Efficient Triangulation-Based Pathfinding

Douglas Demyen and Michael Buro
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{demyen|mburo}@cs.ualberta.ca

Abstract

In this paper we present a method for abstracting an environment represented using constrained Delaunay triangulations in a way that significantly reduces pathfinding search effort, as well as better representing the basic structure of the environment. The techniques shown here are ideal for objects of varying sizes and environments that are not axis-aligned or that contain many dead-ends, long corridors, or jagged walls that complicate other search techniques. In fact, the abstraction simplifies pathfinding to deciding to which side of each obstacle to go. This technique is suited to real-time computation both because of its speed and because it lends itself to an anytime algorithm, allowing it to work when varying amounts of resources are assigned to pathfinding. We test search algorithms running on both the base triangulation (Triangulation A* – TA*) and our abstraction (Triangulation Reduction A* – TRA*) against A* and PRA* on grid-based maps from the commercial games Baldur's Gate and WarCraft III. We find that in these cases almost all paths are found much faster using TA*, and more so using TRA*.

Introduction

Pathfinding continues to be a critical area in many fields, not least of which are robotics and games. For the former, it is important to have a technique that incorporates the size of the robot so that a path can be found which will not result in damage to the equipment. In the latter, it is of paramount importance that paths be found very quickly, as there is seldom much time alloted to pathfinding, and that the paths found be close to optimal, in order to give the illusion of intelligent movement. Our technique addresses both concerns, finding the majority of paths tested in less than 1 ms.

Different methods of abstracting search space have so far been successful in speeding up search, sometimes with a minor reduction in solution quality. Hierarchical A* (Holte *et al.* 1996) searches layers of increasingly abstracted representations of the search space to produce heuristics for the layers below. HPA* (Botea, Müller, & Schaeffer 2004) divides an environment into sectors, and caches the path lengths between entry points of these sectors to quickly produce a nearly-optimal path across the whole environment. Recently, PRA* (Sturtevant & Buro 2005) forms cliques of adjacent nodes at each level to form a more abstract level until the whole environment is a single node. Search is performed on a suitably abstract level and projected down and refined on lower levels back to the original graph.

Similarly to how HPA* and PRA* attempt to separate low-level from high-level decision-making when pathfinding the same way a human won't decide their exact steps before walking to the store, TRA* attempts to reduce the decisions that need to be made. This is equivalent to how a human might decide on which side of a shrub to walk when crossing a yard, and taking for granted the best path around that side of the shrub.

Polygonal representations of an environment offer several advantages over grid representations in pathfinding including exact representation of straight barriers at arbitrary angles, retention of all valid paths, and fewer cells in larger, open areas. A successful method of polygonal representation is triangulation which has many properties useful for pathfinding, and thanks to recent work (Kallmann, Bieri, & Thalmann 2003) can be updated dynamically as well. Triangulations are also interesting in that they lend themselves well to abstraction and determination of the size of objects that can move through the various parts of the environment.

This paper will first explore considerations for determining the largest objects that may move through a triangle, and how to find the shortest path for such an object through a sequence of triangles. We will then describe the structure of the abstraction used, an algorithm for its construction, and the information it contains. Next we will look at the search process in both the base triangulation and our abstracted version, and then compare them experimentally also with other state-of-the-art pathfinding algorithms. Finally, we summarize the contribution of this paper, draw some conclusions, and suggest future research directions.

Triangulation-Based Pathfinding

Given an environment represented with a polygonal description — one that has barriers between traversable terrain and obstacles described as line segments (see Fig. 1a) — we wish to create a triangulation. This is done by inserting edges between these line segments' endpoints until all spaces are divided into triangles (Fig. 1b).

This is in fact a constrained triangulation, since the barriers are required to be edges in the triangulation. These are called *constrained* edges, and the ones added to complete the triangulation are called *unconstrained* edges. In terms of pathfinding, we can cross unconstrained edges but not constrained ones. Making a constrained triangulation *Delaunay* specifies that the unconstrained edges be such that the minimum angle of the triangles is maximized. This guarantees the optimal path will not cross any triangle more than once,

