Algorithms Used in Pathfinding and Navigation Meshes

Corey Trevena

Introduction

Finding the shortest path between a starting point and end point has many algorithmic solutions, but as demands for the shortest path move from static to dynamic environments, these algorithms have needed to become more complex. This paper will address the algorithms used in this pathfinding, starting with Dijkstra's Algorithm, and expanding on Dijkstra the A* algorithm, Dynamic A* (D*) algorithm, and the Anytime Dynamic A* (AD*) algorithm. The core of this paper will focuses on Dynamic A* and Anytime Dynamic A*, before wrapping up with software-based solutions for pathfinding in environments called Navigational Meshes (Kallman & Kapadia, 2014).

History

Finding the shortest path is a well-known problem in Computer Science, but it is by far not a new problem for humanity. From the shortest paths of trade routes, to the shortest routes to ship cargo by, the shortest path problem has been the focus of mathematical research long before computers were ever able to be utilized (Schrijver, 2010). Matrix methods were developed in 1946 for their applications to communication networks, and shortly after shortest-length paths were addressed by Bellman-Ford and Dijkstra in 1956 and 1959, respectively (Schrijver, 2010). In recent years, the shortest path problem has had to contend with the complexities of the real world through robotics, and the virtual world via video games and simulations.

Dijkstra's Algorithm

Dijkstra's algorithm requires a connected graph where all edges have non-negative weights. A node is selected to be the starting point and is initialized to a distance of zero, and all other nodes' tentative distances to infinity. The nodes are then traversed, starting from the source node, expanding outward to the next node of the lowest tentative distance. All of the connected unvisited neighbors from the current node have their distances calculated by: Distance to current node + Distance from current node to neighbor. If the value returned is less than the node's current tentative distance, it replaces the value with the lesser one. All nodes are initialized to infinity to ensure discovery by the algorithm. The algorithm continues through the weighted graph, updating the distances of unvisited nodes when needed, adding nodes to the visited set once all of its neighbors have been considered. This continues until the goal node is reached and added to the visited set. Once the goal node has been added to the visited set, the shortest path from the source node to the goal has been found.

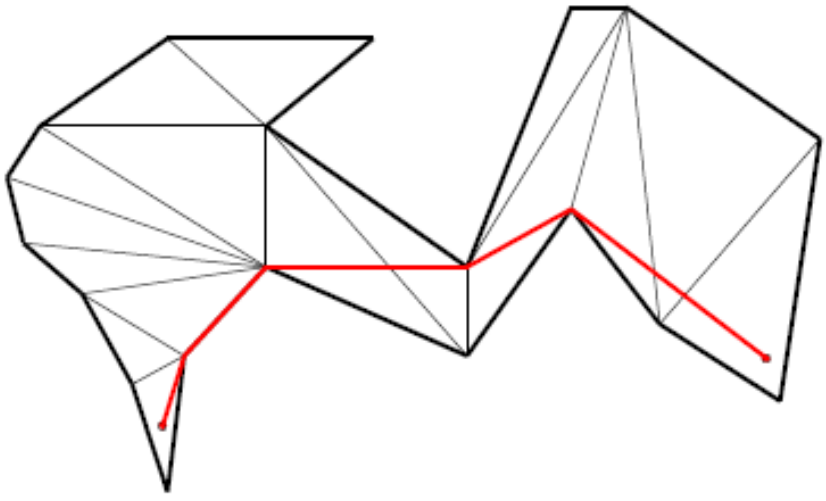

Figure 1: *The Euclidean Shortest Path between two points inside a triangulated simple polygon. (Kallman & Kapadia, 2014, p.2) Start and endpoints are placed within the polygon, the red line between is the ESP.*

Concepts and Terminology: Basic Visual Representations

The first concept that is important in understanding visualizing pathfinding are the Euclidean Shortest Paths (ESPs). These paths, when no shorter path exists, are globally optimal and show the shortest path between two points (Kallman & Kapadia, 2014, p.1). As shown in Figure 1, this can be simply shown when obstacles in an environment are reduced to a simple polygon, with the start and endpoints located within the bounds of the polygon. The ESP is shown in red between the two points.

