Fast Pathfinding Based on Triangulation Abstractions

Doug Demyen—BioWare Corp.,

doug.demyen@gmail.com

Michael Buro—University of Alberta

mburo@cs.ualberta.ca

Pathfinding is arguably the most fundamental AI task in current video games. No matter the technique used for the decision making of in-game characters, they lose the desired illusion of intelligence if they cannot navigate about their surroundings effectively. Despite its importance and that it is a well-studied problem, pathfinding is often performed using techniques that do not provide or take advantage of information on the structure of the environment.

In this article, we present an approach to pathfinding that addresses many of the challenges faced in games today. The approach is fast, uses resources efficiently, works with complex polygonal environments, accounts for the size of the object (for example, character or vehicle), provides results given varying computational time, and allows for extension to dynamic pathfinding, finding safe paths, and more. At the heart of this approach is an abstraction technique that removes all information from the environment that is extraneous to the pathfinding task.

Motivating Example

As an example, imagine a man planning a route between two houses in a city. If the originating house is in a bay, for example, the man can assume that as long as the destination house is not in that same bay, the start of the route will be to leave the bay.

After that, he won’t consider turning off onto side streets from which there is no exit unless they contain the destination because they would be dead ends. When the route reaches main roads, the man needs only consider at which intersections to turn; he ignores making decisions partway between intersections because the only possible options are to proceed or turn back, which would be nonsensical.

Each intersection represents a decision point in planning the route—they are where the man will decide to travel on the north or south side of the stadium for instance. After a series of these decisions, the route will reach the destination, and although the man ignored dead-end streets and going partway between intersections to this point, he can still plan a route to a house in a cul-de-sac or in between intersections on a street. You will also notice that the path is formed at the high level of streets, and after it is formed, particulars such as lanes can be determined. After all, it would make no sense to consider using each possible lane if after forming the complete path, it becomes obvious that the left lane is preferred or perhaps necessary. Our algorithm follows a similar high-level human-like decision-making process.

Outline

We will start by introducing triangulations, our environment representation, and in particular Dynamic Constrained Delaunay Triangulations (DCDTs), which provide many advantages for this work. We will cover some considerations for pathfinding with this representation and some for the extension to nonpoint objects (specifically circular objects with nonzero radius). From there, we describe the abstraction method used to achieve the simplified representation of the environment and how the search uses this

information. Finally, we provide some experimental results, draw conclusions, and suggest possible extensions to the work.

Pathfinding in Triangulations

Here we will introduce some different triangulations as well as how they are constructed and considerations for their use as an environment representation and for pathfinding.

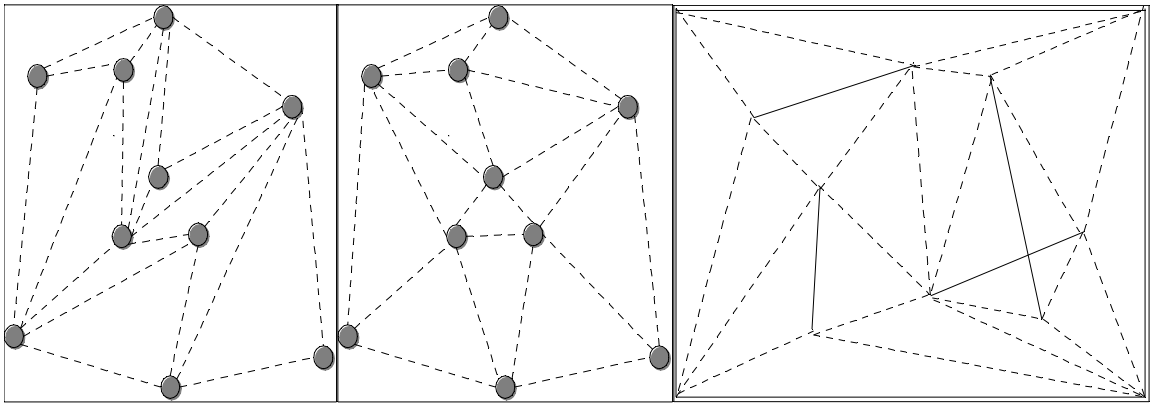
Types of Triangulations

A fundamental aspect of the methods represented here is the use of triangulations to represent the environment. Here we will briefly cover the different types of triangulations and how they relate to pathfinding.

Given a collection of vertices in two dimensions, a triangulation (see Figure 1a) is formed by joining pairs of these vertices by edges so that no two edges cross. When no further edges can be added, all faces in the convex hull of the vertices are triangular.

A special case is a Delaunay Triangulation (DT) (see Figure 1b) that specifies that the minimum interior angle of the triangles in the triangulation must be maximized. This avoids thin triangles wherever possible, which is a useful property that we will explore later. DTs can be constructed from triangulations by taking (convex) quadrilaterals in the triangulation formed by two triangles sharing an edge, and replacing that shared edge with one joining the other two vertices in the quadrilateral, whenever this results in a shorter diagonal.