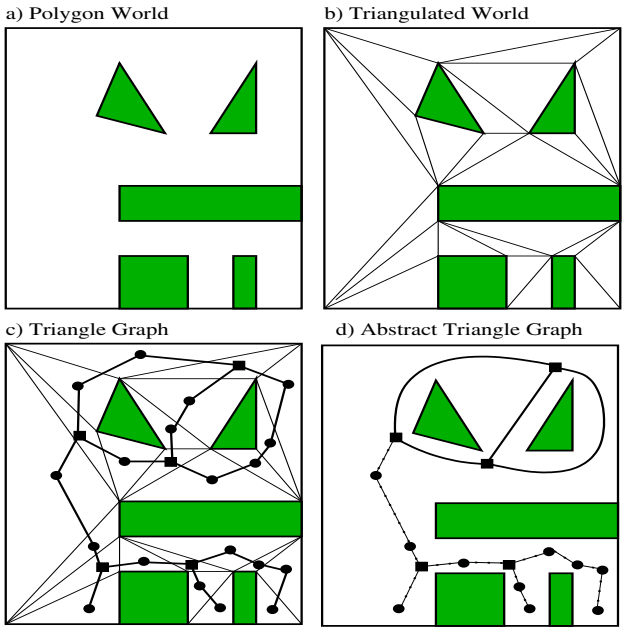

Figure 1: How a reduced triangulation graph is constructed from a polygonal world description

by avoiding thin triangles where to get between two edges, an object would have to exit and re-enter through the other.

In its simplest form, pathfinding in a constrained triangulation is done by hopping from triangle to triangle. This presents a challenge because we do not know the path the object will take through each triangle. In (Kallmann 2005), states are considered the midpoint of a triangle edge, their children the midpoints across the triangle, and the object's path is made up of straight-line segments between these. In addition, each visited triangle is marked so it cannot be visited again. This method can result in suboptimal paths, but in practice performs well. Since we are concerned with producing optimal results where we do not have to pay much of a penalty, we take a different approach.

Triangulation A*, or TA*, is our triangulation pathfinding algorithm. Similar to above, we consider the states in the search to be triangles and their children to be triangles adjacent across unconstrained edges. However in order to avoid pruning the optimal solution, we made some changes.

Because we do not know the actual distance through the sequence of triangles until reaching the goal, we must consider all paths to a triangle in order to obtain the optimal solution. When considering a triangle in either TA* or TRA* (Triangulation Reduction A*, described later), we estimate the cost incurred (the g-value) and the heuristic (the h-value) from *any* point on the entry edge of the triangle. The heuristic is calculated as the Euclidean distance between the goal and the closest point to it on this edge. We know this heuristic to be both admissible and consistent, which is helpful since the incurred cost must also be estimated.

The estimation of the g-value as it turns out is critical in the efficiency of the search. The higher this value is, the fewer nodes are searched, much like with the h-value. However, the g-value — and by extension the sum of this and the h-value — must be no greater than their true values if the anytime algorithm is to converge on the optimal solution. This algorithm works as follows: we search as usual until we find a path to the goal, we run the modified funnel algorithm described later to determine the actual cost of this path, and we continue searching, calculating lengths of paths found and updating our best path each time we find one shorter, until our best path is no greater than the sum of the g- and h-values of the remaining search nodes, at which point we know we have the shortest possible path.

In order to achieve the highest possible underestimate for the g-value, we take the maximum of a number of known admissible values: the Euclidean distance between the start and the closest point to it on the entry edge of the current triangle; the distance between the start and the goal minus this node's h-value; the parent node's g-value plus the difference between its h-value and that of the current node; and the parent node's g-value plus the shortest path between its entry edge and that of the current triangle. All these combine to create a fairly accurate g-value without TRA* having to query individual triangles.

It turned out that a large portion of the time in the above method was spent finding the triangle containing the starting point. This is because the point location used in (Kallmann 2005) simply "walked" from a fixed starting triangle progressively closer to the point. To help speed this up, we implemented a sector-based method wherein we overlaid a grid of points on the triangulation and determined in which triangle each was contained. This allowed the point location to begin "walking" from the triangle around the closest grid point, resulting in much better performance. For the experiments reported later, we used a modest grid size of 10×10.

Non-Point Objects

It is often the case in games that moving objects have some size that must be considered while pathfinding. This is usually handled by means of "growing" the obstacles in the environment corresponding to the size of the objects and pathfinding within this environment as with a point object. While this approach has been shown to work, it has a drawback in that this environment must be calculated for each size (and shape) of object. This may not be singly disadvantageous if there are few such object footprints, however if there are many or we desire to find a path for an object of arbitrary size, the time for such a calculation and the can become unwieldy.

