# From Node to Node: A Visualization of Pathfinding Algorithms

Jarett M. Sutula
Faculty Advisor: Brian Gormanly
*School of Computer Science and Mathematics*
*Marist College*
May 19th, 2021
Poughkeepsie, New York, USA
Jarett.Sutula1@marist.edu
Brian.Gormanly@marist.edu

***Abstract*—Built off of the foundation of graph traversal, pathfinding has evolved from basic game artificial intelligence to uses in the development of GPS and robotics [1]. With an ever-focused push for efficiency, pathfinding has seen many advancements and discoveries that have helped scientists and developers approach traversal more effectively. The aim of this paper is to provide a small background to the concepts of pathfinding and an understanding of the various pathfinding algorithms. These algorithms will be implemented in a grid-based app that will allow its users to visualize the differences between algorithms and how they work. The app will show step-by-step how each algorithm tackles a grid drawn up by the user, and this paper will serve as a summary of research done on the algorithms.**

***Index Terms*—Path planning, Shortest path problem, Graph theory**

## I. Graph Theory and Traversal

A graph is a structure made up of a set of vertices, or nodes, that are connected to other nodes by edges [5]. Graphs have been an interesting target of research due to their directions, connections, orientations, and weights. These are all important in the study of graph traversal, which holds the origins of pathfinding and its algorithms.

Graph traversal is a common programming problem that aims to find the most efficient way to traverse a series of nodes and edges. Some algorithms aim to find multiple paths between nodes, some aim to find the shortest possible path between nodes, and some aim to find the lowest cost between nodes. These approaches have found their way to pathfinding, where researchers and programmers are considering the optimal path between two nodes, the intricacies of the obstacles between two nodes, and the minimization of the cost of traversal.

Pathfinding refers to the computer science problem of finding the shortest path between a starting node and an ending node. Our approach to pathfinding for our app takes a more grid-based approach to graph traversal, where neighboring nodes become neighboring units on a grid. Two nodes are selected - one represents the starting point, often referred to as the root, and the other node represents the destination. In all forms of pathfinding algorithms, nodes on the grid are marked either visited or unvisited. Every time an algorithm expands its search, the nodes it reaches are switched from unvisited to visited, which allows it to continue searching without wasting time and cost by revisiting nodes it has already been to. This approach is clear in both graph-based traversal and grid-based traversal.

## II. Breadth First Search

Breadth First Search, or BFS, acts as a pathfinding algorithm but traces its roots to graph traversal. The basic concept of BFS is that it starts at a root node and searches its immediate neighbors, looking for the desired destination node [6]. If it does not find its destination node, it will go to each of the neighboring nodes and search their neighbors. This continues until it reaches the destination node. BFS was published in the late 1950s to find the shortest path to the exit of a maze. By exploring in all directions, it guarantees that it will eventually find the shortest path.