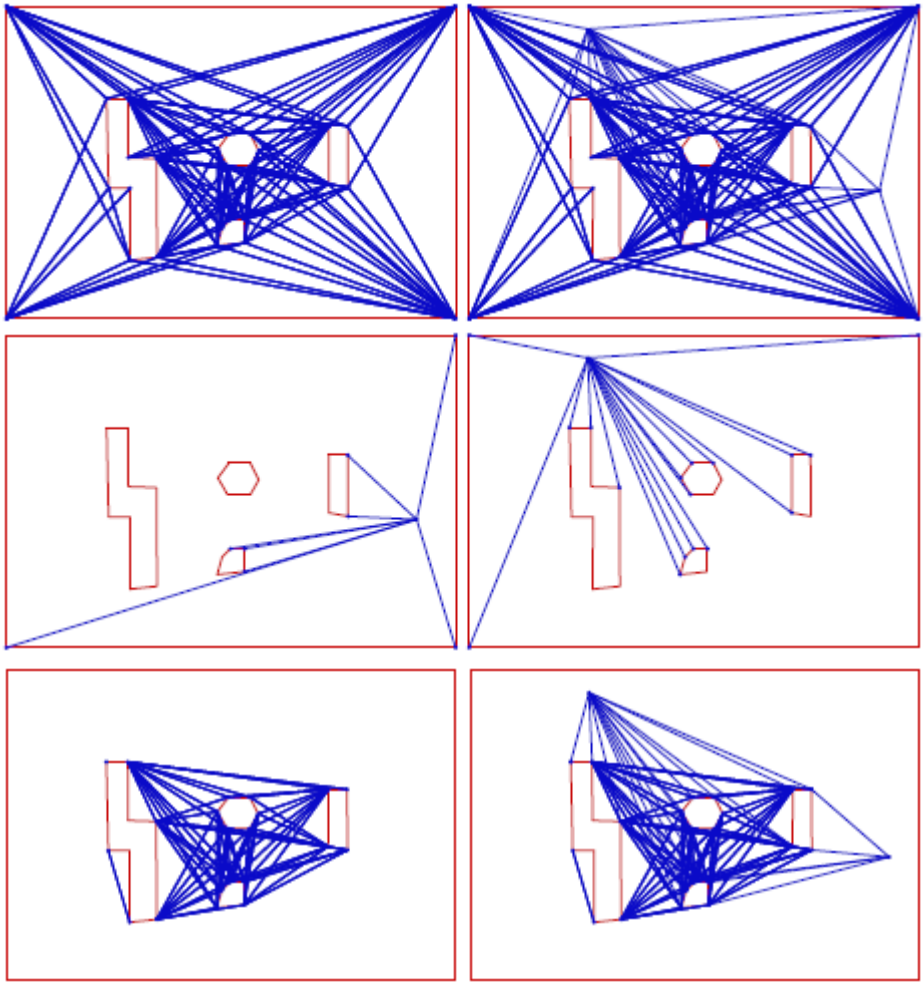

Figure 2: *"The Euclidean shortest path between p and q can be found by searching the visibility graph of S (top-left) augmented by the edges connecting all visible vertices to p and q (top-right). The [middle] diagrams show the added edges connecting the visible vertices to p and q." (Kallman & Kapadia, 2014, p.2)The bottom diagrams are popular optimizations on visibility graphs, discarding the edges that lead to non-convex vertices both in processing the environment (bott-left) and with source and destination points (bott-right). (Kallman & Kapadia, 2014, p.3).*

