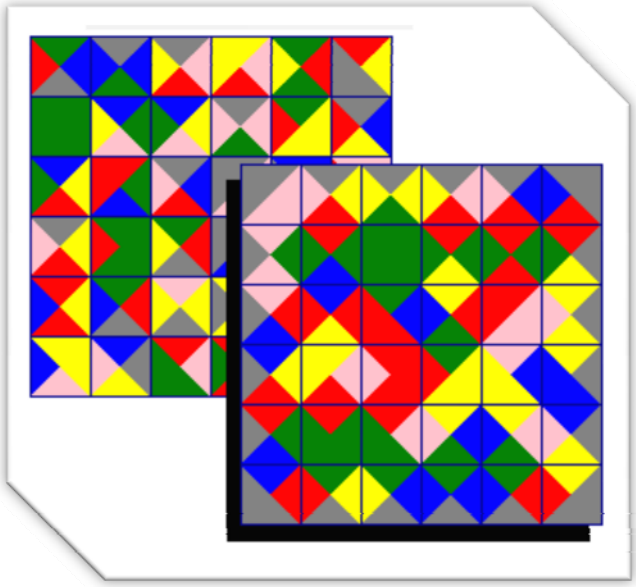
# Design and Analysis of ACO-algorithms for edge-matching problems



Carl Martin Dissing Söderlind

Kgs. Lyngby 2010

DTU Informatics
Department of Informatics and Mathematical Modelling

Technical University of Denmark

Building 321, DK-2800 Kgs. Lyngby, Denmark
Phone +45 4525 3351, Fax +45 4588 2673
reception@imm.dtu.dk
www.imm.dtu.dk

# Abstract

*Ant colony optimization algorithms are getting more and more accepted as optimization algorithms since the first ve rsion was introduced in 1991. Back then it did not perform very well but several improvements changed that and the ACO algorithms are now seen as very well performing algorithms on a number of problems.*

*The edge matching puzzle is an old but interesting problem which has gained extra attention by mathematicians by the Eternity II release in 2007 with a first price of $2.000.000 for the first to solve a very hard edge matching puzzle.*

*The purpose of this thesis is to try to create an ACO algorithm to work on edge matching puzzles.*

*The problem will be studied in order to see what makes one puzzle harder to solve than another.*

# Preface

This thesis is the final project at the Department of Informatics and Mathematical Modeling at the Technical University of Denmark for receiving the Master of Science degree in Engineering, M. Sc. Eng.

The thesis is a result of five months research from November 2009 - March 2010 where the main topic was creation of an optimization algorithm for the edge matching problem

Kgs. Lyngby, March 2010

Carl Martin Dissing Söderlind, s031850

# Acknowledgements

I would like to thank my supervisor at the department of Informatics and Mathematical Modeling at the Technical University of Denmark, Carsten Witt, for guidance during the project.

I would also like to thank Paul Fisher at the department of Informatics and Mathematical Modeling at the Technical University of Denmark for arranging the project and setting me up with my supervisor.

At last I would like to thank family and friends in my everyday life for keeping up my spirit.

# Contents

**Abstract** i

**Preface** iii

**Acknowledgements** v

**List of Tables and Figures** ix

**1 Introduction** 1

1.1 Thesis statement ..... 1
1.2 Motivation ..... 2
1.3 Outline of the thesis ..... 2

**2 Background** 3

2.1 Simple algorithms on $\mathcal{NP}$ problems ..... 3
2.2 Previous work with edge matching puzzles ..... 7
2.3 Previous work with ACO on $\mathcal{NP}$ problems ..... 8

**3 Theory** 9

3.1 Edge matching puzzle ..... 9
3.2 ACO ..... 15

**4 An ACO algorithm for the Edge Matching Puzzle** 21

4.1 Introductory algorithms ..... 21
4.2 The ACO for the Edge Matching Puzzle ..... 22

**5 Implementation** 26

5.1 The puzzle ..... 27
5.2 The backtracking algorithm ..... 28
5.3 The greedy algorithm ..... 28
5.4 The local search algorithm ..... 29
5.5 The ACO version 1 ..... 29
5.6 The ACO version 2 ..... 30

**6 Results and evaluation 31**

6.1 The puzzles ..... 31
6.2 The backtracking algorithm ..... 32
The greedy algorithm ..... 33
6.3 The local search algorithm ..... 34
6.4 The ACO algorithm version 1 ..... 34
6.5 The ACO algorithm version 2 ..... 36

**7 Conclusion 38**

**8 Bibliography 40**

# List of Tables and Figures

Table 1 – Piece matches – signed/unsigned ..... 10
Table 2 – Selected puzzles ..... 31
Table 3 – Results of greedy algorithm ..... 33
Table 4 – Results of local search algorithm ..... 34
Table 5 – Finding optimal values ACO version 1 ..... 35
Table 6 – ACO version 1 results ..... 35
Table 7 – Finding optimal values ACO version 1 ..... 36
Table 8 – ACO version 2 results ..... 37

Figure 1 – A puzzle with and without border ..... 11
Figure 2 – 4x4 board with 24 colors ..... 13
Figure 3 – Illegal placement of border and interior piece ..... 13
Figure 4 – Joins of pieces ..... 15
Figure 5 – ACO version 1 graph representation ..... 23
Figure 6 – ACO version 2 graph representation ..... 24
Figure 7 – The application ..... 26

CHAPTER 1

# Introduction

## 1.1 Thesis statement

The edge matching puzzle is about placing pieces on a board in a right way. The puzzle is solved when every piece is placed and every side of the pieces matches the neighboring pieces' sides.

This thesis is about finding as good solutions as possible to the edge matching puzzle problem with cleverly designed algorithms and without the use of exhaustive search. Previously contributions of finding solutions to the problem will be looked into to gain knowledge.

The first step is to analyze the combinatorial structure of the edge matching puzzles. Find out how much it affects the complexity of the puzzle depending on the board size, how many colors the problem consists of and other variables.

The next step is to implement the algorithm. It should be based on the ant colony optimization algorithm and should take use of the observations in the analysis to get a fine tuned algorithm for the problem. This will in ACO be used in the computation of the desirability to choose one piece over the others.

At first a small problem with a few colors and a small board size will be constructed to determine if the tunings are working or not. When usable tunings has been obtained more complex problems can be tried solved.

The main idea for this project came from the Eternity II puzzle game (1) which is an edge matching puzzle problem proven to be so complex so that no solution will be found in reasonable time.

## 1.2 Motivation

If good solutions can be found to the edge matching problem, using a specially designed ant colony optimization algorithm, the results and experience can often be used in other similar projects. If one idea works on one specific problem it is very likely it will work on others as well.

Since ACO is a fairly new group of algorithms there are still plenty of problems to try out which is why it would be interesting to use it on the edge matching puzzle. If good results are gained the ACO will add another problem to its success list, proving once again that it belongs in the top of optimization algorithms.

## 1.3 Outline of the thesis

This thesis is divided into chapters that start by giving a general knowledge of the background of $\mathcal{NP}$ problems and how they can be solved with simple optimization algorithms as well as with ACO algorithms in chapter 2. Previous solving methods to the edge matching puzzle are also mentioned in this chapter.

From here the focus goes to the theory of the edge matching puzzle and of the ACO algorithms in chapter 3. It can be seen what separates a hard problem from a weak problem dealing with the edge matching puzzle and for the ACO algorithms the different techniques are reviewed.

In chapter 4 the suggestion for an ACO algorithm for the edge matching puzzle problem is shown. Some simple algorithms for the problem are reviewed first before going to the actual ACO algorithm.

Chapter 5 explains how the algorithms from chapter 4 are implemented.

In chapter 6 the results of the algorithms are shown and an evaluation of each of the results is given.

Chapter 7 is the conclusion of the thesis which will evaluate the whole process as well as give suggestions for improvements.

CHAPTER 2

# Background

In this chapter previous work on different subjects will be discussed. In the first subsection examples of how simple optimization algorithms work on selected problems are looked into. The will provide knowledge of how algorithms are applied to problems that can be used later in the creation of the ACO algorithm for the edge matching puzzle. In the next subsection different strategies of how to obtain acceptable solutions to the edge matching puzzle is studied. This will give an understanding of the problem and its limitations. The final subsection will have examples of how ant colony optimization algorithms are applied to similar problems.

