Near Optimal Hierarchical Pathfinding using Triangulations

Author: Zal Bhathena
Advisors: Ahmed Gheith,
Donald Fussell
Department of Computer Science
University of Texas at Austin
zalbhathena@gmail.com

Abstract

In game programming, A* is the standard algorithm used for pathfinding. For large search spaces, A* can consume large amounts of CPU and memory resources. HPA*, an algorithm that alleviates these problems, represents the search space as a grid and divides the grid into clusters, then does an on-line search to find the clusters to travel through. This allows HPA* to calculate the majority of the absolute path asynchronously while the agent is in transit, reducing the start up delay that would occur from standard A*. We present THPA*, an extended HPA* which provides many features absent from HPA*. THPA* has support for search space modifications at run time, finding paths for non-zero radius agents, and handling polygonal obstacles.

Contents

Chapter 1

Introduction

In game programming, pathfinding is a very important area. Often times, pathfinding can become a bottle neck for performance, especially in real time strategy (RTS) games. In RTS games, there can be upwards of a thousand agents trying to navigate to their destination while avoiding collisions and attempting to take relatively short routes. Calculating these paths is expensive and can consume large amounts of CPU time, even if done correctly. The more traditional approach to pathfinding is to use an algorithm, typically A*, to find a path avoiding all static obstacles (terrain, structures, map boundaries, etc.) and to use steering forces to avoid dynamic obstacles (other agents or dynamic portions of the map). The A* algorithm will create sub goals for an agent to travel to and the steering forces will determine the velocity vector the agent has on its way there.

1.1 Near-Optimal Pathfinding

Pathfinding is not only resource-intensive, but it is also almost impossible to do optimally and quickly for large numbers of dynamic agents. This is the reason for non-optimal pathfinding algorithms that provide near-optimal paths much quicker than optimal pathfinding algorithms. HPA* (Hierarchical Pathfinding A*)[1], created by Botea, Muller and Schaeffer, is one of these near-optimal pathfinding algorithms that has been used in the RTS genre.

The idea behind this algorithm is to first divide the map of obstacles into a grid, where each space is either empty or contains an obstacle. From there, the grid is divided into disjoint rectangles called clusters, and the positions (inter-nodes) on the grid that connect clusters to each other are found. The HPA* algorithm will initially calculate the high level path over all inter-nodes (the distances between inter-nodes are cached during a pre-processing phase) and then asynchronously calculate the path between each subgoal. This allows the usually slow A* to be computed only over a much smaller search space, while the more intensive parts are calculated asynchronously while the agent is traveling between subgoals.

However, there are some pitfalls to HPA*. First, there can only be rectangular obstacles as it is not intuitively obvious how to represent a non-rectangular obstacle in the grid. Second, obstacles cannot be removed from the map as many of the inter-node distances would need to be recalculated. Third, HPA* cannot find paths for agents of arbitrary radius. Finally, the search space size increases linearly with the area of the map. This is undesirable for large maps with relatively few obstacles in them.

1.2 Triangulation A*

Triangulation A* (or TA*), another variation on A*, represents the search space as a triangulation of the map of obstacles. Unlike HPA*, TA* produces optimal paths while reducing the running time compared to A*.

TA* first triangulates the map using the vertices for the boundary of the map and each obstacle. A triangulation is a subdivision of a graph into simplices (which are always triangles for the 2-dimensional case which we will be examining). There are different ways to triangulate a graph, but most applications use Constrained Delaunay Triangulations for their nice properties such as tending to avoid skinny triangles. The resulting Delaunay triangulation is then treated as the search space for the map. TA* treats each triangle as a vertex and each pair of triangles that share an unconstrained (or non-obstacle) edge as a graph edge. Finally, standard A* is performed over the search space with some heuristic to account for the triangle search space.

There are many advantages to TA* over HPA*. First, TA* can use dynamic constrained Delaunay triangulations (DCDTs) to build the search space. DCDTs are incrementally built and can dynamically add and remove constraints [4], allowing for map obstacles to be added or destroyed (such as a bridge being demolished or a building being raised). Second, TA* can support any polygonal obstacles at arbitrary locations on the map (rather than rectangles that must be located on grid coordinates). Finally, TA* can find paths for non-zero radius obstacles[2].

1.3 THPA*

Although TA* may seem to be vastly superior to HPA*, we can draw inspiration from the principles of HPA* to speed up TA*. This is the foundation of our work in this paper and the idea behind the THPA* algorithm that will be outlined. We aim to be able to partition a triangulated graph such that the number of inter-edges is significantly smaller than the total number of edges in the graph. We will also use the pre-processing that HPA* requires to cache the edge lengths for agents of different radii. In addition, we will allow this partition to change dynamically as map obstacles are removed and added and update the cache accordingly. These changes will allow us to implement the ideas that make HPA* powerful while also keeping the speed and flexibility of TA*.

The rest of this thesis is organized as follows. §2 and §3 cover background information pertaining (respectively) to the HPA* algorithm and the TA* algorithm. §4 presents the THPA* algorithm. §5 discusses modifications to THPA* to allow for non-zero radius agents. §6 discusses search space modifications after pre-processing ends. §7 discusses the experiments performed and an analysis of THPA*'s performance. Finally, §8 concludes.

Chapter 2

HPA*

In game programming, optimal paths are often too expensive to be calculated. This problem naturally suggests the usage of near-optimal pathfinding for increased speeds. One such algorithm, HPA*, is commonly found in pathfinding implementations for games, such as in [8]. This chapter will discuss the HPA* algorithm and its implementation, advantages, and shortcomings.

2.1 Implementation

In Botea, Muller, and Schaeffers paper Near Optimal Hierarchical Pathfinding, they create the HPA* algorithm which represents the search space as a grid and divides the grid into clusters, then doing an *on-line search* to find the clusters to travel through.

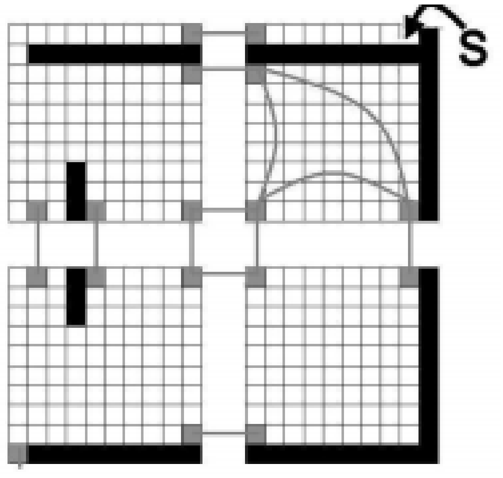
2.1.1 Pre-processing the grid

The first step of HPA* is to define a topological abstraction for the map. First, the map is divided into a grid with $n \times m$ tiles. The map is then divided into disjoint rectangles called *clusters* that contain elements from the grid. For each border between two adjacent clusters, a set of entrances connecting them is defined. An entrance is the set of all tiles along the border of clusters C_1 and C_2. Consider the set of all tiles l_1 and l_2 on the border connecting C_1 and C_2. We denote $symm(t)$ as the set of all tiles in C_2 that share a side with any tile $t \in C_1$. If $t \in C_1$ then the tiles in $symm(t)$ are contained in C_2. An entrance e is the set of tiles that fit the following conditions:

1. The border limitation condition: $e \subset l_1 \cup l_2$. This condition states that an entrance is defined along and cannot exceed the border between two adjacent clusters.
2. The symmetry condition: $\forall_{t \in l_1 \cup l_2} t \in e \Leftrightarrow symm(t) \in e$
3. The obstacle free condition: an entrance contains no obstacle tiles.
4. The maximality condition: an entrance is extended in both directions as long as the previous conditions remain true.

From here, we now build the *hierarchical graph*, $G_{hier} = (V_{hier}, E_{hier})$ with weight function w that maps from edges to real numbers, from all entrances. For each entrance we define a set of vertices called *abstract nodes*. Each entrance must have at least one abstract node and for any abstract node t in entrance e, $t \in V_{heir} \Rightarrow symm(t) \in V_{hier}$. For each abstract node t and its symmetric abstract nodes $symm(t)$, we define *inter-edges* between t and each of the nodes in $symm(t)$. Note that for HPA* that each inter-edge e has $w(e) = 1$. Also, for any cluster C and for any two abstract nodes $t_1, t_2 \in C$, we define *intra-edges* between t_1 and t_2. For any intra-edge $e = (v_1, v_2)$, $w(e)$ will be the optimal distance between v_1 and v_2 found by A* pathfinding.

Figure 2.1: Abstract nodes, intra-edges, and inter-edges are shown in light grey. Intra edges are only shown for the top right cluster.



During pre-processing, the hierarchical graph is built and the weight of all the edges are cached. For the case of HPA*, we only need to cache the intra-edge weights.

2.1.2 On-line Search

The first phase of on-line search inserts start tile S and goal tile G into the hierarchical graph. This is done by connecting S and G to the abstract nodes contained in their respective clusters. After they have been inserted, we can find the path between S and G in the abstract graph. This provides an abstract path from which we can find the absolute sequence of moves required to navigate through the map in a path-refinement step. In addition, we can use path smoothing to improve the path given in the path-refinement step.

2.2 Advantages and shortcomings

The advantage provided by HPA* is the low runtime of the abstract path generation. We can find the abstract path and delay finding the entire absolute path. This allows us to asynchronously calculate the absolute path for each step of the abstract path. If we choose good sizes for our clusters (clusters with $O(\sqrt{nm})$ area), then we can potentially reduce the running time of our pathfinding by $O(\sqrt{n})$.

However, HPA* comes with many restrictions. First, we can only use a grid with rectangular obstacles that fit neatly into the tiles. Non-rectangular obstacles may violate the obstacle free condition. Second, we cannot modify the map (such as destroying an obstacle such as a bridge) without recalculating many of the cached values that were calculated during pre-processing, creating huge performance bottlenecks. Third, there is no intuitive way to handle agents of non-zero radius. Finally, the search space size increases proportionally with the area of the map. This is not desirable as we can have large barren maps that give long run times where there are obvious paths that could be calculated quickly using other pathfinding techniques.

Chapter 3

TA*

TA* is a pathfinding algorithm that represents an environment as a set of walkable triangles and finds the shortest path from one point to another. This algorithm has support for insertions and deletions of obstacles, as well as non-zero radius obstacles and is also relatively fast. These features of TA* offer what HPA* cannot.

3.1 Triangulations and Environment Representation

In the previous chapter, we looked at HPA* and a grid environment representation for the map. Now we will discuss triangulations as a form of environment representation in preparation for discussing TA*.

3.1.1 Grids vs Triangulations

For any variation of A*, a search space must be created from a map. There are several ways to represent an environment, one of them being a grid representation as covered in the previous chapter. This representation is one of the most common, yet least effective ways to represent an environment. Obstacles cannot always fit neatly into a grid, and usually spill outside the bounds of each tile. For example, in 3.2 the slanted rectangles cannot be accurately represented and there are tiles that contain only parts of an obstacle. This gives pathfinding algorithms issues in deciding what to do with agents that want to navigate through a tile that has an obstacle in it. Should we treat this tile as being impassable? Should we weight it based on the amount of space it has available? It is obvious that this representation has a whole host of issues.

