Pursuit and Evasion

Bachelor Thesis

by

Felix Blumenberg
870605-0633
Fredrik Båberg
880312-0511
Mats Malmberg
860802-0338

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
Feb 2011

Abstract

Abstract in ENGLISH

Sammanfattning

swedish Abstract pï¿½ SVENSKA

Contents

List of Figures

Chapter 1

Introduction

1.1 Objektiv

The goal of this research is to solve a pursuit evasion problems in a two dimensional environment, with the help of heuristic problem solving methods. In short, the problem is: given a environment with static obstacles and a number of pursuers ensure that an evader cannot exist within the environment. A complete formulation of the problem can be seen in Chapter 2

The problem itself is solved. An algorithm providing an optimal solution already exists. The problem yet to be solved, is the problem of finding a solution within a reasonable computational time, it is here where heuristic methods comes in to play

With great risk for loss of optimality, our goal is to use heuristic methods to find a solution with a reasonable computational time

This report is therefore a study of the balance between a reasonable computational time and the loss of optimality.

1.2 Organization Of the Report

1. Chapter 2 contains a detailed formulation of the problem and are approach to it.
2. Chapter 3 explains the simulation environment we built.
3. Chapter 4 contains a description of the algorithms we decided to use.
4. Chapter 5 contains the results of our simulations.
5. Chapter 6 contains a discussion of our results, comparison and conclusions.

1.3 Related work

Relaterat arbete bla de tre givna artiklarna

1.4 Background

1.4.1 What is optimization

Optimization is a mathematical discipline, for solving various types of problems. With the aim of finding the best, available, values of some objective function given a defined domain. If one wants to know more advise NAME ON LITETUR

Optimization Theory offers a variety of solution method to a wide range of problems. For the solution process of these problems, it is important to identify what kind of problems one deals with. Relevant distinction can be done by studying the problem complexity.

Complexity is more or less computational time. There are many different complexity classes of problems, but two of the most fundamental is/are the P and NP

1.4.2 P & NP problems

The distinction between these two reviles the difficulty of ore problem. I wonât go in to specific detail, concerning sub classes etc, only a brief intuitive description. For a more detailed and interesting reed advise[sisst referens 4]

P is the set of problems which can be solved by a deterministic Turing machine using a polynomial amount of computation time. CobhamâEdmonds thesis holds that P is the class of computational problems which are "efficiently solvable" or "tractable". In practice, some problems not known to be in P have practical solutions, and some that are in P do not, but this is a useful rule of thumb.

NP refere to Nondeterministic Polynomial time. and can be intuitively described as the set of problems for which, a found solution can be verified to actually be the correct solution, using a deterministic Turing machine using a polynomial amount of computation time.

A set of descriptions that leads one to wonder whether

$$P = NP \tag{1.1}$$

In what the essence is: Suppose that solutions to a problem can be verified quickly. Then, can the solutions themselves also be computed quickly? Or stated differently, douse it exists such an algorithm that can calculate such a solution quickly? A unsolved question considered by many to be the most important problem in the feeld in computer science.

ALTERNATIV 1 Within the class NP there are a few subclasses one of them are called NP-hard Related work has stated that the problem studied in this report are in fact of the class NP-hard [deras artickel]. And hens are chances of finding, an algorithm that produces, an optimal solution in polynomial time are greatly reduced. As stated above

there is still an open question whether such algorithms actually exist for NP problems.

ALTERNATIV2 A more thurol description of the different NP subclasses, definitions etc

[4]

Computers and Intractability: A Guide to the Theory of NP-Completeness (Author), M. R. Garey (Author), D. S. Johnson

1.4.3 (the need for) a near optimal solution

Due to the fact that the problem in question is NP-hard and thus the possibilities of finding an optimal solution within reasonable computing time are/is seen to be very low. Does the logical thread has brought us to be prepared to sacrifice optimality This sacrifice leads us to a larger spectrum of possible approaches to the problem.