The simple polygon representation of Figure 1 is misleading, as the generic case in finding Euclidean shortest paths often includes multiple corridors and exits (Kallman & Kapadia, 2014, p.2). These multiple corridors and exits are multiple routes that need to be considered by the algorithm to reach the endpoint. A visibility graph, as shown in Figure 2, is a graph of the set S of all polygonal obstacles existing in some space, and is composed of all segments connecting vertices that are visible to each other in S (Kallman & Kapadia, 2014, p.2). As shown in Figure 2, the obstacles (in red) block the path between points p and q, with augmented edges (in blue) providing multiple paths around these obstacles. Thus, after these edges connecting the visible vertices are established, graph search algorithms such as Dijkstra and A* can then be applied to the visibility graph to find the Euclidean shortest paths from p to q.

The A* Algorithm

A* is a Dijkstra variant that uses a heuristic cost function h to progress faster towards the endpoint q without lessening the optimality of the solution (Kallman & Kapadia, 2014, p.3). The heuristic cost h is the "cost-to-go", which unlike distance traveled so far (g), is often implemented as distance to the endpoint. This is superior to Dijkstra due to the fact that A* takes into account both the distance traveled, and the distance

Algorithm 1 A* Graph Search

```
1:  function SHORTESTPATHINGRAPH ( p, q )
2:      Initialize priority queue Q with p;
3:      Mark node of p as visited;
4:      while ( Q not empty ) do
5:          s ← Q.remove();
6:          if ( s = node containing q ) then
7:              return branch from s to q;
8:          for all ( neighbors n of s ) do
9:              if (n not visited or g(n) > g(s) + d(s, n)) then
10:                 Set the parent of n to be s;
11:                 Set g(n) to be g(s) + d(s, n);
12:                 Insert n with cost g(n) + h(n, q) in Q;
13:                 Mark n as visited;
```

Figure 3: *The A* Algorithm (Kallman & Kapadia, 2014, p.3)*

left to reach the goal node (Kallman & Kapadia, 2014, p.3). This lets A* favor nodes closer to its goal, rather than expanding outwards to the next node of the lowest tentative distance.

Dijkstra's algorithm is guaranteed to find the shortest path possible, but A* is far faster at finding a path to the goal node. This means that A* is better for real-time pathfinding, due to its speed. It still runs into issues with real-world pathfinding problems, due to the ever-changing conditions present within dynamic environments, needing to recalculate its entire path when new information is received.

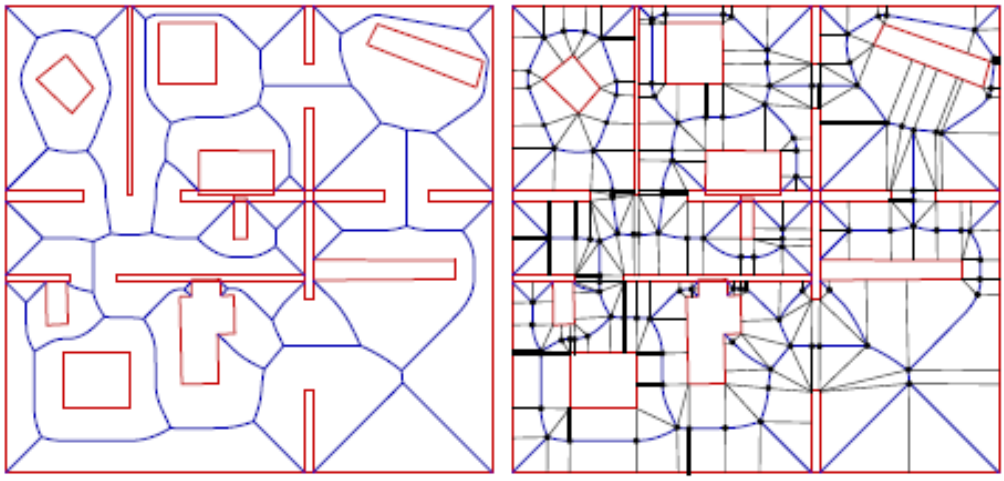

Figure 4: *The medial axis (left) represents points of maximum clearance. This can be decomposed, such that each edge is associated to its closest pair of obstacle elements (right). (Kallman & Kapadia, 2014, p.4).*