Another approach that has been popular amongst game developers is to triangulate the search space with some constraints. A triangulation is a graph that is made up entirely of triangles. In a constrained triangulation, edges are constrained and unconstrained edges are added between end points of the constraints until the graph is triangulated. 3.3 shows the environment from 3.1 with a triangulation representation.

The major advantage from this approach is that each triangle is either entirely filled by an obstacle, or entirely free from obstacles. This allows us to create a search space that is easily navigated. In addition, we have also found a solution to the earlier problem with HPA* where the search space size was $O(n * m)$. The search space size for a triangulated representation of an environment is $O(|V|)$ for any graph $G = (V, E)$.

Figure 3.1: Environment to be represented.

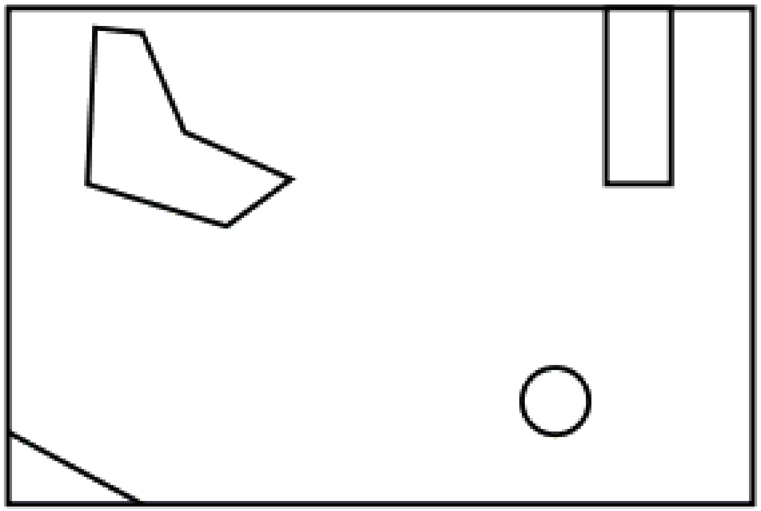


Figure 3.2: Grid representation of the environment.

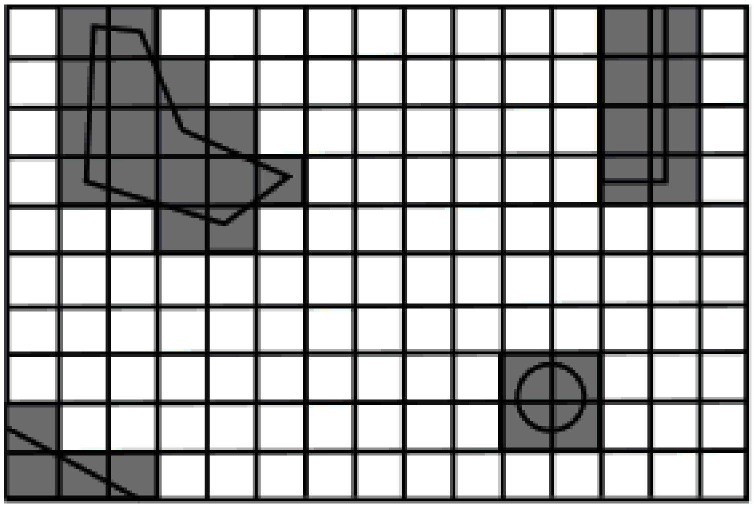
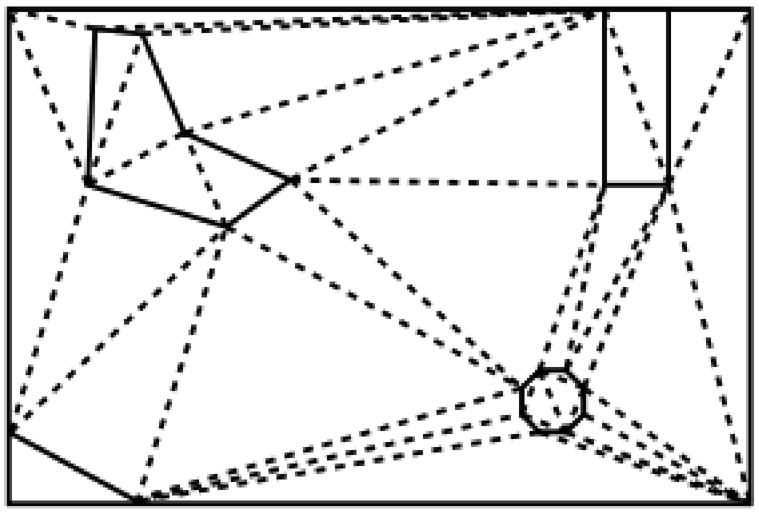


Figure 3.3: Constrained triangulation representation of the environment.



Lemma 1: From Euler's formula, we know that $V - E + F = 2$. We also know in any graph with at least 3 vertices, every face must be a triangle because it is impossible to add a unique edge to an existing triangle. We know that each edge borders two triangles so $3F = 2E$. Therefore

$$F = \frac{V}{2} - 1 = O(V)$$

■

For sparse environments, this gives us a much better search space size. However, for densely packed environments like mazes with very thin corridors, we may see poorer performance.

Also note that $E = O(V)$ (this should be obvious). This fact will come in handy when discussing THPA* in §4.

3.1.2 Delaunay Triangulations

Delaunay Triangulations are a specific set of triangulations where the minimum interior angle of all triangles in the triangulation is maximized. This is equivalent to the other requirements given in [4]. Delaunay triangulations also tend to avoid skinny triangles. Notice how 3.5 is made of much skinnier triangles than 3.6. Delaunay triangulations are a much studied topic in computational geometry and there are many $O(n \log n)$ algorithms to build both Delaunay and Constrained Delaunay Triangulations for n vertices.

Kallmann's Dynamic Constrained Delaunay Triangulation Algorithm

TA* uses Kallmann's Dynamic Constrained Delaunay Triangulation (DCDT) algorithm to generate the triangulation representation of the environment. This algorithm is an incremental algorithm that allows for deletions and insertions of constraints. This is especially useful in games where the map is constantly changing (structures could be being built and destroyed constantly), and provides a solution to one of HPA*'s shortcomings.

Figure 3.4: A set of points to triangulate.

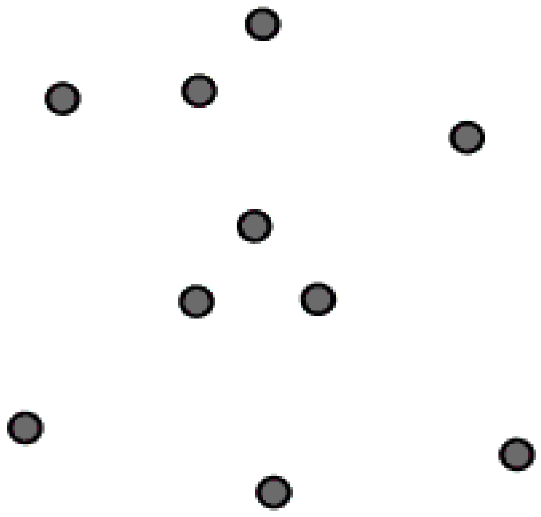


Figure 3.5: The same set of points non-Delaunay triangulated.

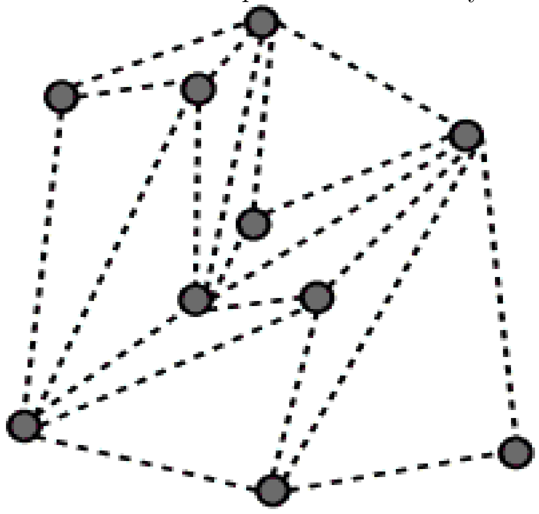
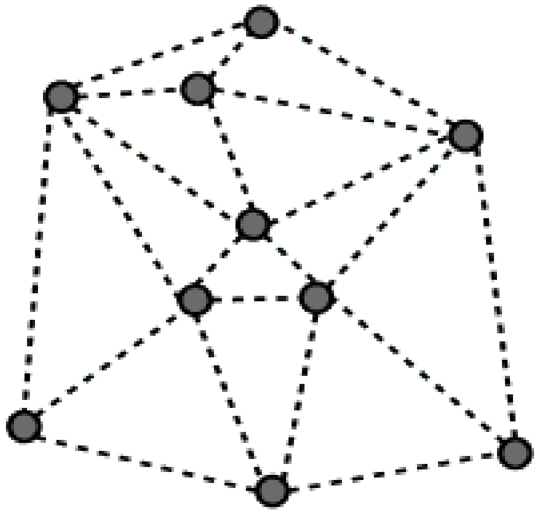


Figure 3.6: The same set of points Delaunay triangulated.



The algorithm takes $O(n^{4/3})$ time to construct a triangulation, but can reach $O(n \log n)$ if dedicated data structures are used. The implementation for this algorithm is quite involved and described in [4].

3.2 Triangulation A*

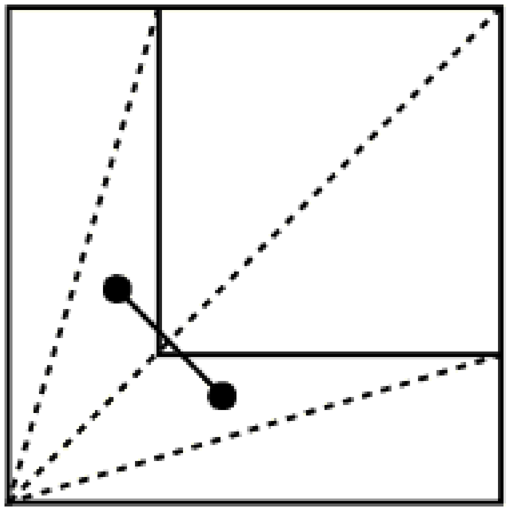
3.2.1 A* Introduction

Here we detail A* and some of the terms used for the algorithm. We start with the search space S and each state $s \in S$. For each s we have $Succ(s) \subset S$ where every $s' \in Succ(s)$ can be reached by s by a single action a. Also, there is a cost $c(s, a, s') \geq 0$ associated with each action. During the actual search algorithm, s' can be denoted as the child state of parent state s.