## 2.1 Simple algorithms on $\mathcal{NP}$ problems

$\mathcal{NP}$-complete problems are problems that cannot be solved in polynomial time. The definition can be seen in (2).This means that there is no efficient way to find a solution to the problem. To solve these problems, or at least to reach an acceptable solution, there are different approaches spread over a number of variants of algorithms. A number of problems will be reviewed before the solving techniques are applied to them.

### 2.1.1 The problems

The first problem mentioned is probably the most widely used as an example known as the travelling salesman (TSP). A number of cities all have to be visited by a salesman. The salesman can go from any one city to another until all cities have been visited exactly once. The total distance is all the distances between the cities added together. The problem is to find the shortest path, or a path of acceptable length, for the salesman to travel.

The next problem is the set covering problem (SCP) which is about choosing a minimum number of sets, all containing a number of elements, while cover-

ing all the elements. The different sets can hold any number of elements and the elements can be present in more than one set.

The last problem is the quadratic assignment problem (QAP) where a set of facilities must be assigned to a set of locations. With given flows between the facilities and given distances between the locations the problem lies in placing the facilities on the locations to get a high flow between the facilities and at the same time a short distance between the locations. It is the sum of the product of these parameters that must be minimized. This problem can be seen as an expanded version of TSP since it is about minimizing distance but is at the same time known to be one of the hardest problems to solve because of the extra facility parameter added to the problem.

### 2.1.2 Backtrack

If you want to be sure that you have the optimal solution to problems, where you do not know exactly what the best result is, one way is to use a backtracking algorithm. This algorithm is very time demanding and is in general only usable with small instances of a problem to find a solution. The backtracking algorithm can be found in almost all literature dealing with optimization problems. In short the algorithm works by calling a recursive function which creates all possible states from the state it was called from. Again for each of these states the recursive function is called. Each time a state has reaches a point where it no longer can create new states it backtracks to the previous state and rejects that state as a solution. When the algorithm is finished all the solutions to the problem has been returned.

The Travelling Salesman problem is one problem where the algorithm can be applied if the number of cities is not too high. This is dependent on the efficiency of the computer running the algorithm and how long time you are willing to let it work. Since The Travelling Salesman problem is about finding the shortest cycle in a graph it does not matter at which vertex it starts. The backtracking algorithm is initialized with the construction graph and a random vertex. The algorithm calls the recursive method with the initialized parameters and the method goes though all the edges of the vertex to find the neighboring vertices. For each of these vertices the recursive method is called once again. For each of the instances there is kept track of which vertices that has already been visited. When the algorithm is complete all possible routes has been found and from here it should be straight forward to locate the shortest

route. Since the shortest route length is not known beforehand all possible solutions have to be found before determining which is shortest.

With the set covering problem all possible combinations also have to be tried before the smallest set can be determined. The algorithm adds one set after another and at all time makes sure that the set covers at least one new element which has not yet been covered. When all elements are covered the list of selected sets is a solution. The solution is saved and the algorithm backtracks from the last state to add different sets until all possible combinations are tried out.

The quadratic assignment problem is as mentioned an extension of TSP so the obvious way to backtrack would be in the same manner. The difference is that for each location there is also a facility that must be placed. The search area increases exponentially for each location added to the problem comparing to the TSP. So much smaller instances can be handled by the backtracking algorithm in comparison to a similar TSP problem.

### 2.1.3 Greedy

An algorithm that is fast but for a lot of problems does not return a very good solution is the greedy algorithm. As the name suggests it will always choose the option that gives the best result in the current state. The basic greedy algorithm will always return the same result, because the best solution in the moment will never change no matter how many times the algorithm is run. Custom made greedy algorithms can be implemented to find alternative solutions depending on parameters such as a starting point.

For the Travelling Salesman problem the greedy algorithm will for each step choose the edge with the shortest distance. To guarantee that a full cycle, where all vertices have been visited, edges will be removed each time an edge has been chosen. There is no guarantee that the removed edges should not be part of the solution which is why the solution given by this algorithm cannot be guaranteed to be the best.

For the set covering problem the greedy algorithm would simply continue to select the set that has the most elements in it that has not already been covered, until all elements are covered. This will definitely not guarantee the best result. When choosing the set which covers most elements there are no guarantee that the rest of the elements will be in separate sets, and by that creating a bad solution.

The quadratic assignment problem would for each step choose the product with the highest value. Just like the TSP a lot of edges would be deleted, where one of them might be required in the best solution.

### 2.1.4 Local Search

The local search algorithm is an algorithm that from one solution tries to find new solutions by making simple mutations. A mutation is a random move that makes one difference in the solution. This could be swapping two edges in a graph such that the outgoing vertex of edge one now point to the ingoing vertex of edge two and vice-versa. When the mutation is done the new solution is checked to see if it is better than the old. If this is the case the new solution is now the solution from where the local search is run, otherwise the old solution remains. At sometime during the local search the solution will most likely reach a local optimum meaning that no matter where the mutation is made no better solution is found, and this is even though the best solution has not been found yet. To get out of the local optimum more mutations have to be made before checking the solution. When making more mutations the algorithm is often called k-opt where k is the number of mutation made before the check. In (3) they come up with a suggestion for choosing the k value according to a Poisson distribution with $\lambda = 1$ since all possible mutations of a solution have the same probability to get chosen. This allows the k to be variable but at the same time tries to minimize the number of mutations since the probability curve falls the higher the k gets.

Using the local search on the travelling salesman problem the mutation works as in the explanation by swapping two edges which is actually a 2-opt since two edges are changed. 3-opt is another version where, as the name suggests, three edges are swapped, but here more connections can be made making the algorithm work more but probably getting better results and possible better to avoid local optimum.

For the set covering problem a mutation would be to remove a set covering some element and choose another set that covers the element. After that redundant sets should be removed if present. This means sets that are covering elements where all of them already been covered by other sets.

The quadratic assignment problem is again not that different from the TSP. It would work like it does for TSP but it would have a lot more edges to choose from making it harder to solve.

## 2.2 Previous work with edge matching puzzles

Pierre Schuas and Yves Deville came up with very well working algorithm to get a good result of the Eternity II puzzle (4). Their algorithm is basically a hybrid of to solving methods. The first method is using constraint programming. The constraints of a problem are explained in chapter 3.1. A number of variables are defined representing the pieces and the possible placements of them. A solver tries to assign the variables taking the constraints into account. The solver works like a custom made backtracking algorithm that takes the constraints into account when placing the pieces. The conclusion made by Schaus and Deville is that the eternity II puzzle is too hard for constraint programming if all the constraints should be met. They come up with the idea to remove some of the constraints so that some non-matching edges are allowed. The positions for the pieces with non-matching edges are not chosen randomly. By choosing, as they put it themselves, only some of the black positions on a chess board they are guaranteed not to have two positions being adjacent to each other. This is important in the next method they will apply. The non-matching pieces and their positions will be used in an assignment algorithm to find the best matches for the pieces. The assignment algorithm maximizes the total matches by finding the best assignment of the pieces. Several algorithms exist for this where the Hungarian algorithm is one of the most used (5; 6) As their best result they matched 458 of 480 on the Eternity II puzzle which is one of the best known results made by an algorithm.

As expected the constraint programming in itself does not provide good solutions to such a big problem as the Eternity II problem, but on smaller instances it can be used and it is very well suited to find all solutions to a problem. Using the assignment algorithm is actually very clever since perfect assignment can be found very fast when choosing the pieces the way they do. Looking at their solution one must conclude that their algorithm is very effective.

Jorge Muñoz, German Gutierrez, and Araceli Sanchis have made and algorithm based on local search to try out on the Eternity II puzzle(7). Here there are two different mutations. One is where two randomly pieces are exchanged and the other where a randomly chosen piece is rotated. They claim that the number of mutations before the check should be between 1 and 10. This sounds very likely since it corresponds to the Poisson distribution as explained in 2.1.4. The result they get with this algorithm is not as impressive as the previous with only 366/480 at most.