If the environment is being represented as a grid, more problems can arise due to the fact that it is often unnatural to represent objects by way of grid squares. For example, a circular object is often represented by a square or a "rasterized" circle, which are imprecise. If, on the other hand, the environment is triangulated, objects that are circular (or otherwise radially symmetrical) can be added around the obstacles by use of the Minkowski Sum (by approximation to a regular polygon), however this leads to a large number of thin, sliver-like triangles which complicate pathfinding and have other undesirable effects on triangulations. In both

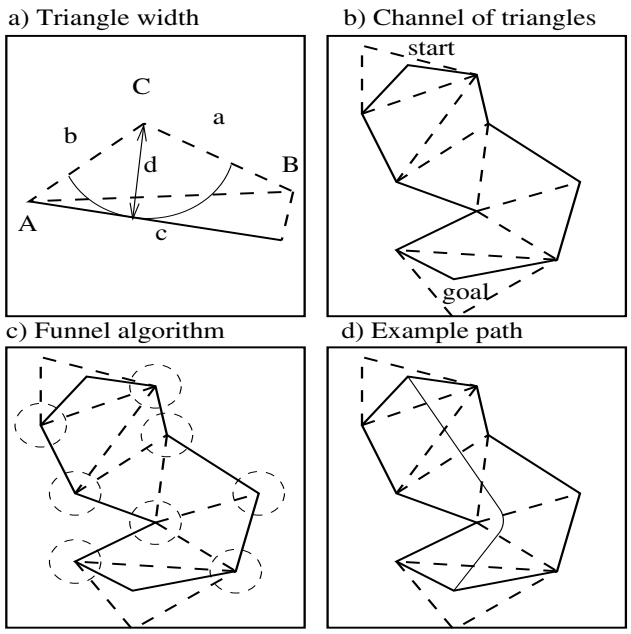

Figure 2: Object radius considerations

cases, object shapes that aren't roughly radially symmetrical are seldom used in practice because of the complexity it adds to the problem.

Width Calculation

Triangulations offer a unique opportunity to accommodate differently sized and shaped objects. At the time of writing, the focus was on circular objects because of their prominence in games. However, we believe with some calculation, other shapes could be managed as well. When considering a path for an object through a triangle, one must simply check that the "width" through the triangle between the entry and exit edges be large enough to accommodate that object (for a circular object, it must be twice the object's radius). This width, when traveling between (unconstrained) edges a and b of a triangle (see Fig. 2a), can be calculated as the closest obstacle to vertex C in the region between the rays Ca and Cb, where an obstacle is either a vertex in the triangulation, or a point on a constrained edge.

One can check that a width of $2r$ through a triangle between edges a and b is necessary and sufficient for a path to exist for an object of radius r between those edges in that triangle. This eliminates the need for a separate representation of the environment for each size of object. While in the worst case determining the width of one triangle is linear in the number of triangles in the environment, the properties of a triangulation make it likely that calculating the widths of *all* triangles in the environment will be similarly linear.

Modified Funnel Algorithm

A search through a triangulation does not yield a particular path, but a series of adjacent triangles with the start position in the first triangle and the goal position in the last. From this, we construct a *channel* (Kallmann 2005), a simple polygon with the start and goal positions as vertices and which traces the perimeter of the triangles in between (Fig. 2b).

Once we have this channel, we wish to find the shortest path within it to use for object motion. For a point object, this can be found in time linear in the number of triangles in the channel with what is called the *funnel algorithm* (Hershberger & Snoeyink 1994). Obviously, since we are dealing with larger objects, we desire an algorithm which can find the shortest path within a channel, while keeping at least distance r from the vertices of the channel (with the exception of the start and goal vertices). It turns out that with some small modifications to the funnel algorithm, we can find such a path also in linear time. We assume that such a path is possible; if a channel cannot yield a valid path for an object of radius r, it would not be considered by the search.

The original funnel algorithm is described in (Hershberger & Snoeyink 1994), which we will not cover for lack of space. The simple modification to accommodate circular objects with radius r is conceptually adding circles of radius r around each interior vertex, as in Fig. 2c. When considering the angle between two vertices, a segment tangent to the circles around the vertices is used instead of a segment connecting the vertices.

