A Heuristic Approach to the Multiagent Pursuit and Evasion Problem in Polygonal Environments.

Bachelor Thesis

by

Felix Blumenberg, Fredrik Båberg, Mats Malmberg
{felixb, fbagberg, matsmalm}@kth.se

under the guidance of

Doctoral Student Johan Thunberg

and

Professor, Ph.D. Xiaoming Hu

Department of Mathematics
Division of Optimization & Systems Theory
Royal Institute of Technology KTH, Sweden
May 2011
Abstract

In this paper heursitic algorithms are developed for the pursuit evasion problem in polygonal enviroments. In this problem, continuous trajectories shall be constructed for a group of pursuers, searching for an evader, in such a way that the evader is guaranteed to be seen at some time during the search. Three fundamentally different heuristic methods are considered: tabu search, genetic algorithms and greedy methods. The result is three heuristic algorithms. Two algorithms are readily implemented in ANSI C, yielding solutions of high quality compared to previous work. The report attains and evaluates statistics on runtime of the algorithms. The algorithms are compared considering the quality and efficiency for a vast amount of randomly generated enviroments.

Key-words: Pursuit and Evasion, Heuristic algorithms, tabu search, greedy methods, genetic algorithms.
Sammanfattning

I detta arbete utvecklas heuristiska algoritmer för ett avsökningsproblem i polygonmiljöer med mobila robotar. Problemet består i att konstruera kontinuerliga banor för en grupp av robotar, sökande efter en inkräktare, så att inkräktaren garanterat blir sedd vid någon tidpunkt under sökningen. Tre i grunden olika heuristiska metoder behandlas för att skapa algoritmer: tabusökning, genetiska algoritmer och giriga metoder. Resultatet är tre algoritmer, varav två har implementerats i ANSI C, som snabbt ger lösningar av hög kvalitet jämfört med tidigare arbete. Statistik på körtider och lösningskvalitet för ett stort antal slumpmässigt genererade områden har tagits fram och utvärderats.

Nyckelord: Pursuit and evasion, sökalgoritmer, Heuristiska algoritmer, tabu, giriga metoder, genetiska algoritmer.
# Contents

1 Introduction ................................................................. 1
   1.1 Objective ................................................................... 1
   1.2 Background ............................................................... 1
      1.2.1 What is optimization ............................................ 1
      1.2.2 P and NP problems .............................................. 1
      1.2.3 A near optimal solution ....................................... 2
      1.2.4 Heuristic methods .............................................. 2
   1.3 Outline of the report ................................................... 2

2 Problem formulation .......................................................... 3
   2.1 The pursuit and evasion problem with multiple pursuers .......... 3
   2.2 Discretized problem .................................................... 3
   2.3 The problem formulation of this report .............................. 4
   2.4 Approach ................................................................. 4

3 Simulation environment ....................................................... 5
   3.1 Map generator .......................................................... 5
   3.2 Network generator ........................................................ 6

4 Methods ........................................................................... 8
   4.1 The genetic algorithm .................................................. 8
      4.1.1 Description of genetic algorithms ............................. 8
      4.1.2 Development process of the genetic algorithm ............. 10
      4.1.3 The genetic algorithm of our problem ....................... 12
      4.1.4 Our implementation of the genetic algorithm .............. 13
   4.2 The greedy algorithm .................................................. 14
      4.2.1 Description of greedy algorithms ............................. 14
      4.2.2 Development process for the greedy algorithm ............. 14
      4.2.3 The greedy algorithm for our problem ....................... 15
      4.2.4 Implementation of the greedy algorithm ..................... 17
   4.3 The tabu search method ............................................... 19
      4.3.1 Description of Tabu search methods .......................... 19
      4.3.2 Development process of the tabu search algorithm ........ 19
      4.3.3 The tabu search algorithm for our problem ................. 22
      4.3.4 Implementation of the tabu search algorithm ............... 23

5 Results ............................................................................. 25
6 Discussion

6.1 The simulation ........................................... 28
6.2 Analysis of the simulations ............................... 28
6.3 Analysis of the algorithms ................................. 29
  6.3.1 Genetic ................................................. 29
  6.3.2 Greedy ................................................ 29
  6.3.3 Tabu .................................................. 30
6.4 Future work .............................................. 30

7 Conclusion .................................................. 31

References ..................................................... 32
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Flowchart of a genetic algorithm</td>
<td>9</td>
</tr>
<tr>
<td>4.2</td>
<td>Flowchart of the implemented genetic algorithm</td>
<td>12</td>
</tr>
<tr>
<td>4.3</td>
<td>Flowchart of greedy algorithm</td>
<td>16</td>
</tr>
<tr>
<td>4.4</td>
<td>Generic flowchart of tabu algorithm</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>Final algorithm flowchart of tabu algorithm</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Examples of environments used</td>
<td>26</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Objective

The aim of this report is to construct heuristic algorithms that solve the pursuit and evasion problem in two dimensional polygonal environments. The original pursuit and evasion problem is fully formulated in chapter two. Informally it can be formulated as "given an environment with static obstacles, and a specific number of pursuers, construct a search strategy for the group of pursuers such that the evader is guaranteed to be seen". In [2] a framework is presented for solving the problem. In this framework a method is provided for finding the optimal solution. In practice this algorithm is not very applicable though, since even for small environments the computational time is considerable. This motivates the introduction of heuristic methods. By sacrificing optimality, our aim is to use heuristic methods to find sufficiently good solutions within a reasonable computational time. These methods have, to the best of the authors' knowledge not been used in this context before.

1.2 Background

1.2.1 What is optimization

The aim of optimization is to find the best available value of an objective function, given a defined domain. The mathematical theory of optimization offers a variety of methods to solve a wide range of problems. To approach these problems and find a solution, it is important to identify the characteristics of the problem considered. Relevant distinctions can be made by studying the problem's complexity. Complexity is strongly related to the computational time. There are many different complexity classes of problems, but two of the most fundamental are the P and NP.

1.2.2 P and NP problems

The distinction between these two reveals the difficulty of our problem. This will only be a brief description, for a more detailed description see [3]. Informally one can say that P are problems that are easy, and NP are problems that are difficult. In NP there is a subclass called NP-complete. NP-complete are the hardest problems in NP [6]. Such a problem is called NP-hard and in NP. NP-hard are problems that are at least as hard as the hardest problems in NP. But such problems need not be in NP [6].
1.2.3 A near optimal solution.

Related work has stated that the problem studied in this report is in fact of the class NP-hard [1]. One consequence of the problem being NP-hard is that one can not construct analytic algorithms that provides an optimal solution within a reasonable amount of time [6]. Thus we are imposed to sacrifice optimality, in order to gain computational efficiency. This sacrifice opens a large spectrum of possible approaches to the problem.

1.2.4 Heuristic methods

Heuristic methods is a branch of methods used in computer science and mathematics. According to Ferland et. al. [4] “A heuristic search method can be seen as a procedure taking advantage of the problem structure in order to identify a good solution within a reasonable amount of computing time.”

In general heuristic methods are not guaranteed to provide optimal solutions. In some specific situations though, it can be proven that the solution of a certain heuristic algorithm is optimal. Despite this concern of optimality, heuristic methods often provide extremely efficient and relevant algorithms. Very often they greatly reduce the computational time needed to find a solution. There are several occasions where a heuristic method is preferred to an analytic method. Examples are:

- When there is no known algorithm for solving a specific problem, a heuristic is the only way to approach a solution.
- An algorithm can sometimes be difficult to implement. Heuristics can often be used instead as they are easier to implement and known to produce good results.
- When the problem is too difficult to solve efficiently and quickly with analytical methods. A heuristic method could overcome that and give an acceptable but not necessarily optimal solution.

The last example corresponds to the problem considered in this report.