## 2.3 Previous work with ACO on $\mathcal{NP}$ problems

Ant colony optimization algorithms have been applied to several problems and new variants of ACO are still being tested on some of the earliest problems applied to ACO, because they are the most tested upon. The most tested problems are the ones already discussed in the previous subchapters. The description of the three main ACO algorithms can be found in chapter 3.2.

When applying ACO algorithms to the travelling salesman problem a number of ants are placed randomly on different cities ant from here they walk the edges until all cities have been visited. According to a decision policy the edges are chosen from two aspects: The desirability to choose an edge is given by how much it has been travelled before, and a heuristic function of how good the choice is. For the TSP it is preferred to find the shortest path so the heuristic function should return a higher value for a short edge than a long. This is done very simple by dividing one with the length of the edge. More formal the function will be $h(n) = 1/d_{i,j}$, where $d_{i,j}$ is the distance between vertex i and j. In the decision policy it is possible to control the influence of the desirability and the heuristics with some parameters.

There are a lot of different ACO algorithms that have been applied to the TSP and according to (8) the ant system variant, which was the first ACO algorithm, failed to find good solutions when the number cities rise above 30. With the introduction of ant colony system this algorithm was able to compete with other optimization algorithms. This is also done in (8) and it is made clear that ACS is very well suited to solve TSP problems when looking at their test results.

In (9) a comparison of different ACO for the set covering problem has been tested. The problem can be defined as a matrix with the columns representing the different sets where the elements are the rows. If an element is in a set a 1 is on that place in the matrix, otherwise a 0. By choosing columns the rows with a one are marked as covered. When a number of columns and all the rows have been covered a solution is found. For ACO the pheromone value is placed on the columns and is the desirability to choose that set. The heuristic function is how many uncovered elements a set contain.

The algorithm starts up with an empty solution and ads set after set using the pheromone and the heuristic function to decide what set to choose next. Also for this problem the ACS has been found to create the best solutions.

# Theory

In this chapter the two main subjects of the thesis, the edge matching puzzle and the ant colony optimization algorithms, are reviewed. This is important knowledge which is used in the next chapter when the ACO algorithm is applied to the edge matching puzzle.

## 3.1 Edge matching puzzle

The Edge Matching Puzzle is an optimization problem proven to be NP-complete by Erik D. Demaine & Martin L. Demaine(10). Because of this it is interesting to analyze the problem in order to determine what separates a puzzle that is easy to solve to one being very hard to solve. The hardness of a problem is determined by the probability to find a solution. Even though a problem has very few solutions it is not necessarily a harder problem than one that has a lot of solutions.

This section will focus on how the puzzle is built up, having a number of different constraints and properties, and how much each constraint and property of the puzzle contributes to the hardness of the problem.

### 3.1.1 Matching sides

A constraint that is present in all edge matching puzzles is the condition saying that the touching sides of two adjacent pieces must have the same value for all pieces to reveal a complete solution. The value is often a color but it can in general be anything from letters to images as long as they are comparable. This constraint can in itself be split into two constraints, with signed and unsigned matching sides. The unsigned case is where the value of the touching sides of two adjacent pieces is exactly the same. In the signed case a new property comes in play which states that any sides must be paired up with its predefined mate. In a puzzle where the values are colors two mates could be a light and a dark version of the same color. In the version of a puzzle with

letters the mates could be a capital and a noncapital letter. Signed sides tend to create less possible solutions to the puzzle but at the same time make it easier to solve via a backtracking method since less matches must be tried out. To outline this imagine n pieces all having one side value in common. In the unsigned version all combinations must be tried out, which implies that all pieces can be matched together. The following equation returns in how many different ways n pieces with equal side values can be matched.

$$total\ matches\ (unsigned) = \frac{(n-1)\cdot n}{2}$$

The equation creates all combinations of matches of the pieces and removes double pairs by dividing by two.

In the signed version where n sides match with m mating sides the equation is very simple.

$$total\ matches\ (signed) = n \cdot m$$

Since the n sides will never match each other there is no need to contract one and divide by two, as in the above equation, to find all the matches.

| Total Pieces | Total combinations (unsigned) | Total combinations (signed) case 1 (n,m) | Total combinations (signed) case 2 (n,m) |
|---|---|---|---|
| **2** | 1 | (1,1) 1 | - |
| **4** | 6 | (2,2) 4 | (1,3) 3 |
| **6** | 18 | (3,3) 9 | (2,4) 8 |
| **10** | 45 | (5,5) 25 | (8,2) 16 |
| **11** | 55 | (5,6) 30 | (3,8) 24 |

**Table 1 – Piece matches – signed/unsigned**

In Table 1 it is visualized how the unsigned pieces create more matches than the signed. The signed pieces can be split into several cases and in the table is shown two different splits. In case 1 the pieces are as close as possible to an even split which gives the most combinations. In case 2 a random split is made just to show that the total combinations always will be less than an even split. The hardness of the problem is not directly affected whether the sides are signed or not. By signing pieces you just get an extra constraint which either leads to an easier solvable problem or harder since it actually is more dependent on the size of the board which can be seen in a later subsection.

The total number of sides that must match before the puzzle is solved completely is given by the formula

$$maximum\ score = n \cdot (m-1) + m \cdot (n-1)$$

This is called the maximum score and is used to compare a found result to see if it is a completely solved puzzle.

### 3.1.2 Border

A constraint that is optional for the puzzle is whether or not the border pieces of the puzzle should be of a special value. This means that for the puzzle to be correctly assembled the entire puzzle should have a common border value.

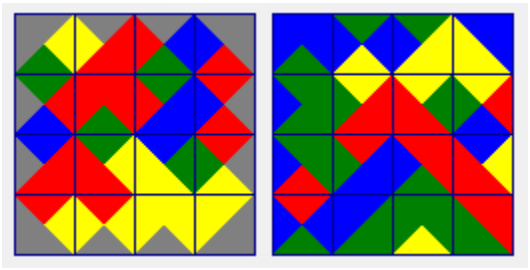


**Figure 1 – A puzzle with and without border**

A puzzle with a fixed border value decreases the number of solutions notably comparing to a puzzle without the border value. It is pretty obvious since in a puzzle with no border constraint, all pieces can be matched with the border. As long as all the pieces not in the border match one another it does not matter how the border pieces are placed.

### 3.1.3 Board size

The board size is the property with the most influence of the total combinations that can be made. The bigger the board get the more combinations can be assembled with the pieces. With a board of size $n \cdot m$ there are $(n \cdot m)!$ possible ways to place the pieces on the board if the pieces do not rotate and without the border constraint. The rotation of the pieces is the property that has the most influence of the number of combinations. In a $n \cdot m$ puzzle

where the pieces cannot be moved around but only rotated, exponentially many combinations can be made. Each time one piece rotates the rest of the $n \cdot m - 1$ pieces can also rotated in four different ways. The puzzle will have $4^{n \cdot m}$ combinations just by rotating. By combining the move and rotation of the pieces the final equation will be

$$total\ combinations = (n \cdot m)! \cdot 4^{n \cdot m}$$

Taking the border constraint into account there will be a lot less combinations. The corner pieces can only be placed in four different locations and can only be in one rotation giving $4!$ combinations. The border pieces can only be placed in $(m-2) \cdot 2 + (n-2) \cdot 2$ different locations and can also only be in one rotation this gives $((m-2) \cdot 2 + (n-2) \cdot 2)!$ combinations. The final equation will be

$$total\ combinations =$$

$$4! \cdot \big((m-2) \cdot 2 + (n-2) \cdot 2\big)! \cdot ((m-2) \cdot (n-2))! \cdot 4^{(m-2) \cdot (n-2)}$$

Using this formula it can be determined that two puzzles of the same size, $n_1 \cdot m_1 = n_2 \cdot m_2$ , gives the most total combinations the more equal n and m are. So to create the hardest board it should be as close to square as possible.

### 3.1.4 Number of colors

As a simplification the side values of the pieces will from now on be referred to as colors. How the number of colors contributes to hardness of the problem is, like signed and unsigned pieces, highly dependent on the size of the board and how evenly the colors are distributed. It is easy to see that with few colors, distributed on a number of pieces, more matches between the individual colors can be done than with many colors, distributed over the same amount of pieces. With a few colors there is a higher probability that more solutions exists than with a lot of colors. Nevertheless it is not necessarily easier to find one out of many solutions to a problem than it is to find a solution to a problem that only has that single solution.

