Pathfinding with Hard Constraints
- Mobile Systems and Real Time Strategy Games Combined

Erdtman, Samuel & Fylling, Johan

Department of
Interaction and System Design
School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden
Contact Information:
Author(s):
Johan Fylling, Samuel Erdtman
J.fylling@gmail.com, samuel@erdtman.se

External advisor(s):
Alexander Ekvall
Pixelknights AB
alexander.ekvall@pixelknights.com

University advisor(s):
Rune Gustavsson
Department of Interaction and System Design

Department of Interaction and System Design
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : + 46 457 38 50 00
Fax : + 46 457 102 45
ABSTRACT

There is an abundance of pathfinding solutions, but are any of those solutions suitable for usage in a real time strategy (RTS) game designed for mobile systems with limited processing and storage capabilities (such as the Nintendo DS, PSP, cellular phones, etc.)? The RTS domain puts great requirements on the pathfinding mechanics used in the game; in the form of demands on responsiveness and path optimality. Furthermore, the Nintendo DS, and its portable, distant relatives, bring hard constraints on the processing- and memory resources available to said mechanics.

This master thesis aims to find a pathfinding solution well suited to function within the above mentioned, narrow domain. From a broad selection of candidate solutions, a few promising subjects are treated to an investigative empirical study; with the goal of finding the best “fitting” solution, considering the domain.

The empirical study shows that the triangle-based TRA* solution and the hierarchical-abstraction influenced Minimal Memory solution are both very promising candidates. Depending on how one exactly defines the domain, either one of the solutions could be considered the ‘best’ choice. Since the overall performance of TRA* showed a slight advantage, this solution was further investigated by running an implementation on one of the intended domain platforms; the Nintendo DS.

This paper is structured to serve as a guide, of sorts, to some very interesting, and diverse, pathfinding solutions. In the spirit of this effort, all of the more important aspects of these solutions, and the pathfinding domain as a whole, are thoroughly explained.

Keywords: Pathfinding, Nintendo DS, RTS-game.
Preface

The time spent on this master thesis has been very interesting, but also very intense; our social lives committed collective suicides sometime around the middle of the semester. To make things short, here is the list of clichés we don’t want to say; but do, anyways:

- It has been a learning experience.
- It has been fun.
- Planning is good, and we plan to do it for our next project.
- We hope this paper can be of future use to someone, somewhere, somehow.

To everyone at Pixelknights: a big ‘thank you’, and keep up the good work.

We wish to extend our humble thanks to our advisors: Alexander Ekvall, at Pixelknights; and Rune Gustavsson, at BTH.

For those who have read this paper and intend to give TRA* a try: beware of CDT. It haunts us at night.
1 Introduction .................................................................................................................. 7
2 Background ...................................................................................................................... 9
3 Problem definition ......................................................................................................... 10
  3.1 Contributions ............................................................................................................. 11
4 Search Space Abstractions ............................................................................................. 12
  4.1 Grids .......................................................................................................................... 12
  4.2 Waypoints .................................................................................................................. 13
  4.3 Triangles ................................................................................................................... 14
  4.4 Hierarchical Solutions ............................................................................................... 15
5 Heuristics ......................................................................................................................... 17
  5.1 Manhattan distance .................................................................................................. 17
  5.2 Euclidean Distance ................................................................................................ 17
  5.3 Squared Euclidean Distance .................................................................................... 17
  5.4 Diagonal Distance .................................................................................................... 17
  5.5 Additional Improvements ......................................................................................... 18
6 Search Strategies ............................................................................................................. 19
  6.1 Breadth First ............................................................................................................ 19
  6.2 The A* Family .......................................................................................................... 19
  6.3 Hierarchical Pathfinding A* .................................................................................... 22
  6.4 Partial Refinement A* ............................................................................................ 24
  6.5 Minimal Memory Abstraction .................................................................................. 25
  6.6 Triangle search ......................................................................................................... 26
7 Empirical Study ............................................................................................................... 30
  7.1 Choice of Algorithm Rationale .............................................................................. 30
  7.2 Implementation Specifics ......................................................................................... 31
  7.3 Experiment Setup .................................................................................................... 33
8 Results .............................................................................................................................. 34
  8.1 Time to complete ................................................................................................... 34
  8.2 Memory usage ........................................................................................................ 39
  8.3 Path optimality ....................................................................................................... 42
  8.4 Analysis ................................................................................................................... 43
9 Real World Connection .................................................................................................. 44
  9.1 Path cache ............................................................................................................... 44
  9.2 Experiment setup .................................................................................................... 45
1 INTRODUCTION

In science, pathfinding is the task of finding an uninterrupted (traversable) path – preferably the best – between two, or more, points. This is a universal problem that applies in all areas where some kind of movement takes place; e.g. robotics, video games, military movements, network traffic, etc. All these areas are dependent on good pathfinding solutions to function in a desirable manner. Finding the shortest path between points A and B is, however, often only part of the larger problem. If there are no additional constraints on the solution other than the primary task of finding the shortest path, there exist very simple search algorithms that will always find the best path (if there is one). Although these algorithms can find the optimal path, they are often themselves suboptimal solutions when considering the greater picture. A fast paced video game would no doubt suffer if the NPCs (Non-Player Character), whom are expected to act smart and react even faster, had to stop for several seconds to contemplate their next move every time they wanted to move to another geographical location. Video games are but one of several examples of systems where, apart from path optimality, there are several aspects that factor into how well a particular pathfinding solution is suited for the task.

The task of pathfinding belongs to the infamous group of NP-hard problems, meaning that the size and structure of the search space has a very strong, adverse affect on the execution time of the algorithm. There exist a vast amount of pathfinding solutions, ranging from the very simple and elementary (e.g. Dijkstra’s algorithm, A*, etc.), to the more complex (e.g. hierarchical- and cooperative solutions). This paper attempts to find a solution that is suited for the very specific task of pathfinding in an RTS game developed for the Nintendo DS, but the results are applicable for other problem areas with similar constraints. This specific problem area puts very hard demands on both execution time and memory usage, while path optimality is of slightly less importance. This makes it possible to treat path optimality as a performance variable, just as the speed- and memory aspects of the algorithms. A solution that delivers suboptimal paths could, in other words, be the better choice under the right circumstances.

It is important to note that even though this thesis has its focus on RTS-games for the Nintendo DS, the information and results it contains have a much broader field of relevancy. None of the constraints or restrictions mentioned in this paper are exclusive to the above mentioned domain. Even if you have the processing resources of NASA, there is no harm in using a pathfinding solution capable of running efficiently on a cellular phone, if the output is the same – except for the gross waste of available CPU time, that is.

Section 2 (Background) of this document gives a brief background, by explaining the intended use of the results generated by this thesis. The problem is given a distilled definition in section 3 (Problem definition); where also the practical and theoretical contributions of the thesis are mentioned.

The findings of the literature review, which serves as a basis for a large portion of this thesis, is presented in: section 4 (Search Space Abstractions), where different types of world abstractions are discussed; section 5 (Heuristics) detailing on the impact of different heuristics; and in section 6 (Search Strategies) where several pathfinding solutions are given thorough descriptions.

A select group of pathfinding solutions are treated to an empirical study, where traits of contextual relevance and statistical significance are found and examined. This study is explained in section 7 (Empirical study), and its results are presented and analyzed in section 8 (Results).

A firm connection to the intended real world application area is given in section 9 (Real World Connection), where the performance of the final solution is tested on a Nintendo DS.

In section 10 (Conclusion and Discussion), the findings of this thesis are discussed.

¹Constraints are discussed in the problem definition in section 3.
Several areas requiring additional attention, in form of further theoretical- and practical study, are mentioned in section 11 (Future Work). Here, possible improvements and changes of some of the examined pathfinding solutions are also proposed.
2 BACKGROUND

Pixelknights AB is a Swedish company specialized in developing games for handheld devices, such as the Nintendo DS gaming device. The company might be best described using the words of its staff:

‘Pixelknights is a game development company based in Sweden. Since its founding in 2005, the company’s vision has been to become the world’s leading developer of real time strategy games for handheld platforms.

Being led by the enthusiastic Peter Horvath, CEO and founder of Pixelknights, the company backbone is made out of a group of determined entrepreneurs working hard to realize this vision.

Pixelknights utilizes optimized pipelines specifically created for handheld console development. This, in combination with vast experience of the genre and breakthrough design, makes the company effectively overcome platform limitations that today is a recurring problem when dealing with handheld consoles. By doing so, Pixelknights is right now developing new, innovative, competitive and appealing real time strategy games for handheld devices.

Currently, Pixelknights is developing for the Nintendo DS and the PlayStation Portable. [W1]

For an upcoming project they are to develop a real-time strategy game (RTS) for the DS. The limited memory- and CPU facilities of the Nintendo DS, in combination with the specific challenges brought by the RTS-setting, brings several very challenging, and interesting, requirements to the pathfinding solution that will ultimately be used in the game. Hopefully, this paper will be able to shed some light on the subject of pathfinding.

Figure 1: Origo DS screenshot.

Figure 2: Origo DS screenshot.
3 Problem definition

Ever since the emergence of the first primitive search algorithms, there has been a solution to the most basic definition of the pathfinding problem, in which there are no spatial or temporal constraints on the solution. Given enough time and memory, even the most basic of brute-force search algorithms can exhaustively search the solution space to find the optimal path between two points. This definition of the problem is, however, very optimistic. Computers are, by the laws of physics, no Turing Machines, and there is, and will always be, an upper bound on how much memory they can facilitate; thus all pathfinding solutions have spatial constraints. The temporal constraint comes from the simple fact that there is often a querying system that has requested the path for one reason, or another, and can’t wait indefinitely for an answer.

There are three groups of constraints of interest to this research:

- **Temporal**: the time it takes a pathfinding solution to find the most optimal path it can generate.
- **Spatial**: the amount of memory that must be allocated by the pathfinding solution during runtime.
- **Path Optimality**: the degree to which the resulting path corresponds to the shortest path possible in a Newtonian universe.

On a Nintendo DS, the spatial constraints can be considered very hard. All pathfinding algorithms need some form of representation of the world they are supposed to find paths in; a simplified abstraction of the world, which the algorithms can understand. The memory architecture of the DS poses great limitations on how much information a world abstraction can contain. Memory available for dynamic allocation during runtime is, for the same reasons, very restricted.

In any video game, responsiveness is of great importance. The speed of the pathfinding algorithm used in an RTS game can greatly affect how responsive the game feels to the user; when a unit is ordered to move, it should, ideally, do so without any noticeable delay. To further add to the temporal constraints, pathfinding is seldom the only concurrent task of a video game, and is often given a small percentage of the processing time, in favor of graphics- and sound processing, and other systems of the game. Considering this, and with several units requesting path information simultaneously being a common thing in most RTS games, the temporal constraints of pathfinding in an RTS game can be viewed as hard.

The path optimality constraint of an RTS game can be considered to be moderate. If a path exists between two points, the pathfinder must generate a result, but not necessarily the shortest possible. As long as the movements of units aren’t prone to peculiarities, or can be viewed as moronic by the player, close enough is often accepted as good enough.

When combining the constraints of an RTS game with those generated by the DS, the spatial- and path optimality constraints are transferred largely unchanged. The temporality constraint must, however, be modified, as the limitations of the DS influence the performance of the algorithm. By itself, the DS doesn’t generate any temporal constraints, as there are no requirements on how long time the system can work before it must generate an answer. But, as previously mentioned, much attention must be given the responsiveness of an RTS, and the limitations of the DS make the need for a fast algorithm somewhat more vital than on a modern PC; making the temporal constraints very hard.

---

2 See Appendix C.
3 Although speed is of importance even on a cutting edge PC, the lack of speed is more noticeable on a more limited system.
Considering the above, the problem definition for this research is as follows: what pathfinding solutions are there, if any, satisfying the very hard spatial- and temporal constraints, and moderate path optimality constraints posed by an RTS game running on a Nintendo DS?

### 3.1 Contributions

This paper describes several approaches to pathfinding – ranging from the very basic, to the more advanced – that are all thoroughly described and compared. This is based on a theoretical literature study, designed to single out solutions that specialize in memory- and/or speed efficiency in such a manner that they are within, or close to, the bounds of the constraints described in the problem definition.

Based on an analysis of the solutions collected during the literature review, a few promising candidates have been selected for closer inspection. These have been implemented in C++, so that they could be run in a testing environment, making the collection of real world data a possibility. This data is used in an empirical study to further analyze and compare the selected methods; both to see how they fare against each other, but also to see if it is possible to replicate the results presented by their creators. Many of the algorithms described in this paper can be found as pseudo code in appendix A.

The solution deemed to most adhere to the constraints described in the problem definition has also been ported to be executed on a Nintendo DS. Measurements have been collected from this setup, and have been used to give this study a firm real world connection, since the actual performance of the solution, within its intended environment, can be presented. It cannot be stressed enough, however, that even though a Nintendo DS has served as the testing bed for this thesis, the trialed pathfinding solutions are in no way restricted to only this platform. The solutions will run equally well on hardware of comparable capabilities; and even better on faster devices with larger memory resources. One of the reasons for selecting the Nintendo DS is that it is a very popular gaming device, while at the same time being no power-horse.

---

4 This includes all solutions implemented by the authors, along with some additional methods that deserved special attention.
4 Search Space Abstractions

It is not unusual that a world, in which a path is wanted, is too complex to be efficiently searched by a pathfinding algorithm. The algorithm needs an abstraction of the world that contains only necessary information that is easily accessible. Most world abstractions are realized through some kind of grid- or waypoint structure; why these can be seen as the two main categories into which most other abstractions are ordered. Hierarchical- and certain triangular structures introduce changes noteworthy enough to be discussed separately.

4.1 Grids

The tiles (smallest atomic unit) of a grid structure can take any form that allows for repetitive tiling, among which 4-way tiles, 8-way tiles, texes and hexagons belong to the more usual; which is why they are given focus in this section.

![Figure 3: The grid structures: 4-way-tile, 8-way-tile, tex and hexagon.](image)

Different tile-representations can bring slightly different results when searching them; which is further investigated by Björnsson et al. [1].

The creation of a grid representation can be done by superimposing a grid structure on the world map (see Figure 4), and marking each tile with the information one wish to translate from the ground it covers. The size of the tiles, and what information each tile contains that affect the pathfinding process, depends on how precise one want the algorithm to be. For instance: is it enough to only store if a tile is traversable as a binary value, or must also parameters that affect movement speed be kept (e.g. foliage, hills, swamps, etc.).