1.3 Outline of the report

In Chapter 2 the pursuit and evasion problem is fully explained, and also how we intend to approach the problem to find a solution. Some keywords in the report are presented here, thus it is suggested to browse this chapter even if the reader is familiar with the problem.

In Chapter 4 and Chapter 3 the process of constructing and implementing the algorithms and the simulations is described. Given that the reader is familiar with the problem presented, these chapters can be read separately.

In Chapter 6 the results of the work is evaluated and discussed. To follow this discussion it is not vital to have read about the process on forehand, though it is suggested that the Figures 4.2, 4.3 and 4.5 are viewed and understood before reading this chapter.
Chapter 2

Problem formulation

This report is an extension of the paper “A Boolean Control Network Approach to Pursuit Evasion Problems in Polygonal Environments” [1]. Our main purpose is to use three conceptually different heuristic methods to try to construct algorithms that efficiently solves the problem stated in the section below. We will also try to implement these algorithms in ANSI C to evaluate the quality and efficiency of the algorithms. The efficiency is quantified as the runtime needed for the algorithm to construct a solution. The quality of the solution is quantified in terms of path length. We will also discuss how the efficiency and quality of the solutions depend on the size of the environment and the number of pursuers.

2.1 The pursuit and evasion problem with multiple pursuers

Following the previous work of Johan Thunberg et al. [1], the pursuers and the evader are modelled as points moving in the polygonal free space, $F$. Let $e(\tau)$ denote the position of the evader at time $\tau \geq 0$. It is assumed that $e: [0, \infty) \rightarrow F$ is continuous, and that the evader is able to move arbitrarily fast. The initial position $e(0)$ and path $e$ is not known to the pursuers. At each time instant, $F$ is partitioned into two subsets, the cleared and the contaminated, where the latter might contain the evader and the former might not. Given $N$ pursuers, let $p_i(\tau): [0, \infty) \rightarrow F$ denote the position of the $i$:th pursuer, and $P = \{p_1, ..., p_N\}$ be the motion strategy of the whole group of pursuers. Let $V(q)$ denote the set of all points that are visible from $q \subset F$, i.e., the line segment joining $q$ and any point in $V(q)$ is contained in $F$.

**The Original Problem (Pursuit Evasion)** Given an evader, a set of $N$ pursuers and a polygonal free space $F$, find a solution strategy $P$ such that for every continuous function $e: [0, \infty) \rightarrow F$ there exists a time $\tau$ and an $i$ such that $e(\tau) \subset V(p_i(\tau))$, i.e., the pursuer will always be seen by some evader, regardless of its path.

2.2 Discretized problem

The discretized problem from [1] is considered, in which the pursuit evasion problem is modelled as a Boolean Control Network. In that problem the target is to maximize the number of nodes that are guaranteed not to contain an evader, that is the number of nodes in state 3 (see notations below).

In this report we will use the following notations:

- **Tile**: A region in [1] will be called a tile, which corresponds to a node in the Boolean Control Network.
- **Area**: An area is a set of tiles.
**Interior tiles** All tiles that are part of an area are called interior tiles to that area.

**Path** A path is a sequence of connected nodes in the Boolean Control Network.

**State** A state describes the condition for a node in the Boolean Control Network. State 1 corresponds to a node that contains a pursuer. State 2 to a node that is directly visible for a pursuer, state 3 to a node that is guaranteed not to contain an evader and state 4 to a node that may contain an evader.

**Secured** A secured tile corresponds to a node in state 1, 2 or 3.

**Feasible solution** A feasible solution is a set of paths in the Boolean Control Network such that for every path each tile part of the path has been in state 1 in sequence.

**Complete solution** A complete solution is a feasible solution such that every node in the Boolean Control Network is secured when the trajectories given by the solution is followed.

**Incomplete solution** An incomplete solution is a feasible solution such that there exists nodes in the Boolean Control network which are in state 4 when the trajectories given by the solution is followed.

### 2.3 The problem formulation of this report

Given the discretized pursuit and evasion problem presented in Section 2.2, the aim of this report is to:

- Construct three heuristic algorithms that solve the problem.
- Implement the algorithms and evaluate their efficiency and the quality of the solutions presented.
- Collect data from the results of implemented algorithms’ and discuss whether any new conclusions can be made on how to approach the original problem.

### 2.4 Approach

The approach to the aim of the report given in Section 2.3 is the following:

- Find three relevant heuristic methods for our problem and do an in-depth research on them.
- From the methods chosen, construct three algorithms that solves the problem formulated in Section 2.2.
- Create a simulation environment for the algorithms.
  
  To be able to compare the heuristic methods a simulation environment is to be constructed. It is required that the simulation environment can to construct random feasible environments of specified size, run the algorithms and print the results into a file.
- Implement the algorithms.
- Run the algorithms to collect adequate data.
  
  The collected data for each algorithm is the runtime and solution paths for the pursuers. To be able to compare data, the size of the environment, obstacle density and the number of pursuers is also to be collected.
- Evaluate the data and draw conclusions.

All implementations are to be made in ANSI C [5], due to computational efficiency.
Chapter 3

Simulation environment

In order to attain the data needed for a comparison of the algorithms that were to be developed, it was necessary to construct a testing environment. It was decided to create this environment by the use of two separate parts. One part is called the “Map Generator”. This part creates a map of the environment, tests the feasibility and prints feasible environments to an output file. The other part is called the “Network Generator”. This part reads an environment from a file, creates a graph network to the corresponding map and gives each node in the network its relevant information.

3.1 Map generator

The Map Generator (MG) creates random feasible environments. An environment is feasible if its corresponding graph is simply connected and can be divided into a finite set of tiles regions. Given the desired size and the density (percentage of obstacles per total area) as inputs, the MG creates a square shaped feasible environment with randomly placed obstacles and saves the map in an external file. For simplicity we have chosen to construct environments consisting only of square tiles. We suggest that this does not result in a loss of generality since any feasible environment can be approximated arbitrarily good by a sufficiently fine meshing of squares.

We will now show in a more detailed manner how the MG works. First we present some pseudo-code describing the algorithm and then some in depth comments to the code.