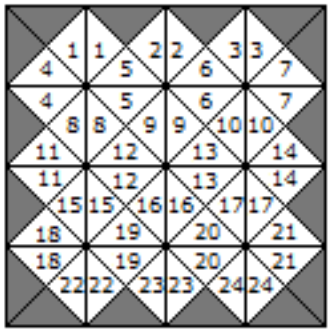



**Figure 2 – 4x4 board with 24 colors**

Figure 2 shows a board of size 4 x 4 with 24 different colors and a border. It is easy to see that only one solution exists (unless the whole board is rotated which will create four solutions). Even though there is only one solution it is extremely easy to find it. Since every color is only found on two pieces those two have to be placed next to each other. So no matter what piece is chosen as a start piece, the rest of the pieces can be placed trivially.

Having unique pieces in a puzzle is preferable if you don't want to see redundant solutions. A formula to create unique pieces from a number of colors can be found in (11)

$$total\ unique\ pieces = \frac{c^4 + c^2 + 2 \cdot c}{4}$$

Here c is the number of colors. It can be seen that the number of unique pieces rises quite fast the more colors are added to the puzzle.

In a puzzle with border the colors in the border pieces that are adjacent to other border pieces will never pair up with the colors of the interior pieces. See Figure 3 for clarification. So even though the same color exists in both the border pieces and the interior it can be seen as two different colors that will never pair up.

**Figure 3 – Illegal placement of border and interior piece**

If there are some colors in a puzzle that occurs less than others it is wise to combine these before the other colors since they create less permutations. This concludes that en even distribution of the colors maximizes the hardness of the problem. The even distribution is of cause for both the border colors and the interior colors.

The interesting question is to locate the ratio of the board size versus the number of colors which creates the hardest puzzles. In (12) a formula for this is listed when figuring out the hardness of the Eternity II puzzle. The formula is trying to prove that there is only one expected solution to the problem by finding the number of colors needed for this to be true. The only information the formula uses is the size of the board and from this determines the number of colors. Here is how the formula is written in (12).

Let M = Interior edge types.
Let B = Border edge types

On average there are 2 joins per interior piece.

Interior solutions = $195! \cdot 4^{195} / M^{2 \cdot 196} = 1$

$195! \cdot 4^{195} / M^{392} = 1$

$M = (195! \cdot 4^{195})^{1/392}$

$M = 16.85$

Interior edge types = 17

On average there is 1 border edge type join per border piece.
On average there is 0.5 interior edge type joins per border piece.

Border solutions = $56! \cdot 4! / B^{60} \cdot M^{0.5 \cdot 56} = 1$

$56! \cdot 4! / B^{60} \cdot 17^{28} = 1$

$B = \left(56! \cdot 4! / 17^{28}\right)^{1/60}$

$B = 4.97$

Border edge types = 5

This formula is written for the Eternity II puzzle but can easily be rewritten to fit with any problem. Here is an explanation of how the formula should be interpreted. The statements that there are 2 joins per interior piece, 1 border

edge type join per border piece and 0.5 interior edge type joins per border piece can be visualized in Figure 4 by respectively green, blue and orange arrows.

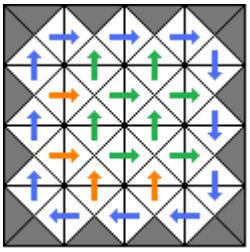


**Figure 4 – Joins of pieces**

In the first equation, to calculate the number of colors for the interior pieces, the numerator is all the possible combinations of the interior pieces. This is similar to the equation seen in the board size section. The reason why it is 195 when it should be 196 is because in the Eternity II puzzle one position of a piece is known beforehand. The denominator is the yet unknown number of colors spread out on the 196 pieces times 2 for the joins. The general equation will be

$$M = \left(((n-2)\cdot(m-2))!\cdot 4^{(n-2)\cdot(m-2)}\right)^{1/2\cdot(n-2)\cdot(m-2)}$$

The second equation, to calculate the number of colors for the border pieces, the same procedure is used as the first equation. The general equation will be

$$B = \left((2(n-2)+2(m-2))!\cdot 4!/M^{0.5\cdot(2(n-2)+2(m-2))}\right)^{1/(2(n-1)+2(m-1))}$$

Using this formula to get the number of colors and distributing them evenly will create some hard puzzles to solve.

## 3.2 ACO

Ant colony optimization algorithms where first introduced in 1991 by Dorigo (13). He suggested the so called Ant System algorithm which is the simplest

of its kind. Like all versions of ant colony optimization algorithms this algorithm was based on the behavior of real world ants that are laying a trail of pheromone to find a shortest path to a source of food. Whenever an ant travels from one point to another it deposits an amount of pheromone on the trail causing other ants to prefer this route. A number of ants choose different routes, some of them overlapping, causing a stronger pheromone value on those parts of the route. The pheromone on the routes evaporates over time causing the shortest route to be used more frequently and at last leaving the longest route unused.

ACO algorithms add additional features extending the real world ants properties. Any heuristic information in a problem is known and used when deciding the next move. The algorithm can also keep track of states the ants have already visited to prevent them from going twice.

The algorithms are intended to work on graph problems and in most cases a problem can be seen as such. In some cases it is fairly easy to make the conversion where in other cases the conversion is hard to find. A problem can even be converted in several ways and it is up to the designer to pinpoint the pros and cons of each to decide which one is best.

The algorithms works by assigning a number of ants to random vertices in the problem graph. From here each ant has to decide which vertex to move on to next. The decision policy is one of the things that vary in between the different algorithms but in general the decision is based on the heuristic information combined with the pheromone value from previous ant tours. A tour is when the individual ants have travelled through the graph and some end criteria is fulfilled. This is at the same time a solution to the problem which may or may not be accepted based on some criteria. When the ants have ended their tour it is time to update the pheromone value on the edges. This also differs throughout the algorithms but in basic the pheromone on the edges will be updated in two steps. First the pheromone will evaporate on all edges with pheromone on them. Then the travelled edges will gain some pheromone according to how good the solution was.

According to (14) three ACO algorithms have shown to be the most successful ones and the details of each will be listed here.

### 3.2.1 Ant system (AS)

As mentioned, Ant System was the first ACO algorithm introduced by Dorigo in 1991, which became the inspiration for later algorithms. At initialization of the algorithm a predefined number of ants are placed on random vertices in the graph. Each ant moves to a neighboring vertex decided by the decision policy defined in this way:

$$p_{i,j} = \begin{cases} \frac{(\tau_{i,j})^{\alpha}(\eta_{i,j})^{\beta}}{\sum_{l \in \theta_i}(\tau_{i,l})^{\alpha}(\eta_{i,l})^{\beta}}, & if\ j \in \theta_i \\ 0, & otherwise \end{cases}$$

- $p_{i,j}$ is the probability for an ant in vertex i to move to vertex j.
- $\tau_{i,j}$ is the pheromone already deposited on the edge between vertex i and j also known as the desirability.
- $\eta_{i,j}$ is the heuristic information on the edge between vertex i and j. This is usually some static information that can be computed in the initialization of the algorithm but in some cases it is computed during runtime.
- $\theta_i$ is a list containing the allowed vertices that can be visited from vertex i. This list is dependent on the specific problem that is tried solved. If a vertex is not in the list the probability of choosing that is zero.
- $\alpha$ and $\beta$ are the parameters for controlling the influence of the pheromone and the desirability. Setting these values is up to the designer and has a huge impact on the solution. It is often by systematic testing the optimal values are found.

When all the ants have completed their tours it is time to update the pheromone values on the edges. First the value will get decreased and then for all the edges that were traversed by the ants will gain a pheromone value according to how good a solution the individual ant created. The pheromone update equation looks like this and will apply to all edges in the construction graph:

$$\tau_{i,j} = (1-\rho)\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}$$

- $\rho$ is the evaporation rate which lowers the existing pheromone value on the edge.
- $m$ is the total number of ants in the system.
- $\Delta\tau_{i,j}^{k}$ is the pheromone deposited by the k-th ant if it has visited the edge between vertex i and j. If so the amount deposited is determined of how good the total solution of the k-th ant's tour is. This is done for all m ants in the system.