Another version of a triangulation is a Constrained Triangulation (CT), which specifies that certain edges must be included in the final triangulation. We now make the distinction between these predetermined edges (called constrained edges) and those added during the triangulation process (called unconstrained edges). CTs are constructed in the same way as regular triangulations but with the constrained edges added first to ensure they are included. Constrained edges that cross are broken up at the intersection points. When used as an environment representation, a CT uses constrained edges to indicate barriers between traversable and obstructed areas.



*** Insert Figure 1a, Figure 1b, Figure 1c Here ***

Figure 1 (a, b, c) Examples of (from left to right) regular, Delaunay, and Constrained (Delaunay) Triangulations.

A CT can also carry the Delaunay property, forming a Constrained Delaunay Triangulation (CDT) (see Figure 1c). CDTs are formed from CTs using the same edge-flipping technique for creating DTs, with the added proviso that constrained edges cannot be flipped. CDTs maximize the minimum interior angle of the triangles as much as possible while maintaining constrained edges. This is the representation used by the

techniques described in this article but with one more technique that makes it ideal for use in games.

A technique presented by Marcelo Kallmann [Kallmann03] allows for the creation of DCDTs. This algorithm handles the online addition and removal of vertices or constrained edges in an existing CDT with minimal performance cost. Constraints are added to or removed from those already present, affected unconstrained edges are removed, and the surrounding area is re-triangulated, and then the Delaunay property is propagated out to areas that have since lost it. This update requires minimal resources and can be done in real time.

Triangulations offer many advantages for pathfinding over other environment representations, such as the ability to handle edges that are not axis-aligned. Specifically, triangulations can represent environments with straight barriers perfectly and can represent curved barriers using a number of short segments, providing an approximation that is superior to axis-aligned methods.

When stored, triangulations often require fewer cells than grid-based methods. This not only presents an advantage for pathfinding but also provides more information about the environment. For example, the traversibility of a tile contains no information on the surrounding area, whereas assuming all vertices and constrained edges in a triangulation represent obstacles (otherwise, they just add unnecessary complexity to the representation), a triangle indicates the distances to obstacles in each direction. This makes triangulations a perfect candidate for working with different-sized objects; you can determine if an object can pass through a section of the triangulation with relative ease using a technique introduced later.

Considerations for Pathfinding

The basis of triangulation-based pathfinding is the idea that paths are formed by moving from triangle to adjacent triangle across unconstrained edges, much like moving between traversable adjacent cells in tile-based environments. However, when using tiles, the exact motion of the object is known to go through the centers of the tiles (at least before smoothing) as the path is being formed. If you assume during the search that the path goes through the center of the triangles it traverses, the approximation of the path's length can be very poor because triangles are typically much larger than tiles. This can lead to suboptimal paths that can spoil the illusion of intelligence by moving an object to its destination by a longer than necessary path. Here we present requirements for finding optimal paths on a triangulation, which together form the first search algorithm presented, Triangulation A* (TA*).

Pathfinding (and, in fact, all heuristic search) uses a pair of values to guide its search: the distance traveled to the current point in the search, or g-value, and an estimate of the distance remaining, or h-value. To find an optimal path, the h-value must be no more than the actual distance remaining to the goal because overestimates could make the search abandon a branch leading to an optimal solution. The g-value is assumed to be exact, so when the search reaches somewhere that was reached by a shorter path, the current path is abandoned because taking the other path must be shorter.

However, for triangulations, a path being searched may enter a triangle through one edge and then leave through one of the two others. The full path between the start and goal points as a result of this decision (and likely subsequent ones) produces different paths leading to this triangle, and so the distance covered to reach it cannot be known

exactly during the search. Therefore, to produce an optimal solution, we must introduce two constraints. The first is that the g-value must not be larger than the true distance between the start of the search and the current triangle as it is reached in the final path to the goal. This follows the same logic in preventing the search from abandoning a potentially optimal path. The other constraint is that we cannot eliminate a node in the search simply because it was reached with a potentially shorter path because we do not know which path was shorter.

These requirements fit well with an *anytime algorithm*, that is one that finds a solution and improves it as long as it is given more resources. As with any other point in the search, when the goal is reached, the shortest path is not immediately known. Therefore, even after the goal is found, the search continues, accepting paths to the goal shorter than the best one currently known. Search is determined to have found an optimal path when the length of the shortest path found is less than the sum of the g- and h-values of the paths yet to be searched. This follows from these being underestimates of the path length, so any paths remaining in the search must be longer than the best one found.

Other Enhancements

To find the triangle that contains the start (and goal) point, you must perform a task called *point localization*. An inefficient approach, such as performing a greedy walk along adjacent triangles, would mask any benefits the triangulation could afford.