The g-value of a search state is the distance traveled so far from its ancestor states. The h-value of a search state is the distance remaining to the goal. This is found using some heuristic. We want this heuristic to be both *admissible* and *consistent*. A heuristic is admissible if it never overestimates the distance to the goal. More formally, $\forall_{s \in S} h(s) \leq h{*}(s)$ where $h * (s)$ is the actual shortest distance between s and the goal. A heuristic is consistent if it follows a version of the triangle inequality where for any state s and its successor s',$h(s) \leq h(s') + c(s, a, s')$. A heuristic that is consistent is also always admissible. In addition, if a heuristic is both consistent and admissible, it guarantees that the resulting A* search is optimal. Finally, the f-value is calculated by $f(s) = g(s) + h(s)$. This is equivalent to the entire estimated cost from the start to the goal and is always an underestimate with admissible and consistent heuristics.

A* starts by taking the start state and adding it to a priority queue that is sorted by f-values of its elements. During each step, the state with the lowest f-value is removed from the priority queue and if it is not the goal state, its successors are added to the priority queue. If the goal state is found we have found a path from the start to the goal, and with

Figure 3.7: The path between the midpoints of these triangles goes through a third triangle.



a good heuristic A* is guaranteed to be optimal. If a goal state is not found by the end of the algorithm, there is no possible path from the start to the goal.

3.2.2 Naive A* over a Triangulated Search Space

For a triangulated search space, it is not intuitive how to implement A*. It is not obvious what the g-value, h-value and cost function should be. In a grid, it is obvious what the path between any two states should be (a line between their midpoints), but this same technique does not work for triangulated search spaces. If we assume that objects will always travel over a simply defined path, such as through the midpoints of each triangle, then we run into several problems. First, the path between two states s and $s' \in Succ(s)$ may not be completely contained in s and s', such as in 3.7. Second, the paths produced are poor estimates and we won't even know the exact distance traveled to a particular triangle, even if we use path smoothing techniques.

Next, we try another approach. We assume that objects travel through the midpoints of unconstrained edges. We still run into the same problem of poor paths being generated. Even after path smoothing, a non-optimal path is generated, as seen in 3.9. This happens because (using Euclidean distance as the h-value), the search neglects to expand triangles whose actual path is optimal but has sub-optimal estimated path.

3.2.3 Triangulation A*

[2] creates TA*, which is a variation of A* that finds optimal paths in a triangulated search space, avoiding the problems posed by the naive A* approaches. The search starts by placing the initial triangle s_{start} into the priority queue with $g(s_{start}) = 0$ and $h(s_{start})$ being the Euclidean distance from the start to the goal point.

The rest of the algorithm proceeds similarly to A* with several modifications. At each step, the state with lowest f-value is taken off the priority queue and expanded. The successor function returns a child states for every triangle sharing an unconstrained edge

Figure 3.8: Triangle midpoints produce poor paths.

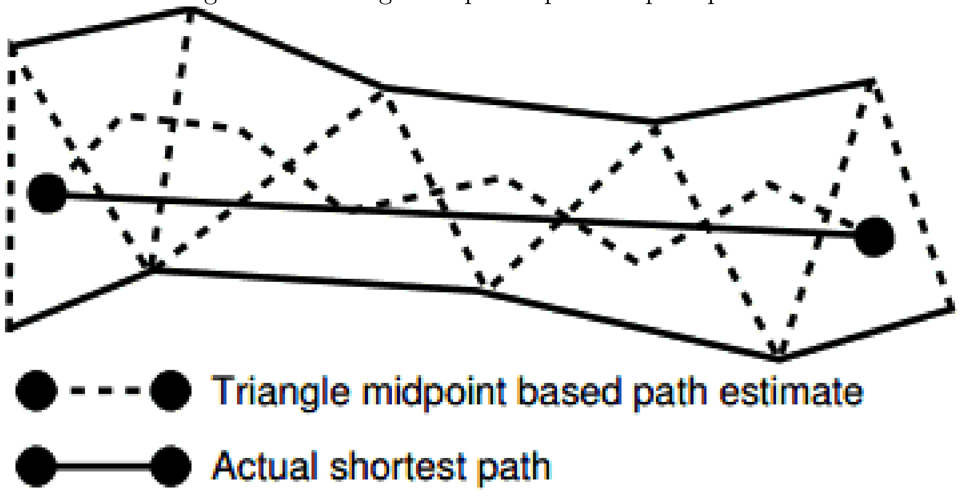
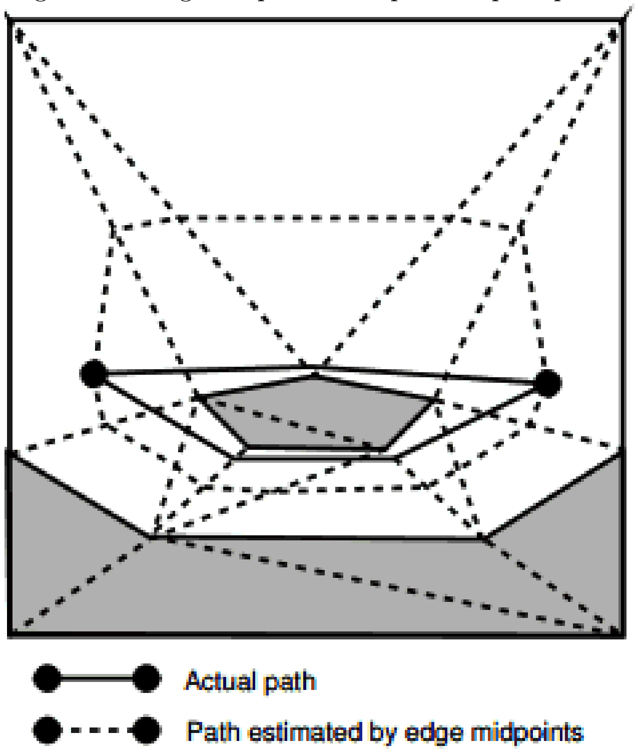


Figure 3.9: Edge midpoints also produce poor paths.



with the current triangle. In addition, the child state will not be generated if it is already an ancestor of the current triangle. This is important in preventing cycles and speeding up the search. This edge is used to calculate the g-value and h-value of these states. The h-value is simply the Euclidean distance from the goal to the closest point to the goal in the edge.

The g-value is calculated as the maximum of a number of lower bounds. This is done to provide an accurate estimate without overestimating. The estimates for a state s and successor s' for an object of radius r are as follows:

1. The distance between the goal and the closest point on the edge between s and s'
2. $g(s) + r\theta$ where θ is the angle formed by the entry edges for s' and s
3. $g(s) + (h(s) - h(s'))$

The maximum of these lower bounds provides an accurate g-value without overestimating. These modifications to A* provide an optimal path from each search (a more detailed look into this guaranteed is given in [2]) .

Chapter 4

THPA*

In this chapter we introduce Triangulation HPA* (THPA*), a pathfinding algorithm we develop that provides speedups and additional features for HPA*. THPA* first incrementally creates a triangulated search space and partitions it into clusters and an abstract graph during pre-processing. When the algorithm is run, it finds the abstract path of subgoals on the abstract graph. After the abstract path is found, the absolute path is calculated asynchronously. Finally, every time an agent reaches a subgoal, the absolute path for the agent up until this subgoal is returned. The details algorithm will be described in the following chapter and all additional details will be covered in the proceeding chapters.

4.1 Pre-processing

Similar to HPA*, we start with a pre-processing step. During this step, all the work needed to build the triangulation and abstract graph is done.

4.1.1 Triangulating the Search Space

First we build the search space. We first assume that the map only contains polygonal obstacles. If there any non-polygonal obstacles, we can approximate them with an appropriately shaped polygon. Next we create the triangulation. This can either be done incrementally or all at once. If done incrementally, we add constraints to the triangulation until the triangulation is complete.

One difference between TA* and THPA* is how we choose the search space from the triangulation. In TA*, the vertices are the triangles, and there are edges connecting triangles that share an edge. However, we cannot use TA* to calculate the cached edge weights because the TA* heuristic depends on the choice of start and goal states. Instead, THPA* treats each vertex in the triangulation as a vertex in the search space, and all valid edges (those that do not travel through obstacles) as edges between vertices.

4.1.2 Partitioning

There are three steps required to create the abstract graph. First we partition the search space into clusters. Then we find all abstract nodes and create the abstract graph from these nodes. Finally, we insert the intra-edges and inter-edges and cache their weights.

Partitioning is arguably the most difficult step of THPA*. In order to be able to make any guarantees about the speed of THPA*, we need to make the number of *inter-partition cuts* (equivalent to in meaning to inter-edges) and the size of each cluster as small as possible. However, the optimization version of either of these problems is NP-hard [3] so we have to use approximations. In addition, the algorithm we choose to partition the search space must also be incremental if we are to allow modifications to the search space. We detail several algorithms that fit these criteria in §5.

Any algorithm we choose to partition our search space must support the following operations:

1. Vertex Addition
2. Edge Addition
3. Vertex Deletion
4. Edge Deletion

We partition our graph as the triangulated search space changes. During each modification to the search space, we maintain a set of removed and added vertices and edges. Incremental Delaunay triangulation algorithms may modify the graph many times during vertex/edge addition or deletion, so we add every removed vertex to the *remove set* and every added vertex to the *add set.* Each time a vertex or edge is added to a set, we check if the other set contains the vertex or edge. If so, we remove that vertex or edge from both sets. This will give us a set of vertex/edge additions and deletions we need to apply to our partition.

We will now describe 2 partitioning algorithms. Although we only discuss two approaches, any partitioning algorithm that is incremental, tries to minimize the number of inter-partition cuts, and creates k clusters of approximately equal size can be used as the partitioning algorithm.

Multilevel k-way Partitioning

Multilevel partitioning is a very general partitioning algorithm that contains three steps: coarsening, partitioning, and uncoarsening. The coarsening step transforms the original graph into a much smaller one by collapsing vertices and edges. The partitioning step uses a partitioning algorithm to partition the smaller graph. Finally, the uncoarsening step maps the partitioned smaller graph back to the original graph and returns the partition.

Note that this approach avoids the actual partitioning problem by utilizing a separate partitioning algorithm for the partitioning step. As a result there are many different schemes for multilevel partitioning, each with their own advantages and heuristics. [6]

The particular multilevel partitioning algorithm we will look at is multilevel k-way hypergraph partitioning algorithm described in [7]. This algorithm has many components to it which makes it difficult to implement, so we use the METIS library [5] which provides an implementation of this algorithm. The library provides the ParMETIS library which supports incremental partitioning as well. The METIS library will be further discussed in §8 in which we use a Java implementation of this algorithm in our experiments.

Although this algorithm is fast in practice, it has the downside of providing no theoretical guarantees. The following algorithm attempts to provide a solution to this shortcoming.

Minimum Cut Trees

The second approach modifies an existing partitioning algorithm based on minimum cut trees to create an incremental min-cut partitioning algorithm. We have not implemented this algorithm due to its complexity but reference it because of its strong theoretical guarantees.

This algorithm maintains an *in cluster weight* and *out cluster weight* for every vertex, and attempts to maximize the in cluster weight and minimize the out cluster weight for all vertices. The algorithm must also be given a parameter α which serves as a lower bound for the total in cluster weight and an upper bound for the total out cluster weight. This is very convenient for THPA* as we aim to minimize the out cluster weight (identical to the number of inter-partition edges or inter-partition cuts). The algorithm itself is quite complicated and is described in [9].