At initialization the value $\tau_0$ needs to be set or the algorithm will not work. For Ant System it is often set to a very low value so that it does not influence the choice that much but still has a chance to be chosen.

The Ant System provided a new technique for optimizing solutions to hard computational problems but was still not able to compete against similar algorithms as stated in (15). This leads to the successor of Ant System.

### 3.2.2 Ant colony system (ACS)

Ant Colony System is the first improved ACO algorithm introduced in 1997 by Dorigo and Gambardella (13). The algorithm has not changed a lot from Ant System but has nevertheless a great impact on the algorithm. The decision policy has changed into a pseudo-random proportional rule, making the decision dependant of a random variable, q, uniformly distributed over [0…1] deciding whether an ant should act greedy and follow the edge that has the best value or if it should use the decision policy from AS.

$$s = \begin{cases} \arg\max_{u \in \theta_r} \left\{ (\tau_{r,u})^{\alpha} \cdot (\eta_{r,u})^{\beta} \right\}, & if\ q \le q_0 \\ AS\ decision\ policy, & otherwise \end{cases}$$

$q_0$ is a parameter set by the designer and the higher this gets the greater is the probability that the greedy choice is taken. Since this decision favors exploitation of a known good solution it is needed to decrease some of the pheromone on the edges of the good solutions so that unexplored edges will be travelled

to find new solutions. This is done with the so called local pheromone update. It updates the last travelled edges by all the ants with the equation:

$$\tau_{i,j} = (1 - \varphi)\tau_{i,j} + \varphi\tau_0$$

$\varphi$ is the pheromone decay coefficient that in principle provides the same as the evaporation rate by lowering the pheromone value on the edge. $\tau_0$ is the initial pheromone value as in Ant System. This equation diversifies the solutions found by the individual ants in a single tour.

As in Ant System a global pheromone update is performed when all ants have completed their tours but in Ant Colony System only the ant that created the best solution gets to update the edges. The equation is then applied only to these edges:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \rho\Delta\tau_{i,j}^{best}$$

$\Delta\tau_{i,j}^{best}$ is a value defined by how good the solution to the problem is. The designer can either choose to use the best known solution to create this value or the recent solution found by the best ant.

### 3.2.3 Max-Min ant system ($\mathcal{MM}$AS)

The Max-Min ant system is another improvement to the original algorithm proposed in 2000 by Stützle and Hoos (13). As with the ant colony system it is only the only which created the best solution that gets to update the edges. This is done in the same way as in ant colony system but as the name of the algorithm suggests there are upper and lower bounds of the pheromone values. Two constants representing theses values are set and if the pheromone values either increases or decreases one of these values it is set to be equal to it.

$$if\ \tau_{i,j} < \tau_{min}\ then\ \tau_{i,j} = \tau_{min}$$

$$if\ \tau_{i,j} > \tau_{max}\ then\ \tau_{i,j} = \tau_{max}$$

The minimum value is often experimentally chosen but a good approximation is to use $\tau_0$ from ant system. The maximum value is usually set using the optimal solution of a tour. If the optimal solution is not known beforehand, it can be set to a best-tour solution.

The decision policy is directly taken from ant system.

CHAPTER 4

# An ACO algorithm for the Edge Matching Puzzle

Now that the Edge Matching Puzzle and the principles behind ACO algorithms have been covered it is time to look into the creation of an algorithm to optimize the Edge Matching Puzzle problem.

## 4.1 Introductory algorithms

Since an Edge Matching Puzzle can be constructed in many difficulty levels the first algorithm that comes to mind, dealing with a rather easy problem, is the backtracking algorithm. Problems of smaller size will be solved quickly with this method but since the search area grows exponentially as the board size grows this method will at some point not be usable. The backtracking algorithm is able to find all solutions to a problem and will do so given enough time. In smaller problems this can give an idea of how many solutions a problem can have given different properties like the size of the board, the number of colors and the distribution of these. The algorithm will be very simple by going through all the spaces on the board, trying out all pieces in all combinations. It is easy to see how the search area will grow just by adding an extra row to a board.

The next algorithm which is interesting to apply to the Edge Matching Puzzle problem is the Greedy algorithm. Each time a piece must be placed the one that fits in a position that will maximize the partial score is chosen. This is done until all pieces are placed. Using this approach an optimal solution is not guaranteed even if you try all permutations of the piece list. It is consistent with the general definition that an optimal solution of some problems cannot always be found with a greedy algorithm.

Another way to create a custom greedy algorithm is to place them row or column wise and them pick the best piece available each time a piece needs to be placed. The board will most likely have a lot of pieces that match on one side

and less pieces that match on the other side since the selection of available pieces is cut down each time a piece is placed. This algorithm will though eventually give an optimal solution if all permutations of the order of the piece list are tried. Eventually a list will come up where the first piece in the list is the perfect match for each position.

The last algorithm of interest is the local search algorithm. The idea is actually the same as in(7) where the mutation operators should be either a swap between two pieces or rotation of one piece. The algorithm should also be k-opt where the k is chosen from a Poisson distribution. At initialization of the board is should be run once with the greedy algorithm because local search algorithms usually perform a lot better when working from a solution which is somewhat good.

## 4.2 The ACO for the Edge Matching Puzzle

Each version of the ant colony optimization algorithms all takes a problem represented as a graph as an input. With the edge matching puzzle problem it is of great influence how this graph is made. One way to make it is to create a vertex for each position on the board and a vertex for each piece. In the case where the problem does not requires a border all pieces can fit in all positions on the board. This means that there will be an edge from all piece vertices to all position vertices. Since all pieces can be rotated in four positions each piece vertex is split into four and they all get an edge to each of the positions. A board with $N$ positions and therefore also $N$ pieces will have a graph with $4 \cdot N^2$ edges. In a problem with borders the amount of edges will be a little less but nothing that has a big influence with a large board. Depending on the algorithm the visited edges will gain an amount of pheromone when a solution has been obtained. The amount of pheromone on each edge will be the desirability to choose a piece in a specific rotated state and place it on the position it points to. In a puzzle with borders it is a requirement that border pieces only can be placed in border positions and only in one rotated state and for the corner pieces the same applies. The interior pieces cannot be placed in the border but they can be rotated in all four states. See Figure 5

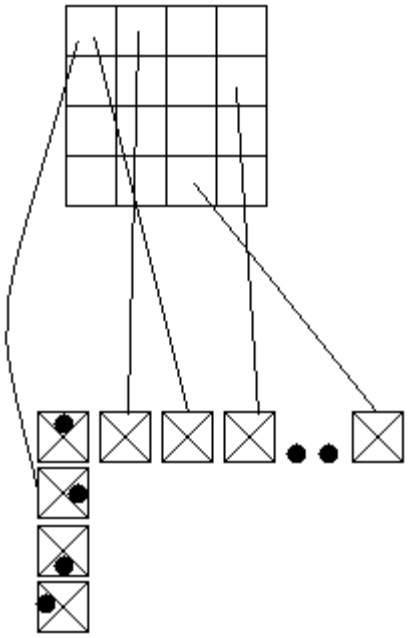


**Figure 5 – ACO version 1 graph representation**

Another way to construct the graph is by making edges between all sides of all pieces, except the sides of the individual pieces. All sides of a piece will have a unique vertex pointing to the other pieces vertices. In this way it is no longer the position of a piece on the board that matters but instead the neighbors of the individual pieces. A board with $N$ pieces will consist of $N \cdot 4$ vertices and $N \cdot (N-1) \cdot 4$ edges where all vertices are connected, except the ones that belong to the same piece. In a problem with borders there are two types of vertices; the ones in the border which only has edges to other vertices lying in the border, and the ones in the inside part of the board. The corner pieces will only have two vertices that are border specific. The corner pieces also have two border specific vertices but also one vertex which is inside specific. The pheromone on the edges will in this case be the desirability to choose a pair of pieces that has been adjacent to each other before. See Figure 6. It has been simplified a lot since a whole board where all sides of all pieces are connected with edges would create very confusing image.