There is a simple but improved way to handle this task. First, the environment is divided into rectangular cells (for our experiments, a modest 10 x 10 grid was used). When the triangulation is constructed, triangles are tested as to whether they lie on the center point of any cells. If so, the triangle is recorded for that cell. When locating a point, its containing cell is determined easily, and the process of moving progressively closer to it is started from the triangle covering the midpoint of that cell. This results in shorter point localization times, allowing the full advantage of the triangulation-based methods.

In some cases, the possibility of the search visiting a triangle multiple times could mean the search converges more slowly on the goal. However, for maximum flexibility, we want to find the first path quickly in case the pathfinding task is not given much time. Therefore, we modified the search algorithm to only expand each triangle once until the first path has been found, after which they can be expanded again. This makes the first path available earlier without affecting the algorithm's ability to converge on an optimal path.

Triangle Width

One of the main challenges of pathfinding is dealing with objects larger than points. Incorporating this constraint is necessary to achieve paths that do not bring objects into collision with obstacles in the environment. A popular method for achieving this result

is to enlarge the obstacles in the environment by the radius of the object and then perform pathfinding as if for a point object. This technique has the drawback that a separate representation of the environment must be calculated and stored for each size of object, resulting in compounded time and memory costs. An advantage to the use of triangulations for pathfinding is their aptitude in handling this kind of problem.

We have developed a method for measuring the “width” of all triangles in a CDT, which is, for any two (unconstrained) edges of the triangle, the largest circular object that can pass between those two edges. We use circular objects because they require no consideration for orientation, and, in most cases, the pathfinding footprint of game objects can be fairly well approximated by a circle of some radius.

After this is calculated for all triangles in the triangulation, pathfinding for an object of any size can be done as if for a point object except that paths which traverse between two edges of a triangle with a width less than the object’s diameter are excluded. The calculation does not require much processing and memory and is done once only. This allows for objects of any size, eliminating the restrictive need to create game objects of discrete sizes for the sole purpose of pathfinding.

Finding the width for the traversal between two edges of a particular triangle is equivalent to finding the closest obstacle (a vertex or point on a constrained edge) to the vertex joining those two edges, in the area between them. If one of the other vertices of the triangle represents a right or obtuse angle, the closest obstacle is the vertex representing that angle, and the width of the triangle is the length of the edge joining this vertex to the one where the two edges meet.

Otherwise, if the edge opposite the vertex in question is constrained, the closest obstacle is the closest point to the vertex on that edge, and the width is the distance between them. Finally, if the edge opposite the vertex being considered is unconstrained, a search across that edge will determine the closest obstacle to that vertex. This search is bounded by the shorter of the distances to the other two vertices in the triangle because they are potential obstacles. It considers vertices in the region formed by the extension of the edges of the original triangle for which the calculation is being done and constrained edges in this region that would form acute triangles if their endpoints were connected to the base vertex.

Note that because the search is always bounded by the distance to the closest obstacle found so far and that Delaunay triangulations make it impossible for the search to traverse any triangle multiple times, this operation can be performed on a triangulation very quickly.

Modified Funnel Algorithm

The result of pathfinding in a triangulation is a sequence of adjacent triangles connecting the start to the goal called a *channel*. However, because triangles are larger than tiles, it does not translate directly into an efficient path through them. Luckily, you can find the shortest path through a channel quickly using a *funnel algorithm* (see Figure 2a). This algorithm has the effect of conceptually pulling a rubber band through the channel between the start and the goal, producing a sequence of line segments touching the vertices of the channel and forming the shortest path.

However, this operation is meant for point objects, and our generalized solution seeks to find shortest paths for circular objects of nonzero radius. Therefore, we developed a modified version of this algorithm (see Figure 2b) that basically consists of adding a circle with the same radius as the object centered around each vertex in the channel except the start and goal vertices. The shortest path is found by a similar method but now consists of arcs along these circles and line segments between and tangent to them to avoid collision with the obstacles.

Some considerations to keep in mind are that this algorithm assumes that the channel is wide enough to accommodate the object in question. Although this technique produces the optimal path for the object through the channel, it assumes the object is capable of traveling in a curve. If this is not the case, the object can approximately follow the arcs produced by this algorithm by traveling in several short straight segments, turning in between.

