Graph representations of 3D spaces

A comparison of three approaches

Colin Snow

May 2021

1 Overview

Graph-based path planning algorithms are fast, efficient, and versatile, but they can only ever be as good as the graph they are given. Since most physical things do not lend themselves well to a literal interpretation as a graph, an algorithm is needed to transform a spatial representation into a graphical one. Given the importance of this problem, many such algorithms have been developed, and each has features and drawbacks that make them useful in different scenarios.

While there is a relatively diverse set of algorithms for this purpose, all of them rely on a relatively similar structure. The main decisions to be made are where to add nodes and how to connect the nodes together. This may seem simple, but decisions about these factors have a huge effect on performance.

In this paper I will present three such algorithms ranging from relatively trivial to quite complicated and compare the relative strengths and weaknesses of each when used for finding the shortest path between two positions.

2 Model building and Assumptions

For the sake of simplicity we will make some assumptions about our world model. First, we assume that travel speed is unweighted. This means that the time to travel some set distance is the same everywhere in the space. Second, we assume that the space consists of a finite number of 3D polygonal obstacles (no curves). This provides a set of vertices on each obstacle which can be used to define its bounds. Any obstacle that is not polygonal could simply be fit with an enclosing polygon to theoretically infinite resolution though often a rectangular box provides adequate results.

All path planning will be done with A* using a euclidean distance heuristic. This heuristic is admissible because of our first assumption, so all paths found will be optimal for the given representation. However, this does not necessarily mean that they are optimal paths in the space as the graph representation may not perfectly represent the space.

All spaces are assumed to be 3D, but all algorithms presented should work in any number of dimensions. Some diagrams are shown in 2D for clarity.

3 Algorithms

3.1 Grid-Based

One of the simplest and most straightforward methods of graph creation is to create a grid of points that fill the space. An appropriate grid spacing must be chosen which represents fine detail accurately enough without making the total graph size too large.

3.1.1 Algorithm

A grid of nodes is created which covers an x,y,z space. Each node has an associated list of neighbours which contains all nodes directly adjacent or diagonal to it (26 per node). These neighbours represent the edges of the resulting graph. Any node that is placed within an obstacle is then marked blocked and can not be used in path planning. Once the graph is made an optimal path can be found between any two nodes as long as one exists.

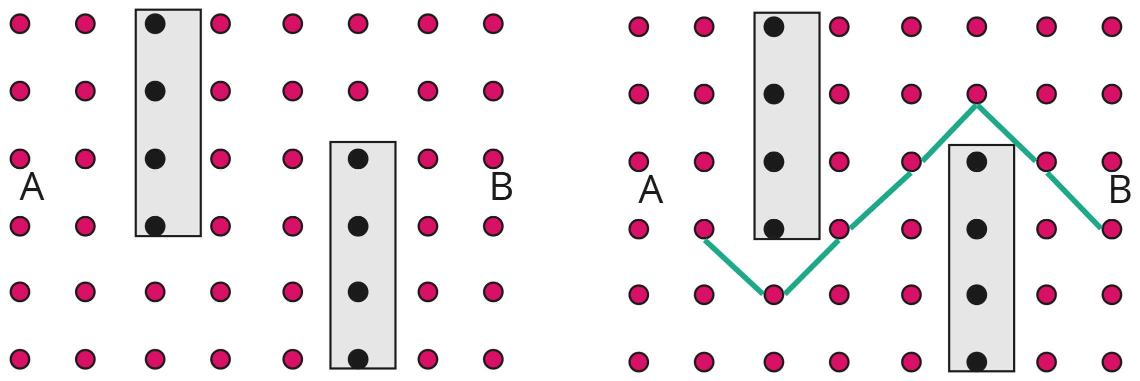

Figure 1: Left: A grid of nodes with those that intersect an obstacle marked as blocked. Right: A path from A to B in this graph.

3.1.2 Pseudocode

```
def grid_creator(space_size, cell_size):

    nodes = create grid of nodes that fill space_size with resolution cell_size

    for each node n:
        if n is in obstacle:
            mark n as blocked
        set neighbours of n to all cells adjacent or diagonal to n

    return nodes
```

3.1.3 Runtime

Since the algorithm creates nodes evenly within the given space, generating the graph is $O(n^3)$ in the size of the input space in both space and time.