![Figure 4: Example of a grid abstraction superimposed on an organic map.](image)

As seen in figure 1 and 2 above, grid structures are conceptually simple, and fairly simple to construct. The number of nodes that must be processed by the pathfinding algorithm is, however, quite great. A game map with a size of 512x512 tiles, for an example, amounts to 262144 nodes, which is quite a big number, indeed, when compared to the abstractions that are covered later in this text. One of the greater disadvantages of grid structures becomes apparent when observing maps containing large open spaces. These spaces are then represented by large groups of tiles containing the same information, which can be viewed spatially as large chunks of redundant data, and temporally as a large amount of nodes that must be examined without, individually, bringing much change to the internal state of the algorithm. One can change the resolution of the grid to affect the amount of tiles, but with a potentially adverse effect on precision.
The quadtree [2] is an alternative to classic uniform grids. Quadtrees support tiles of variable size, which reduce the above mentioned redundant tiling of ordinary grid structures to a degree, giving a structure bearing resemblance to the waypoint graphs mentioned in the next subsection. The creation of a quadtree is a somewhat more complex process than that of creating a uniform grid. The process starts by splitting the world map once horizontally and once vertically; dividing it into four equal tiles. If the underlying ground of that tile is entirely traversable or non-traversable, the tile is denoted as such. If not, the process is repeated on that tile; dividing it into four equal parts, and examining each part to see what to do with it. At some point this process will reach a pre-set size limit on the tiles. When this happens, the tile in question is labeled as the majority of the ground-type it covers.

When the tiling is completed, the centers of all the tiles in the quadtree are connected to form a graph, in which the actual pathfinding searches are made, alternatively, the nodes are placed on the borders between the tiles, where all nodes that share a tile are connected to each other.

As can be seen when comparing the Figures of this section, the total number of tiles can be decreased significantly when using quadtrees, compared to when using uniform grids. The disadvantage is that paths can look strange, and be far from optimal, without any form of post-process refinement. Placing nodes between bordering tiles, instead of only in their centers, raise the granularity of the abstraction, thus improving the quality of the paths. Doing so will, however, somewhat lessen the node-reducing qualities of a quadtree.

4.2 Waypoints

In waypoint-based abstractions, the world map is populated with waypoints/nodes, whom are interconnected with collision free links/edges; creating a graph.

The algorithms used for searching waypoint-graphs are conceptually very much the same as those used for searching grid structures. In fact, a grid can be seen as a very dense waypoint-graph, where there is a node for every traversable tile, connected to the nodes covering the adjacent tiles.

When constructing the waypoint-graph that is to be used by the pathfinding resources of a game when it searches the corresponding game map, it is not an unusual practice to manually place the nodes. This brings a great advantage in that a human mind is capable of making complex judgment calls that might be very hard for an algorithm to approximate. Nevertheless, there exist a very large and diverse set of methods to automatically generate graphs based on the topology of the world maps.

The Probabilistic Roadmap Method (PRM) is one of the simplest methods for creating a graph. It works simply by randomly inserting waypoints/nodes into the world map, and then connecting them to form a graph [3, pp. 923] – see Figure 4. Because of its simplicity, and the fact that it actually works, has made PRM, and its conceptual siblings, a common approach in robotics. Because of its wide use, the method has been subject to an equally wide range of improvements. The focus of most improvements is aimed at how the waypoints are placed, attempting to optimize spread and minimize the number of edges [4] [5] [6]; keeping the branching factor of the graph as low as possible. One of these refinements of RPM makes radical changes to how waypoints are placed and is, by the authors (Nieuwenhuisen et al), specifically intended for use in video games. This method places the
waypoints on the edges created by a voronoi\textsuperscript{5}-diagram; minimizing the number of nodes simultaneously as a smooth roadmap is created (see Figure 6).

Another approach for creating waypoint graphs is to use navigation meshes where point-of-visibility (POV) graphs\textsuperscript{6} \cite{7} \cite{8}, belong to the more common. POV-graphs are created by placing nodes in the vicinity of all convex corners in the world map, and then connecting all nodes that has a clear line of sight between them \cite{9} \cite{10} \cite{11}. An example of such a graph can be seen in Figure 6.

### 4.3 Triangles

One possible abstraction, which can be seen as a special case of the grid abstraction\textsuperscript{7}, is to divide the search space into a mesh of interconnected triangles. In order to maintain a grid-like structure, the triangles must abide by two rules: every unique triangle must cover a surface that is completely disjoint from the area covered by all other triangles, so that no overlapping of triangles can exist; and, an edge of a triangle can border at most one other triangle, so that all triangles border at most three other triangles (one on each side). Triangular tiles have the positive effect of only having three neighbors; resulting in a low branching factor in the corresponding graphs. Another benefit of triangular tiles is that triangles can vary in size; large surfaces don’t necessarily result in many triangles.

1934, Boris Delaunay defined a method for constructing a triangulation of a set of points in the plane \cite{12}, in which the smallest angle of all the triangles are maximized; making all the triangles as equilateral as possible. A triangulation is considered Delaunay if all contained triangles has no other points within their circumcircles\textsuperscript{8} than their own. This is a solid method for triangulating environments of polygonal nature, where regions of traversable and non-traversable ground are represented by polygonal bodies; i.e. every transition from traversable space to non-traversable space is represented by the side of a polygon. The corners of the polygons would then serve as the points which would be

---

\textsuperscript{5} Voronoi diagrams place edges as far away from the objects as possible.

\textsuperscript{6} POV-graphs are also known as visibility- and corner graphs.

\textsuperscript{7} See section 4.1 Grids.

\textsuperscript{8} All circles can be defined by any three points on its border, and by constructing a circle using a triangle’s three corner points, one gets that triangle’s circumcircle.
triangulated. A problem of basic Delaunay triangulation is that there is no guarantee that all, if any, of the polygonal representation’s edges will be translated into the triangulation. These edges must survive the triangulation, if the triangulation is supposed to be a viable abstraction of the polygonal environment.

Constrained Delaunay Triangulation (CDT) is a suboptimal hybrid of basic Delaunay triangulation, where it is possible to introduce edges that would otherwise not be part of the Delaunay triangulation. The benefit is obvious, as it is possible to guarantee that all the edges of all polygons in the environment are present in the triangulation. The side effect is that no triangle can be guaranteed to be strictly Delaunay.

The triangles in a CDT can themselves be used as nodes, as in the case of TRA*. It is also possible to use them as an aid in placing waypoint nodes, where one, for instance, can place the nodes on the center of each unconstrained edge.

4.4 Hierarchical Solutions

The fourth, and final, approach of abstracting a world map is to create a hierarchical structure. These structures can be considered as layers of more and more simplistic abstractions. By doing this, it is possible to search higher, more simplistic, layers first, and using this information to limit the search space when analyzing lower, more detailed, levels. Sections 6.3, 6.4 and 6.5 describes three different types of hierarchical pathfinding solutions (HPA*, PRA* and MM).

Hierarchical abstractions bring two main advantages when compared to non-hierarchical structures. The first is that search time is significantly reduced; there are studies that suggest an improvement in magnitudes of ten [13, pp. 27] [14, pp. 5]. This temporal advantage can be traced to the fact that higher levels of abstraction are simpler, and thus faster to search, and gives clues to how lower layers look. The second advantage is found in the native support for time splicing found in hierarchical abstractions. There is no need to search the entire path on all abstraction levels before delivering the first part of the path, as a rough sketch of the path exists already after searching the first layer [14]. This is a great advantage when there are hard constraints on reaction time, as in robotics and video games. For instance, a unit in an RTS game can start moving before the entire process of finding the most optimal path is completed.

* See section 6.6.
There exist several types of hierarchical abstractions, but most can be ordered into two main categories: those where higher layers link directly to lower layers; and those where the layers are themselves contained in a graph structure. PRA* (Section 6.4) is an example of the first, HPA* (Section 6.3) of the second, and MM (Section 6.5) is a hybrid.
5 Heuristics

In pathfinding, the heuristic is an estimate of the distance between two points. More precisely: the distance between the node currently being examined by the pathfinding algorithm and the goal node; i.e. the estimate of how far one has left to travel to journey’s end. The degree to which, and the manner to how, this estimate corresponds to reality can greatly affect the performance and behavior of different pathfinding algorithms [3, pp. 94.]. As an example, one could examine how A* would behave if the heuristic delivered extreme values: in one extreme, where the estimate is always zero, the heuristic would have no affect on the algorithm, making it behave like a breadth-first search; on the other side of the spectra, where the heuristic would always be a gross overestimate, the search would not be guaranteed to produce the most optimal path, as the path already traveled by the algorithm would have little, to no, affect on the search. Both of the above described extremes defeat the purpose of using heuristics. What is wanted is a third extreme: the perfect estimate. This estimate would always perfectly reflect the actual distance between two points. This would, however, require the actual path between the points to be known beforehand the search; bringing the entire purpose of the pathfinding into question. [W2]

The heuristic function of a pathfinding algorithm is frequently called during a search. Which means it is often advantageous to balance the exactness of the heuristic with its complexity.

5.1 Manhattan distance

A straightforward method for creating a heuristic for pathfinding is to calculate the Manhattan distance from the position currently under evaluation, to the goal position. This is a very solid solution if movement is restricted to the horizontal and vertical planes (i.e. 4-way tiles), making the heuristic a very exact estimate of the shortest possible path to the goal position. Should movement be possible in more directions (e.g. diagonally), the shortest possible path is no longer a guarantee, and the heuristic will thus occasionally be an overestimate. The result of this can be that an A* search no longer guarantees the shortest path between two points. Positive side effects are, on the other hand, that the search will move faster towards the target position; not exploring sidetracks not in the direct path. The calculations of the Manhattan distance is defined as the distance on the x-axis between start and goal, summed with the distance on the y-axis.

5.2 Euclidean Distance

The Euclidean distance represents the shortest path possible between two nodes – a straight line. This means the Euclidean is never an overestimate, but is instead rather often an underestimate, giving the heuristic a rather low influence in the overall fitness function used when evaluating a node. An additional disadvantage of this heuristic is that it requires the square root of a value to be calculated, an operation that is quite costly to performance.

5.3 Squared Euclidean Distance

To remove some of the cost when calculating the Euclidean distance, it is possible to skip the expensive square root in the formula. A danger of this is that the heuristic part of the cost function will be grossly overweight, with enormous repercussions for longer paths; i.e. in a worst case scenario, A* will find suboptimal solutions. This imbalance problem could, however, be solved by replacing the other half of the cost function, the g value, with its own square.

5.4 Diagonal Distance

The diagonal distance is admissible for 8-way-tiles, in the same manner as the Manhattan distance is admissible for 4-way-tiles, but it is more complex. To calculate the diagonal distance, one will first have to check if the movement is mainly vertical or horizontal. The shorter side is multiplied by the
cost of the diagonal\textsuperscript{10} movement and summed up with the remaining of the long side, subtracted by the shorter side.

### 5.5 Additional Improvements

The heuristic calculations does not have to be static, one can change both the heuristic and the implication of the heuristics in runtime. Furthermore, floating-point calculations impact performance negatively. As mentioned earlier, a perfect heuristic would make A* only expand along the shortest path and newer stray from it. One way to achieve this is to provide pre-calculated heuristics from every tile/node to every other tile/node; this would however require much larger amounts of memory for representation. Therefore, a generalization could help; e.g. a coarse grid, or waypoints [W2].

\textsuperscript{10} The cost might be $\sqrt{2}$ for uniform grids.
6 Search Strategies

This section contains descriptions of some of the strategies available for pathfinding. It does, however, not intend to include all strategies ever invented, but does contain a broad spectra of ideas and implementations; from the simplest breadth first implementation, to more complex ideas using hierarchical structures. Each section contains a description of how one algorithm searches its abstraction, and if a special abstraction is needed its creation is described alongside the algorithm. Some of the algorithm descriptions are complemented with pseudo code, which can be found in appendix A.

As mentioned in the paragraph above, the following sections does not contain descriptions of all available search strategies, since all strategies are not relevant to what this thesis is trying to accomplish; finding a suitable algorithm for use in an RTS intended for the DS console. Some solution strategies that has been left out are: cooperative pathfinding [16] [17], which demands too much resources; and more complex solutions for handling dynamic environments [18] [19] [20] for the same reasons.

6.1 Breadth First

This is a simple uninformed search strategy for path finding algorithms. With its great simplicity come great limitations. If an optimal path is to be guaranteed, breadth first search assumes that all link costs are uniform. Furthermore, breadth first search also demands great amounts of memory and CPU time when the search space grows [3, pp. 73].

To search with breadth first search, an open list using FIFO (first in first out) is needed to store the nodes before it is their turn to be expanded. In addition to the open list, some kind of structure to remember parent values and which nodes that has been visited; this could be done either by marking the nodes visited or having an additional structure to store them in.

Whit these data structures it is simple to do breadth first search: the starting node is pushed on to the open list and the loop is started. The loop checks if the open list is empty; if so, no path exists and the search is aborted. If there are nodes in the open list, the first one is popped to be used as the current node. The nodes adjacent to current that is not marked is marked and pushed on to the open list, and the loop starts from the beginning again; checking if the open list is empty.

6.2 The A* Family

A* is the base of a whole set of algorithms that just makes smaller changes to this ‘parent’-algorithm in order to make the search faster and more memory efficient. This section will show some of these variations and explain their differences. It is, however, not intended to be a complete comparison of the entire A* family; it will just give a brief introduction to it.

Starting with A*, the base case, A* is an informed search method [3, pp. 97]. Meaning that it estimates the distance from its current position to the goal, and has that estimate in mind when selecting the next step in the search. The search can be done on just about any abstraction, with only minor changes to the algorithm.

Searching with A* is relatively simple [W3], but some components has to be known before it can be described in a simple manner. First the g value, which is the real cost of walking from the start to the current node; second, the h value, which is the heuristic distance from the current node to the goal node; and lastly, the f function, which is the sum of h and g. An example can be seen in figure 10. To

---

11 Such as D*.
12 Not only for pathfinding, but in other situations depth first might be considered simpler.
13 If cost is not uniform Dijkstra could be an alternative.
14 See section 5 on heuristics.
15 The estimated cost to the goal stat does not have to be a distance.
be able to perform a search, one need more than measurements, one will need some data structure
to store the nodes (contains at least position, parent and g) as. For this A* uses two; an open list,
which stores the nodes that has been seen but not yet expanded (this list is kept sorted by the nodes
f value); and a closed list, containing the nodes that has been visited.\footnote{For efficiency it is recommended to use a hash, because when needing to check if a node is in the closed list
it can be done in constant time.}

Now when the basics have been explained, it is time to look at how the actual search is performed.
The first step is to put the starting node into the open list; creating an initial setting to start with.
Then, a loop is started; picking the fist node from the open list and denoting it as the current node.
The current’s adjacent nodes are opened, and if they exist in the closed list they are disregarded. If
not found in the closed list, they are searched for in the open list. If the node is found in the open list,
the g value of the new node is compared with the g value of the one in the open list, and the smallest
will be stored in the open list. If the node was not found in the open list, it will just be added to it.
When all of the current node’s adjacent nodes have been expanded, the loop restarts picking the
first node from open list naming it current, repeating the procedure until the end node is found (for
pseudo code of A*, see appendix A.1 A*).

The greatest disadvantage of A* is that it tends to use quite a lot of memory storing the expanded
and visited nodes in the open- and closed lists, and when these lists becomes large, it also takes a lot
of time to keep the open list sorted. These are the main points that the different variations of A*
tries to address.