The algorithm also gives several bounds on running time. For k being the weight of the largest cluster, all vertex additions and deletions can be handled in $O(k^3)$ time and edge additions and deletions can be handled in $O(1)$ time. This gives us an $O(nk^3)$ time algorithm for constructing the initial partition or performing n operations to modify the partition. Additionally, clusters produced using this algorithm tend to be small if $n = \Theta(m)$ where n is the number of vertices and m is the number of edges. Therefore if $k = \log n$ we have a polylogarithmic running time [9]. For triangulations, we know that every vertex can have no more than three edges. Also, most graphs for maps in game applications tend to have connected components with relatively large size, so we can safely state that the graphs that THPA* will apply to have $n = \Theta(m)$. Finally, [9] provides heuristics for reducing the time required to compute an edge/vertex addition or deletion to $O(k^2)$.

Note that the time complexity for cluster maintainance over edge and vertex updates is an open problem, giving us no guarantees on these running times.

4.1.3 Creating the Abstract Graph

Now we must create the abstract graph. During the partitioning phase, we will mark each vertex that is part of an inter-partition cut, and mark all inter-partition cuts. After partitioning is complete, we will loop over all marked vertices and edges and add them to our abstract graph.

4.1.4 Caching Edge Weights

Finally, we must calculate the weight for all inter-edges and intra-edges and cache them for use by weight function $w(e)$. First we find the intra-edge weights. For every cluster $C_i \in C$ (where C is the set of all clusters) and every $v_1, v_2 \in C$, we find the smallest f-value at $v2$ starting from the closest point in $v1$ using TA* and cache the path length.

Next we find the inter-edge weights. For each two abstract nodes from different clusters that are neighbors, we cache the Euclidean distance between them. Once we have cached the inter edge values, we are done with the pre-processing phase.

4.2 On-line Search

The on-line search for THPA* is very similar to the on-line search in HPA*, using A* to find the shortest path in the abstract path. Our cost function is the weight function

$w(e = (v_1, v_2))$ and our h-score is the Euclidean distance from the goal to closest point in the triangle for the state.

We first insert the start state and goal state into the abstract graph. We connect the start and goal states to the other abstract nodes in their cluster and find the intra-edges and inter-edges that need to be created, caching their values with the techniques used in pre-processing.

Finally, we add the start state to the priority queue and begin A*. The path returned by this A* is the abstract path. We treat every state in the abstract path as a subgoal.

4.3 Absolute Path

As soon as the abstract path has been found, we can begin calculating the absolute path asynchronously. The path from the start state to the first subgoal is the only part of the absolute path that needs to be calculated before an agent can begin moving. We will add this absolute path to the *calculated path*. When the agent reaches the end of the calculated path, it asks for the absolute path to the next subgoal and adds it to its calculated path until finally the agent reaches the goal.

Note that we assume that the time it takes for an agent to travel from one subgoal to another is much larger than the time it takes to calculate the absolute path between these subgoals. This is a safe assumption in game programming as agents usually move with very low speeds compared to the time it takes to calculate the path it needs.

To calculate the absolute path between subpaths, we use TA*. After the absolute path has been found, we can use the funnel algorithm (described in the following section) to find the points to travel through.

4.3.1 Pathsmoothing

After the absolute path has been found, we still do not have an absolute path but rather a list of triangles. We want to find the shortest path between the start and goal point inside these triangles. The final step for finding the absolute path is to use the funnel algorithm to find the actual points the agent must pass through. This list of points is what is returned when the agent asks for the next absolute path.

Funnel Algorithm

In [2], an algorithm for finding the shortest path inside the list of triangles is provided. Before discussing this algorithm, let us first define several terms. The *channel* is the simple polygon in which we want to find the shortest path. The *interior edges* are the unconstrained edges that will be crossed by the shortest path. The *interior triangles* are any triangles in the list that do not contain the start or goal point.

The algorithm provided is the *funnel algorithm*, which finds the path for a point object in time linear in the number of triangles. The funnel algorithm considers three structures: the *path*, the *apex*, and the *funnel*. The path is the line segments that show the shortest path up until the current point. The funnel is two line segments that show the area in which the next part of the shortest path must be in. The apex is the point in which the funnel and path meet.

The algorithm starts with an empty path, the apex as the start point, and the funnel being the segments that connect the start point to the first interior edge. The funnel is

Figure 4.1: The path, apex and funnel for the funnel algorithm

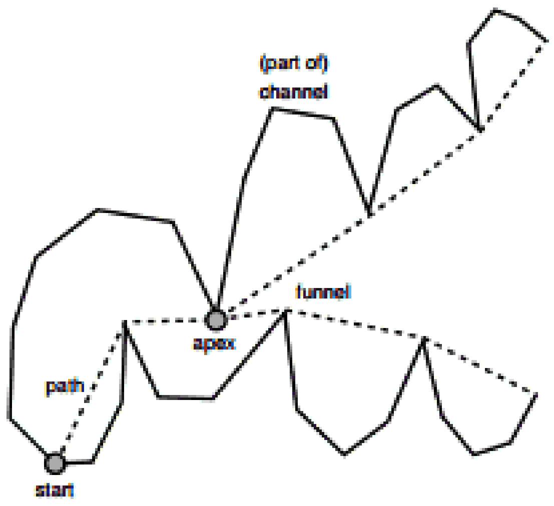
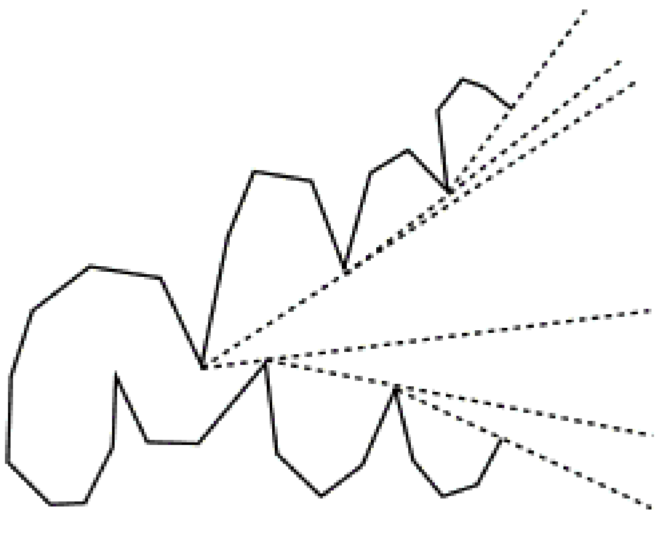


Figure 4.2: The wedges for each vertex.



stored in a deque stucture and each additional interior edge is processed with the vertex that has not been visited being stored in the correct side in the funnel deque. Vertices are popped off from that side of the funnel until the wedge in which the current vertex resides is discovered (wedges are shown in 4.2). After this, the vertex is added to the funnel deque on the correct side.

If the apex is popped off, the next vertex to be popped off becomes the new apex and a segment containing the old apex gets added to the path. Once the final interior edge is processed, we add the goal point and find the final path. The pseudo code for this algorithm is given in the appendix.

Chapter 5

Non-zero Agent Radius

HPA* and A* can only handle agents of zero radius. If a path using one of these algorithms was to be found and an agent with non-zero radius was to navigate along this path, it might collide with obstacles. In §3, we mentioned that TA* can handle pathfinding of agents with non-zero radius. This chapter will describe how TA* handles this and we adapt THPA* to handle pathfinding for non-zero radius agents.

5.1 TA* with Non-zero Radius Agents

In [2], a method is developed to calculate the maximum diameter of an object that can travel between two unconstrained edges shared by a triangle in a constrained triangulation. We will overview this method and provide pseudo code for the algorithm.

In order to find the maximum diameter of an object that can travel between two unconstrained edges shared by a triangle, we need to find the closest obstacle to the vertex joining these two edges. We assume that the agent is a circle with diameter r, but this approach can be applied to non-circular agents as well.

There are three possible cases that determine the maximum diameter. The first is if either $\angle CAB$ or $\angle CBA$, where C is the shared vertex, is either a right angle or obtuse. In this case, we assume without loss of generality that $\angle CAB$ is obtuse or right. The maximum allowed diameter is the length of edge b, the edge directly across from vertex B.

The second case is edge c, the edge across from C, is constrained. In this case, the maximum allowed diameter is the distance from C to the closest point on c.

The final case is where c is unconstrained. When a search enters a triangle via an edge, it checks the other two edges. We say that each edge is the segment between vertices U and V. If either $\angle CUV$ or $\angle CVU$ is not acute, the search terminates and the current upper bound returns. Next, we check the distance from C to the closest point on UV. If this distance is greater than the current upper bound, then the search terminates and the current upper bound is returned. Otherwise, the current upper bound is updated to be this distance. Finally, if UV is constrained, the search returns the current upper bound. Otherwise, the search proceeds to the adjacent triangle shared by this edge.