In the worst case, A* visits every node of a graph so the upper bound on path planning is also cubic with respect to the input space. In practice this is rarely true and it is often more determined by the path length and the complexity of obstacles.

3.1.4 Advantages

- Since it requires almost no information about the underlying space besides checking if an obstacle exists at a point it is very adaptable to new situations. This method could easily be applied to more complicated mesh models.
- The implementation is incredibly easy and each graph point can be created very quickly as neighbours can be calculated with simple geometry.

3.1.5 Disadvantages

- The size of the graph grows rapidly with the size of the input space. For models which have fine details in some regions or large regions of open space the graph will store large amounts of extraneous information.
- Movement is limited to either Cartesian or diagonal paths. This rarely provides an optimal path though many smoothing algorithms exist to mitigate this issue.
- In spaces with fine details it is possible for a grid to miss small paths or shortcuts which no node happens to land within.

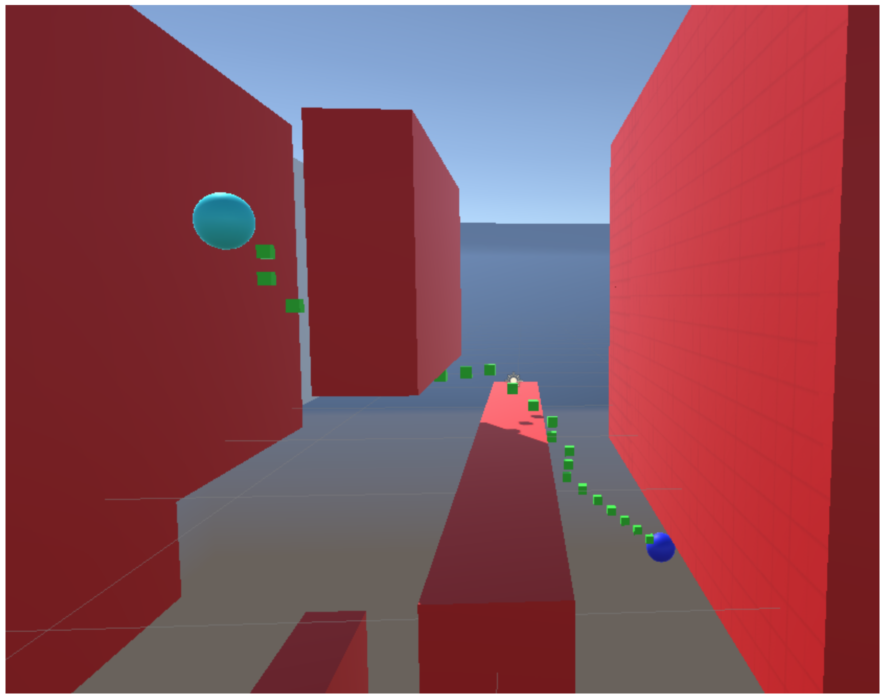

Figure 2: A grid representation of a space with obstacles in red and the optimal path in green.

3.2 Visibility Graph

We saw that the main disadvantage of the grid method was its adverse scaling effects due to a very naive understanding of the underlying space. Therefore, we want to find a way to more intelligently choose nodes and edges to minimize extraneous information.

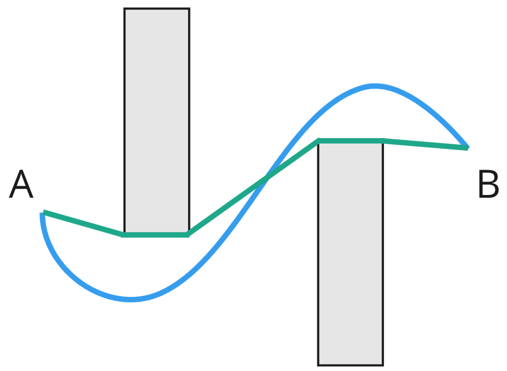

Figure 3: Two possible paths that can be taken from A to B. The blue graph does not touch any obstacles while the green graph always hugs the inside edge.

Suppose you are in a space like the one presented in figure 3 and are trying to get from A to B. While both the blue and green paths are valid, the green is much shorter and in this case is actually the optimal path. When we observe what is unique about this green path we see that it is always tight to the corners of the obstacles. In fact, this behavior is not coincidental but a necessary quality of any optimal path in this type of space. The optimal path will always be either a straight line or series of straight lines which all start or end at an obstacle.

