Thesis no: BCS-2014-04

Bachelor Thesis

HPA* Used With a Triangulation-Based Graph

Robin Engman

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Contact Information:
Author(s):
Robin Engman
E-mail: roen11@student.bth.se

University advisor:
Martin Fredriksson
Dept. Creative Technologies

Faculty of Computing	Internet	:	www.bth.se
Blekinge Institute of Technology	Phone	:	+46 455 38 50 00
SE–371 79 Karlskrona, Sweden	Fax	:	+46 455 38 50 57

Abstract

Context. Pathfinding is an important phase when it comes to AI. The AI needs to know how to get from one point to another when there are obstacles ahead. For that reason, different pathfinding algorithms have been created.
Objective. In this paper a new pathfinding algorithm, THPA*, is described, and it will also be compared to the more common algorithms, A*, and HPA* which THPA* is based on.
Methods. These algorithms are then tested on an extensive array of maps with different paths and the results consisting of the execution times will be compared against each other.
Results. The result of those tests conclude that THPA* performs better in terms of execution time in the average case; however it does suffer from low quality paths.
Conclusions. This paper concludes that THPA* is a promising algorithm albeit in need of more refinement to make up for the negative points.

Keywords: Pathfinding, Graph Traversal, AI

Special thanks to:

Bernhard Kornberger,
the creator of the Fade2D program used for triangulations

Nathan Sturtevant,
for the map and path data sets

I also give special thanks to my supervisor:

Martin Fredriksson,
for guiding me through my work as best as he could

List of Figures

Contents

Chapter 1

Introduction

1.1 Problem Domain

The problem of finding the shortest path between two nodes, A and B, is the basics of many applications that calculate the routes from A to B. An algorithm called "Dijkstra's algorithm" was conceived by Edsger Dijkstra in 1956, and published in 1959[1], to solve this exact problem of finding the shortest path between different nodes.

Dijkstra's algorithm is however not something that solves all the problems in graph traversal, and therefore other algorithms with more specific goals have been created. Dijkstra's algorithm, for example, solves for the shortest distance from one node to every other node in the graph. This could be very time consuming if we just want to know the distance to one node. Therefore, algorithms have been developed to lessen the search time, like the algorithm A*, and there are algorithms that do not lessen the search time itself, but the search space which in turn lessens the time it takes to find a path. Those kinds of algorithms are called graph partitioning algorithms or graph abstraction algorithms.

An abstraction of a graph is a way to represent a world but with less information than the real world actually contains. For example, an object like a rock could instead be represented as a square that someone cannot pass through, and an area of grass could be a square were someone can pass through. When this concept is applied to a large area you will get a grid of many squares detailing if they are passable or non-passable. These grids in turn can be used as the nodes for a pathfinding algorithm to find a traversable way from point A to B.

Another way to abstract a world would be to perform a triangulation of it. This consists of putting points that represents the edges and corners of solid objects like trees and rocks, into a 2D plane, and then connecting those points to create triangles out of all points. The triangles that are gotten can, like the squares, be marked as traversable or non-traversable to create a network of triangles where the triangles act as the nodes for a pathfinding algorithm.

Those are two ways of abstracting a world into a more useful format for pathfinding, as computer resources are limited and we want to find these paths as fast as possible. An area where efficient pathfinding is important is in video games. The player does not want to sit and wait for the computer to finish path calculations for all objects in the game and it is therefore important that the calculations are done in a timely manner. That is the reasoning for limiting the amount of search space and improving the search time for the pathfinding in a game.

When it comes to games it is also important to separate static and dynamic graphs. A static graph is a graph which the user can expect to never change, therefore there is no need to take into account that any changes will occur. A dynamic graph, however, has the ability to change at any time, therefore changing the structure and connections of the nodes in the graph.

Game worlds can be represented by a grid-based graph, whether they are dynamic or static, to lower the complexity of searching for paths. For such graphs, an algorithm called HPA* was developed[2]. HPA* works by abstracting the abstraction, which is the grid, into sectors and assigning entrances between these sections along their borders. These entrances in turn work as the nodes between the sectors in this abstracted graph. Enhancements for HPA* have been researched, like DHPA*[3], which trades search speed for memory usage and slightly less optimal paths. DHPA*, however, often outperforms HPA* in dynamic environments. The same paper also introduces SHPA*[3], which only works on static environments, but uses slightly less memory than HPA*. Other enhancements for HPA* have been tested that do not change the general idea of HPA*, but instead changes the algorithms used for HPA*[4].

The game environment can also be represented as polygons, and that give many advantages over the grid representation[5]. One way to do that is to use the triangulation method, with the help of something called Delaunay triangulation, and there has been work done to show that the triangulation of an environment is both efficient[5] and it can be updated dynamically[6]. Kallman has also presented diverse navigation queries in a triangulated graph, showing how dynamic objects can behave[7].

In Efficient Triangulation-Based Pathfinding, Demyen and Buro successfully apply the standard pathfinding algorithm A* to a graph extracted from a triangulated environment[5]. The conclusion from that experiment showed that their algorithm TA* found most paths faster than when A* was used on a grid. It is also said that many techniques which were originally meant for grid representations have potential with a triangulated graph, and that is where this paper will look further.

This paper will use the standard HPA* algorithm, but instead of a grid as a graph, it will be used on a graph gained from a triangulation. This means that HPA* will have to be modified to work with a triangulated graph, while still retaining the concept of it. The new algorithm, which this paper will call THPA* (Triangulation Hierarchical Pathfinding A*), will be described with regards to how the nodes in the graph are sectioned, and how the actual pathfinding will work. THPA* will then be compared in terms of execution time to the original

HPA* and A*, both on a grid-based graph. The graph data that will be focused on in this paper will be of a static nature and only one path will be searched at a time.

1.2 Research Question

What is the speed-up for the THPA* algorithm, on a triangulation-based graph, compared to A* and HPA* on a grid-based graph, in terms of execution speed?

1.3 Goal and Purpose

The goal of this paper is to apply the HPA* concept to a graph that has been conceived by Delaunay triangulation to create a new algorithm, THPA*. The new algorithm will then be compared to HPA* and A* on grid-based graphs. The data that will be measured and compared is the execution time for the different algorithms.

From creating THPA*, this paper expects to see a speed-up in execution time compared to the original HPA*, and even A*. This paper also expects to show that another pathfinding technique that was meant for grid-based graphs can work with triangulation-based graphs.

1.4 Research Approach

To be able to measure execution times and the difference between this paper's presented technique, THPA*, and the already existing techniques, HPA* and A*, there is a need for an implementation of the techniques. For HPA* this paper will use the concept from the original paper[2] as a base for the implementation, and A* will also be of the standard form, meaning it will be using the best-first search approach. The new technique, THPA*, will follow the same concept of HPA*, but instead of a grid-based graph, it will be using a triangle-based graph.

These three pathfinding algorithms will then be used on a static environment where the execution time of finding a path from A to B will be measured and compared. The static environment will be represented as graphs, where HPA* and A* uses a grid-based graph, and THPA* will use a triangle-based graph. These graphs are as stated will be static, meaning they will not change their connections.