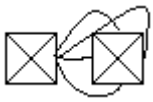


**Figure 6 – ACO version 2 graph representation**

The algorithm needs knowledge about the heuristic function in order to choose a piece to be placed on the board. This can be determined quite easy. When the algorithm wants the heuristics of a piece in a certain position it needs to looks at the four adjacent positions. If some of them have not been filled out with a piece yet they are skipped otherwise the value of the side pointing towards that certain position is noted. The total number of adjacent positions that have pieces placed is noted. For each available piece, in all rotation states, the number of sides that match is stored as $m_{adj}$. This is in the range [0…4] but can never be higher than four. The heuristic function will then return the value

$$\mathrm{h} = m_{adj} + 1$$

The reason for adding one is to avoid that h will be zero and mess up the decision policy. This value must be computed during runtime of the algorithm since it is not known which pieces are placed at what position at initialization.

Where version 1 of the algorithm gets its pheromone value directly from the edge that connects a piece to a position, version 2 has to get the pheromone from several edges. It must get it from the edge of each pieces which is adjacent to the position and then combine them to find a common value.

When it is time to update the edges the pheromone that should be added is simply the number of pieces in the entire puzzle that match

The performance of each ant colony optimization algorithm is easiest to decide by experimentation. One can though argue that the Ant System is outdated and instead go directly to the question of which of Ant Colony System and Max-Min Ant System that performs best. Even though they work in different ways, they use the same pheromone update function and the same heuristic value.

At initialization of the two algorithms a number of variables have to be set. For both of them the influence of desirability versus heuristics, $\alpha$ and $\beta$, as well as the number of ants and the evaporation rate of the pheromone are im-

portant variables which cannot be determined right away but must be found trough experimentation.

When placing the pieces the order of this can be an important issue. The straight forward way is to start in a corner and placing them either row or column wise. For the first piece placed the heuristic information returns zero since no pieces are adjacent. The next piece gets the heuristic information with information from the previous piece and later on the heuristic information is fetched with information from two adjacent pieces. Inserting pieces this way can faultily favor some of the early inserted pieces because of the lack of heuristic information. This may be prevented if the number of ants is high enough since the probability of them choosing different start pieces is then higher.

Another way to insert the pieces is by doing it completely random. Each time a piece is inserted a random position of the board is chosen and from here the heuristic information and the desirability to choose a piece is fetched. This way can though create bad solutions since it does not use the heuristic information very well while putting the first pieces down and then when it is used the best fit pieces might already be used. A mix of the two piece insertion methods is to choose a random position and place the first piece. Then from this position randomly choose one of the four adjacent positions and do this again until all positions has been filled. In this way the heuristic information is used while placing all pieces except the first. This approach will be tried out on the greedy algorithm first to see if good results can be found.

CHAPTER 5

# Implementation

The algorithms implemented are the ones from the previous chapter and they are all implemented in C#. The whole thing is build up with a graphical user interface to create puzzles and then choose an algorithm to run on the problem.

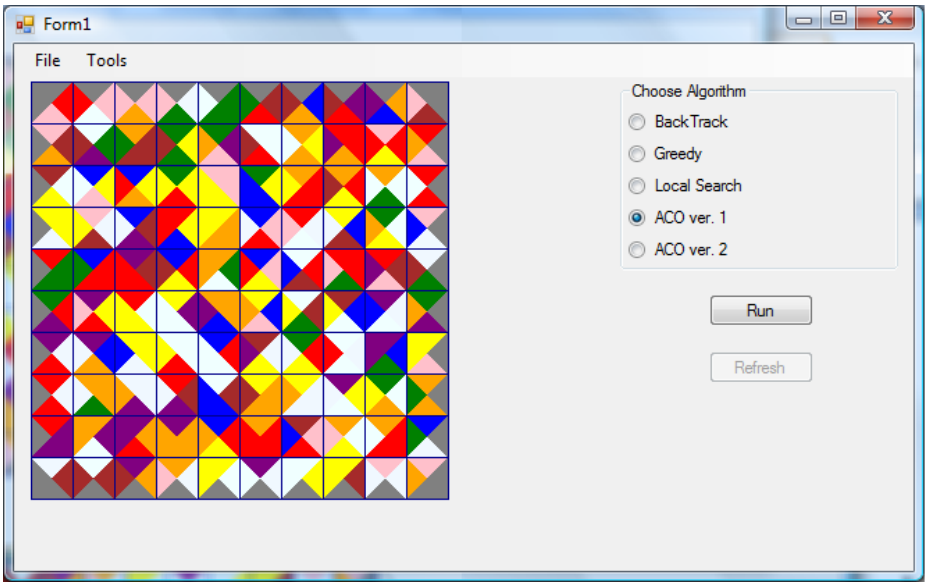



**Figure 7 – The application**

When an algorithm has been chosen and started it will run in a new thread so the program does not block during heavy computation. By doing it this way the algorithm tries to find a better and better solution in the background and with a click on a button the best solution so far is shown. When clicking the refresh button it is important that the algorithm is not in the middle of creating a solution which would cause a wrong board to be displayed since it is not done computing yet. This can be avoided using a semaphore to control when to refresh the solution. A semaphore is a structure that can hold a number of tokens. They can be taken and put back. When there are no tokens to take the

code will have to wait until it can take one. So by taken a token when a solution is being created and putting it back when done, and the same for refreshing, the two code parts will never run simultaneously.

## 5.1 The puzzle

The first thing needed is to be able to create a puzzle. A puzzle is an n x m board consisting of just as many pieces. A puzzle can be created in many ways but in order to guarantee that there is at least one solution the easiest way is to create a perfect puzzle and then shuffling the pieces. One way is to start from a corner and create a piece. The piece's colors should be determined by the total number of colors that has been decided for the puzzle. A random color is assigned to each side of the piece. When placing the next piece the previously placed adjacent piece decides the color of the adjacent side on the new piece. The rest of the sides are again chosen randomly. This is done for each piece and at the end of the puzzle creation a complete puzzle is returned. If a border is required a simple check to see if a piece about to be placed is in the border is performed, and in that case the side not being adjacent to any pieces is set to be the border color.

To prevent that two pieces are exactly the same each of the pieces are compared to each other at the end of the creation. If this is the case the method returns null and is called again. The more colors the greater is the chance of creating a puzzle where all pieces are unique.

To create the hardest possible puzzle just from the size of the board the formula in 3.1.4 is used to find out how many colors should be in the border pieces and the interior pieces. It is computed how many sides needs to be filled for both the border pieces and interior pieces. When this is done the colors are distributed as evenly as possible between each other to create the hardest puzzle. The colors are then assigned randomly to the board while keeping track of how many of each has been placed. The method will end up with a perfect board, which should be one of the hardest instances of its size with expected to have only one solution. This of cause cannot be guaranteed.

## 5.2 The backtracking algorithm

The backtracking algorithm works by taking as input an empty board and a list of pieces that fit on the first position. This includes rotations of the individual piece. For each of the pieces in the list the piece is placed on the board. The same method is recursively called again now with the new board and a revised piece list where the just placed piece has been removed and all other pieces that do not match the placed piece is also removed. This is done either until all pieces have been placed, which mean that a complete solution has been found, or until the method is called with an empty list of piece which means that there were no pieces left that would fit in the position.

The algorithm has the possibility of returning the first found solution and exit or to go through all possible combinations and return all possible solutions to the problem. This will of cause be impossible in large instances of a problem.

## 5.3 The greedy algorithm

The greedy algorithm is called with the problem as a parameter. The greedy algorithm has been implemented with the possibility to place the pieces column wise or by random. For the column wise property each position on the board is accessed using a double for-loop. For each position the pieces that may be placed there, being corner, border or interior piece, is picked out. The algorithm determines the sides of the adjacent positions and then finds the first occurrence of the pieces that match. If such a piece does not exist the next best piece is chosen. This is done for each of the positions until the board is filled out with pieces. The algorithm has been made a little custom since it shuffles the list of pieces before it is about to place them. This will help getting different instances of the board if it is run multiple times.

The random piece placement approach works by first selecting a completely random position on the board and calls a method that inserts a piece. It then randomly moves in one of the four directions by calling the method recursively. If it is not a valid position that is out of the border or a piece has already been placed there it backtracks to a new position. It finds the best matching piece by looking at the neighboring positions to see how many colors that should match. By tests this approach has proven to create very bad solutions and is therefore not implemented in any of the later algorithms.