Pseudo-code, Map Generator:

```
input variables:
Size; // Specifies the width and height of the square matrix A.
NumberOfEnv; // Specifies how many feasible environments to create.
Obstacle; // Specifies the number of obstacles in percent, e.g.
number of obstacles per total area of A.

while ( i < NumberOfEnv ){
    A = CreateMatrix(Size); // Create a matrix A of dimensions Size*Size
    PlaceObstacle(A, Obstacle); // Place obstacles in the matrix A
    if (Test(A)=TRUE){ // If the environment is feasible
        fprintf( fileOK, "\n \n");
        i++;
    }else{
        fprintf(file NotOk, "Does not work: \n");
    }
}```
The steps of the pseudocode is described in more detail below:

- First we introduce the needed variables to define how many and what kind of environments to create.
- The algorithm starts off by entering a while loop running until the loops content has created the desired amount of feasible environments.
- CreateMatrix(Size) creates a matrix of dimensions (Size × Size) with every element equal to one, which corresponds to a tiles that a pursuer or evader can be located on.
- PlaceObstacle(A, Obstacle) takes the input matrix A and using a randomizing function rand() places zeros in the matrix. The zeros corresponds to obstacles, e.g. tiles that can not be seen through and is not possible for a pursuer or an evader to be located on.
- The function Test (A) tests if all the elements containing a 1 in the given matrix A can be connected. If A is connected the environment is feasible and the function returns TRUE, if not it returns FALSE. First Test() finds the first element in A equal to one, starting from the upper left corner going to the right. Then it performs a breadth first algorithm to test if all tiles can be found from the starting point. If so, the environment is connected, and thus feasible.

3.2 Network generator

The Network Generator (NG) generates a node network from an environment matrix. Each node contains data of its adjacent nodes, all the nodes visible from it and its current state. The input to NG is an environment matrix.

_Pseudo-code, Network Generator:_

1. A = environmentFromFile();
2. Node B = createNodeMatrix();
3. for(Node N in B):
   3.1. setName(); // Set name to the row and column for N in B.
   3.2. setMove();
   3.3. setVision();

We will now describe the steps and the functions in the pseudo code above in more detail:

- environmentFromFile() sets A to an environment matrix, consisting of zeros and ones, which is read from an input file. The matrix in the file could either be generated by the MG or constructed by hand.
- A Node is a data structure that contains a name, pointers to all adjacent Nodes and pointers to all Nodes that can be seen by the actual Node.
- createNodeMatrix() sets B to a matrix of the same dimensions as A, where each element is of the data type Node.
- setMove() creates pointers from the current node N to all feasible vertically and horizontally adjacent nodes.
• `setVision()` creates a list of pointers to all visible nodes in B from the current node N. A node B is visible to N if both B and N can be contained inside a rectangle that does not contain non-feasible tile.
Chapter 4

Methods

In this report it was decided to use the heuristic concepts from greedy methods, tabu search and genetic algorithms to construct the algorithms. These methods have been intentionally chosen so that each method strongly differs in its characteristics from the other two. Greedy methods are local and deterministic in their approach. Both tabu search and genetic algorithms are stochastic and global in their approach, but they differ significantly in how they examine and construct feasible solutions. The reason for this decision was to examine if some conclusion could be made about if any specific characteristics would be favourable for solving our problem.

4.1 The genetic algorithm

4.1.1 Description of genetic algorithms

Genetic algorithms, or GA’s, are based on the idea of evolution, that the most suited individuals tend to live longer and reproduce. A population is simulated in an artificial world using a combination of reproduction, gene crossover and mutation, with a given goal to achieve. The population consists of individuals, each representing a feasible solution. Each individual contains two forms of a solution, “the chromosome, which is the raw ‘genetic’ information (genotype) that the GA deals with, and the phenotype which is the expression of the chromosome in the terms of the model...” [7]. A chromosome contains one or more genes, which is a “representation of a single factor value for a control factor” [7]. During each generation in a GA reproduction is performed to create new individuals, called children.

Due to the need to simulate the population and evaluate individuals, often multiple times, GA’s are not well suited for all kind of problems. When there exists an analytical solution it may be better to use that. If however the problem can be simulated both problems without analytical solutions and problem with complicated analytical solution can be handled by GA’s.

There are different ways to implement genetic algorithms, with variations in how each step is performed. This yields many different versions of genetic algorithms. First there are two different kinds of GA’s, steady-state and generational. Steady-state maintains and alters one population by replacing its individuals through reproduction, while generational replaces an old population with a new one. In Figure 4.1 a flowchart of a genetic algorithm is shown. Descriptions of each step follows below.

The first step is to obtain an initial population, either by an existing set of solutions or by creation.

To be able to compare individuals, and later determine if the goal is achieved, the individuals has to
be evaluated. For that purpose a fitness function is used, which is to calculate a score for an individual depending on how well it fulfils the given goal.

![Flowchart of a genetic algorithm.](image)

To make the solutions evolve a reproduction step is performed, in which existing individuals in the population are combined. During this step two individuals are selected and used to create two new individuals. The new individuals are created first by crossover, in which genes are combined from the parents, and then the individual is subject to mutation. The step is thereafter repeated for the population until a desired population size is obtained.

There are different methods to select individuals for reproduction, of which five are Fitness Proportionate Selection, Random Selection, Fit-Fit, Tournament selection and Elite selection.

- In Fitness Proportionate Selection, the probability of choosing a more fit individual is higher than to select a less fit individual. One often used method based on this is Roulette Selection.

- In Random Selection, the probability to be selected is equal for all individuals.

- With Fit-Fit, at each step the two most fit individuals are selected, but the same individual cannot be chosen more than once at each selection procedure.

- Tournament selection chooses a number of individuals stochastically, and then take the best of them.

- In Elite selection the best individual is chosen. This selection procedure should be used in combination with another selection method.

For Crossover there are also different methods, such as n-point crossover and uniform crossover.

- In n-point crossover, the parent’s chromosomes are cut into $n+1$ fragments from which the children receives chromosomes. The $n$ stands for the number of fragments to be created, and the child receives chromosomes alternating between the parents, so that the first child gets the first fragment from the first parent, the second segment from the second parent, the third segment from the first parent and so on. The second child gets the first fragment from the second parent, the second fragment from the first parent, and so on.

- In uniform crossover, “Given two parent chromosomes of length 1, each parent copies 1/2 genes to each child, with the selection of the genes being chosen independently.” [7].

9
Mutation is important in GA’s since it makes sure that solutions that are not possible to find by combining individuals in the initial population can be found. It corresponds to the variance in nature when genes are copied.

When reproduction has been completed, in order to not increase the population size, a replacement is performed. For this there are several methods, such as Weak Parent, Both Parents, Weakest Individual and Random replacement.

- In Weak Parent, a weaker parent is replaced by a stronger child.
- In Both Parents, the children replaces the parents.