1.5 Delimitation and Disposition

This thesis will not discuss or investigate:

- The memory aspects of the different algorithms.
- The execution speed and process of converting an environment into a grid or triangle based graph.
- The difference between a static and dynamic environment as this thesis will only focus on a static environment.
- The realism of the paths that are found.

In chapter 2, this paper explains the background of the techniques leading up to THPA*. The paper will go through the grid-based approach that HPA* uses and how the algorithm works with such graphs. Then the triangle-based approach which will explain what Delaunay triangulations are and how they can be seen as graphs in a game environment.

In chapter 3, THPA* will be described in its entirety. This paper will explain how the HPA* sectioning approach will work together with a triangle-based graph instead, and how the traversing of such a graph will be done.

In chapter 4, the method for measuring and the data sets used for the experiments will be presented. It will also deal with such things as how the algorithms were implemented, the environment where it was run on, and the potential problems with this paper's method.

In chapter 5, all the results gathered from the experiments are presented and analysed.

In chapter 6, a summary of the paper and suggested future work will be dealt with.

Chapter 2

Background

2.1 Pathfinding

Pathfinding is the way to traverse a graph and then extract different ways for how to get from point A to point B in the graph. Depending on where the path is going to be used some paths are better than others. For example, if we would like to move a unit in a game from one point to the other on a flat terrain with walls as obstacles while conserving for example stamina, then the shortest path would be best for that unit. However, if we would like to move a unit around on a terrain with different surfaces that damages the unit, then a longer route might be better to make sure the unit is not destroyed.

No matter what the path is going to be used for, it is important to make sure a path exists at all and also figure out how to get there. That is where pathfinding algorithms such as A* comes in.

These pathfinding algorithms requires the user to have a graph ready for them, it does not really matter how the nodes are connected with edges or the cost of going from one node to another, because a pathfinding algorithm will find the best path to use depending on the situation, and of course which algorithm that is used. These graphs can be constructed from information from different environments, whether it is a real life environment or a game environment. There are two ways of abstracting these environments that this paper will go through, grid-based and triangle-based.

The graphs for a pathfinding algorithm usually contain nodes and edges. Nodes can be seen as the information holders as they mostly contain data such as positions, height, penalties, or anything that will be related to that exact node. The edges are the roads connecting these nodes; they contain only the information of how difficult it is to pass from one node to the other. The difficulty can be simple things such as distance between nodes, or it can contain more information such as telling if a path is uphill, making the road more difficult.

Once all this information has been gathered inside one graph, the user tells the pathfinding algorithm to find a path between A and B, using the given algorithm, and the pathfinding algorithm will give you the path. This path will contain which nodes to visit and in which order a unit should visit them to reach B from A.

This path can also be post-processed to adjust it for current needs, whether it is smoothing the path to make it more realistic, or scrambling the path to simulate a more incoherent behaviour.

As soon as the path is completed it is up to whoever asked for it to use it. This means that the pathfinding algorithm has no responsibility for what is done to a finished path.

For pathfinding in games you will often need abstractions of the game environment. This is a way to represent the environment using graphs that do not match the environment entirely, but close enough to feel right when looking at it. These abstracted graphs are then used with different pathfinding algorithms, for example HPA* uses a grid-based graph as its basis,[2] and TA* uses a triangle-based graph as its basis.[5]

The upcoming sections will describe these different forms of abstraction and their pathfinding algorithms in more detail.

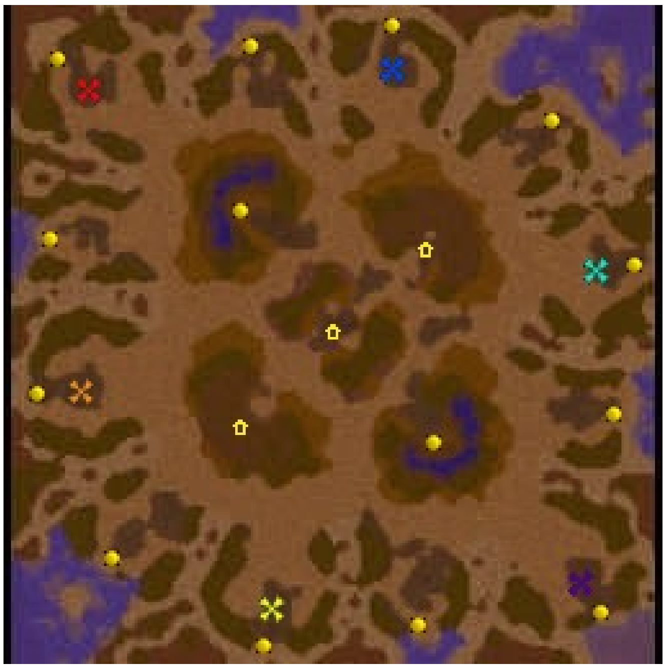

Figure 2.1: *The original Warcraft III map Scorched Basin.*

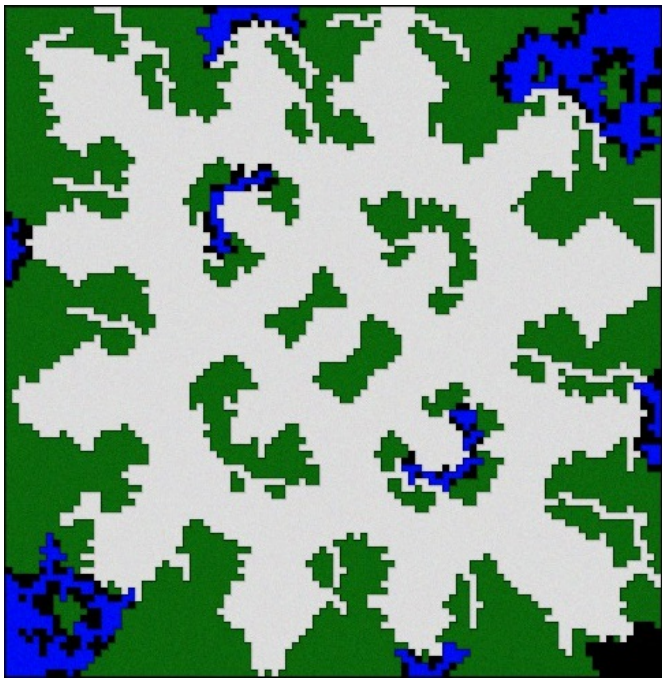

Figure 2.2: *The grid-based abstraction of the Warcraft III map Scorched Basin. White is passable terrain. Green, blue and black is non-passable.*

2.2 The Grid-Based Approach

The grid-based representation of a game environment is the abstraction of all the objects in a game world, and is in its most simple form squares that are either passable or impassable. The passable squares are then connected with each other to form a graph. In this graph, the squares representing the passable environment are the nodes, and the paths between these squares are the edges. This graph containing all the information about the passable terrain and how one square is connected to another, is an abstraction of the game environment as it just barely represents the real environment.