The Medial Axis

Visual graphs have one key issue: they fail to address constraints that a pathfinding object may have. Such a key constraint could be clearance between the object and its obstacles. These constraints are taken into account by spatial partitioning

structures (Kallman & Kapadia, 2014, p.4), such as Voronoi diagrams and with medial axis, as show in in Figure 4. These paths shown by the medial axis may not be Euclidean shortest paths, but in trade-off naturally allows the integration of clearance constraints (Kallman & Kapadia, 2014, p.4). The medial axis approach allows for algorithms to find the best locally shortest path for an object quickly; thus path can then be easily interpolated towards the medial axis in order to reach maximum clearance when needed (Kallman & Kapadia, 2014, p.4).

Dynamic and Anytime Algorithms: Overview

Path planning for the real world involves dealing with an inherently uncertain and dynamic place; models are difficult to obtain and quickly go out of date and time for deliberation is often very limited (Likhachev et al. 2005, p.1). Thus, replanning algorithms are needed to correct precious solutions on updated information in this ever-changing, dynamic environment. In addition to dynamic environments, when a pathing problem is complex, the optimal solution may not be able to be found within the deliberation time (Likhachev et al. 2005, p.1). Anytime algorithms aim to address this issue by finding a suboptimal solution quickly, and then improve upon this solution until time for planning is exhausted (Likhachev et al. 2005, p.1). The interesting issues in modern pathfinding are both the dynamic and complex. Robotics, for example, run into path planning problems that both require replanning, and need to find a solution within a limited timeframe. Thus, an anytime and dynamic algorithm is required in order to solve these modern-day problems. Before joining dynamic and anytime approaches, the dynamic expansion on A* will be discussed.

key(s)
01. return $[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))]$;

UpdateState(s)
02. if s was not visited before
03. $\quad g(s) = \infty$;
04. if $(s \neq s_{goal})\ rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
05. if $(s \in OPEN)$ remove s from *OPEN*;
06. if $(g(s) \neq rhs(s))$ insert s into *OPEN* with key(s);

ComputeShortestPath()
07. while $(\min_{s \in OPEN}(\text{key}(s)) \dot{<} \text{key}(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start}))$
08. $\quad$ remove state s with the minimum key from *OPEN*;
09. $\quad$ if $(g(s) > rhs(s))$
10. $\quad\quad g(s) = rhs(s)$;
11. $\quad\quad$ for all $s' \in Pred(s)$ UpdateState(s');
12. $\quad$ else
13. $\quad\quad g(s) = \infty$;
14. $\quad\quad$ for all $s' \in Pred(s) \cup \{s\}$ UpdateState(s');

Main()
15. $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty$;
16. $rhs(s_{goal}) = 0$; *OPEN* $= \emptyset$;
17. insert s_{goal} into *OPEN* with key(s_{goal});
18. forever
19. $\quad$ ComputeShortestPath();
20. $\quad$ Wait for changes in edge costs;
21. $\quad$ for all directed edges (u, v) with changed edge costs
22. $\quad\quad$ Update the edge cost $c(u, v)$;
23. $\quad\quad$ UpdateState(u);

Figure 5: *D* Lite Algorithm. (Likhachev et al. 2005, p.3)*

Dynamic A* (D*) and D* Lite

The dynamic expansion for A* is called Dynamic A*, or D* for short. Dynamic A* aims to be able to replan its optimal path when new information arrives (Likhachev et al.

2005, p.1), thus being able to address dynamic environments. Dynamic A* is a popular choice for a replanning algorithm, and has been shown to be up to two orders of magnitude more efficient than A* replanning from scratch, when new information is received (Likhachev et al. 2005, p.2). Dynamic A* is optimized further into D* Lite, and since these two are fundamentally similar, D* Lite will be covered since it has been found to be slightly more efficient than Dynamic A* (Likhachev et al. 2005, p.2).