The result of this algorithm is a path consisting of arcs around vertices and line segments tangent to these arcs, between them, as shown in Fig. 2d. This produces the optimal path within the channel for an object of the given radius. However, this path requires that the object be capable of curved motion, so when this is not available, it can be approximated by straight segments.

Triangulation Graph Reductions

Each triangle in the triangulation is mapped to a single "node" in the abstract graph. These nodes are categorized by degree between 0 and 3, inclusive, which refer to the number of adjacent graph structures. Specifically, a triangle is mapped to a degree-n node when $3 - n$ of its edges are such that: either the edge is constrained or the triangle across that edge is mapped to a degree-1 node.

We will cover a brief example based on the graph in Fig. 3 to illustrate how these nodes are categorized. First we go through the triangles in the graph to determine which of them have one or fewer unconstrained edges. In this example, the two triangles at the top of the "Y" structure have one unconstrained edge each since they have only one adjacent triangle. These are mapped to degree-1 nodes. If any were encountered with no unconstrained edges, these would be mapped to degree-0 nodes.

Next, we put the triangle to which these are adjacent on to a queue for processing. We go through this queue and find that the triangle next to the top left one now has only one adjacent triangle not mapped to a degree-1 node, so this triangle is mapped to a degree-1 node and the triangle at the fork is added to the queue. This process continues until the queue is empty. We will notice that this happens when the triangle at the base of the "Y" is reached: it has two adjacent triangles not mapped to degree-1 nodes. If a connected component of the triangulation is acyclic (i.e. the triangle graph

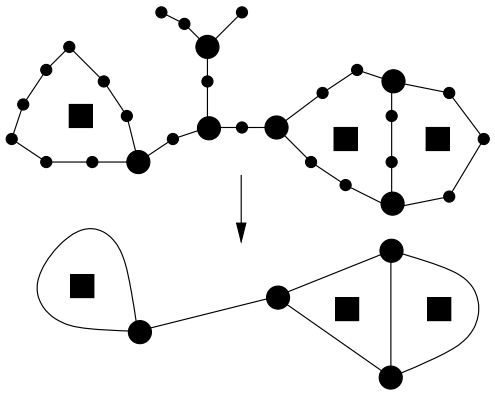

Figure 3: Typical triangulation graph and its reduced form.

forms a tree), this component's triangles will all be mapped to degree-1 nodes, since they will "collapse" from all sides.

At this point we go through the other triangles in this graph and find unmapped triangles with no constrained edges or adjacent triangles mapped to degree-1 nodes. These are mapped to degree-3 nodes, and then we cross each edge and visit each adjacent unmapped triangle in turn, mapping them to degree-2 nodes, until no such node is adjacent or the triangle qualifies to be mapped to a degree-3 node.

In this way, the edges of the graph at the bottom of Fig. 3 would be formed, the triangles on them being mapped to degree-2 nodes. The degree-3 nodes form the vertices of this graph. Any remaining unmapped triangles map to degree-2 nodes, since they would form one or more "rings".

Degree-n nodes are connected to n adjacent graph structures. Since all degree-0 nodes are based on triangles with all constrained edges, they aren't adjacent to anything. Degree-1 nodes are adjacent to a degree-2 node, which we call the "root". For example the degree-1 nodes in the "Y" in Fig. 3 are adjacent to the degree-2 node at the bottom of it. The exception is when the triangulation graph is acyclic, as described above, when the degree-1 nodes in this unrooted tree aren't adjacent to anything.

Degree-2 nodes are adjacent to degree-3 nodes, as shown in Fig. 3 where degree-2 nodes on an edge are adjacent to the degree-3 nodes that form the endpoints of that edge. Again, if the degree-2 nodes form a "ring", this is an exception and they aren't adjacent to anything. Finally, degree-3 nodes are adjacent to other degree-3 nodes either directly adjacent or across chains of degree-2 nodes, of which there must be 3.

This algorithm presented is linear in the number of triangles in the triangulation. As well, in the case that the environment changes, and the triangulation is repaired locally as in (Kallmann, Bieri, & Thalmann 2003), the abstract graph can be repaired locally using in the best case the triangles that were modified, and in the worst case, the connected component that was modified.