This abstraction of the environment can be achieved in different ways. For example, we could create the entire graph by ourselves just by looking at the game environment and manually input all the information about the nodes and the edges between them, which will take quite long if the game environment is large. A user can also produce an abstraction by just marking all the squares as passable or impassable, and then let an algorithm take care of forming the nodes and connecting them. This can go one step further as the user can let the objects placed in the environment be passable or impassable, and then let the squares intersecting this object be marked as passable or impassable. The last approach works well for level editors as the map editor editing the environment does not have to manually input node data or having to mark single squares by themselves.

There is however a problem that can occur when using the grid-based graph, and that problem is that sometimes, not everything in the game environment is a square. This problem can lead to having terrain that is not blocked by an object completely becoming impassable, as a tiny piece of the object might intersect with a square, leaving the rest of the square empty.

This problem can be circumvented by making sure the models used for the objects fit neatly into the squares, meaning some freedom is lost with the design of models. A user can also circumvent the problem by making the squares used to represent the environment smaller, but that will increase the amount of nodes that will have to be searched. In other words, if a user needs to use a grid-based representation, then the user needs to understand that it will come with these drawbacks. However, a grid-based representation is well suited for partitioning, since the squares will fit neatly into another square bounding volume. The environment can then be further abstracted with these partitions, which leads to a certain algorithm called HPA*.

A grid abstraction can be made more complex, by introducing such things as height and one-way paths. The height of the environment will add more information to adhere to when creating the abstracted graphs, as at a certain height difference between nodes, there will not be a path between them. A good example of such height differences are cliffs and plateaus. The one-way path in the graph means it is a directed edge. These one-way paths can for example be, you can jump down this cliff, but you cannot get back up on it.

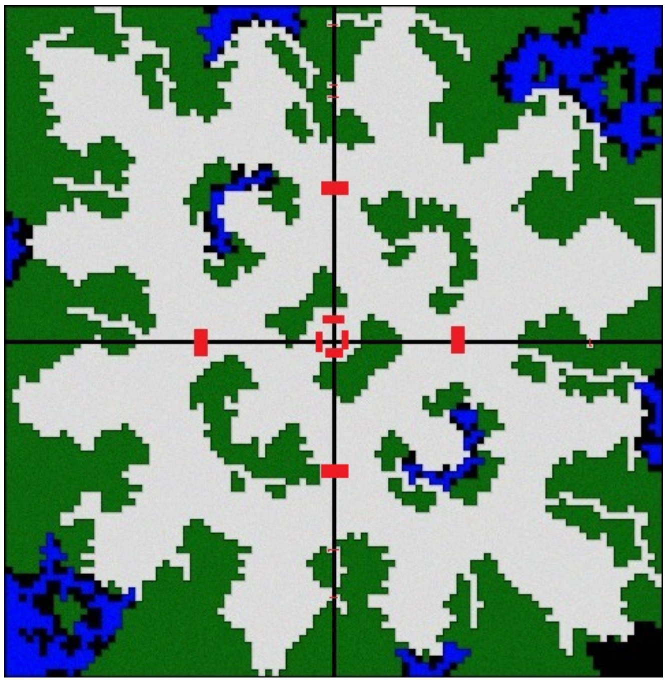

Figure 2.3: *The entrances, marked in red.*

2.3 HPA*

HPA* is an algorithm that uses an abstracted graph of an already existing grid-based graph. What that means is that the grid-based graph described in the previous section has been further reduced to create a new graph. The nodes of this new graph now contain their own graphs, which is the old grid-based graph but now partitioned into smaller sections of the original grid. Botea, Müller, and Schaeffer describes it at looking at the new graph's nodes as cities, and those cities contain their own network of roads and locations, which is the different sections of the old grid-based graph. The basis behind their approach is that a human does not normally plan their entire route exactly, but instead they take it in steps. For example, they can plan which route to take from city to city, but they will only plan which way to take inside of the city, once they are actually there.

The first step when creating the abstraction of the grid is to define the size of the larger squares, also called clusters. These clusters will cover the grid and divide it into smaller sections. This is the partitioning of the grid and the clusters can be seen as the cities. The partitioned areas of the grid will then be searched for something called entrances between them; this can be seen as the nodes that connect different cities.

The entrances are used to create the graph, and the entrances themselves are the nodes and the connections between them are the edges. These entrances are defined as the maximal obstacle-free segment along the common border of two adjacent clusters. The entrances that are extracted are then connected both inside their own clusters, intra-edge, and to entrances in other clusters, inter-edge. The length of the inter-edges is always 1 as the transition between clusters will always be done from one edge of a line segment of a cluster, to another. The intra-edges are however computed by looking for optimal paths inside a cluster.[2]

When all of the entrances have been defined and the edges between them have been computed, the new graph is finished and is ready to be used with hierarchical pathfinding.

The pathfinding for HPA* is pretty straight forward. As always when looking for a path, a unit has a start position and a goal. Both the start and goal positions are temporarily inserted into the newly abstracted graph in their relevant clusters, and then an optimal path to the closest entrance is searched for locally in the relevant clusters. Once a path has been found, an edge is connected between the two temporary nodes and their closest local entrances.

Once the start and goal position have been inserted fully into the graph, the algorithm A* is run on the newly abstracted graph to find the shortest abstract path between all these clusters. After the abstract path is found, there are optional steps such as path refinement.

Path refinement is the process of translating the abstract path into a low-level path. This means the path inside each cluster that the abstract path passed through will be calculated when needed. The paths between entrances inside a cluster can be saved when doing the pre-processing of the intra-edges, which means a table look-up, is sufficient. If not, A* can be used in the clusters again to find the optimal path between two entrances.

The HPA*-algorithm outperforms A* by up to 10 times[2], however, it suffers from non-optimal paths. This does not mean the abstract path itself is non-optimal, as it will find the optimal path in that graph, but the low-level grid might not have used the entrances defined in the clusters and therefore the refined path could lose some optimality.

2.4 The Triangle-Based Approach

The triangle-based approach is a different way to create an abstraction of a game environment. It is achieved by doing a triangulation of the world using the obstacles' vertex points as the basis for something called a constrained triangulation. The edges shared for one object are called constrained edges and every triangle in a triangulation contains at least one of these constrained edges' endpoints. The line segments between the obstacles' vertex points become what are known as unconstrained edges and these are used to connect the constrained edges' endpoints till a triangle is formed.

The constrained edges in a triangulation can be seen as the impassable squares in the grid-based approach, as an object cannot pass over such an edge. The unconstrained edges are the opposite and can therefore be seen as the passable squares. That means that if two triangles share an unconstrained edge, there is a path between the triangles. However, the edges do not serve as nodes. Instead, for triangle-based graphs, the edges are what decides if there is a path or not, and not the triangle itself, compared to the grid-based graph where the squares decided if there was a path or not, and the edges where placed according to the nodes.

The triangles for the triangle-based graph are the nodes of the graph, and the edges of the triangles are what decide if there is a path between two triangles. If two triangles share an unconstrained edge, an edge for the graph will be inserted between the two triangle nodes. The fact that it is a triangle also means that there's a maximum of three edges that can be connected to the same triangle node.