"D* Lite maintains a least-cost path from a start state $s_{start} \in S$ to a goal state $s_{goal} \in S$, where S is the set of states in some finite state space" (Likhachev et al. 2005, p.2). D* does this by storing the cost from each state s to the goal in g(s), and stores a one-step lookahead cost rhs(s) which satisfies:

$$rhs(s) = \begin{cases} 0 & if\ s = s_{goal} \\ min_{s' \in Succ(s)}(c(s,s') + g(s')) & otherwise, \end{cases}$$ where $Succ(s) \in S$ denotes

the set of successors s and c(s,s') denotes the cost of moving from s to s', the arc cost (Likhachev et al. 2005, p.2). Any number of goals can be incorporated, and in such a case, s_{goal} is the set of goals. The state is called consistent if the cost to the goal, g(s), is the same as the one-step lookahead cost rhs(s). Consistent states are unchanged, while overconsistent (if $if\ g(s) > rhs(s)$) or underconsistent states are in need of updating. The algorithm also utilizes a priority queue that helps it to focus its search and run more efficiently (Likhachev et al. 2005, p.2). The priority (key value) of a state s in the queue is (Likhachev et al. 2005, p.2):

$$key(s) = \begin{matrix} = [k1(s), k2(s)] \\ = \left[\min\left(g(s), rhs(s)\right) + h(s_{start}, s), \min\left(g(s), rhs(s)\right)\right] \end{matrix}$$

The algorithm restricts its attention to the relevant paths, and thus ensures that a least-cost path will have been found when the algorithm has finished (Likhachev et al. 2005, p.3).

```
1:  function KEY(s)
2:      if (g(s) > rhs(s)) then
3:          return [rhs(s) + ε · h(s, s_goal); rhs(s)];
4:      else
5:          return [g(s) + ·h(s, s_goal); g(s)];

6:  function UPDATESTATE(s)
7:      if (s ≠ s_start) then
8:          s' = arg_{s'∈pred(s)} min(c(s, s') · M_C(s, s') + g(s'));
9:          rhs(s) = c(s, s') · M_C(s, s') + g(s');
10:         prev(s) = s';
11:     if (s ∈ OPEN) remove s from OPEN;
12:     if (g(s) ≠ rhs(s)) then
13:         if (s ∉ CLOSED) insert s in OPEN with key(s);
14:         else insert s in INCONS;
15:     Insert s in VISITED;

16: function COMPUTEORIMPROVEPATH(t_max)
17:     while (min_{s∈OPEN}(key(s) < key(s_goal) ∨ rhs(s_goal) ≠
    g(s_goal) ∨ Π(s_start, s_goal) = NULL) ∧ t < t_max do
18:         s = arg_{s∈OPEN} min(key(s));
19:         if (g(s) > rhs(s)) then
20:             g(s) = rhs(s);
21:             CLOSED = CLOSED ∪ s;
22:         else
23:             g(s) = ∞;
24:             UpdateState(s);
```

Figure 6: *Anytime Dynamic A* algorithm: Typical modern implementation (Kallman & Kapadia, 2014, p.8).*

Anytime Dynamic A* (AD*)

The Anytime expansion on Dynamic A* is Anytime Dynamic A*. The algorithm is an expansion on D*, aiming to help solve the problem D* has in a time-constrained environment. Anytime algorithms must be satisfied with their solution to a pathing problem by the end of their allotted computing time since real-world scenarios, such as those faced in drones and robotics, do not have unlimited time to sit and completely calculate out the globally optimal solution after each step (Likhachev et al. 2005, p.3).

