+.*

Proof. From the relation $V^*(s_z) = \gamma \sum_{s'} T(s_z, a^*, s')V^*(s')$ that was developed in the previous proof, we can conclude that for all $s_z \in S^Z$, the resulting value function is determined solely by the value at another state, through the discounted future reward. Thus, to know the value for any state $s_z \in S^Z$, we must look to another state to define the value.

Consider a chain of states in S^Z, $\{ s_z^{(1)}, s_z^{(2)}, \cdots, s_z^{(n)} \}$ and suppose that each element in the chain is the result of the optimal action at each step that satisfies $a* = argmax_a \gamma \sum_{s'} T(s_z, a, s')V^*(s')$. What can we say of the value of these states? We can say nothing, as none of them have any immediate value $R(s) > 0$. Let us say that at the next optimal action a^* we reach a state in S^+. At this point in time we can definitively say that $V(s_z^{(1)}) > 0$ as the state in S^+ has an immediate

reward greater than 0, and thus through the discount factor all states in our chain obtain some positive value.

Let us repeat this experiment for all states in S^Z. In general, starting from any state $s_z \in S^Z$ and taking the optimal action a^* at each step, we will form a chain of $s_z^{(1)}, s_z^{(2)}, \cdots, s_z^{(n)}, s_p$ with length $n+1$ that will terminate in a state $s_p \in S^+$. At each step in the chain due to zero immediate reward in S^Z, the value $V(s_z^{(k)}) = \gamma \times V(s_z^{(k+1)})$, where $k = \{1...(n-1)\}$. And finally when the chain terminates at $s_p \in S^+$, $V(s_z^{(n)}) = \gamma * V(s_p)$. Thus by induction we have shown that all states in S^Z have a value that is determined by a state in S^+. □

Illustrating this concept:

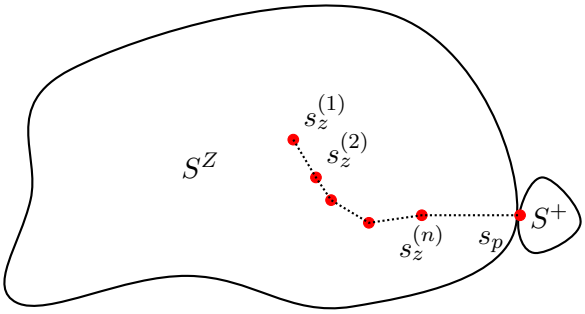

Figure 2.4: Optimal sequence of actions through S^Z until a point in S^+ is reached.

Thus, we have established that all states where reward is zero are deterministic with respect to states with positive reward, and that the maximum of the value function cannot occur where reward is zero. Thus, in order to fully determine the value function, we need only consider the states where reward occur. This is a key conclusion that underpins the algorithm.

We now expand on the nature of the maximum value at each state of the optimal value function.

Theorem 7. *Given the optimal policy π^* and the associated value function V^*, the optimal value function at each state is equivalent to:*

$$V^*(s) = \max_{\pi} V^{\pi}(s), \forall s \in S$$

Proof.

$$V^* \geq V^\pi, \forall \pi$$
$$V^*(s) \geq V^\pi(s), \forall \pi, \forall s \in S$$
$$\geq \max_\pi V^\pi(s), \forall s \in S$$

As we know by definition that $V^* \in V^\pi$ and we have already proven that V^* is in fact the maximum value at each state, we can strengthen our statement with:

$$V^*(s) = \max_\pi V^\pi(s), \forall s \in S$$

□

Thus we need only prove that the algorithm produces the maximum value at each state in S^+.

2.2.7.2 Reward Collection

We now examine the possible ways that each reward can possibly be collected:

Definition 8. *Given a policy π, select an initial state $s^{(0)}$ and follow the policy. The resulting path through the state space is $p = \{s^{(i)}, ...\}, \forall i = \{0...\infty\}$. If a given state $s_k = s^{(i)}$ for some i, then we say that s_k has been visited. If a reward r_k is present at s_k, then we additionally say that reward r_k has been collected. We denote the count of the number of times that r_k has been collected as N_k.*

Theorem 8. *All rewards $R = \{r_1, r_2, ..., r_N\}$ are collected either once or infinitely under a given policy π. That is, for a given reward $r_i \in R, N_k = \{1, \infty\}$, and only rewards falling within a minimum cycle of a local maximum in the value function are collected infinitely.*

Proof. To prove that all rewards in an MDP are collected at least once, we note that a the optimal policy is the optimal action from all states $s \in S$. Given that all rewards $R(s)$ must by definition fall within an $s \in S$, then we can conclude that every reward will be collected at least once.

For the remainder of the proof we make the simple observation that every point in the value function is by definition either a local maximum or not a local maximum at at given state s_i.

From this, if a state $s_i \in S$ is not a local maximum, then the optimal action a^* will cause the state s_i to be exited in favor of a next state s_j, which is a neighbor state of s_i with maximum value. This process will continue until a local maximum is reached.

When a local maximum s_M is reached, we necessarily then enter a minimum cycle $C^*(s_M)$ which by definition is a cycle where a primary reward and optionally one or more secondary rewards are collected infinitely. Formally, a local maximum is thus defined as: $V(s_i) \geq V(s) \forall s \in S | \delta(s, s_i) = 1, \forall s_i \in \mathcal{C}^*(s_M)$.

Therefore, a given reward must be collected once or infinitely, and only rewards in a minimum cycle (which is a local maximum) are collected infinitely. □

We note that the propagation operator $\mathcal{P}$ forms an exponential decay curve from the peak value.

The value functions for the baseline and delta baseline are simply the propagation of the peaks, $\mathcal{P}_{\mathcal{B}^i}(s)$ and $\mathcal{P}_{\Delta^d}(s)$ respectively.

Theorem 9. *The value function for the combined baseline is the sum of the baselines for each peak.*

Given two reward sources r_p at state s_p and r_s at state s_s, where $r_p \geq r_s$ and r_s is within the minimum cycle of r_p, the value function for the combined peak is equal to the sum of the baselines of each peak:

$$V(s) = \mathcal{P}_{\mathcal{B}^p}(s) + \mathcal{P}_{\mathcal{B}^s}(s) \tag{2.19}$$

Proof. For any state $s \in \mathcal{S}$, we will show that the value function in 2.19 satisfies the Bellman Optimality Equation.

For the case $s = s_p$, we have:

$$V(s_p) = r_p + \gamma \max_a V(T(s, a)) \tag{2.20}$$

It should be noted that $\max_a V(T(s,a)) = V(s_s) = \gamma\mathcal{B}^p(s) + \mathcal{B}^s(s)$, since any other action will take the agent to state with distance 1 to s_p and distance 2 to s_s, which will have value $\gamma\mathcal{B}^p + \gamma^2\mathcal{B}^s$, which is less than $V(s_s)$. Thus we have:

$$\begin{aligned} V(s_p) &= r_p + \gamma(\gamma\mathcal{B}^p + \mathcal{B}^s) \\ &= \mathcal{B}^p + \gamma\mathcal{B}^s \\ V(s) &= \mathcal{P}_{\mathcal{B}^p}(s) + \gamma\mathcal{P}_{\mathcal{B}^s}(s), \forall s \in S \end{aligned} \tag{2.21}$$

which is consistent with the value function in 2.19.

For the case $s = s_s$, it is similar to the case $s = s_p$.

For the case $s \neq s_p$ and $s \neq s_s$. We first note that for a 2D grid world MDP with two neighboring reward state s_p, s_s, the effect of an action is to lead the agent one step further from the one reward state, e.g. s_p or one step nearer to this reward state. Assuming for our current state s, the distance from s to s_p , denoted as $\delta(s, s_p)$, is n. And the distance from s to s_s, denoted as $\delta(s, s_s)$ is $n+1$ (it can also be $n-1$, and the proof would be similar). Then after one action, $\delta(s, s_p) = n-1$ or $\delta(s, s_p) = n+1$, and $\delta(s, s_s) = n$ or $\delta(s, s_s) = n+2$. Then according to Bellman Equation, we have

$$V(s) = \gamma \max_a V(T(s,a)) \tag{2.22}$$

Since $\gamma^{n-1}\mathcal{B}^p > \gamma^{n+1}\mathcal{B}^p$ and $\gamma^n\mathcal{B}^s > \gamma^{n+2}\mathcal{B}^s$, the action that leads to $\delta(s, s_p) = n-1$ and $\delta(s, s_s) = n$ will be the optimal action. So we have

$$\begin{aligned} V(s_p) &= \gamma(\gamma^{n-1}\mathcal{B}^p + \gamma^n\mathcal{B}^s) \\ &= \gamma^n\mathcal{B}^p + \gamma^{n+1}\mathcal{B}^s \\ &= \gamma^{\delta(s,s_p)}\mathcal{B}^p + \gamma^{\delta(s,s_s)}\mathcal{B}^s \\ V(s) &= \mathcal{P}_{\mathcal{B}^p}(s) + \gamma\mathcal{P}_{\mathcal{B}^s}(s), \forall s \in S \end{aligned} \tag{2.23}$$

which is consistent with the value function in 2.19. □

Thus we have identified the three possible ways that a reward in S^+ can be collected. How do we then select between these alternatives in order to find the optimal value function V^*?

2.2.7.3 Constructing Value Function

Here we show that the optimal value function is formed from a combination of the baselines defined in the previous section.

Definition 9. *Let $R = \{r_1, r_2 ... r_N\}$ be the set of rewards sources in an MDP, and let $|R| = N$ be the number of reward sources. Let the set of all possible baseline value functions be $\mathcal{P}_\mathcal{B} = \{\mathcal{P}_{\mathcal{B}_1}, ... \mathcal{P}_{\mathcal{B}_N}\}$. Similarly, let the set of all possible combined baseline value functions be $\mathcal{P}_\Gamma = \{\mathcal{P}_{\Gamma_1}, ... \mathcal{P}_{\Gamma_N}\}$ and the set of all possible delta baseline value functions be $\mathcal{P}_\Delta = \{\mathcal{P}_{\Delta_1}, ... \mathcal{P}_{\Delta_N}\}$. Then let $\mathcal{M} = \mathscr{P}(\mathcal{P}_\mathcal{B} \cup \mathcal{P}_\Gamma \cup \mathcal{P}_\Delta)$ be the power set of all possible baselines and $M \subset \mathcal{M}$ be one such selected combination of baselines. We denote a specific value function for a baseline with within M as M_i.*

Definition 10. *For a specific combination of baselines $M \in \mathcal{M}$, we define the value function V^M as the maximum value over all value functions in M:*

$$V^M(s) = \max_{M_i \in M} M_i(s), \forall s \in S$$

We denote the set of all value functions formed by $\mathcal{M}$ as $V^\mathcal{M} = \{V^M\}, \forall M \in \mathcal{M}$.

Definition 11. *We denote as V^α as the region between the optimal value function V^* and the zero-function $V_\emptyset(s) = 0$.*

$$0 \leq V^\alpha(s) \leq V^*(s), \forall s \in S$$

We pause now to consider these definitions and informally relate them to traditional well known intuitions between policies and value functions. We note that traditionally V^π is defined as the set of value functions formed by all possible policies $\pi \in \Pi$. We also note that value iteration iteratively searches through a countably infinite set of functionals that estimate V^*, asymptotically approaching V^*, and that the set of such functions becomes finite when a stopping criterion such as the bellman

residual is used. We note that there are an uncountably infinite number of functions $f(s) \in V^{\alpha}$, many of which cannot be part of V^{π} because no policy can generate these functions under the MDP.

In general, a policy π^M can be extracted from any value function $V^M \in V^{\mathcal{M}}$, and this π^M is guaranteed to fall within Π because Π by definition contains all possible policies for the given MDP definition.

We can think of V^M as considering a subset of the original MDP problem, where the state and action space are identical, but with a subset of the rewards. Therefore, when a policy is extracted from V^M and then applied to the full MDP formulation, a function in V^{π} is generated. Thus, generally, $V^{\mathcal{M}}$ lies outside of V^{π}. However, $V^{\mathcal{M}}$ and V^{π} both contain the optimal solution V^* (which will be proven below) and thus the optimal solution within $V^{\mathcal{M}}$ is also an optimal solution within V^{π}.

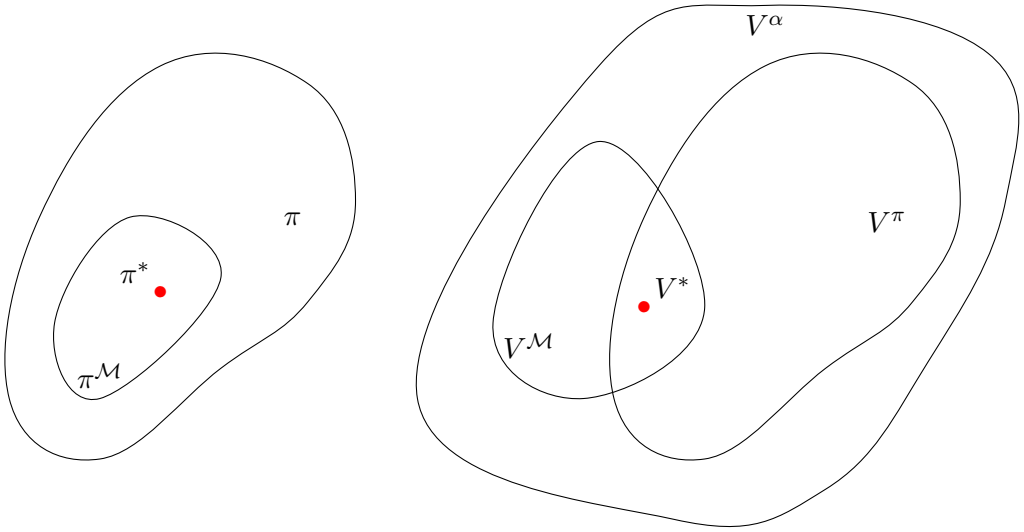

Figure 2.5: Depiction of the relationship between policy, value function, and optimal solution for $V^{\mathcal{M}}$

Theorem 10. *At every state in S^+, there is a value function in $V^{\mathcal{M}}$ that is at least as large as any in V^{π}:*

$$\forall s \in S^+, \max_{M \in \mathcal{M}} V^M(s) \geq \max_{\pi \in \Pi} V^{\pi}(s)$$

Proof. From the previous section, we have identified the three possible ways that a reward r_i at state $s_i \in S^+$ can be collected: the baseline $\mathcal{P}_{\mathcal{B}^i}(s)$ with peak value $\mathcal{B}^i$, the combined baseline $\mathcal{P}_{\Gamma^i}(s)$ with peak value Γ^i, and the delta baseline $\mathcal{P}_{\Delta^i}(s)$ with peak value Δ^i which we will denote as the set $\mathcal{M}_{s_i} \subset \mathcal{M}$.

Given that the set $\mathcal{M}_{s_i}$ represents the values functions that can possibly result at state s_i, then there must be a maximum among them such that $\exists m_{max} \in \mathcal{M}_{s_i} | \forall m \in \mathcal{M}_{s_i}, \mathcal{M}_{s_i}(m_{max}) \geq \mathcal{M}_{s_i}(m)$. The maximum possible value at s_i is then defined by $\mathcal{M}_{s_i}(m_{max})$ and is thus equal to $V^*(s_i)$. Given then that $V^*(s_i)$ is an upper bound at s_i for both $V^\pi(s_i)$ and $\mathcal{M}_{s_i}(m_{max})$:

$$\forall s_i \in S^+, \max_{M \in \mathcal{M}_{s_i}} V^M \geq \max_{\pi \in \Pi} V^\pi(s_i)$$

□

We can extend the above to cover the entire value function:

Theorem 11. *At every state in the whole of S, there is a value function in $V^{\mathcal{M}}$ that is at least as large as any in V^π:*

$$\forall s \in S, \max_{M \in \mathcal{M}} V^M(s) \geq \max_{\pi \in \Pi} V^\pi(s)$$

Proof. Given that we now know from the previous theorem the maximum value for all states in S^+, then from Theorem 6 we can say that the value of all states in S are known.

To show that the values in S^Z are maximum, we recall that the propagation operator $\mathcal{P}$ forms an exponential decay curve from the peak value v_p at state s_p of the form:

$$\forall s \in S, \mathcal{P}_p(s) = v_p \times \gamma^{\delta(s,s_p)},$$

where $\delta(s, s_p)$ is the distance from s to the peak at s_p.

The exponential decay curve has the property that at a given state s_i, if two peak values p_1 and p_2 are considered, and supposing that $p_1 \geq p_2$, then $\forall s \in S, \mathcal{P}_{p_1}(s) \geq \mathcal{P}_{p_2}(s)$. Thus, if we know

the peak of the value functions in $s \in S^+$ are maximum, then we know that the corresponding exponential decay curve is maximum in S^Z as well. □

Theorem 12. *The optimal value function V^* lies within $V^{\mathcal{M}}$ and is in fact the element-wise maximum of all value functions in $V^{\mathcal{M}}$.*

$$\forall s \in S, V^*(s) = \max_{M \in \mathcal{M}} V^M(s)$$

Proof. From the above proofs, we know that at any state $s \in S^+$, the maximum possible value is $V^{max}(s) = \max_{M \in \mathcal{M}} V^M(s)$, and we know that the states in S^Z can be determined by a peak in S^+. However, there are multiple such peaks in S^+ which might determine the value of a given state $s_z \in S^Z$.

Recalling that the optimal value function is the maximum possible value at every state $s \in S$, and therefore that it is the maximum possible value at every state $s_z \in S^Z$, it is clear then that the maximum value at s_z must be the maximum of all possible value functions in M evaluated at s_z.

$$\forall s_z \in S^Z, V^*(s_z) = \max_{M \in \mathcal{M}} M(s_z)$$

Given that $V^M(s) = \max_{M_i \in M} M_i(s), \forall s \in S$, this is equivalent to:

$$\forall s_z \in S^Z, V^*(s_z) = \max_{M \in \mathcal{M}} V^M(s_z)$$

Thus we now know the maximum value at every state in both S^Z and S^+, and therefore S as a whole:

$$\forall s \in S, V^*(s) = \max_{M \in \mathcal{M}} V^M(s)$$

□

We may therefore conclude that the optimal value function V^* is the max over each state $s \in S$ of the value function from the possible combinations of the peaks in $\mathcal{M}$. This forms the core of the algorithm and completes the proof that the algorithm calculates the optimal value function V^*.

2.3 Experiments

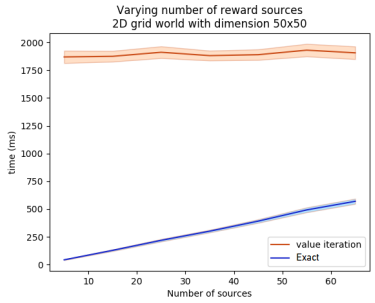

Figure 2.7 Varying number of reward sources

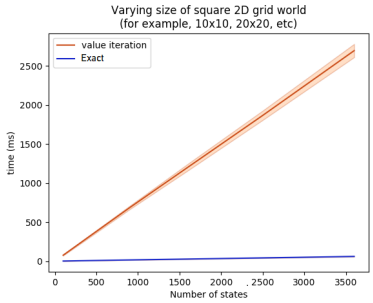

Figure 2.8 Varying number of states

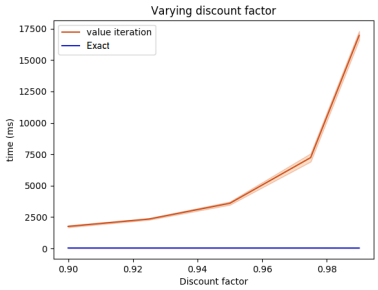

Figure 2.9 Varying discount factor

Figure 2.10: Experimental results showing the improved performance of the algorithm as compared to value iteration when varying the number of reward sources, varying the number of states, and varying the discount factor.

Figure 2.7 shows the effects of varying the number of reward sources on the performance of the algorithm. For this result, a 50x50 grid world was used. The x-axis shows the number of reward sources used for a test configuration and the y-axis shows the length of time required to solve the MDP. For each test configuration, 1000 randomly generated configurations were created for the number of reward sources specified in the test configuration with reward values ranging from 1 to 10. For each generated configuration, value iteration and our proposed algorithm (named `Exact`) were run to obtain performance measurements. As an additional check, the exact solution calculated by this algorithm was compared to the value iteration result to ensure they produced the same result (within a tolerance due to value iteration approximating the exact solution with the use of a bellman residual as a terminating condition.) In the plot, the bold line is the average and the colored envelope shows the standard deviation for each test configuration.

The figure shows that as the number of reward sources increases, value iteration remains invariant of the number of reward sources. For the algorithm proposed in this chapter, for small numbers of reward sources the algorithm clearly outperforms value iteration. As the number of reward sources increases, however, we expect an intersection point will occur and value iteration will begin to perform better.

Figure 2.8 shows the effects of varying the size of the state space on the performance of the algorithm. For this a fixed number of reward sources (5) were used, and only the size of the state space was varied (by making the grid world larger). The x-axis shows the number of states in the grid world (e.g., $10 \times 10 = 100, 50 \times 50 = 2500$) and the y-axis shows length of time required to solve the MDP. For each grid world size, 1000 randomly generated reward configurations with the fixed number of reward sources were generated. The results show that value iteration quickly increases in execution time when the state space increases whereas the algorithm proposed in this chapter increases a much slower rate.

Figure 2.9 shows the effects of varying the discount factor on the performance of the algorithm. For this test, a fixed number of reward sources (5) and state space size (50x50) were used, and only the discount factor was varied. The x-axis shows the discount factor and the y-axis shows the length of time required to solve the MDP. For each discount factor, 1000 randomly generated reward configurations with the fixed discount factor were generated. The results show that value iteration increases apparently exponentially with the discount factor, whereas the algorithm proposed in this chapter is invariant to the discount factor. This follows from the exact calculation of the value based off the distance, where the discount factor is simply a constant that is used in the calculation.