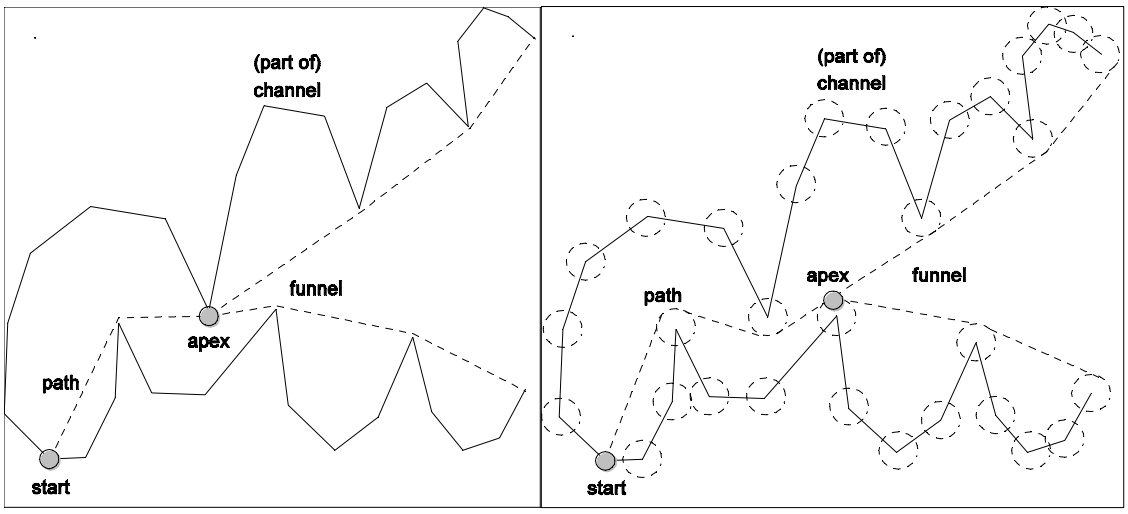


Figure 2 (a, b) The funnel algorithm (left) determines paths for point objects, and the modified version (right) produces paths for circular objects of some radius.

Triangulation Abstraction

The most important part of the process we use to reduce the pathfinding graph produced by the triangulation (see Figure 3a) is a simple classification of each triangle as a node in the abstract graph by *level*. We do this by assigning each triangle an integer value between 0 and 3 inclusive, indicating the number of adjacent graph structures. The graph resulting from this procedure (see Figure 3b) caries additional information about the structure of the environment.

Level-0 nodes, or *islands*, are simply triangles with three constrained edges. These are easily identified when the algorithm passes over the triangles in the triangulation.

Level-1 nodes form *trees* in the graph and represent dead ends in the environment. There are two kinds of level-1 trees in a reduced graph: *rooted* and *unrooted*. The root of a rooted tree is where the tree connects to the rest of the graph (via a level-2 node). Unrooted trees have no such connection; they are formed in areas of the graph that do not encompass other obstacles.

Level-1 nodes are identified as triangles containing two or fewer constrained edges and containing at most one unconstrained edge across which is a level-2 node. Level-1 nodes with two constrained edges are easily found in a first pass over the triangulation, and for each of these found, the triangle across the unconstrained edge is put in a queue for processing as a possible level-1 node. Each triangle on the queue is evaluated if it now fits the description of a level-1 node, and if so, is classified as one; the unclassified triangle adjacent to it across an unconstrained edge (if one exists) is put on the queue for processing. This process will propagate through a rooted tree until the root is reached, or for an unrooted tree, throughout the whole connected component.

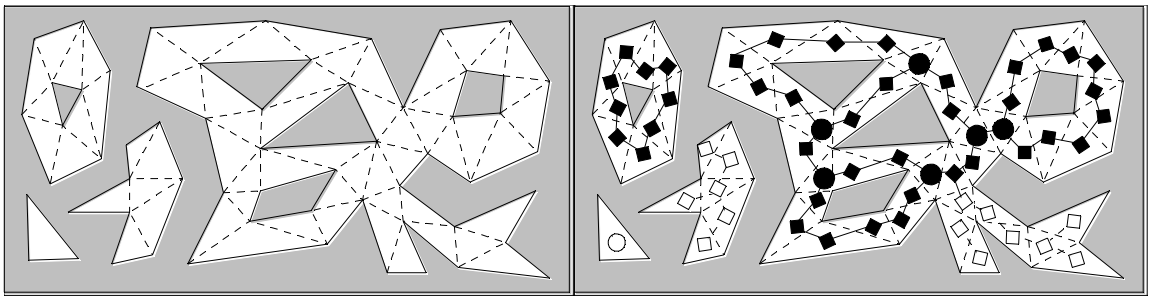


Figure 3 (a, b) A triangulation (left) is reduced to an abstract graph (right) where circles, squares, filled squares, and filled circles represent level-0, -1, -2, and -3 nodes, respectively.

Level-2 nodes represent *corridors* in the environment and are adjacent (across unconstrained edges) to two nodes that are either level-2 or level-3. A connected group of level-2 nodes can form a corridor between two distinct level-3 nodes, a loop beginning and ending at the same level-3 node, or a ring with no level-3 beginning or end. All triangles remaining after the level-0, -1, and -3 nodes are identified are classified as level-2.