The algorithm shown in Figure 6 is the typical implementation of Anytime Dynamic A* found today (Kallman & Kapadia, 2014, p.8). AD* works just as D* Lite above, but unlike D* Lite, instead of processing all inconsistent nodes only those whose costs are beyond the inflation factor ϵ are expanded (Kallman & Kapadia, 2014, p.8). The algorithm performs an initial search by expanding each state once (ϵ_0), while keeping track of already expanded nodes that have become inconsistent due to changes in their neighbors, inserting these inconsistent nodes into an INCONS list. If there are no world changes to create inconsistent nodes, decrease ϵ iteratively, improving path quality until an optimal solution, $\epsilon = 1$, is reached (Kallman & Kapadia, 2014, p.8).

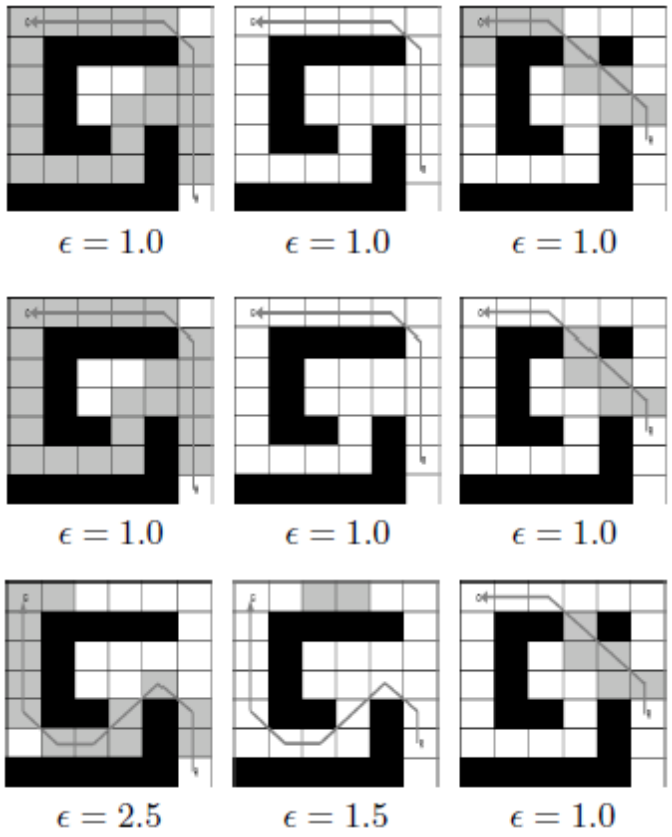

Figure 7: [Top-to-bottom] *Example of A*, D* Lite, and AD* in robotic pathfinding through a dynamic environment. (Likhachev et al. 2005, p.5-6)

Figure 7 shows all three of the algorithms discussed so far, pathfinding through a dynamically-changing environment. Black cells are obstacles and white cells are free space. The algorithm starts at the bottom-right node, and works its way towards the goal node in the top-left with each iteration. The cost of moving cell to cell in this example is one. The cells expanded by each algorithm for each step are shown in grey, and the resulting paths are shown as a dark grey line (Likhachev et al. 2005, p.5).

As shown in this example, each algorithm reacts differently to this dynamic environment. A* searches down both corridors to the goal and decides upon the most optimal path. After two steps, as a gap in the wall is discovered, it must stop and replan from scratch this new optimal path to the goal. In total, A* had to expand upon 31 cells to reach its optimal solution (Likhachev et al. 2005, p.5).

D* Lite acts much like A* does in its initial search, finding the optimal path and proceeding in two steps. Unlike A*, when the gap in the wall is discovered, D* Lite reuses its previous search information. Thus, the algorithm only must calculate the path through the gap in the wall to see that it is now the optimal path. In total, D* Lite only had to expand upon 27 cells, 15% less expansions than A*.