To understand why, let's go back to the blue line. We know that the shortest distance between two Cartesian points is a straight line, so it seems intuitive that we should be able to shorten this path by straightening parts of it. Imagine that we pick two points on the blue path and connect them with a line. If this line does not hit an obstacle then it must be shorter than the section of the path it bypasses. If we then continue this process we will eventually end up with the green line which can no longer be shortened in this way. The simplest intuitive way to think about this is to imagine taking a string and laying it along the path then pulling it tight. This tightened path will have all straight sections and will touch obstacles at the end of each section.

Let's take a step back and realize what this means for our graph. If the optimal path must only contain straight lines and must only change direction at obstacles, the only places that need nodes are the outlines of those obstacles. Any other position in the space can not be a bend in the optimal path. In 2D this is simply the corners of the obstacles, while in 3D it also includes the edges between corners. Since these edges can not be represented by a finite set of points, for all but the largest obstacles the corners work just fine in 3D as well.

Now that we know where to put the nodes, we need to figure out how to connect them with edges. The visibility graph model provides a very simple way of doing this. For every node, its neighbours are the set of all nodes which have an unobstructed path from the current node. This assures that at each bend in the path any node which could possibly be the next connection is a neighbour of the current node.

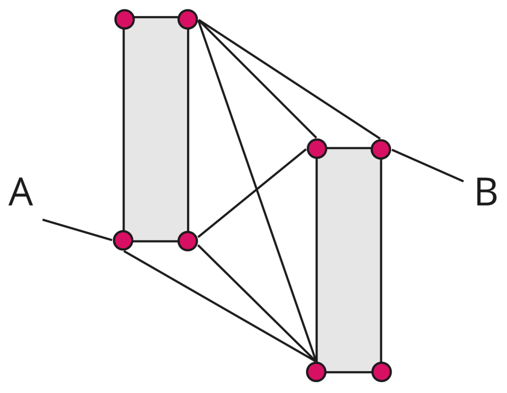

Figure 4: A visibility graph representation. Each node is connected to all other nodes which are "visible" from the current node.

3.2.1 Algorithm

Create a set of nodes which are positioned at the outer vertices of each object. The number of points to use for each object should be determined by application. For each node, add every node which has a clear path from the current node as a neighbour. A* can again be directly applied to the graph to find the optimal path between two nodes. If the nodes represent all of the external features of the objects (such as the 2D case where all corners are nodes) then this path will be the true optimal path.

3.2.2 Pseudocode

```
def visibility_graph(objects):

    points = empty list
    for each object:
        add a point to points for each vertex of the object

    for each point:
        for each other point:
            if a clear path exists between them:
                add other point to neighbours of point

    return points
```

3.2.3 Runtime

Since the algorithm only depends on the objects and not their positions, it is not bounded by the size of the space. Instead, it is bounded by the number of obstacles. Each point can have as many neighbours as there are other points, so the size of the graph grows quadratically with the size of the input. Te runtime is dependent on the algorithm used, with the naive method (implemented here) running in $O(n^3)$ and faster algorithms like Lee's Visibility Graph Algorithm running in $O(n^2 log(n))$.

3.2.4 Advantages

- If all edges of the obstacles are covered by nodes the path returned will be the true optimal. While this is not feasible for $>$2 dimensions the algorithm usually still produces very good results.
- Since all possible connections are represented it is possible to make large jumps in space if conditions are favorable, dramatically reducing the time to calculate a path.
- The size of the graph does not scale with the size of the space, and both large and small details can be described with adequate detail.

3.2.5 Disadvantages

- The number of edges grows rapidly with the number of obstacles. This makes the size of the graph in memory grow rapidly as well. Also, since A* checks all neighbours of a node in each iteration, nodes with large numbers of neighbours take a long time to compute.
- It is less adaptable to complex geometry than the grid method but as long as a suitable enclosing polygon can be found it should work well.