Level-3 nodes are the most important in the pathfinding search because they identify *decision points*. Search from a level-3 node can move directly to level-3 nodes adjacent across either unconstrained edges or level-2 corridors and represent choices as to which direction to pass around an obstacle. After level-0 and level-1 nodes are identified, level-3 nodes are those triangles with neither constrained edges nor adjacent level-1 nodes.

Abstraction Information

In addition to each triangle's level, the abstraction stores other information about each node in the environment for use in pathfinding. The adjoining node is recorded for each direction depending on its type. For level-1 nodes in rooted trees, the root of the tree is

recorded for the edge through which it is reached. For level-2 nodes not in rings, they are the level-3 nodes reached by following the corridor through the edges for which they are recorded. For level-3 nodes, they are the level-3 nodes adjacent directly or across level-2 corridors in each direction.

The abstraction is also where a triangle's widths (between each pair of edges) are held. It also stores the minimum width between the current triangle and each adjoining node so the search can tell instantly if the object can reach that node.

We also included an underestimate of the distance between the current triangle and each adjoining node to be used in the search to improve the accuracy of this value and make the search more efficient.

Abstraction Search

Finding a path on the reduced triangulation graph requires more steps than performing the search on the base triangulation. First, a number of special cases are examined to determine if a search of the level-3 nodes needs to be done at all, then the start and goal points need to be connected to level-3 nodes on the most abstract graph, and finally, a search between level-3 nodes is run. This is the basis for Triangulation Reduction A* (TRA*), described later. As before, at each step, the width of the triangles being traversed is checked against the diameter of the object for which pathfinding is being performed, and paths that are too narrow for it are not considered.

The simplest check performed is to see if the endpoints are on the same connected component in the environment—that is, they are not in separate areas divided by constrained edges. Because identifying the different components requires no more

processing on top of the reduction step, we can instantly see if there are any possible paths between them. If they are on different connected components, no path can exist between them. If they are on the same one, there is a path between them, and the only question is whether it's wide enough to accommodate the object. You can then check whether the endpoints are in the same triangle; if so, the path between them is trivial. This covers when the endpoints are in the same island triangle.

Next we check whether the endpoints are in an unrooted tree or in a rooted tree with the same root. In these cases, we can search the tree for the single path between the start and the goal. Because trees are acyclic (no two triangles can be joined by multiple paths that do not visit other triangles more than once), we can eliminate aspects of the search meant for finding the shortest path because only one exists (other than those containing cycles that needlessly lengthen the path). The result is a simplified search where the midpoints of the triangles are considered as exact points on the path, the Euclidean distances between them are used as distance measures, and no triangle needs to be considered twice. Also in the case of rooted trees, the search need not venture outside the tree. Note that these searches are so localized and simple that they are almost trivial in nature (see Figure 4a).

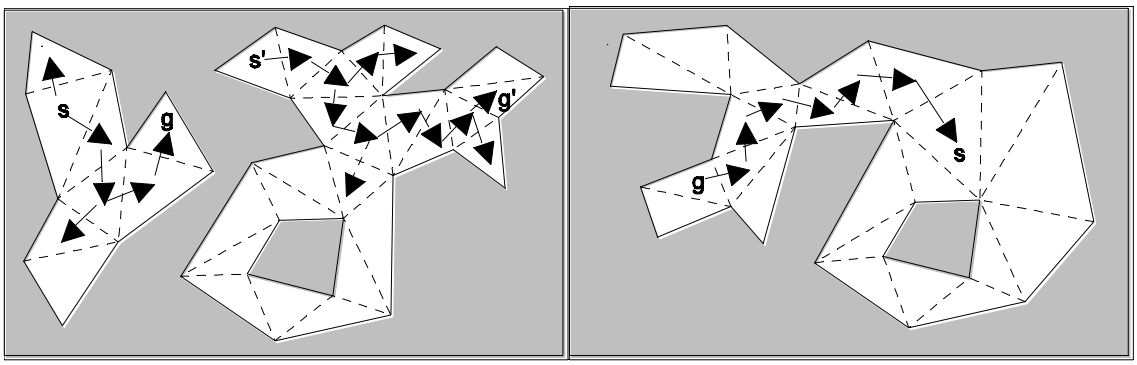


Figure 4 (a, b) Cases where the endpoints are in the same tree and a path is easily found.

Then for search endpoints in level-1 nodes, we search moves to the root of the tree. In some cases, the other endpoint will be at the root of this tree. This can be determined instantly and the optimal path constructed easily by simply moving along the one (acyclic) path to the root of the tree (see Figure 4b). Otherwise, the search next examines patterns with level-2 nodes.

If both endpoints are on level-2 nodes (or in level-1 trees rooted at level-2 nodes) on a ring or the same loop (see Figure 5a), there are two possible paths between them—going clockwise or counterclockwise around the ring or loop. Both of these paths are fully constructed, and the shorter of the two is taken as the optimal path.