## 5.4 The local search algorithm

When starting the local search algorithm the board is run though the greedy algorithm once in order to get a somewhat good solution. After that it is time to perform the mutations. To decide how many mutations that should be performed an approximation to the Poisson distributed is used. The probability of flipping a coin k times showing heads is very close to the Poisson distribution. So a random generator just needs to flip a coin as long as it shows heads and when it hits tails this the number of times flipped is k. The built in random generator in C# has a method called nextDouble which returns a value between 0.0 and 1.0. This can be used to simulate the coin by making >0.5 heads and <0.5 tails.

If the puzzle has a border then the swap mutation may only be allowed for pieces in their own region. This means interior pieces can only switch with other interior pieces, border pieces only with other border pieces and corner pieces only with other corner pieces. And of cause it only makes sense to use the rotate mutation on the interior pieces. After the mutations the boards score is calculated and if it is better than the previous, then this will become the new board.

## 5.5 The ACO version 1

For both versions of the ACO algorithm the essential part of the implementation is how the edges are created. For version 1 there should be an edge between every piece, in its four rotated states, to every position on the board. If all of these edges should be created at initialization of the algorithm it would be an awful lot of edges. Even a small puzzle of 6x6 would create $4 \cdot (6 \cdot 6)^2 = 5184$ edges. Since it is only N edges that are used at each ant tour it seems pointless to create them all at once. Instead they are created at runtime when they are chosen by the decision policy. In order for this to be possible there must be a pheromone counter for uncreated edges such that when they are created the right amount of pheromone is deposited on them.

The algorithm can be called to either use the ACS or MMAS, where the difference simply lies in the different ways the edges should be updated either while the ants are travelling the edges or at the end of an ant tour.

## 5.6 The ACO version 2

Version 2 of the ACO algorithm has a very different structure in the edges. Here all sides of each piece should have edges between one another. This will create even more edges than in version 1, and is therefore also first created when it shall be used.

# Results and evaluation

In this chapter the results of the developed algorithms will be revealed. A number of puzzles of different difficulties is created and run through the algorithms. All the tests have been performed on a Windows Vista 64-bit machine, Intel Quad Core 2.83GHz with 4 GB ram.

## 6.1 The puzzles

The puzzles tested will range from being so easy solvable so that a backtracking algorithm can be used to very hard as the Eternity II puzzle. This means different board sizes and number of colors. The pieces of the puzzles will all be unsigned since the complexity of a signed piece more or less corresponds to just having more colors. In Table 2 the puzzles are listed with the attributes that differs them from one another.

| Name | Board size | Num. colors | Color distribution | Borders |
|---|---|---|---|---|
| **P1** | 4x4 | 4 | Random | Yes |
| **P2** | 4x4 | 6 | Even | Yes |
| **P3** | 4x4 | 8 | Random | Yes |
| **P4** | 6x6 | 4 | Random | Yes |
| **P5** | 6x6 | 9 | Even | Yes |
| **P6** | 6x6 | 12 | Random | Yes |
| **P7** | 10x10 | 10 | Random | Yes |
| **P8** | 10x10 | 10 | Random | No |
| **P9** | 10x10 | 14 | Even | Yes |
| **P10** | 16x16 | 23 | Random | No |
| **P11** | 16x16 | 23 | Even | Yes |
| **EII** | 16x16 | 22 | Even | Yes |
| **P0** | 12x12 | 50 | Random | Yes |

**Table 2 – Selected puzzles**

## 6.2 The backtracking algorithm

To test the formula from 3.1.4, claiming that there is only one expected solution if a puzzle is created with the number of colors given by the size of the board, P1-P6 is made small enough for the backtracking algorithm to find all the solutions. P2 and P5 are the problems that are expected to have only one solution where P1 and P4 with fewer colors than the optimal should create more solutions. P3 and P6 should also create either one or few solutions but these should be easier to find because of the less combinations the pieces have to match to each other.

For the problems P1-P3 the solutions where all found very fast. P1 had four unique solutions. P2 and P3 only had one solution each. The boards P1-P3 are a little too small to conclude anything on the results since the solutions were found so very fast. With P4-P6 better conclusions can be made. With P4 it was very easy to find one solution, but when trying to find all solutions, it took a very long time. After 10 minutes the algorithm was manually stopped and at that moment more than 1500 unique solutions were found. With P5 it took a couple of seconds to find the first solution. It took 15 minutes to find the total of three unique solutions. P6 only had one unique solution and it was found immediately. These results are very close to the expected since a lot of solutions were found to P4, only one solution to P6 and P5 was hard to solve.

This concludes that the formula from 3.1.4 does create hard problem and even though there may be more than one solution, they are by far the hardest to find.

P0 has been created to show how easy it is for the backtracking algorithm to find the solutions to a problem that has too many colors compared to the board size. Even though the board is 12x12 it only takes the algorithm a few seconds to locate all solutions which are four, and by that only one unique since the board can be rotated itself.

## The greedy algorithm

Since the greedy algorithm shuffles the pieces before each run it is only necessary to run it once on each problem and stop it after a certain period of time when no improvements are found. The greedy algorithm is implemented to place the pieces both in the regular way by placing them column wise and by placing them going out from a random point. By placing the pieces using the random point approach the maximum number of matching pieces for P7 is 141/180 and no improvements are found after 15 minutes. The regular way to place the pieces gives much better results of 155/180 in the same amount of time. P8 has no border and thus gives much better results. With the regular piece placement the number of matching pieces is 170/180 and the random piece placement gives 161/180. All results of P7-P11 and EII can be seen in Table 3.

| Problem | Regular piece placement | Random piece placement |
|---|---|---|
| **P7** | 155/180 | 141/180 |
| **P8** | 170/180 | 161/180 |
| **P9** | 161/180 | 145/180 |
| **P10** | 405/480 | 367/480 |
| **P11** | 400/480 | 363/480 |
| **EII** | 408/480 | 369/480 |

**Table 3 – Results of greedy algorithm**

From this table it can clearly be seen that the random piece placement is not that effective as first expected.

## 6.3 The local search algorithm

The local search algorithm is run ten times for each problem, and stopped when no improvements has been found in a while. The average result and the best result can be seen in Table 4 – Results of local search algorithmTable 4.

| Problem | Average score | Best score |
|---|---|---|
| **P7** | 142,4/180 | 146/180 |
| **P8** | 156,7/180 | 159/180 |
| **P9** | 145,8/180 | 149/180 |
| **P10** | 370,4/480 | 380/480 |
| **P11** | 373,9/480 | 386/480 |
| **EII** | 384,3/480 | 388/480 |

**Table 4 – Results of local search algorithm**

The results of the ten runs can be somewhat far from each other. This is due to the fact that the local search sometimes reaches a local optimum fast and from here it is not able to create better solutions.

## 6.4 The ACO algorithm version 1

A lot of parameters can be set on the ACO algorithm and it is very hard to determine what the best settings to get the best result are. The influence of the desirability and the heuristics can be set by the $\alpha$ and $\beta$ values and the precise ratio between them is hard to find, and it does not get easier that the number of ants also is variable. A lot of permutations of the three parameters can be made and for P7 this has been tried out to find a good combination. All the different permutations have been run three times where each approximately makes 1500 computations per ant. The average result is noted.

| ACO type | Number of ants | $\alpha$ | $\beta$ | Result |
|---|---|---|---|---|
| **MMAS** | 2 | 1 | 1 | 75/180 |
| **MMAS** | 5 | 1 | 1 | 76.66/180 |
| **MMAS** | 2 | 2 | 1 | 48.33/180 |
| **MMAS** | 5 | 2 | 1 | 59.33/180 |
| **MMAS** | 2 | 1 | 2 | 85.33/180 |
| **MMAS** | 5 | 1 | 2 | 91.66/180 |

| | | | | |
|---|---|---|---|---|
| **ACS** | 2 | 1 | 1 | 94.33/180 |
| **ACS** | 5 | 1 | 1 | 95/180 |
| **ACS** | 2 | 2 | 1 | 86.66/180 |
| **ACS** | 5 | 2 | 1 | 86.66/180 |
| **ACS** | 2 | 1 | 2 | 95.66/180 |
| **ACS** | 5 | 1 | 2 | 106/180 |

**Table 5 – Finding optimal values ACO version 1**

If the $\beta$ value gets much larger than the $\alpha$ value the algorithm leans more against a greedy algorithm and thus reveals results looking more like the ones from 0. This is not the intended behavior because we really want more information gained from the pheromone values. Increasing both of the influence values does not really give better results since the probability of choosing edges is more or less the same as if both values were decreased. But in general a little higher influence of the heuristic value has shown to create better results.