All tests were performed on a high-end "gaming class" Alienware laptop with a quad-core Intel i7 running at 4.4 GHz with 32GB RAM without using any GPU hardware acceleration (i.e., CPU only). All code is single threaded, python only and no special optimization libraries other than numpy were used (for example, the python numba library was not used to accelerate numpy calculations.) Both value iteration and the proposed algorithm use numpy. The results presented here are meant to most

fairly present the performance differences between the algorithms, thus further optimizations should yield improved performance beyond what is presented here.

2.4 Conclusion

This chapter presents a novel approach to solving deterministic non-terminating MDPs exactly which we believe is the first example of this technique. This new algorithm's computational speed greatly exceeds that of value iteration for sparse reward sources and, furthermore, is invariant to the discount factor. The complexity of the algorithm is $O(|R|^2 \times |A|^2 \times |S|)$, where $|R|$ is the number of reward sources, $|A|$ is the number of actions, and $|S|$ is the number of states. Memory complexity for the algorithm is $O(|S| + |R| \times |A|)$.

This chapter lays the foundation for future chapters which extend or optimize this algorithm to obtain additional performance benefits.

Next we examine a variant of the algorithm we call "Memoryless" that removes all dependencies on the size of the state space $|S|$.

CHAPTER 3. POSITIVE REWARDS: MEMORYLESS ALGORITHM

3.1 Introduction

In this chapter based on Bertram and Wei (2018b) and Bertram et al. (2019), we propose an extension to `Exact` which we name `Memoryless` that **removes the dependence on the size of the state space** resulting in time complexity of $O(|R|^3 \times |A|^2)$ and memory complexity of $O(|R| \times |A|)$ for the same restricted class of MDPs. Rather than outputting the full value function, `Memoryless` outputs an ordered list in which rewards should be processed using the same techniques as described in Chapter 2. We propose a companion algorithm that can efficiently follow the optimal policy by calculating the value of neighboring states on-demand. We show performance against both value iteration and the `Exact` algorithm for tractable state spaces.

3.2 Methodology

In Chapter 2 the `Exact` algorithm was discussed in detail. The algorithm locates "peaks" in the value function. At each iteration, the `Exact` algorithm selects the most valuable peak and updates an intermediate value function represented by an array in memory. The intermediate value function is the optimal solution of an MDP with the same environment but a subset of the rewards. The iterations continue until all rewards have been considered and results in the optimal value function for the original MDP.

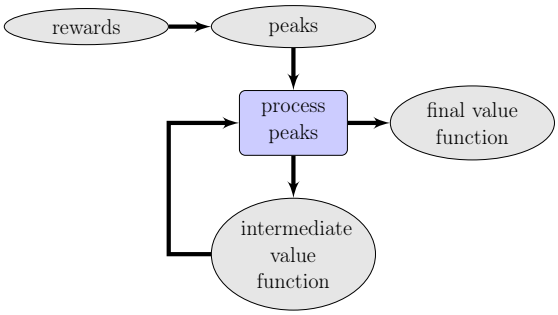

Figure 3.2 `Exact` algorithm

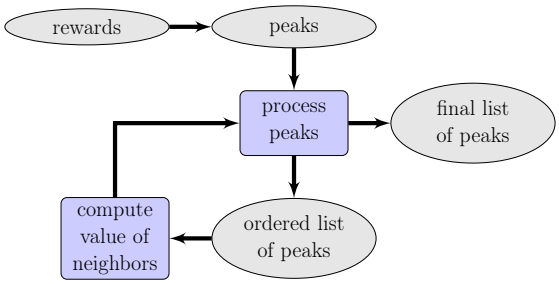

Figure 3.3 `Memoryless` algorithm

Figure 3.4: By maintaining a list of peaks and computing the value of states on demand, the `Memoryless` algorithm eliminates the need for storing the intermediate value function as a table in memory.

In the proof for the `Exact` algorithm in Chapter 2, it was shown that the complete value function can be determined from these peaks. As the algorithm processes each peak, it examines neighboring states, referring to the intermediate value function to look up values of these neighboring states. Note however that the number of neighboring states that are looked up is typically a very small number (on the order of $O(|R| \times |A|)$). In essence, while only the values of a few states are needed, unfortunately the values of all states are computed for each iteration of the algorithm.

Instead, the `Memoryless` algorithm computes the neighboring state values on demand from a list of the peaks sorted by order in which they were processed by the algorithm. A mechanism is proposed to calculate the value of any state from this ordered list. During each iteration of the algorithm, this method calculates the value of any required states on demand and results in a final ordered list of the peaks.

This change to the `Memoryless` algorithm severs its dependence on the size of the state space $|S|$, trading between additional computation time and memory storage. The intermediate value function can be viewed as a lookup table that improves computational efficiency at the expense of additional memory; `Memoryless` essentially sacrifices this lookup table method for a slower computation-based method that requires a pass through the list of peaks, a $O(|R|)$ operation. However, when the number of rewards is small, this trade off can be acceptable, especially considering that the algorithm

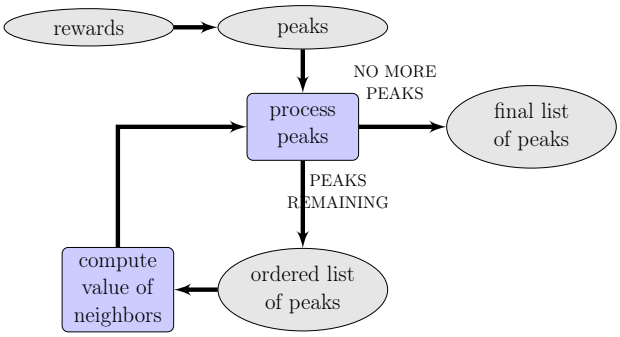

Figure 3.5: `Memoryless` continues processing until no more peaks remain to be processed and arrives at a data structure that can be used to determine the value function of the MDP. In the `Memoryless` algorithm, we maintain a list of the peaks and compute the value of states on demand. The `Exact` algorithm from Chapter 2 therefore has a similar limitation to value iteration in that the entire state space must fit into memory. The `Memoryless` algorithm has no such dependency and can in theory represent even a continuous state space (with infinite states).

is no longer dependent on the size of the state space $|S|$. For problems with very large state spaces, indeed, this trade off makes the problem tractable.

We now discuss the methodology for calculating the distance between two states. For an arbitrary graph, computing the shortest path through the graph is $O(V \log V + E)$ where V is the number of vertices and E is the number of edges. However, in our flight planning problem formulation (i.e., when the dimensions of the state space S map to an underlying metric space) we can use special knowledge of the state space to directly compute distances with a metric such as Euclidean distance to compute shortest paths in constant time. Because of this special circumstance, we permit ourselves to omit the cost of searching a general graph from the run time of the algorithm and note that it increases by a factor of $O(V \log V + E)$ if a general graph search with Dijkstra's algorithm is used. Note that all complexity factors shown below assume this constant time Euclidean distance metric.

`Memoryless` uses a heap-based priority queue which takes $O(\log N)$ for insertion and deletion which we use to keep lists of peaks which are of order $|R|$. Given that we assume a small number for $|R|$, we assume the insertion and deletion times are negligible compared to the run time of the algorithm.

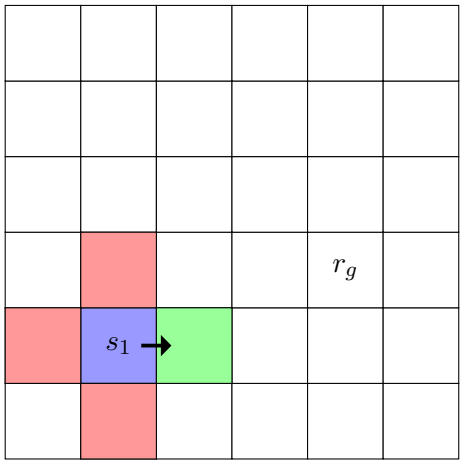

Figure 3.7 Neighbors' values from initial state.

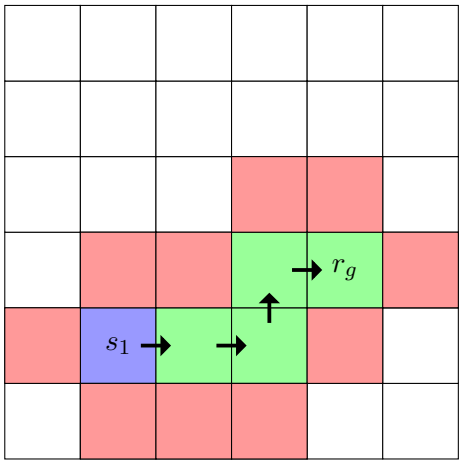

Figure 3.8 Neighbors' values along entire path.

Figure 3.9: Illustration of `Memoryless` algorithm calculating neighboring states on-demand as it follows the optimal policy. The optimal neighbor is shown in green, and the sub-optimal neighbors are shown in red. The initial state is shown in blue and labeled s_1. State containing reward labeled r_g. The optimal policy is shown with arrows. The optimal path is followed by computing the value of only a subset states, where un-colored states are not computed at all. When the number of states $|S|$ is very large, the number of on-demand computations can be very small compared to the total number of states.

Algorithm 2 `Memoryless`

```
1: procedure MEMORYLESS (rewardSources)
2:     processedPeaks ← empty list
3:     sortedPeaks ← PrecomputePeaks( rewardSources )
4:     while sortedPeaks is not empty do
5:         deltaPeaks ← ComputeDeltas( processedPeaks )
6:         sortedPeaks ← PruneInvalidPeaks( processedPeaks)
7:         maxPeak ← max( [ sortedPeaks, deltaPeaks ] )
8:         sortedPeaks ← RemoveAffectedPeaks( maxPeak )
       return processedPeaks
```

Line 2 initializes an empty list to track which peaks have been processed by the algorithm. Line 3 pre-computes baseline peaks and combined peaks based off a list of reward sources and stores them in the form of a sorted list, sorted by value of each peak. Lines 4-8 continue until we have exhausted the potential peaks and each iteration of the loop whittles away at the list of possible peaks. Line 5 computes delta peaks for any remaining reward sources by calculating neighboring states values on-demand. Line 6 removes any peaks that have become invalid due to broken minimum cycles. Line 7 selects the peak with maximum value. Line 8 removes any other potential peaks in the list that are affected by selecting the peak with maximum value. Rather than returning a value function, we instead return the ordered list of peaks that have been processed by the algorithm.

We next examine the ValueOnDemand function, presenting it out of the calling tree order so that we can characterize its computational complexity to understand its impact on the rest of the code:

```
1: procedure VALUEONDEMAND(previousPeaks, desiredState)
2:     maxValue ← MIN_FLOAT
3:     for all previousPeaks do
4:         priValue ← pri_value × γ^φ(desiredState,priState)
5:         secValue ← sec_value × γ^φ(desiredState,secState)
6:         maxValue ← max(maxValue, priValue, secValue)
       return maxValue
```

The function iterates over all previously selected peaks, keeping track of the maximum value that could be derived from any of the previous peaks, which is the value of the state given the rewards that are represented by the selected peaks. This is at worst a $O(|R|)$ operation, which grows from

$O(1)$ to $O(|R|)$ as the rewards are processed. Note here that the data structure alluded to here for a peak contains fields for a primary and secondary state. For baseline and delta peaks only the primary is used, for combined peaks both the primary and secondary field are filled in; this is an artifact of implementation details of how the code represents combined peaks.

```
1: procedure PrecomputePeaks(rewardSources)
2:     list ← empty SortedList
3:     for all rewardSources do
4:         list.add( baseline peak for reward source )
5:     for all rewardSources do
6:         nbr ← find neighboring state with highest reward
7:         if nbr is not empty then
8:             list.add( cycle peak for reward source )
       return list
```

Line 2 initializes a sorted list that is sorted by value of the peaks. In Lines 3-4, a baseline peak is computed for each reward source. In lines 5-8, if any reward sources are next to each other, their combined peaks are computed. Note that at this stage, the new ValueOnDemand function is not called; because no peaks have been selected, the value function at this point is assumed to be zeros everywhere.

PrecomputePeaks() is a $O(|R| \times |A|)$ algorithm that is done one time at the beginning of the algorithm and yields a list with worst case length of $O(|R| \times |A|)$ entries (but only if the reward sources are all adjacent to each other).

```
1: procedure ComputeDeltas(processedPeaks)
2:     list ← empty SortedList
3:     for all reward sources do
4:         currentValue = ValueOnDemand( processedPeaks )
5:         compute delta of reward and currentValue
6:         nbr ← find neighboring state with highest value using ValueOnDemand
7:         list.add(max(deltapeak, neighborvalue))
```

Line 2 initializes a sorted list that is sorted by value of the peaks. Lines 3-7 compute delta peak for any reward sources that remain. Lines 4-6 use the new ValueOnDemand function to compute

the value of the current and neighboring states. Line 6-7 properly sort the delta with respect to neighboring states.

ComputeDeltas(valueFunction) in Chapter 2 was a $O(|R| \times |A|)$ algorithm that is done for each pass of the loop, but with the addition of the $O(|R|)$ ValueOnDemand function, the complexity grows to $O(|R|^2 \times |A|)$.

```
1: procedure PruneInvalidPeaks( processedPeaks)
2:     for all remaining peaks do
3:         nbr ← find neighboring state with highest value using ValueOnDemand
4:         if nbr > peak then
5:             list.remove( peak )
```

Lines 2-5 remove any peaks that have become invalid.

PruneInvalidPeaks() in Chapter 2 was a $O(|R| \times |A|)$ algorithm that is done for each pass of the loop. With the ValueOnDemand function, it now grows to $O(|R|^2 \times |A|)$.

```
1: procedure RemoveAffectedPeaks(list, state)
2:     for all remaining peaks do
3:         if peak is affected by state then
4:             list.remove( peak )
```

Lines 2-4 remove any peaks that have been eliminated by the choice of the peak with maximum value.

RemoveAffectedPeaks operates over the $O(|R| \times |A|)$ *sortedPeaks* list, but this also shrinks by $O(|A|)$ entries each pass.

3.2.0.1 Time Complexity

The main loop of the `Memoryless` function is a $O(|R| \times |A|)$ function, but the ComputeDelta and PruneInvalidPeaks functions are both $O(|R|^2 \times |A|)$ due to their usage of the ValueOnDemand function, bringing the overall algorithm complexity to $O(|R|^3 \times |A|^2)$. Note here there is no dependence upon the size of the state space $|S|$.

For environments where the connected distance is not easily determined (arbitrary transition graph), then the complexity to determine the distance between states must be taken into consideration. However, it is assumed that this can be precomputed offline because T is assumed to be stationary.

For environments like the 2D grid world where the structure of the space is known, determining the connected distance between states is a $O(1)$ calculation, which can be represented as a simple function call to determine the neighbors of each state on-demand.

3.2.0.2 Memory Complexity

Memory complexity for the `Memoryless` algorithm is $O(|R| \times |A|)$

The `Memoryless` function is $O(|R|^3 \times |A|^2)$. Memory complexity for the algorithm is $O(|R| \times |A|)$. Note here there is no dependence upon the size of the state space $|S|$.

3.2.1 Extracting Optimal Trajectory

It is trivial to follow the optimal policy of a solved MDP. Given the current state s, we use the value function to determine which action is most valuable and then take that action. If the optimal policy is followed for each step, it will always follow the optimal trajectory. A powerful observation is that if the value function can be computed in the local neighborhood surrounding the current state, then the optimal policy in that local neighborhood can be determined and the optimal action from the current state can be taken. This means that the full trajectory need not be computed; instead, only the optimal policy from any given state is computed and over time this will result in the optimal trajectory, resulting in an incredible computational savings.

However, if we wish to compute the trajectory, say to visualize the optimal trajectory or to provide the trajectory to a lower level tracking controller, we present a simple algorithm that allows the trajectory to be extracted from the policy. This relies on an *ExecuteAction* module that can simulate one time step forward into the future.

Algorithm 3 FollowLocalPolicy

```
1: procedure FollowLocalPolicy(processedPeaks, initialState)
2:     currState ← initialState
3:     while True do
4:         neighbor ← FindMaxNeighbor( processedPeaks, currState )
5:         action ← DetermineAction( currState, neighbor)
6:         currState ← ExecuteAction( action )
```

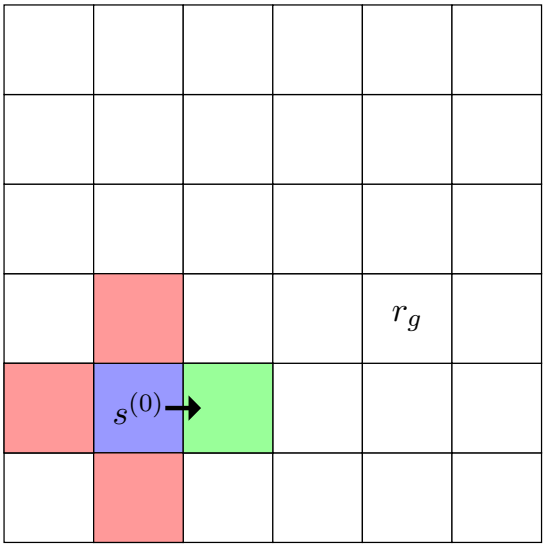

Figure 3.11 Neighbors' values from initial state.

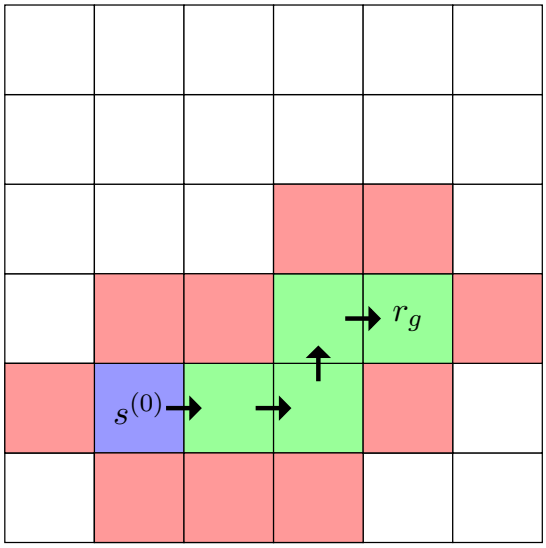

Figure 3.12 Neighbors' values along entire path.

Figure 3.13: Illustration of algorithm calculating neighboring states on-demand as it follows the optimal policy. The optimal neighbor is shown in green, and the sub-optimal neighbors are shown in red. The initial state is shown in blue and labeled $s^{(0)}$. State containing reward labeled r_g. The optimal action is shown with arrows. The optimal path is followed by computing the value of only a subset states, where un-colored states are not computed at all. When the number of states $|S|$ is very large, the number of on-demand computations can be very small compared to the total number of states.

3.3 Experiments

Figure 3.15 Varying number of reward sources

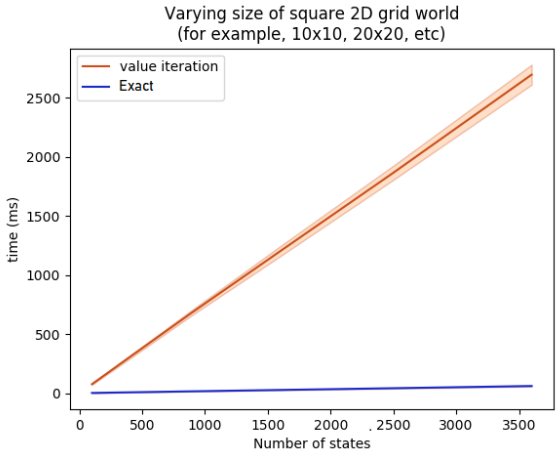

Figure 3.16 Varying number of states

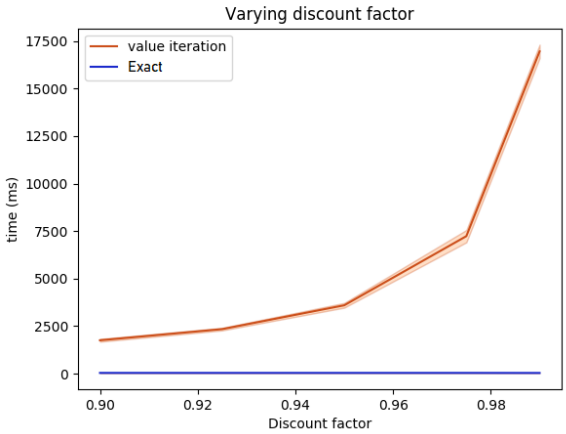

Figure 3.17 Varying discount factor

Figure 3.18: Experimental results showing performance of the proposed algorithm `Memoryless` as compared to value iteration and the `Exact` algorithm. (a) For small numbers of rewards, `Exact` and `Memoryless` are comparable in performance. After a certain point, `Memoryless` begins to perform more slowly than both algorithms but recall that `Memoryless` **has no dependency on the size of the state space** $|S|$. (b) Where `Exact` had a barely visible dependence on the state space size, `Memoryless` is invariant to the number of states. (c) Both `Exact` and `Memoryless` remain invariant to the discount factor.

Figure 3.15 shows the effects of varying the number of reward sources on the performance of the algorithms. For this result, a 50x50 grid world was used. The x-axis shows the number of reward sources used for a test configuration and the y-axis shows the length of time required to solve the MDP. For each test configuration, 10 randomly generated configurations were created for the number of reward sources specified in the test configuration with reward values ranging from 1 to 10. For each generated configuration, value iteration, `Exact` and `Memoryless` were run to obtain performance measurements. As an additional check, the exact solution calculated by this algorithm was compared to the value iteration result to ensure they produced the same result (within a tolerance due to value iteration approximating the exact solution due to the use of a Bellman residual as a terminating condition.) In the plot, the bold line is the average and the colored envelope shows the standard deviation for each test configuration.