6.2.1 IDA*

First out of the A*-based algorithms is IDA* (Iterative Deepening A*) \cite{pp. 101}, which has little in
common with A*, but falls into the family because it uses the same estimate ($f = g + h$). If overlooking
this similarity, IDA* is a totally different algorithm. IDA* has neither a open- nor a closed list to store
and sort nodes in, instead it uses the same approach as iterative deepening (which in turn builds on
depth first search); just as the name implies.

Depth fist search expands the right- (or left-) most child to the bottom of the search space and then
backtracks to the closest parent still having children, and there again expand the rightmost child not
expanded yet. Iterative deepening uses the same approach, but does not allow the search to go to
the bottom of the search space - instead it has a limit for how many levels the search is allowed to
expand, if the goal is not found within the first limit the limit is increased and the search is done
again, and again until the goal is within the limit. IDA* does just this, but instead of limiting the num-
ber of levels of the graph to expand it puts a limit on the f value of the nodes.

This gives IDA* the same limitations and benefits as iterative deepening. It will use less memory be-
cause the number of simultaneously opened nodes is never greater than the depth to which the
search goes. In addition to this, it does not have to sort lists of nodes, or even have special entries for
the nodes, which is quite a big change from A*. However, just as depth-first-search it has no ability
to detect loops. Even though it will not hang in the loops, it will be very inefficiently used time, and as
graphs in pathfinding has numerous loops, a lot of time will be spent here. In addition to this, IDA*
performs the search from start the starting node for each iterated search. Due to the exponential
growth in nodes for each layer this is not the biggest problem, but it is a factor of the algorithm that
slows it down. To address these problems, a set of algorithms which use some memory to avoid
loops has been developed; e.g. ME-IDA*, using a transposition table; and Fringe, storing only the
frontiers\(^\text{17}\) of the expanded tree.

### 6.2.2 RBFS

Recursive Best-First Search is, just like IDA*, an attempt to obtain linear memory usage while mimicking
best first search [3, pp. 101]. It makes use of the \( f = g + h \) -function, just as IDA*, allowing it into
the A* family, but as with IDA, this is where the similarities ends. RBFS uses neither a open- nor a
closed list, which means that it has to keep track of nodes in a different way, or revisit a lot of nodes.
RBFS, like IDA*, goes for the second solution.

RBFS starts up with a call to the recursive\(^\text{18}\) function with the arguments: search space, initial
node, and best value \( f \) infinite. The recursive function then evaluates the in-node to see if it is the goal, if
so it returns success and the path is backtracked through the recursion. If the in-node was not the
goal it continues to analyze the nodes adjacent to the in-node, meaning that it calculates the \( f \)-value
of all successors. When the \( f \)-values are calculated, the best (the lowest) is selected. If the \( f \)-value
of the best node is larger than the limit, the algorithm cannot walk further on this path for now, so it re-
turns the best \( f \) value and a message of failure; that it has not found the goal. On the other hand, if
the best node's \( f \)-value was lower than the limit, it calls itself with the best node, the second best \( f-
\)value, and \( f \)-limit\(^\text{19}\). In this way, the algorithm then continues walking back and forth until it finds the
goal.

As seen above, the RBFS algorithm has no way of knowing if a node has been visited earlier, this
makes it weak in situations where loops are common. Furthermore, every time the limit is exceeded,
RBFS has to backtrack and re-expand nodes. In small and simple search spaces this does not become
a big overhead, but when the search space becomes larger and more complicated, a lot of time will
be spent revisiting nodes.

### 6.2.3 SMA*

SMA* stands for ‘simplified memory-bound A*’, and is a sibling to memory-bound A* (MA*) [3, pp.
103]. Both are designed to use all memory available, in contrast to IDA* and RBFS, that both have lin-
ear memory usage, and A* that does not limit itself in any way.

Searching with SMA* starts just like A*, and continues like A* until the available memory is full, then
the node with the worst \( f \)-value is removed. If all nodes have equally good \( f \)-values, the oldest is re-
moved and the newest is expanded. When removing the node, the values of it are stored in its par-
ent, so that the quality of that path will still be known even though the exact path is not known.

One important things to consider with SMA* is that a solution can be unreachable; e.g. if available
memory is three nodes and the shortest path is five nodes then it will be impossible for the algorithm
to find a solution. With the same amount of memory, however, this is an impossible feat for A* also.
There are also situations where a path can be found, but not the optimal one.

The fact that SMA* removes nodes can result in the need to revisit nodes. This is no problem in sim-
pler search spaces where this might happen once or twice, but in more complex search spaces this can
result in a constant switch between paths in the search space, which in turn can result in behavior
similar to thrashing in memory paging. SMA* is, nevertheless, a stable solution in many cases where
the cost of expanding nodes is greater than the cost of the additional computations.

---

\(^{17}\) See section 6.2.4

\(^{18}\) Could as most recursive functions be implemented iteratively, but it has a recursive structure.

\(^{19}\) This is not a complete description of how the algorithm works, it just gives a brief overview.
6.2.4 Fringe

Fringe (as in border) was developed by Björnsson et. al. [21] to eliminate the major drawbacks\(^2\) of a IDA*, while keeping the advantage of not having to sort any open list. The behavior of Fringe can be adjusted to resemble either IDA* or A*. Increasing the A* resemblance, however, also increases the amount of sorting needed. In the report by Björnsson et.al. [21], they show that even though fringe expands more nodes than A*, it shows an increase in speed of up to 40 %. This is deduced from the fact that Fringe has no list to sort, while A* spends most of its processing time sorting its open list.

The idea is to do a search like IDA*, with a limit on how far to go on each iteration but too keep a list containing the frontier between iterations. A comparison between IDA* and fringe can be seen in figure 11.

![Figure 11: Visualization of IDA* (to the left) and fringe expanding nodes in a tree. [21, pp. 2]](image)

As can be seen in figure 11, Fringe stores the nodes in the frontier for the next iteration, so that all earlier nodes do not have to be revisited every iteration. In the example in figure X, Fringe expands nine nodes while IDA* expands 14, which is 55% more just in this simple example. The Frontier is divided in two groups: now, and later. The now-group is those that have an f-value smaller than the limit; and the later-group, those with an f-value greater than the limit. To avoid repeated states in an efficient manner, Fringe requires a perfect hash\(^2\). And to simulate a more A*-like behavior, the now-part of the frontier could be sorted.

A search with Fringe starts by putting the start node in the now list. When the start has been inserted, a loop starts by retrieving the first element in the now-list and denoting it as the current node. The current node’s adjacent nodes are iterated over and checked if they exist in the cache. If so, the g-value and parent-node is updated if needed. If the node did not exist in the cache it is added to the cache, and the frontier list is updated. When all the current node’s adjacent nodes have been iterated over, the next element from the now-list is selected as the new current node. If the now-list is empty, a new limit is defined and the later-list becomes the now-list, and the procedure is repeated until the goal is found, or both later- and now-lists are empty; meaning that no path exists\(^2\).

6.3 Hierarchical Pathfinding A*

Different hierarchical methods has existed for a while in paths planning [22], but the HPA* solution was introduces as the first thorough study of the subject in 2004 by Botea and Müller [13], and further enhancements introduced by Jansen, and Buro [23]. The great advantage of hierarchical pathfinding is that the path can be calculated in independent steps; allowing for time slicing without greater effort. It also increases search speed by restricting the search space on lower abstraction levels.

---

\(^2\) The major drawbacks of IDA* are it can not detect repeated states, revisits all nodes each iteration, and opens in left to right manner compared to A* best first.

\(^2\) Hash table with on place for each instance, giving constant access time.

\(^2\) For pseudo code see section A.2 Fringe.
HPA* could be said to work in three steps: pre-processing, searching and path smoothing. The first step creates the graph and its hierarchy. The second step utilizes the graph. Finally, the third step refines the path; making it more optimal and look more natural.

### 6.3.1 Pre-Processing

This phase takes a map in a grid- or polygon structure, and starts by dividing the map into a set of sectors (see Figure 12). After the map has been divided, each section is analyzed and nodes are placed on the section border that can be crossed (see figure 12). Then the sectors are connected through the nodes on the borders (see figure 12). The cost of the internal paths is calculated simply by doing an internal path search.

This is one way of creating a graph without any domain knowledge. It could be done in different ways, but this one is used by Botea and Müller. The next step of the pre-processing is to take the graph to the next hierarchical level. Botea and Müller does this by taking a cluster of NxN sections, and repeating the initial process described above all over again; selecting nodes on the borders (see figure 13), and calculating internal cost by local path searches on the layer below. This process can be repeated until the desired level of abstraction is reached.

### 6.3.2 Searching

The searching in the abstraction can be done by any generic pathfinding algorithm; e.g. A*, RBFS, Fringe, etc. There are, however, some additional steps: first the start- and end nodes has to be inserted at each level of the abstraction; connecting them to the abstraction graph. When the start and end are connected, the search is done first in the top layer, and then by iteratively searching lower layers until the lowest layer has been reached. For each level, the level above restricts the search space by only allowing the lower layer to search within the area corresponding to the layer above. This will limit the amount of nodes that will be needed to expand, and according to the creators of the algorithm, increase the processing speed by several magnitudes. When the lowest layer is reached and searched, a refinement can be done by searching on tile-level from node to node, to get the path on tile-level.

---

23 A normal N value would be 2, giving a square of 4 sectors.
6.3.3 Path smoothening
This is done, as mentioned in the subsection above, to make the path look more natural, and to increase optimality. The process used by Botea and Müller is a very simple approach, but still achieves good results. The smoothening is done by taking the first node on the searched path and check if the Euclidian path to the next node is collision-free. If so, the next, and the next after that, is checked, until the Euclidian path is not collision-free any more, then this process is repeated from the last node that was collision-free.

6.4 Partial Refinement A*
This search method was first introduced by Sturtevant and Buro [14] in 2005. In their presentation of PRA* they use a clique abstraction (see figure 14) to search in. Sturtevant and Jansen [24], however, use PRA* with a set of different abstractions (cliques, sector, radius, line, etc.). The common ground for all these abstraction methods is that all merge locally adjacent tiles (nodes) into a higher abstraction in different ways. By doing this several times, the abstraction will finally end with a tree-structure with the root node containing all nodes that are reachable from each other. The fact that all reachable nodes end up in one node makes it possible to control if a path is existent before starting to do any calculations.

![Figure 14: Illustrations of the creation of the PRA* abstraction. The merging continues until there are only one cell left.](image)

PRA* is very similar to HPA*, and the main difference is that HPA* relies on a graph with the need to connect start and end node. With PRA*, the start and end node can be mapped to single tiles that in turn is the lowest layer in the abstraction-tree. To contain the abstraction-tree demands a somewhat greater amount of memory [15, pp. 4].

Searching in with PRA* can be done in a set of ways. The most intuitive way of searching would be to begin with the connectivity check; checking if the nodes meet somewhere in the tree. If the nodes where connected, one would continue by searching from the level that the nodes first connected and then search downwards, level by level, to the direct tile-mapping. This is one way of doing it, but a few different approaches are suggested by Sturtevant and Buro which only have smaller changes in efficiency to increase optimality. The first change is to, instead of starting to search in the top of the abstraction, start half way through, to balance between time and optimality [14, pp. 4]. The search effort will be somewhat increased because there will be more nodes to search the first pass. A second thing to consider is to allow the search function not only to expand the children of the previous search, but also to expand the closest adjacent nodes of the children [14, pp. 4].

---

24 The node where start and end first meet.
6.5 Minimal Memory Abstraction

This method was developed to, as the name implies, minimize the additional memory needed to store the map-abstraction alongside the map. It adds as little as 3% to the original map size, according to [15]. The method was first mentioned by Sturtevant [15] in 2007, and is similar to both HPA* and PRA*, but has focus on using very little memory.

6.5.1 Abstraction

To create the minimal memory abstraction (MM), the map is divided into a course-grained uniform grid, where each cell has a width between 4 and 16²⁵ tiles (see figure 16), where each cell is called a sector. Then, each sector is considered individually to insert a node at each region, a connected area, (see figure 16). When the regions are marked, the borders between the sectors are walked along to see which regions connect to each other (se figure 16).

Selecting the node location within a region can be done in several different ways, where the simplest is to just place it at the first tile found in the region. Another solution is to measure the distance to all the other cells in the region and selecting the cell with the smallest sum. The approach recommended by Sturtevant [15, pp. 4] is to minimize the nodes needed to expand when searching between the region node and its adjacent regions.

6.5.2 Searching

Searching in MMA* is relatively simple. It is done in a three steps: first the real nodes/tiles are transformed to the abstract layer; then a generic algorithm, like A*, can be used to find the path in the abstracted search space; lastly, the abstract path is refined.

Finding the region-center-node can be done in two ways, either one sacrifices some additional memory to mark the tiles with a reference to the region center node, then one have the region node

Figures 15 and 16: Start and end is found in a common tile at level 6, the search is started at level 3 (l/2), and performed down to tile level.

**Figure 15:** MM abstraction creation; first image shows section division; second image, region node marking; and third image displays region nodes connected.

²⁵ 4 to 16 are the widths that are used in [15].
without any work. If the memory is sacred, one could make a small local search within the sector to find the region node.

Searching in the abstraction after the abstract start and end node is found is done by any standard search algorithm; e.g. A*, IDA*, Fringe etc. Then the hard part is to do just enough work to get a good path out of the abstract path, because just using the abstracted path can result in very suboptimal paths or, in the worst cases, paths that are not traversable (see figure 17 for example).

Figure 17: Examples of three suboptimal solutions and a pretty good solution (the rightmost picture).

It is here the refinement comes in to avoid these kinds of problems. Start by looking at the first image in figure 17, which shows all problems with just using the nodes at a path. The change from the first to the second image is that the first and last node has been dismissed, and start and end is used instead. This drastically improves the quality of the paths and is simple and intuitive to do. The next step is to avoid paths bumping into edges; this problem might even result in paths where the unit gets totally stuck. However, it is simply solved by making a search between nodes, creating a path at tile level for each local path. Now the Path looks quite good but it could be more slimed around corners. To slim the path around corners one can dismiss the last tiles in the tile path and start the next search from there instead. This will increase calculation some, since the paths to search becomes slightly longer, and the last work will be discarded.

MMA* can be seen as a compromise/combination between HPA* and PRA*, as it searches an abstract layer first. Furthermore, it uses a sector based approach as HPA*, but inserts less nodes by just having one node per region instead of having several on the borders. Similar to PRA*, it refines paths by a corridor defined by the layer above in the abstraction [14].

6.6 Triangle search

In recent times, pathfinding solutions that abstract the search space into a triangle mesh has become a promising field of research; three of which will be described in this section. The first solution that will be described was first mentioned in 2005 by Kallmann [25]. Later, Demyen and Buro further expand the subject when they present two variations of the approach in 2006 [26]. Earlier, pathfinding in simple polygons and funneling algorithms have been investigated by Hershberger and Snoeyink [27].

