Sneaky Thief

Stan Kerstjens[1], ing. R.J.H.M. Stevens[2], Sina Maghami Nick[3], Rob Clinch[4], and Sharon Hallmanns[5]

Department of Knowledge Engineering, Maastricht University

12th June 2015

[1]i6048794 : s.kerstjens@student.maastrichtuniversity.nl
[2]i6079131 : rj.stevens@student.maastrichtuniversity.nl
[3]
[4]i6044978 : r.clinch@student.maastrichtuniversity.nl
[5]

Preface

Abstract

List of Figures

List of Abbreviations and Symbols

Contents

This page is intentionally left blank

Chapter 1

Introduction

The problem posed can be divided into three main problems. The first problem is that guards put into an unknown environment should be able to create a 'mental map' of the environment. This is what we will call the exploration task. The second problem is that the guards should monitor the entire area as closely as possible to make it as difficult as possible for a possible intruder to pass through the area unseen. This will be called the coverage task. The third problem is that once an intruder has been detected, the guards should be able to actually catch the intruder. This will be called the pursuit task.

In every of these three main problems the guard should figure out two things, namely where it should go, and how it gets there. The 'where'-question is largely dependent on the specific task at hand, e.g. even if an intruder would only be one unit of distance away, but the task is exploration, then the guard does not necessarily have to move towards the intruder.

The 'how'-question, on the other hand, is in many cases independent of the exact task that we want to achieve, and therefore finding the shortest route to the destination does usually suffice, and is often even the best route. We will call the 'how'-question the pathfinding problem.

The final task of the guards is to combine the three main problems and decide when the guard should solve what problem. The intuitive solution would be to explore until the entire environment is known, and then start, and continue, covering the area. During this process pursuit is started as soon as an intruder is detected. This intuitive answer, however, runs into issues once it is not known to the guard how big the area actually is, so it will never know if the map if fully explored or not. Another possible issue arises when an intruder, for instance, disappears behind a wall. The pursuing guard does not detect the intruder anymore and stops pursuit. He should, however, have the intelligence to predict where the guard has gone.

Chapter 2

Approaches

2.1 Path finding

As already mentioned in the introduction, the answer to the question on what path to take to a certain location is in many cases not important as long as we find the shortest path. This is why we can first discuss general methods of path finding without referring to what goal the moving of the agents serves. There are also cases where the shortest path is not the best path. This is heavily dependent on the task, and will therefore be discussed in the sections dealing with the specific tasks.

2.1.1 A-star

The A-star algorithm is a general purpose algorithm that searches a graph and is capable of providing the optimal path from one node to another. A-star is a heuristic search, and like many heuristic search algorithms its effectiveness depends on the heuristic employed (Hart, Nilsson & Raphael, 1968).

A-star uses an evaluation function f which depends on the cost function g and the heuristic function h.

$$f = g + h \tag{2.1}$$

At every iteration it will expand the node that has the lowest cost as predicted by the evaluation function (see alg. 1).

In our simulations the nodes of the graphs will be simple coordinates, so for our heuristic function we can simply take the direct distance to the destination coordinate as our heuristic function.

A disadvantage of the A-star algorithm in path finding is that it will not identify a certain coordinate as unreachable until all possible routes have failed. As this would take to long in most simulations to compute, it is necessary to provide a maximum amount of nodes the algorithm is allowed to explore before it has to give up the search. This, of course, is at the risk of falsely concluding that a certain position is unreachable.

Another disadvantage of the A-star algorithm is that it is necessary to discretise the world in order for the algorithm to run. In real-world applications for robotic systems, most robots will not find themselves in a discrete world. Therefore the A-star algorithm will be very unlikely to find the optimal path

Algorithm 1: A-star algorithm (Hart et al., 1968)

```
Data: Starting node s; Set of target nodes T; evaluation function f
1  Mark s as 'open' and calculate f(s);
2  Select the open node n with the smallest f;
3  if n ∈ T then
4  |  Mark n 'closed' and terminate;
5  else
6  |  Mark n 'closed';
7  |  Let A be the successors of n;
8  |  for a ∈ A do
9  |  |  Calculate f(a);
10 |  |  if a is not closed ∨ f(a) < f when s was closed then
11 |  |  |  Mark a as open;
```

in continuous world applications. If, however, the discretisation is sufficiently narrow the proposed path will probably be sufficient for most purposes. Unfortunately, narrowing the discretisation inherently leads to the need for more computing power.