The figure shows that as the number of reward sources increases, value iteration remains invariant of the number of reward sources and `Exact` grows slowly. In contrast, we see the tradeoff of increase in time complexity which is traded for not having to hold the value function in memory. For `Memoryless` , for small numbers of reward sources the algorithm clearly continues to outperform value iteration. As the number of reward sources increases, however, an intersection point will occur and value iteration will begin to perform better. However, as the size of the state space increases so to does the execution time of value iteration, so the exact point where the intersection occurs will be problem-specific.

Figure 3.16 shows the effects of varying the size of the state space on the performance of the algorithms. For this a fixed number of reward sources (5) were used, and only the size of the state space was varied (by making the grid world larger). The x axis shows the number of states in the grid world (e.g., $10 \times 10 = 100, 50 \times 50 = 2500$) and the y axis shows length of time required to solve the MDP. For each grid world size, 10 randomly generated reward configurations with the fixed number of reward sources were generated. The results show that value iteration quickly increases in execution time, `Exact` grows very slowly, and `Memoryless` is invariant of the state space size.

Figure 3.17 shows the effects of varying the discount factor on the performance of the algorithms. For this test, a fixed number of reward sources (5) and state space size (50x50) were used, and only the discount factor was varied. The x axis shows the discount factor and the y axis shows the length of time required to solve the MDP. For each discount factor, 10 randomly generated reward configurations with the fixed discount factor were generated. The results show that value iteration increases apparently exponentially with the discount factor, whereas `Exact` and `Memoryless` are both invariant to the discount factor. This follows from the exact calculation of the value based off the distance, where the discount factor is simply a constant that is used in the calculation.

All tests were performed on a high-end gaming class Alienware laptop with a quad-core Intel i7 running at 4.4 GHz with 32GB RAM without using any GPU hardware acceleration (i.e., CPU only). All code is single threaded, python only and no special optimization libraries other than numpy were used (for example, the python numba library was not used to accelerate numpy calculations.) Both value iteration and the proposed algorithm use numpy. The results presented here are meant to most fairly present the performance differences between the algorithms, thus further optimizations should yield improved performance beyond what is presented here.

3.4 Conclusion

This chapters presents a novel extension to the `Exact` algorithm from Chapter 2 named `Memoryless` that eliminates any dependency on the size of the state space. This new algorithm's computational speed greatly exceeds that of value iteration for sparse reward sources and, furthermore, is invariant to both the discount factor and the number of states in the state space. Performance of the `Memoryless` algorithm is $O(|R|^3 \times |A|^2)$, where $|R|$ is the number of reward sources, $|A|$ is the number of actions, and $|S|$ is the number of states. Memory complexity for the algorithm is $O(|R| \times |A|)$. We also propose an algorithm to follow the optimal policy using this technique which at each iteration is $O(|R|)$ that leads to an efficient method to both solve the MDP and follow the optimal policy at run time. Given the quick time to solve the MDP, it also lends itself to allowing the reward source locations to change arbitrarily between time steps. Given the lack of dependence on the size

of the state space, this algorithm provides a way to solve previously intractable MDPs for which the state-action space was too large to solve exactly (that is, without resorting to approximation methods.)

For deterministic environments with sparse rewards such as certain robotics and unmanned vehicle problems, this new method's performance allows computation to be performed with very minimal memory footprint allowing computations to be performed on very low-performing and low-power embedded hardware. If the number of rewards is sufficiently small, the `Memoryless` algorithm could also perform sufficiently well to allow for real-time constraints to be met in an embedded environment such as a robot or unmanned vehicle.

To our knowledge, this is the first time that MDPs can be solved exactly without a full representation of the state space held in memory or relying on iterative convergence to the optimal policy or value function. If this method can be appropriately extended to a larger subset of MDPs (e.g., stochastic MDPs), it could result in broad impacts to the efficiency of solving certain types of MDPs useful in robotics and related spaces.

This chapter and the previous chapter have laid the foundation for an efficient method to solve MDPs efficiently. The next chapter takes a step back and explains intuitively why the algorithm works and how these intuitions can be used to explain the actions of the algorithm.

CHAPTER 4. EXPLAINABILITY AND PRINCIPLE OF OPPORTUNITY

4.1 Introduction

In this chapter based on Bertram and Wei (2018a), we discuss the results of the previous two chapters and show how they result in an explainable interpretation of the actions taken by an agent following the optimal policy.

Specifically, we can determine the following about a given MDP:

1. determine which rewards will and will not be collected given an initial state,
2. whether a given reward will be collected only once or continuously,
3. which local maximum within the value function the initial state will ultimately lead to.

We also show how to create a map of the state space to identify regions that are dominated by one reward source and can fully analyze the state space to explain all actions. We provide a mathematical framework to underpin the claims in this chapter.

Researchers in many fields have long sought interpretable models that humans can understand. For example, in Shortliffe (2012) describes expert systems that provide explanations on medical diagnoses. Examples of other use of the term explainability are Van Lent et al. (2004) and Gunning (2017). Explainability is a property of an algorithm in which the resulting actions of the algorithm can be explained in a way that humans can understand it. This is a qualitative definition that is difficult to quantify, and it exists along a spectrum. For example, a set of if statements (or a decision tree) is very explainable to a human: every action can be traced back to a specific clause of the set of if statements (or branches of the decision tree.) As a counter example, a deep neural net is not very explainable: it may be very difficult to understand how a neural net classifies one image as a cat and another image as a dog. Markov Decision Processes optimal policies have historically been explained as maximizing the expected future reward given the current state. In this chapter, we

show how we can now extend our ability to explain the actions of the optimal policy in terms of a dominant reward among the rewards that can potentially be collected from the current state.

To the author's knowledge this is the first work that is able to trace the policy directly to the rewards in this fashion.

4.2 Methodology

In this chapter, we use this list of peaks and extend the mathematical analysis in Chapter 2 to show that baseline peaks are sufficient to determine how the rewards will be collected. As in Chapter 2, we restrict our analysis to positive real rewards.

4.2.1 Dominance

First we show the following, which intuitively is the natural result of viewing the optimal policy as a hill climb through the value function:

Theorem 13. *For a fully connected MDP, the optimal policy always leads to a local maximum from every initial state.*

Proof. From Chapter 2, it was shown that for a given policy all rewards are collected either once or infinitely, and that if a reward is collected infinitely, it is part of a minimum cycle that is a local maximum of the value function.

If we consider an initial state s_i, a set of rewards $R = \{r_1, \cdots, r_N\}$, and the optimal policy π^*, we define the path taken through the state space by following the optimal policy as $\mathcal{K} = \{s^{(1)}, s^{(2)}, \cdots, s^{(k)}\}$, where k represents the k-th step through the state space. Note that for a non-terminating MDP, this path continues forever. Let us denote the portion of this infinite path which leads to its maximum value as $\mathcal{K}^+ \subset \mathcal{K} | V(\mathcal{K}(i)) < V(\mathcal{K}(j)), \forall i \in \{1, \cdots, k\}, \forall j \in \{i+1, \cdots, k\}$, where we label the maximum i that satisfies the condition as k_{max}. At each step $i \in \{1, \cdots, k_{max}\}$ of this path we may either collect a reward $r_n \in R$ or no reward. If $i < k_{max}$, then we know by the definition of k_{max} that $V(\mathcal{K}^+(i)) < V(\mathcal{K}^+(i+1))$ and that reward r_n is collected only once (i.e., it is a delta reward). If $i = k_{max}$, then we know that $V(\mathcal{K}(i+1)) \leq V(\mathcal{K}(i))$ and we have reached a

local maximum in the value function where a minimum cycle must then form. We denote the state at which the local maximum occurs as as $s_{\mathcal{K}}$ and the value at this state as $V_{\mathcal{K}}$.

Thus following the optimal policy must necessarily result in a path that leads ultimately to a local maximum in the value function. □

This proof also shows why delta peaks can never be local maximums, and that only baseline peaks and combined peaks can be local maximums. Conversely, any baseline peak or combined peak that is selected by `Exact` or `Memoryless` is also a local maximum.

We now define the concept of a *dominant peak* which determines the local maximum that the optimal policy will guide an agent to from a given initial state.

Definition 12. *From an initial state s_i, the* dominant peak *is the peak located at $s_{\mathcal{K}}$ where the agent reaches the local maximum $V_{\mathcal{K}}$ along the optimal path $\mathcal{K}^+$ by following the optimal policy π^*.*

In the case of the two or more peaks that all have equal value at a state s_i, they are said to be *co-dominant*. The optimal policy at these points depends on how the policy extraction algorithm handles the case where multiple actions all lead to states with the same value. Some implementations may deterministically choose, say, the lowest numbered action among an ordered set of actions, while others may select an action randomly among multiple such actions. Without loss of generality, note that we only discuss a deterministic implementation in this chapter.

By this definition, we know that we need not consider any delta peaks, as they are by definition collected once and cannot form a local maximum. Given that we know the set of rewards R, and can determine the baseline peaks $\mathcal{B}$ and combined peaks Γ, how do we determine which of these candidates are the dominant peak at a given state s_i?

Recall the notation from Chapter 2 of the propagation operator $\mathcal{P}$ which calculates the value function from a given peak. The formal notation for a baseline peak's value function is

$$\mathcal{P}_{\mathcal{B}^b}(s) = \gamma^{\delta(s,s_b)} \times \frac{r_b}{1-\gamma^{\phi(s_b)}}, \tag{4.1}$$

where s_b is the state at which reward r_b is collected, $\delta(s, s_b)$ is the distance from state s to state s_b, and $\phi(s_b)$ is the minimum cycle distance for the MDP.

The formal notation for a combined peak's value function is defined as:

$$\mathcal{P}_{\Gamma^{p,s}}(s) = \mathcal{P}_{\mathcal{B}^p}(s) + \mathcal{P}_{\mathcal{B}^s}(s) \tag{4.2}$$

where s_p is the state at which primary reward r_p is collected and s_s is the state at which secondary reward r_s is collected.

To evaluate the discounted future reward of a peak from a state s_i, we simply evaluate these value function definitions at s_i.

From Chapter 2, we know that the value function formed by any subset $M \in \mathcal{M}$ of peaks lies within $V^{\mathcal{M}}$ and that the value function $V^M \in V^{\mathcal{M}}$ formed by the peaks is determined by:

$$V^M(s) = \max_{M_i \in M} M_i(s), \forall s \in S \tag{4.3}$$

Thus, at a given state $s_i \in S$, the value at the state is the maximum value of all the value functions within M evaluated at state s_i. Let us denote the set of peaks that form the value functions in M as P. Let us denote the peak with the maximum value at s_i as P_{max} and its value function as M_{max}, and let us define the subset of peaks which does not contain P_{max} as $P_{sub} = P \setminus P_{max}$ and the corresponding subset of value functions as $M_{sub} = M \setminus M_{max}$.

Now let us consider any subset of the peaks P that still contains the peak P_{max}, $P_{equiv} = \{P_{max}, P'_{sub}\}$ where $P'_{sub} \subset P_{sub}$ and the corresponding value functions $M_{equiv} = \{M_{max}, M'_{sub}\}$ where $M'_{sub} \subset M_{sub}$. We note that the value of M_{equiv} evaluated at s remains the same as M_{max} evaluated at s. In fact, from the perspective of the agent at state s, the value function would remain the same even if the peaks with value functions in P_{sub} were not present. Thus, we say that P_{max} *dominates* the peaks in P_{sub} at s_i or that P_{max} is the *dominant peak* at s_i.

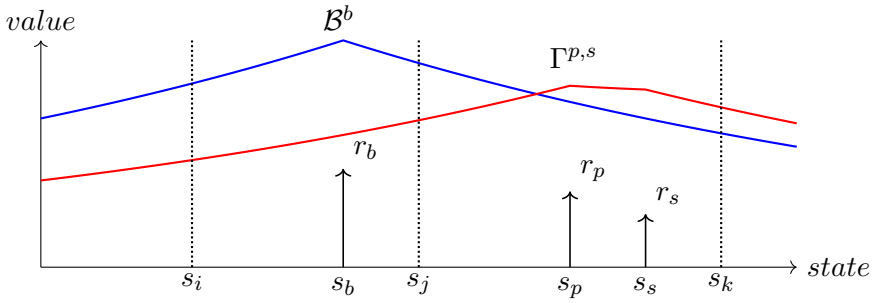

Figure 4.1: Illustration of dominant peak. At state s_i and s_j, the peak $\mathcal{B}^b$ dominates $\Gamma^{p,s}$. At state s_k, the peak $\Gamma^{p,s}$ dominates $\mathcal{B}^b$.

Theorem 14. *If a peak $P_{max} \in P$ located at state s_p is dominant at s_i, then the agent will reach state s_p where the local maximum is formed by the dominant peak.*

Proof. Recall our definition of the optimal path $\mathcal{K}^+$ which describes the hill climb that is performed by following the optimal policy from state s_i.

Let us denote the dominant peak at s_i as P^i_{max} and the corresponding value at s_i as V^i_{max}. Let us denote the value at s_i of any other peak $p_{sub} \in P_{sub}$ as V^i_{sub}.

Let us consider what happens as we follow $\mathcal{K}^+$ when we take one step from s_i to a next state s_j and decrease the distance to P_{max} by 1, we can say for certain that that the value of our peak V_{max} at s_j will increase compared to the value at s_i due to the geometric progression of the discount factor:

$$\begin{aligned} V_{max}(s_i) &= \gamma \times V_{max}(s_j) \\ V_{max}(s_j) &= \frac{1}{\gamma} \times V_{max}(s_i) \end{aligned} \tag{4.4}$$

When we consider the change in the value of the other peak p at s_j, we have three cases to consider. The step from s_i to s_j may:

1. cause the distance to p to increase by 1. In this case, $V^i_{sub} > V^j_{sub}$, and since $V^j_{max} > V^i_{max}$ and $V^i_{max} > V^i_{sub}$, then $V^j_{max} > V^j_{sub}$. Therefore our dominant peak remains dominant at s_j.

2. cause the distance to p to stay the same. In this case, $V^i_{sub} = V^j_{sub}$, and since $V^j_{max} > V^i_{max}$ and $V^i_{max} > V^i_{sub}$, then $V^j_{max} > V^j_{sub}$. Therefore our dominant peak remains dominant at s_j.

3. cause the distance to p to decrease by 1. In this case, $V^j_{sub} > V^i_{sub}$, and in fact $V^j_{sub} = \frac{1}{\gamma} \times V^i_{sub}$. We know that $V^i_{max} > V^i_{sub}$, so therefore:

$$\begin{aligned} \frac{1}{\gamma} \times V^i_{max} &> \frac{1}{\gamma} \times V^i_{sub} \\ V^j_{max} &> V^j_{sub} \end{aligned} \tag{4.5}$$

   Therefore, our dominant peak remains dominant at s_j.

By induction, this continues until we reach the end of $\mathcal{K}^+$, which we defined as the maximum of $\mathcal{K}$, where the local maximum lies and the minimum cycle occurs.

Therefore, we have proven that if a peak P_{max} is dominant at initial state s_i, $\mathcal{K}^+$ will terminate at the local maximum formed by peak P_{max}. □

From this result, we have shown that from a given state s_i, we can determine the resulting local maximum we will be drawn to during the hill climb when following the optimal policy.

If desired, we can therefore iterate over every state in the state space and determine the dominant peak, and from this information construct a map of the state space that shows the regions of the state space that are attracted to each peak. We will describe this region as the *region of dominance* for the corresponding dominant peak, and a state is said to lie within a *dominated region* of a peak.

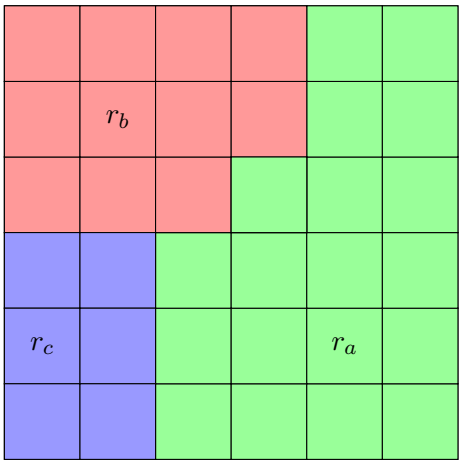

Figure 4.2: Illustration of a map showing the the dominant peak for each state in the state space. The red region shows the region of dominance for r_b, the blue region shows the region of dominance for r_c, and the green region shows the region of dominance for r_a.

4.2.2 Identifying Collected Rewards

Intuitively, we can see that it is only possible to collect rewards that are in the dominated region that the initial state lies within. However, we can do better and determine exactly which rewards will and will not be collected from a given initial state.

Theorem 15. *Given the dominant peak at a state s_i with a value at that state of V_{dom}, any delta peak with a value at s_i of $V_\Delta > V_{dom}$ will be collected. Conversely, any delta peak with a value at s_i of $V_\Delta < V_{dom}$ will not be collected.*

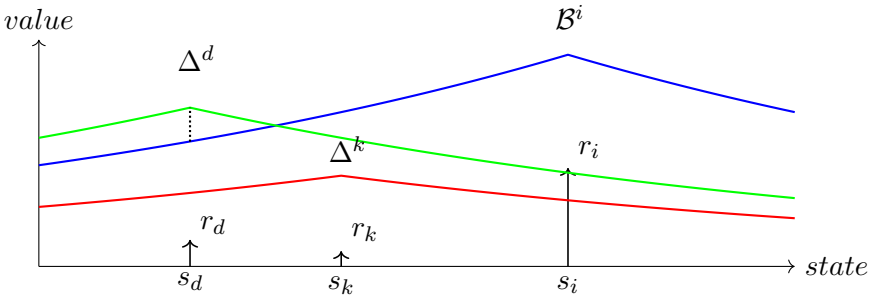

Figure 4.3: Illustration of baseline peak (blue), a delta peak (red) that will not be collected, and a delta peak (green) that will be collected.

Proof. In Chapter 2, it was shown that the optimal value function $V^* \in V^{\mathcal{M}}$ and that V^* is equal to the element-wise maximum of $V^{\mathcal{M}}$. Let us denote as P^* the combination of peaks and the corresponding value functions $M^* \in \mathcal{M}$ that result in the optimal value function V^*. Let us assume that at a state s_i there exists a dominant peak P_{dom} with a value at s_i of V_{dom}, and further assume that a delta peak P_Δ with value at s_i of V_Δ such that $V_\Delta > V_{dom}$, and finally that if there are more delta peaks, the delta peak P_Δ is the one which has maximum value among them at s_i.

Then, it is clear from the definition of V^* that V_{dom} is not the maximum value at s_i and that it is in fact V_Δ. This then implies that the delta peak P_Δ is selected and by definition is collected once along the path to the dominant peak, which may cause a divergence of the optimal path from the path that would result in following the dominant peak directly. This represents a case where the cost of diverting away from the direct path to the dominant peak is overcome by the benefit of obtaining the reward from the delta peak.

Similarly, if $V_\Delta < V_{dom}$, then V_{dom} is the maximum value at s_i and the delta peak will not be collected along the optimal path. This represents a case where the cost of diverting away from the direct path to the dominant peak is not justified by the collection of the reward. □

With this proof, we now have a way to identify which rewards will be collected. Given the list of optimal peaks from the `Memoryless` algorithm and an initial state s_i, the rewards associated with the peaks listed below are collected as follows.

1. the dominant peak (which is either a baseline peak or a combined peak)
2. any delta peaks whose value $V_\Delta > V_{dom}$ at state s_i.

We denote this set of peaks that are collected as P^c and the corresponding set of rewards R^c as the *collected rewards*. No other rewards are collected when following the optimal policy from state s_i.

4.2.3 Relative Contribution

We define the *relative contribution* of a collected peak $p \in P^c$ through the following procedure.

Definition 13. *Given the set of collected peaks P^c with a length of k, we must order them in decreasing order by their value as evaluated at state s_i, which we will define as the list P^{ord} also with length k. The maximum value of this list would then be the first element $P^{ord}(0)$ which is equivalent to $V^*(s_i)$. We then append to this ordered list a trailing value of 0, denoted as $P^{prepared}$ which then has length $k+1$. The difference in value, $\mathcal{D}$, between the peaks is then defined as:*

$$\mathcal{D}_i = P^{prepared}(i) - P^{prepared}(i+1), \forall i \in \{1, \cdots, k\}$$

The *relative contribution* of the collected peaks is then the ratio $\frac{\mathcal{D}_i}{V^*(s_i)}$, which could then be expressed as a percentage. This percentage can be used to determine how strongly a given collected reward is influencing the optimal policy at state s_i, which provides a deeper understanding of the action and improves the explainability of the Markov Decision Process optimal policy.

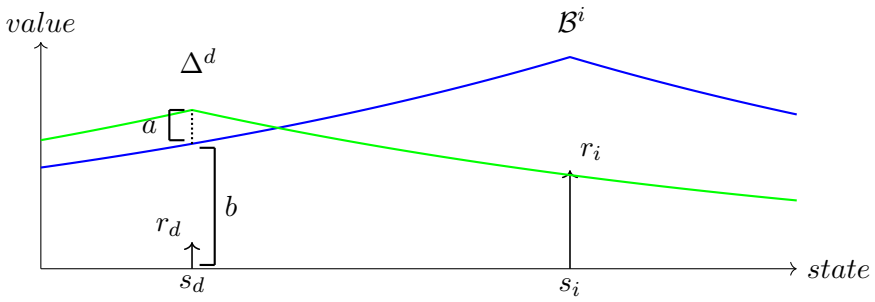

Figure 4.4: Illustration of baseline peak (blue), and a delta peak (green) The value at state s_i is $V(s_i) = a + b$, where b is the contribution from the baseline peak and a is the contribution from the delta peak. The relative contributions are the ratios $\mathcal{D} = \{\frac{a}{V(s_i)}, \frac{b}{V(s_i)}\}$ from which we can express as a percentage how much each reward source is contributing to the value at the state s_d (or any other state.)