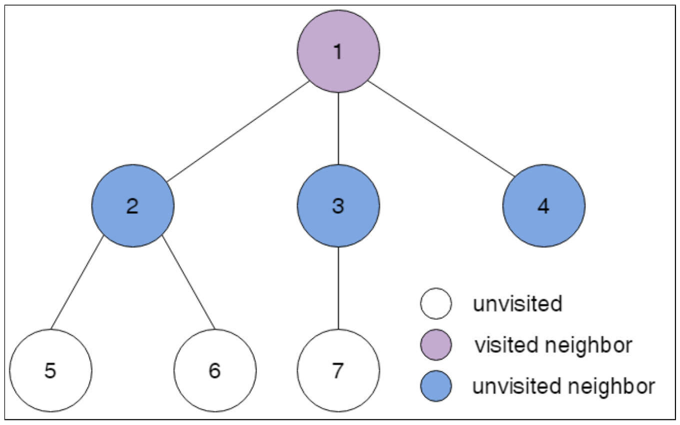


Fig. 1. Step 1 of BFS graph traversal

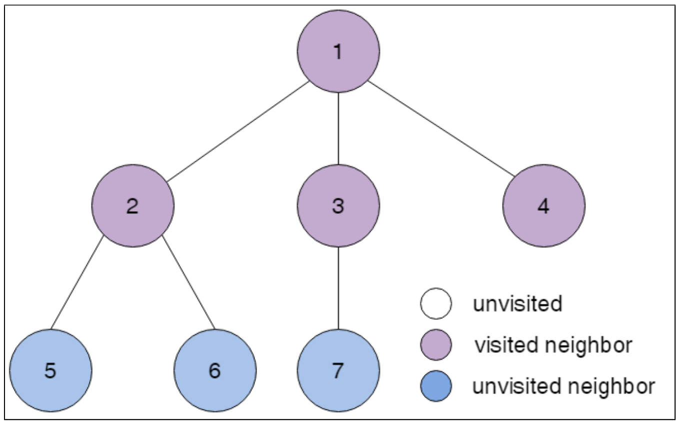


Fig. 2. Step 2 of BFS graph traversal

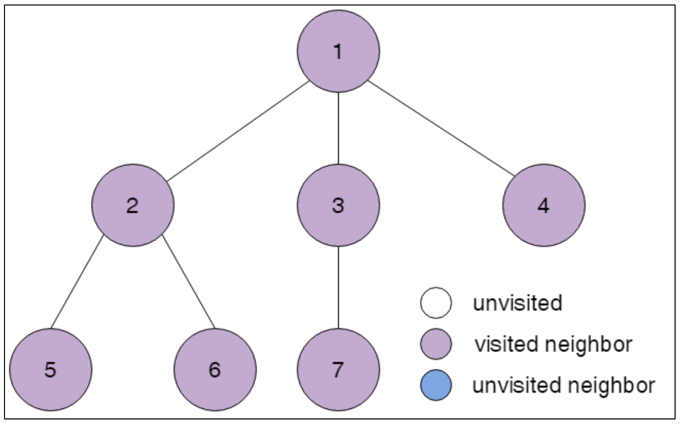


Fig. 3. Step 3 of BFS graph traversal

BFS finds extensive usage in procedural map generation, which is a staple in modern-day video games. BFS guarantees that it will find the shortest path between two nodes, but the nature of its movement means that it can be inefficient if the two nodes are very far apart. Despite not being very cost-effective, BFS does have some practical cases in road networking, computer networking, and particular pathfinding problems that deal with maze generation or looking for nearby places of interest. Since BFS is simply moving in all directions, computation is simple which may prove to be an advantage over other algorithms.

BFS works similarly when traversing through a grid-based graph. The start node's neighbors are searched in all 4 directions - North, South, East, and West. Each 'turn' a new neighbor is searched and its neighbors are added to the queue.

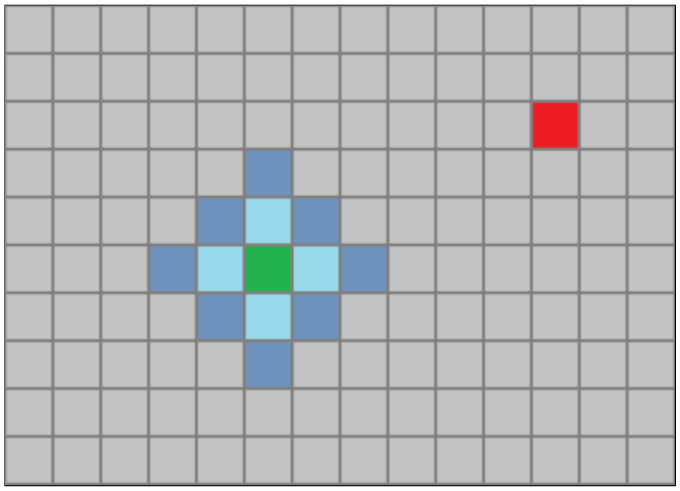


Fig. 4. An example of BFS working in a grid. The green node is the start node, the red node is the destination node, the light blue nodes are visited neighbors, and the dark blue nodes are unvisited neighbors waiting in the queue.

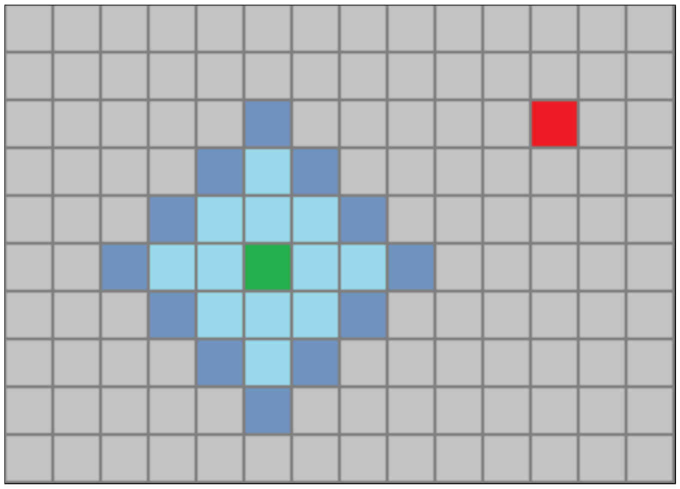


Fig. 5. The next step in BFS working in a grid. The previous unvisited neighbors have been visited, and their neighbors have been added to the queue.