1.4.4 Pursuit and evasion problem with multiple pursuers.

The pursuit evasion problem, also known as "Lion and man" [5], is a problem where an pursuer is to find an evader in an environment. An extension to this is the multi pursuer version, which is used in [1],[2] and [3].

The Pursuit Evasion problem. *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* τ *and an i such that* $e(\tau) \in V(p_i(\tau))$*, i.e., the pursuer will always be seen by some evader, regardless of its path.*[1]

1.4.5 Heuristic methods

Heuristic methods are a bransch of methods used in computer science and mathematics to make educated guesses to help find solutions to a problem. Generally heuristic methods douse not provide correct / optimal solutions but it offers a reasonably fast computed solution. Heuristic methods are therefore often used, very powerful, in combination with deterministic methods[fotnot], to speed up the process.

The characteristics of a heuristic is partly that there is no evidence of that it works, or else it would be a deterministic method. The idea of heuristic methods is thus to improve the chances of finding the desired solution. Pure random guesses forms no heuristic method.

There are several occasions where a heuristic method is preferable to a deterministic method. Here are a few:

- There is no known algorithm for solving a specific problem, a heuristic is the only way to approach a solution

- A good algorithm can be difficult to implement and heuristics can sometimes be accepted because it is easy to describe in a programming language and that heuristic methods are known to produce good results.
- The problem is too difficult to solve efficiently and quickly with deterministic methods. A heuristic method could overcome that and give an acceptable but probably not an optimal solution

Where the last point corresponds to the problem we are dealing with.

[fotnot] deterministic method or deterministic algorithms are algorithms that Given a particular input always produce the same output.

Chapter 2

Problem formulation

- presentera multi pursuer pursuit and evasion problem
- frklara svrigheterna givet att problemet r NP-svrt. En av svrighterna som vi belyser extra r den ytterligare dimensionen av obestmt antal jagare. - motivera nyttan och konsekvenserna av att angripa problemet med heuristik -

original problem is the multipursuer version of the pursuit evasion problem. we base our work on paper 1 "boolean control" to see if we can develop the idea of a discretation of the enviroment further and use heuristics to find a good solutions in a short time. our main purpose: test three heuristic algorithms with different conceptual basis and collect data to compare under what kind of conditions each method excels or fails.

2.1 Our problem formulation

the original problem, given by guidas et tal [referens!!] is known to be NP-hard. Here we have relaxed version of the problem.

the headline is self explanatory, the ide her is to discuss the P vs NP aspekt. maybe this will be incorporated with another part of the introduction

2.2 Our Approach

We will now present our intended approach the problem formulated in the preceeding section. The line of work can be divided into four subtasks:

- Create an simulation enviroment.
- Create three fundamentaly different algorithms, using aknowledged heuristic ideas.
- Run the algorithms to get comparable data.
- Evaluate and present the data attained, followed by a discussion.

As outlined above the approach as a whole is heuristic, and thus the need for a testing enviroment is essential. We have decided to implement an enviroment using C, due to the computational effiency of the language and because it is more or less the standard language in computations of this type. We have two main specifications on the enviroment. First, it should be able to, in a stand alone application, create a large amount of randomly generated maps with obstacles (the rate of obstacle versus total area is here on called density) and store these in a file. Second, we want the simulation enviroment to read a map from a file, run the specified algorithms and store desired data in an output file. For further details on how the simulation enviroment is implemented read chapter 3.
Given the multipursuer problem and the simulation enviroment, the next subtask is to create the algorithms and run the simulation. We have decided to implement three fundamentaly different algorithms, so that one could outline under what circumstances each one of them excels, or fails. Using ideas from acknowledged heuristic methods we have chosen the following three concepts as our guidelines when creating our algorithms: Greedy, Genetic programming and Tabu search.
Greedy (G): In G one creates a cost function to describe how good a specific move is. in each iteration the algorithm is supposed to use this function to determine the best possible move for each step, thus giving a local approach to the finding of an optimal solution.
Genetic Programming (GP): The main idea is to generate a big population of possible (not nessecarily good) solutions called populations. in each itteration the algorithm compares the quality of the solution and takes the best
Tabusearch (TS): The main idea of TS is that it keeps a dynamic list of "bad descissions". For each iteration it makes "good descitions" by comparing with the tabu list. TS is in some sens a merge of GS (global searching) and G (local searching)
Further information on how the algorithms are implemented and more in depth theory is found in chapter 4.
So the three different algorithms under consideration are in their approaches fundamentaly different, in how they asses the problem. There are ofcourse many other heuristical approaches that could provide similar results, but by comparison we decided that these three are relatively easy to implement on our problem and covers a large aspect of different approaches.
In order to evaluate how well the algorithms' results are we need to compare them in some sense. Given a specific enviroment, number of pursuers and density we decided to compare the following data for each excecution of the simulation:

- computational time
- the path length of the pursuers

The results are presented in chapter 5 - 6, followed by a discussion in chapter 7

Chapter 3

Simulation Environment

In order to attain the data needed for a comparison of our different algorithms it was nessecary to construct a good testing enviroment. We decided to create this enviroment by the use of two separate parts. One part is called the "Map generator". This part creates a map of the enviroment, tests the feasibility and prints feasible enviroments into an output file. The other part is called the "Network generator". This part reads in an enviroment 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 enviroments. A feasible enviroment is described more in detail in section 2, but in short one could say that an enviroment is feasible if it is simply connected and can be divided into a finite set of convex regions. Given the desired size and the density (percentage of obstacles per total area) as inputs, MG creates square shaped feasible enviroment with randomly placed obstacles and saves the map in an external file. For simplicity we have chosen to construct enviroments consisting only of square regions. We suggest that this does not result in a loss of generality since any feasible enviroment can be approximated arbitrarily good by a sufficiently fine meshing of squares.
We will now show in a more detailed maner 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:

```
1  input variables:
2  int Size;  // Specifies the width and height of the square matrix  A.
3  int NumberOfEnv; // Specifies how many feasible enviroments to create.
4  int Obstacle; //  Specifies the number of obstacles in percents, e.g. number of
5
```

```
6  while ( int i < NumberOfEnv ){
7   int A = CreateMatrix(int Size);
8   PlaceObstacle(int A, int Obstacle);
9   if (Test(int A)=TRUE){
10  fprintf( fileOK, "\n \n");
11  i++;
12  }else{
13  fprintf{file NotOk, "Does not work: \n");
14  }
15 }
```

- So, first we introduce the needed variables to define how many and what kind of enviroments to create.
- The algorithm starts off by entering a while loop running until the loops content has created the desired amount of feasible enviroments.
- The function CreateMatrix(int arg) creates an matrix of dimensions $(arg \times arg)$ with every element equal to one. In our map a one corresponds to a feasible square region for the pursuer or evader to thread.
- Next we call the function PlaceObstacle(int A, int Obstacle). This function takes the input matrix A and using a randomizing function rand() places zeros in the matrix. The zeros corresponds to obstacles, e.g. squares that can't be seen through and can't med treaded.
- The function Test (int A) tests if the ones in the given matrix A are connected. If A is connected the enviroment is feasible and the function returns TRUE, if not it returns FALSE. First Test() finds the first element equal to one, starting from the upper left corner going to the right. When such element has been found its index is pushed on a stack. Next we enter a loop that runs for as long as the stack is not empty. First it pops an index from the stack and checks if this index has been accounted for (e.g. if it lies within a vector C). If not accounted for, all the neighbor elements that are equal to one has their index pushed on the stack. Lastly the poped index is put into C, since it has been accounted for. The algorithm starts over by poping a new index and rerunning the loop.

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. For input the NG uses an environment matrix that is either created by the MG or

by hand.

Pseudo-code, Node Network Generator:

```
1. int 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();
```

- environmentFromFile() sets A to an environment matrix, consisting of zeros and ones, which is read from an input file. The file can be generated by the MG or constructed by hand.