4.3 Stability and Sensitivity

We can also use this degree of contribution in a form of stability or sensitivity analysis. If a dominating reward dominates the other rewards by a large margin (e.g., dominating reward contributes 75%, other rewards contribute 25% in total), then we can say that that state and nearby states are likely stable and relatively insensitive to small disturbances in the dominated rewards (e.g., changes in location or magnitude). However, if a dominating reward dominates the other rewards by a small margin (e.g., dominating reward contributes 55%, other rewards contribute 45% in total), then we can say that the state and nearby states are likely unstable and are sensitive to small disturbances in the dominated rewards. Here we mean stability in the sense of which region of dominance a given state or nearby states will fall within before and after small disturbances are introduced to the dominated rewards. Generally, if the state tends to stay within the same region of dominance after the disturbance is introduced, it will have a higher stability margin. Thus, in Figure 4.4, states near the rewards r_a, r_b would be expected to be more stable than states near the boundary between their two regions.

4.4 Principle of Opportunity

In Bellman (1957), Bellman describes dynamic programming (in the optimization sense) and defines the Principle of Optimality, which has since become an important principle in computer science (e.g. Dijkstra's algorithm), Markov Decision Processes (e.g. Value Iteration), and Optimal Control (e.g. Hamilton-Jacobi-Bellman equation).

With our new intuition and proofs regarding the `Exact` algorithm and explainability presented in this chapter, we can also describe the actions of the optimal policy of a Markov Decision Process in a new way.

At each step of the Markov Decision Process, we have a potential to collect many rewards which we can term as an *opportunity*. The optimal action is then the reward which offers the most opportunity, which we can describe as the Principle of Opportunity.

This new perspective helps to link the mathematical framework originally proposed by Bellman with the mathematical framework described in this thesis. This Principle of Opportunity also serves to link these mathematical frameworks to our own intuitions about how humans select actions when presented with multiple competing rewards.

Note that the Principle of Opportunity is compatible with Bellman's Principle of Optimality and is simply an alternative explanation.

4.5 Conclusion

In this chapter, we have presented a novel approach to explaining why the optimal policy for a Markov Decision Process selects a specific action, relating the action to the degree in which they are driven by various reward sources. This reduces the opaqueness of Markov Decision Processes and can be used to analyze the state space to determine which regions of the state space will be attracted to given local maximums of the value function.

This algorithm is based on the research and methods proposed Chapters 2 and 3 and is therefore subject to the same restricted class of Markov Decision Processes. If the methods can be expanded to work on a more general class of MDPs, then the method described in this chapter should also be applicable to this more general class of MDPs.

CHAPTER 5. NEGATIVE REWARDS: FastMDP ALGORITHM

5.1 Introduction

Up to this point, all chapters have discussed algorithms which have focused on positive rewards only. In this chapter based on Bertram et al. (2019) and Bertram and Wei (2019) we examine the effect on the value function of introducing negative rewards into MDPs. We show that negative rewards do not follow the same pattern as positive rewards and describe the result of our investigation.

While we were unable to find a way to compute negative rewards exactly, we offer an alternative way to represent negative rewards approximately as "risk wells" which have useful properties. For problems where these risk wells appropriately model negative rewards, we offer an algorithm based of `Exact` and `Memoryless` which can compute the value function efficiently.

We also show that through reordering of operations, we achieve an algorithm which is $O(|R|)$ where $|R|$ is the number of rewards in the MDP with no dependence on the size of the state space $|S|$. This is an important contribution to the state of the art and provides a way for MDPs to be broadly used, even in cases where the state space is continuous.

5.2 Methodology

5.2.1 Negative Reward

In MDPs in which the state space maps to an underlying metric space, such as a robot navigating a 2D plane environment, negative rewards behave very differently than positive rewards. Negative rewards do not propagate outward with an exponential decay curve like positive rewards do. Instead, they create negative "spikes" in the value function that primarily effect only a single state where the reward occurs as illustrated in Figure 5.1. This happens because in an environment like this the negative reward is easy to avoid by simply stepping around it.

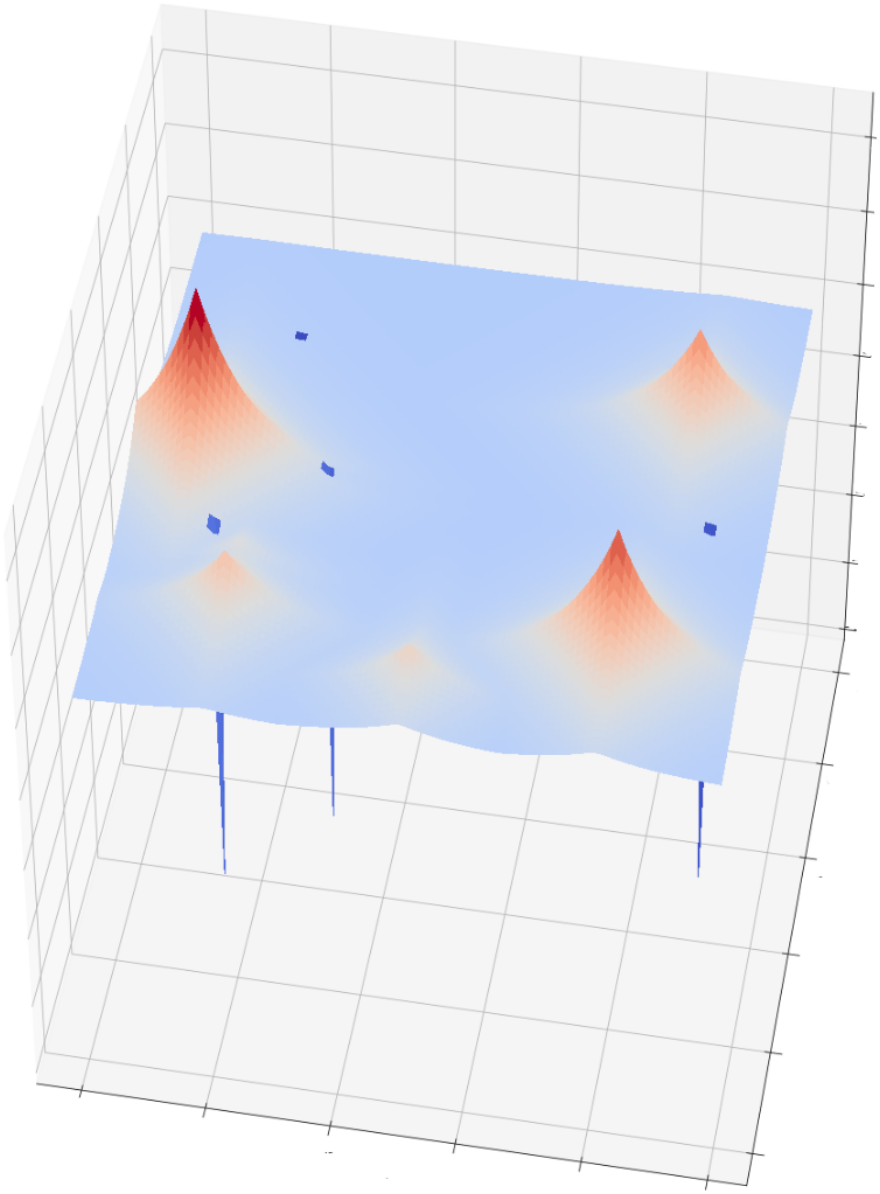

Figure 5.1: Negative rewards placed within an MDP make sharp negative spikes in the value function. Negative rewards to not decay outward like positive rewards do.

While we discuss specific applications in more detail in later chapters, for the problem of aircraft collision avoidance, intuitively we seek a method to model obstacles such as towers, terrain, and other aircraft as risks that should be avoided. Being close to an obstacle should be riskier than being further away from an obstacle, and beyond a certain threshold, an obstacle presents effectively no risk and can be safely ignored. We term this a "risk well" and illustrate a risk well in Figure 5.2.

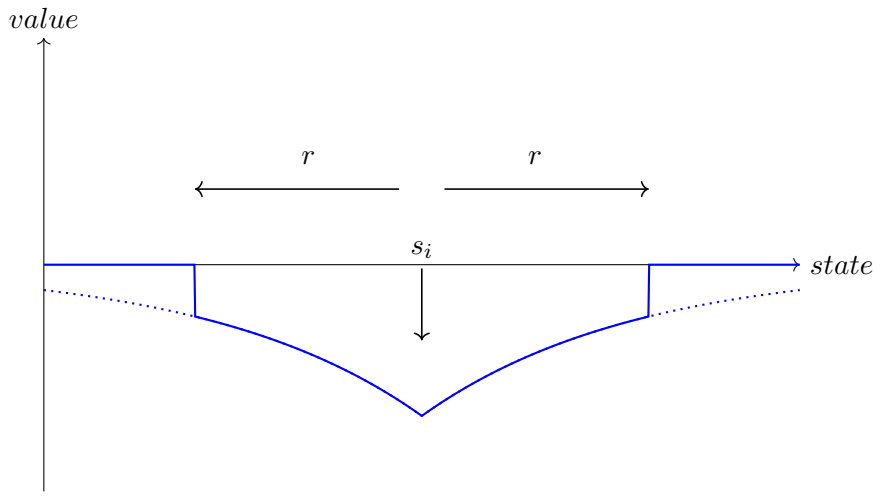

Figure 5.2: Desired form for negative rewards. At state s_i (e.g. location of an obstacle) we have the most negative reward. As we get further from s_i the negative reward decays. Beyond some radius r we assume there is no risk and truncate the exponential decay to a value of 0. We term this a "risk well".

Through our investigation of negative rewards in an MDP formulation, we found that modeling negative rewards as a single point was ineffective as this only materially affected states in the very immediate vicinity of the negative reward. We found that to model our obstacles as a risk that decreased with distance to the obstacle, we had to manually construct a reward function with many negative rewards that explicitly encoded the risk. This explicit construction of the risk may take hundred or thousands of negative rewards in the MDP, which would not lead to good performance.

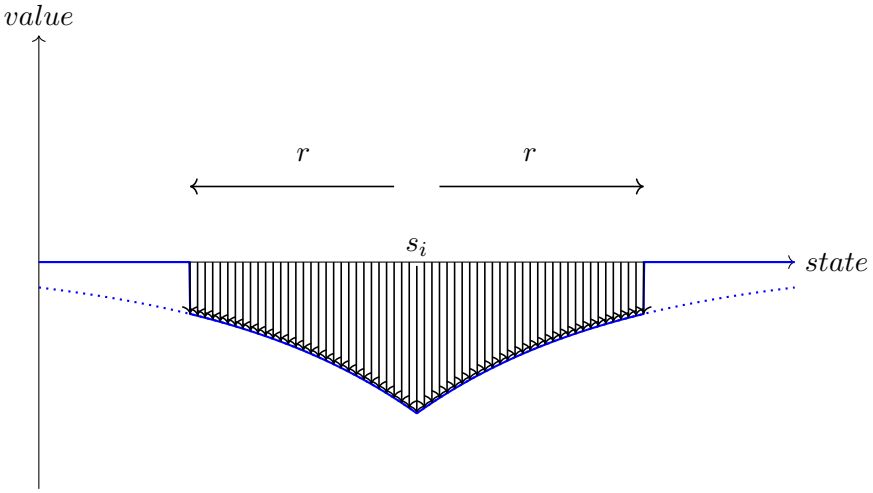

Figure 5.3: A risk well composed of many negative rewards. Solving this with value iteration yields a very close approximation of the shape we desire.

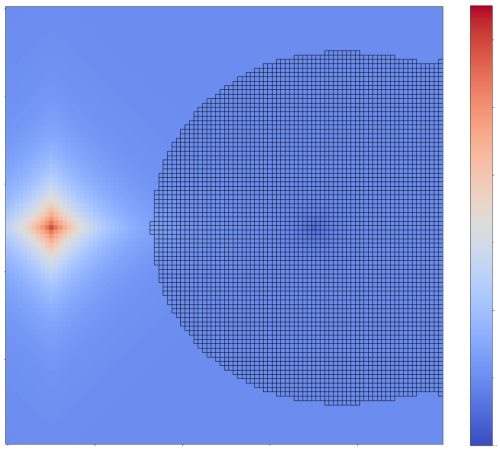

Figure 5.5 Overhead view of risk well composed of multiple negative rewards

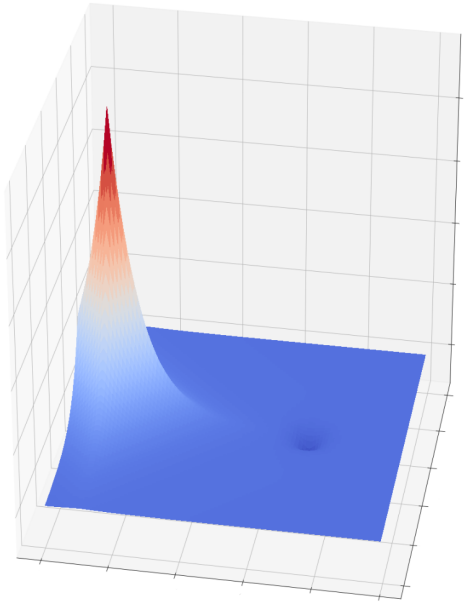

Figure 5.6 3D view of risk well

Figure 5.7: Constructing a risk well manually from hundreds of individual negative rewards of appropriate magnitude.

We instead came up with a method to obtain the desired negative reward shape in a way which allowed us to reuse the efficient `Memoryless` algorithm.

5.2.2 Standard Positive Form

To efficiently compute the shape of a risk well without having to explicitly model it with multiple negative rewards, we can instead temporarily treat the negative reward as if it were a positive reward. We can solve for its shape efficiently with the `Memoryless` algorithm, and then negate the resulting value function to arrive at the desired shape of the risk well, as illustrated in Figure 5.12.

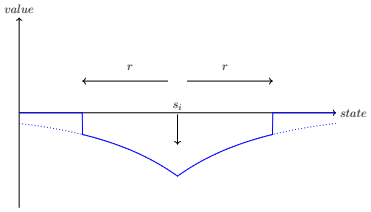

Figure 5.9 Desired shape of a risk well.

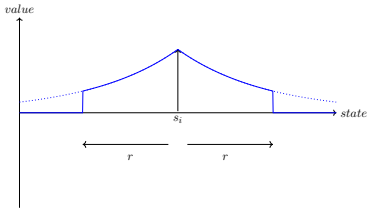

Figure 5.10 Convert to Standard Positive Form and solve with `Memoryless` Truncate result beyond radius r.

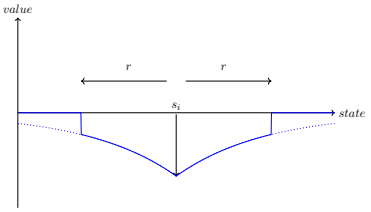

Figure 5.11 Negate value function to efficiently arrive at the desired shape.

Figure 5.12: Using Standard Positive Form to efficiently compute a risk well with a single negative reward without having to use many (hundreds or thousands) of explicit rewards.

This leads to a procedure where we isolate the positive and negative rewards into two subproblems. We solve the positive rewards with `Memoryless` as we normally would.

With the negative rewards isolated into a separate MDP, we temporarily negate all of the negative rewards so that they are all positive which we term *Standard Positive Form*. The `Memoryless` algorithm can then compute the resulting value function for the negative rewards in Standard Positive Form. We can then negate this value function to arrive at an approximation of the value function of the risk wells of the original MDP. When the resulting value functions of the two subproblems are added together, we then arrive back at a close approximation of the original value iteration solution obtained from all of the positive and negative rewards together. See Figure 5.13

for how separating the problem into two subproblems, combining the results of the subproblems, and comparison to value iteration.

5.2.3 Reordering of Operations to Improve Efficiency

As we learned in Chapter 2, the value function that results from an MDP with a set of positive rewards is formed from the corresponding peaks. We solve for each peak individually and use the max operator to recover the resulting value function. When we break our MDP up into separate subproblems for positive and negative rewards in standard positive form, we employ the max operator in both subproblems.

It is important to note that we can further decompose an MDP subproblem with positive rewards into yet smaller MDP subproblems, each with a single reward. This offers an opportunity to solve each subproblem in parallel, and then to combine the results at the end with a max operation. We use this fact to significantly improve performance of the algorithm.

To compute the resulting value at any state, we follow the following procedure to compute the value at a state s:

1. Break up the MDP problem with N rewards into N separate subproblems, each with a single reward, n of which are positive and m of which are negative. Keep the subproblems with positive and negative rewards in separate lists, $\{MDP_1^+, \cdots, MDP_n^+\}$ and $\{MDP_1^-, \cdots, MDP_m^-\}$.

2. Convert any of the m subproblems which contain a negative reward into Standard Positive Form.

3. Solve all N MDP subproblems using the `Memoryless` algorithm. (As each subproblem contains a single reward, the solution is trivial and can be considered constant time as the number of rewards is known.)

4. Extract the value from the n positive reward MDP subproblems:

$$V^+(s) = \max_{i=1}^{n} V^*_{MDP_i^+}(s) \tag{5.1}$$

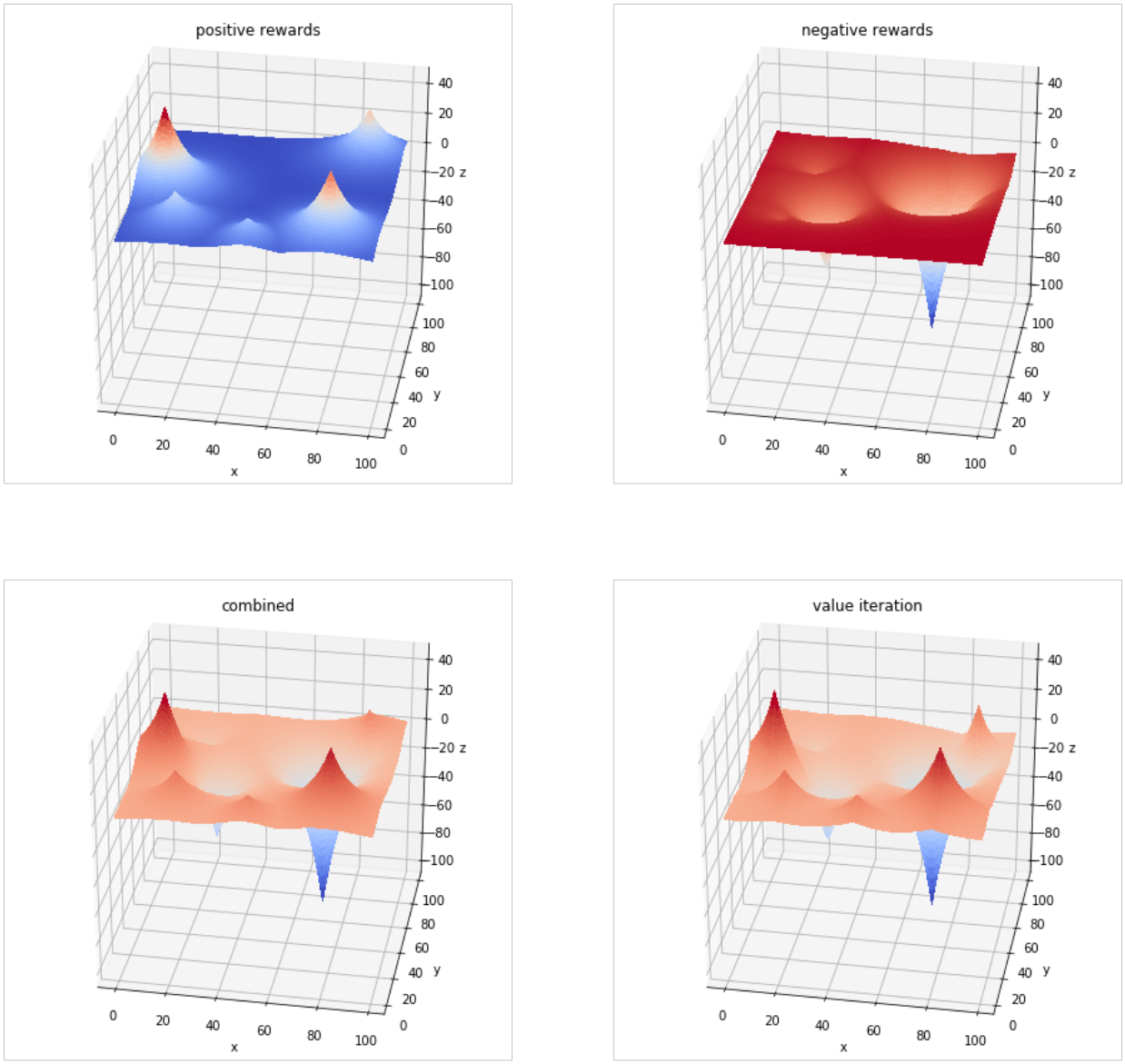

Figure 5.13: Separating MDPs into positive and negative rewards into sub problems, reassembling the results, and comparison to value iteration results.

5. Extract the value from the m negative reward MDP subproblems with rewards in standard positive form

$$V^{-}(s) = \max_{i=1}^{n} V^{*}_{MDP_i^{-}}(s) \tag{5.2}$$

6. Sum the results of the two subproblems together, while converting from standard positive form back into a negative value to obtain the true value of the value function.

$$V^{*}(s) = V^{+}(s) - V^{-}(s) \tag{5.3}$$

Decomposing the problem into very small MDPs with a fixed number of rewards makes each of the subproblems constant-time operation yielding an overall linear $O(N)$ where N is the number of rewards.

5.2.4 Algorithm

Note that for clarity the pseudocode is presented in a loop form, but in practice the code can be optimized to perform some operations in parallel with hardware assistance such as Single-Instruction-Mutiple-Data (SIMD) co-processing units.

5.3 Conclusion

This chapter presents a method to incorporate one form of negative rewards which we term *risk wells*. We present an algorithm which can efficiently compute an approximation of the value function composed of positive rewards and negative rewards modeled as risk wells in $O(N)$ time, where N is the total number of rewards (positive or negative) in the MDP.

We also hint at some parallelization in the computation of the value function which we will exploit in future chapters.