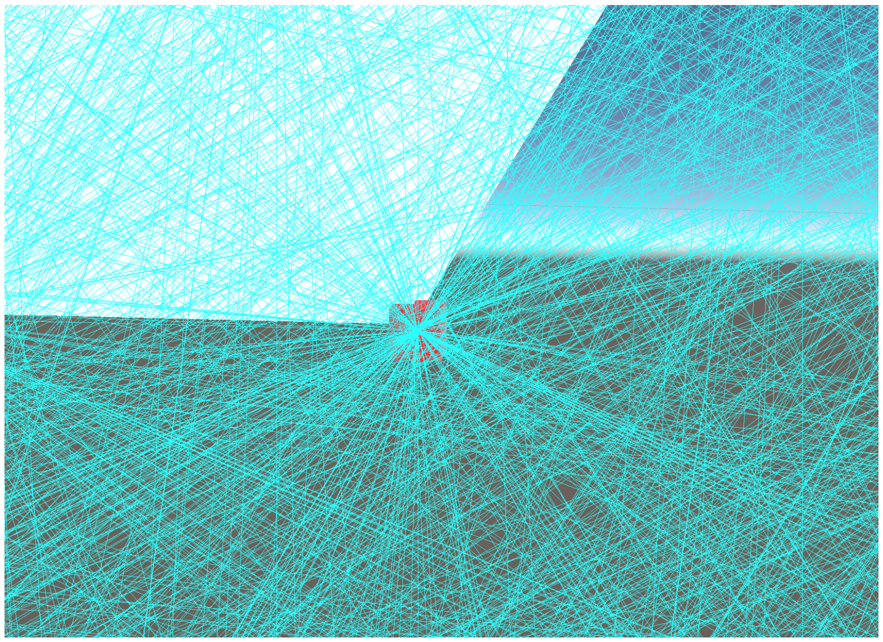

Figure 5: A zoomed in view of a single node of a visibility graph with about 50 objects. Each blue line represents an edge and there are so many that it is difficult to see through them. This graph contains over 93000 edges.

3.3 Navigation Mesh

The visibility graph solves many of the scaling problems of the grid approach, but still scales quite dramatically if the number of obstacles is large. The idea of creating a navigation mesh follows directly from this issue. We want to reduce the number of edges needed to define the graph while losing as little information as possible. To do this, we must reduce the number of edges that each node has while still keeping the graph connected and maintaining as many advantageous links as possible. The visibility graph already provides a set of good nodes so we will use the same ones for the mesh.

Meshing is the process of connecting vertices to form polygons. It most commonly used to create convex meshes from a 3D model, but can also be used for a variety of other tasks. In this case, we will use meshing to more intelligently choose which vertices to use in our graph.

We will use the Bowyer-Watson algorithm to find a Delaunay triangulation of the space and then use this information to create the graph.

A Delaunay triangulation is a set of triangles (in 2D or tetrahedrons in 3D) which connect points from some set P such that no point in p is within the circumcircle (circumsphere in 3D) that passes through the points of the triangle. This triangulation has several advantageous properties. First, because no vertex can be in any other polygon (triangle, tetrahedron) and the polygons share edges and vertices, the set of polygons perfectly fills the space bounded by the outer most points with no overlaps. This means that there is a path from every node to every other node but that many of the extraneous paths are removed. Second, the Delaunay triangulation maximizes the minimum angle of all triangles in the mesh. This means it tends to avoid ”sliver” triangles in favor of more even shapes. Preventing these ”sliver” triangles means that less polygons are needed and that paths tend not to have to divert to follow the long end of a thin triangle.

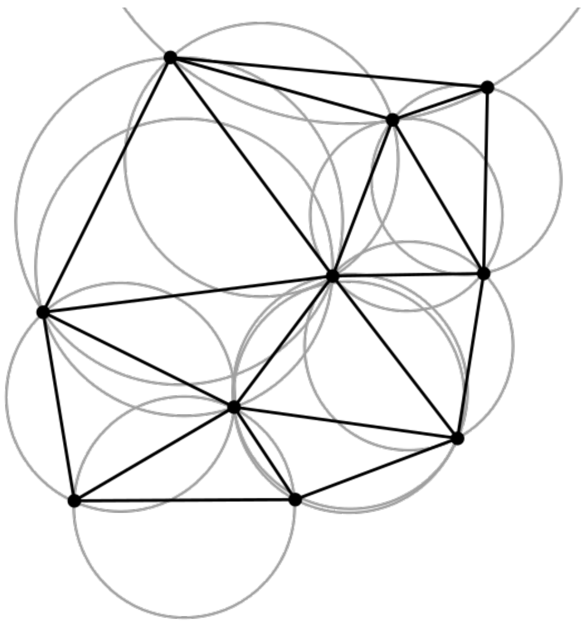

Figure 6: A Delaunay triangulation of points with their circumcircles. No node is within a circumcircle. From Wikipedia.