If the level-2 nodes associated with the endpoints are on the same corridor (see Figure 5b), we form one path along that corridor and determine its length, and then the level-3 nodes found by going the opposite directions are considered the start and goal nodes for the level-3 search, respectively. The level-3 node search is performed as usual from here, except that the search now has an upper bound: the length of the path already found.

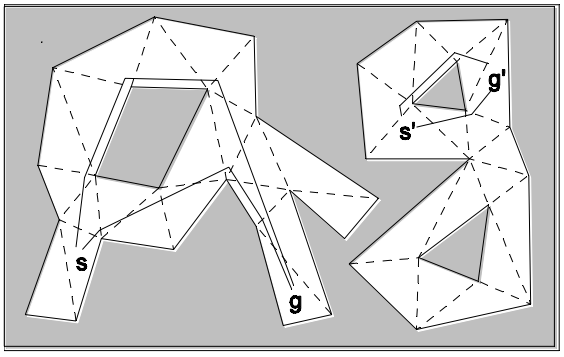


Figure 5 (a, b) The start and goal can also be on the same level-2 corridor, loop, or ring.

If none of these cases applies, the search travels from the level-2 nodes associated with the start to the level-3 nodes on either end of that corridor. These are the starting points for the level-3 node search. If the starting point is on a level-3 node, there is only one starting point for this search. The same procedure is performed for the goal point—potential goals are the level-3 nodes at either end of the corridor if the goal point is on a level-2 node, and if it was on a level-1 node, from the corridor on which the goal node's tree is rooted. If the goal point is on a level-3 node, that is one goal for the level-3 search.

The search from here is performed similarly to TA*, except instead of moving across unconstrained edges to adjacent triangles, it moves across corridors of level-2 nodes to other level-3 nodes. A few additional techniques are available for estimating distances on the abstract graph. The same tests for g- and h-values, the anytime algorithm, and the revisiting of nodes are performed as before.

Discussion

The criteria that decide about the adoption of new algorithms in video games are their space and time requirements, quality of the results, versatility, and simplicity. Usually at least one of these conditions is violated—in our case, it's simplicity.

The implementation of TA* and TRA* relies on efficient code for point localization and maintaining Delaunay triangulations dynamically. For this, we use Marcello Kallmann's DCDT library [Kallmann03] whose point localization procedure we improved. Dealing

with arbitrarily located points usually complicates computational geometry algorithms due to limitations of integer or floating point-based computations. The DCDT library we used is general and complex. However, for new game applications, it's conceivable that all line segment endpoints are located on a grid, and segments only intersect in grid points. This constraint greatly simplifies the DCDT algorithm. In addition, the TA* and TRA* abstraction and search mechanisms are not exactly easy to implement, although the software provided on the ***insert CD-ROM icon*** CD-ROM can help AI programmers get familiar with the technique and test it in their settings.

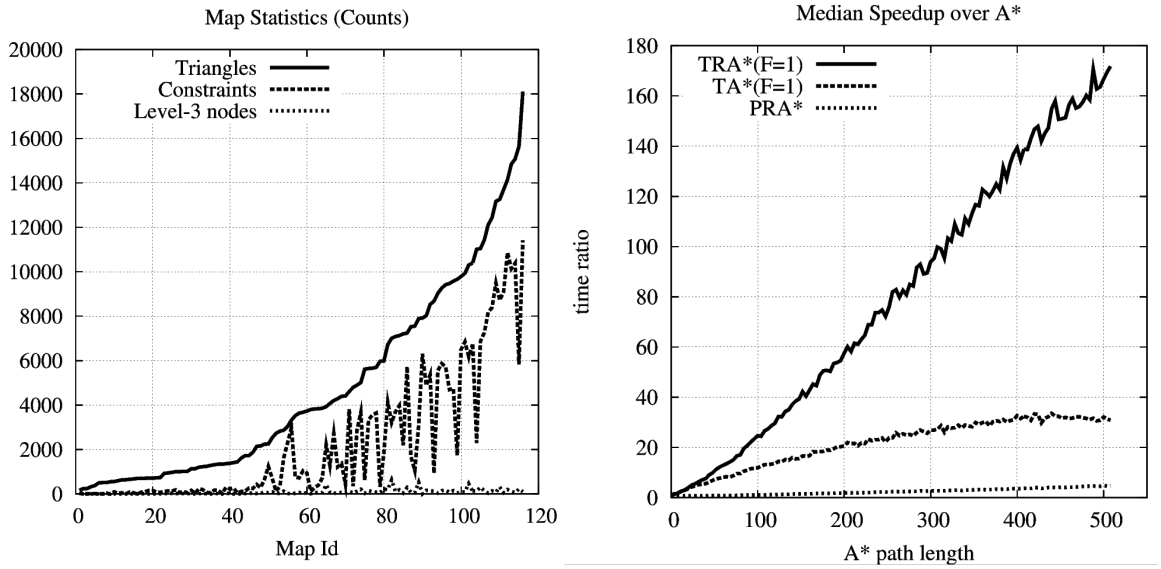


Figure 6 (a, b) Environments have few triangles and level-3 nodes, giving TA* and TRA* greater speedup over A* than even enhanced grid-based methods such as PRA*.