Algorithm 4 `FastMDP`

procedure FastMDP(*ownshipState*, *worldState*)
  // Build a list of positive rewards which represent goals
  posPeaks ←*build pos rewards*
  // Build a list of negative rewards (risk wells) which represent penalties
  negPeaks ←*build neg rewards in Standard Positive Form*
  // Determine neighboring reachable states
  $reachStates \leftarrow neighbors(currState, actions)$
  // Compute the value at each reachable state
  $trueVals \leftarrow$ space for each state
  for $state \in reachStates$ **do**
    // First for positive peaks
    for $p_i = posPeak_i \in posPeaks$ **do**
      $d_p \leftarrow \|state - \textbf{location}(p_i)\|_2$ ▷ distance
      $r_p \leftarrow \textbf{reward}(p_i)$
      $\gamma_p \leftarrow \textbf{discount}(p_i)$
      $posValues_i \leftarrow |r_p| \cdot \gamma_p^{d_p}$
    $posMax \leftarrow \max_i posValues_i$
    // Next for negative peaks (in Standard Positive Form)
    for $n_i = negPeak_i \in negPeaks$ **do**
      $d_n \leftarrow \|state - \textbf{location}(n_i)\|_2$ ▷ distance
      $\rho_n \leftarrow negDist_i < \textbf{radius}(n_i)$ ▷ within radius
      $r_n \leftarrow \textbf{reward}(n_i)$
      $\gamma_n \leftarrow \textbf{discount}(n_i)$
      $negValues_i \leftarrow int(\rho_n) \cdot |r_n| \cdot \gamma_n^{d_n}$
      $negMax \leftarrow \max_i negValues$
    $trueVals[state] \leftarrow posMax - negMax$
  // Identify the most valuable action
  $bestActionIdx \leftarrow \arg\max(trueVals)$
  // For illustration, the corresponding value
  $maxValue \leftarrow trueVals[bestActionIdx]$

CHAPTER 6. APPLICATION: COLLISION AVOIDANCE

6.1 Introduction

In this chapter based on Bertram et al. (2019), we describe our first application of the `Exact` and `Memoryless` algorithms by showing how they can be used to allow a UAV to navigate to a goal while avoiding collisions with intruder UAVs.

NASA, Uber and Airbus have been exploring the concept of Urban Air Mobility (UAM) Gipson (2017); ube (2017); Holden and Goel (2016); air (2018, 2017) where vertical takeoff and landing (VTOL) aircraft may be either human piloted or autonomous for passenger transport in personal commuting or on-demand air taxiing. UAM operations are expected to fundamentally change cities and people's lives by reducing commute times and stress. Development of efficient algorithms for vehicle technology and airspace operation will be critical for the success of UAM. A critical question is whether structured air space management is required or whether a more loosely controlled "Free Flight" model is possible. Due to the computational complexity of free flight, most research is focusing on a structured approach. In this chapter we propose a online computational guidance algorithm which could be used on board an aircraft with limited computing power to support a Free Flight paradigm, or to provide backup planning capability on board the aircraft to enable safe and efficient flight operations in on-demand urban air transportation.

The concept of "Free Flight" was proposed primarily for future air transportation applications because it has the potential to cope with the ongoing congestion of the current ATC system. It was shown in previous work Hoekstra et al. (2002); Bilimoria et al. (2003) that free flight with airborne separation is able to handle a higher traffic density. Besides, free flight can also bring fuel and time efficiency Clari et al. (2001). In a free flight framework, it is implied that aircraft will be responsible for their own separation assurance and conflict resolution. The loss of an airway structure may make the process of detecting and resolving conflicts between aircraft more complex.

However, previous study Tomlin et al. (1998) shows that free flight is potentially feasible because of enabling technologies such as Global Positioning Systems (GPS), data link communications like Automatic Dependence Surveillance-Broadcast (ADSB) Kahne and Frolow (1996), Traffic Alert and Collision Avoidance Systems (TCAS) Harman (1989), and powerful on board computation. Also, automated conflict detection and resolution tools Krozel and Peters (1997) will be required to aid pilots and/or ground controllers in ensuring traffic separation and conflict resolution.

In this chapter, a computational guidance algorithm with collision avoidance capability is proposed using Markov Decision Processes (MDPs), where the input of this algorithm is the position of other obstacles such as aircraft, and the position of one or more destinations. Through on board sensed information of other obstacles or aircraft, the algorithm will perform online sequential decision making to select actions in real-time with on board avionics. The series of actions will generate a trajectory which can guide the aircraft to quickly reach its goal and avoid potential conflicts. The algorithm operates efficiently and can fully recompute its guidance to support online replanning in the presence of dynamically changing obstacles. The proposed algorithm provides a potential solution framework to enable autonomous on-demand free flight operations in urban air mobility.

6.2 Collision Avoidance Related Work

There have been many important contributions to the topic of guidance algorithms with collision avoidance capability for small unmanned aerial aircraft which can be roughly categorized based on the following criteria:

- Centralized/Decentralized Schouwenaars et al. (2004): whether the problem is solved by a central supervising controller (centralized) or by each aircraft individually (decentralized).
- Planning/Reacting Siegwart et al. (2011): The planned approach generates feasible paths ahead of time; whereas the reactive approach typically uses an online collision avoidance system to respond to dangerous situations.

- Cooperative/Non-cooperative: whether there exists online communication between aircraft or between aircraft and the central controller.

In centralized methods, the conflicts between aircraft are resolved by a central supervising controller. Under such scenario, the state of each aircraft, the obstacle information, the trajectory constraint as well as the terminal condition are known to the central controller (thus centralized methods are always cooperative), and the central controller in return designs the individual whole trajectory for all aircraft before the flight, typically by formulating it to an optimal control problem. These methods can be based on semidefinite programming Frazzoli et al. (2001), nonlinear programming Raghunathan et al. (2004); Enright and Conway (1992), mixed integer linear programming Schouwenaars et al. (2001); Richards and How (2002); Pallottino et al. (2002); Vela et al. (2009), mixed integer quadratic programming Mellinger et al. (2012), sequential convex programming Augugliaro et al. (2012); Morgan et al. (2014), second-order cone programming Acikmese and Ploen (2007), evolutionary techniques Delahaye et al. (2010); Cobano et al. (2011), and particle swarm optimization Pontani and Conway (2010). Besides formulating this problem using optimal control framework, roadmap methods such as visibility graph Hoffmann et al. (2004) and Voronoi diagrams Howlet et al. (2004) can also handle the path planning problem for aircraft. However, calculating the exact solutions will become impractical when the state space becomes large or high-dimensional. To address this issue, sample-based planning algorithms are proposed, such as probabilistic roadmaps Kavraki et al. (1994), RRT LaValle (1998), and RRT* Karaman and Frazzoli (2011). These centralized methods often pursue the global optimality of the solution. However, as the number of aircraft grows, the computation time of these methods typically scales exponentially. Moreover, these centralized planning approaches typically need to be re-run, as new information in the environment is updated (e.g. a new aircraft enters the airspace).

On the other hand, decentralized methods scale better with respect to the number of agents and are more robust since they do not possess a single point of failure Pallottino et al. (2006). In decentralized methods, conflicts are resolved by each aircraft individually. Decentralized methods can be cooperative and non-cooperative. Researchers have proposed several algorithms under the case

where the communication between aircraft can be successfully established (cooperative) Wollkind et al. (2004). Algorithms in Purwin et al. (2008); Desaraju and How (2011) are based on message-passing schemes, which resolve local (e.g. pairwise) conflicts without needing to form a joint optimization problem between all members of the team. In Schouwenaars et al. (2004), every agent is allotted a time slot in which to compute a dynamically feasible and guaranteed collision-free path using MILP. In Inalhan et al. (2002), the author recast the global optimization problem as several local problems, which are then iteratively solved by the agents in a decentralized way. In Decentralized Model Predictive Control approach Richards and How (2004), the aircraft solve their own sub-problem one after the other and send the action to other subsystems through communication.

Model Predictive Control Shim and Sastry (2007); Shim et al. (2003) can be used to solve collision avoidance problem but the computation load is relatively high. Potential field method Sigurd and How (2003); Langelaan and Rock (2005) is computationally fast, but in general they provide no guarantees of collision avoidance. Machine learning and reinforcement learning based algorithms Kahn et al. (2017); Zhang et al. (2016); Ong and Kochenderfer (2016); Chen et al. (2017) have promising performance, but usually need a lot of time to train. Monte Carlo Tree Search algorithm Yang and Wei (2018) does not need time to train before the flight and it can finish in any predefined computation time, but the aircraft can only adopt several discretized actions at each time step. A geometric approach Han et al. (2009); Park et al. (2008); Krozel et al. (2000); Van Den Berg et al. (2011) can be also applied for the collision avoidance problem and the computation time only grows linearly as the number of aircraft increases. DAIDALUS (Detect and Avoid Alerting Logic for Unmanned Systems) Muñoz et al. (2015) is another geometric approach developed by NASA. The core logic of DAIDALUS consists of: (1) definition of self-separation threshold (SST) and well-clear violation volume (WCV), (2) algorithms for determining if there exists potential conflict between aircraft pairs within a given lookahead time, and (3) maneuver guidance and alerting logic. The drawback of these geometric approaches is that it can not look ahead for more than one step (it only pays attention to the current action and does not take account of the effect of subsequent actions) and the outcome can be local optimal in the view of the global trajectory.

Our proposed algorithm is an alternative to previous work Yang and Wei (2018), where instead of using a Monte Carlo Tree Search algorithm, we propose a novel online method for solving a subclass of MDPs with very efficient performance for problems with sparse rewards.

See also Section 1.3 for information on literature related to MDPs.

6.3 Methodology

We will formulate the problem as a Markov Decision Process problem and solve it with the `Memoryless` algorithm. For the UAV problem in this chapter, we will assume that our UAVs operate effectively in a 2D plane which will maximize potential conflicts and require all corrections to be performed laterally.

6.3.1 State Space

We define the environment in which the UAV operates as a $24km \times 24km$ square area in which there is a goal and a configurable number of intruders. We discretize the MDP state space into an 800x800 grid of states.

The state includes all the information the ownship needs for its decision making: the position, heading, and velocity of the ownship, the goal position, and each intruder(s) position, heading and velocity. The ownship position (o_x, o_y), heading o_θ and velocity o_{v_x}, o_{v_y}, the goal position (g_x, g_y), and for each intruder $\forall k \in K$, the position $(i_{k,x}, i_{k,y})$, heading $i_{k,\theta}$, and velocity i_{k,v_x}, i_{k,v_y} are all concatenated into one long vector.

$$s = [o_x, o_y, o_\theta, o_{v_x}, o_{v_y}, g_x, g_y, i_{1,x}, i_{1,y}, i_{1,\theta}, i_{1,v_x}, i_{1,v_y}, \cdots, i_{m,y}, i_{m,\theta}, i_{m,v_x}, i_{m,v_y}], \tag{6.1}$$

where m represents the number of intruders.

6.3.2 Action Space

The set of possible actions that can be taken are heading commands from $0, \cdots, 2\pi$ in steps of $\frac{\pi}{12}$.

$$A = \{0, \frac{\pi}{12}, \frac{2\pi}{12}, \frac{3\pi}{12}, \cdots, \frac{23\pi}{12}\}. \tag{6.2}$$

6.3.3 Dynamic Model

The ownship kinematic model is:

$$\dot{x} = v\cos\theta \tag{6.3}$$

$$\dot{y} = v\sin\theta, \tag{6.4}$$

where $v = \sqrt{v_x^2 + v_y^2}$ is the speed of the aircraft.

The ownship speed v is fixed at $50m/s$. At each step the ownship is restricted to performing a change in heading of $\dot{\theta}$ of up to $\pm 15°$.

6.3.4 Reward Function

We model the goal as a positive reward of 100. To model the UAV risk, we define a "risk well" as a negative reward of -500 which decays at a rate of 0.96 for up to 1500 meters from the center of the well, after which there is no negative reward.

For each intruder, we place a risk well at the location the intruder will be in 2 seconds and a second risk well at the location the intruder will be in 4 seconds. This assumes that the intruders will maintain a constant heading and velocity and is used as a way to model the risk over the next 4 seconds.

For the overall MDP containing all rewards, we use a discount factor of 0.999. This provides a strong attraction to the goal globally over the state space. We found in early experiments that in

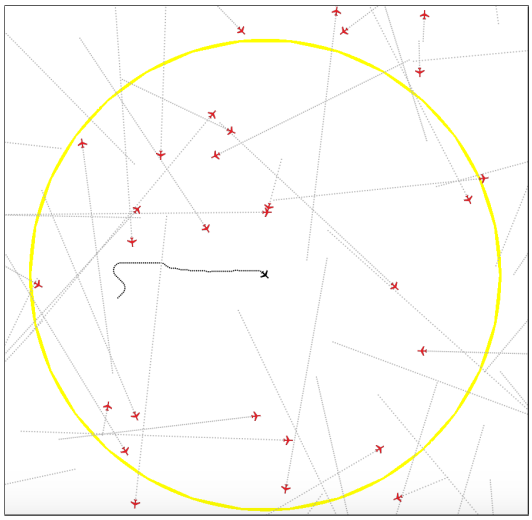

Figure 6.2 Deterministic intruders

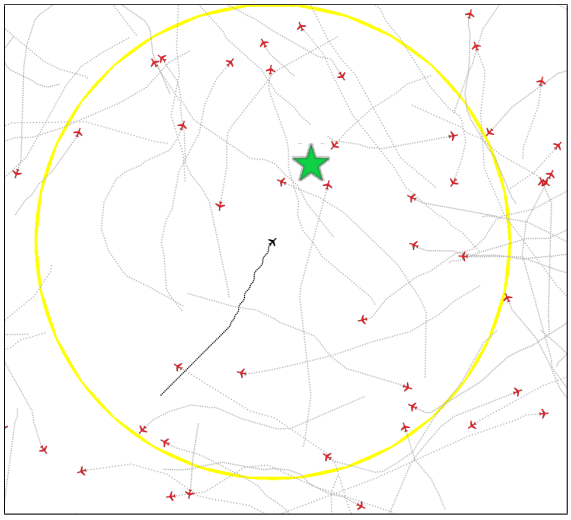

Figure 6.3 Stochastic intruders

Figure 6.4: Experimental results showing deterministic and stochastic intruders. Deterministic intruders are spawned in random locations with random heading and velocities (within predefined limits), but during flight they maintain constant heading and airspeed. Stochastic intruders are spawned identically, but there is a small probability that they will change their heading by up to $\pm 25^\circ$ at each time step making it very difficult to predict their future position with any certainty. Ownship is in black, intruders are in red, and goal is a green star. Light shaded paths are intruder past trajectories, and the dark shaded path is ownship past trajectory. The yellow circle illustrates the boundary beyond which intruders will be ignored.

an MDP the impact of negative rewards remains relatively isolated to the state where the negative reward occurs. Thus the negative rewards we place in the space are largely unaffected by the discount factor.

With a normal MDP formulation, we would need to insert many hundreds or thousands of individual negative rewards to model the risk wells for each intruder. With the algorithm we present in this chapter, we instead construct an MDP in standard positive form and represent the risk wells as a single negative reward of 500 with a discount factor of 0.96. Each intruder receives its own MDP, and as there are two risk wells per intruder, there are two rewards of 500 in each intruder's MDP.

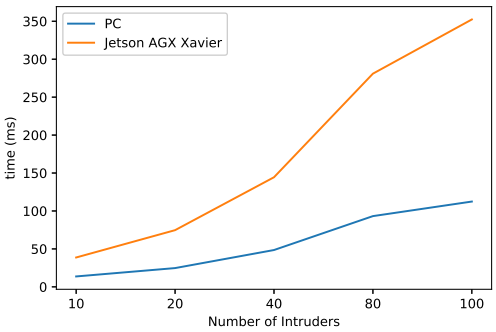

Figure 6.6 Timing performance as number of intruders increases

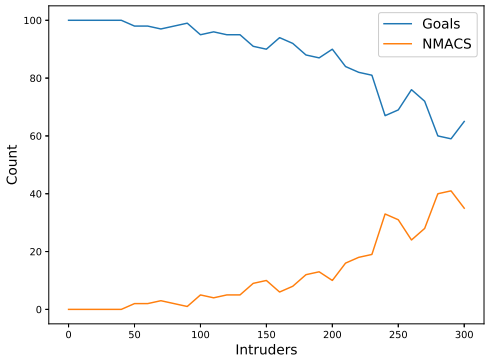

Figure 6.7 Collision avoidance performance as intruder density increases

Figure 6.8: Experimental results showing the performance of the algorithm. (a) shows time to compute the solution as the number of intruders increases is roughly $O(m)$ where m is then number of intruders. (b) shows the ability to reach the goal and the number of near midair collisions (NMACs) as the number of randomly turning intruders in the space increases. Note that as the airspace becomes more crowded, at some point it becomes nearly impossible to make it through the waves of intruders. Also, there may be situations where the random position of the intruders leaves no feasible path for collision avoidance.

6.4 Results

We define a radius of $6km$ around the ownship that defines the radius of consideration of intruders. Only intruders within this radius of the ownship will be modeled in the problem and all other intruders will be ignored.

We use the `FastMDP` algorithm defined in Chapter 5 to solve the MDP containing positive and negative rewards. We demonstrate this planner in a 2D aircraft simulation showing an overhead view of the ownship, the goal, and the intruders as shown in Figure 6.4.

The intruders are driven by a simple policy, which may either be deterministic or stochastic during an experiment. Intruders are spawned randomly in the space. Deterministic intruders maintain heading and airspeed during their flight. Stochastic intruders have a small probability of randomly changing their heading up to $\pm 25^\circ$ at each time step. In either case, if an intruder reaches the

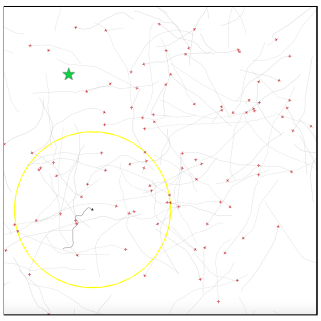

Figure 6.10 100 intruders

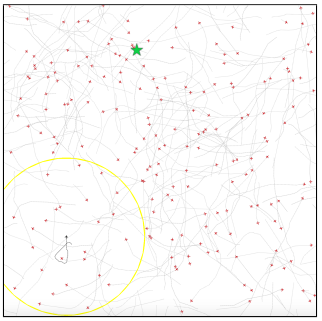

Figure 6.11 200 intruders

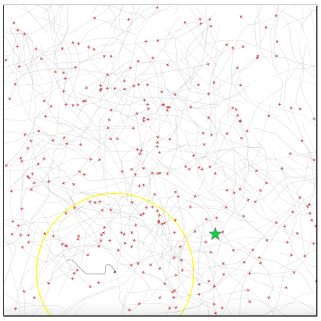

Figure 6.12 300 intruders

Figure 6.13: Visualization of different number of intruders to illustrate the difficulty of the collision avoidance problem.

boundary it is re-spawned in a new random location. A new MDP is created and solved at each time step allowing for dynamically changing obstacles.

As shown in Figure 6.6, decomposing the problem into very small MDPs with a fixed number of rewards makes each of the sub-MDPs a constant-time operation yielding an overall linear $O(m)$ performance where m is the number of intruder aircraft. For Figure 6.6 the code was run in a mode where it considered only a specific number of aircraft. Low level performance timers were used to record start and stop times of the algorithm's key processing phases: decoding observations, solving all of the MDPs, and computing the action from all of the MDPs' solutions. These times were summed into a value that captures the amount of time the algorithm runs each cycle within the overall simulation. The simulation ran for approx 1000 iterations to account for any variation. The mean of these iterations is plotted in Figure 6.6.

Timing tests were run on two computers. First a PC with a 2.8 GHz Intel i7 CPU. The second platform was an ARM based NVIDIA AGX Xavier board running in MAXN (30W + mode) with jetson_clocks.sh run to maximize clock speeds. Both tests were in python, single-threaded with no special hardware assistance such as GPUs or "hidden" computational libraries such as numba. Numpy is used by the algorithm.

In Figure 6.7, we instead study the agent's ability to avoid near midair collisions (NMACs) as we increase the number of intruders in the state space. For this measurement, we allow the agent to run for up to 10,000 steps. The state space is 800×800, so this provide ample ability for the agent to reach the goal even with extreme collision avoidance, but also prevents an infinite run which never terminates because it is infeasible to reach the goal. For each number of intruders, we run 100 episodes to determine how many times we reach the goal. We also record how many episodes result in a near midair collision (NMAC), which we define as coming within 150 meters of an intruder at any point during the episode. If an NMAC is detected, then the episode is terminated. Thus we should expect that as the number of NMACs grow, we should also see the number of goals reached reduce by an equal amount. The intruders in this experiment were the stochastic ones which at each time stamp have a small probability of changing their heading by $\pm 25°$. This results in a very unpredictable and challenging environment for the aircraft to maneuver within, especially when the number of intruders increases.

Sample videos showing the algorithm in action are available at: `https://youtu.be/NWI8T-SgHcU`

6.5 Conclusion

In this chapter, we have presented a novel computational guidance algorithm for flight planning for Unmanned Aerial Mobility (UAM) based on Markov Decision Processes. We present `Memoryless` , an efficient algorithm for solving MDPs that has no dependence on the size of the state space. We show how the algorithm can be used to solve a path planning and collision avoidance problem, and demonstrate that the algorithm's performance is suitable for online processing with real-time constraints. As the algorithm has no dependence on the size of the state space of the MDP, it is suitable for resource constrained embedded computing environments where memory and computation power is severely limited. In future work, we plan to integrate the algorithm into more advanced flight simulators and investigate multi-agent performance.