Pathfinding in a triangle-based abstraction is done, as in most other approaches, by translating the start- and end points into the context of the abstraction; here, this is the process of locating the triangles in which the start- and end points reside. Even though the triangle-based abstractions discussed in this section have a lot in common with uniform grid-abstractions, it is a much more complex task to locate individual abstraction cells based upon real-world coordinates. The reason for this is that, unlike the cells in uniform grids, triangles can take any size and angular shape, which makes the transform from a coordinate pair into a triangle somewhat harder than a simple change in base-coordinate systems. The method used by Kallman for solving this shortcoming is a simple triangle traversal; starting at some pre-defined triangle in the abstraction, and then, triangle by triangle, traversing the abstraction along the direction vector between the previously mentioned triangle and the desired coordinates, until the triangle containing the coordinates is found (see figure 18). This method can be very time-consuming, should the pre-defined triangle and the actual sought triangle be
located on opposite sides of the map. In an attempt to remedy this problem, Demyen and Buro superimpose a larger grid onto the triangle-abstraction. By mapping each sector in this grid to the triangle occupying its center – if a triangle is large enough to cover the center of several sectors, all sectors are mapped to that triangle – it is possible to restrict the afore mentioned triangle traversal to the maximum span of a single sector.

When considering the actual pathfinding algorithm, the approach suggested by Kallmann sees the centers of each triangle’s traversable (i.e. not bordering a collision area) edges as nodes. Interconnected these nodes form a grid that can be used by any generic pathfinding algorithm; estimating the distance to the end-node (h-value), and incrementing the cost of the path (g-value) for each node the path contains. As can be seen in figure 19, this solution suffers if path optimality is a matter of greater importance. The solution does indeed generate the most optimal path that can be found through the provided nodes, but since these nodes can have very peculiar positions there are no guarantees that the found path is optimal, even if funneling is used to optimize the path within the sequence of triangles.

Demyen and Buro handle the problem of sub-optimal paths slightly differently than Kallmann. As can be seen in figure 20, the path generated by simply passing through the center of the nodes is very different when compared to a funneled path through the same sequence of triangles. Which is why Demyen and Buro introduce new methods for calculating the g- and h-values [26, pp. 2]; making the estimates more closely resemble that of the funneled path.

Furthermore, Demyen and Buro are not satisfied with just looking at the first path generated by the solution, as there is no guarantee that it really is the most optimal path that can be found. Because of this, they generate a triangle-path, funnel it, and then iteratively generate additional paths which are funneled in order to see if the path generated by the latest iteration is more optimal than those that precede it. This approach has the adverse effect of requiring all possible paths between the start-
and end points to be examined in order to guarantee that the most optimal path has been found. This feature, however, gives the solution traits of an anytime algorithm, as it at any time — past the first iteration — can be interrupted, and still produce a path. It might not be the best path, but it is a path.

The funneling algorithm used in both solutions could be done in linear time [27]. And to further enhance the result of the funneling Demyen and Buro introduces the concept of radius on objects, as can bee seen in figure 21, to give a more usable and natural looking path. This, however, does not increase the degree of time complexity.

By introducing additional pieces of data stored in each triangle, it is possible to take unit-size into consideration when searching the abstraction for a path; thus not considering triangles too small for some object to pass through. To make this possible, for each edge pair in each triangle one must store the size of the largest object that could pass through the triangle via those edges (see figure 22).

As mentioned earlier, Demyen and Buro also suggest another triangle search method in Triangulation Reduction A* (TRA*). This approach identifies simpler, but common, structures within the abstraction (threes, loops, and graphs) which it uses to greatly optimize its performance, with the cost of additional information about the abstraction being stored.

The abstraction used by TRA* is created by labeling each triangle as type 1, -2, or -3. First, all triangles with only one unconstrained edge are labeled as type 1 triangles. Then, iteratively, each triangle bordering a type 1 triangle, and having one constrained edge, are also labeled as type 1 triangles, until no more type 1 triangles can be found. When all triangles of type 1 has been identified, all triangles that have no constrained edges, and doesn’t border any triangles yet labeled are denoted as type 3 triangles. Lastly, all unmarked triangles are labeled as type 2.

Figure 21: The figures display (from left to right): triangle path, tunneling without considering unit-radius, and tunneling with unit-radius considerations.

Figure 22: An object trying to pass from the parent-triangle to the next-triangle has to have a diameter lesser then the height of the triangle it is trying to pass.

Figure 23: Creation of the TRA* abstraction.
As can be seen in figure 23: type 1 triangles form trees; type 3, when interconnected, form graphs; and triangles of type 2 form the ‘loops’ that connect the other two structures together.

By making use of the above mentioned structures, TRA* minimize the size of the graph that must be searched, which is a great advantage as threes and loops are not as costly to traverse. The creators of the solution specify a set of special cases that should be checked for in order to get the most out of TRA* [26, pp. 4]:

1. The Start- and end triangles are members of the same type 1 three, in which case one must only perform a three traversal to get the path.

2. The start- or end triangle is the root of a three containing the other triangle. Here, one need only walk from the triangle within the three to its root.

3. The start- and end triangles are on the same type 2 loop. The loop is traversed in both directions, so that the shortest of the two can be selected as the more optimal path.

4. The start- and end triangles are on the same corridor. One path is found by walking through the corridor, and a second path is given by searching the connected type 3 graph. The shorter of the two is selected as the more optimal path.

As with TA* the width of units can be considered. Another likeness to TA* is that more optimal paths could be found by iteratively re-searching the abstraction.
7 **Empirical Study**

This section describes the choice of algorithms for the empirical study, the implementation specifics for these algorithms, and finally it details on the experiment setup.

### 7.1 Choice of Algorithm Rationale

The criterion for choosing the algorithms examined more closely in the empirical study goes back to the problem definition, which can be found in section 3. In that section, the constraint categories are identified: spatial, temporal, and path optimality. Where the first two are hard constraints and the latter is considered somewhat looser. The spatial constraint can here be divided into three categories for measurement: initial-, storage-, and peak memory usage. *Initial* memory usage being the memory needed to load the algorithm and its required facilities into memory. The *storage* category is the amount of space that is needed for the world abstraction; this criterion is somewhat looser because the game cartridges can contain up to 256 Mb. Finally, *peak* is the maximum memory demand during runtime; it is constrained by the 4 megabytes of system memory, just like the *initial* memory usage. Temporal constraints demands efficiency, due to the limited processing capabilities of the DS, furthermore if time splicing could be introduced to deliver partial paths quick, it would be beneficial. Optimality is the one thing that could be sacrificed, however not at any cost. This can be summarized to a set of criteria: initial memory usage, peak memory usage, storage memory usage, efficiency, time splicing ability, and optimality.

As can be seen in sections 4, 5 and 6, there exist a wide variety of algorithms, abstractions and combinations of these ideas, to address the pathfinding problem. Table 1 shows some variants, and an evaluation of them, based on the above mentioned criteria.

**Table 1:** Table displaying a set of search strategies; an evaluation score for them; and the choice of algorithms marked with a * in the index column. The grading of the algorithms is done after a thorough literature survey on the solutions presented, and with minor empirical experience of them. The scale is - - to + +, where - - is really bad and + + is exceptionally good, only one – or + is just better or worse than average.

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Speed</th>
<th>Initial</th>
<th>Peak</th>
<th>Storage</th>
<th>Optimality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A* grid</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>A* POV</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>3</td>
<td>A* quadtree</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>Fringe grid</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>Fringe POV</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>6</td>
<td>Fringe quadtree</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Triangle search</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>TA*</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>9</td>
<td>TRA*</td>
<td>++</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>10</td>
<td>MM</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>11</td>
<td>HPA*</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>12</td>
<td>PRA* (cliques)</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

When looking at Table 1, and the selected algorithms\(^{26}\), one notices that the selection is not done just according to who has the most plus signs. The reason for this is to increase the variety of the algorithms being compared in this study. It is desirable to have solutions using grids, waypoints, triangles and hierarchies so that a wide picture could be drawn.

\(^{26}\) A* grid, A* POV, Fringe grid, Fringe POV, TRA*, MM.
POV is selected because Pixelknights previously used POV in *Origo: Prolog*, so this will give them a reference point. Furthermore, navigation meshes has been a commonly used approach in games. The grid solution is important to include for getting a reference point to other studies made [13] [15].

The reason for having both Fringe and A* is to see if it is possible to get significant improvements by only changing the generic algorithm searching the different abstractions. Björnsson et al. [21] claim that Fringe is up to 40% faster than A*. The choice to have both Fringe and A* for both grid and POV is to further increase the experience of these generic algorithms. Furthermore, the cost for running them on both is minor. If any significant difference between Fringe and A* is noticed this could have effect on more complex algorithms in the future, by changing the commonly used A* to Fringe.

MM and TRA* are selected because their inner workings function in totally different ways. TRA* use triangles, while MM is a hierarchical search method. To select TRA* before TA* and simple triangle search is simple because TRA* has all that these have, plus more, with the only disadvantage of slightly more memory usage. Choosing MM over HPA and PRA was a harder choice, but as PRA is said to use large amounts of memory [15, pp. 4] it was not really relevant. HPA might have been just as good a choice, but because the large benefit of HPA is in large search spaces the choice fell on MM, as it minimizes memory usage while having the hierarchical advantage to some degree. Further interesting things to consider in the evaluation are; MM can time splice, simply allowing the first part of the path to be delivered quickly; while TRA* can take unit size into account when searching.

### 7.2 Implementation Specifics

The implementations of A* use a C++ STD map container to store the closed list, giving a logarithmic access time to all objects in the list. The unique identifier for the map is composed by the specific node’s x-value added with its y-value, multiplied with the number of columns. To the open list an STD vector is used, which is sorted by the STD sort function, and searched by STD find.

The Fringe implementation follows the specification given in its paper of origin [21] to most details; except in the tile case, where it uses a map for the cash instead of a perfect hash. The addressing of the map is just like in the A* case of the map.

Specific for the POV implementation is the manner to which nodes are connected. To avoid need of string pulling, the implementation tries to connect to all nodes while checking for intersection with all borders. This will have clear complications for the speed, since this operation has a time complexity of $O(n^2)$. However, the alternative with locality tables creates somewhat strange paths if string pulling is not used.

#### 7.2.1 TRA* Specifics

Since the authors of TRA* doesn’t go into the specifics of the, it is not guaranteed that the implementation used in this paper is exactly the same. Some liberties have been taken when attempting to approximate the behavior of the algorithm. However, the main features should be equal. Below follows a description of the explicit differences between the original TRA* solution, and the solution used in this empirical study; where, among other things, it is described how details about TRA* not explained by its creators have been handled.

First out, triangle measurements has been compromised due to time limitations. Removing triangle measurements gives only insignificant gains in speed\(^{27}\), lessening memory need by a few bytes per triangle\(^{28}\), but most importantly it removes the possibility to consider unit size.

---

\(^{27}\) The speed gain comes from a few if-statements not having to be evaluated.

\(^{28}\) The memory gain would more exactly be 3 measurements per triangle and a measurement associated with each connected triangle.
An additional, apparent change is that the implementation used here gives no room for looking at additional, more optimal, paths. This might jeopardize path optimality a bit, but on the other hand, all the measurements in [26] were generated using only one iteration.

In the description of the triangles in [26] it is mentioned that they have reduced the triangles to an average size of less than 183 bytes. This size is not close to what our implementation ends up with, even if the triangle measurements where added, it would not be close. With all the information needed, the triangle size ended up at 56 bytes, and this is not optimized in any way.

The most interesting pieces of data stored in the triangle implementation are type, loop index, connected triangles and connected triangle distance. In addition to this, the triangle contains information on adjacent triangles and constrained edges. The type-variable describes the type of the triangle; 0 = non-traversable triangle, 1 = tree-triangle, 2 = loop-triangle, and 3 = graph-triangle. Tree-triangles connect to a root of type 2, if any. Loop-triangles create individual loops and connect between level 3 triangles. Graph triangles connect loops to form a graph, which is searchable with e.g. A*. The loop index is a unique identifier of a tree, a loop or a graph. As a complement to the connected triangles information contained in each triangle, the triangles stores the distance between their own center-points and the center-points of the connected triangles, which is tunneled in the preprocessor.

With the triangle structure described above it is possible to check for path existence in constant time. This check is done with the help of the loop indices of the triangles connected to the triangle of origin. This check works on the fact that all larger structures in TRA* (trees, loops and graphs) know what other structures they are connected to. That is: each tree knows which loop, if any, it is connected to; each loop knows which graph, if any, it is connected to; and no graph is connected to any other graph, as they then would be the same graph. For instance, if the start- and end triangles are both members of tree structures, one would first check if they were members of the same tree, then if their trees were connected to the same loop, and lastly, if those were connected to the same graph structure. These are the only steps required; if the answer is ‘yes’ at any step, there is at least one existing path between the start- and end triangles, otherwise no path exists. The same steps can be permuted to every possible combination of start- and end triangle structure membership.

7.2.2 MM Specifics

The implementation of MM is to the best of our understanding very similar to the algorithm created by Sturtevant [15]. However, one change has been introduced to the solution used in the empirical study. MM searches from node to node and then it cuts the last part of the local path and thereby starts a few tiles back in the path with the next search, this will give a nice, natural looking path. The change introduced is that no cutoff is made, and if the path from one center to the next on the path is collision free no search is done. The implication of this is that the path will be less optimal, in varying degrees, depending on the size of the sectors. The benefit of the change is that low level searches can be held to a minimum, thereby optimizing the solutions temporal performance.

To locate the first abstract nodes, a breath first search is done, which uses an STL queue and an STL map. However, the fastest solution would have been to use extra storage along with the tiles to address its region node directly, but this was dismissed due to the extra memory usage it would introduce.

Search in the abstracted node-network is done by Fringe-search, which has shown to be faster than A* in search spaces where the open list of A* grows large.

Searches on tile level between region nodes is done with an A* implementation. Using Fringe would probably not have given the gains in this case because the search space between two region nodes is quite small; i.e. will not generate a large open list to store and sort.

29 Measured with sizeof in C++.
7.3 Experiment Setup

To test the algorithms, a set of random maps where created by inserting random sized blocks into a map array, until the blocks covered a certain percentage of the map. 30, 40 and 50 percentage levels were used in this study; see figure 24. These maps where preprocessed for each algorithm to produce the searchable abstractions required for each solution.

Figure 24: Example maps: 32x32 tiles, with a tile size of 16x8 pixels, 30 percent non-traversable and 50 percent.

These maps where created with different sizes and with different block width and -height settings, with a maximum of 30 tiles on block width and -height. All maps used have a size of 512x512 tiles; this is somewhat larger than what is relevant to Origo DS, where 128x128 represent a large map. Despite this, the size 512x512 is used because it is the size used in many other studiers [13] [14] [15]. Furthermore, this size was needed to get some distinction between the algorithms, as smaller maps would generate results of such small values that the differences would be hard to measure. The reasons for creating random maps are two; no algorithm should have level design benefits when searching; furthermore, there were no time to design organic maps, and no preexisting maps where available.

The computer used as a testing rig had a 2.6 GHz Pentium 4 processor, with 512 MB ram. The machine ran Windows XP SP 2. Furthermore, the environment to run the tests in, and the search algorithms themselves, are all compiled with Microsoft Visual Studio express 2005, optimizing for speed, in release mode.