The space requirement of TRA* is only slightly larger than the original polygonal map description because the size of the abstraction is linear in the number of islands in the world, which is usually orders of magnitudes smaller than the total number of triangles.

Moreover, compared with grid-based representations, the space savings when using triangulations at the lowest level can be substantial if there are big unobstructed areas (see Figure 6a). In the experiments we touch on here [DemyenBuro06, Demyen06], we used 120 maps taken from *Baldur's Gate* and *Warcraft 3* scaled up to 512 x 512 tiles, and the total memory requirement for TRA* was at most 3.3 MB. We did not try to optimize memory consumption, and with 184 bytes per triangle allocated by the DCDT library, there is certainly room for improvement.

TRA*'s runtime can be broken down into two components: map preprocessing time (triangulation, reduction, sector computation) and actual pathfinding time. The most complex maps could be preprocessed within 400 milliseconds (ms) on an Athlon 64 3200+ computer, which were split roughly in half between triangulation and reduction. The median preprocessing time was 75 ms. In this set of experiments, we focused on static environments. However, you can repair triangulations and the reduced graph efficiently if changes are local. TA* and TRA* are considerably faster than grid-based A* . We observed 170× median speedups over A* for TRA* and 30× for TA*, for finding the first approximation of optimal paths of length 512 (see Figure 6b). The absolute times for TA* (see Figure 7a) and TRA* (see Figure 7b) show they work well for real-time applications.

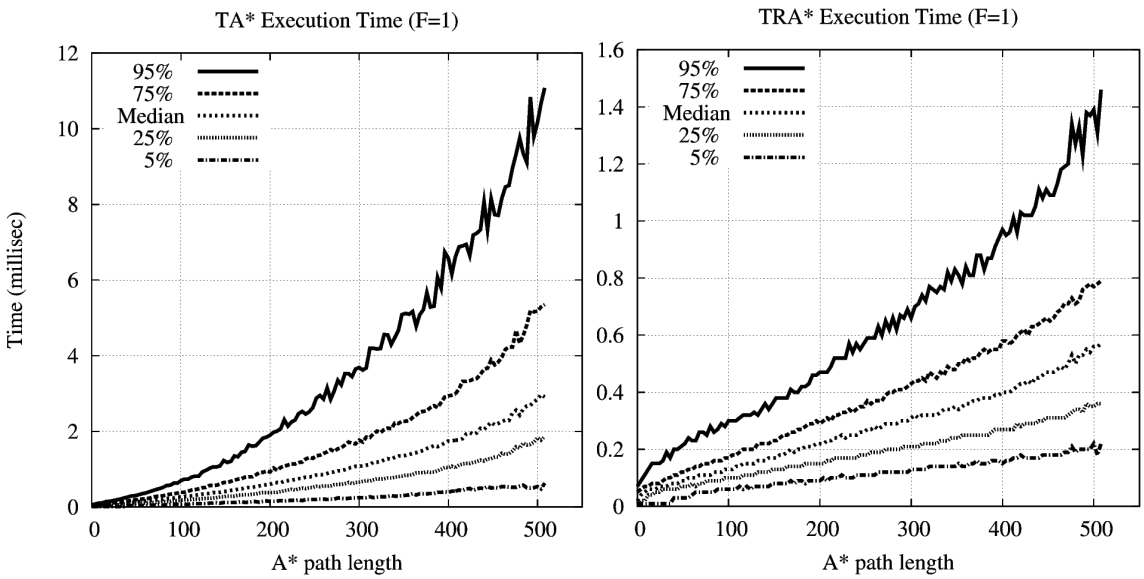


Figure 7 (a, b) TA* and TRA* find a path within a couple milliseconds.

In over 95% of the considered cases, the length of the path first reported by TA* is shorter than the grid-A* path. We know that A* computes shortest paths, so this statement doesn't seem correct. However, the object motion in grid-A* is restricted to eight directions, whereas in triangulation-based pathfinding, objects can move freely. The TRA* path quality reaches that of grid-A* if after finding the initial path, we continue to search for better paths for the time it took to find the first. Thus, equating path quality, TRA* is about 85 times faster than grid-A* when finding long paths in the maps we considered. Note this is an abridged version of a more complete experimental analysis provided in the accompanying thesis [Demyen06].