A way to make sure that these triangles in the triangulation are as optimised as possible is to use a Delaunay triangulation. This makes sure that the best path found between two points in a graph of triangles, will never cross the same triangle more than once.[5]

Delaunay triangles have the fundamental property of in a set of points in a plane, for 2D triangulations, that the circumcircle of each triangle contains no other points than the points associated with the triangle. That means that the triangles will strive to be as close to equilateral as possible so to avoid that no other points will intersect with a triangle's circumcircle, this also means that the minimum angle in a triangle will be as high as possible.[8]

The circumcircle of a triangle is defined as the unique circle that passes through all the triangle's vertices. That means the more thinly a triangle is, the bigger it will be compared to the triangle it is associated with.

This triangle-based graph does not seem like it would fit neatly into the clusters that the HPA* utilises, since more often than not, triangles will cross over into different clusters. However, that could be used as an advantage to entrance finding which will be described in the THPA* chapter. The basis of this paper is also to use the concept of HPA* and apply it on a triangle-based graph, and to find if THPA* can be speedier than HPA*

Chapter 3

THPA*

3.1 Overview

THPA* is the new algorithm that this paper will present. It will use a triangle-based graph achieved from a Delaunay triangulation, and it will then apply the HPA* concept of clustering to make a hierarchal graph of the triangulation.

The first step that will be described in this chapter is how the sectioning of the triangles will be done with the help of rectangular clusters.

The second step described is about the defining of the entrances between these clusters.

The third step will explain how the entrances are connected to each other.

The last step will be how to achieve a low-level path from the abstraction of the triangle-based graph.

3.2 Sectioning of a Triangle Graph

The sectioning of the triangles is only done by storing the entrance triangles inside their connected rectangle clusters. This means that the rectangles only have data on those triangles, and there is no information about the rest of the triangles in the world. The rest of the triangles are still stored in the original triangle-based graph.

3.3 Defining Entrances

The first thing that has to be done when defining the entrances between clusters is to find every possible candidate for an entrance. There are two possible scenarios for these candidates. The first one is the triangles that intersect two rectangular clusters that lie next to each other. This is the easier one of the scenarios as only two triangle vs. rectangle collision tests are needed. The second scenario is when a triangle is contained by one rectangle, but intersects with the edge of another rectangle. This could possibly create false entrances as we now have no idea if there is actually a path between these rectangles. To prevent this from

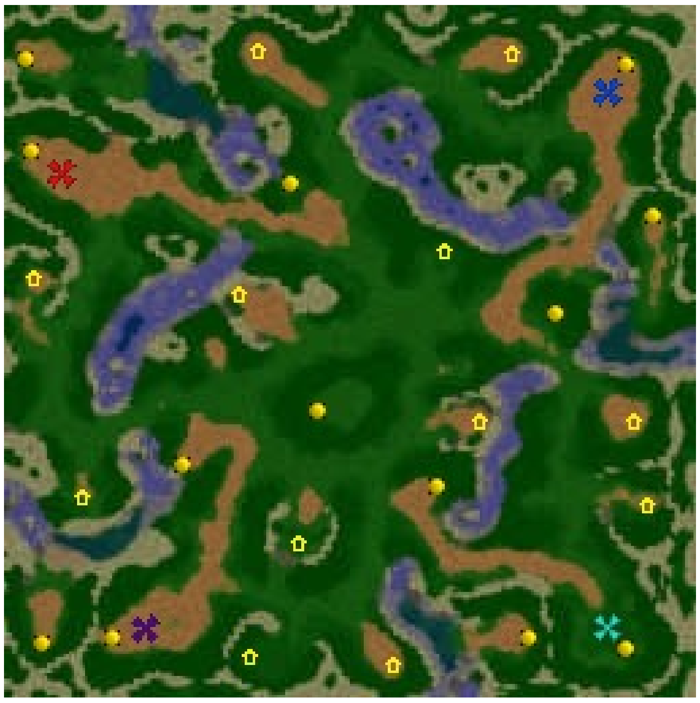

Figure 3.1: *The original Warcraft III map Tranquil Paths.*

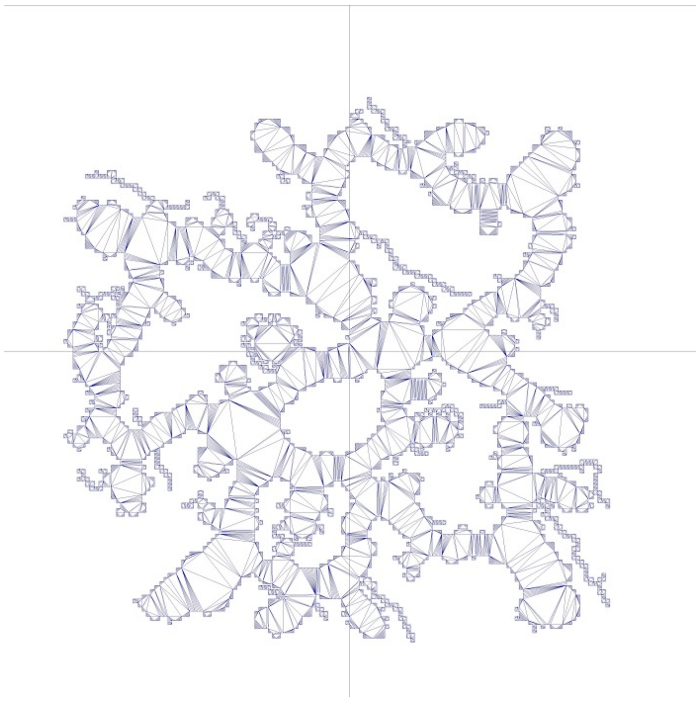

Figure 3.2: *The triangulated abstraction of the Warcraft III map Tranquil Paths.*

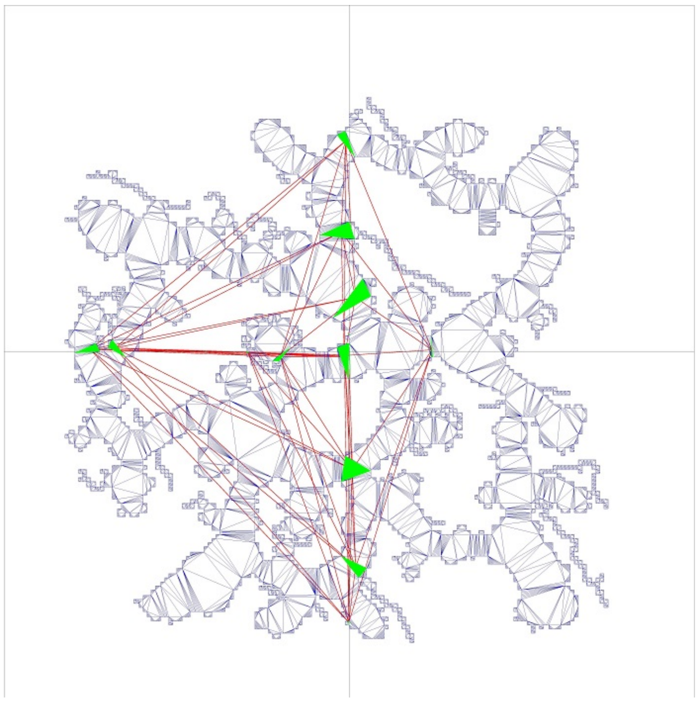

Figure 3.3: *The entrances, marked in green, and their connections, the red lines.*

happening, we must check if any of the problem triangle's neighbours intersect with both rectangles, or if one of them is fully contained in the other rectangle. If one of those cases is true, then we have another candidate for an entrance.