Figure 5.1: The triangle edges for which maximum diameter will be calculated for

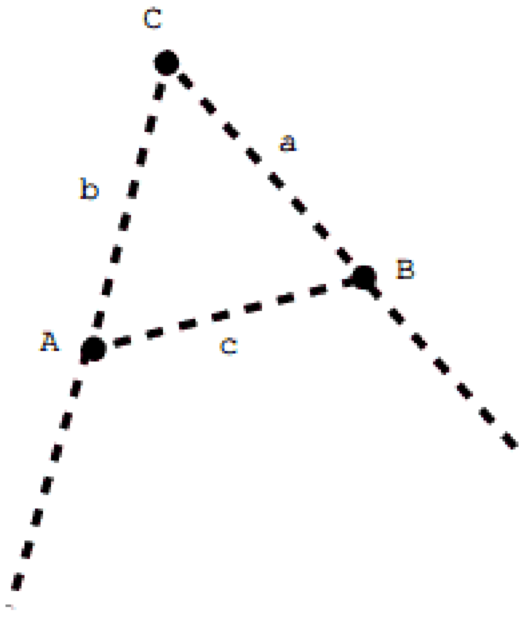


Figure 5.2: Angle $\angle CAB$ is obtuse

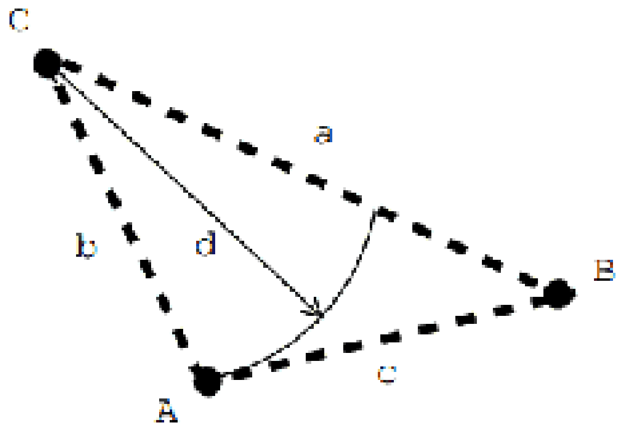


Figure 5.3: Edge C is constrained

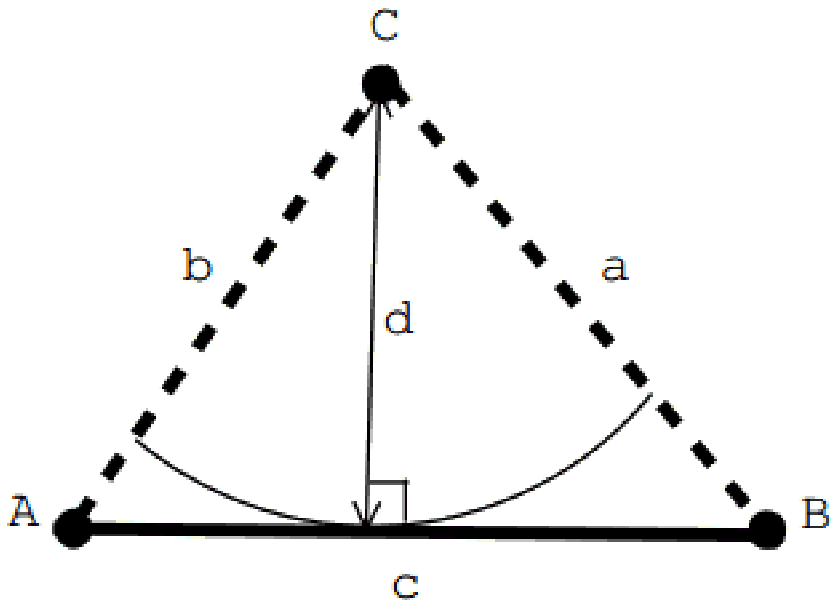


Figure 5.4: Edge C is unconstrained

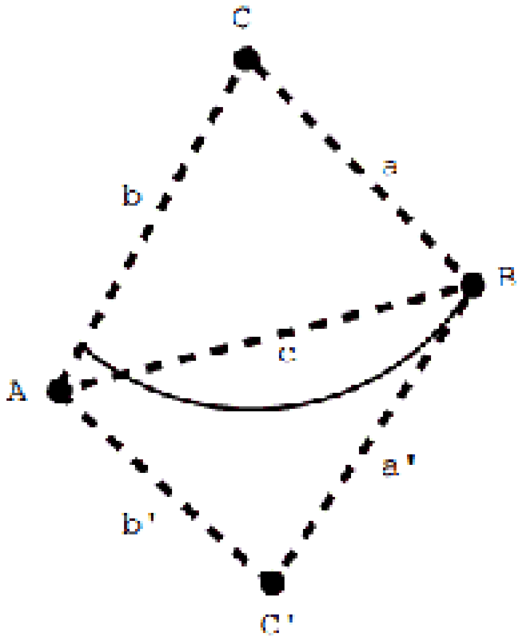
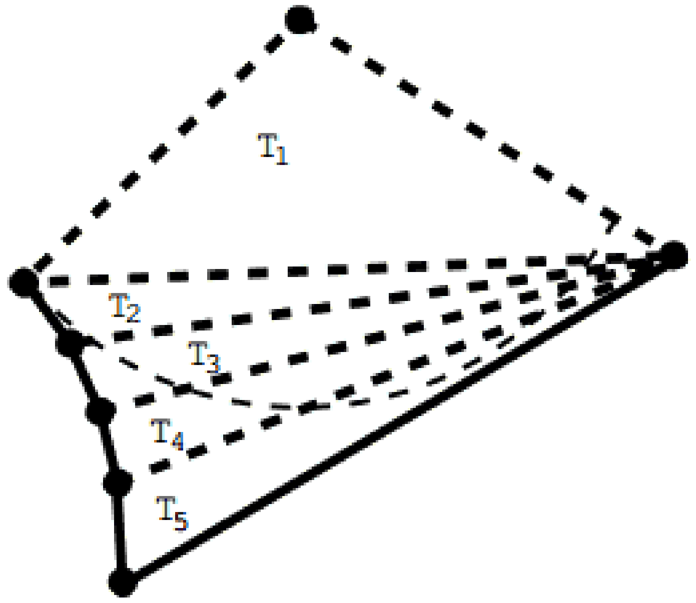


Figure 5.5: Skinny triangles producing poor running time for the final case.



The first and second cases run in constant time. The final case runs in worst case time proportional to the number of triangles in the triangulation. This case occurs when there are many skinny triangles, as shown in 5.5. However, this is an incredibly rare case in Delaunay Triangulations. The proofs for correctness and time complexity are found in [2].

Now all that remains is to apply the techniques we just mentioned to allow TA* to find paths for agents of a given radius. First, we must provide the agent radius r to TA*. Now during the state expansion step in TA*, we find the maximum diameter for an agent passing through the entrance edge for the current triangle and the edge shared by the current triangle and the expanded one. If it exceeds the agent diameter, we do not add it to the priority queue. This will prevent the agent from ever traveling through edges that are too small.

5.2 THPA* with Non-zero Agent Radius

Now that we have TA* that can find paths for agents with non-zero radius, let us adapt THPA* to also find paths for these agents. The main obstacle in using our modified TA* in THPA* is that we can no longer cache the edge weight independent of agent radius. We must now create a cache that provides edge weights for inter-edges and intra edges with a parameter for agent radius.

5.2.1 Pre-processing

There are several approaches to handling this problem. Each provides trade-offs between memory, preprocessing time, and flexibility.

Fixed set of Radii

The easiest is to only allow a fixed set R of agent radii. Now during pre-processing, we have to repeat the caching step $|R|$ times in order to cache the g-score for each agent radius. Despite its naivety, this approach is the most attractive for game applications. Many games will only have a small set of agent sizes, such as small, medium, large. This fits in nicely with this approach to our problem.

Ranges of Radii

Another approach is to store the edge weight for each edge based on the range the agent radius is in. For any edge e, we will define a set R of agent radii where $\forall_{r_i,r_j \in R} i > j \Leftrightarrow R_i > R_j$. We now have an inequality $R_0 < R_1 < ... < R_{|R|}$. If we are to ask for the edge weight for an element in the cache with agent radius r, we would find the first $r < R_i$ such that $i = |R|$ or $r \geq R_i + 1$. We would then access the associated value in the cache for this radius.

Let us now discuss how we find the edge weight for each value in R. For each any edge (v_1, v_2), calculate the edge weight for an agent of radius 0. As we traverse each vertex in the path using TA*, we store d_{min}, where d is the maximum diameter of an object that can travel between two unconstrained edges shared by that triangle. If d_{min} happens to be 0 (or whatever the starting d_{min} was for the proceeding runs), we set d_{min} to the radius plus some constant. We do this to account for the corner case where every pair of triangles along the path allows agents of radius 0 (or d_{min} for proceeding runs) to travel between them. Once we have finished this initial search, we begin finding the values in R. We now calculate the edge weight for an agent of radius d_{min}, set d_{min} to infinity, and find the new d_{min} in all vertices traveled along the path found. This d_{min} is then added to R as the last element i, and the value in the cache corresponding to R_i is the edge weight calculated. We now search for the edge weight for an agent with radius d_{min} and keep finding values in R until d_{min} is infinity. We provide this algorithm in the appendix.

The advantage of this approach is that we are now able to find the edge weight for agents of arbitrary radius. However, this comes at a cost. Creating the cache is now incredibly expensive. There can be up to $O(k)$ edge weights for each intra-edge in the abstract graph where k is the size of the cluster the intra-edge is located in. This can become incredibly expensive as we have to calculate the edge weight for every vertex in the abstract graph, which can vary and sometimes be $O(n)$ for certain partitioning algorithms without theoretical guarantees. This gives up to $O(nk)$ calculations of TA*, where each calculation takes $O(k^2)$ time, giving us an upper bound of $O(nk^3)$.

Ranges of Radii with constraints

The final approach we discuss is very similar to the ranges of radii approach. The only difference is that we constrain the ranges. This can be done by specifying that radii can only be integer values, or that they can be no larger than some specified constant. This approach doesn't necessarily give better upper bounds on running time, but can certainly improve running time.

Additional Details

An additional optimization we can make is to lazily cache the d values for each vertex in the abstract graph. This allows us to avoid the recomputation of a potentially linear time function.

Finally, if the triangulation is modified after the pre-processing phase, we have to handle the changes to the d-value cache and the edge weight cache. Approaches to this problem will be discussed in §7.

5.2.2 On-line Search

The changes to on-line search are trivial. The only change is that when the cache is accessed, a radius must be provided. Also, when the start and goal states are inserted into the graph, we only need to calculate the edge weight for the additional intra-edges and inter-edges for the agent radius we are using for the search.

5.2.3 Path smoothing

In the previous chapter, we introduced the funnel algorithm, which allowed us to find the absolute path within the path of triangles between two subgoals. However, this algorithm only works on point-sized agents. Fortunately, [2] provides a modified funnel algorithm that can find the shortest path within a list of triangles for agents of arbitrary radius. The algorithm is detailed in [2], but we provide the pseudo code in the appendix.

Chapter 6

Modifying the Search Space

In this chapter, we discuss modifications to the search space after pre-processing. Modifications to the search space occur when new obstacles are added to the map, or existing obstacles are modified or removed. This can occur quite frequently in games (such as when a building is destroyed or erected in a real-time strategy game). After each modification, we need to adjust the values in the cache and recalculate the paths for any agents that may be affected by the changes.

6.1 Cache Modifications

When the map is modified, it will have some number of polygons removed or added to it. The first step is to modify the triangulated search space using the incremental Delaunay triangulation algorithm discussed in 3.2.1. As we change the search space, we maintain add and remove sets as discussed in 4.1.2. The partition is now modified using the incremental partitioning algorithm we chose and the abstract graph is recalculated according to the steps given in 4.1.2.

Once the partition has been modified, we can now recompute the edge weights for specific values in the abstract graph. First, we find the set of clusters $C_{modified} \in C$ where every cluster in $C_{modified}$ contains at least one vertex or edge in the add set or remove set. Now that we have the set of clusters that have been modified, we recalculate the edge weight for all intra edges in these clusters. Next, we find the set of vertices in the add sets that belong to some inter-edge. We now recalculate edge weight for all inter-edges that contain these vertices. Finally, we remove all vertices in the remove set from the cache.

When handling non-zero radius agents, we may have to include an additional step. If we have cached d-values for search space, we recalculate the d-values for vertices in the add set and remove the vertices in the remove set from the cache.

6.2 Re-pathing for Agents

The final step after modifications to the search space and cache is to find all agents that would be affected that are currently navigating through the search space. At the bare minimum, we need to find all agents that now have invalid paths. The naive solution to this is to recalculate the path for all agents that navigate through any of the clusters that have had edges or vertices removed. This will prevent any agent from continuing along a potentially invalid path.

However, there are many disadvantages to this approach. First, there may be agents that can find a shorter path, but are now traveling along an incredibly sub-optimal one. Second, if there is a high density of agents, we will have to recompute a large number of paths, which is very expensive. Finally, we will be recalculating paths for agents that may not need to be recalculated.

Chapter 7

Experiments

7.1 Test suite

The test suite is comprised of 4 algorithms (A*, HPA*, TA*, and THPA*) over two types of maps (each with either uniformly distributed or nonuniformly distributed obstacles).