- In Weakest Individual, the children replaces the two weakest individuals in the population, if the children are more fit.
- In Random replacement, the children replaces random individuals in the population.

To make sure the algorithm ends at some point, a termination condition has to be used, for instance a maximum number of generations, a limit in fitness sum, a median fitness, best individual or worst individual.

- With Fitness sum, the algorithm will terminate when the sum of the fitness for the population is less than or equal to a specified value.
- With Median fitness, a range for the fitness is specified.
- With best individual, the algorithm will terminate when the minimum fitness drops below a convergence value. This guarantees at least one good solution and decreases the runtime for the algorithm since the entire population does not have to converge to a solution.
- Worst Individual is close to best individual, but as the name suggests the fitness of the worst individual is considered instead of the best.

To adapt genes to computers, an encoding has to be used. According to literature [7] binary encoding is the fastest encoding, where each gene is encoded as 0 and 1, but integer encoding and string encoding can also be used. If binary encoding is used, ‘bit-flip’ can be used for mutation, where a 0 in a gene representation is switched to a 1, or vice versa.

4.1.2 Development process of the genetic algorithm

The first idea for development was to use an existing framework for Genetic Algorithms, such as LibGA [9] or GAUL[10], to save time on programming. Due to the implementation of the libraries, for this application it was determined to be faster to write a new implementation.

After reading relevant parts of the source code from existing libraries and source code that was obtained after contacting Johan Thunberg et. al., for ideas, a first attempt to a program was written. The first version was a boolean control network, where feasible solutions were generated by using a random function to generate numbers, representing moves, between 0 and 4 to indicate if the pursuer were to move left, right, up, down or stand still. For each pursuer, an integer based encoding was used, which was easy to implement and debug compared to binary encoding. An encoding based on pointers to the node in the node network (see Section 3.2) was also evaluated, but no obvious advantages were found. The algorithm was extended to a generational GA instead of steady-state, because of easier implementation and a more intuitive termination condition in number of generations.
At first a simple fitness function with two variables was used, Equation (4.1), in which \( S_4 \) is the number of tiles in state 4 and \( \text{steps} \) is the total number of steps taken to minimize \( S_4 \).

\[
(1 + S_4 + \text{steps})
\]  

Equation (4.1)

A change of selection procedure to Fitness Proportionate Selection made it necessary to switch to a function which was to be maximized, Equation (4.2), where the numerator was used to scale the fitness value closer to positive integer values.

\[
\frac{1000}{(1 + S_4 + \text{steps})}
\]

Equation (4.2)

The final version of the fitness function was similar to Equation (4.2). By using the total number of tiles, and the maximum allowed steps, called MaxSteps, the fitness function was limited to positive integer values, which should be easier for a computer to calculate, compared to the decimal values in Equation (4.2).

\[
\text{Tiles} - S_4 + \text{MaxSteps} - \text{steps}
\]

Equation (4.3)

In comparison Equation (4.2) gives a more fit individual a higher fitness value than Equation (4.3), but using Equation (4.3) was not considered to be a limitation, as a high selection pressure can lead to premature convergence [7].

For selection, Random, Tournament and Fitness Proportionate Selection were considered, all three in combination with Elite Selection. Random selection was considered the easiest to implement and was used to create a first working version of a GA, but it was replaced with Tournament selection as it gives a more fit individual an advantage. In the final version a Fitness Proportionate Selection was used, which gives more fit solutions an even greater advantage and thereby helps the population converge even faster. Elite Selection was used to make sure the two best solutions were not lost between generations.

To make the crossover operation easy to implement, a version of n-point crossover was used. One path from each pursuer was used alternating from each parent.

Due to the integer encoding, and node network representation, mutation by 'bit-flip' was not easily implemented. Instead a gene was selected by a random function, and thereafter a random point of the gene was chosen to be replaced. To make sure the partly modified gene corresponded to a feasible path in the environment the gene was altered from the selected part of the gene until the end of the gene.

To make sure the overall fitness of the population increased, a weak parent replacement was used.

At first a population size of 400 was used, which was what worked best after 10 trial runs at environments of size 5x5, in combination with a limit of 100 generations and 200 allowed steps per pursuer. Mutation frequency was set to 5% after testing different values, which is more than the often used value in literature [8] of 1-2%, but still not very large. The parameters varied, and due to the increasing computation time with increasing population size a maximum of 2000 individuals was used, after trying different times for different population sizes, in combination with a high mutation rate of 75% and a maximum of 800 generations. This was in order to examine as many different feasible solutions as possible when working with environments larger than 5x5 tiles, without having an population size of over 4,000 which at some attempts took more than 10 minutes for a single generation. The final version of the program had a more dynamic population size and step length, see Section 4.1.3.
4.1.3 The genetic algorithm of our problem

The final version of the genetic algorithm had a maximum population of 2000 individuals and 600 steps\(^1\), but during initialization the population size and step length was limited to a lower value if possible. This was done by creating an initial population of 400 individuals, evaluating each and see if any complete solution was found. If so, the maximum step length was limited to the number of steps used in the complete solution, and an additional 100 individuals were added to increase the probability of having a diversity in the population. In Figure 4.2 a flowchart of the implemented version of a genetic algorithm is shown, with more descriptions following.

![Flowchart of the implemented genetic algorithm.](image)

As written in Section 4.1.2 the initial population was generated by a random function, which gave each pursuer a sequence of numbers that represented each step.

Each individual was evaluated using Equation (4.3), and sorted by fitness in decreasing order, which facilitated selection.

The two individuals with the best fitness value were added to the new population, and thereafter the selection process selected individuals for breeding. Fitness proportionate selection was used, in which a random number between 0 and the sum of the fitness of all individuals was generated. The fitness value was thereafter added for each individual, from the most to the least fit, until the number previously generated was reached, and that individual was selected. This was repeated to select a second parent. The crossover copied genes alternating between each parent, so that two different children were created. A mutation step was performed with a probability of 75\%, where a part of a gene was replaced as described in the Section 4.1.2. Finally the best individuals of the parents and children were placed in the new popula-

\(^1\)Depending on the size of the environment the maximum number of steps was set between 50 and 600.
lation, and the reproduction repeated until a new population of the same size as the previous was created.

As a termination criteria the fitness score of up to $99\%^2$ of the individuals in the population were compared. If the individuals compared both were a complete solution and had an equal fitness score, the algorithm was terminated. As $99\%$ is a very high convergence criteria, it was only used for small environments to avoid premature termination. As a second termination criteria the maximum number of generations was set to 800, to make sure the algorithm would terminate even if no complete solution was found.

4.1.4 Our implementation of the genetic algorithm

The algorithm was implemented using ANSI C. Every time a random value was to be obtained the command:

```
((int)((double)rand() / ((double)RAND_MAX + 1)*SCALE_FACTOR))
```

was called, which generates a number in the range $[0, SCALE\_FACTOR)$. For random seed number

```
srand(time(0))
```

was used. No alternative random functions were evaluated during the development.

The population was implemented as an array, which was sorted between each generation using quicksort [11] in descending order after fitness value. Each individual was represented by a struct containing the number of nodes in state 4, the length of a path needed to reach a complete solution, a gene for each pursuer and the fitness score. Each gene represented a path for a pursuer.

To avoid premature convergence to an incomplete solution, a modification to the termination condition was made. For each individual the number of tiles in state 4 was kept, so that even if the population had the same fitness value the algorithm would not terminate unless a complete solution was found.

---

$^2$Values between 50% and 99% was used.
4.2 The greedy algorithm

4.2.1 Description of greedy algorithms

Greedy algorithms can be either iterative or recursive. Any problem that can be solved recursively can also be solved iteratively [21]. For simplicity we will consider the greedy algorithms as being iterative unless stated otherwise. In each iteration the algorithm has a set of possible alternatives on how to push the algorithm towards a solution. A cost function designates a cost to each alternative. At the end of the iteration the algorithm with the best cost, be it maximum or minimum, is chosen as a part of the solution.

There is no general way of determining whether a greedy algorithm provides an optimal solution or not, but there are two very important properties that usually helps to determine if an optimal solution can be provided. These are the greedy-choice property and the optimal substructure property [18]. The greedy choice property states that an global optimal solution can be arrived at by making locally optimal choices. In other words, at each iteration a choice can be made without reconsidering previous iterations. The optimal substructure property states that the problem can be divided into subproblems, which each have an optimal solution. If these two properties are fulfilled, then a solution can be constructed by the optimal subsolutions. Hopefully this is an optimal solution, but if one wants to be certain more rigorous proofs are needed.

4.2.2 Development process for the greedy algorithm

When constructing a greedy algorithm one often starts by finding some part of the final solution and then extend this to find a correct generic question for recursion. As in the examples given in the online lectures given by prof. Sunder Vishwanathan [17] the question and answer yielding a correct algorithm may not be intuitive. In the approach of creating the greedy algorithm for our problem we assume that it is possible to find the best movement strategy by locally finding the best next step until the environment has been secured. So the generic question would informally be “what is the best next step for the pursuer team?”. In the development process for the greedy algorithm there have been a couple of candidates on how to answer this question. At last only one seemed like a good choice, presented in Section 4.2.3.

The first candidate “algorithm 1” was to make the secured area our objective function. In each iteration we consider the next move for each pursuer in the team, and try to move the whole group so that we maximize the secured area (the objective function). We would then have an objective function and constraints that depend on the environment and pursuer positions. The idea was to try and formulate this as a linear programming problem, since there exists a lot of framework for solving linear programming problems. After consideration this naive approach presented several drawbacks. First off, for many environments it is sometimes necessary to let go of secured areas [22]. This could not be allowed by algorithm 1 with a greedy approach. Also there are a lot of cases where the best move is not an increase in secured area, but rather guarding or transportation to strategic positions. This was allowed, but not very likely to be chosen by the algorithm due to the greedy approach. It was thus realized that the algorithm somehow must look further than just the head on approach of targeting the secured area.

The next candidate was an attempt to extend the formulation of algorithm 1 to have dynamic constraints, here on called “algorithm 2”. The idea was to introduce some sort of tactics to the pursuer team, and thus make the group cooperate in a favourable manor. For each pursuer a certain tactic would be given, corresponding to constraints to the optimization problem. In order to formulate these constraints, we needed more information. Here the idea of introducing different areas was first met. The common vision of the pursuer team divides the environment into several subareas, not visible by the team. De-
Depending on the state and geometry of these subareas each pursuer was supposed to either guard, secure or divide a given subarea. This is dynamic information about the environment, thus giving dynamic constraints for each pursuer. In this case it was problematic to find a general formulation of how to choose a tactic, and how to formulate these tactics as constraints for the objective function. Also there were too many special cases and intuition involved for a possible implementation.

For the third candidate “algorithm 3” the idea was to use the extra environmental information about the non visible areas and somehow apply this to a simple greedy cost function. By designating costs to each feasible tile it would be easy to make a greedy choice. By construction each hunter can in general move to at most four tiles or stand still. Thus by giving each of these five alternatives a value that quantifies how good the move is we’ll find the best over all strategy of the team. With this setup the objective function for each iteration is to maximize the sum of the tile-values that each pursuer moves into. So, how does one quantify what a good tile is? The obvious answers such as field of vision and state are, by themselves, insufficient information for a good algorithm. But by using the dynamic information of the areas, resulting from the pursuers combined vision, we can find more parameters to quantify the best move. In order for the pursuer team to spread out and cooperate we designate a unique area for each pursuer to approach. This is done by adding a value to the tiles that give the shortest path for a specific pursuer to approach its designated boundary. Furthermore we add a value to all moveable tiles depending on their unique guarding properties of prioritized areas. The algorithm will now make good tactical decisions, given that the added values are correctly adjusted.

The candidate used as a basis for the final algorithm was algorithm 3. To arrive at the algorithm presented in Section 4.2.3 several examples has been tested manually on random environments. In the manual execution the aim was to delete all human intuition from the algorithm and strictly quantify the valuation of the tiles with simple numbers or questions, suitable for a computer. A more in depth explanation of the algorithm is given in the Section 4.2.3.

4.2.3 The greedy algorithm for our problem

In this section the final greedy algorithm will be explained in detail, step by step. Before the algorithm is started a prefunction is executed to find all static information of the environment. Since these static conditions only need to be evaluated once, and can be evaluated at any time they are not an interesting part of the algorithm itself but will be described in the Section ?? concerning the implementation.

In order to make a greedy choice possible it is necessary to give each tile under consideration a specific cost. At the end of an iteration the algorithm will make a greedy choice by moving each pursuer to an adjacent tile with the highest cost. The cost function will add one or several values $\alpha$, $\beta$, $\gamma$, $\delta$, $\epsilon$ to a tile depending on its geometrical and strategical properties in the current situation. The cost for a specific tile is the sum of all the parameters added to the tile. The values of the parameters $\alpha$, $\beta$, $\gamma$, $\delta$, $\epsilon$ are to be adjusted in the implementation so that the algorithm behaves properly.

The algorithm is written in an iterative way, and an overview is given by the flow chart in Figure 4.3 In each iteration the algorithm finds and executes the best move for each pursuer. Each step in the flowchart will be described below

1. For the first iteration the input is the starting positions of the pursuers. Later when the algorithm is running, the current position of the pursuers will be the input to each iteration.

2. At the beginning of each iteration a decision is made whether to make another iteration or not.
Figure 4.3: Flow chart of greedy algorithm.

An iteration should be executed if there exists tiles in environment that are not secured and if the breaking condition is not met. The breaking condition describes the maximum amount of iterations allowed and are given by the main program running the algorithm, so that if no solution can be found the algorithm will abort in due time.

3. Here we find the total field of view of the pursuers team. This will divide the environment in seen and unseen areas.

4. All areas not visible by the pursuer team are given a priority, a boundary and if possible an extended boundary. A boundary to an area is the set of all tiles that are both adjacent to some interior tile and also seen by some pursuer. The extended boundary to an area is the set of all tiles such that the whole area can be seen from each tile. The priority is determined by the geometric properties of the area and the state of the interior tiles of the area. If the area is secured it is only relevant to guard its boundaries, thus secured areas are not to be designated in any of the preceding steps. Contaminated areas can be categorised in four different types. In descending order of priority they are:

   • Areas with only one boundary tile, where the whole area can be seen from the boundary tile.
   • Areas with several boundaries tiles, where the whole area can be seen from some boundary tile.
   • Areas with only one boundary tile, but who cannot be fully seen from the boundary.
   • Other areas.

5. In this step a table of possible choices for the pursuers is created. As mentioned above the secured areas should not be a part of the table. The extended boundaries are to be used when measuring the shortest path to a boundary for a pursuer. This is because our aim is to see the area, and the extended boundaries usually provides shorter paths for the pursuers. Each row in the table corresponds to an certain area and each column corresponds to a certain pursuer. Every element in the table corresponds to the number of tiles in the shortest path for the given pursuer to a given area's boundary or extended boundary.
6. Given the table created we now want to choose as many elements as there are columns, since the columns corresponds to pursuers. The choice is to be made in such a way that there is at most one element chosen from each row and column and so that the sum of the chosen elements is minimized. Also there is a constraint that the rows corresponding to areas that can be fully seen from their boundaries must be chosen if possible. A chosen element $c_{i,j}$ corresponds to designating the area of row $i$ to the pursuer of column $j$. If there are more pursuers than areas, an area can be designated to more than one pursuer but all areas must be designated to at least one pursuer. This step of the algorithm is a special case of the assignment problem and can be solved for instance by the Hungarian method.

7. Given the designation made in the previous step, for each pursuer there is at least one path of shortest distance to the designated boundary. For each pursuer, add a value $\alpha$ to the tiles being the first step of the paths with the shortest possible distance to the designated boundary.

8. By construction each pursuer has at most five feasible tiles that it can move to. For each one of these tiles, add a value $\gamma$ for every boundary tile to a secured area that is uniquely seen by this pursuer.

9. For each pursuer:
   - Add a value $\beta$ the tile being closest to any contaminated area.
   - For every area where the whole interior can be seen from its boundary. Add a value $\delta$ to all adjacent tiles where this boundary can be seen.
   - Add a value $\epsilon$ to the tiles where the pursuers has the largest field of vision.

10. In the previous steps we have now added at most five values to each tile in the proximity of each pursuer. For each pursuer find the tile with the largest sum and move the pursuer into this tile.

11. When the movement is made, update the states of the environment correspondingly. Save the path taken so far. Return the updated states and the current positions of the pursuers to the start of the algorithm.

4.2.4 Implementation of the greedy algorithm

Even though a ready-to-run implementation was not made, many parts of the implementation were finished. In the prefunction all the needed static information about the environment is evaluated and saved to a struct.