RTA-star

An adaptation of A-star to be more accommodating to real-time applications is Real-Time A-star (RTA-star). When the computational resources are insufficient to at every step recalculate the entire path from the agent to the goal, this method should be employed. It is very similar to regular A-star with the adaptation that an agent is immediately moved to the most promising direction, instead of first calculating the entire path. This way A-star loses its optimality, but it will execute a lot faster, which is necessary when computing long paths for many agents (Korf, 1990).

2.1.2 RTTE-h

The RTTE-h algorithm uses various geometrical features of the obstacles that the agent encounters to determine the best moving direction as an angle, and also the utilities of all discretized moving directions.

The merging phase of the algorithm determines the utilities as is described in algorithm 8.

The RTTEh algorithm 10 provides answers to two shortcomings of the A-star algorithm for path search. First, it is a continuous space algorithm, so it is not dependend on any kind of discretisation for real world applications. Second, it is capable of identifying whether a certain coordinate in space is unreachable without having to explore a large amount of possible routes.

Another property that will be usefull later on is that this algorithm returns an ordering of all moving directions, instead of just the best one.

Although the algorithm operates in continuous space, the moving directions are still discretized. In real world applications this is usually not the case, and this will therefore lead to a suboptimal result.

Algorithm 2: RTTE-h algorithm (Undeger & Polat, 2010)

```
Data: The agent's current cell s
Result: Utilities of neighbours of s
Mark all moving directions of s as open;
Propagate rays over the edges of the moving directions;
for each ray r hitting an obstacle do
    Let o be the obstacle hit by r;
    Extract border b from o;
    Detect closed directions of cell s using b;
    Extract geometric features of o;
    Determine the best moving direction to avoid o;
Merge the results (see alg. 8);
return utilities;
```

Algorithm 3: Merging Phase of RTTE-h (Undeger & Polat, 2010)

```
Data: current coordinate of the guard s; current coordinate of the target
      t; results from the rays r.
if All neighbours of s are closed then
    return failure;
Determine most constraining obstacle m from r;
if m exists then
    Let proposed direction d be the direction that gets around m;
else
    Let proposed direction d be direct direction to t;
for each neighbour n of s do
    if n is closed then
        Let utility of n be 0;
    else
        Let dif be the smallest angle between d and the direction of the
        cell n;
        Let the utility of n be (181 − dif)/181;
return utilities;
```

2.2 Exploration

2.3 Coverage

Any intruder will try to be undetected. In order to increase the chance of detecting an intruder we need to make sure that the environment is monitored as closely as possible. This is the problem of coverage. Given the environment we want our agents so move in such a way that no intruder can move through the environment undetected. The intuitive answer behind this problem is to try to disperse the guards over the environment as well as possible, which is exactly what the StiCo algorithm aims to do.

2.3.1 StiCo

One way of covering the entire area of the environment is to try to equally disperse the guards over the area. An algorithm that attempts to do this is the so called StiCo algorithm. In the StiCo algorithm all agents lay down a pheromone trail that other agents can detect. The principle is that once an agent detects a pheromone it changes its direction so to not cover an area that another agent already covers. In an non-open environment also the walls and other obstructions can be seen as these pheromones. In nature the pheromones are scents. In our world, the pheromones will be beacons dropped by the agent which can be seen by other agents (Ranjbar-Sahraei et al., 2012).