Nonuniform maps are generated between 50 and 2000 obstacles of varying size placed pseudo-randomly on integer coordinates of the map. Nonuniform maps are always 300 units wide and tall. Uniform maps are created by placing distributing a block on each coordinate with even x and y coordinates. Uniform maps range from size 10 to 100.

A* creates its search space by dividing the map into a grid where each node is a one width square on integer coordinates and its neighbors are the nodes that share a side with it. HPA* chooses its search space identically to A*. Clusters are of width and height equal to $\sqrt{n}$ where n is the size of the search space. The heuristic chosen is Euclidean distance for both A* and HPA*.

TA* uses Kallmann's DCDT library, TriPath, to create the search space from the map. TriPath also contains a TA* implementation that is used to find shortest paths in the search space.

The search space for THPA* is found identically to that of TA*. The search space is partitioned into $\sqrt{n}$ parts, where n is the number of abstract nodes in the abstract graph, using the Galois library, a java implementation of METIS [10]. The abstract nodes are those that are connected to nodes in other clusters. All intra-edges and inter-edges weights are found using A*. The abstract path is found by A* with the Euclidean distance heuristic over the abstract graph. The absolute path is found by TA* with the funnel algorithm for path smoothing.

All code (with the exception of TriPath which is written in C++) is Java code and run on a 2.5 GHz Intel Core i5 with a maximum heap size of 8 GB.

7.2 Results

First let us discuss how far from optimal both HPA* and THPA* are. HPA* is within 5.967% of the optimal path length found by A* (both unsmoothed). THPA* is within 4.378% of the optimal path length found by TA* (both smoothed). Although we are comparing smoothed to unsmoothed paths, we can see that THPA* finds similar paths in terms of near optimality.

Next we will analyze the correlation between running time and graph size over nonuniform maps. We found no correlation for A* and HPA*, weak positive correlation for THPA* and a strong ($R^2 = .997$) positive linear correlation for TA*. THPA* and TA* have similar speeds, HPA* is the next fastest and A* is the slowest algorithm. We will see in our uniform results that THPA* scales better in larger graphs than TA* for our trials.

The results for A* and HPA* may seem incorrect initially, but remember that the area of each map is constant (300 units), so run times for A* and HPA* should remain relatively constant, which explains the lack of correlation for these algorithms. There is a linear correlation for TA*, which seems correct given that the upper bound for TA* is $O(n^2)$, so the average case may be much smaller. Finally, the weak positive trend for THPA* is due to a high variation in the running times of each test. This may be caused by the randomness in the graphs providing ideal and less than ideal searches for the abstract graph A* search.

Figure 7.1: Graph of run time vs graph size for A*

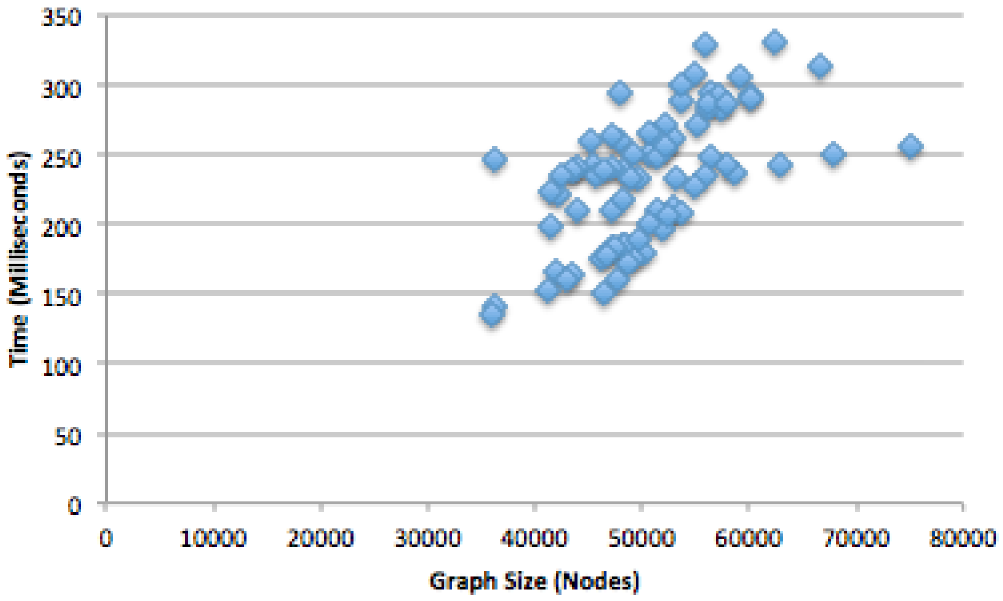


Figure 7.2: Graph of run time vs graph size for HPA*

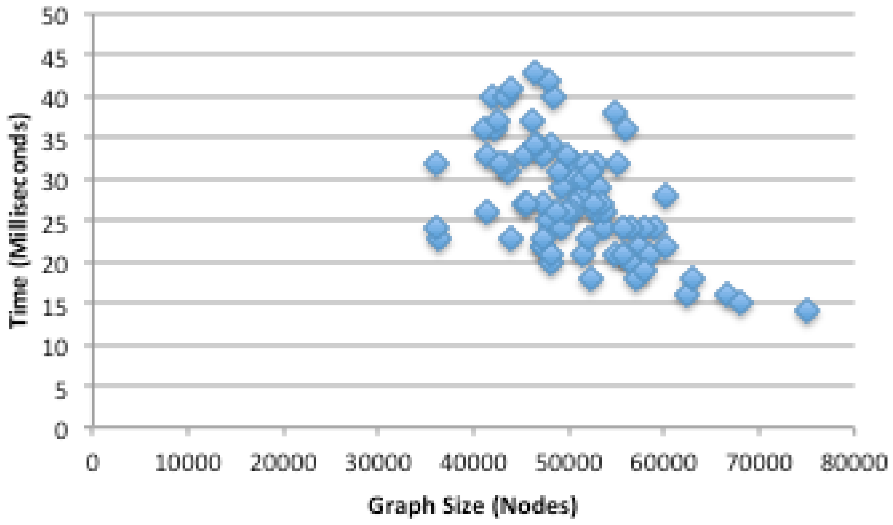


Figure 7.3: Graph of run time vs graph size for TA*

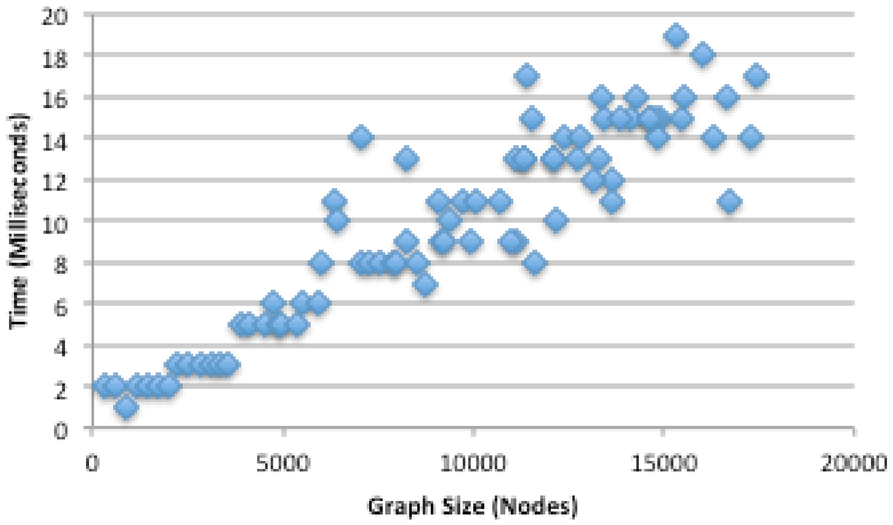
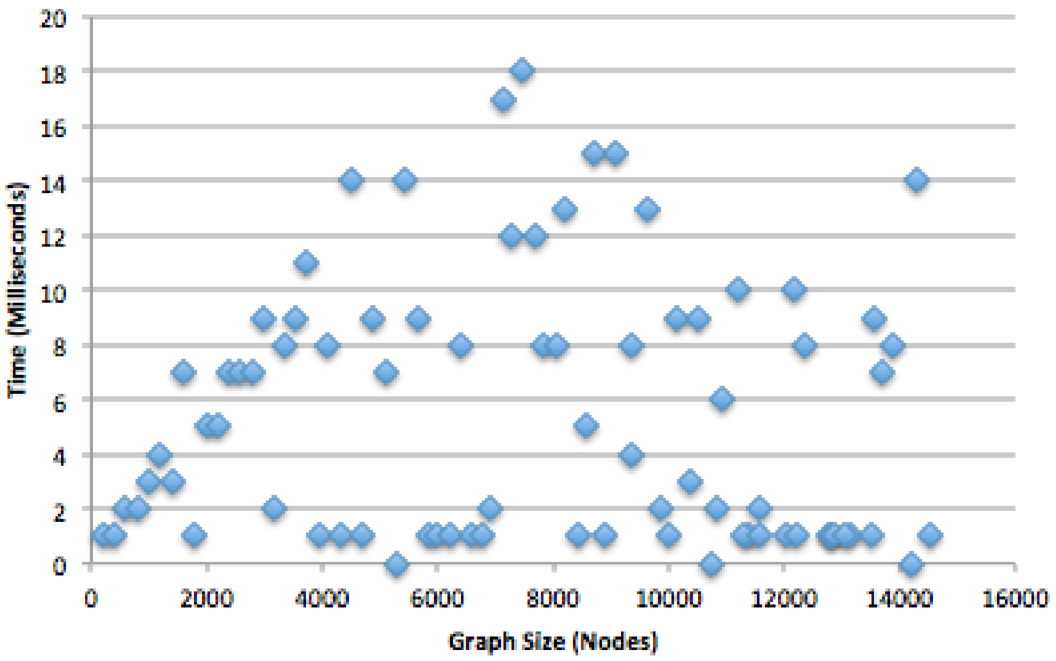


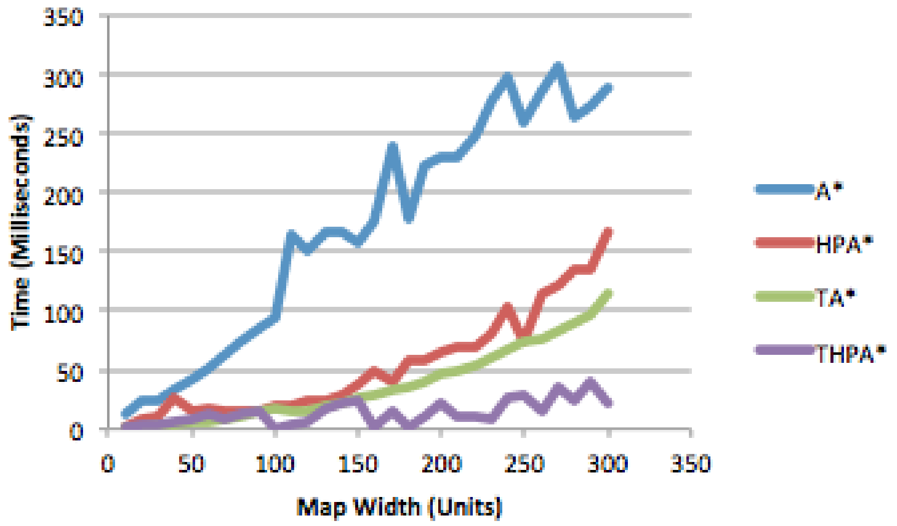
Figure 7.4: Graph of run time vs graph size for THPA*



Next, we will analyze the correlation between running time and graph size over uniform maps. The number of obstacles grows proportionally with the area of the graph, so we expect TA* and THPA* to perform relatively worse than in the nonuniform case. The data in 7.5 supports this claim. THPA* is clearly the fastest algorithm, never taking more than 40 ms. TA* and HPA* perform similarly, with a slight advantage to TA*. Compared to the nonuniform trials, the running time for the search in TA* is much closer to that of HPA* because of the less than optimal conditions for TA* (obstacle numbers growing proportionally to graph size). Finally, A* is clearly the slowest algorithm, growing quadratically with the map width (linearly with the number of obstacles however).