To avoid freak values, each search algorithm was run on 20 different maps: six 30-, six 40- and eight 50 percent density maps. Paths with similar lengths was collected in spans of 500 pixel units, and averaged over, to create a smooth graph with a feasible amount of data points. For each map, 100 random paths were searched, giving 2000 paths for each algorithm. These paths were selected and saved on file in advance, to ensure that all algorithms got equally complex paths to search. Further, all points were ensured to be on traversable ground.

The measurements collected are initial memory usage, peak memory usage, time in ms, and path length in pixels (it is important to note that path lengths are not measured in tiles). The initial memory usage is the amount allocated for the abstraction when loaded from file, plus the data structures that the algorithm needs; this is measured so that the balance between memory usage and time can be examined. Peak might be one of the most important values; it shows if the algorithm at all will be able to fit in memory in the current configuration. Time is measured so that the efficiency of algorithms can be compared. Finally, the path length is a tool for examining the path optimality of an algorithm, and to have something to plot the graphs against. The optimality is measured as the path length divided by A* POV path length. Further, all algorithms are given the possibility of one milestone; MM sets time to first delivery; POV implementations sets connection time; and TRA* sets time before tunneling.

---

30 A program in which it was possible to request a set of random paths, set memory measurements, and to visualize the paths with a GUI created in SDL.
8 Results

In the following subsections the data collected in the empirical study is presented, and explained. In Section 8.4 conclusions about this data are drawn.

Most measurements in this section are presented relative the length of the path that generated the data point. To avoid noisy graphs, the data points in the illustrations represent an average, rather than individual measurements. Each data point represents an average of all collected measurements between the point in question and the point that precedes it. The points are evenly spaced, with a distance of 500 length units between each point.

8.1 Time to complete

The collected data suggests that the time required to find an optimal path\textsuperscript{31} is very much dependent on how long that path will be. This holds true for every pathfinding solution tested, although the degree of which this dependence factor into the search time vary from algorithm to algorithm.

![A*-Grid Time](image-url)

Figure 25: The time A* requires when generating the most optimal path it is capable of when searching on a grid abstraction. The data points suggest a trend resembling a standard power function.

\textsuperscript{31} As optimal as allowed by the algorithm in question.
Figure 26: The time Fringe requires when generating the most optimal path it is capable of when searching on a grid abstraction. The data points suggest a trend resembling a standard power function.

Measurements from the two algorithms searching directly on a grid abstraction (Fringe and A*) suggest a clear trend present in both sets of data points. Despite the great similarity in the trends of how much time the algorithms need before being able to give a response, Fringe is quite much faster than A*. The difference in time to completion between the two solutions rises exponentially relative the path length. Much of the reason for why Fringe surpasses A* can be traced to the fact that Fringe has no need for sorting the nodes in its open list, whereas sorting translates into a cost in A* that grows with the amount of nodes it must expand – which, in turn, is affected by how long the resulting path is. These results are comparable with those described by Björnsson et al. [21]; where Fringe has a performance advantage over A* of up to 40%.

Figure 27: The time MM requires when generating the most optimal path it is capable of. The data points suggest a trend resembling a standard power function.
As can be seen in Figure 27, the time to completion for MM rise relative to the path length until paths of 6000 length units, where the curve seems to flatten out, resulting in a logarithmic behavior. This is, however, believed to be an artifact of the world maps used more than of the pathfinding solution. Since all world maps used are of the same bounded size, it is reasonable to assume that the longer paths are of a higher complexity than shorter paths; zig-zaging across the map. For such paths to be possible the world map must be of relatively high density and complexity; which means fewer high-level sectors are needed for the world abstraction. This, in turn, means that the point at which the algorithm has expanded all high-level sectors is reached earlier than on less dense maps. Beyond this point, the time it takes to find a path is relatively independent of the length of that path, as the only additional processing time comes from the amount of low-level tiles expanded, which amounts to a small percentage of the total time to completion (can be seen hinted at in Figure 28). MM is heavily based on Fringe, which further supports the assumption that the trend is an exponential- or standard power function of the path length, rather than a logarithmic function. If the world being searched is infinitely large, the time to completion of MM will probably grow exponentially (or as a power function) relative the path length. In most bounded worlds, however, the time to completion will probably behave as in figure 28; rising exponentially with the path length, to eventually hit a “roof”\(^{32}\) where the curve will converge towards a horizontal line.

This specific implementation of MM makes use of a Fringe based search algorithm, but the solution must seldom expand as many nodes as pure Fringe because of the way it structures and layers the world abstraction. Collected measurements suggest that, on average, about 91\% of the time required to generate a path is spent on searching the higher layer of the hierarchical world abstraction. This opens for the possibility of time splicing, since every individual part of the search on the lower layer can be searched and delivered when, or if, it is needed. This opens up for allowing the algorithm to spend more resources on the lower layer, which would increase the overall optimality of the paths delivered.

---

\(^{32}\) Where this roof is located depends on several factors; e.g. the size of the map, how many traversable sectors there are, and the ratio of how many sectors must be searched on a lower level.
When analyzing the graph in Figure 29, the conceptual complexity of TRA* has its obvious payoffs; this solution is by far the fastest of all those tested in this thesis. The data suggest an almost linear trend-curve. The size of the world maps used for testing the solutions put limitations on how complex the maps can be, which makes it hard to predict how the solution would behave if the complexity rose significantly. A viable guess is that, as the number of level-3 nodes grow, the solutions take on some more of the characteristics found in the pure Fringe- and A* solutions; making the trend a slow-starting, standard power function.

Figure 29: The time TRA* requires when generating the most optimal path it is capable of. The data points suggest a trend resembling a linear function.

Figure 30: The time A* requires when generating the most optimal path it is capable of when searching on a POV abstraction. If disregarding the first few data points, the trend is a standard power function.
The time Fringe requires when generating the most optimal path it is capable of when searching on a POV abstraction. If disregarding the first few data points, the trend is a standard power function. The fringe- and A*-based POV implementations exhibit a slightly different behavior when compared to the rest of the algorithms (see Figure 30 and 31). In the implementations used, the start- and end points of the wanted path are connected to every, from them, visible node. This is done to include the start- and end points into the graph of nodes that makes up the abstraction; making a search at all possible. The results of this overhead are clearly noticeable in the measurements, and the time required for the actual search is only a fraction of this initial overhead. The peculiar behavior of the graphs, where the first few data points show a declining trend, can also be explained by the initial connection overhead. Chance would have it that the paths of lesser length can be found on world maps containing more nodes and borders, resulting in the initial connection procedure taking longer time. Even if the overhead of the POV based solutions was greatly reduced, or even eliminated altogether, the time solemnly required for the search is still magnitudes more than that required for MM and TRA* (as can be seen in Figure 30 and 31, there is a difference of about 250 ms between the smallest and the largest data point).

An important performance measurement of pathfinding algorithms is the time they require to find that there is no path between the start- and end points of the search. If it is assumed that these points are both located on traversable ground, the only possibility for there not being a path is if the areas the points can be found in are disjoint. All nodes within at least the smallest of these areas must be expanded before grid- and POV based pathfinding solutions can discover that the areas are disjoint from each other. If the areas are large, this process is very time consuming; explaining why the average time, shown in Table 2, for these solutions to discover nonexistent paths is very high. The world abstraction used by TRA* gives the solution a great advantage in that it can discover non-

<table>
<thead>
<tr>
<th>Nonexisting path discovery</th>
<th>Algorithm</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*-Grid</td>
<td>1,466.13</td>
<td></td>
</tr>
<tr>
<td>Fringe-Grid</td>
<td>612.52</td>
<td></td>
</tr>
<tr>
<td>A*-POV</td>
<td>655.40</td>
<td></td>
</tr>
<tr>
<td>Fringe-POV</td>
<td>637.51</td>
<td></td>
</tr>
<tr>
<td>MM</td>
<td>8.66</td>
<td></td>
</tr>
<tr>
<td>TRA*</td>
<td>3.21</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: This table describes the average time each of the tested pathfinding solutions requires for detecting that the wanted path doesn’t exist.

33 It is fully possible to reduce this overhead, for instance by using sectors.
existent paths in constant time\textsuperscript{34}, the results of which are clearly visible in Table 2. Despite this, this aspect of the solution is quite much slower than expected. The way TRA* is designed, it should not take more than 1 ms to discover that a path doesn’t exist\textsuperscript{35}. The reason for why the data suggest otherwise, we believe, is due to a flaw in the implementation – which should be easy to eradicate, given time. The ability to virtually instantaneously discover that a path doesn’t exist is a big advantage for any pathfinding algorithm. This doesn’t, however, exclude the possibility to have other resources than the pathfinder itself check for such occurrences; e.g. by keeping track of disjoint areas, and not query the pathfinding resources if the endpoints are not within the same area.

8.2 Memory usage

The amount of memory every search of a given algorithm must, at minimum, have allocated in runtime is denoted as the initial memory usage. This initial value is excluded in the graphs contained in this section, but is mentioned as a total average in the figure description for each graph. The reason for this exclusion is that the different world maps used in this study generate different values, and inclusion in the graphs brings erratic behavior that makes them hard to analyze.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{A*_Grid_Memory.png}
\caption{The graph depicts the dynamic memory requirements of A* when using a grid-based abstraction. The trend is a standard power function. The average initial memory usage of 262181 bytes is excluded from the graph for clarity reasons.}
\end{figure}

\textsuperscript{34} This trait is explained in section 7.2.1.

\textsuperscript{35} On the computer that served as testing rig, that is.
Figure 33: The graph depicts the dynamic memory requirements of Fringe when using a grid-based abstraction. The trend is a standard power function. The average initial memory usage of 262180 bytes is excluded from the graph for clarity reasons.

The initial memory usage of the basic grid based Fringe- and A* solutions mainly corresponds to the size of the grid abstraction, where every tile is represented by an 8 bit Boolean\(^{36}\). Memory allocated above the initial bound is directly proportional to the amount of nodes/tiles the algorithms expand during runtime.

Figure 34: The graph depicts the dynamic memory requirements of MM. The trend is linear, but the standard power function hinted at in the first half of the data points is more likely. The average initial memory usage of 518708 bytes is excluded from the graph for clarity reasons.

The graph representing the memory usage of MM in figure 34 shows a great resemblance in behavior to the graph representing the algorithm’s required time to completion in section 8.1. The amount of memory used during runtime is directly related to the amount of nodes the algorithm must process, making the discussion about the time to completion of MM in section 8.1 applicable to how the algorithm utilizes memory relative to path length. By following this discussion, it is reasonable to assume

\(^{36}\)The size required for the abstraction can be reduced by a factor of eight, as a Boolean value can be represented by a single bit.
that, in an unbounded world, the memory usage of the algorithm will follow the trend created by the first few data points in the graph.

![TRA* Memory](image1)

**Figure 35:** The graph depicts the dynamic memory requirements of TRA*. The trend, for this limited set of data points, is linear. The average initial memory usage of 243867 bytes is excluded from the graph for clarity reasons.

The reason for the linear trend in how TRA* allocates memory in runtime is closely related to the design of the solution; as with the case of its temporal behavior. The component of TRA* requiring the most memory in runtime is its level-3-node processing facility, but since the usage of this component is minimized, it has little influence on the solution’s overall temporal behavior.

![A*-POV Memory](image2)

**Figure 36:** The graph depicts the dynamic memory requirements of A* when using a POV-based abstraction. The trend is a standard power function. The average initial memory usage of 262819 bytes is excluded from the graph for clarity reasons.
The graph depicts the dynamic memory requirements of \( A^* \) when using a POV-based abstraction. The trend is a standard power function. The average initial memory usage of 252067 bytes is excluded from the graph for clarity reasons.

The POV implementations of Fringe and \( A^* \) exhibit very much the same behavior as their grid-based counterparts. The only difference can be found in the degree to which the memory usage corresponds to the path length. The POV solutions require far less memory since they perform their searches on an abstraction containing fewer nodes.

8.3 Path optimality

The path optimality of a solution is measured by how the paths it generates compares to the most optimal counterparts of those paths. Of the solutions examined by this thesis, the POV solutions generate the most optimal paths, which is why these are used as basis when comparing the solutions.

The reason for why the grid based solutions (MM included) are of lesser optimality can be traced to the fact that they have fewer degrees of freedom than waypoint based solutions. A distance that one of the POV based algorithms can cover by following a single vector might require algorithms using a grid abstraction to zig-zag along that same vector. Another difference, which is of no favor to the solutions using a grid abstraction, is that paths generated on a grid must pass through the centers of the tiles they cover, while the POV solutions has no such restrictions.

Compared to the grid-based abstractions, TRA* performs well when path optimality is considered. It should however be noted that these results are a product of a design choice aimed at optimizing the solution’s temporal performance. If a slight sacrifice in speed is acceptable, TRA* is fully capable of delivering paths as optimal as those generated by the POV-based solutions.
8.4 Analysis

The data presented in the preceding sections, in many ways, favor TRA*. None of the other solutions tested in this study can deliver a path as fast as TRA*. The trend suggested by the collected data points gives a hint towards the solution being fully capable of supporting larger and more complex world maps. Another temporal benefit of the solution is its capability of quickly being able to decide if a path exists or not, which prevents spending processing resources on a task that will ultimately be fruitless.

When regarding the optimality of the paths delivered by the solutions, the differences present in the data are of small significance; especially when considering the relatively loose path optimality constraint of the domain. And as mentioned earlier in this paper, TRA* can be implemented in such a way that, on the expense of some performance, it is capable of delivering paths as optimal as the POV-based solutions.

There is one situation in which TRA* might not be the preferable choice: when the world map is already described as a grid structure; which is certainly not unprecedented when the world is a game map, which is the domain of this thesis. In such cases, the overhead of keeping the CDT-abstraction in memory can be a high price. Even though TRA* outperforms MM on many fronts, the latter – and other grid-based solutions – might be preferable when the original world map is a grid structure, as the lowest abstraction layer then is already present, and need not be made available as a separate structure.

As a side note: evidence support the statement that the Fringe algorithm is as good as, or in some cases even superior to, A* when it comes to speed. In the light of this, Fringe might be a viable alternative as a component in more complex solutions; e.g. TRA*, MM, PRA, etc.
9 Real World Connection

Of the pathfinding solutions tested in the empirical study, TRA* was selected as the candidate most fitting for further adaptation to the Nintendo DS. The main reason for favoring TRA* was its proficiency for finding paths faster than the other solutions were capable of.

Each triangle in the CDT-based abstraction used in TRA* require much more memory than any other tested abstraction, but the total number of triangles needed to represent a world is not as dependent on the size of the world, as it is on the structure of the world itself. If structures that generate large amounts of triangles are avoided, there is no guarantee that larger worlds will generate more triangles than smaller ones. That being said, the larger the world is, the greater is the risk for occurrences of complex objects that require a larger amount of triangles for representation. Since it is possible to represent triangles using less memory than first stated by Demyen and Buro [26], and that the amount of triangles can be implicitly controlled by how the world is designed, the spatial requirements of TRA* are not as severe as first thought. As the world maps grow, TRA* might actually require less memory than other solutions where memory usage is directly connected to the size of the world. As a trivial example: if one magnifies a world tenfold, without changing the relative structure of it, a grid would use ten times more memory for representing it; whereas the abstraction used by TRA* would remain largely unchanged, since the layout of the triangles would remain the same.