The StiCo algorithm needs a base movement from which to start. This base movement is moving at a constant forward and angular velocity, which results into describing a circle (if not obstructed). Any time the agent encounters a pheromone trail, it simply changes its angular velocity to move another direction. To determine what direction is the best one to avoid the pheromone trail the agent has two different sensors: an interior, and an exterior. The interior sensor is the one on the inside of the circular motion, and the exterior one is the one on the outside of the circular motion described by the agent. If the interior sensor picks up the pheromone trail, the easiest way to avoid the pheromones is to invert the angular velocity. If the exterior sensor picks up the pheromone trail the agent should temporarily increase its angular velocity to avoid the pheromone trail (see alg. 4, fig. 2.1) (Ranjbar-Sahraei et al., 2012).

Algorithm 4: Iteration of the StiCo Algorithm (Ranjbar-Sahraei et al., 2012)

```
1 while no pheromone encountered do
2 |   Move with velocity v and angular velocity a;
3 if Encountered pheromone at interior sensor then
4 |   a ← −a;
5 else
6 |   while Pheromone is detected do
7 |   |   increase a;
8 |   Set a back to the original value;
```

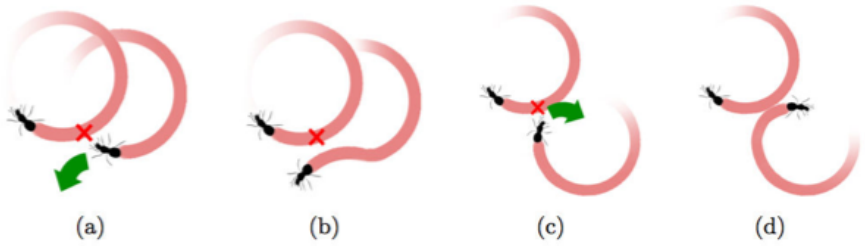

Figure 2.1: StiCo coordination principle (a) and (c) before pheromone detection (b) and (d) after pheromone detection (Ranjbar-Sahraei et al., 2012)

A great advantage of the StiCo algorithm, apart from its simplicity, is that no prior map of the environment is needed. Therefore, we can skip the entire exploration phase and not bother with trying to figure out when exactly an agent should explore the environment, and when it should cover the known map. Also there is no initial time the agent needs to spend exploring.

The other side of the coin is that the agent also does not use a map of the environment. It does not intelligently cope with features of the environment such as entries, doors, windows, exits, etc.

Moreover, the algorithm often reaches an equilibrium, namely when no territories, the area described by the circular motion of an agent, of the agents intersect. There, however, is no guarantee that this equilibrium covers the entire area. One could argue that if this is the case, there are simply not a sufficient amount of guards guarding the premise, but it is quite obvious that in a mansion with only one guard, the guard should not just stand in the main hall walking circles, hoping to run into an intruder.

2.4 Pursuit

When considering the problem of pursuit, there is again a very simple intuitive answer, namely to go directly to the location of the intruder following the shortest path. This will often be the best solution if you are a lone guard pursuing an intruder. Yet even then, this strategy will only work if either the intruder is not actively avoiding the guard, or the guard is faster than the intruder. As we are not prepared to assume any of these conditions, we need to act slightly more intelligently.

2.4.1 MTES

The Real-time Moving Target Evaluation Search (MTES) is an assisting algorithm that prevents an agent in retracing it steps. The concept is very simple: every visited cell is considered as an obstable, until the target becomes unreachable through visited cells. At this point it will clear the history and start over.

In a static environment where the target of the path search does not change there is a very low risk that an algorithm searching for the shortest path will ever try to visit the same cell twice. When the target moves, however, a path-finding algorithm might be advised to just go back and forth, which is undesirable.

An example where this history could help is when two guards are pursuing an intruder around an obstacle. If the shortest path to the intruder is always chosen, the intruder and guards will continue to walk circles around the obstacle. If the guard, however, holds a record of where he's already been, he will try alternative paths as soon as one cycle has been made.

This assisting algorithm requires the use of another path finding algorithm. The RTTE-h algorithm particularly suitable, since it does not only recommend a single moving direction, but multiple. If the first moving direction is blocked by a history cell, MTES can just select the second recommendation (see alg. 8).