The nodes in the abstract graph, in addition to containing the adjacent graph structures, also contain spatial information to be used in searching the graph. Each node contains information on the adjacent node (if any) in each direction, a lower bound on the distance to that triangle to help estimate g-values, and the narrowest point between the triangles. We also record the width between pairs of unconstrained edges through the triangle.

Both the triangulation itself, plus all the abstraction information can be stored in an average of less than 183 bytes per triangle, so that even the largest maps in our experiments could be stored in around 3MB, the majority of those tested in under 1MB, and if the environments were designed for triangulation, considerably less. This is an important consideration, especially for games.

Further Reductions

There are extensions that are possible for the graph and abstraction layers. Further abstraction is achievable if one collapses doubly-connected components of the abstract graph into single nodes of a more abstract graph. Where on the abstract graph the nodes represent decision points for which way to go around an obstacle, in this new graph, they would represent "rooms" in the environment which contain multiple paths between their entry and exit points.

This graph would then be a tree of doubly-connected components. Because it is a tree, the highest level of the pathfinding problem would become trivial as there is only one path between any two points in a tree. The best paths between each pair of entry points of each doubly-connected component could even be cached, after which the search function would only have to get from the start and goal points on to this new abstracted graph.

Abstracted Triangulation Searches

We here introduce TRA*, which searches the abstraction described above. While these algorithms both consider triangles as states, TA* generates children as the triangles adjacent across unconstrained edges, and TRA* generates degree-3 nodes adjacent across degree-2 corridors. This greatly benefits search because the number of degree-3 nodes is only dependent on the number of obstacles (there are $2n - 2$ for n obstacles) and not their features.

TRA* also has many cases for which no real search must take place. For example, if both the start and the goal are in an unrooted degree-1 tree or if they are both in a tree rooted at the same degree-2 node, we can find the path using a simple search. This search can consider just the midpoints of the triangles and only expand each node once, since we know that in a tree, there is only one path between any two points. Therefore, once we have a channel between the start and goal points, it must contain the shortest path between them and we can stop. If either the start or goal is in the root triangle of a tree containing the other, we can simply walk from the tree along the single path to the root.

If both are on the same degree-2 edge or in trees rooted in the same edge, we form one path by walking between them on that edge, and then consider the start and goal to be the degree-3 endpoints of that edge and continue with the degree-3 search to check for any shorter paths. If, however, those endpoints are the same degree-3 node, we form another path across that node to compare with the other path and take the shortest. Similarly if both are on a degree-2

ring, we form a paths going each way around the ring and take the shortest. Only if none of these cases hold do we need to perform a search of the degree-3 nodes.

The search of the degree-3 nodes is performed as follows. If the goal is on a triangle mapped to a degree-1 or 2 node, the goals of the search are considered to be the degree-3 endpoints of the edge on which the goal lies or the tree containing the goal is rooted, otherwise it is simply the node mapped to by the triangle containing the goal. Likewise, the queue for the search algorithm is initialized with either the node mapped by the triangle on which the start lies if that is degree-3, or those adjacent to it if not. Search then proceeds as before moving between the degree-3 nodes of the abstract graph, with the same considerations as TA*.

Experiments

While one could argue that our technique is not comparable to grid-based pathfinding techniques, this type of environment offers both a plethora of maps used in commercial games and fast algorithms with which to compare ours. Hence we took this opportunity to see if TA* and TRA* could beat these other techniques "at their own game".

For comparison sake, we used the exact same maps, and start and goal points as in the PRA* experiment presented in (Sturtevant & Buro 2005). The data was kindly made available by the authors. In the interest of simplicity, we also used the results from that experiment, with the exception of halving the execution times to adjust for the different CPU speeds. The set of maps consisted of 75 Baldur's Gate maps — a grid of tiles marked traversable or untraversable — and 41 WarCraft III maps — a grid of different types of terrain and heights in which paths cannot cross height differences without ramps, or boundaries between different types of terrain. All maps were scaled to 512×512, while maintaining their connectivity. Each map contained 1280 paths with length between 0 and 511, 10 in each of 128 "buckets" (a path with length l belonging to bucket i exactly when $i = \lfloor l/4 \rfloor$). Both TA* and TRA* were run on the total 148480 paths over 116 maps on a computer with an AMD Athlon64 3200+ processor and 1GB of RAM using Microsoft Visual Studio .NET 2003. As with A* and PRA*, neither TA* or TRA* have been highly optimized for speed.