3.3.1 Algorithm

The Bowyer-Watson algorithm can be used to efficiently find the Delauny triangulation of a set of points. It works in any number of dimensions, but the three dimensional case will be described as that is what was implemented. First, the points are completely surround by a tetrahedron. The size and shape of this are not important as long as all points lie inside it. An empty array of tetrahedra is created. In each iteration of the algorithm, a point is chosen from the list of points and is added to the space. Any tetrahedra whose circumsphere encloses the new point is deleted. Next, all faces that made up the deleted tetrahedra are combined and any face which appears more than once is deleted. Note that this means that if a face appears twice both instances are deleted not just one. The remaining faces create a polygonal hole in the mesh with the new point somewhere in the middle. Next, tetrahedra are created by adding the new point to each of the remaining faces, thus filling the hole exactly. None of these tetrahedra can circumscribe the new point as they all have it as one of their vertices and they can not circumscribe any other point as the process of redefining them makes their circumcircles smaller and those circles already did not contain any other point. Thus at each step of the iteration the rules of Delaunay triangles are met. When all nodes are incorporated the triangulation is complete.

Creating a graph from the triangulation is then quite trivial. The neighbours of a node are just the set of all nodes which share a tetrahedron with the current node. This creates a graph with many of the same properties as the visibility graph, but with no intersecting edges and significantly fewer connections. Finally, all edges which intersect an obstacle are deleted which prevents any motion within obstacles.

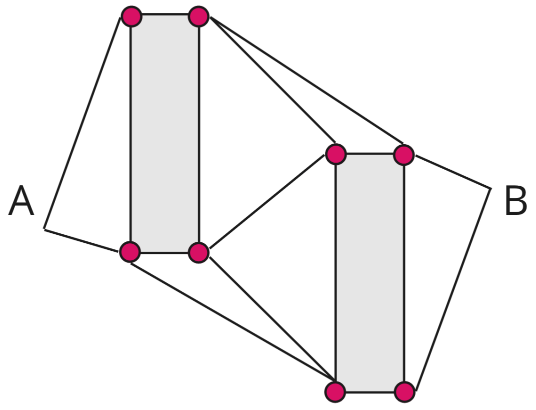

Figure 7: One possible triaingulation of a set of points with edges that intersect obstacles removed.

3.3.2 Pseudocode

```
def bowyer_watson(points):

    tetrahedra = empty list
    bad_tetrahedra = empty list

    enclosing_tet = create tetrahedron which encloses all points

    add enclosing_tet to tetrahedra

    for each point in points:

        for each tetrahedron in tetrahedra:
            if the tetrahedron circumsphere contains point:
                add it to bad tetrahedra

        faces = empty list

        for each tetrahedron in bad_tetrahedra:
            tet_faces = faces of tetrahedron
            add tet_faces to faces

        new_faces = remove any face which appears more than once in faces

        for each face in new_faces:
            new_tet = create new tetrahedron from face and point
            add new_tet to tetrahedra

        remove all items in bad_tetrahedra from tetrahedra

    for each tet in tetrahedra:
        remove tet if it contains a point from enclosing_tet

    return tetrahedra
```

3.3.3 Runtime

The theoretical runtime of Bowyer-Watson is $O(n^5/3)$ for n points in 3D, but I have found that the order in which points are added has a significant effect on efficiency. In my experimentation ordering points by x value (or any other linear sorting) before generating the mesh could speed it up by well more than 10 times. This is because the runtime is highly dependent on the number of tetrahedra it needs to change in each step and adding the points randomly tends to create a worst-case scenario for this as it tends to put points in regions that have many overlaps.

The triangulation contains $O(n^3/2)$ tetrahedra in 3D, a much more reasonable scaling factor compared to the visibility graph.