By design, TRA* is capable of always delivering the optimal path between two points. This is, however, a trait of the algorithm that didn’t get translated into the version tested on the DS. As mentioned in section 3, path optimality can be sacrificed in benefit of speed, which is favorable when the constraints of the domain are considered. As a result of optimizing the algorithm for speed, paths might not always be optimal. But since the constraint governing path optimality isn’t hard, as opposed to the other constraints, this behavior is accepted.

9.1 Path cache

In most RTS games larger group movements are usual. This means that several units will request very similar paths within a very small timeframe. Instead of performing what is essentially the same search several times, it is possible to reuse an already calculated path by saving every untunneled path generated by the pathfinding algorithm in a cache. This is possible since the “corridor” of triangles that makes up an untunneled path can serve as a highway between the two end-triangles. If a path is already in the cache, a unit only needs to request a tunneling of that path, which is done with linear time complexity, instead of performing the entire search a second time. Since memory is an issue, there must be an upper bound on how many untunneled paths the cache can contain, which can be solved by having its elements prioritized; rising the priority of a path every time it is used, and removing the element of lowest priority when a new path is introduced and the cache is at its limit.

Checking if a path is stored in the cache is done by examining every element in the cache, to see if the end-points of the path are contained in the end-triangles of the cached, untunneled path. It is important to note that the paths in the cache can be stored without any information about direction. This is useful in such cases where units travel back and forth between roughly the same spots, since these two paths need only be stored once in the cache.

---

37 The same argument can be transferred to other solutions; e.g. the size of POV is dependent on the number of convex corners in a world, and not the size of the world, per se.

38 As allowed by the abstraction. Any variance between the triangular abstraction and the world will be translated into the path optimality.

39 A “corridor” of triangles that has yet to be refined by a tunneling algorithm is considered untunneled.

40 E.g. units patrolling an area, units transporting resources between some storage depot and the area of harvest, etc.
In the implementation used in this thesis, no paths spanning two triangles, or less, are stored in the cache. The time it takes to locate these paths in the cache rivals the time it takes the pathfinding algorithm to actually generate the paths.

9.2 Experiment setup

In order to collect real world data, the solution was compiled with devkitPro\textsuperscript{41}, and run on a Nintendo DS Lite. Two different world map sizes were used (128x128 and 64x64)\textsuperscript{42}, each with three different density settings (30-, 40-, and 50 percent impassable ground). These maps has been generated with the same tools and parameters used in the empirical study, as to provide comparable data. Finally, 1000 random paths were searched on each map, resulting in a total of 6000 paths. Since TRA* is capable of detecting nonexistent paths in constant, near negligible, time, endpoints not resulting in traversable paths are not included in the data presented in the following section.

No data about the solution’s spatial behavior was collected in the real world scenario, as neither internal, nor external, changes that would result in different results than those generated by the empirical study in section 8.2 has been introduced.

During data collection, the cache described in section 9.1 was not used; since the data should be as closely comparable to that of the empirical study as possible, however miniscule the changes might be.

Due to some problems when running the code on the hardware, and severe lack of time at the end of the project, the tunneling was disabled in the real world scenario. However the time complexity of tunneling is linear to the amount of triangles in the path, and corresponds, on average, to 1.69% of the total search time required.

Some of the above described differences in the systems run in the empirical study and the real world scenario result in different behavior between the systems when path optimality is considered. Paths generated by the solution used in the real world scenario are less optimal, which should be taken into consideration when reading the following sections.

9.3 Results and conclusion

The results gathered when running the pathfinding solution on the Nintendo DS are displayed in Table 4. When analyzing the average time of searching all paths of each map, one can see that processing time is a function of the maps’ complexity, rather than their size; which has been mentioned as a trait of TRA*. Although the world maps generate different results, these differences are very small. A delay of 10-20 ms between an, by the player, issued order and unit movement is impossible to notice. If this average search time is small enough is impossible to say without knowing the processing time requirements of the systems the pathfinding solution will be collaborating, and competing, with when operating within the context of a video game. The results are, however, promising.

When considering the average time it takes to search any random path generated on the test maps, the peak values gathered are somewhat anomalous. In fact, each map generates a few peak values that are far above all other collected. From the collected data, it is not possible to establish a connection between these peak values and the number of triangles in the paths. Because of this, it is not a stretch to conclude that these anomalous peak values are the product of some bug(s) within the code. If they indeed stem from erroneous code, and are not a trait of the solution in itself, it should be possible to locate the areas of problem, and correct them. This would, however, not have any noticeable effect on the average search times, as the peak values are rare enough to play only a miniscule role in the equation.

---

\textsuperscript{41} The standard homebrew environment using the g++ compiler.
\textsuperscript{42} The sizes correspond to the large and medium maps used in Origo DS.
As seen in Table 4 TRA* produces paths quickly, this is however with the processor dedicated to searching paths, so if the times is small enough when lots of paths is requested and other AI functions has to be calculated, is still a question for future work.

The memory usage of the solution is of quite larger concern than its temporal traits. The abstraction is, despite what one might first think, not the larger culprit, as the number of triangles required can be controlled by clever level design. This, in itself, is nothing of extreme importance, as the memory required by each triangle can be heavily optimized to improve on the measurements collected in the empirical study (Section 8). What are of concern, however, are the memory requirements of the code by itself. TRA* is, structurally, very complex; requiring many different types of algorithms, that use different libraries, to cooperate. When compiled, the .nds file, containing the solution and one world map, end up at approximately 300 KB, which is well beyond the maximum of 100 KB that was the goal. This must, however, be taken into the context of an entire video game. There is a big possibility that much of the resources (e.g. the STL-libraries) used by TRA* will be used elsewhere in the code for the game.

<table>
<thead>
<tr>
<th>Map*</th>
<th>Time (ms)</th>
<th>Nr paths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Peak</td>
<td>Average</td>
</tr>
<tr>
<td>64x64-30-1</td>
<td>49</td>
<td>11</td>
</tr>
<tr>
<td>64x64-40-1</td>
<td>48</td>
<td>10</td>
</tr>
<tr>
<td>64x64-50-2</td>
<td>82</td>
<td>17</td>
</tr>
<tr>
<td>128x128-30-1</td>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>128x128-40-1</td>
<td>64</td>
<td>11</td>
</tr>
<tr>
<td>128x128-50-1</td>
<td>78</td>
<td>13</td>
</tr>
<tr>
<td>Average</td>
<td>63.00</td>
<td>11.83</td>
</tr>
</tbody>
</table>

Table 4: Data collected on Nintendo DS. (*Map, NxN represents the map size in tiles, the second number is the density of the map in percent non traversable, third number is a index to make map name unique)

As seen in Table 4 TRA* produces paths quickly, this is however with the processor dedicated to searching paths, so if the times is small enough when lots of paths is requested and other AI functions has to be calculated, is still a question for future work.

The memory usage of the solution is of quite larger concern than its temporal traits. The abstraction is, despite what one might first think, not the larger culprit, as the number of triangles required can be controlled by clever level design. This, in itself, is nothing of extreme importance, as the memory required by each triangle can be heavily optimized to improve on the measurements collected in the empirical study (Section 8). What are of concern, however, are the memory requirements of the code by itself. TRA* is, structurally, very complex; requiring many different types of algorithms, that use different libraries, to cooperate. When compiled, the .nds file, containing the solution and one world map, end up at approximately 300 KB, which is well beyond the maximum of 100 KB that was the goal. This must, however, be taken into the context of an entire video game. There is a big possibility that much of the resources (e.g. the STL-libraries) used by TRA* will be used elsewhere in the code for the game.
10 Conclusion and Discussion

The amount of different pathfinding solutions available to choose from is a group that could almost qualify for being countably infinite. Because of this, it can be presumed impossible to test them all. This paper has attempted to provide a comparison of a wide spectrum of solutions within a very narrow domain.

The data presented in the results section\(^{44}\), in many ways, favor TRA* (with MM as a very close runner up). None of the other solutions tested in the empirical study can deliver a path as fast as TRA*.

The trend suggested by the collected data points gives a hint towards the solution being fully capable of supporting larger and more complex world maps. Another temporal benefit of the solution is its capability of quickly being able to decide if a path exists or not, which prevents spending processing resources on a task that will ultimately be fruitless. These abilities of TRA* translate well into the domain of this thesis, as is evident by the good results shown in the separate real world connection study in section 8.

There is one situation in which TRA* might not be the preferable choice: when the world map is already described as a grid structure; which is certainly not unprecedented when the world is a game map, which is the domain of this thesis. In such cases, the overhead of keeping the CDT-abstraction in memory can be a high price. Even though TRA* outperforms MM on many fronts, the latter – and other grid-based solutions – might be preferable when the original world map is a grid structure, as the lowest abstraction layer then is already present, and need not be made available as a separate structure. In the section of this paper discussing the real world connection, it is also evident that TRA* might be a too structurally complex\(^{45}\) solution to be used in the domain described by this thesis.

There is some evidence supporting the statement that the Fringe algorithm is as good as, or in some cases even superior to, A* when it comes to speed. In the light of this, Fringe might be a viable alternative as a component in more complex solutions; e.g. TRA*, MM\(^{46}\), PRA, etc.

Mobile devices and video games are by themselves very demanding environments when it comes to the performance of individual algorithms used within any larger application. Combined, they form a difficult beast to master, indeed. It is our firm belief that both TRA* and MM has earned their places on this scene. It saddens us that we did not have the time or resources to test all of the solutions we came across, and found interesting, during our initial literature survey. There are probably several pathfinding solutions out there that are just as good, maybe even better, and if there are no such solutions, there soon will be. The task of pathfinding is still a very busy subject of research, and until we can find the best possible path in any arbitrary search space within constant time, it will not be abandoned.

\(^{44}\) Section 8.
\(^{45}\) The implementation of the solution grows too large, spatially, to be maintainable.
\(^{46}\) The implementation of MM evaluated in this study makes use of Fringe when searching the higher abstraction layer.
11 Future Work

When working with the TRA*- and MM pathfinding solutions, some ideas on how to improve/change these methods’ behavior were formed. The time allotted for this master thesis was not enough to implement and test these changes, but some of them could be interesting to investigate further, outside the bounds of this thesis. In this section, these ideas are given a brief presentation.

As mentioned in section 8 (results) and section 9 (real world connection), TRA* requires too much memory, in its current state, to be a feasible path finding solution on the DS. It is possible to run the system on the machine as a standalone application without any problems, but the memory requirements proved to be well outside the bounds of Pixelknights’s needs and demands. Therefore, work must be done to find ways of shrinking the implementation, so that it is useful on limited consoles, such as the DS. When looking at memory usage, triangle compression is an interesting topic. In the implementation of TRA* presented in this paper, no real effort was spent on information storage within the triangles, which leaves great opportunities for improvements; e.g. numerical- and Boolean values could be stored in bit-arrays instead of integers, floats and bools.

When looking at different world maps, most of them contain a set of sliver triangles, these triangles makes the distance measurements between connected triangles inaccurate; because of the great offset of the triangles’ mass centers. One idea for solving this is to introduce extra nodes on the constrained edges generating these sliver triangles; effectively exchanging every sliver triangle with two, or more, non-sliver triangles. This would increase the number of triangles, but only slightly, and it might be worth it if increases the likelihood that TRA* can find an optimal path on the first iteration.

Most modern games allow for dynamic obstacles; in an RTS game, such obstacles can be buildings and bridges being constructed, and destroyed. TRA* has no natural capability of solving this problem, as the process of changing the CDT in order to introduce the changes is far too time consuming to be performed in real time. To maintain several CDT representations of the map – one for each possible change to the map – is for memory reasons not a possibility if the map is to be truly dynamic; e.g. if the game allows for buildings being built anywhere, one version of the CDT must be kept by the game for every possible configuration. An idea, however, is to investigate the possibility of dividing the CDT into sectors, where each such sector could be triangulated individually, without affecting the other sectors. Another thing deserving some investigation is the possibility of, during runtime, define already existing edges as constrained or unconstrained.

Because of the limitations of the DS, and the demands of an RTS, it would be interesting to take a closer look at time-splicing, and examine the possibilities of TRA* delivering a first part of the path before all calculations are completed. For this purpose, it is very important to examine the different algorithms that make up TRA*, to see where time is spent in the solution, and if it is possible to split it into parts, all capable of delivering a piece of the path. An obvious splitting point of TRA* is before the tunneling process; this is, however, very late into the pathfinding process, and earlier points for time splicing are preferred.

In section 9 (real world connection), a path cache is added to the solution, but since no in-game testing has been done, nothing can be said of how effective such a cache would be. In the case of TRA*, this cache would be very much dependent on how long time the task of tunneling takes; if the tunneling contributes to a large part of the total time required to generate a path, the benefits of a cache might be lost. A larger case study is required to determine how the cache should behave in order to be more efficient; e.g. how many paths should it contain, how should it, and its members, be manipulated, etc. One possible improvement to the cache described in section 9 is to not only check the end-triangles of each path when checking for equality, but also neighboring triangles. By doing so, the hit-rate of the cache might increase, as there is no guarantee that all units in a group are positioned within the same triangle.
MM was not the main focus of this thesis, but nonetheless, the solution did spawn some ideas that could be interesting for some future work.

One could see MM as a hierarchical pathfinding solution, sporting two hierarchical layers (the lower tile-layer and the higher node-layer). As world maps grow larger, it could be beneficial to extend the hierarchical layering of MM\textsuperscript{47}. There are many ways of accomplishing this, but since HPA and TRA are a part of this study, solutions based on these lies close in mind. To investigate and compare the possibilities of these augmentations could prove to be very interesting work, indeed.

The placement of the region nodes in MM can be done in several different ways, depending on what one wants to achieve; as discussed by Sturtevant [15]. To further improve on the solution used in this thesis it would be beneficial to optimize for visibility between the nodes; lessening the amount of low-level searches needed to be made between region nodes.

In Section 6 (Search Strategies), some hierarchical pathfinding solutions are given some of the spotlight. It would be interesting to examine how these solutions, and other like them, perform as anytime solutions; i.e. taking a closer look on the speed of their first delivery, and the quality thereof.

In order to develop really effective pathfinding solutions for RTS games in environments of hard constraints there is a big need in understanding how the game in question is played. Maybe the troops in a game very rarely traverse entire paths over a certain length; often being redirected to other goal-points. How often are certain paths, and variances of them, reused? Are certain parts of the world map traversed more often than others, and why? These questions, and countless others like them, are not answered in this paper, but play a big part in how the pathfinding facilities of a game are queried. Which, in turn, give important clues to how these facilities are best designed. A study should be performed to gather the information needed to gain greater insight into these matters. This study would examine how human- and AI players order about their troops, with the goal of finding any patterns that could be exploited to further enhance the performance of pathfinding solutions within the domain.