Both TA* and TRA* performed admirably in almost all cases. Despite not benefiting from the abstraction information, TA* still found most paths faster than the grid-based methods (see Fig. 4 and note the scale on the time axes). This is a testament to the usefulness of triangulations for reducing the number of nodes in the environment. One can also note that because the environments were grid-based, non-axis-aligned barriers were approximated by jagged edges, and there were more triangles in the environment than if it was built using off-axis segments.

Of course, TRA* does not suffer from this effect, its size only depending on the number of obstacles, not their shape. As a result, TRA* performed better overall, as seen in Fig. 4, demonstrating the effectiveness of the abstraction on the triangulation, reducing the effective number of nodes in the environment even further. One advantage TA* had over TRA* is that since TRA* skipped between degree-3 nodes

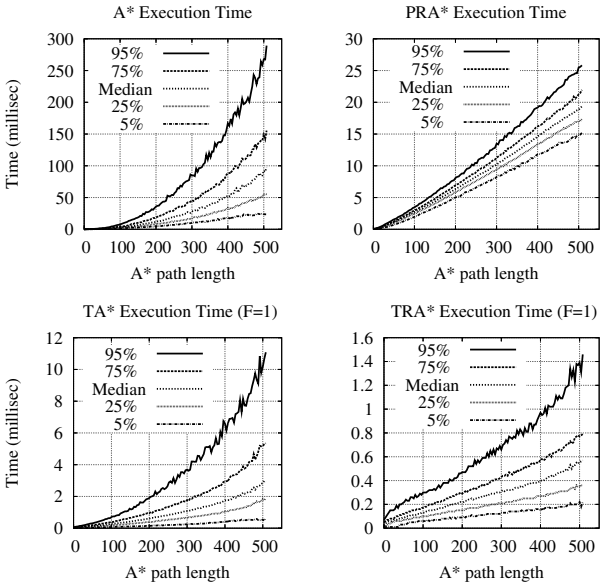

Figure 4: Time percentiles for A*, PRA*, TA*, and TRA*

in search, it was not able to calculate as accurate g-values as TA*. As a result, TRA* was sometimes led astray more and also could not prove its solution to be optimal as quickly.

Due to the aforementioned estimations and the need to consider all paths to each node, one thing that posed a problem to the triangulation searches was the presence of many, especially small obstacles — this was most apparent in the WarCraft III maps, which were often sprinkled with trees. Besides creating more triangles for TA* to search and degree-3 nodes for TRA* to search, this forced them to consider a large number of paths to each node. In the case of small obstacles, these paths differed very little in their g-values, keeping the searches from pruning them.

Because of this, a small fraction of searches would take several times longer than the others. Since these would not be acceptable in a real-time setting, we modified TA* and TRA* slightly to not expand a triangle twice until the first path was found. This allowed for the first paths to be found much faster, without affecting subsequent paths.

The path quality of the anytime algorithms are shown in Fig. 6. The F values refer to the multiple of the time to find the initial path at which the statistics were taken. The graphs show the 75^{th} and 95^{th} percentiles for the path lengths for the algorithms, divided by the length of the path found by TA* with $F = 10$, or TA*(10). This was chosen because in most cases, TA*(10) was optimal, but to address the times when it was not, the "bound" line was added to these graphs. Since we know that the A* paths were optimal when constrained to a grid, this line represents the minimum length a path *could* have without this constraint. The bound for the n^{th} percentile is calculated as the $(100 - n)^{th}$ percentile of the A* length, divided by a constant $C \approx 1.0824$ which represents how much longer a grid-based path could possibly be over its arbitrary motion equivalent.

We see when looking at the bound that TA*(10) is defi-

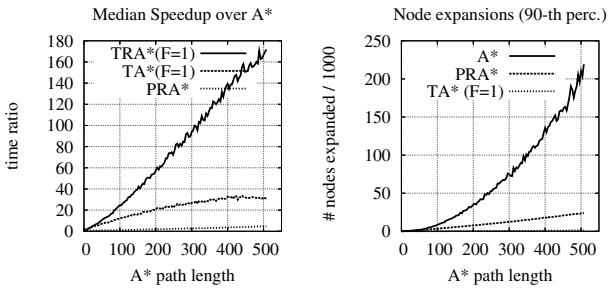

Figure 5: Median speedup and 90^{th} node expansion perc.