Triangulation-based pathfinding as we described it is not only fast but also versatile. TA* and TRA* can be regarded as anytime algorithms: The more time we invest after the initial search phase, the shorter paths become. These algorithms also find optimal

paths for moving circles of varying size, which is useful for group pathfinding when we use bounding circles. Triangulations are also very suited for detecting strategic terrain features, such as chokepoints.

Conclusion

We have shown the usefulness of triangulations for environment representations, both in efficiency and for the benefits it affords to pathfinding. We have also shown enhancements to pathfinding on the triangulation itself, providing an anytime algorithm for finding better paths when given more resources and converging on the optimal path.

The main contribution of this work, however, is the reduction step performed on the triangulation. On top of identifying useful structures in the environment, it allows for much faster pathfinding. Coupled with the many opportunities for extending this work for different needs and situations outlined next, we hope the efficiency and flexibility of these techniques will find application in the games industry.

Future Work

One of the most exciting aspects of these techniques is their suitability to further extension. Among these is the ability to deal with dynamic environments. For example, if mobile obstacles block an object's path, it could possibly steer around the object within its channel to avoid running the pathfinding search again. If pathfinding is being done for a group of objects, one search could yield a channel for all objects to use. In the case of a narrow path and many or large objects, more paths could be found, and the objects split between them to meet up at the goal. If paths are being found for multiple objects going in different directions, you could avoid collisions by recording at which

times each object will be going through a triangle when its path is found. How crowded a triangle is at any time could be calculated based on the size of the triangle and the size and number of objects going through it at that time. When finding paths for subsequent objects, those going through crowded triangles could be avoided, and some steering should be adequate to avoid collisions.

There are also several possible extensions if more precomputation is a desired tradeoff for more speed. For example, precalculating the best paths between level-3 nodes would require a fraction of the memory required by most navigation mesh approaches. The pathfinding task would only require moving from the start and goal to adjoining level-3 nodes and fetching the rest from a table. The level-3 node graph could be abstracted even further by collapsing any doubly connected components of this graph into single nodes in a higher-level graph. This graph would then consist of a group of trees, and because paths in trees are trivial, only pathfinding between the entry points of the doubly connected components would be necessary. If some suboptimality is acceptable, you could even precalculate and cache paths between these entry points for lightning-fast pathfinding with minimal memory cost.

If pathfinding needs to be done for only a few sizes of objects, separate environment representations could be constructed for each. The exchange for the increased memory would be instant knowledge of a path existing for a particular object and not having to test paths for minimum width requirements.

You could also use these techniques in more complex cases. Pathfinding on the surface of 3D environments could be done by triangulating the passable surfaces. Overlapping areas such as bridges could be handled by forming separate triangulations and creating

virtual links between the edges. These links could also be given costs to simulate additional time or effort for moving between meshes by jumping or climbing ladders, for example.

If objects need to take paths with certain properties, such as being clear of enemies or containing enemies whose total power is less than the object, then other information such as the “threat” of an enemy can be localized to a triangle and propagated through the abstract graph in the same way as triangle widths. The pathfinding search could then avoid returning paths that traverse corridors where the total enemy power is greater than a certain threshold.

Source Code and Demo

The software that accompanies this book contains Marcello Kallmann’s DCDT implementation ***insert CD-ROM icon*** , with the work shown here built on top. Functions of interest are `SearchPathBaseFast`, `SearchPathFast`, and `Abstract`, which implement TA*, TRA*, and the reduction process, respectively. The executables are found in the se/bin directory—setut.exe will run a GUI for visualizing pathfinding in a reduced triangulation. Press “6” when the program opens to see the DCDT, noting the red constrained edges, gray unconstrained edges, yellow level-1 trees, green level-2 corridors, cyan level-0 islands, and magenta level-3 decision points. Click two points to find a path between them; the black lines are the channel, and the blue lines are the path. You can also drag the obstacles around and see the triangulation, abstraction, and path change. The information contained in the abstraction for the triangle over which the mouse is currently positioned is printed in the console window.

Acknowledgments

We thank Marcelo Kallmann for making his DCDT software available to us, allowing us to get so far so quickly. Financial support was provided by NSERC and iCore.

References

[DemyenBuro06] Demyen, D. and Buro, M., "Efficient Triangulation-Based Pathfinding." *Proceedings of the AAAI Conference*, Boston (2006): pp. 942–947.

[Demyen06] Demyen, D., "Efficient Triangulation-Based Pathfinding." Master Thesis, Computing Science Department, University of Alberta, Edmonton, Canada. Available online at *http://www.cs.ualberta.ca/~mburo/ps/thesis_demyen_2006.pdf*, 2006.

[Kallmann03] Kallmann, M. et al., "Fully Dynamic Constraint Delaunay Triangulations." *Geometric Modeling for Scientific Visualization*, Springer Verlag, 2003: pp. 241–257,.