Algorithm 5: Iteration of MTES (Undeger & Polat, 2010)

```
   Data: Current cell s
 1 Let d be the proposed direction as determined by RTTE-h;
 2 if d exists then
 3 |   Let n be the neighbour cells of s with minimum visit count;
 4 |   Let c be the cell in n with maximum utility;
 5 |   Move to c;
 6 |   Increment the visit count of s;
 7 |   Insert s into history;
 8 else
 9 |   if History is not empty then
10 |   |   Clear history;
11 |   |   Go to next iteration;
12 |   else
13 |   |   Stop search with failure;
```

2.4.2 Blocking Escape Directions

A possible way of catching an intruder that is faster than the guards is to use the fact that there multiple guards pursuing a single intruder. In this case the guards should surround the intruder as much as possible, as to block all the possible direction in which he could escape. Then the guards can draw in and catch the intruder. The most naive way of doing this is to disperse the available guards equally amongst the circumference of an intruder. The guard closest to the intruder will always move directly towards the intruder following the shortest path. The other guards will establish the amount of pursuing guards (n) and each block one of the angles resulting from dividing the total angle in which the agent can move in equal parts ($2\pi/n$).

Algorithm 6: Determining the Blocking Location

Data: The guard p; the current cell of the guard s; the current cell of the intruder t; the number of guards n
Result: Blocking location

Let (h_x, h_y) be the coordinate of cell s;
Let (t_x, t_y) be the coordinate of cell t;
if $n = 1$ *or* p *is the nearest guard to* t **then**
    return (t_x, t_y);
else
    Calculate the set of escape directions e;
    Determine the map m that maps the guard to e optimally;
    Let esc be the escape direction assigned to p in m;
    return the blocking location using esc (alg. 18) ;

To exactly calculate the blocking location in the last step of algorithm 5 we execute algorithm 18.

Algorithm 7: Calculating the Blocking Location

Data: the guard p; the velocity of the guard v_h; the velocity of the intruder v_p; the escape direction esassigned to p

Let ϵ be a small number (0.5);
Let ε be a very small number(0.05);
Let α be the smallest angle between es and the direction from p to the intruder;
Let d_{max} be the maximum permitted distance between the blocking location and the intruder;
if $\epsilon < \alpha < 180 - \epsilon$ **then**
    if $(\sin\alpha)(1+\epsilon)(v_p/v_h) \leq 1$ **then**
        Let θ be $\arcsin((\sin\alpha)(1+\varepsilon)(v_p/v_h))$ **if** $\theta < 180 - \alpha - \epsilon$ **then**
            Let $pdir$ be the guard direction using θ;
            Let bl be the intersection point of lines passing through es and $pdir$;
            if *Distance between bl and the intruder* $> d_{max}$ **then**
                return the point with distance d_max from the intruder in the direction of es;
            else
                return bl;
        else
            return the point with distance d_max from the intruder in the direction of es;
    else
        return the point with distance d_max from the intruder in the direction of es;
else
    return the location of the intruder;

Chapter 3

Methods

3.1 Simulator environment

3.1.1 Time perception

The time in the simulator is perceived continuouly to allow for continuous velocity, even in a discretized world. Although the moves of the agents on the map are fully discretized, the simulator takes into account how long this unit move has taken to perform, and only allows the same agent to move again once all other agents have caught up with his 'time'. In other words, each agent has his own progress in time, and is only allowed to move when his time is lower (or equal) than all the other agents.

3.1.2 Game Rules

Chapter 4

Results

Chapter 5

Conclusions

References

Hart, P., Nilsson, N. & Raphael, B. (1968, July). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, *4*(2), 100-107. doi: 10.1109/TSSC.1968.300136

Korf, R. E. (1990). Real-time heuristic search. *Artificial intelligence*, *42*(2), 189–211.

Ranjbar-Sahraei, B., Weiss, G. & Nakisaee, A. (2012). A multi-robot coverage approach based on stigmergic communication. In *Multiagent system technologies* (pp. 126–138). Springer.

Undeger, C. & Polat, F. (2010). Multi-agent real-time pursuit. *Autonomous Agents and Multi-Agent Systems*, *21*(1), 69–107.

Appendices

Appendix A

Other Data