Once all candidates have been extracted, the next thing to do is to expand them so that only one entrance exists per actual opening between two rectangles, since an opening can consist of many triangles. To do this, we go through all possible candidates and check if their neighbours are also possible candidates. If they are, they get added into the list of triangles that will define one entrance per opening. This check is done recursively so that when no new possible candidate neighbours exist, the expansion stops. After all triangles have been assigned to their respective openings, an average of their centres is calculated. The triangle that exists on that point will be the entrance triangle between two rectangle clusters.

3.4 Connecting Entrances

Like HPA*, the now calculated entrances have to be connected to each other to create the abstracted graph. To do this we use A* between entrances that exist in one rectangle to see if any of these are connected. If they are, a two way connection is made and the cost of going between these two entrances is stored. Once all entrances have been connected in their respective rectangles, we now have a finished abstracted graph of the world. All the steps up to here only has to be done once for a static graph, however, if it was a dynamic graphs you would need to re-section, re-define, and re-connect everytime a triangle that lies on a border is changed.

3.5 Finding a Path

Once all the pre-processing has been done, the new graph is now searchable. To find a path a user inputs a start and an end point. These two points are used to find the triangles in the graph that will now be used as start and end triangle nodes. These new nodes will be inserted into the abstracted graph by connecting them to the entrance nodes in their respective rectangles. Once all the connections have been made, A* star is run on the abstracted graph. When A* has found a path in the abstracted graph, we now have a list of the nodes that have been used to get from the start node till the end node. We use these nodes in the list as the new start and end nodes. This means that the first node in the list is now used as the start node, and the second node in the list is now the end node. Again, A* is used to find a path between these new nodes. However, this time the path will be more refined as it is now actually done on original triangle graph. The next search has the second node in the list as the start node and the third node as the end node, and it continues like that until the last node in the list is reached. We now have multiple smaller paths that connect into one large path, which is the final path from start to end. This path can then be refined to create more "realistic" movement for an AI.

3.6 Path Quality

As with HPA*, THPA* suffers from low path quality. This comes from the fact that the search will always path towards entrances before crossing over to another rectangle cluster, even if the node you would like to go to is only two steps away, but in another rectangle. The fact that there will sometimes be very large triangles also means you will lose precision if you use a naive method like moving towards the centre of each triangle. This can also create a lot of criss-crossing as a unit moves from one triangle to another, for example if you have a square and you want to move straight through it, if it consists of two triangles instead then you will first have to move to the right, and then to the left. This triangle problem can however be solved by a good funnel algorithm.[5]

Chapter 4

Method

4.1 Approach and Selection

As the purpose of this paper is to measure the speed-up for THPA* compared to both HPA* and A* when it comes to finding paths, it is important to use a steady method of measuring their execution times. For measuring these algorithms timestamps will be used. The timestamps will be achieved by querying the system for its time right before the algorithm starts looking for a path, and right after the algorithm has delivered its path. The time between these two timestamps will be the execution time of the algorithm.

Since the study objects of this paper are the different algorithms, THPA*, HPA*, and A*, the algorithms will all be used on the same maps. The only difference between the input is the fact that THPA* uses a triangle based graph, and HPA* and A* use a grid-based graph. The graphs that are conceived will be abstractions of 36 maps from Warcraft III, which will give a variety of different looking graphs to test the execution speeds on. These maps also come with different scenarios averaging about 1000 paths per map, giving a total of 36 000 paths to be tested for each algorithm.

For the triangulations an already existing program will be used. This program is called Fade2D and it was made by Bernhard Kornberger.

4.2 Measuring

A simple software is created to handle the measuring of the execution times of the different algorithms. This software will load a graph from a file into memory and then use the applicable algorithm to find specified paths between different points of the graph. To make sure that these paths are as expected the software will render a 2D-image of the abstraction for the environment, and then the paths will be rendered.

The time stamps will be achieved by using the C++ function QueryPerformanceCounter(), which will be implemented right before the algorithm is called, and right after it ends. This means it will not take anything else in the code into account. The difference between these two time stamps will then be seen as the algorithms execution speed for that certain path. It is important that each algorithm's execution time is measured using the same path, because shorter paths will be found faster, and longer paths will take longer.

Each abstraction of a map will have a set amount of paths to be measured. These paths have been attained from Nathan Sturtevant's Moving AI Lab at the University of Denver where the maps used for the experiments have also been attained.

The software will output all execution times to a file, with one file per map. The data will be written in three different columns with the algorithm used showing at the top. This data will then be calculated to find the speed-up of the THPA* algorithm which will be showed in percentage values. This speed-up will be used in the analysis of the Results chapter.

4.3 Experiment Environment

As the environment that this paper is testing these simulations on is important for the resulting execution times, I will present it here:

CPU Intel Core i5-2320, 2.00GHZ
GPU NVIDIA GeForce GTX 560 Ti
RAM 6.00GB
OS Windows 7 Home Premium

The hardware has a large impact on the speed of the algorithms as different CPU speeds will give quite different results. The software that this paper's testing environment will be implemented in will be Microsoft Visual Studio 2012 and the language used will be C++. For the rendering part DirectX11 will be used. This software is used since the Delaunay triangulation algorithm is in it and because the writer of the test program has more experience with the software.

The settings for the C++ compiler will be the standard for Release mode. This means no debug options are enabled as these could slow down the algorithms considerably. This specific program will also only run on one core for simplicity's sake and becaues of time constraints.

4.4 Execution

The tests are performed by feeding the implementation files containing the data of the graph structures for the different algorithms. Once all algorithms have been used for the same map, the result data will be output to a file. After this file has been written, the implementation is ready for a new file representing a new map. When all maps that are going to be used have been fed to the implementation, the result data files will then be used to calculate the speed-up between the different algorithms.

4.5 Method Discussion

The use of the time stamp method meant result in minor error margins as the execution time needed to query the end time will be included in the total time. This method also only allows the program to be run on one core at a time and such might return different speeds if more cores are used.

The way the program is compiled will have major influences in the execution time. This program was compiled in default release, 32-bit, mode in Microsoft Visual Studio 2012. Changing the compiling method will result in differing times.

The data structure behind the A* algorithm is also important to keep track of. In this program an STL(Standard Template Library) heap is used which results in O(1) add and delete, but O(n) search. If a different structure is used it will change the execution time of all algorithms since different structures will have differing add, delete, and search times.

The method can be compromised if we do not make sure that the test program is getting all the resources at all times, which means it best to run it on a computer when as little as possible is being run in the background. The program should also be dedicated to its own core.

Chapter 5
Results

5.1 Observations

The THPA* algorithm performs better than both A* and HPA* in the average cases. However, A* still has THPA* and HPA* beat when it comes to minimum times. This comes from the fact that THPA* and A* will path towards entrance nodes before heading to the goal, even if the goal node is only two jumps away from the start node, but it is in a different cluster. A* will however have disastrous maximum times if the graph has a lot of dead ends in it.