## III. Depth First Search

Unlike BFS, which uses a queue to follow the First In First Out (FIFO) approach, Depth First Search (DFS) is the polar opposite. DFS uses a stack to follow the Last in First Out (LIFO) approach. DFS sees use in graph traversal, just like BFS, but it sees less use in pathfinding.

DFS is often referred to as the backtracking algorithm. Unlike BFS which explores all of the neighbors of a node, DFS takes a tree or a graph and explores it as far as it can down a path before turning around and backtracking. Applications of DFS include maze creation and traversing trees.

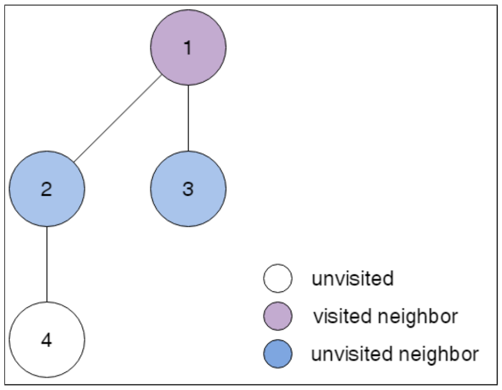


Fig. 6. Step 1 of DFS graph traversal

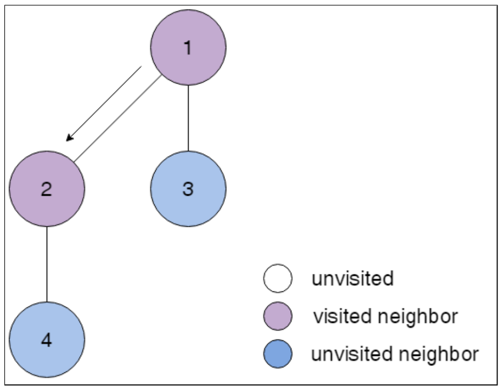


Fig. 7. Step 2 of DFS graph traversal

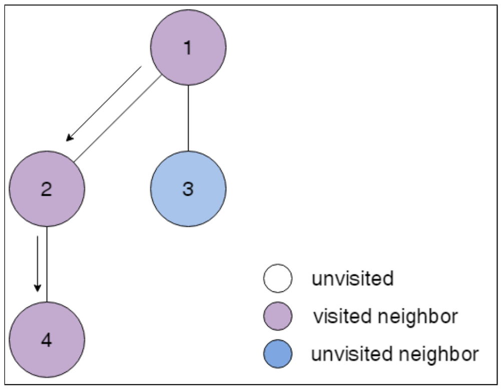


Fig. 8. Step 3 of DFS graph traversal

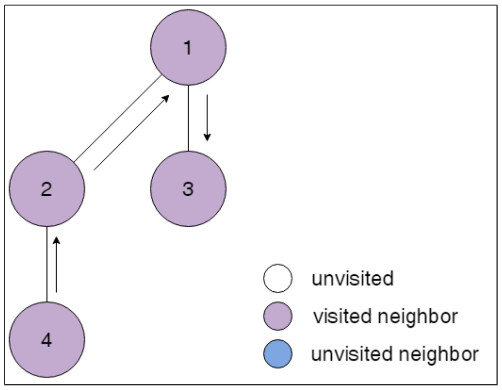


Fig. 9. Step 4 of DFS graph traversal

In the case of grid-based pathfinding, however, DFS struggles because unlike BFS it cannot guarantee the shortest path. Due to the exhaustive nature of BFS, DFS typically uses less memory than BFS, but it is less optimal for finding the shortest path. Both algorithms come with their practical uses, but BFS wins out in pathfinding. DFS is simply a poor choice for any simple pathfinding problem. It is still important to recognize the algorithm, however, as it has its own practical uses outside of simple pathfinding.