CHAPTER 7. APPLICATION: PURSUIT EVASION

7.1 Introduction

In this chapter based on Bertram and Wei (2019) we show how the `FastMDP` algorithm can be applied to 3D pursuit/evasion. Pursuit/evasion games pit two opponents against each other such that the pursuer must capture the evader. Within the aerospace community, pursuit/evasion of aircraft has long been of interest and is seeing a resurgence of interest due to a growing capability and acceptance of autonomous unmanned aircraft. Additionally, pursuit/evasion games are interesting in that they pose scalability challenges especially to UAV swarm applications. Problem formulations which lead to efficient and effective pursuit/evasion for 1 versus 1 (1v1) contests do not always allow efficient formulation with larger contests with multiple members per team (e.g., 2v2, 10v10). For problem formulations and algorithms that can support larger teams, it may be possible to solve the problem offline, but it may be exponentially harder and challenging in an online manner.

In this chapter, we propose a pursuit/evasion problem formulation based on Markov Decision Processes (MDPs) and the `FastMDP` algorithm from Chapter 5 to efficiently solve the problem even for large teams. The algorithm seamlessly switches between pursuit and evasion while simultaneously avoiding collisions with other aircraft and the ground. The algorithm is adaptable to multiple aircraft types through the use of forward projection of the aircraft dynamics, and a pseudo-6dof model is presented.

Our main contributions for this work are:

- Extension of the 2D algorithm with discrete state space Chapter 6 to a continuous 3D state space;
- Addition of a forward projection module that allows the algorithm to support any arbitrary aircraft type;

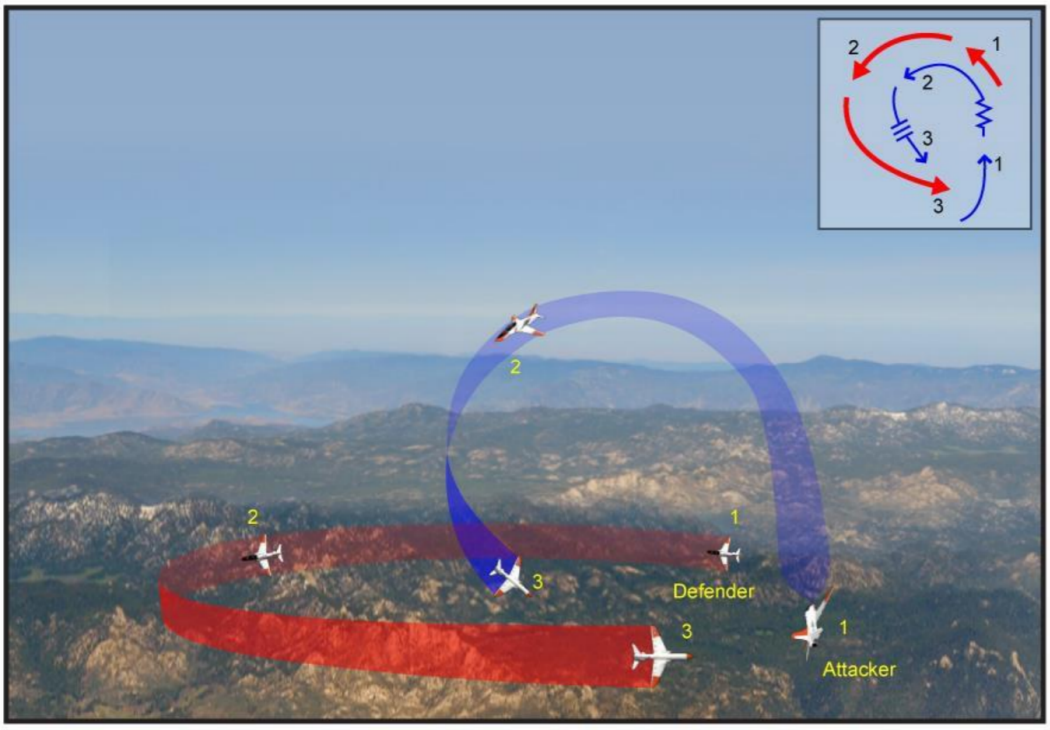

Figure 7.1: Example of a high yo-yo maneuver from public domain CNATRA of Naval Air Training(CNATRA) (2018) training manual.

- Demonstration of efficient algorithm performance that scales to large team sizes

We additionally develop a 3D visualization tool to evaluate the algorithm and to provide insight to readers on the complexity of the problem.

7.2 Pursuit Evasion Related Work

There is extensive work from many communities which address different approaches to pursuit-/evasion. We describe several approaches and discuss how they relate to Markov Decision Process approach used in this chapter.

Eklund et al. (2005) described a nonlinear model predictive control (NMPC) approach to a pursuit/evasion problem using a set of cost functions with repulsive and attractive natures to shape the behavior of the pursuer. An iterative optimization method was used to produce a solution at each time step using simplified aircraft dynamics. Multiple matrices in the NMPC formulation required tuning to obtain good behavior. It is worth noting that the cost functions used in their work are analogous to reward functions used for Markov Decision Processes.

Schopferer and Pfeifer (2015) proposed a method to perform flight planning in the presence of a uniform wind field, with the aircraft motion modeled with trochoids. The three dimensional flight path is constructed by superimposing a horizontal and vertical solution to obtain an approximate 3D path. A probabilistic roadmap planner is used to generate global plans.

Vector fields approaches have also been used for pursuit/evasion problems. Gonçalves et al. (2010) described a vector field approach for convergence, circulation, and correction around a closed loop pattern. Lawrence et al. (2008) presented a vector field approach for circular (or warped circular) patterns, and also describes a switching mechanism to handle waypoint following or arbitrary paths. Stable tracking of the vector field is explored using Lyapunov techniques. Vector fields can be viewed as similar in nature to the optimal policy that is generated by solving a Markov Decision Process. Where vector fields are generally applied over a continuous state space, MDP optimal policies normally describe actions that are intended to cause a transition from the current discrete state to a desired next discrete state.

Within the robotics and computational geometry community, pursuit/evasion is often considered in a different context. The pursuer(s) are attempting to search through an environment to observe the evader(s), similar to security guards searching through a museum for a potential intruder. Often in these problem formulations, the goal is identifying the minimum number of pursuers needed in order to guarantee that if an evader is present within the environment that it will be detected, and is not focused on tracking or chasing the evader as in the target problem of this chapter. However, these works are instructive as the algorithm used in this chapter is built on the recognition that an MDP can be represented as a graph. Examples of this type of pursuit/evasion problem are Guibas et al. (1997); LaValle et al. (1997); Kehagias et al. (2009). An example of graph based pursuit/evasion problem applied to graphs of infinite nodes is Lehner (2016), where they describe the problem as a cop-and-robbers problem and define a winning strategy as preventing the robber from visiting a node in the infinite graph infinitely many times. This allows strategies which either catch the robber or force the robber to flee 'to infinity'. Markov Decision Processes are normally viewed as a tree of sequential actions, but can also be understood as a graph. As most MDP problems normally have a discrete state space, this graph would normally also have a finite number of nodes. Our method provides a way to support MDP problem formulations with continuous state spaces, and the corresponding graph would then have an infinite number of nodes. Like the cop-and-robbers problem above, forcing an adversary to flee would be an acceptable strategy for our aircraft pursuit/evasion problem as well.

Shengde et al. (2014) proposed a continuous-time Markov Decision Process (CTMDP) approach where variable time steps are allowed to be taken within a discretized state space where the transition function is defined instead as a transition rate function, allowing the possible resulting state transitions to be predicted with varied time steps. The large state space is simplified by classifying the states into neutral, advantaged, disadvantaged, and mutually disadvantaged categories and a Bayesian method is used to determine the transition probabilities. Pursuit/evasion within a 2D grid world environment is considered.

Within the optimal control community, one area of related work is Differential Dynamic Programming (DDP) which uses dynamic programming to iteratively improve a local optimal control policy. Sun et al. (2018) used DDP to solve an adversarial aircraft pursuit/evasion problem, terming their approach as game-theoretic DDP (GT-DDP) by combining DDP with a min-max problem formulation. Differential Dynamic Programming and Markov Decision Processes have much in common and both stem from Bellman's original work on dynamic programming Bellman (1957). Where the optimal control field focuses on the Hamilton-Jacobi-Bellman (HJB) equation and differentiable dynamics, MDPs often generalize the dynamics into a (deterministic or stochastic) transition function which captures uncertainty about the environment through probabilities (similar to those used for Markov chains.) Comparing Sun et al. (2018) to this chapter's work, GT-DDP in Sun et al. (2018) does have a much richer capability to incorporate system dynamics, but this comes at the expense of additional computation time and a need for convergence of the iterative nature of the algorithm.

The most relevant chapter to this work is McGrew et al. (2010) which describes a Markov Decision Process based pursuit/evasion problem for aircraft using approximate dynamic programming. A state space was formed from a set of features which minimized mean squared error using a forward-backward search. Trajectory sampling was used to obtain training data that would be likely to have value during training. Reward shaping was used to guide the exploration to the desired behavior in the form of a scoring function heuristic developed by an expert. Rollout was used to extract a refined policy from the approximation computed via approximate dynamic programming (ADP) and was accelerated with a neural net. The dynamics model for the airplane used is a Dubin's airplane without any vertical components or altitude modeled.

There are some subtle differences between this chapter and the work in McGrew et al. (2010). McGrew et al. (2010) is a good example of using a variety of practical techniques to deal with the intractability of large MDP state spaces, whereas this work explicitly uses a state space designed to be intractable by traditional MDP methods via the use of a continuous state space resulting in an MDP with an infinite number of states in order to demonstrate scaling to continuous state spaces. McGrew et al. (2010) uses a 2D aircraft model, where this chapter uses a 3D pseudo-6DOf model to

demonstrate scaling to a continuous 3D state space and to demonstrate full maneuvering by the aircraft (e.g., loops, rolls, spirals). In this chapter, no reward shaping is required to speed up or aid convergence, as the underlying MDP is solved directly without relying on typical methods used for approximate dynamic programming. And finally, in McGrew et al. (2010) 1v1 pursuit/evasion is explored where in this chapter scaling to 10v10 teams is demonstrated.

Also of note are Park et al. (2016) and Zhang et al. (2018). Park et al. (2016) used a higher fidelity 3D model and a min-max approach over a sliding window to demonstrate 1 vs 1 pursuit/evasion, and while the behavior in simulation appears promising, the real-time performance of the algorithm is not reported. In Zhang et al. (2018), a reinforcement learning approach is taken using deep Q-learning using a 2-layer multi-layer perceptron as the function approximator, and with a modified epsilon-greedy exploration strategy where a heuristic function used in place of random action in order to avoid wasteful actions during exploration. Performance is examined in 2D.

7.3 Methodology

We use the algorithm described in Chapter 5 as the underlying guidance and collision avoidance algorithm which demonstrated collision avoidance in a 2D environment. The algorithm is extremely efficient and the chapter demonstrated good performance on a discretized state space. We extend the method to demonstrate performance in a continuous state space while also extending it to a 3D environment to demonstrate scaling to the higher dimensional space. Demonstration of scaling is further highlighted by showing large teams performing pursuit/evasion together. Finally, we introduce a pseudo-6DOF model allowing the aircraft to roll, pitch, and perform complex aerial maneuvers which serves to further demonstrate the power of this approach.

7.3.1 Dynamic Model

The aircraft kinematic model is a pseudo 6 degree of freedom (pseudo-6DOF) model which approximates fixed wing aircraft motion given inputs similar to stick and throttle inputs. The model provides a way to study the algorithms behavior without requiring full aerodynamics to be modelled.

The algorithm needs this pseudo-6DOF model to provide "forward prediction". This means that from a given current state, the model must be able to calculate the future state of applying a given set of possible control actions for a fixed number of timesteps. Any model which satisfies this requirement can be integrated with the algorithm, including full-fidelity 6DOF fixed-wing models, helicopters, quad rotors, and models with underlying autopilot controllers.

The model used is an extension of the pseudo-6DOF formulation in Park et al. (2016) and also incorporates a few additional terms in the model in Huynh et al. (1987). It should be considered as a simplified model of Huynh et al. (1987).

- n_x: Throttle acceleration directed out the nose of the aircraft in g's
- V: Airspeed in meters/second.
- γ: Flight path angle in radians.
- x, y, z: position in NED coordinates in meters where altitude $h = -z$
- ϕ: Roll angle in radians
- ψ: Horizontal azimuth angle in radians
- α: Angle of attack in radians with respect to the flight path vector

The inputs to the model are: (1) the thrust n_x, (2) the rate of change of angle of attack $\dot{\alpha}$ and (3) the rate of change of the roll angle $\dot{\phi}$.

The equations of motion for the aircraft are:

$$\dot{V} = g\left[n_x \cos\alpha - \sin\gamma\right], \tag{7.1}$$

$$\dot{\gamma} = \frac{g}{V}\left[n_f \cos\phi - \cos\gamma\right], \tag{7.2}$$

$$\dot{\psi} = g\left[\frac{n_f \sin\phi}{V\cos\gamma}\right], \tag{7.3}$$

where the acceleration exerted out the top of the aircraft n_f in gs is defined as:

$$n_f = n_x \sin\alpha + L, \tag{7.4}$$

with a lift acceleration of $L = 0.5$. Here, 1 "g" is a unit of acceleration equivalent to 9.8 m/s^2. L was chosen to provide some amount of lift while in flight to partially counteract gravity and provide a stable flight condition with a low positive α angle of attack in the pseudo-6dof model. For a true aerodynamic model, this lift varies by the velocity (Mach number), but this level of detail is omitted in our simplified pseudo-6dof.

The kinematic equations are:

$$\dot{x} = V\cos\gamma\cos\psi \tag{7.5}$$

$$\dot{y} = V\cos\gamma\sin\psi \tag{7.6}$$

$$\dot{z} = V\sin\gamma. \tag{7.7}$$

While this model is not aerodynamically comprehensive, it is sufficient to describe aircraft motion suitable for examining the algorithm behavior without loss of generality. Again, our algorithm can integrate with any aircraft dynamic model that provides a forward prediction.

7.3.2 Forward Projection

In order to determine the future state resulted from a given action, we use forward projection to simulate the dynamics forward in time. We use a discrete time step of 0.1 seconds and apply the control actions at each time step for a specified number of time steps.

For the purposes of determining the future state of an action, we forward project for 1 time step (0.1 second). After selecting an action and applying it to the simulation, we advance the simulation one time step (0.1 seconds). Thus an action is chosen at a 10 Hz rate with a 1 second forward projection horizon.

The simulated future states can be viewed as an approximation of the reachable states, and are applied to the solution of the Markov Decision Process (MDP) to determine the value of the potential future states the agent might reach. Thus the agent follows the optimal policy of the MDP at each time step by determining which future reachable state is most valuable, and then takes the action in the next time step that will lead it towards that state.

Each team is provided with different aircraft performance limits which serve to provide the "blue" team (team 0) with a performance advantage over the "red" team (team 1) and prevents deadlocks where neither team is able to obtain an advantage over the other. Table 7.1 lists the performance limits, where the speed of sound $Mach = 343\ m/s$. These limits were chosen to represent a highly maneuverable subsonic UAV and do not represent any real aircraft.

Table 7.1: Limits on aircraft performance for each team

Team	V_{min} (Mach)	V_{max} (Mach)	$\dot{\psi}_{\min}$ (rad/s)	$\dot{\psi}_{\max}$ (rad/s)	$\alpha_{\min}$ (rad)	$\alpha_{\max}$ (rad)
Blue	0.1	0.35	-1.5	-1.5	-.009	.69
Red	0.1	0.30	-1.3	-1.3	-.009	.52

7.3.3 State Space

We define the environment where the aircraft operates within a 25 km by 25 km by 25 km volume which is treated as a continuous state space. There are two teams of aircraft in this environment: a "blue" team and a "red" team. Each aircraft (an "ownship") is controlled by our proposed algorithm, and aircraft on the blue team have a slight performance advantage over aircraft on the red team.

The state includes all the information each ownship needs for its decision making: the full aircraft state of the ownship, the position and velocity of every teammate aircraft, and the position and velocity of every opponent aircraft.

Each ownship is aware of its own aircraft state produced by the pseudo-6DOF model. For each ownship, the state is formed by concatenating the following:

- ζ the pseudo-6DOF state: position x, y, z, the heading angle ψ, the roll angle ϕ, the flight path angle γ, the pitch angle θ, the angle of attack α, and the speed V.
- for each teammate $f_j, \forall j \in J$: the position $f_{j,x}, f_{j,y}, f_{j,z}$ and velocity $f_{j,v_x}, f_{j,v_y}, f_{j,v_z}$, and
- for each opponent aircraft $i_k, \forall k \in K$: the position $i_{k,x}, i_{k,y}, i_{k,z}$ and velocity $i_{k,v_x}, i_{k,v_y}, i_{k,v_z}$

$$s_o = [\zeta, f_1, \cdots, f_j, i_1, \cdots, i_m] \tag{7.8}$$

where j represents the number of teammates, and m represents the number of opponents.

7.3.4 Action Space

Inputs to the model are (1) the thrust n_x, (2) the rate of change of angle of attack $\dot{\alpha}$ and (3) the rate of change of the roll angle $\dot{\phi}$.

The action space is then:

$$A = \{\dot{\alpha}, \dot{\phi}, n_x\}. \tag{7.9}$$

There are two teams of aircraft $k \in \{0, 1\}$ where team $k = 0$ is the "blue team" and $k = 1$ is the "red team". When the teams' aircraft have equivalent performance, simulations often result in a stalemate which represent a Nash equilibrium where neither aircraft is able to gain advantage over the other. In these cases, simulation will not naturally terminate. Therefore, in the simulations we provide a performance advantage to the blue team which more naturally leads to simulations that terminate.

Table 7.2: Action choices for each team

Team	$\dot{\phi}$ (rad/s)	$\dot{\alpha}$ (rad/s)	n_x (g's)
Red	-1, -.8, $\cdots$, .8, 1	-.5, -.4, $\cdots$, .4, .5	0, 1, $\cdots$, 6
Blue	-1.5, -1.2, $\cdots$, 1.2, 1.5	-.5, -.4, $\cdots$, .4, .5	0, 1, $\cdots$, 8

7.3.5 Reward Function

The primary mechanism to control the behavior of an agent in a Markov Decision Process (MDP) is through the Reward Function. By providing positive and negative rewards to the agent, it is able to determine which actions lead to positive reward and the solution of an MDP maximizes the expectation of future reward. In our pursuit evasion problem, we will use positive and negative rewards that are coupled together to create tension between potential actions. For example, we will place a positive reward near the location of an aircraft to attract other aircraft, but we will also place a negative reward at the aircraft to prevent a collision. A natural equilibrium develops between these positive and negative rewards that generates the desired behavior of approaching another aircraft without colliding with it.

Following the approach used in Chapter 5, we will treat each negative reward as a "risk well", which is a region of negative reward (i.e., a penalty) which is more intense at the center and decays outward until a fixed radius is reached, where after no penalty is applied. We present our reward function in terms of the behaviors we wish to obtain in Table 7.3. In this table, $\hat{p}$ represents the current position of an aircraft (teammate or opponent) and $\hat{v}$ represents that aircraft's current linear velocity. In some cases we project the aircraft's position forward in time with an expression $\hat{p} + \hat{v}t$ and then define a range of time as in $\forall t \in \{0, 1, 2\}$ to indicate that we create a reward at the location of the aircraft at each timestep in the future indicated by the range of t.

All aircraft also receive a penalty below a certain altitude which prevents the aircraft from plummeting into the terrain. For this chapter, $h_{\max}$ is the maximum height of the terrain that is

Table 7.3: Rewards created for each ownship

For each teammate:

Magnitude	Decay factor	Location	Radius	Time steps	Comment
-100	.97	$\hat{p} + \hat{v}t$	$150 + 10t$	$\forall t \in \{0, 1, 2, 3, 4, 5\}$	Collision avoidance (5 rewards)
10	.999	$\hat{p}$	∞	N/A	Weak formation flight or clustering

For each opponent:

Magnitude	Decay factor	Location	Radius	Time steps	Comment
-300	.99	$\hat{p} + \hat{v}t$	$\hat{v}t$	$\forall t \in \{0, 1, 5, 10\}$	Collision avoidance (4 rewards)
200	.999	$\hat{p}$	∞	N/A	Pursuit

loaded into the simulation. We define a minimum safe altitude known as the "hard deck" in which we will allow the aircraft to fly. Any aircraft which goes below the hard deck for the purposes of the game has crashed and is removed from the simulation. We define the hard deck $h_{\text{deck}} = h_{\max} + 500$. For any state with an altitude of h from the hard deck up to an altitude of $h_{\text{penalty}} = h_{\text{deck}} + 1000$, a penalty is applied $r_{\text{penalty}} = -(10000 - h)$ which is a very strong negative reward that will override any other positive rewards in the game.

7.3.6 Algorithm

The algorithm used here is based off the `FastMDP` algorithm which efficiently solves the Markov Decision Process (MDP) problem by recognizing that the MDP rewards act as peaks in the value function and provide a structure to the resulting value function that can be exploited. Using this approach, in Chapter 6 were able to solve a 2D guidance and collision avoidance problem in a discretized state space very efficiently. The representation from Chapter 6 however cannot handle 3D position and does not handle a continuous state space or aircraft dynamics that are important for pursuit evasion.

We alter the `FastMDP` algorithm by extending it to handle 3D aircraft positions in a continuous state space. This alone is somewhat novel for Markov Decision Processes as they normally are restricted to discretized state spaces or require a function approximation technique to represent the value function.

We use forward projection to determine states that are reachable from the current state. We precompute the set of actions each agent can perform at a given time step (900 actions for team 0 (blue team), and 600 actions for team 1 (red team)). We forward project each of these actions for 1 time step (0.1 seconds) and then for 10 time steps (1.0 seconds). The 1.0 second forward projection is used as a window or horizon in which to estimate the potential value of each action the agent could take. Whichever action leads to a state with the highest value in the MDP is chosen as the action to perform. The action is selected and is used for 1 time step, where forward projection is again performed with a new 1 second planning horizon.