```c
struct Greedy{
    int SolutionPath[];
    struct Node NodeMatrix[][];
    int BreakCondition[];
    HashTable;
}
```

The SolutionPath is an array where index 0 contains the number of pursuers, index 1 contains the iterations made (initially set to zero by the prefunction), and the rest of the indices are the coordinates for each pursuer. The NodeMatrix is the graph created by the simulation environment described in Chapter 3. The break condition is simply an integer to describe the maximum allowed iterations. Since the algorithm needs to know the distance and the path between two tiles and this is saved into a hashtable. To find the shortest path and the distance between two tiles the A-star algorithm is used.
The implementation of the algorithm described in the Section 4.2.3 calls for the usage of a couple of interesting algorithms to solve some of the steps. In step four we want to find the interior of all areas. This was implemented by first taking any tile not part of the pursuers vision. Make a breadth first search to find and mark all the interior tiles of this area. Find a new unmarked and unseen tile and execute another breadth first search. Repeat this until all tiles not visible by the pursuer team are marked. The breadth first algorithm's time complexity is $O(|V| + |E|)$ [19], where $V$ is the number of vertices in the graph and $E$ is the number of edges.

For step six in the algorithm (the assignment) the implementation is designed to exhaust all possible combinations and then pick the combination giving the smallest sum. The combinations are found by usage of a queue (first in, first out). Here one could try to use the Hungarian method [20] instead, but since the size of the table is usually not very large this algorithm seemed like overkill.
4.3 The tabu search method

4.3.1 Description of Tabu search methods

Tabu search is a heuristic optimization method designed to find the global optimum. The algorithm has the ability to climb out of a local minimum, and then explore other parts of the domain. There are several methods on how this ability can be acquired. Tabu search uses a so-called tabu list, intended to make already visited regions in the domain forbidden “tabu”. Tabu search algorithms are based on certain conceptual building blocks and basic ideas [12] [14] [15]. Most of them are easy to understand but widely varying in difficulty to implement. If well implemented they will give you a highly sophisticated search algorithm [16] [13].

The main building block is the tabu list. The tabu list gives the algorithm the ability to render certain regions or moves tabu. The tabus can be created based on the information at hand, both for a given moment and from prior experiences of the search [16]. The essential problem when constructing tabu search algorithms is about to formulate intelligent rules, that will determine if a specific move should be tabu or not.

Another important building block is the tabu overriding mechanism, in literature this is usually named as the aspiration criteria. The aspiration criteria checks whether a certain move should be approved, even if it initially was set to be tabu [16] [13].

With these two building blocks one can construct almost any type of logical structure. This makes the tabu search method highly implementable and adaptive. The tabu search method is stochastic, meaning that the tabu list and aspiration criteria only deals with randomly selected solutions. This means that the current solution is not necessarily a good one. One of the strengths of the tabu search algorithms is that any solution produced tells something about the problem at hand, independent of the solutions’ quality. In practice this means that a bad solution also yields important information. Or, to quote Thomas Edison

“I have not failed 1,000 times. I have successfully discovered 1,000 ways to NOT make a light bulb.”

This insight might seem somewhat naïve at first sight, but if implemented in a good way it yields powerful results [16].

4.3.2 Development process of the tabu search algorithm

The development process of the tabu search algorithm for our problem will here be described in detail. Due to lack of time, the alternatives that were easiest to implement was usually preferred. The development process originated from a general flowchart of the algorithm [16] [13], given in figure 4.4. By following the flow chart, in step one, we arrive at the first issue to resolve. The issue is whether to construct a set of feasible solution, or one single feasible solution at a time. Since the genetic algorithm already had implemented a way of constructing sets of feasible solutions, it was decided to construct one feasible solution at a time.

Two alternatives on how to construct this single feasible solution was considered. Either by generating the feasible solution, one pursuer step at a time, or by a series of steps. It was decided to construct the feasible solution with one pursuer step at a time. This was a hard decision to make, since both alternatives seemed to yield highly promising but very different algorithms. The implementation of making a
series of moves would result in a receding horizon approach, where one had to either:

- Produce a large set of series of moves. Sort these by fitness\(^3\), then choose one to check against the tabu list and aspiration criteria.

- Generate a single series of moves to check against the tabu list and aspiration criteria.

The implementation of this approach would require that the tabu list and aspiration criteria had abilities of a more analysing type. This implied that it would be very difficult to implement within the given time.

This settled, we can now move on to discuss the ideas that form the tabu list. Seven different rules where considered:

\(^3\)Described in 4.1.1
1. For one feasible solution save the past M number of moves for each pursuer and render these moves tabu to be returned to.

2. Do not allow moves that results in N number of lost secured tiles.

3. Make geometrically based tabus for critical areas and tiles:

   A corridor where the end point can be seen should not be walked down by a pursuer. Also, a area with an obstacle that can be circulated would need two pursuers to be secured. Thus it would be tabu to go about and try to secure it alone. And finally if one had a tree like corridor system, going about to solve it would only be allowed with the right amount of pursuers.

4. Work together rule:

   Two pursuers never really need to see each other, only share visible areas. Since this rule is general in its formulation it should be called upon in the aspiration criteria.

5. High valued areas rule:

   Tiles visited in a complete solution should be given a value bonus. Or implemented differently, tiles not visited could be set to be tabu. This is an attempt to incorporate the idea of getting information even from bad, but complete, solutions.

6. Low valued areas rule:

   Some tiles, that are part of an incomplete solution are probable to be of low interest. Hence these tiles could be given a penalty.

7. Treading in secured areas is not probable to yield promising results. Thus this should be given a penalty, or not allowed.

All these tabu rules are to be ranked and given a priority level that would be used in the aspiration criteria. Also many of the tabu rules listed above needed some sort of worst case scenario handling, in order to avoid that the algorithm would freeze.

We will now discuss the aspiration criteria. As understood from the tabu rules presented above a lot is to be incorporated in the aspiration criteria. It needs to have the ability of checking whether a tile is of high or low value. Which in turn means that step 5 in Figure 4.4 needs a fitness calculation function that would depend on the tile visited and the shape of the feasible solution. Unfortunately this aspiration criteria would be difficult to implement. Thus this advanced aspiration criteria was rejected for a simpler one. In the end the more simple aspiration criteria, only came to be a worst case handling, to avoid the algorithm from freezing. This also means that tabu rules (3), (4), (6) and (7) no longer could be implemented. Tabu rule (5) got simplified to: tiles not visited where set to be tabu.

A comment on tabu rule (5) is that when this rule was first implemented, the idea was that X number of past complete solutions were analysed to check which tiles had been walked upon at least once, and render the rest tabu. The problem was however that this wasn’t strict enough, so X was set to one. Which in turn led to a new problem. The problem that arouse was that the algorithm now, under certain circumstances, was too hasty in returning a final solution, meaning that the final solution was of poor quality. This was partially solved by a new stopping criteria and a “go about it again” criteria.

The final version of the algorithm can be viewed in Figure 4.5 and is also presented in Section 4.3.3.
4.3.3 The tabu search algorithm for our problem

In Figure 4.4 and in Section 4.3.2 a general step by step explanation was made of the tabu search algorithm. In Figure 4.5 a modified version can be viewed, intended to illustrate the final algorithm. A step by step explanation will be made of Figure 4.5 below. Observe that some functions from to the general algorithm have been removed.

There are two elements in the algorithm that affect the convergence. These are the tabu rule (5) and the Best-solution-found-so-far criteria. The tabu rule (5) is explained in the previous section. The Best-solution-found-so-far criteria prohibits the algorithm from searching for solutions longer than the best length found so far. It should be noted that both convergence helping elements depend on the existence of a complete solution. This means that if the algorithm is unlucky in finding the first complete solution, the computational time will rise accordingly. As a side note, this considerable insufficiency might have been avoided with the more advanced aspiration criteria, discussed in Section 4.3.2.