\textsuperscript{47} This would further increase the memory requirements of the solution. This makes this option uninteresting, considering the constraints of the domain of this thesis. Nevertheless, it could be of interest in domains not suffering the same spatial constraints.


12 References


12.1 Web resources


Appendix A. PSEUDOCODE

This appendix contains pseudo code for a set of interesting algorithms, A*, Fringe, CDT, TRA, and MM.

A.1 A*

In data: start, the start point.
       end, the end point.

WHILE current is not equal to end
   IF openList is empty
      RETURN no path exists.
   ENDIF

   SET current to first in openList
   INSERT current in closedList

   FOR each node adjacent to current
       IF node does not exist in closedList
           IF node exists in openList
               IF openList node g > node g
                  SET openList node to node
               ENDIF
           ELSE
               INSERT node in openList
           ENDIF
       ENDIF
   ENDFOR
ENDWHILE

The code is written as explained in [W2].

A.2 Fringe

In data: start, the start point.
       end, the end point.

INIT cash
INSERT start in cash
SET f-limit to CALL heuristic(start)
SET found to false
REPERAT
   SET f-min to infinite
   FOR each node current in frontier-now
       IF current f > f-limit
          SET f-min to CALL min(f, f-min)
          CONTINUE
       END IF
   IF current equal to end
      SET found to true
      BREAK path found
   END IF
FOR each node adjacent to current
  SET new-g to current g + CALL cost(current, node)
  IF node exists in cash
    IF new-g < cash node g
      CONTINUE do next adjacent
    END IF
  END IF
  IF node exists in frontier-now or frontier-later
    REMOVE node from frontier
  ENDIF
  INSERT node in frontier after current
  INSERT node in cash
END FOR
REMOVE current from frontier
SET f-limit to f-min
END FOR
UNTIL found is true
This algorithm was invented by Björnsson et al. [21].

A.3 Constrained Delaunay Triangulation

DivideAndConquer:
In data: points, a set of points.
Out data: set of edges.

IF size of points < 4
  RETURN edges connecting points
ELSE
  CALL Sort (points)
  SET left-edges to CALL DivideAndConquer(left half of points)
  SET right-edges to CALL DivideAndConquer(right half of points)
  SET lr-base to CALL FindLRBase(left-edges, right-edges)
WHILE has more lr-bases
  SET l-candidate CALL GetCandidate(left-edges)
  SET r-candidate CALL GetCandidate(right-edges)
  IF has left and right candidate
    IF r-candidate endpoint is within circumference of lr-base and l-candidate
      INSERT edge between lr-base and r-candidate startpoint in edges
    ELSE
      INSERT edge between lr-base and l-candidate startpoint in edges
    END IF
  ELSE IF has left candidate
    INSERT edge between lr-base and l-candidate startpoint in edges
  ELSE IF has right candidate
    INSERT edge between lr-base and r-candidate startpoint in edges
  ELSE
    BREAK no more lr-bases
  END IF
SET lr-base to last edge in edges.
END WHILE

INSERT left-edges in edges
INSERT right-edges in edges
END IF

FindLRBase:
In data:  left-edges, set of edges
           Right-edges, set of edges
Out data: lr-base
OBS:     Sort sorts on y lowest first.

IF all points in left-edges and right-edges has the same x value
    RETURN edge between lowest point in left-edges and highest point in right-edges
ELSE
    SET left-points to CALL Sort(points in left-edges)
    SET right-points to points in right-edges

    WHILE no candidates found
        SET left to next point in left-points
        FOR each point in right-points
            IF edge between point and left not intersects any edge in left-edges
                INSERT the edge in left-candidates
            END IF
        END FOR
    END WHILE

    SET candidate to edge in left-candidates with smallest clockwise angle from y-axis

    SET changed to true
    WHILE changed equals true
        SET right to candidate right side point.
        FOR each point in left-points
            IF edge between point and right not intersects any edge in right-edges
                INSERT the edge in right-candidates
            END IF
        END FOR
    END WHILE

    SET candidate to edge in right-candidates with smallest clockwise angle to y-axis
    IF candidate has not changed
        SET changed to false
    ELSE
        SET left to next point in left-points
        FOR each point in right-points
            IF edge between point and left not intersects any edge in left-edges
                INSERT the edge in left-candidates
            END IF
        END FOR
    END IF
    SET candidate to edge in left-candidates with smallest clockwise angle from y-axis
    IF candidate has not changed
        SET changed to false
    END IF
GetCandidate:
In data:  edges, set of edges
Out data: candidate edge
OBS: Finding right candidate is laterally reversed to finding left candidate, this affects sort and ClockwiseAngle.

SET candidates to CALL FindeCandidates(edges)
CALL Sort(candidates)
FOR each candidate in l-candidates
   IF candidate is last in l-candidates
      RETURN candidate
   ELSE
      IF next candidate endpoint is within circumference of lr-base and candidate
         REMOVE candidate from edges
      ELSE
         RETURN candidate
      END IF
   END IF
END FOR
IF CALL ClockwiseAngle(lr-base, candidate) < 180
   RETURN candidate
END IF
RETURN no candidate
This algorithm was invented by Guibas and Stolfi [28], but this implementation resembles [W4] more.

Inserting constraint edges:
In data:  triangles, a set of Delaunay triangles.
       edges, a set of constrained edges.
Out data: Constraint Delaney triangulation

FOR each edge in edges
   SET intersected-triangles to triangles intersected by edge.
   SET polygon to merge of intersected-triangles
   SET fist-polygon to first half of polygon when divided by edge
   SET second-polygon to second half of polygon when divided by edge

   SET first-edges CALL DivideAndConquer(points of first-polygon)
   SET second-edges CALL DivideAndConquer(points of second-polygon)

   INSERT CALL CreateTriangles(first-edges) in triangles
   INSERT CALL CreateTriangles(second-edges) in triangles
END FOR
RETURN triangles
This algorithm was invented by [W4].

A.4 Tunneling
In data:  tunnel, triangles forming a tunnel
start, the start point, residing in the first triangle in tunnel
der, the end point, residing in the last triangle in tunnel

Out data: path, set of points that represents the path.

INSERT start in path
IF size of tunnel equals one
   INSERT end in path
   RETURN path
END IF

SET left-g to first common point between first and second triangle in tunnel
SET right-g to second common point between first and second triangle in tunnel

SET left-start to start
SET right-start to start

FOR each triangle in tunnel, starting from second and ending one from last
   SET candidate to last point in triangle, left-g and right-g is first and second
   IF candidate is not visible from both left-start and right-start
      IF first time
         SET left-dist to dist from left-start to left-g
         SET left-start to left-g
         SET right-dist to dist from right-start to right-g
         SET right-start to right-g
      ELSE
         SET left-dist to left-dist * 2 + dist from left-start to left-g to right-g
         SET left-dist to left-dist * 2 + dist from left-start to left-g to right-g
      END IF
      IF left-dist < right-dist
         INSERT left-start in path
      ELSE
         INSERT right-start in path
      END IF
   END IF
   SET left-dist to dist from last point in path to left-g
   SET left-start to left-g
   SET right-dist to dist from last point in path to right-g
   SET right-start to right-g
END IF
ELSE IF next triangle has corner left-g
   SET right-g to candidate
ELSE
   SET left-g to candidate
END IF
END FOR

IF end is not visible from both left-start and right-start
   SET left-left-dist to dist from last point in path to left-start to left-g to end
   SET left-right-dist to dist from last point in path to left-start to right-g to end
   SET right-left-dist to dist from last point in path to right-start to left-g to end
   SET right-right-dist to dist from last point in path to right-start to right-g to end
   IF left-left-dist is smallest of left-right-dist, right-left-dist, and right-right-dist
IF $\text{left-start}$ is not equal to $\text{start}$
  INSERT $\text{left-start}$ in path
END IF
INSERT $\text{left-g}$ in path
ELSE IF $\text{left-right-dist}$ is smallest of $\text{left-left-dist}$, $\text{right-left-dist}$, and $\text{right-right-dist}$
  IF $\text{left-start}$ is not equal to $\text{start}$
    INSERT $\text{left-start}$ in path
  END IF
INSERT $\text{right-g}$ in path
ELSE IF $\text{right-left-dist}$ is smallest of $\text{left-left-dist}$, $\text{left-right-dist}$, and $\text{right-right-dist}$
  IF $\text{right-start}$ is not equal to $\text{start}$
    INSERT $\text{right-start}$ in path
  END IF
INSERT $\text{left-g}$ in path
ELSE
  IF $\text{right-start}$ is not equal to $\text{start}$
    INSERT $\text{right-start}$ in path
  END IF
INSERT $\text{right-g}$ in path
END IF
ELSE IF $\text{left-start}$ is not equal to $\text{start}$
  SET $\text{left-dist}$ to dist from last point in path to $\text{left-start}$ to end
  SET $\text{right-dist}$ to dist from last point in path to $\text{right-start}$ to end
  IF $\text{left-dist}$ < $\text{right-dist}$
    INSERT $\text{left-start}$ in path
  ELSE
    INSERT $\text{right-star}$ in path
  END IF
END IF
INSERT $\text{end}$ in path

This implementation is inspired by [27].

A.5 TRA*

In data:  $\text{start}$, the start triangle.
  $\text{end}$, the end triangle.
Out data:  triangle-path, a set of triangles forming a tunnel.

IF $\text{start}$ equals $\text{end}$
  INSERT $\text{end}$ in triangle-path
  RETURN true
ELSE IF $\text{end}$ is loop-reachable from $\text{start}$
  IF $\text{start}$ type equals 1
    IF $\text{end}$ type equals 2 or $\text{start}$ root equals $\text{end}$
      RETURN CALL TreeTraverser($\text{start}$, $\text{end}$, triangle-path)
    ELSE
      CALL TreeTraverser($\text{start}$, $\text{start}$ root, triangle-path)
      RETURN CALL TRASearch($\text{start}$ root, $\text{end}$, triangle-path)
    END IF
  ELSE
    IF $\text{end}$ type equals 1
      RETURN CALL TreeTraverser($\text{start}$, $\text{end}$, triangle-path)
ELSE
    IF end type equals 2 and start is on circular loop, at most one type 3 triangle on loop.
    RETURN CALL LoopTraverser(start, end, triangle-path)
    ELSE
    RETURN CALL GraphTraverser(start, end, triangle-path)
    END IF
END IF
ELSE
    RETURN CALL GraphTraverser(start, end, triangle-path)
END IF
ELSE
    IF start type equals 1
    CALL TraverseTree(start, start root, triangle-path)
    RETURN CALL TRASearch(start root, end, triangle-path)
    ELSE
    IF end type equals 1
    CALL GraphTraverser(start, end, triangle-path)
    RETURN CALL TRASearch(end root, end, triangle-path)
    ELSE
    RETURN CALL GraphTraverser(start, end, triangle-path)
    END IF
END IF
END IF

This algorithm was invented by Demyen , and M. Buro [26].

A.6 MM

In data: start, the start point.
     end, the end point.

OBS:  SearchTile: any search algorithm working on the lowest layer.
       SearchGraph: any search algorithm searching a graph.
       FindRegionCenterNode: any search algorithm working on the lowest level that can be ad-
       justed to not cross virtual borders.

SET startNode CALL FindRegionCenterNode(start)
SET endNode CALL FindRegionCenterNode(end)

SET pointPath CALL SearchGraph(startNode, endNode)

IF size of pointPath < 3
    INSERT CALL SearchTiles(start, end) in path
ELSE
    INSERT CALL SearchTiles(start, second in pointPath) in path
    FOR each point in pointPath except the last
        INSERT CALL SearchTiles (point, next point) in path
    ENDFOR
    INSERT CALL SearchTiles (second from last in pointPath, end) in path
ENDIF
This algorithm was invented by Sturtevant [15].
Appendix B. GLOSSARY

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td>Generic search algorithm (see section 6.2).</td>
</tr>
<tr>
<td>Breath Fist Search</td>
<td>Simple search algorithm (see section 6.1).</td>
</tr>
<tr>
<td>CDT</td>
<td>Constrained Delaunay Triangulation, used to create abstractions for TA* and TRA* (see section 4.3 and 6.6).</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>Breath first search with considerations for link-cost.</td>
</tr>
<tr>
<td>DS</td>
<td>Dual Screen (in the context of Nintendo DS).</td>
</tr>
<tr>
<td>Environment</td>
<td>Unless otherwise defined, the word ‘environment’ refers to the physical, or virtual, world in which paths between two, or more, points can be calculated.</td>
</tr>
<tr>
<td>Euclidean Distance</td>
<td>The shortest distance between points a and -b (see section 5.2).</td>
</tr>
<tr>
<td>Fringe</td>
<td>Generic search algorithm, similar to IDA* (see section 6.2.4).</td>
</tr>
<tr>
<td>Grid</td>
<td>A search abstraction (see section 4.1).</td>
</tr>
<tr>
<td>Heuristic</td>
<td>Information, in this case an estimate of the cost to the goal.</td>
</tr>
<tr>
<td>HPA*</td>
<td>Hierarchical Pathfinding A*, a pathfinding algorithm (see section 6.3).</td>
</tr>
<tr>
<td>IDA*</td>
<td>Iterative Deepening A*, a generic search algorithm (see section 6.2.1).</td>
</tr>
<tr>
<td>Manhattan Distance</td>
<td>The distance along the x-axis plus the distance along the y-axis between two points in a plane (see section 5.1).</td>
</tr>
<tr>
<td>MM</td>
<td>Minimal Memory abstraction, a pathfinding algorithm (see section 6.5).</td>
</tr>
<tr>
<td>Navigation mesh</td>
<td>Another name for waypoint graph, often used in games (see section 4.2).</td>
</tr>
<tr>
<td>NP-hard</td>
<td>Nondeterministic Polynomial-time hard.</td>
</tr>
<tr>
<td>Optimality</td>
<td>Path length divided by the length of POV path (Fringe of A*).</td>
</tr>
<tr>
<td>POV</td>
<td>Point Of Visibility, method for creating waypoint abstractions (see end of section 4.2). Also known as visibility graph, corner graph.</td>
</tr>
<tr>
<td>PRA*</td>
<td>Partial Refinement A*, a pathfinding algorithm (see section 6.4).</td>
</tr>
<tr>
<td>PRM</td>
<td>Probabilistic Roadmap Method, method for creating a waypoint abstractions (se section 4.2).</td>
</tr>
<tr>
<td>Qualtree</td>
<td>A non-uniform grid (see end of section 4.1).</td>
</tr>
<tr>
<td>RBFS</td>
<td>Recursive Breath-First Search, generic search algorithm, (see section 6.2.2).</td>
</tr>
<tr>
<td>Region</td>
<td>A connected area within a sector, in the context of MM (section 6.5).</td>
</tr>
<tr>
<td>RTS</td>
<td>Real Time Strategy, game genre including games such as Warcraft and Command &amp; Conquer.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Search Space Abstraction</strong></td>
<td>Representation of the world that is to be searched.</td>
</tr>
<tr>
<td><strong>Sector</strong></td>
<td>A larger (containing e.g. 8x8 tiles) section of a grid (see section 6.3 and 6.5).</td>
</tr>
<tr>
<td><strong>SMA</strong>*</td>
<td>Simple Memory bound A*, an A* fork that considers memory limitations (see section 6.2.3).</td>
</tr>
<tr>
<td><strong>TA</strong>*</td>
<td>Triangle A*, a pathfinding algorithm (see section 6.6).</td>
</tr>
<tr>
<td><strong>TRA</strong>*</td>
<td>Triangulation Reduction A*, a pathfinding algorithm (see section 6.6).</td>
</tr>
<tr>
<td><strong>Tile</strong></td>
<td>The smallest unit in a grid abstraction.</td>
</tr>
<tr>
<td><strong>Waypoints</strong></td>
<td>A search abstraction (see section 4.2).</td>
</tr>
<tr>
<td><strong>World</strong></td>
<td>See ‘environment’.</td>
</tr>
<tr>
<td><strong>World map</strong></td>
<td>See ‘environment’.</td>
</tr>
</tbody>
</table>
Appendix C. Nintendo DS

Processor
Main CPU: 67 MHz ARM946E-S
Co-CPU: 33 MHz ARM7TDMI

Memory
Main memory: 4 MB RAM
Video memory: 656 KB RAM
Other: 512 KB texture memory for each screen (2 x 512 KB).