It can be seen that the number of ants has an influence on the result. The only problem with a high number of ants is that it takes very long time to create a single solution. Five ants is a good choice, not being too low and not taking too long time to create solutions. It can also be seen that ACS produces far better solutions than MMAS. The rest of the problems will therefore use the optimal settings which are ACS with 5 ants, $\alpha$ being one and $\beta$ being two.

| Problem | Average score | Best score |
|---|---|---|
| **P8** | 115/180 | 118/180 |
| **P9** | 104/180 | 109/180 |
| **P10** | 241/480 | 244/480 |
| **P11** | 239/480 | 241/480 |
| **EII** | 248/480 | 252/480 |

**Table 6 – ACO version 1 results**

These results have shown to be not as good as the neither the greedy algorithm nor the local search, but I believe that with even more tweaking of some of the inner parameters, such as how much pheromone should be added to the edges and exactly what the heuristic function should return, the algorithm will be able to return acceptable results. These values have of cause also been experimented with but at the moment the algorithm was created they were found to work best. For the 10x10 boards it seems at the moment that a maximum value is reached at some time which indicates that a wrong solution has been

given too much desirability so that it gets chosen again and again. For the larger boards of 16x16 the computation takes way to long time for the algorithm to reach a good solution. There is a huge amount of edges that must be evaluated and this is simple not very time effective. So with more patience the 16x16 board might create a few more solutions.

## 6.5 The ACO algorithm version 2

The same procedure as with the first version of the algorithm has been used to locate the best values of the parameters to see if there might be a difference in the optimal choices. This is again performed on P7.

| ACO type | Number of ants | $\alpha$ | $\beta$ | Result |
|---|---|---|---|---|
| **MMAS** | 2 | 1 | 1 | 49/180 |
| **MMAS** | 5 | 1 | 1 | 69.33/180 |
| **MMAS** | 2 | 2 | 1 | 42/180 |
| **MMAS** | 5 | 2 | 1 | 45.33/180 |
| **MMAS** | 2 | 1 | 2 | 61/180 |
| **MMAS** | 5 | 1 | 2 | 62.33/180 |
| | | | | |
| **ACS** | 2 | 1 | 1 | 93/180 |
| **ACS** | 5 | 1 | 1 | 94.66/180 |
| **ACS** | 2 | 2 | 1 | 87/180 |
| **ACS** | 5 | 2 | 1 | 95/180 |
| **ACS** | 2 | 1 | 2 | 98/180 |
| **ACS** | 5 | 1 | 2 | 110.33/180 |

**Table 7 – Finding optimal values ACO version 1**

The same trends as with the first version of the algorithm can be seen. The results with MMAS are more or less equal to the ones from the first version but around 10 lower. This is actually a little disappointing since I thought that this version would perform better than the first. Then again it actually it does perform better but only with ACS. Especially it seems that the pheromone value has more influence on the result than in version 1.

Here we saw that the algorithm gave pretty good results with ACS but are still best, as in version 1, with 5 ants, $\alpha$ being one and $\beta$ being two.

| Problem | Average score | Best score |
|---|---|---|
| **P8** | 106/180 | 111/180 |
| **P9** | 106/180 | 109/180 |
| **P10** | 237/480 | 240/480 |
| **P11** | 237/480 | 242/480 |
| **EII** | 243/480 | 246/480 |

**Table 8 – ACO version 2 results**

The results of version 2 seem to be a little better but then again, when working with the large board of 16x16, the algorithm has too much to calculate. Version 2 has much more edges to work with than version 1 so good results requires long computation time.

# Conclusion

The thesis was about designing an algorithm for the edge matching puzzle problem using ant colony optimization algorithms. The means was to analyze the edge matching problem and then come up with a suggestion for an ACO algorithm.

By analyzing the structure of edge matching puzzles I have found the properties that makes one problem harder than another. It is not just the obvious claim that the bigger a board gets the harder it is. This is not the case since very easy problem can be created with large board and either very few or very many colors. I have found the real source of a puzzles hardness which is a combination of the board size along with a specific number of colors. A border on the puzzle increases the hardness even more and with unique pieces the hardness maximizes.

By creating simple solvers for the problem I got an idea of how good solutions I could expect to find with the final algorithm. This also gave a good idea of how an algorithm is used to solve specific problems. When using an existing solving technology a lot of adjustments have to be made for each different problem that it is intended to solve.

The results of my ACO algorithms for the edge matching puzzles are unfortunately not very efficient compared to the known solvers. The main problem lies in the fact that both variants creates way too many edges for the algorithm to work efficiently. With more patience the larger algorithms might return better results. It seemed like the smaller problems at one point got into a local optimum and could not reach better results. This might could have been avoided if I had given more focus to some of the variables of the ACO algorithms. The pheromone value, which is added by the best ant, is simply the score of the problem which at first seemed like a good choice. The research could have gone more into determine this value which might have gotten different results. Instead of just using the score one could have drawn other aspects into this value. This applies for a lot of the variables that can be set for ACO algorithms.

Even though I did not get especially good solutions I think that it will be possible to find good results with ACO algorithms. One should avoid the massive number of edges that I had with my algorithms. On the other hand I think it is hard to imagine an ACO algorithm for the edge matching puzzle which does not require a whole lot of edges. If you somehow omit the rotation for the pieces you get four times as few edges which would help a lot working with the board sizes I have worked with. Then another method must be used to handle the rotation of the pieces.

I think the edge matching puzzle has been very interesting to work with, especially the analysis of the hardness. It has also been interesting to create and implement a working ACO algorithm and see how the nuts and bolts of it work.

# Bibliography

1. **TOMY.** Eternity II. [Online] www.eternityii.com.

2. **Fisher, Paul.** *Computationally Hard Problems.* 2008.

3. **Jens Scharnow, Karsten Tinnefeld,Ingo Wegener.** *Fitness Landscapes Based on Sorting.*

4. **Pierre Schaus, Yves Deville.** *Hybridization of CP and VLNS for Eternity II.* 2008.

5. http://en.wikipedia.org/wiki/Hungarian_algorithm. [Online]

6. **Castello, Beryl.** The Hungarian Algorithm. [Online] http://www.ams.jhu.edu/~castello/362/Handouts/hungarian.pdf.

7. **Jorge Muñoz, German Gutierrez, Araceli Sanchis.** *Evolutionary Genetic Algorithms in a Constraint Satisfaction Problem: Puzzle Eternity II.*

8. **Marco Dorigo, Luca Maria Gambardella.** *Ant colonies for the traveling salesman problem.* 1996.

9. **Lucas Lessing, Irina Dumitrescu, Thomas Stützle.** A Comparison Between ACO Algorithms for the Set Covering Problem.

10. **Erik D. Demaine, Martin L. Demaine.** *Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity.*

11. Squares. [Online] http://www.polyforms.eu/coloredpolygons/sqrindex.html.

12. **Owen, Brendan.** Eternity II. [Online] http://eternityii.mrowen.net/design.html.

13. **Marco Dorigo, Thomas Stützle.** *Ant colony optimization.* 2004.

14. **Dorigo, Marco.** Ant Colony Optimization. [Online] http://www.scholarpedia.org/article/Ant_colony_optimization.

15. **Vittorio Maniezzo, Luca Maria Gambardella, Fabio de Luigi.** Ant Colony Optimization.

16. **Marco Dorigo, Thomas Stützle.** THE ANT COLONY OPTIMIZATION METAHEURISTIC: ALGORITHMS, APPLICATIONS, AND ADVANCES.

17. **Marco Dorigo, Luca Maria Gambardella.** Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. 1996.

18. **Marco Dorigo, Vittorio Maniezzo, Alberto Colorni.** *The Ant System - Optimization by a colony of cooperating agents.* 1996.

19. **Thomas Stützle, Marco Dorigo.** ACO Algorithms for the Quadratic Assignment Problem.