Algorithm 5 Pursuit Evasion with `FastMDP`

procedure PURSUIT EVASION(*ownshipState*, *worldState*)
  $\mathbf{S}_0 \leftarrow$ *randomized initial aircraft states*
  $\mathbf{A} \leftarrow$ *list of actions for each team (precomputed)*
  $\mathbf{L} \leftarrow$ *list of limits for each team (precomputed)*
  $\mathbf{S}_{t+1} \leftarrow$ allocated space
  while both teams have aircraft remaining **do**
    for each ownship **do**
      $s_t \leftarrow \mathbf{S}_t[ownship]$
      $k \leftarrow$ **team**$(ownship)$
      // Build peaks per Table 7.3
      $\mathbf{P}^+ \leftarrow$*build pos rewards*
      $\mathbf{P}^- \leftarrow$*build neg rewards in Standard Positive Form*
      // Perform forward projection per Section 7.3.2
      $\mathbf{\Delta_1} \leftarrow fwdProject(s_t, \mathbf{A}[k], \mathbf{L}[k], 0.1\ s)$
      $\mathbf{\Delta_{10}} \leftarrow fwdProject(s_t, \mathbf{A}[k], \mathbf{L}[k], 1.0\ s)$
      // Compute the value at each reachable state
      $\mathbf{V}^* \leftarrow$ *allocate space for each reachable state*
      for $s_j \in \mathbf{\Delta_{10}}$ **do**
        // First for positive peaks
        for $p_i \in \mathbf{P}^+$ **do**
          $d_p \leftarrow \|s_j - \mathbf{location}(p_i)\|_2$ ▷ distance
          $r_p \leftarrow \mathbf{reward}(p_i)$
          $\gamma_p \leftarrow \mathbf{discount}(p_i)$
          $\mathbf{V}^+(p_i) \leftarrow |r_p| \cdot \gamma_p^{d_p}$
        $V^+_{max} \leftarrow \max_{p_i} \mathbf{V}^+$
        // Next for negative peaks (in Standard Positive Form)
        for $n_i \in \mathbf{P}^-$ **do**
          $d_n \leftarrow \|s_j - \mathbf{location}(n_i)\|_2$ ▷ distance
          $\rho_n \leftarrow negDist_i < \mathbf{radius}(n_i)$ ▷ within radius
          $r_n \leftarrow \mathbf{reward}(n_i)$
          $\gamma_n \leftarrow \mathbf{discount}(n_i)$
          $\mathbf{V}^-(p_i) \leftarrow int(\rho_n) \cdot |r_n| \cdot \gamma_n^{d_n}$
        $V^-_{max} \leftarrow \max_{p_i} \mathbf{V}^-$
        // Hard deck penalty
        if $\mathbf{altitude}(s_t) < penaltyAlt$ **then**
          $V_{deck} \leftarrow 1000 - \mathbf{altitude}(s_t)$
        else
          $V_{deck} \leftarrow 0$
        $\mathbf{V}^*[s_t] \leftarrow V^+_{max} - V^-_{max} - V_{deck}$
      // Identify the most valuable action
      $i_{max} \leftarrow \arg\max_s(\mathbf{V}^*)$
      // For illustration, the corresponding value
      $maxValue \leftarrow \mathbf{V}^*[i_{max}]$
      // And the next state when taking the action
      $s_{t+1} \leftarrow \mathbf{\Delta_1}[i_{max}]$
      $\mathbf{S}_{t+1}[ownship] \leftarrow s_{t+1}$
    // Now that all aircraft have selected an action, apply it
    $\mathbf{S} \leftarrow \mathbf{S}_{t+1}$

All of these steps are optimized as much as possible for operation on a CPU. As the code is implemented in python, an optimization library known as numba is employed which recompiles key sections of the code as C code to obtain faster operation. Additionally, the code is written to take advantage of the numerical library numpy to perform vectorized operations over arrays. No GPU is used.

7.4 Experimental Setup

We demonstrate this MDP based planner in a 3D aircraft simulation showing a view of the two teams of aircraft. The simulation covers a configurable sized volume which contains a configurable number of team members on each of the two teams.

Simulation begins with both teams spawned randomly on opposing sides of the environment. The teams must each avoid collisions with team mates while simultaneously pursuing members of the opposing team using only the reward system we have defined above.

At each time step, the simulation generates the state updates for each ownship. Each ownship creates and solves its own MDP. Each ownship forward projects each possible action by 1 second, and then uses the solution of the MDP to determine which action results in the highest valued future state. The action selected with this method will then be applied in simulation for 1 timestamp (0.1 seconds). The actions of all aircraft from both sides are selected and performed simultaneously without knowing the selected actions of any other aircraft in the simulation. Simulation then advances by one time step. Note that a new MDP is calculated at each time step, which is made possible by the performance of the `FastMDP` algorithm.

In this pursuit/evasion game, we define a pursuer "capturing" an opponent if it is in a certain region behind the evading aircraft. The "control point" is defined as the position the evader was at 3 seconds previously. If the pursuer is within 100 meters of the control point and relative angle between the two velocity vectors of the aircraft is within 60 degrees, then the pursuer is close to the control point and pointing at the evader and we consider this a sufficient condition for the pursuer to be able to "capture" the evader (e.g., within range of some weapon). The pursuer must maintain this

condition for 30 consecutive time steps in order to successfully "hit" the evader, which is analogous to a weapon taking some time to track the evader. This is indicated visually in the simulation as a red pulsing rectangle around an aircraft that is in danger of being captured.

We build a scoring system that tracks the number of airplanes that have been captured. When a team's airplane is captured, the opposing team is awarded one point. Thus complete success is when one team reaches a score that equals the number of airplanes on the opposing team. A "win" is described as one team scoring higher than the other, with the other team necessarily incurring a "loss", and a "draw" is when both teams score the same.

We define a metric P_{win} to study the effect of the algorithm over N runs which is defined for a team as the number of wins the team obtained W over the number of runs: $P_{\text{win}} = \frac{W}{N}$. This metric can be applied to 1 vs 1 encounters and can scale to larger teams as well.

The P_{win} measurement alone is not sufficient. Beyond the probability of win, we also wish to define a metric that describes the survivability of the team. In a 10 vs 10 game, it is clearly better when when winning if all 10 of the teammates survive as compared to a win when only 1 of the teammates remain at the end. If we define the number of aircraft at the beginning of the contest as N_{t_0} and the number remaining at the end of the contest as N_{t_f}, then we can define the ratio of teammates that survived a given contest i as $P_{s_i} = N_{t_f}/N_{t_0}$. Over m contests, we define the overall probability of survivibility as $P_s = \frac{1}{m}\sum_{i=1}^{m} P_{s_i}$ where m is the number of contests and is the average probability that the team will survive the contest.

7.5 Results

In Figure 7.5, results are shown for a typical 1 versus 1 (1v1) encounter. As blue has a performance advantage, it is able to maneuver more effectively and is able to capture the red aircraft. Figure 7.7 shows the actions selected by the blue aircraft during this run, while Figure 7.9 shows the values of the pseudo-6DOF state variables during the run.

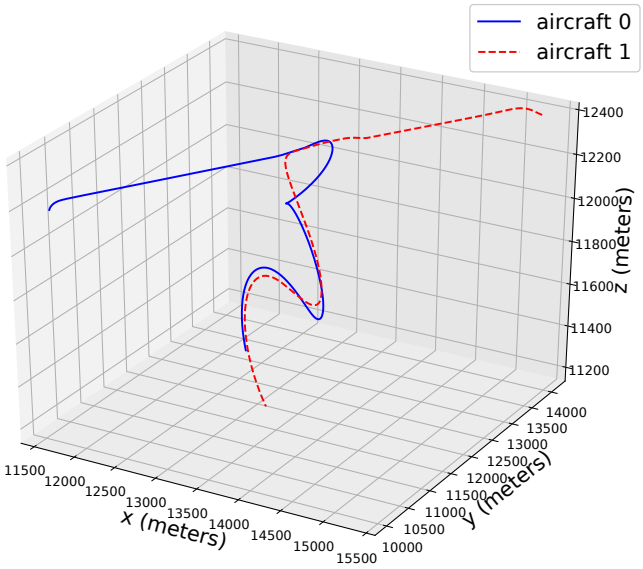

Figure 7.3 Trajectory of a sample 1v1 pursuit/evasion run

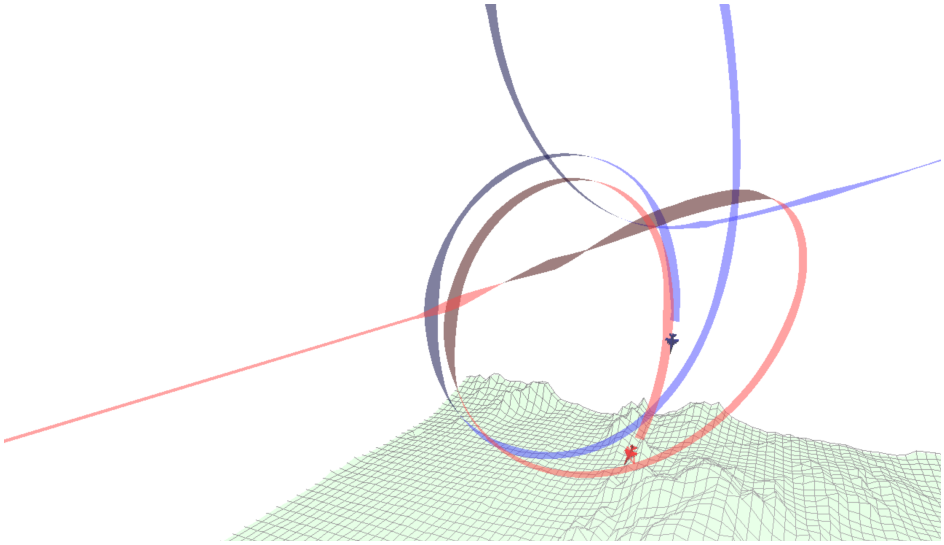

Figure 7.4 The same 1v1 run in a 3D visualization

Figure 7.5: Experimental results showing the performance of the algorithm for a 1v1 pursuit/evasion run. (a) shows the trajectories of two aircraft in a standard Matlab style plot. (b) shows the trajectories in a 3D visualization developed for this chapter where ribbons are used to show historical attitude a 3D aircraft is used to more readily show current aircraft attitude. Links to videos are provided for the interested reader in the results sections.

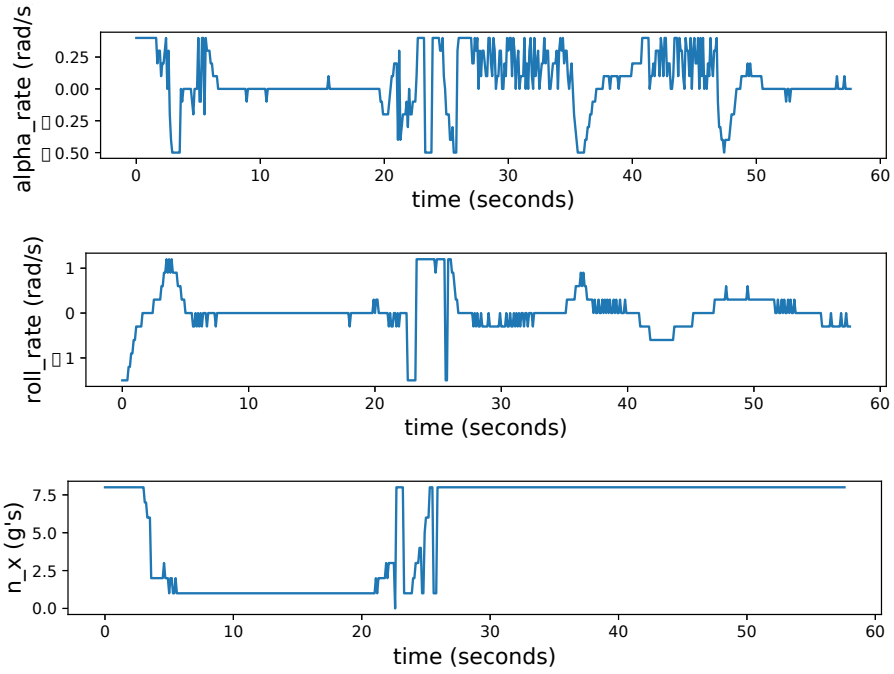

Figure 7.7: Experimental results showing the actions taken by the pursuer (blue aircraft) over time. Alpha rate here is analogous to pushing forward or pulling back on the stick. Roll rate is analogous to moving the stick from side to side. n_x is analogous to a throttle setting.

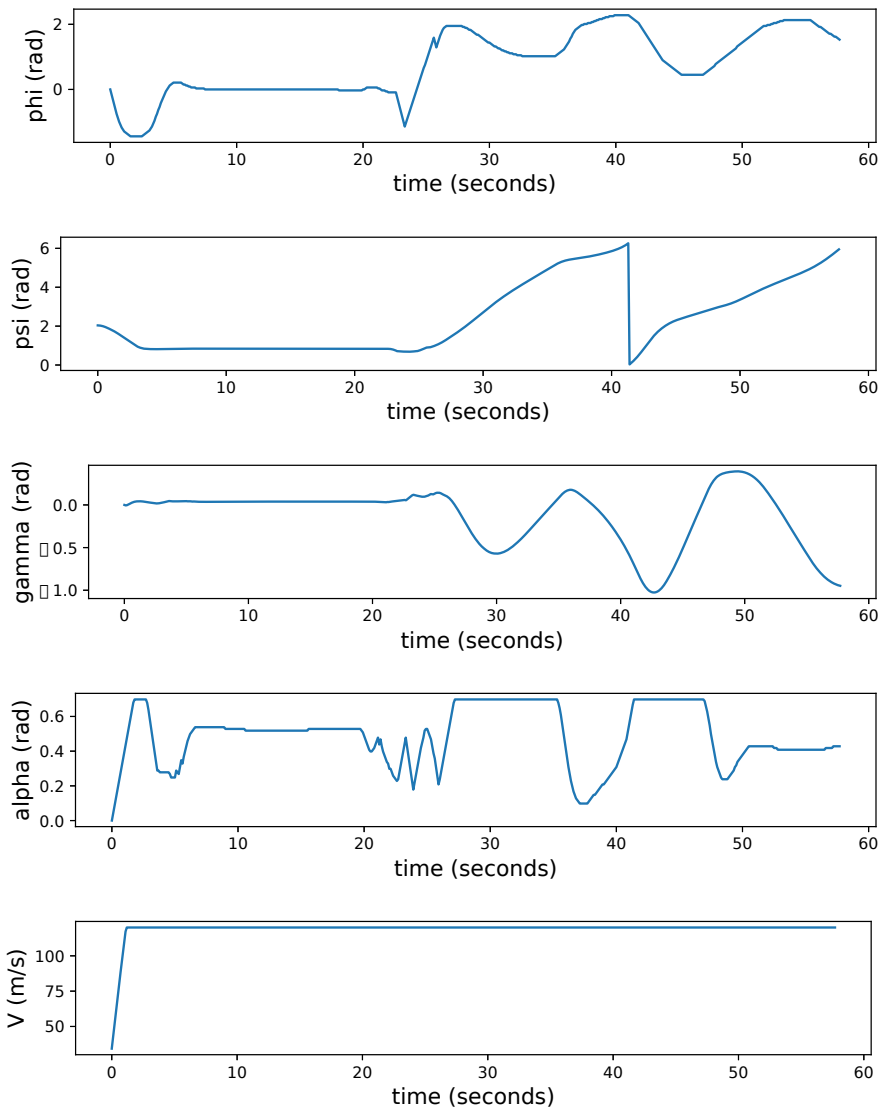

Figure 7.9: Experimental results showing the dynamics of the pursuer (blue aircraft) over time.

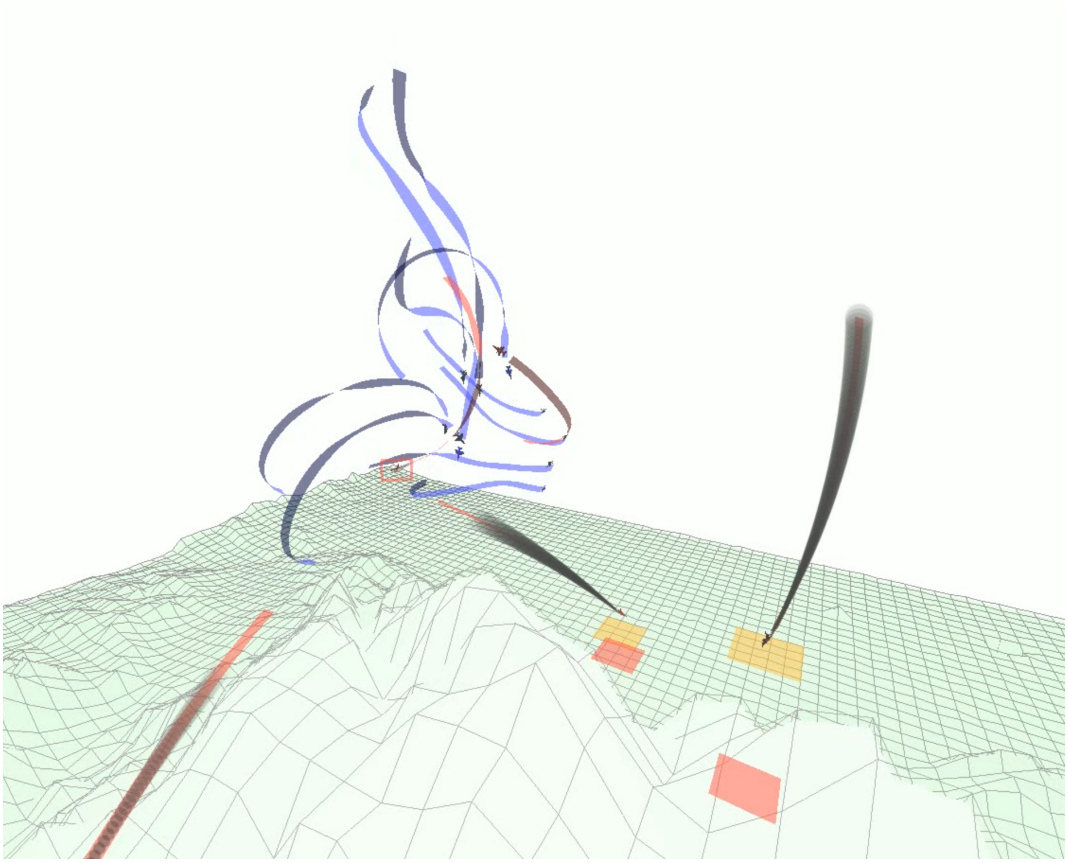

Figure 7.10: Screenshot from 10v10 video showing red rectangles indicating an aircraft is in danger of being captured. Once captured, an explosion is indicated, the aircraft loses all thrust, and smoke is emitted by the aircraft until it reaches the ground. As the aircraft approach a minimum safe altitude known as the hard deck (1000 ft above the maximum terrain height) an animated yellow and red square under the aircraft indicate that the aircraft is receiving a penalty for being too close to the ground and is attempting to pull up in response.

The P_{win} of the blue team for all experiments is shown in Table 7.4. This is an indicator that the algorithm is functioning correctly as the blue team was given an advantage in the selection of actions and in aircraft dynamics. Better dynamics allows the aircraft to maneuver into an offensive position more readily, leading to an expected high P_{win}. Also as expected as the airspace volume becomes more crowded and complex due to the increase in team size, the probability of survivability P_s tends to decrease.

Table 7.4: Probability of win P_{win} and Probability of survivability P_s of blue team as team size increases

Team Size	P_{win}	P_s
1v1	100%	100%
2v2	100%	100%
3v3	100%	100%
4v4	100%	100%
10v10	100%	99%
100v100	100%	97%

The amount of processing time required to formulate and solve the MDP for each agent at each timestep is summarized in Table 7.5. Processing was performed on a laptop with an Intel i9-8950HK CPU at 2.90 GHz. While the code is written in Python, it does take advantage of the Numba and Numpy Python libraries that are used to perfom optimized computation loops in C. Additionally, the underlying LLVM library may allow some Numba optimized code to take advantage of SIMD instruction in the CPU. No GPU acceleration is used.

Table 7.5: Processing time required for each agent on red or blue team as team size increases

Team Size	Mean (ms)
1v1	2.26
2v2	2.50
3v3	2.70
4v4	3.16
10v10	5.55
100v100	27.59

Videos of example runs of 1v1, 2v2, 3v3, 4v4, and 10v10 are available for viewing are provided in Table 7.6. Note that the size of the aircraft is exaggerated by a factor of 3 for improved visibility in the video.

Table 7.6: Links to videos

Team Size	URL
1v1	https://youtu.be/zGWXxtJUwk8
2v2	https://youtu.be/Q9O50cqpVtA
3v3	https://youtu.be/6Zok4sj43C4
4v4	https://youtu.be/qhI6av3oJN4
10v10	https://youtu.be/6twTWNRurwo

7.6 Conclusion

We have presented an efficient problem formulation for pursuit/evasion problems that scales to large numbers of teams (100v100) while remaining computationally efficient. This method formulates the problem as a Markov Decision Process (MDP) and uses a recently proposed approach in Bertram et al. (2019) to efficiently solve the MDP and is suitable for embedded systems commonly found on aircraft. The use of "risk wells" to represent the potential future actions of friendly and opposing aircraft allows the problem to remain tractable even as the number of aircraft per team increases.

CHAPTER 8. FUTURE WORK SUMMARY AND DISCUSSION

This thesis describes insights about the nature of Markov Decision Processes (MDP) and of their solutions which led to an extremely fast way to solve certain MDPs (those in which the state space maps to an underlying metric space and rewards are located at states within the state space.) Applications of the algorithm are presented which demonstrate the utility of the algorithm in aerospace related problems including navigating to a goal while avoiding collisions and a 3D pursuit evasion problem.