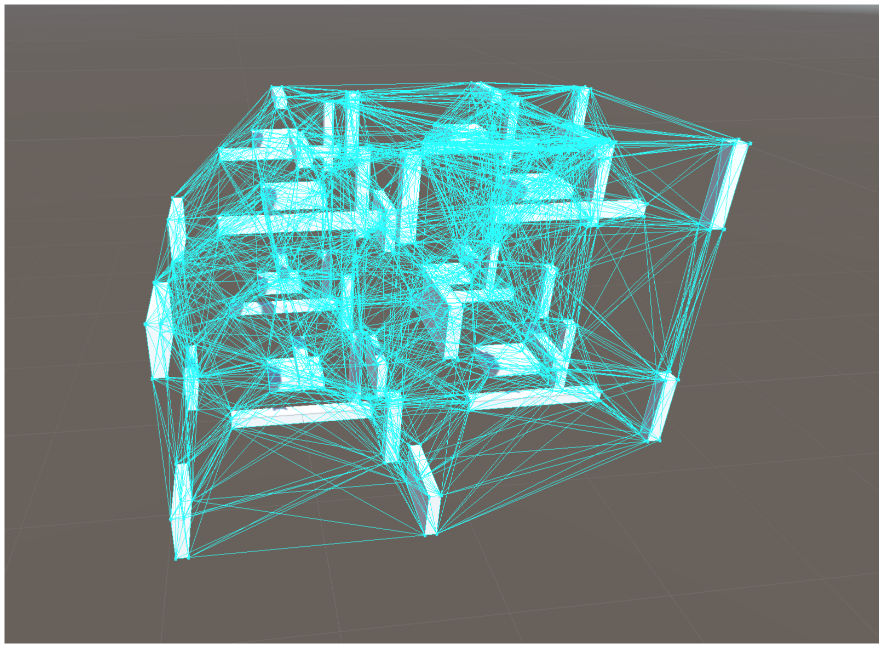

Figure 8: A triangulation of a space with 50 obstacles. Despite having over 300 vertices the number of edges is quite manageable.

3.3.4 Advantages

- It has a much more reasonable space complexity than the grid or visibility graph.

- It relies less on checking for collisions than the visibility graph which can be advantageous in cases where this is expensive.

3.3.5 Disadvantages

- While the complexity of finding the triangulation does not grow incredibly fast for larger graphs, the operation is still quite expensive as finding the circumsphere requires taking determinants and other costly matrix operations. In practice this often makes it slower to generate than the visibility graph for reasonable space sizes.
- As it contains less information than the visibility graph, paths tend to be initially less optimal. However, by employing an algorithm like the one discussed in figure 3 the path can be made just as good as the one from the visibility graph.

4 Choosing a method

Now that we have described these three methods, the obvious question is which one should be used? The answer, unfortunately, is that it depends on the particular application. If you want to write something quickly and your space is small then a grid approach might be a great option. If you need to create very optimal paths in a small graphs with varying complexity then a visibility graph would work great. And if you want to create large graphs without the scaling problems of the other two and have plenty of free time to do implementation then the navmesh is the way to go. Instead of going into excruciating detail about each one I created a chart which compares some of the main points of difference between them.

Factor	Grid	Visibility	Mesh
Implementation Difficulty	Easy	Medium	Hard
Space complexity in size of space	Bad	No Effect	No Effect
Space complexity in number of objects	No Effect	Bad	Ok
Time complexity in size of space	Bad	No Effect	No Effect
Time complexity in number of objects	No Effect	Medium	Ok
Path optimality without processing	Ok	Excellent	Good

5 Conclusion

There are many algorithms available to create graphs from 3D spaces and each comes with its advantages and disadvantages. Choosing an appropriate representation for your application is critical if you want to achieve the path quality and runtime you desire. These are just three possible algorithms and many more exist to solve particular problems, so it is always important to understand whether any particular one is a good fit.

6 Code

All code for this project can be found here.
To run it you will need to have Unity installed and create a scene from the instructions in the link.

References

[1] *A Fast Global Flight Path Planning Algorithm Based on Space Circumscription and Sparse Visibility Graph for Unmanned Aerial Vehicle.* July 2018. URL: https://www.mdpi.com/2079-9292/7/12/375/htm.

[2] *Bowyer Watson Algorithm*. Apr. 2021. URL: https://en.wikipedia.org/wiki/Bowyer%5C%E2%5C%80%5C%93Watson_algorithm.

[3] *Lecture, Game AI, Dr. Stephen Lee-Urban*. Oct. 2019. URL: https://www.cc.gatech.edu/~surban6/2019fa-gameAI/lectures/2019_08_28_searchMovement_continued2-presented%5C%20in%5C%20class.pdf.

[4] *Lee's O(n2 log n) Visibility Graph Algorithm Implementation and Analysis*. May 2012. URL: https://dav.ee/papers/Visibility_Graph_Algorithm.pdf.

[5] *Map Representations, Amit's Thoughts on Pathfinding*. Apr. 2021. URL: https://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html.