The Figure 4.5 will now be described in detail.

1. Obtaining feasible solution by going one step with each pursuer randomly.
   The input of step 1 is the path taken so far. For the first iteration, or if a completely new feasible solution is to be created, this corresponds only to the starting positions of the pursuers.

2. Is it a new best complete solution.
   (a) If yes the complete solution is saved. An update of the tabu list, the aspiration criteria and the stopping criteria is made accordingly. Then move to 6.

3. Is it an incomplete solution.
   (a) If yes update the tabu list, the aspiration criteria and the stopping criteria accordingly. Then move back to 1.

   To note here is the change from the general flowchart in Figure 4.4 to the less sophisticated algorithm.

4. Check the feasible solution against the tabu list, is the feasible solution tabu.
   Meaning check tabu rule (1), (2) and (5).
   (a) if yes the feasible solution is analyzed in the aspiration criteria, whether if the tabu list is to be overrided.
      i. if no, update the tabu list, the aspiration criteria and the stopping criteria accordingly.
         Then move to 6.

5. If yes in step 4 or no in step 4a the tabu list, the aspiration criteria and the stopping criteria accordingly. Move to 1.

6. The stopping criteria. Has it been reached. If no move to 1.
   Here an evaluation is made whether the algorithm is to be set to cycle through iteration or not. Or if it is to restart with a completely new feasible solution or not. Or lastly, if the algorithm is to restart, with a completely new feasible solution but save the information from tabu rule (5) or not.
   The last part here was the incorporation of the “go again criteria”.

7. If yes in 6 terminate the algorithm
4.3.4 Implementation of the tabu search algorithm

In this section two important aspects of the implementation will be discussed. These two are the meaning of the parameter values and obvious shortcomings in the algorithm. With the implementation, a series of parameter values arise. The parameter values can be divided into four different categories.

1. Tabu rule values:
   
   (a) save_past_M_number_of_steps  
   
   (b) N_number_of_lost_secured_tiles  

2. Aspiration criteria values:
   
   (a) Override_N_number_of_lost_secured_tiles

Figure 4.5: Final algorithm flow chart of tabu algorithm.
3. Stopping criteria values:

(a) Max_incomplete_solutions_found_in_a_row
(b) Max_equal_complete_solutions_found_in_a_row
(c) Max_allowed_steps

4. Go again values:

(a) Restart_the_algorithm_P_times
(b) Restart_the_algorithm_Q_times_keep_best_complete_solution
(c) R_steps_is_this_really_the_best_complete_solution

As can be seen in the list above some of the parameters have strong relations to each other. These relations will now be explained.

Parameter values (1) and (2) are related in such a way so that (2) ensures that (1) does not make the algorithm freeze.

Parameter value (4b) and (4c) aims to avoid the algorithm from being too hasty in returning a final solution. This is usually encountered in less difficult environments. The Max_allowed_steps and Restart_the_algorithm_P_times tries to tackle the problem of finding the first complete solution. The computational time, and the probability of finding any solution at all, strongly depends on this relation. As discussed lastly in Section 4.3.2 the algorithms’ computational time will be affected if the first complete solution is found at a late stage in the search. By increasing (3c) the probability of finding any solution should increase. Though this will affect the computational time in a negative way. Both parameter value (3a) and (3b) are to ensure that the algorithm terminates when the search space has been thoroughly examined in promising areas. Parameter (3b) is also an indicator of that the the search has converged to a final solution.

The implementation of algorithms were made piece by piece. Two of these pieces had known obvious shortcomings. Shortcomings that at the time of implementation, were ignored, and due to the lack of time, never were treated. These are

1. A pursuer can’t linger in a tile
   An effect of Tabu rule save_past_M_number_of_moves.

2. Large number of precious semi freezes the algorithm\(^4\).
   Also an effect of Tabu rule save_past_M_number_of_moves.

The belief is that both these shortcomings could have been dealt with.

\(^4\)The number of large has not been investigated but for 20 pursuers things went south
Chapter 5

Results

The algorithms were run on iMac’s with 2.66GHz Intel Core2Duo and 4GB of RAM. Each machine was given environments of a specified size\(^1\), where the density of obstacles was either 25% or 40% of the total area in the environment. Figure 5 shows examples of environments of size 10x10 with (a) 25% obstacle density and (b) 40% density that were used.

For the random environment simulations each machine was given 10 randomly generated environments of equal size. Five starting positions were generated, twice for each environment. To decrease the amount of pursuers the starting positions firstly generated were omitted. Each setup of starting positions, number of pursuers and environment was run with the same conditions four times.

Each algorithm was given a set of parameters that were tuned in order to find a solution with two pursuers, however each environment was run with 5 to 2 pursuers. The algorithms were also given a maximum number of steps allowed for each pursuer to take, set to be long enough to find a solution. The algorithms were then to find a solution of as few steps as possible for a given number of pursuers. If for a given amount of pursuers neither of the algorithms were able to find a solution within the maximum steps allowed, no further attempts of finding a solution with these conditions were made. Neither were any further attempts made for fewer pursuers on these conditions. In table 5.1 the data acquired is summarized.

For the Manhattan\(^2\) grid the simulation conditions were different. A 5x5 and a 9x9 Manhattan grid was used with 5 to 2 pursuers, starting positions was the upper left corner for all pursuers and cases.

In Table 5.1 the “Average step difference” describes the difference in path length of complete solutions. This value is attained by subtracting the sum of the length of all complete solutions of the tabu search algorithm from the sum of the length of all complete solutions of the genetic algorithm and divide this by the number of successful runs for the given environment. The “Average time” is the time for each run divided by the number of runs, where both successful and unsuccessful runs were counted.

\(^1\)Environments of size 5x5, 9x9, 10x10, 15x15 and 20x20 were used

\(^2\)A special symmetric environment, see [1]
<table>
<thead>
<tr>
<th>Size</th>
<th>Pursuers</th>
<th>Runs</th>
<th>Unsolved (Genetic)</th>
<th>Unsolved (Tabu)</th>
<th>Avg steps (Genetic)</th>
<th>Avg steps (Tabu)</th>
<th>Avg step diff (Genetic - tabu)</th>
<th>Avg time Genetic (sec)</th>
<th>Avg time Tabu (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5x5</td>
<td>5</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>1.20</td>
<td>1.20</td>
<td>0.000</td>
<td>0.037</td>
<td>0.003</td>
</tr>
<tr>
<td>5x5</td>
<td>4</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>1.55</td>
<td>1.55</td>
<td>0.000</td>
<td>0.049</td>
<td>0.008</td>
</tr>
<tr>
<td>5x5</td>
<td>3</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>2.31</td>
<td>2.30</td>
<td>0.013</td>
<td>0.084</td>
<td>0.010</td>
</tr>
<tr>
<td>5x5</td>
<td>2</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>3.31</td>
<td>3.31</td>
<td>0.000</td>
<td>0.116</td>
<td>0.012</td>
</tr>
<tr>
<td>10x10</td>
<td>5</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>13.67</td>
<td>11.29</td>
<td>2.375</td>
<td>45.653</td>
<td>18.046</td>
</tr>
<tr>
<td>10x10</td>
<td>4</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>21.75</td>
<td>14.47</td>
<td>7.281</td>
<td>128.613</td>
<td>44.249</td>
</tr>
<tr>
<td>10x10</td>
<td>3</td>
<td>95</td>
<td>1</td>
<td>12</td>
<td>50.39</td>
<td>23.67</td>
<td>16.831</td>
<td>312.450</td>
<td>76.693</td>
</tr>
<tr>
<td>10x10</td>
<td>2</td>
<td>67</td>
<td>14</td>
<td>22</td>
<td>88.38</td>
<td>42.78</td>
<td>45.602</td>
<td>537.163</td>
<td>80.843</td>
</tr>
<tr>
<td>15x15</td>
<td>5</td>
<td>42</td>
<td>10</td>
<td>23</td>
<td>118.06</td>
<td>54.32</td>
<td>54.747</td>
<td>560.978</td>
<td>1676.596</td>
</tr>
<tr>
<td>15x15</td>
<td>4</td>
<td>17</td>
<td>7</td>
<td>8</td>
<td>132.60</td>
<td>61.78</td>
<td>68.429</td>
<td>663.767</td>
<td>1278.498</td>
</tr>
<tr>
<td>15x15</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.000</td>
<td>1004.250</td>
</tr>
<tr>
<td>20x20</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>9521.394</td>
<td>8059.600</td>
</tr>
<tr>
<td>Mtn (5x5)</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4.00</td>
<td>4.50</td>
<td>-500</td>
<td>0.755</td>
<td>0.045</td>
</tr>
<tr>
<td>Mtn (5x5)</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4.00</td>
<td>5.00</td>
<td>-1000</td>
<td>1.073</td>
<td>0.015</td>
</tr>
<tr>
<td>Mtn (5x5)</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4.00</td>
<td>4.50</td>
<td>-500</td>
<td>2.525</td>
<td>0.033</td>
</tr>
<tr>
<td>Mtn (5x5)</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>9.50</td>
<td>12.50</td>
<td>-3000</td>
<td>2.015</td>
<td>2.047</td>
</tr>
<tr>
<td>Mtn (9x9)</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>13.75</td>
<td>12.00</td>
<td>1.750</td>
<td>16.300</td>
<td>47.023</td>
</tr>
<tr>
<td>Mtn (9x9)</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>25.00</td>
<td>22.00</td>
<td>3.000</td>
<td>52.055</td>
<td>137.183</td>
</tr>
<tr>
<td>Mtn (9x9)</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>91.00</td>
<td>-</td>
<td>-</td>
<td>172.825</td>
<td>175.825</td>
</tr>
</tbody>
</table>

Table 5.1: Data from simulations.