8.1 Algorithm Implementation Improvements

During the development of the algorithm over the time frame covered by this thesis, the algorithm progressed from an abstract idea with modest performance gains over value iteration to a more performance optimized form which achieves state of the art performance for the type of MDPs that the algorithm supports and demonstrated with great effect in the pursuit evasion problem.

The algorithm performance optimizations performed so far have largely focused on improvements to the order of operations resulting in $O(n)$ performance with respect to the number of rewards in the MDP. Most importantly, the algorithm performance optimizations free the algorithm from any dependence on the size of the state space, and can extend to fully continuous state spaces effortlessly. This means that the algorithm is capable of solving some MDP problems that cannot be solved unless function approximation methods such as approximate dynamic programming or deep learning methods are used.

As the algorithm has developed in sophistication, it is clear that there are additional algorithm performance optimizations that can be made, especially in terms of parallelization. Future research could explore methods for performing operations in parallel on GPUs and FPGAs in order to obtain either higher scalability, faster performance, or a combination.

8.2 Stochastic MDPs

There are fundamental aspects of the algorithm that can be improved. While the algorithm can currently exactly solve MDPs which contain positive rewards only, it can only provide an approximation for MDPs which also include negative rewards. Specifically, when negative rewards are placed close to each other within the space, an interaction occurs which causes error in the approximation.

Similarly, if the transition function is allowed to be stochastic, there is a very small error that is introduced into the approximation of the value function. Some experiments were performed which show that there is some non-linear relationship between the result produced by a deterministic MDP and the result produced by a corresponding stochastic MDP in which some uncertainty is introduced in the transition function, such as Gaussian noise on the state that results from a selected action. This relationship needs to be examined in more detail and quantified, which will ideally lead to a form of the algorithm that is more robust to uncertainty. As an aid to future researchers, Figure 8.1 shows an example of a very shallow depression (or "shadow") made in the value function by a negative reward in a stochastic MDP.

8.3 Incorporating Actions

The algorithm currently only handles rewards that are a function of the state ($R(s)$), but does not handle rewards that are also a function of the action ($R(s, a)$). An example of where this would be useful is applying a penalty to excessive roll actions to encourage an MDP to make turns which are more gentle. If would be straightforward to add this to the formulation and algorithm, but it is difficult to relate this to the distance metric that is currently used by the algorithm. If a way could be found that can relate these dimensions, or an alternative distance metric can be found that better fits with the effect of actions then the algorithm can be expanded to support a larger subset of MDPs.

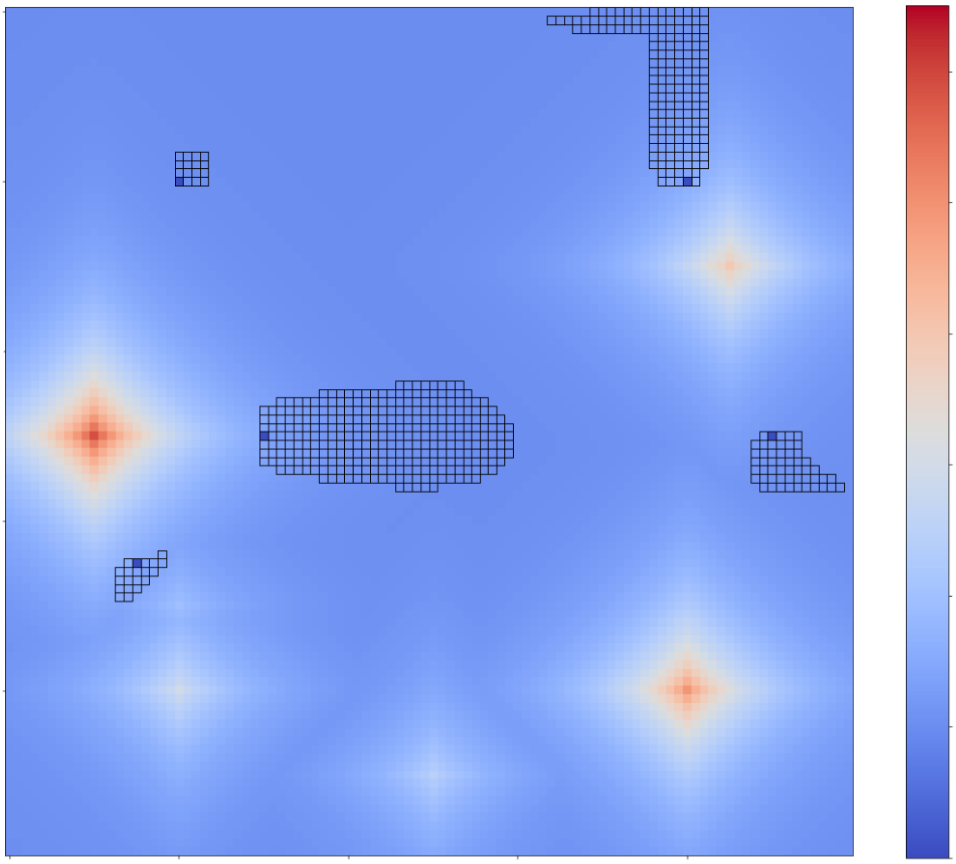

Figure 8.1: Stochastic rewards casting shadows in value function.

8.4 New Applications

The algorithm currently is restricted to state spaces which map to metric spaces in which there is an underlying metric such as Manhattan distance or euclidean distance between states. While it seems evident that the algorithm will never be able to be adapted to all possible MDPs (e.g., arbitrary connections between states as in a randomly connected graph), there should be other application domains which have non-linear spaces over which a useful metric can still be defined. In those cases, the algorithm can still be applied.

It is expected that this algorithm would likely be useful in tasks where multiple agents have independent goals and must avoid each other. Examples might be factory or warehouse floor robots which are utilizing the same space to move goods from incoming to outgoing areas. Aircraft terminal area guidance may also benefit from this algorithm both to manage in-air traffic and also ground operations.

BIBLIOGRAPHY

(2017). Future of urban mobility. `http://www.airbus.com/newsroom/news/en/2016/12/My-Kind-Of-Flyover.html`. Accessed: 2018-08-13.

(2017). Uber elevate | the future of urban air transport. `https://www.uber.com/info/elevate/`. Accessed: 2018-08-13.

(2018). Urban air mobility. `http://publicaffairs.airbus.com/default/public-affairs/int/en/our-topics/Urban-Air-Mobility.html`. Accessed: 2018-08-13.

Acikmese, B. and Ploen, S. R. (2007). Convex programming approach to powered descent guidance for mars landing. *Journal of Guidance, Control, and Dynamics*, 30(5):1353–1366.

Augugliaro, F., Schoellig, A. P., and D'Andrea, R. (2012). Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 1917–1922. IEEE.

Bellman, R. E. (1957). Dynamic programming.

Bertram, J. and Wei, P. (2018a). Explainable deterministic mdps. *arXiv preprint arXiv:1806.03492*.

Bertram, J., Yang, X., and Wei, P. (2018). Fast online exact solutions for deterministic mdps with sparse rewards. *ArXiv preprint arXiv:1805.02785*.

Bertram, J. R. and Wei, P. (2018b). Memoryless exact solutions for deterministic mdps with sparse rewards. *arXiv preprint arXiv:1805.07220*.

Bertram, J. R. and Wei, P. (2019). An efficient algorithm for multiple-pursuer-multiple-evader pursuit/evasion game. *arXiv preprint arXiv:1909.04171*.

Bertram, J. R., Yang, X., Brittain, M., and Wei, P. (2019). Online flight planner with dynamic obstacles for urban air mobility. In *2019 Aviation Technology, Integration, and Operations Conference*.

Bertsekas, D. P. (1995). *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA.

Bilimoria, K. D., Grabbe, S. R., Sheth, K. S., and Lee, H. Q. (2003). Performance evaluation of airborne separation assurance for free flight. *Air Traffic Control Quarterly*, 11(2):85–102.

Boyan, J. A. and Littman, M. L. (2001). Exact solutions to time-dependent mdps. In *Advances in Neural Information Processing Systems*, pages 1026–1032.

Chen, Y. F., Liu, M., Everett, M., and How, J. P. (2017). Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 285–292. IEEE.

Clari, M. S. V., Ruigrok, R. C., Hoekstra, J. M., and Visser, H. G. (2001). Cost-benefit study of free flight with airborne separation assurance. *Air Traffic Control Quarterly*, 9(4):287–309.

Cobano, J. A., Conde, R., Alejo, D., and Ollero, A. (2011). Path planning based on genetic algorithms and the monte-carlo method to avoid aerial vehicle collisions under uncertainties. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4429–4434. IEEE.

Dai, P. and Hansen, E. A. (2007). Prioritizing bellman backups without a priority queue. In *ICAPS*, pages 113–119.

de Guadalupe Garcia-Hernandez, M., Ruiz-Pinales, J., Onaindia, E., Aviña-Cervantes, J. G., Ledesma-Orozco, S., Alvarado-Mendez, E., and Reyes-Ballesteros, A. (2012). New prioritized value iteration for markov decision processes. *Artificial Intelligence Review*, 37(2):157–167.

Delahaye, D., Peyronne, C., Mongeau, M., and Puechmorel, S. (2010). Aircraft conflict resolution by genetic algorithm and b-spline approximation. In *EIWAC 2010, 2nd ENRI International Workshop on ATM/CNS*, pages 71–78.

Desaraju, V. R. and How, J. P. (2011). Decentralized path planning for multi-agent teams in complex environments using rapidly-exploring random trees. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4956–4961. IEEE.

Eklund, J. M., Sprinkle, J., and Sastry, S. (2005). Implementing and testing a nonlinear model predictive tracking controller for aerial pursuit/evasion games on a fixed wing aircraft. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 1509–1514. IEEE.

Enright, P. J. and Conway, B. A. (1992). Discrete approximations to optimal trajectories using direct transcription and nonlinear programming. *Journal of Guidance, Control, and Dynamics*, 15(4):994–1002.

Even-Dar, E., Kakade, S. M., and Mansour, Y. (2005). Experts in a Markov Decision Process. In *Advances in neural information processing systems*, pages 401–408.

Frazzoli, E., Mao, Z.-H., Oh, J.-H., and Feron, E. (2001). Resolution of conflicts involving many aircraft via semidefinite programming. *Journal of Guidance, Control, and Dynamics*, 24(1):79–86.

Gipson, L. (2017). Nasa embraces urban air mobility, calls for market study. `https://www.nasa.gov/aero/nasa-embraces-urban-air-mobility`. Accessed: 2018-01-19.

Gonçalves, V. M., Pimenta, L. C., Maia, C. A., Dutra, B. C., and Pereira, G. A. (2010). Vector fields for robot navigation along time-varying curves in n-dimensions. *IEEE Transactions on Robotics*, 26(4):647–659.

Guestrin, C., Koller, D., Parr, R., and Venkataraman, S. (2003). Efficient solution algorithms for factored mdps. *Journal of Artificial Intelligence Research*, 19:399–468.

Guibas, L. J., Latombe, J.-C., LaValle, S. M., Lin, D., and Motwani, R. (1997). Visibility-based pursuit-evasion in a polygonal environment. In *Workshop on Algorithms and Data Structures*, pages 17–30. Springer.

Gunning, D. (2017). Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web*.

Han, S.-C., Bang, H., and Yoo, C.-S. (2009). Proportional navigation-based collision avoidance for uavs. *International Journal of Control, Automation and Systems*, 7(4):553–565.

Hansen, E. A. and Zilberstein, S. (2001). Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62.

Harman, W. H. (1989). Tcas- a system for preventing midair collisions. *The Lincoln Laboratory Journal*, 2(3):437–457.

Hoekstra, J. M., van Gent, R. N., and Ruigrok, R. C. (2002). Designing for safety: the 'free flight'air traffic management concept. *Reliability Engineering & System Safety*, 75(2):215–232.

Hoffmann, G., Rajnarayan, D. G., Waslander, S. L., Dostal, D., Jang, J. S., and Tomlin, C. J. (2004). The stanford testbed of autonomous rotorcraft for multi agent control (starmac). In *The 23rd Digital Avionics Systems Conference (IEEE Cat. No. 04CH37576)*, volume 2, pages 12–E. IEEE.

Holden, J. and Goel, N. (2016). Fast-forwarding to a future of on-demand urban air transportation. *San Francisco, CA*.

Howlet, J. K., Schulein, G., and Mansur, M. H. (2004). A practical approach to obstacle field route planning for unmanned rotorcraft.

Huynh, H., Costes, P., and Aumasson, C. (1987). Numerical optimization of air combat maneuvers. In *Guidance, Navigation and Control Conference*, page 2392.

Inalhan, G., Stipanovic, D. M., and Tomlin, C. J. (2002). Decentralized optimization, with application to multiple aircraft coordination. In *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, volume 1, pages 1147–1155. IEEE.

Kahn, G., Zhang, T., Levine, S., and Abbeel, P. (2017). Plato: Policy learning using adaptive trajectory optimization. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 3342–3349. IEEE.

Kahne, S. and Frolow, I. (1996). Air traffic management: Evolution with technology. *IEEE Control Systems*, 16(4):12–21.

Karaman, S. and Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894.

Kavraki, L., Svestka, P., and Overmars, M. H. (1994). *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*, volume 1994. Unknown Publisher.

Kehagias, A., Hollinger, G., and Singh, S. (2009). A graph search algorithm for indoor pursuit/evasion. *Mathematical and Computer Modelling*, 50(9-10):1305–1317.

Kochenderfer, M. J. (2015). *Decision making under uncertainty: theory and application*. MIT press.

Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Krozel, J. and Peters, M. (1997). Conflict detection and resolution for free flight. *Air Traffic Control Quarterly*, 5(3):181–212.

Krozel, J., Peters, M., and Bilimoria, K. (2000). A decentralized control strategy for distributed air/ground traffic separation. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 4062.

Langelaan, J. and Rock, S. (2005). Towards autonomous uav flight in forests. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 5870.

LaValle, S. M. (1998). Rapidly-exploring random trees: A new tool for path planning.

LaValle, S. M., Lin, D., Guibas, L. J., Latombe, J.-C., and Motwani, R. (1997). Finding an unpredictable target in a workspace with obstacles. In *Proceedings of International Conference on Robotics and Automation*, volume 1, pages 737–742. IEEE.

Lawrence, D. A., Frew, E. W., and Pisano, W. J. (2008). Lyapunov vector fields for autonomous unmanned aircraft flight control. *Journal of Guidance, Control, and Dynamics*, 31(5):1220–1229.

Lehner, F. (2016). Pursuit evasion on infinite graphs. *Theoretical Computer Science*, 655:30–40.

McGrew, J. S., How, J. P., Williams, B., and Roy, N. (2010). Air-combat strategy using approximate dynamic programming. *Journal of guidance, control, and dynamics*, 33(5):1641–1654.

McMahan, H. B. and Gordon, G. J. (2005). Fast exact planning in markov decision processes. In *ICAPS*, pages 151–160.

Mellinger, D., Kushleyev, A., and Kumar, V. (2012). Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 477–483. IEEE.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning*, 13(1):103–130.

Morgan, D., Chung, S.-J., and Hadaegh, F. Y. (2014). Model predictive control of swarms of spacecraft using sequential convex programming. *Journal of Guidance, Control, and Dynamics*, 37(6):1725–1740.

Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A., Consiglio, M., and Chamberlain, J. (2015). Daidalus: detect and avoid alerting logic for unmanned systems.

of Naval Air Training(CNATRA), C. (2018). Flight training instruction: Basic fighter maneuvering (bfm) advanced nfo t-45c/vmts. Accessed: 2019-09-07.

Ong, H. Y. and Kochenderfer, M. J. (2016). Markov decision process-based distributed conflict resolution for drone air traffic management. *Journal of Guidance, Control, and Dynamics*, pages 69–80.

Pallottino, L., Feron, E. M., and Bicchi, A. (2002). Conflict resolution problems for air traffic management systems solved with mixed integer programming. *IEEE transactions on intelligent transportation systems*, 3(1):3–11.

Pallottino, L., Scordio, V. G., Frazzoli, E., and Bicchi, A. (2006). Probabilistic verification of a decentralized policy for conflict resolution in multi-agent systems. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 2448–2453. IEEE.

Papadimitriou, C. H. and Tsitsiklis, J. N. (1987). The complexity of markov decision processes. *Mathematics of operations research*, 12(3):441–450.

Park, H., Lee, B.-Y., Tahk, M.-J., and Yoo, D.-W. (2016). Differential game based air combat maneuver generation using scoring function matrix. *International Journal of Aeronautical and Space Sciences*, 17(2):204–213.

Park, J.-W., Oh, H.-D., and Tahk, M.-J. (2008). Uav collision avoidance based on geometric approach. In *SICE Annual Conference, 2008*, pages 2122–2126. IEEE.

Pontani, M. and Conway, B. A. (2010). Particle swarm optimization applied to space trajectories. *Journal of Guidance, Control, and Dynamics*, 33(5):1429–1441.

Powell, W. B. (2007). *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons.

Purwin, O., D'Andrea, R., and Lee, J.-W. (2008). Theory and implementation of path planning by negotiation for decentralized agents. *Robotics and Autonomous Systems*, 56(5):422–436.

Raghunathan, A. U., Gopal, V., Subramanian, D., Biegler, L. T., and Samad, T. (2004). Dynamic optimization strategies for three-dimensional conflict resolution of multiple aircraft. *Journal of guidance, control, and dynamics*, 27(4):586–594.

Richards, A. and How, J. (2004). Decentralized model predictive control of cooperating uavs. In *43rd IEEE Conference on Decision and Control*, volume 4, pages 4286–4291. Citeseer.

Richards, A. and How, J. P. (2002). Aircraft trajectory planning with collision avoidance using mixed integer linear programming. In *American Control Conference, 2002. Proceedings of the 2002*, volume 3, pages 1936–1941. IEEE.

Schopferer, S. and Pfeifer, T. (2015). Performance-aware flight path planning for unmanned aircraft in uniform wind fields. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1138–1147. IEEE.

Schouwenaars, T., De Moor, B., Feron, E., and How, J. (2001). Mixed integer programming for multi-vehicle path planning. In *Control Conference (ECC), 2001 European*, pages 2603–2608. IEEE.

Schouwenaars, T., How, J., and Feron, E. (2004). Decentralized cooperative trajectory planning of multiple aircraft with hard safety guarantees. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 5141.

Schuurmans, D. and Patrascu, R. (2002). Direct value-approximation for factored mdps. In *Advances in Neural Information Processing Systems*, pages 1579–1586.

Shengde, J., Xiangke, W., Xiaoting, J., and Huayong, Z. (2014). A continuous-time markov decision process based method on pursuit-evasion problem. *IFAC Proceedings Volumes*, 47(3):620–625.

Shim, D. H., Kim, H. J., and Sastry, S. (2003). Decentralized nonlinear model predictive control of multiple flying robots. In *Decision and control, 2003. Proceedings. 42nd IEEE conference on*, volume 4, pages 3621–3626. IEEE.

Shim, D. H. and Sastry, S. (2007). An evasive maneuvering algorithm for uavs in see-and-avoid situations. In *American Control Conference, 2007. ACC'07*, pages 3886–3891. IEEE.

Shortliffe, E. (2012). *Computer-based medical consultations: MYCIN*, volume 2. Elsevier.

Siegwart, R., Nourbakhsh, I. R., and Scaramuzza, D. (2011). *Introduction to autonomous mobile robots*. MIT press.

Sigaud, O. and Buffet, O. (2013). *Markov decision processes in artificial intelligence*. John Wiley & Sons.

Sigurd, K. and How, J. (2003). Uav trajectory design using total field collision avoidance. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 5728.

Sun, W., Pan, Y., Lim, J., Theodorou, E. A., and Tsiotras, P. (2018). Min-max differential dynamic programming: Continuous and discrete time formulations. *Journal of Guidance, Control, and Dynamics*, 41(12):2568–2580.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Szita, I., Takács, B., and Lörincz, A. (2002). ε-mdps: Learning in varying environments. *Journal of Machine Learning Research*, 3(Aug):145–174.

Tomlin, C., Pappas, G. J., and Sastry, S. (1998). Conflict resolution for air traffic management: A study in multiagent hybrid systems. *IEEE Transactions on automatic control*, 43(4):509–521.

Van Den Berg, J., Guy, S. J., Lin, M., and Manocha, D. (2011). Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer.

Van Lent, M., Fisher, W., and Mancuso, M. (2004). An explainable artificial intelligence system for small-unit tactical behavior. In *Proceedings of the National Conference on Artificial Intelligence*, pages 900–907. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., and Tsang, J. (2017). Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 5392–5402.

Vela, A., Solak, S., Singhose, W., and Clarke, J.-P. (2009). A mixed integer program for flight-level assignment and speed control for conflict resolution. In *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, pages 5219–5226. IEEE.

Wingate, D. and Seppi, K. D. (2005). Prioritization methods for accelerating mdp solvers. *Journal of Machine Learning Research*, 6(May):851–881.

Wollkind, S., Valasek, J., and Ioerger, T. (2004). Automated conflict resolution for air traffic management using cooperative multiagent negotiation. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 4992.

Yang, X. and Wei, P. (2018). Autonomous on-demand free flight operations in urban air mobility using monte carlo tree search. In *International Conference on Research in Air Transportation (ICRAT), Barcelona, Spain*.

Yu, J. Y. and Mannor, S. (2009). Online learning in markov decision processes with arbitrarily changing rewards and transitions. In *Game Theory for Networks, 2009. GameNets' 09. International Conference on*, pages 314–322. IEEE.

Yu, J. Y., Mannor, S., and Shimkin, N. (2008). Markov decision processes with arbitrary reward processes. In *European Workshop on Reinforcement Learning*, pages 268–281. Springer.

Zhang, T., Kahn, G., Levine, S., and Abbeel, P. (2016). Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 528–535. IEEE.

Zhang, X., Liu, G., Yang, C., and Wu, J. (2018). Research on air confrontation maneuver decision-making method based on reinforcement learning. *Electronics*, 7(11):279.