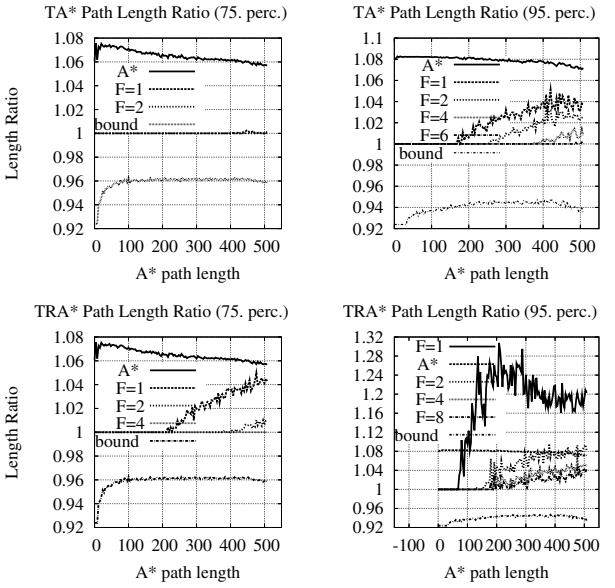

Figure 6: Path length ratio percentiles for TA* and TRA*

nitely within 4% of the optimal path length 75% of the time, and within 6%, 95% of the time. As these graphs show, TA* finds its shortest path most of the time right away, although less so on longer paths. As seen in the top-left graph, in 75% of paths, the first solution found rarely differs from the last, only when looking at the 95^{th} percentile graph do we see that a need for more time to achieve this result.

The bottom graphs show that TRA* requires higher F values to avoid this departure from the final solution. This is because TRA*'s slightly less precise g-values sometimes cause it to take several times longer to find its final solution than its first. However, because TRA* finds its first solution so much faster than TA*, it can often reach its final solution before TA* finds its first. We also see that even when the path degenerates from TA*(10), the path length is usually still less than the optimal path when constrained to the grid.

The fact that the time to find the optimal solution, or one near it, increases with the A* path distance of the path can be used to our advantage. The length of the first path found can be used to determine for how much longer the algorithm should be run in order to be likely to find a path within a certain amount of optimal.

We see when comparing the TA* and TRA* performance in the left graph of Fig. 5 that just the benefit of the triangulated representation of the environment makes TA* many times faster to find a first path than both grid-based methods, and the added benefit of the abstraction makes TRA*'s first solution even faster. We also see how many node expansions are saved using these methods — in the right graph, TA*'s node expansions are barely visible, with TRA*'s (not shown) being significantly less than that.

For the environments used, the preprocessing took roughly 20 μs per triangle, with almost half the time being spent on each the triangulation and abstraction portions and an almost negligible amount on point location sectors.

Conclusion and Future Work

In this paper we have shown several benefits of triangulation-based pathfinding including precisely representing any polygonal environment, pathfinding for non-point objects, producing paths with arbitrary motion, significantly reducing the search space especially if triangulation graph reductions are applied, and providing anytime search algorithms which find the first path very quickly, and refining it to the optimal path. Compared with the standard A* search and its recent improvement PRA* our new algorithms TA* and TRA* perform better on a large set of maps.

There are many possible extensions to this method which warrant further exploration. The natural benefits to providing a channel instead of a single path could be useful when considering multiple units moving in a group. Also the abstraction information would be an excellent basis for terrain analysis, since dead ends, corridors, intersections, and choke points are already identified. Finally, many techniques designed for use with a grid representation have potential on a triangulation with minor modification.

Acknowledgments

We thank Nathan Sturtevant for his assistance with this paper. Financial support was provided by NSERC and iCORE.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *J. of Game Develop.* 1(1):7–28.

Hershberger, J., and Snoeyink, J. 1994. Computing minimum length paths of a given homotopy class. *Computational Geometry Theory and Application* 4:63–98.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI Vol. 1*, 530–535.

Kallmann, M.; Bieri, H.; and Thalmann, D. 2003. Fully dynamic constrained delaunay triangulations. In *Geometric Modelling for Scientific Visualization*. Springer-Verlag. 241–257.

Kallmann, M. 2005. Path planning in triangulations. In *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, 49–54.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. *AAAI* 1392–1397.