In a grid-based approach that is not utilizing diagonals, there are only four possible directions to head to: North, East, South, and West. Instead of grabbing the first node in the queue as BFS does, DFS grabs the last node in the queue, which will be predetermined by the order of directions of DFS. In our case, we stuck with the basic priority order of North $>$ East $>$ South $>$ West. That means if the current node has a North neighbor, we move there. If they do not have a North neighbor, we try their East neighbor, and so on.

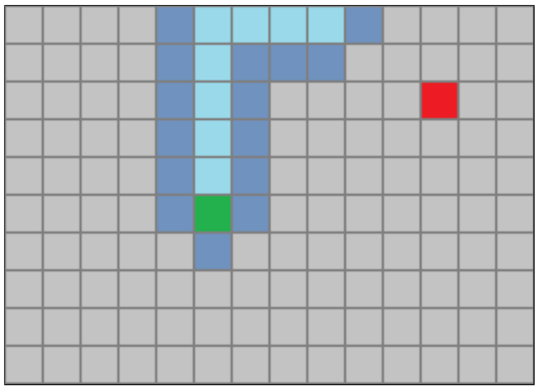


Fig. 10. An example of DFS changing directions once it can no longer go in the first direction (in this case, with no more North neighbors, it moves East).

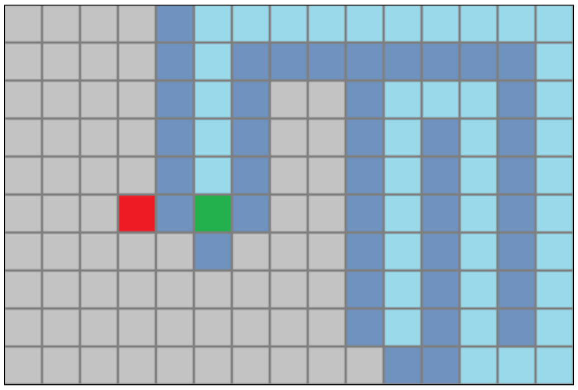


Fig. 11. An example of DFS working in a grid. The green node is the start node, the red node is the destination node, the light blue nodes are visited neighbors, and the dark blue nodes are unvisited neighbors waiting in the queue. DFS will search in one direction until it cannot go any further, then it will try a different direction.

Figure 11 shows why DFS possesses flaws that make it less than effective for simple pathfinding. If the destination node started just 2 or 3 units left of the start node, DFS could be searching the entirety of the grid looking for it simply because the direction order is predetermined Since, in this example, West is the last direction to be checked, it will always prioritize free nodes that are North, East, or South of it before it begins to check for anything to the West. This isn't very optimal for the majority of simple pathfinding. It is, however, great for solving mazes - it follows the same concept in maze theory that if you hold your hand against the wall and keep moving, you will eventually find the end.