The average time for both HPA* and THPA* is better than A* by quite a lot, and this is because most of the graphs had a lot of dead ends in them, resulting in high times for A*. HPA* and THPA* will better get around these dead ends as they already have some information about where entrances already exist.

The main drawback for THPA* and HPA* is of course the low quality paths, there are however ways to refine them.[2][5] This will make the paths approximate A*, but A* is the best algorithm for finding the shortest path without any refining.

The maps were all very different from each other and have all been used in the real game Warcraft 3, and therefore should be a good indicator for the quality of both HPA* and THPA*. It is important to note that any gain in execution speed with THPA* and HPA* will most likely be needed to refine their paths, this is however something that this thesis will not discuss.

5.2 Result Analysis

All maps - Number of paths: 41646

Time(ms.)	Avg.	Min.	Max.
A*	1652.9100	0.0017	38714.9000
HPA*	35.1578	2.7214	1062.6900
THPA*	20.7521	1.6114	259.6570

From the results we can see that THPA* and HPA* is better than A* in the average case, and THPA* is better than HPA*. This is to be expected as THPA*

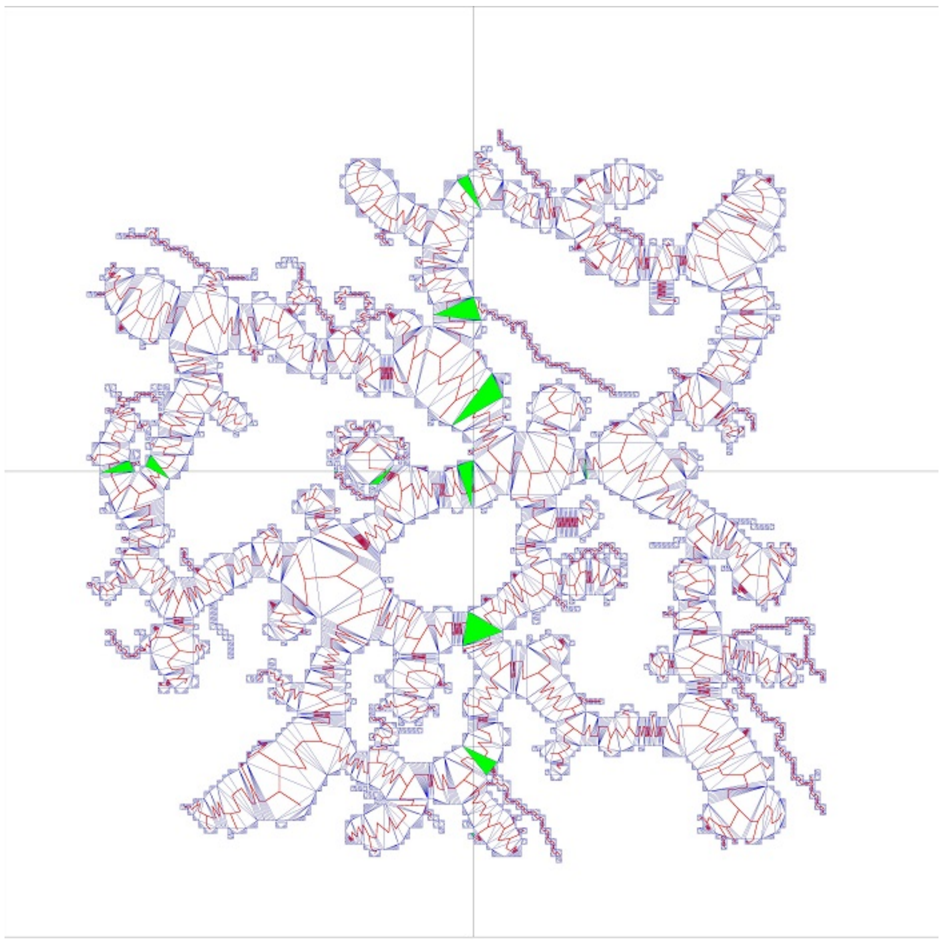

Figure 5.1: *The rendering of all paths on the map. Notice how the paths will only cross rectangles at an entrance, marked in green. In some cases it may look like they pass over at the wrong spot, but that is because a triangle can belong to two different rectangles at once.*

and HPA* will avoid many of the dead ends that exist in a graph where A* will have to face the dead end and then return to an earlier position to try another path.

The fact that THPA* is faster than HPA* is because the triangulated graphs will contain a lot less nodes than the grid-based graphs. The triangulated graph also only has a maximum of three neighbours, where the grid could have a maximum of eight if diagonal movement is allowed.

A* has the best minimum time but also the worst maximum time. It has the best one because it does not path towards entrances and goes straight to the goal, and this works very well for it if there are no obstacles in the way. It gets the worst time because it does go straight to the goal, meaning it will most likely hit more than one dead end on the way.