The data suggests that for densely packed maps, HPA* is the best algorithms for speed. However THPA* is still a viable choice given the utility it provides (dynamic search space modifications, non-zero radius agents, etc).

Figure 7.5: Graph of run time vs graph size for all algorithms over uniform maps



Next, we will analyze the correlation between inter-edges, graph size and running time to find the abstract path for THPA* over nonuniform maps. First note that the Galois METIS implementation attempts to minimize the number of inter edges. From 7.6 we can see that the number of inter-edges grows approximately linearly ($R^2 = .9037$) in relation to the size of the graph. The behavior for running time of THPA* in respect to the total inter-edges is not surprising given our THPA* results. The variability in the running times produces high variability in this graph.

Figure 7.6: Graph of total inter-edges vs graph size for THPA* over nonuniform maps

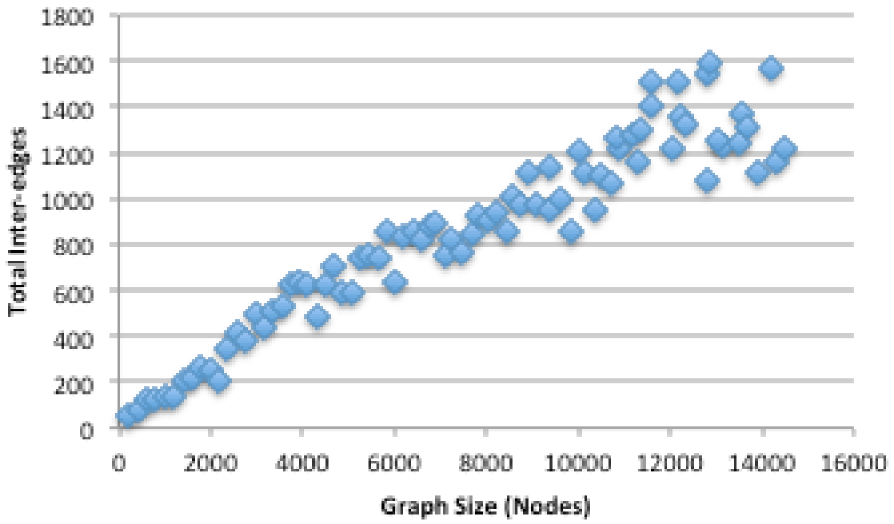
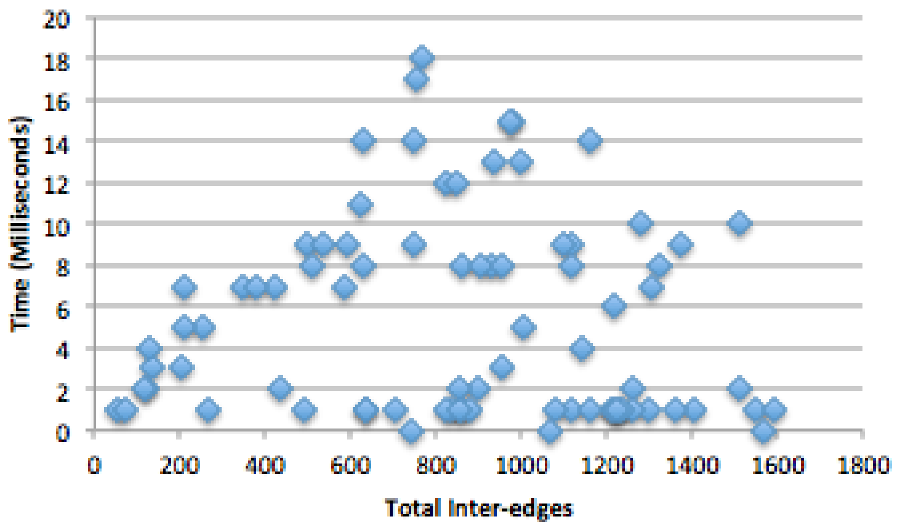


Figure 7.7: Graph of running time to find the abstract path vs total inter-edges for THPA* over nonuniform maps



Finally, we will analyze the correlation between intra-edges, graph size and running time to find the cache for THPA* over nonuniform maps. First note that the number of intra-edges should increase quadratically with the size of the graph, as shown in §4. The graph of intra-edges in 7.8 grows quadratically ($R^2 = .9642$) with graph size, which is what we expect. The running time for finding the cache in 7.9 is approximately quadratic ($R^2 = .9642$). This makes sense as both total intra-edges and running time to find the cache grow quadratically in respect to graph size.

Figure 7.8: Graph of total intra-edges vs size of the graph for THPA* over nonuniform maps

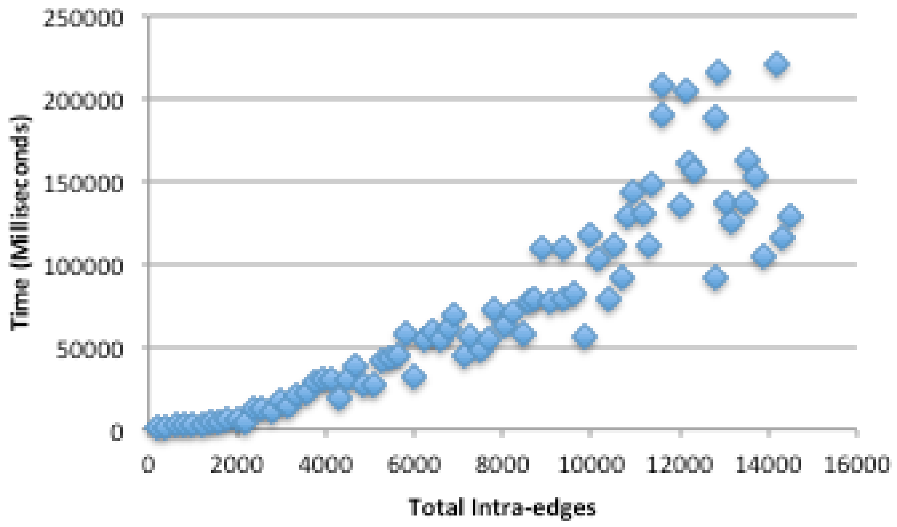


Figure 7.9: Graph of running time to find the cache vs graph size for THPA* over nonuniform maps

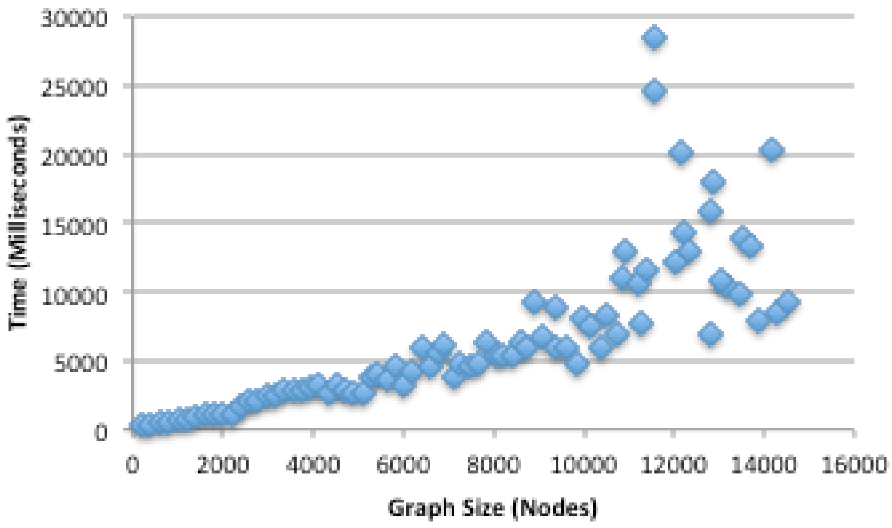
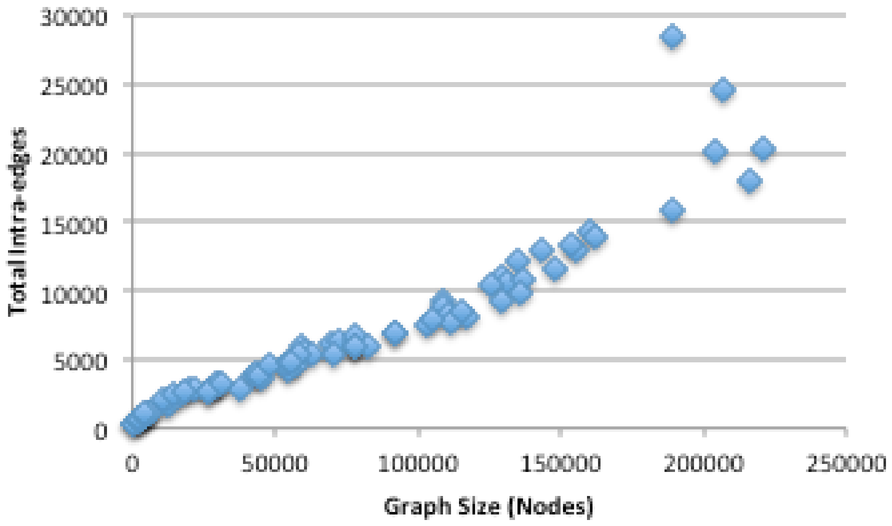


Figure 7.10: Graph of running time to find the cache vs total intra-edges for THPA* over nonuniform maps



Chapter 8

Conclusion

HPA* is a powerful algorithm, but does not have support for many features that we would like in game programming. We designed the THPA* algorithm, which combines the TA* and HPA* algorithms to provide speedups to TA* from the concepts provided by HPA*. This algorithm has support for search space modifications at run time, finding paths for non-zero radius agents, and handling any polygonal obstacles.

Our algorithm creates the triangulated search space, partitions this search space and creates an abstract graph with cached edge weights during pre-processing. In addition, we can guarantee that there are $O(\sqrt{m})$ edges in the abstract graph by using the algorithm described in [9] and setting the α parameter.

Our results show considerable speedups from TA* to THPA* and competitive running times for pathfinding with HPA* for maps where area of the map and number of obstacles are independent. Although pathfinding was fast, the running times to find the cache were rather slow. They were most definitely polynomial (approximately quadratic), but were the bottleneck for preprocessing.