Anytime Dynamic A* starts with an initially suboptimal solution of $\epsilon = 2.5$, and 15 cell expansions. In its next step, the algorithm refines its solution by reducing down to $\epsilon = 1.5$. After another step, when the agent discovers the gap in the wall, the algorithm must only expand upon those cells that are directly affected by this new information, the 5 between it and the goal (Likhachev et al. 2005, p.5-6). In total, Anytime Dynamic A* only had to expand upon 20 cells to reach its optimal solution, far less than D* Lite and A*. Because AD* reuses previous solutions, and is able to repair invalidated solutions, it

is able to provide anytime solutions in dynamic environments very efficiently (Likhachev et al. 2005, p.6).

Navigation Meshes and Software Implementations

Pathfinding environments have evolved over the decades in computing. With faster hardware and graphical processing capabilities, larger and more complex environments have been created virtually to test pathfinding algorithms. With advanced sensor capabilities and advances in robotics, the real world has also opened up to the possibility of autonomous drones in need of pathfinding through ever-changing environments. A navigation mesh must represent the free environment efficiently so that pathfinding within them is optimal. There are several basic properties a navigation mesh must adhere to. A navigation mesh should represent the environment in $O(n)$ number of cells or nodes to allow search algorithms to operate efficiently, facilitate the computation of quality paths, provide an efficient mechanism for computing paths with arbitrary clearance from obstacles, be robust in order to allow for unpredictable dynamic updates, and should efficiently update itself when there are changes to the environment (Kallman & Kapadia, 2014, p.5).

These are most crucial in virtual world simulations, and are best seen in the software implementations of pathfinding. Many of the software implementations for pathfinding are paid licenses. One software implementation known as PathEngine does have a free demo package, and will be discussed in this paper as well as utilized during the presentation.