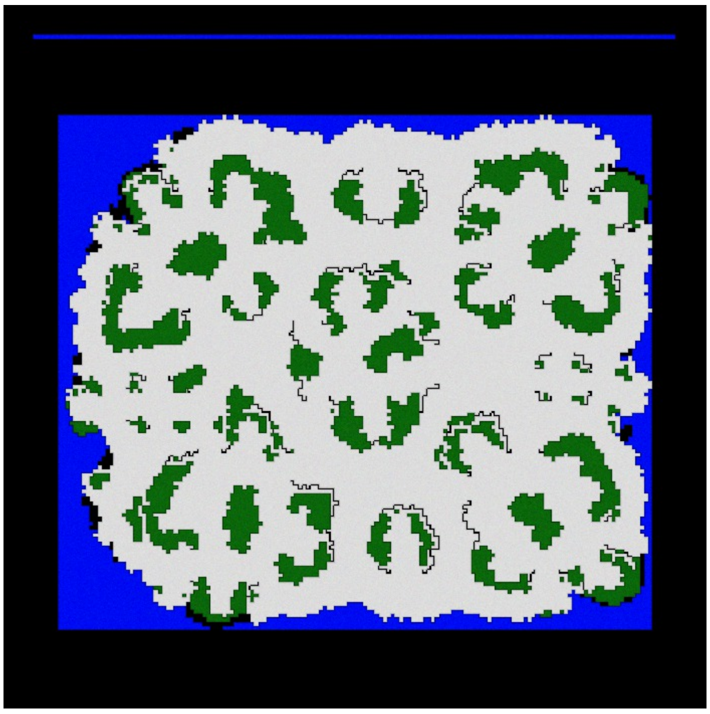

Figure 5.2: *Blasted Lands.*

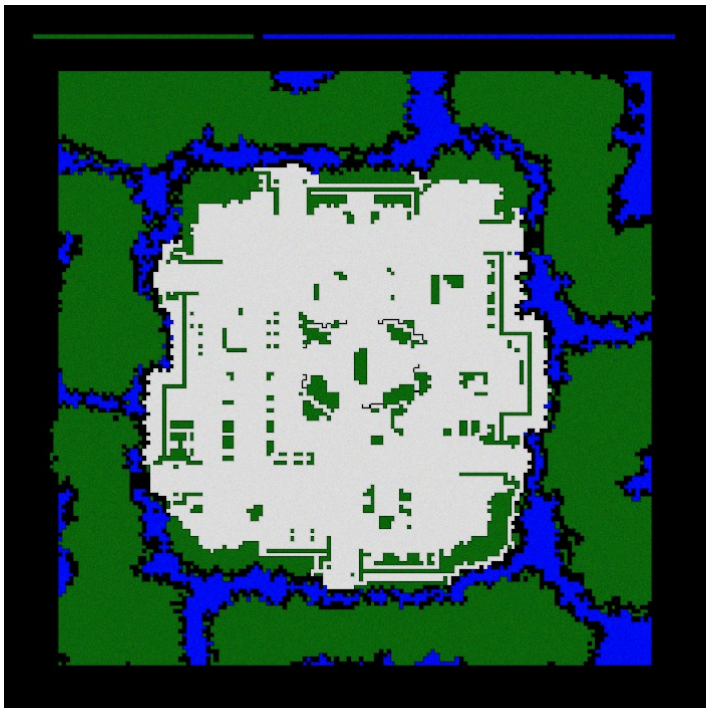

Figure 5.3: *Stromguarde.*

The final number of different paths checked was 41646. That means each map had an actual average of 1157 paths per map. More information on the exact number of paths is found in the appendix.

A good example of where A* runs into many dead ends and where THPA* really outshines A* is the map Blasted Lands. On that map A* had an average of 787ms to find a path, where THPA* only had an average of 18ms. This map contains many obstacles and no large open areas and that is the reason to why A* performs so bad.

An example of where A* is not as terrible is for the map Stromguarde. This map did not contain as many paths to be searched but the average for A* is still quite low at around 3ms. This is because this map is a bit smaller than Blasted Lands and it also does not contain many dead ends in the middle of the map like Blasted Lands.

Chapter 6

Conclusions and Future Work

6.1 Summary and Conclusions

In the first chapter we were introduced to the world of pathfinding and how there are different ways to go about it, with the help of abstracting and different graph traversal algorithms. We also saw that fast and reliable pathfinding is also important when it comes to games especially.

In the second chapter we went through some different approaches to abstracting and also an algorithm that uses one of these abstracting for their pathfinding purposes. The abstracting approaches were grid-based and triangulation-based, and the pathfinding algorithm was HPA*.

In the third chapter the new algorithm THPA* was explained. We first looked at how the sectioning of the triangles would work. The second step was to define the entrances, and the third step was how to connect these entrances. The last step was how to perform a pathfinding search with the new abstraction.

In the fourth chapter we explained the method that was used to obtain results. The measurements were done on execution time only with the help of time stamps. The algorithms were given the different abstractions of the same maps and then multiple test paths were run on them. At the end potential flaws with the method were handled.

In the fifth chapter the result of the execution times were presented and analysed. All the execution times for all paths were turned into an average, a maximum, and a minimum. This showed the tendencies of the different algorithms because of their structure.

To summarise, the algorithm THPA* did perform better in most cases compared to A* and HPA*, but it does still suffer from low quality paths. THPA* is still a very good candidate for real use as the execution time is very low and that gives room for path refinement. It also avoids using grids which means a user could have better paths around more round objects.

6.2 Contributions

This paper has shown that using techniques meant for grid-based A* on a triangle-based graph can lead to significant improvements in execution time. It has also contributed with real data of the execution times when comparing THPA* to HPA*. The guidelines for how to do THPA* have also been presented so that anyone can use the same concept to create better algorithms.

6.3 Future Work

There are many things in this paper that have not been measured or been taken into account. For example how good is the path quality compared to A*, or would THPA* work well with dynamic graphs? These are all things that can be worked on more.

As future work there are also possible extensions of THPA* that could be made to both increase path quality and to use it in a dynamic world. There are also grid-based path finding techniques that have a potential with triangle-based graphs.

References

[1] E. W. Dijkstra, *A Note on Two Problems in Connexion with Graphs.* Numerische Mathematik 1, (1959), 269–271

[2] Adi Botea, Martin Müller, and Jonathan Schaeffer, *Near Optimal Hierarchical Path-Finding.* `http://webdocs.cs.ualberta.ca/~jonathan/PREVIOUS/Grad/Papers/jogd.pdf`, (2004)

[3] Alex Kring, Alex J. Champandard, and Nick Samarin, *DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds.* `https://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/viewFile/2131/2543`, (2010)

[4] M. Renee Jansen and Michael Buro, *HPA* Enhancements.* `https://www.aaai.org/Papers/AIIDE/2007/AIIDE07-017.pdf`, (2007)

[5] Douglas Demyen and Micheal Buro, *Efficient Triangulation-Based Pathfinding.* `https://www.aaai.org/Papers/AAAI/2006/AAAI06-148.pdf`, (2006)

[6] Marcelo Kallmann, Hanspeter Bieri, and Daniel Thalmann *Fully Dynamic Constrained Delaunay Triangulations.* `http://infoscience.epfl.ch/record/100269/files/Kallmann_and_al_Geometric_Modeling_03.pdf`, (2004)

[7] Marcelo Kallmann, *Navigation Queries from Triangular Meshes.* `http://graphics.ucmerced.edu/papers/10-mig-navq.pdf`, (2010)

[8] D. T. Lee and B. J. Schachter, *Two Algorithms for Constructing a Delaunay Triangulation.* `http://www.personal.psu.edu/cxc11/AERSP560/DELAUNEY/13_Two_algorithms_Delauney.pdf`, (2010)

Appendix A
Map Data

battleground - Number of paths: 1224

Time(ms.)	Avg.	Min.	Max.
A*	721.7645	0.0058	5462.8300
HPA*	25.7320	3.0616	412.4390
THPA*	14.4813	1.7237	90.1809

blastedlands - Number of paths: 1226

Time(ms.)	Avg.	Min.	Max.
A*	787.2309	0.0020	19246.6000
HPA*	33.4231	4.7625	964.1600
THPA*	17.9040	1.7101	223.9190

bloodvenomfalls - Number of paths: 1249

Time(ms.)	Avg.	Min.	Max.
A*	748.5058	0.0020	38458.2000
HPA*	12.2347	3.0616	811.4950
THPA*	8.2788	1.7706	181.0130

bootybay - Number of paths: 1059

Time(ms.)	Avg.	Min.	Max.
A*	463.9509	0.0024	21924.8000
HPA*	4.0865	3.4018	402.2915
THPA*	2.6684	1.6934	95.7104

darkforest - Number of paths: 1231

Time(ms.)	Avg.	Min.	Max.
A*	234.5898	0.0017	13502.8000
HPA*	8.1963	4.0821	571.3050
THPA*	4.8745	1.7118	174.8430

deadwaterdrop - Number of paths: 1029

Time(ms.)	Avg.	Min.	Max.
A*	68.4762	0.0020	9546.1300
HPA*	2.6421	4.7625	290.5195
THPA*	2.0747	1.7352	76.9249