- A Node is a datastructure 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 datatype Node.

- setMove() creates pointers from N to all feasible vertically and horizontally adjacent nodes.

- setVision() creates a list of pointers to all visible Nodes in B from the Node N. Visible Nodes is each Node that can be connected to N by a straight line without passing through a non-feasible Node. The visible nodes are decided by first calculating Umax and Dmax which is the number of columns between N and the first non-feasible Node upwards and downwards in the same row, and Lmax and Rmax which is the number of rows between N and the first non-feasible Node in the same column as N. Each Node up to Lmax columns to the left of N and Rmax columns to the right of N is visible. For each of these Nodes every Node that is feasible and in the same column, at most Umax rows above or the first non-feasible Node, whichever comes first, and at most Dmax rows below or the first non-feasible Node, is visible. Umax and Dmax is re-calculated for every column to the left and to the right of N, and is the minimum of the number of the current Nodes Umax and Umax for the previous Node in the same row, with the same applying for Dmax.

Chapter 4

Methods

A Description of the methods and algorithms we have used.

4.1 Greedy

Greedy

4.1.1 Description

A description of what Greedy is

4.1.2 Algorithm

A description of the algorithm

4.1.3 Implementation

A description of how the algorithm is implemented

4.1.4 Development process

How the development of the algorithm have proceeded.

4.2 Tabu

Tabu

4.2.1 Description

A description of what Tabu is

4.2.2 Algorithm

A description of the algorithm

4.2.3 Implementation

A description of how the algorithm is implemented

4.2.4 Development process

How the development of the algorithm have proceeded.

4.3 Genetic

4.3.1 Description

Genetic algorithms is based on the idea of evolution. Using a combination of reproduction, mutation and survival of the fittest a solution is generated.

4.3.2 Algorithm

A description of the algorithm

4.3.3 Implementation

A description of how the algorithm is implemented

4.3.4 Development process

How the development of the algorithm have proceeded.

Chapter 5

Results

Statistics, tables and a description of the tables. Also why we have chosen these tables etc.

Results for MILP evaluation could also be added here.

Chapter 6

Discusion

This chapter contains analysis of each algorithm, why it did or did not work, how it compares to the other algorithms and a conclusion. The main purpose is to present conclusions from the data presented in the previous chapter.

6.1 Analysis

An analysis of each algorithm, evaluation of why it did or did not work.

6.2 Comparsion and statistics

A comparsion between the algorithms, and perhaps also with MILP.

Chapter 7

Conclusion

A conclusion of our work, and future work.

Bibliography

[1] Johan Thunberg, Xiaoming Hu, Petter Ögren, A Boolean Control Network Approach to Pursuit Evasion Problems in Polygonial Environments

[2] Johan Thunberg, Petter Ögren, A Mixed Integer Linear Programming approach to Pursuit Evasion Problems with optional Connectivity Constraints

[3] Johan Thunberg, Petter Ögren, An Iterative Mixed Integer Linear Programming Approach to Pursuit Evasion Problems in Polygonial Environments, 2010 IEEE International Conference on Robotics and Automation

[4] Brian W. Kernighan, Dennis M. Ritchie, The C - Programming Language, (ANSI C Version)", Prentice-Hall of India Pvt. Ltd., New Delhi, 1998

[5] A. Dumitrescu, I. Suzuki and P. Zylinski, Offline variants of the 'Lion and Man' problem – Some problems and techniques for measuring crowdedness and for safe path planning", Theoretical Computer Science, Vol. 399, June 2008, pp. 220-235.

[6] Chambers, L.D. PRACTICAL HANDBOOK OF GENETIC ALGORITHMS (GAS) APPLICATIONS, VOL 1, CRC Press 1995, ISBN 0-8493-2519-6.

[7] Chambers, L.D. GENETIC ALGORITHMS, VOL 2, CRC Press 1995, ISBN 0-8493-2529-3.