(a) 10x10 environments with 25% obstacle density.

(a) 10x10 environments with 40% obstacle density.

Figure 5.1: Examples of environments used.
From the simulations the following observations were made:

- For environments of size 15x15 and larger the amount of RAM used was significant.
- The results were dependent on parameter values.
- For environments of size 5x5 were solved with good quality for both the tabu search and the genetic algorithm, however tabu search were faster in finding a solution.
- Environments of size 10x10 and larger were not always solved with less than 4 pursuers.
- When tabu search found a solution, it was often of better quality than the genetic algorithm.
- For environments of size 10x10 and larger, the genetic algorithm found a solution more often than tabu search.
- For environments of size 10x10 and larger, the quality of the solution, if found, varied in steps and computation time.
Chapter 6

Discussion

This chapter contains analysis of the data collected when running the algorithms and conclusions made during the working process.

6.1 The simulation

The simulation environment developed was usable, but is not without limitations. Some of the advantages and disadvantages are listed below.

The simulation environment can not merge tiles into one node. Since the complexity of many of the algorithms that were used depends on the number of nodes, merging nodes should lead to a decreased computation time. As the memory usage of the graph is proportional to the number of tiles, memory usage would also decrease.

Using a graph means that existing algorithms that are already written for graphs can be used, which facilitates implementation. During implementation it is also possible for the nodes to contain a collection information.

The decision to read environments from files makes the simulation environment usable for many different environments. New environments could easily either be computer generated or added by hand.

6.2 Analysis of the simulations

First off it should be noted that when varying the size of the environment certain parameters of the algorithms had to be adjusted. For environments of intermediate size a correct adjustment of parameters yielded solutions of significantly higher quality. For larger environments it was necessary in order to attain any solution at all. We found no analytical way of adjusting the parameters. A good understanding of the implementation in combination with intuition and testing usually solved the issue. This does not exclude the possibility that there might be a more decisive approach. In fact, based mainly on our intuition, it is most probable.

As seen in the Table 5.1, given in the Chapter 5, for small environments both algorithms efficiently provides solutions of high quality, even with a small amount of pursuers. The optimality of the solutions on the randomly generated environments is not proven, but it’s likely that the provided solutions in most cases are optimal. This assumption is motivated by the fact that the quality of the solutions rarely differs
and the path length is typically only a few steps long. It should be noted that the average computational time for these solutions is 3 milliseconds for the tabu search algorithm and 37 milliseconds for the genetic algorithm. These are highly efficient results compared to related work [1].

When the size of the environment is increased the solutions found are less probable to be optimal. This assumption is motivated by the fact that the quality of the solutions now vary strongly. Also the computational time severely increases with the size of the environment. An interesting observation though is to compare how the average computational time for the two algorithms changes as the difficulty increases. The tabu search algorithm is highly efficient if it is able to find a first complete solution. But when running on the bigger areas the average time diverges. On the other hand the genetic algorithm provides reasonable average computational times, and is more reliable to find some solution even for difficult environments.

By combining an in-depth understanding of the implementation with the data acquired the following conclusions, and fundamental differences, can be made on the implementation of the algorithms. The tabu search algorithm is strongly dependent on finding a complete solution of sufficient quality, in order to be efficient. The reason behind this is that a maximum step length is set dynamically by the complete solution found, and tiles not visited in this solution are rendered tabu. Thus, with a complete solution of bad quality there are still a vast amount of alternative paths to consider, which in turn affects the efficiency.

The genetic algorithm does not have the same dependency of finding a complete solution, due to its reproduction properties. This motivates the more robust results on efficiency for larger areas, compared to the tabu search algorithm.

6.3 Analysis of the algorithms

This section contains a short analysis of each algorithm and evaluation of why it did or did not work.

6.3.1 Genetic

The genetic algorithm ran a vast number of times for different environments, and found solutions for most environments that it was given. The quality of the solutions did however vary, as can be seen in Table 5.1. The reason for the variation in number of required steps compared to the tabu search algorithm is most probably premature convergence. As the crossover operator implemented was simple, with only swapping paths of the pursuers, it was not possible to explore a vast number of feasible solutions. An attempt to decrease the effect of this was made by the increase in rate of mutations.

As the algorithm does not consider the environment, it should not have problem with the shape of the environment. With the implementation of crossover that is used, the algorithm is required to have a high rate of mutations. A large number of pursuers should however be more advantageous for the algorithm, as there are more ways to combine paths of the pursuers.

6.3.2 Greedy

Although the greedy algorithm was not fully implemented it was manually tested on several environments. There are two situations where the algorithm is known not to perform at its best. The first situation is when the environment is extremely symmetrical. If this is the case the cost function will give equal values to several tiles and some pursuers will be given several equal alternatives. There is no handling for
these events and the algorithm will probably freeze. The second situation is when two or more pursuers start at the same tile. This will result in that in every iteration they have the exact same conditions, thus they will make identical decisions and move as one single pursuer until their shared vision divides the environment into enough areas to be designated. When there are enough areas to be designated the pursuers will be forced to split up and go separate ways. This is also a problem due to symmetry in some sense.

6.3.3 Tabu

The tabu search algorithm went through many simulations, and performed relatively well. The most evident strength of the algorithm is tabu rule (5), which effectively reduced the search space when a complete solution is found. This fact also highlights the main weakness of the algorithm, namely that the algorithm is strongly dependent on finding the first complete solution in order to be efficient. Under certain conditions the tabu rules themselves are known to hinder the search for a complete solution. An example of this is when the starting positions are very spread out. If the pursuers are spread out they must meet in order to be able to cooperate. Since the tabu rule (1) prohibits the pursuers from returning to a tile previously visited, the path taken by the pursuers, in order to meet, hinders further cooperative search in these directions. This problem could probably have been avoided by adjusting the parameter save_past_M_number_of_steps. However this was not investigated in detail. There is also a known problem with tabu rule (1) for a certain type of geometry. Suppose that there is closed area with only one entrance, and that this entrance is a corridor. Once a pursuer enters a geometry of this type the tabu rule (1) will hinder the pursuer to exit the area. This in turn inhibits the general cooperation of the pursuer team. This could be one of the reasons why the tabu search algorithm sometimes failed finding a solutions for a small number of pursuers.

6.4 Future work

Improvement on the implementations of the algorithms and the simulation environment is probable to give more reliable and promising results. Due to the limited amount of time given, for a project of this size, many short cuts were made intentionally. Another suggestion for improvement is to combine the algorithms and create a hybrid method.

Since the algorithms constructed in this report are considered to be very efficient for smaller problems it is suggested to investigate whether a global optimal solution could be attained by solving local sub-problems.
Chapter 7

Conclusion

In this report heuristic algorithms were developed for the pursuit and evasion problem. Three fundamentally different heuristic methods were considered: tabu search, genetic algorithms and greedy methods. The result was three algorithms. Two algorithms and a simulation environment was implemented in ANSI C. The implementation of the simulation environment discretized the considered environment into a number of quadratic tiles. The algorithms were run on a vast amount of randomly generated environments. Data on the results of the executions was attained and evaluated. The data implied that the implemented algorithms performed well on small environments considering both the quality of the solution path and the computational efficiency compared to related work. For bigger environments the implemented algorithms are insufficient and an enhancement is needed in order to attain satisfying results.