divideandconquer - Number of paths: 1245

Time(ms.)	Avg.	Min.	Max.
A*	285.4982	0.0020	36566.8000
HPA*	12.2892	4.0821	701.1550
THPA*	7.0074	1.7223	199.4720

dragonfire - Number of paths: 1268

Time(ms.)	Avg.	Min.	Max.
A*	325.2318	0.0027	26954.0000
HPA*	6.3029	4.7625	759.0800
THPA*	4.2273	1.7230	199.8930

drywatergulch - Number of paths: 1219

Time(ms.)	Avg.	Min.	Max.
A*	95.9101	0.0017	17002.4000
HPA*	5.6340	4.0821	560.5550
THPA*	2.8496	1.6397	259.6570

duskwood - Number of paths: 1264

Time(ms.)	Avg.	Min.	Max.
A*	146.5417	0.0020	17733.0000
HPA*	3.5319	3.4018	636.9250
THPA*	1.5237	1.6798	141.4800

dustwallowkeys - Number of paths: 1266

Time(ms.)	Avg.	Min.	Max.
A*	286.1639	0.0024	20293.0500
HPA*	4.2969	4.0821	883.3650
THPA*	3.2255	1.7087	257.3240

frostsabre - Number of paths: 1180

Time(ms.)	Avg.	Min.	Max.
A*	115.5572	0.0017	21177.2000
HPA*	2.3937	4.0821	429.5395
THPA*	1.2194	1.6982	92.1736

gardenofwar - Number of paths: 1265

Time(ms.)	Avg.	Min.	Max.
A*	148.9618	0.0020	18733.5000
HPA*	3.8119	4.0821	738.8200
THPA*	2.6446	1.7142	171.3030

gnollwood - Number of paths: 1152

Time(ms.)	Avg.	Min.	Max.
A*	65.7629	0.0020	7973.7300
HPA*	1.5489	4.7625	367.5765
THPA*	0.8752	1.7053	126.6030

golemsinthemist - Number of paths: 1261

Time(ms.)	Avg.	Min.	Max.
A*	99.9157	0.0017	16069.0000
HPA*	2.1441	2.7214	484.3845
THPA*	0.9972	1.6699	175.4460

harvestmoon - Number of paths: 1236

Time(ms.)	Avg.	Min.	Max.
A*	72.9047	0.0017	12899.4000
HPA*	3.5121	4.7625	912.7100
THPA*	2.1306	1.6985	217.4690

heart2heart - Number of paths: 1025

Time(ms.)	Avg.	Min.	Max.
A*	169.7744	0.0024	25147.0000
HPA*	0.6697	4.0821	296.4520
THPA*	0.2877	1.6965	65.7524

hillsofglory - Number of paths: 1203

Time(ms.)	Avg.	Min.	Max.
A*	93.7890	0.0017	18345.8000
HPA*	0.6236	3.4018	152.7205
THPA*	0.3179	1.6645	44.4941

icecrown - Number of paths: 1271

Time(ms.)	Avg.	Min.	Max.
A*	134.3767	0.0017	29333.9000
HPA*	2.2103	4.0821	828.1450
THPA*	1.2620	1.7118	221.9320

isleofdread - Number of paths: 1255

Time(ms.)	Avg.	Min.	Max.
A*	65.1356	0.0017	14202.6000
HPA*	2.3105	3.4018	741.5850
THPA*	1.1942	1.6829	220.8400

losttemple - Number of paths: 1177

Time(ms.)	Avg.	Min.	Max.
A*	59.1956	0.0020	12849.7000
HPA*	1.2415	4.0821	371.3270
THPA*	0.8729	1.7315	117.2910

moonglade - Number of paths: 1266

Time(ms.)	Avg.	Min.	Max.
A*	104.8886	0.0020	33606.5000
HPA*	1.2843	4.0821	517.7850
THPA*	1.1136	1.7189	140.2680

mysticisles - Number of paths: 1232

Time(ms.)	Avg.	Min.	Max.
A*	76.1232	0.0017	28239.1000
HPA*	1.6530	4.7625	842.7100
THPA*	1.0314	1.7022	192.8400

nighthaven - Number of paths: 1182

Time(ms.)	Avg.	Min.	Max.
A*	141.8044	0.0017	21347.0500
HPA*	0.5580	4.0821	231.2010
THPA*	0.3976	1.6863	57.0732

petrifiedforest - Number of paths: 1155

Time(ms.)	Avg.	Min.	Max.
A*	63.0645	0.0017	26572.6000
HPA*	2.2589	3.4018	1062.6900
THPA*	1.2567	1.6699	145.8100

plaguelands - Number of paths: 894

Time(ms.)	Avg.	Min.	Max.
A*	44.7297	0.0020	15354.1000
HPA*	0.6310	4.7625	336.0115
THPA*	0.2585	1.7818	70.8170

plainsofsnow - Number of paths: 953

Time(ms.)	Avg.	Min.	Max.
A*	40.7950	0.0017	18205.7000
HPA*	0.2338	3.4018	137.1270
THPA*	0.1078	1.6114	28.7667

plunderisle - Number of paths: 1261

Time(ms.)	Avg.	Min.	Max.
A*	62.6181	0.0017	24463.6000
HPA*	0.9126	4.7625	526.8200
THPA*	0.6036	1.8230	105.4880

riverrun - Number of paths: 1153

Time(ms.)	Avg.	Min.	Max.
A*	89.5400	0.0017	38714.9000
HPA*	2.3214	4.7625	705.7475
THPA*	1.1700	1.6158	148.1635

scorchedbasin - Number of paths: 1176

Time(ms.)	Avg.	Min.	Max.
A*	31.1234	0.0020	11132.8000
HPA*	0.7628	2.7214	326.1770
THPA*	0.4227	1.6703	81.0505

stromguarde - Number of paths: 280

Time(ms.)	Avg.	Min.	Max.
A*	2.8035	0.0238	13667.9000
HPA*	0.1028	19.3901	264.2255
THPA*	0.0958	1.8438	60.9838

swampofsorrows - Number of paths: 1151

Time(ms.)	Avg.	Min.	Max.
A*	38.0936	0.0020	27470.2000
HPA*	1.9809	3.0616	990.5000
THPA*	1.0555	1.7145	137.0700

thecrucible - Number of paths: 1099

Time(ms.)	Avg.	Min.	Max.
A*	18.5809	0.0020	4713.3000
HPA*	0.9772	4.7625	495.9540
THPA*	0.5190	1.7444	111.2310

theglaive - Number of paths: 1061

Time(ms.)	Avg.	Min.	Max.
A*	36.0834	0.0024	36365.5000
HPA*	0.5312	4.0821	388.4885
THPA*	0.2976	1.8893	96.7609

timbermawhold - Number of paths: 1262

Time(ms.)	Avg.	Min.	Max.
A*	87.1891	0.0020	29028.9000
HPA*	1.1211	4.0821	744.6550
THPA*	0.8063	1.8621	176.8300

tranquilpaths - Number of paths: 1216

Time(ms.)	Avg.	Min.	Max.
A*	38.5792	0.0017	15324.9000
HPA*	0.9746	4.7625	510.7050
THPA*	0.5855	1.9230	125.0420