Miscellaneous
Game card: Solid state ROM, supporting up to 2 Gb (256 MB) of storage.
Screens: Two 3” TFT LCDs with a resolution of 256 x 192 pixels. The lowermost display is touch-sensitive.
Wi-Fi: Built-in 802.11b compatible wireless network connection.
Size: 148.7 x 84.7 x 28.9 mm.
Mass: 275 g.

**Appendix D. EMPIRICAL DATA**

### D.1 A*-Grid

Path optimality (average): 95.26 %  
Path divergence (average): 4.74 %  
Initial memory usage (average): 262181 bytes

<table>
<thead>
<tr>
<th>Path Length Intervals</th>
<th>Time (ms)</th>
<th>Peak Memory Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>500</td>
<td>2.69</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>17.82</td>
</tr>
<tr>
<td>1000</td>
<td>1500</td>
<td>32.19</td>
</tr>
<tr>
<td>1500</td>
<td>2000</td>
<td>69.25</td>
</tr>
<tr>
<td>2000</td>
<td>2500</td>
<td>113.13</td>
</tr>
<tr>
<td>2500</td>
<td>3000</td>
<td>172.64</td>
</tr>
<tr>
<td>3000</td>
<td>3500</td>
<td>240.57</td>
</tr>
<tr>
<td>3500</td>
<td>4000</td>
<td>303.19</td>
</tr>
<tr>
<td>4000</td>
<td>4500</td>
<td>395.76</td>
</tr>
<tr>
<td>4500</td>
<td>5000</td>
<td>543.50</td>
</tr>
<tr>
<td>5000</td>
<td>5500</td>
<td>739.31</td>
</tr>
<tr>
<td>5500</td>
<td>6000</td>
<td>681.45</td>
</tr>
<tr>
<td>6000</td>
<td>6500</td>
<td>898.01</td>
</tr>
<tr>
<td>6500</td>
<td>7000</td>
<td>996.48</td>
</tr>
<tr>
<td>7000</td>
<td>7500</td>
<td>1243.97</td>
</tr>
<tr>
<td>7500</td>
<td>8000</td>
<td>1303.90</td>
</tr>
<tr>
<td>8000</td>
<td>8500</td>
<td>1619.26</td>
</tr>
<tr>
<td>8500</td>
<td>9000</td>
<td>1602.92</td>
</tr>
<tr>
<td>9000</td>
<td>9500</td>
<td>1477.64</td>
</tr>
<tr>
<td>9500</td>
<td>10000</td>
<td>1940.56</td>
</tr>
</tbody>
</table>
### D.2 Fringe-Grid

Path optimality (average): 95.26 %
Path divergence (average): 4.74 %
Initial memory usage (average): 262180 bytes

<table>
<thead>
<tr>
<th>Path Length Intervals</th>
<th>Time (ms)</th>
<th>Peak Memory Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>500</td>
<td>1.15</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>5.25</td>
</tr>
<tr>
<td>1000</td>
<td>1500</td>
<td>14.01</td>
</tr>
<tr>
<td>1500</td>
<td>2000</td>
<td>28.79</td>
</tr>
<tr>
<td>2000</td>
<td>2500</td>
<td>45.26</td>
</tr>
<tr>
<td>2500</td>
<td>3000</td>
<td>64.93</td>
</tr>
<tr>
<td>3000</td>
<td>3500</td>
<td>86.17</td>
</tr>
<tr>
<td>3500</td>
<td>4000</td>
<td>104.07</td>
</tr>
<tr>
<td>4000</td>
<td>4500</td>
<td>135.74</td>
</tr>
<tr>
<td>4500</td>
<td>5000</td>
<td>177.10</td>
</tr>
<tr>
<td>5000</td>
<td>5500</td>
<td>234.26</td>
</tr>
<tr>
<td>5500</td>
<td>6000</td>
<td>229.16</td>
</tr>
<tr>
<td>6000</td>
<td>6500</td>
<td>300.38</td>
</tr>
<tr>
<td>6500</td>
<td>7000</td>
<td>304.42</td>
</tr>
<tr>
<td>7000</td>
<td>7500</td>
<td>400.02</td>
</tr>
<tr>
<td>7500</td>
<td>8000</td>
<td>404.33</td>
</tr>
<tr>
<td>8000</td>
<td>8500</td>
<td>453.04</td>
</tr>
<tr>
<td>8500</td>
<td>9000</td>
<td>501.46</td>
</tr>
<tr>
<td>9000</td>
<td>9500</td>
<td>478.91</td>
</tr>
<tr>
<td>9500</td>
<td>10000</td>
<td>598.89</td>
</tr>
</tbody>
</table>
### D.3 A*-POV

Path optimality (average): 100.00%
Path divergence (average): 0.00%
Initial memory usage (average): 262819 bytes

<table>
<thead>
<tr>
<th>Path Length Intervals</th>
<th>Time (ms)</th>
<th>Peak Memory Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
<td>513.80</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>484.33</td>
</tr>
<tr>
<td>1000</td>
<td>1500</td>
<td>438.78</td>
</tr>
<tr>
<td>1500</td>
<td>2000</td>
<td>427.26</td>
</tr>
<tr>
<td>2000</td>
<td>2500</td>
<td>391.48</td>
</tr>
<tr>
<td>2500</td>
<td>3000</td>
<td>404.20</td>
</tr>
<tr>
<td>3000</td>
<td>3500</td>
<td>429.96</td>
</tr>
<tr>
<td>3500</td>
<td>4000</td>
<td>417.73</td>
</tr>
<tr>
<td>4000</td>
<td>4500</td>
<td>444.79</td>
</tr>
<tr>
<td>4500</td>
<td>5000</td>
<td>466.32</td>
</tr>
<tr>
<td>5000</td>
<td>5500</td>
<td>440.30</td>
</tr>
<tr>
<td>5500</td>
<td>6000</td>
<td>492.64</td>
</tr>
<tr>
<td>6000</td>
<td>6500</td>
<td>496.66</td>
</tr>
<tr>
<td>6500</td>
<td>7000</td>
<td>503.85</td>
</tr>
<tr>
<td>7000</td>
<td>7500</td>
<td>508.20</td>
</tr>
<tr>
<td>7500</td>
<td>8000</td>
<td>521.95</td>
</tr>
<tr>
<td>8000</td>
<td>8500</td>
<td>559.18</td>
</tr>
<tr>
<td>8500</td>
<td>9000</td>
<td>663.09</td>
</tr>
<tr>
<td>9000</td>
<td>9500</td>
<td>656.25</td>
</tr>
<tr>
<td>9500</td>
<td>10000</td>
<td>574.25</td>
</tr>
</tbody>
</table>
D.4 Fringe-POV

Path optimality (average): 100.00 %
Path divergence (average): 0.00 %
Initial memory usage (average): 252067 bytes

<table>
<thead>
<tr>
<th>Path Length Intervals</th>
<th>Time (ms)</th>
<th>Peak Memory Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>500</td>
<td>543.37</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>525.31</td>
</tr>
<tr>
<td>1000</td>
<td>1500</td>
<td>474.29</td>
</tr>
<tr>
<td>1500</td>
<td>2000</td>
<td>463.98</td>
</tr>
<tr>
<td>2000</td>
<td>2500</td>
<td>424.55</td>
</tr>
<tr>
<td>2500</td>
<td>3000</td>
<td>434.41</td>
</tr>
<tr>
<td>3000</td>
<td>3500</td>
<td>472.16</td>
</tr>
<tr>
<td>3500</td>
<td>4000</td>
<td>446.60</td>
</tr>
<tr>
<td>4000</td>
<td>4500</td>
<td>486.41</td>
</tr>
<tr>
<td>4500</td>
<td>5000</td>
<td>501.71</td>
</tr>
<tr>
<td>5000</td>
<td>5500</td>
<td>464.93</td>
</tr>
<tr>
<td>5500</td>
<td>6000</td>
<td>523.92</td>
</tr>
<tr>
<td>6000</td>
<td>6500</td>
<td>536.84</td>
</tr>
<tr>
<td>6500</td>
<td>7000</td>
<td>537.28</td>
</tr>
<tr>
<td>7000</td>
<td>7500</td>
<td>531.84</td>
</tr>
<tr>
<td>7500</td>
<td>8000</td>
<td>556.34</td>
</tr>
<tr>
<td>8000</td>
<td>8500</td>
<td>571.27</td>
</tr>
<tr>
<td>8500</td>
<td>9000</td>
<td>681.00</td>
</tr>
<tr>
<td>9000</td>
<td>9500</td>
<td>667.25</td>
</tr>
<tr>
<td>9500</td>
<td>10000</td>
<td>613.86</td>
</tr>
</tbody>
</table>
### D.5 MM

Path optimality (average): 94.64 %
Path divergence (average): 5.36 %
Initial memory usage (average): 518708 bytes

<table>
<thead>
<tr>
<th>Path Length Intervals</th>
<th>From</th>
<th>To</th>
<th>Time (ms)</th>
<th>Time to First Delivery (ms)</th>
<th>Peak Memory Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>500</td>
<td>0.70</td>
<td>0.41</td>
<td>524527</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>0.70</td>
<td>0.41</td>
<td>528766</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1500</td>
<td>1.76</td>
<td>1.48</td>
<td>526757</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td>2000</td>
<td>3.47</td>
<td>3.04</td>
<td>530648</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2500</td>
<td>5.55</td>
<td>5.16</td>
<td>530509</td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>3000</td>
<td>6.59</td>
<td>6.11</td>
<td>529061</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>3500</td>
<td>9.08</td>
<td>8.63</td>
<td>532666</td>
<td></td>
</tr>
<tr>
<td>3500</td>
<td>4000</td>
<td>12.99</td>
<td>12.39</td>
<td>534413</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>4500</td>
<td>16.44</td>
<td>15.74</td>
<td>533798</td>
<td></td>
</tr>
<tr>
<td>4500</td>
<td>5000</td>
<td>18.99</td>
<td>18.06</td>
<td>535423</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>5500</td>
<td>22.65</td>
<td>21.77</td>
<td>537336</td>
<td></td>
</tr>
<tr>
<td>5500</td>
<td>6000</td>
<td>20.69</td>
<td>19.71</td>
<td>535508</td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td>6500</td>
<td>20.75</td>
<td>19.56</td>
<td>527079</td>
<td></td>
</tr>
<tr>
<td>6500</td>
<td>7000</td>
<td>20.56</td>
<td>19.54</td>
<td>535070</td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td>7500</td>
<td>20.16</td>
<td>19.08</td>
<td>527508</td>
<td></td>
</tr>
<tr>
<td>7500</td>
<td>8000</td>
<td>22.42</td>
<td>21.06</td>
<td>532498</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>8500</td>
<td>19.28</td>
<td>17.72</td>
<td>519442</td>
<td></td>
</tr>
<tr>
<td>8500</td>
<td>9000</td>
<td>18.94</td>
<td>17.67</td>
<td>517819</td>
<td></td>
</tr>
<tr>
<td>9000</td>
<td>9500</td>
<td>16.00</td>
<td>14.56</td>
<td>511321</td>
<td></td>
</tr>
<tr>
<td>9500</td>
<td>10000</td>
<td>17.75</td>
<td>16.38</td>
<td>516883</td>
<td></td>
</tr>
</tbody>
</table>
**D.6 TRA***

Path optimality (average): 96.93 %

Path divergence (average): 3.07 %

Initial memory usage (average): 243867 bytes

Average tunneling ratio: 1.69 %

<table>
<thead>
<tr>
<th>Path Length Intervals</th>
<th>Time (ms)</th>
<th>Tunneling Ratio (%)</th>
<th>Peak Memory Usage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>500</td>
<td>1.98</td>
<td>0.00%</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>2.18</td>
<td>0.00%</td>
</tr>
<tr>
<td>1000</td>
<td>1500</td>
<td>2.19</td>
<td>1.32%</td>
</tr>
<tr>
<td>1500</td>
<td>2000</td>
<td>2.71</td>
<td>0.67%</td>
</tr>
<tr>
<td>2000</td>
<td>2500</td>
<td>2.57</td>
<td>1.13%</td>
</tr>
<tr>
<td>2500</td>
<td>3000</td>
<td>2.68</td>
<td>0.96%</td>
</tr>
<tr>
<td>3000</td>
<td>3500</td>
<td>2.44</td>
<td>1.82%</td>
</tr>
<tr>
<td>3500</td>
<td>4000</td>
<td>2.59</td>
<td>1.76%</td>
</tr>
<tr>
<td>4000</td>
<td>4500</td>
<td>2.92</td>
<td>1.47%</td>
</tr>
<tr>
<td>4500</td>
<td>5000</td>
<td>2.42</td>
<td>2.19%</td>
</tr>
<tr>
<td>5000</td>
<td>5500</td>
<td>3.13</td>
<td>2.63%</td>
</tr>
<tr>
<td>5500</td>
<td>6000</td>
<td>2.84</td>
<td>2.16%</td>
</tr>
<tr>
<td>6000</td>
<td>6500</td>
<td>3.49</td>
<td>2.67%</td>
</tr>
<tr>
<td>6500</td>
<td>7000</td>
<td>3.07</td>
<td>1.79%</td>
</tr>
<tr>
<td>7000</td>
<td>7500</td>
<td>3.84</td>
<td>2.14%</td>
</tr>
<tr>
<td>7500</td>
<td>8000</td>
<td>3.34</td>
<td>1.91%</td>
</tr>
<tr>
<td>8000</td>
<td>8500</td>
<td>2.36</td>
<td>3.03%</td>
</tr>
<tr>
<td>8500</td>
<td>9000</td>
<td>5.08</td>
<td>4.55%</td>
</tr>
<tr>
<td>9000</td>
<td>9500</td>
<td>1.88</td>
<td>0.00%</td>
</tr>
<tr>
<td>9500</td>
<td>10000</td>
<td>4.92</td>
<td>1.69%</td>
</tr>
</tbody>
</table>