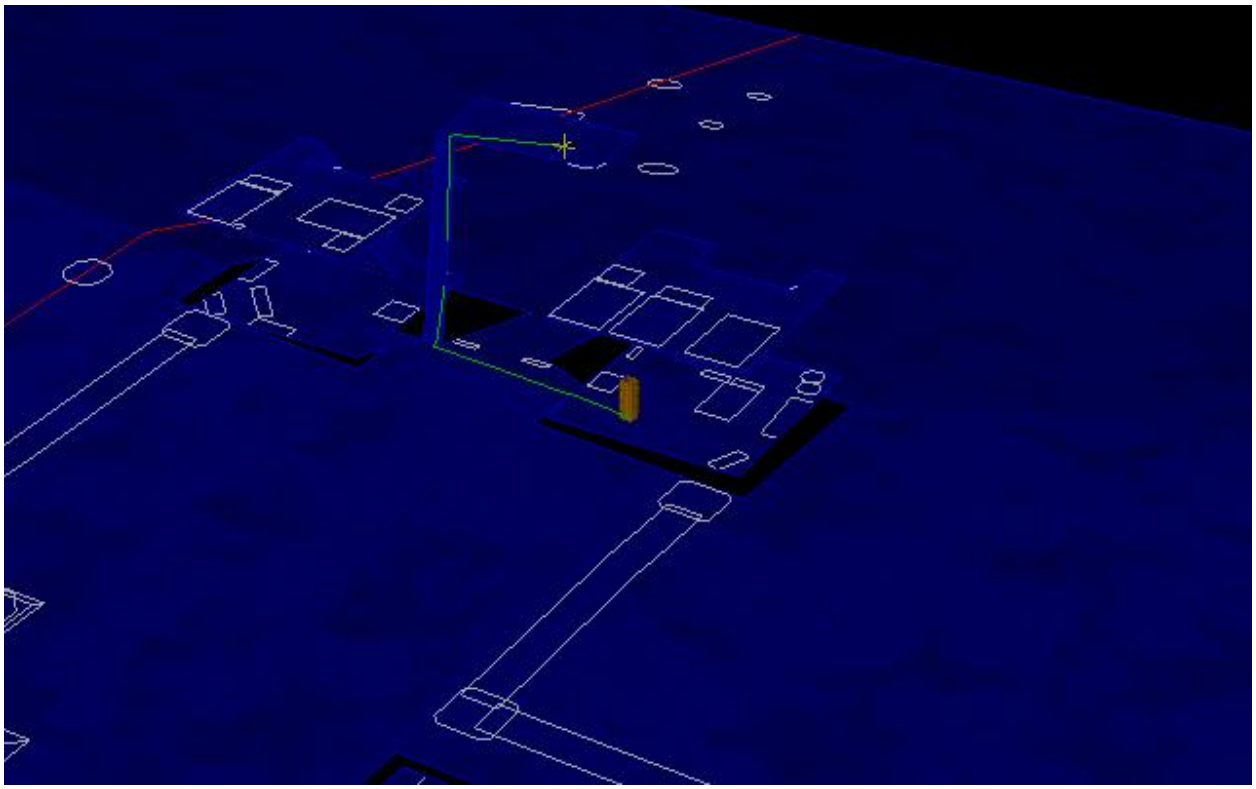

Figure 8: *PathEngine Demo testbed (MeshFederation) in a 3-Dimensional Navigation Mesh (PathEngine, 2014. The 3D orange object is the pathfinding agent, the line extending from it is its path, and the small axis is the path's endpoint. Blue lines are the bounds of the environment, and white-lined outlines are obstacles within the environment.*

As seen in Figure 8, the PathEngine demo set navigates through three-dimensional navigation meshes. PathEngine's primary agent movement is ground-based, built around an "advanced implementation of points-of-visibility pathfinding on 3D ground surfaces" (PathEngine). At the core of this lies a "well-defined agent 'movement model'" (PathEngine); this movement model defines how agents and obstacles interact in the environment, allowing for robust movement by taking into consideration how obstacles and surface edges constrain agent movement (PathEngine). Agent height is also able to be taken into consideration, allowing for pathfinding through complex environments with varying clearances. PathEngine is used in a number of current video games for pathing solutions, such as 'Shadow of Mordor',

‘The Witcher 2’, and ‘Stronghold Crusader 2’, to name a few (PathEngine). These games often have many agents moving around large spaces in real-time, which PathEngine must be able to handle efficiently to avoid slowing frame rates, and optimally in order to provide reliable paths for these agents.

Conclusion

The shortest path problem has been a challenge for the many algorithms. It is a complex problem that has grown over time with the evolution of robotics and computer graphical environments. Of the many algorithms, A* and its variants D* and AD* have shown how solutions have expanded over the years to match the demands of this ever-growing problem. As personal and commercial UAVs become more popular (Amazon Prime Air, for example), and demands for virtual world pathfinding grow with increasing complexity in environments and demand for greater numbers of agents moving around these environments, the demands for faster and more efficient pathfinding algorithms is certain to continue. Further development will be needed as future challenges in pathfinding appear, with further expansions on these algorithms needed to solve them.

Works Cited

Alexander Schrijver. (2010). *On The History Of The Shortest Path Problem.* Retrieved from http://www.math.uiuc.edu/documenta/vol-ismp/32_schrijver-alexander-sp.pdf

Amazon Prime Air. Retrieved from http://www.amazon.com/b?node=8037720011

E. W. Dijkstra. (1959). *A Note on Two Problems in Connexion with Graph*. Retrieved from http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf

Marcelo Kallmann, Mubbasir Kapadia. (2014). *Navigation meshes and real-time dynamic planning for virtual worlds*. Retrieved from https://dl.acm.org/citation.cfm?id=2614028.2615399&coll=DL&dl=ACM&CFID=495311590&CFTOKEN=37719856

Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, Sebastian Thrun. (2005). *Anytime Dynamic A*: An Anytime, Replanning Algorithm*. Retrieved from https://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf

PathEngine. (2014). *PathEngine: Intelligent Agent Movement*. Retrieved from http://www.pathengine.com/

PathEngine Demos. (2014). *PathEngine_SDKBase_05_35.zip*. Retrieved from http://www.pathengine.com/index