Several improvements can be made to our algorithm. First we have no guarantees on the time complexity of vertex and edge updates for our partitioning phase after pre-processing. Also, optimizations can be made to recalculating agent paths and finding which agent paths must be recalculated after modifications to the search space. Additionally, a good choice in heuristic for THPA* has not been discussed and can be explored. Finally, techniques to reduce the number of computations while calculating the cache can be developed (similar to how HPA* ignores many nodes in each entrance edge).

Appendix

Algorithm 1 DistanceBetween(Vertex C, Edge e) : Distance

```
1:  $A, B \leftarrow$ EndpointsOf($e$)
2:  if $A_x = B_x$ then
3:      return $|A_x - C_x|$
4:  else
5:      $rise \leftarrow B_y - A_y$
6:      $run \leftarrow B_x - A_x$
7:      $intercept \leftarrow A_y - (\frac{rise}{run})A_x$
8:      $a \leftarrow rise$
9:      $b \leftarrow -run$
10:     $c \leftarrow run \times intercept$
11:     return $\frac{|a \cdot C_x + b \cdot C_y + c|}{\sqrt{a^2+b^2}}$
12: end if
```

Algorithm 2 SearchWidth(Vertex C, Triangle T, Edge e, Distance d) : Distance

```
1:  $U, V \leftarrow$ EndpointsOf($e$)
2:  if IsObtuse($C$, $U$, $V$) $\vee$ IsObtuse($C$, $V$, $U$) then
3:      return $d$
4:  end if
5:  $d' \leftarrow$ DistanceBetween($C$, $e$)
6:  if $d' > d$ then
7:      return $d$
8:  else if IsConstrained($e$) then
9:      return $d'$
10: else
11:     $T' \leftarrow$ TriangleOpposite($T$, $e$)
12:     $e', e'' \leftarrow$ OtherEdges($T'$, $e$)
13:     $d \leftarrow$ SearchWidth($C$, $T'$, $e'$, $d$)
14:     return SearchWidth($C$, $T'$, $e''$, $d$)
15: end if
```

Algorithm 3 CalculateWidth(Triangle T, Edge a, Edge b) : Distance

```
1: $C \leftarrow$ VertexBetween($a$, $b$)
2: $c \leftarrow$ EdgeOpposite($C$, $T$)
3: $A \leftarrow$ VertexOpposite($a$, $T$)
4: $B \leftarrow$ VertexOpposite($b$, $T$)
5: $d \leftarrow$ min(Length($a$),Length($b$))
6: if IsObtuse($C$, $A$, $B$) $\vee$ IsObtuse($C$, $B$, $A$) then
7:     return $d$
8: else if IsConstrained($c$) then
9:     return DistanceBetween($C$, $c$)
10: else
11:     return SearchWidth($C$, $T$, $c$, $d$)
12: end if
```

Algorithm 4 CacheValuesForEdge(Dictionary *cache*, Vertex v_1, Vertex v_2) : Dictionary

```
1: $d_{min} \leftarrow$ MinimumAgentDiameterForPath($v_1$, $v_2$, 0) //0 is the radius
2: if $d_{min} = 0$ then
3:     $d_{min} = k$ //k is some predefined constant
4: end if
5: while $d_{min} \neq \infty$ do
6:     $d \leftarrow$ MinimumAgentDiameterForPath($v_1$, $v_2$, $d_{min}$)
7:     $c \leftarrow$ CostForPath($v_1$, $v_2$, $d_{min}$)
8:     if $d_{min} = d$ then
9:         $d = d + k$
10:    end if
11:    $cache[d] = c$
12:    $d_{min} = d$
13: end while
14: return $cache$
```

Algorithm 5 Funnel(Channel c, Point s, Point g) : Path

```
p.Clear()
if NumEdges(c) ¡ 1 then
    p.Add(s); p.Add(g)
    return p
end if
AddVertex($s$, $p$)
$f \leftarrow$ FunnelDeque($s$)
$v_l \leftarrow$ LeftEndPoint($c_0$); Add($f$, $v_l$, Left,$p$)
$v_r \leftarrow$ RightEndPoint($c_0$); Add($f$, $v_r$, Right,$p$)
for $i \leftarrow 1$ to NumEdges($c$) do
    $v'_l \leftarrow$ LeftEndpoint($c_i$); $v'_r \leftarrow$ RightEndpoint($c_i$)
    if $v'_l = v_l$ then
        $v_r \leftarrow v'_r$; Add($f$, $v_r$, Right, $p$)
    else
        $v_l \leftarrow v'_l$; Add($f$, $v_l$, Left, $p$)
    end if
end for
Add($f$, $g$, Point, $p$)
return $p$
```

Algorithm 6 Add(FunnelDeque f, Vertex v, Type t, Path p)

```
if $t$=Left then
  loop
    if $f_{Left} = f_{Right}$ then
      $f$.AddLeft($v$);
      break
    else if $f_{Left} = f_{Apex}$ then
      $\theta \leftarrow$ Angle($f_{Left}$, $f_{Left+1}$)
      $\phi \leftarrow$ Angle($f_{Left}$, $v$)
    else
      $\theta \leftarrow$ Angle($f_{Left+1}$, $f_{Left}$)
      $\phi \leftarrow$ Angle($f_{Left}$, $v$)
    end if
    if CounterclockwiseTo($\theta$, $\phi$) then
      $f$.AddLeft($v$)
      break
    end if
    if $f_{Left} = f_{Apex}$ then
      $\psi \leftarrow$ Angle($f_{Apex}$, $f_{ApexType}$, $f_{Apex+1}$, Right)
      AddVertex($f_{Apex}$, $p$)
      $f$.PopApexLeft()
    end if
  end loop
else if $t$ = Right then
  {same procedure with directions reversed}
else if $t$ = Point then
  $i \leftarrow 0$
  while $f_{Apex+i} \neq f_{Right}$ do
    $i \leftarrow i + 1$
    AddVertex($f_{Apex+i}$, $p$)
  end while
end if
```

Algorithm 7 Funnel(Channel c, Radius r, Point s, Point g) : Path

```
1:  $p$.Clear()
2:  if NumEdges(c) ¡ 1 then
3:      $p$.Add(s); $p$.Add(g)
4:      return $p$
5:  end if
6:  $f \leftarrow$ FunnelDeque($s$, $r$)
7:  $v_l \leftarrow$ LeftEndPoint($c_0$); Add($f$, $v_l$, Left,$p$)
8:  $v_r \leftarrow$ RightEndPoint($c_0$); Add($f$, $v_r$, Right,$p$)
9:  for $i \leftarrow 1$ to NumEdges($c$) do
10:     $v'_l \leftarrow$ LeftEndpoint($c_i$); $v'_r \leftarrow$ RightEndpoint($c_i$)
11:     if $v'_l = v_l$ then
12:         $v_r \leftarrow v'_r$; Add($f$, $v_r$, Right, $p$)
13:     else
14:         $v_l \leftarrow v'_l$; Add($f$, $v_l$, Left, $p$)
15:     end if
16: end for
17: Add($f$, $g$, Point, $p$)
18: return $p$
```

Algorithm 8 Add(FunnelDeque f, Vertex v, Type t, Path p)

```
if $t$=Left then
  loop
    if $f_{Left} = f_{Right}$ then
      $f$.AddLeft($v$);
      break
    else if $f_{Left} = f_{Apex}$ then
      $\theta \leftarrow$ Angle($f_{Left}$, $f_{ApexType}$, $f_{Left+1}$, Right); $l_2 \leftarrow$ Distance($f_{Left}$, $v$)
    else if $f_{Left+1} = f_{Apex}$ then
      $\theta \leftarrow$ Angle($f_{Left+1}$, $f_{ApexType}$, $f_{Left}$, Left); $l_2 \leftarrow$ Distance($f_{Left+1}$, $v$)
    else
      $\theta \leftarrow$ Angle($f_{Left+1}$, Left, $f_{Left}$, Left); $l_2 \leftarrow$ Distance($f_{Left+1}$, $v$)
    end if
    if $f_{Left} = f_{Apex}$ then
      $\phi \leftarrow$ Angle($f_{Left}$, $f_{ApexType}$, $v$, $t$)
    else
      $\phi \leftarrow$ Angle($f_{Left}$, Left, $v$, $t$)
    end if
    if CounterclockwiseTo($\theta$, $\phi$) then
      $f$.AddLeft($v$)
      break
    end if
    if $f_{Left} = f_{Apex} \wedge l_2 < l_1$ then
      $\psi \leftarrow$ Angle($f_{Apex}$, $f_{ApexType}$, $v$, $t$)
      AddVertex($f_{Apex}$, $f_{ApexType}$, $v$, $t$)

      AddVertex($v$, $t$, $\psi$, $p$)
    else if $f_{Left} = f_{Apex}$ then
      $\psi \leftarrow$ Angle($f_{Apex}$, $f_{ApexType}$, $f_{Apex+1}$, $Right$)
      AddVertex($f_{Apex}$, $f_{ApexType}$, $\psi$, $p$)
      AddVertex($f_{Apex+1}$, Right, $\psi$, $p$)
      $f$.PopApexLeft()
    end if$f$.PopLeft()
  end loop
else if $t$ = Right then
  {same procedure with directions reversed}
else if $t$ = Point then
  $i \leftarrow 0$
  $t_1 \leftarrow f_{ApexType}$; $t_2 \leftarrow$ Right
  while $f_{Apex+i} \neq f_{Right}$ do
    if $f_{Apex+i+1} = f_{Right}$ then
      $t_2 \leftarrow$ Point
    end if
    $\beta \leftarrow$ Angle($f_{Apex+i}$, $t_1$, $f_{Apex+i+1}$, $t_2$)
    AddVertex($f_{Apex+i}$, $t_1$, $\beta$, $p$)
    AddVertex($f_{Apex+i}$, $t_2$, $\beta$, $p$)
    $t_1 \leftarrow$ Right
    $i \leftarrow i + 1$
  end while
end if
```

Bibliography

[1] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical pathfinding. *Journal of game development*, 1(1):7–28, 2004.

[2] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *AAAI*, volume 6, pages 942–947, 2006.

[3] Gary William Flake, Robert E Tarjan, and Kostas Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4):385–408, 2004.

[4] Marcelo Kallmann, Hanspeter Bieri, and Daniel Thalmann. Fully dynamic constrained delaunay triangulations. In *Geometric modeling for scientific visualization*, pages 241–257. Springer, 2004.

[5] George Karypis. Metis. http://glaros.dtc.umn.edu/gkhome/views/metis, 2013-2014.

[6] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *ICPP (3)*, pages 113–122, 1995.

[7] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[8] Mikko Mononen. Recast and detour. https://github.com/memononen/recastnavigation, 2009-2014.

[9] Barna Saha and Pabitra Mitra. Fast incremental minimum-cut based algorithm for graph clustering. In *Sixth IEEE International Conference on Data Mining Workshops ICDMW06*, pages 207–211, 2006.

[10] Xin Sui, Donald Nguyen, Martin Burtscher, and Keshav Pingali. Parallel graph partitioning on multicore architectures. In *Languages and Compilers for Parallel Computing*, pages 246–260. Springer, 2011.