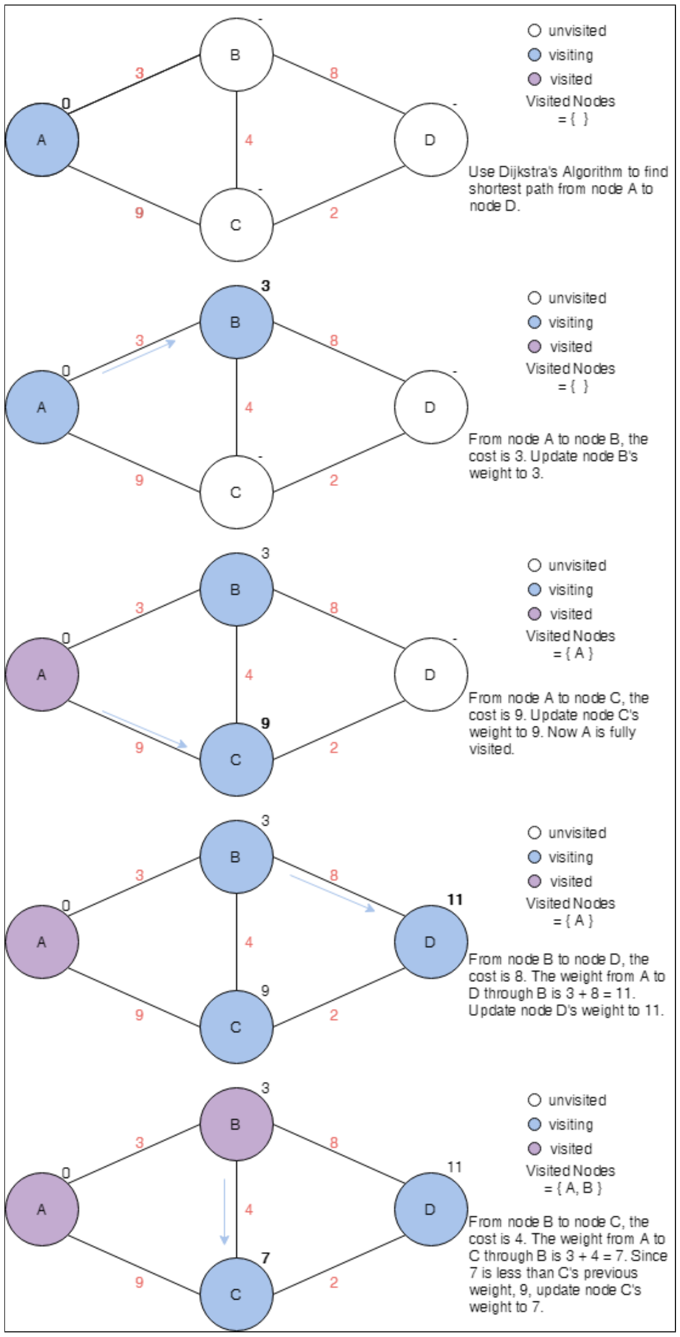
## IV. Dijkstra's Algorithm

Dijkstra's algorithm is interesting in that it offers a different, "cost-effective" approach. When using Dijkstra's, we care less about every path that we come across and we care more about the "cost" of the path.

Dijkstra's, like BFS, guarantees the shortest path between two nodes, so it has plenty of uses in pathfinding. Dijkstra's ability to create cost-effective pathing is useful and has practical uses in GPS/road networking.

Visually, Dijkstra's algorithm works a lot like BFS where we are branching out in multiple directions. In a weighted node graph, where each edge between nodes has a certain cost to it, Dijkstra's algorithm visits all of the current node's neighbors, keeping track of every visited node's distance from the start node and generating a cost from this. It keeps track of the shortest distance from the starting node and the previous node along the shortest path. The distance of the node's neighbors will be compared to the other options, and we are left with an algorithm that guarantees the shortest distance. In a weighted node graph, Dijkstra's algorithm helps us find the shortest path between multiple nodes, which can prove very useful and more effective for complicated pathfinding problems with multiple nodes.

From a grid-based approach, Dijkstra's Algorithm is effectively the same as BFS, since on a grid every cost between nodes is 1. Since the weight of all nodes are the same, Dijkstra's algorithm does not have an advantage over BFS in an equal-cost grid. This is reflected in the app, and the overall approach of Dijkstra's and BFS appear the same. While the grid-based approach may not highlight the effectiveness of Dijkstra's, it does give us a better sight step-by-step of how our various pathfinding algorithms approach simple node A to node B pathfinding.



Use Dijkstra's Algorithm to find shortest path from node A to node D.

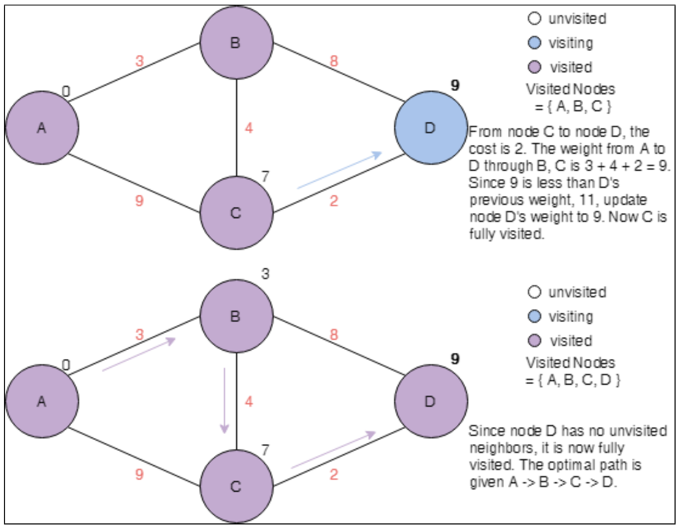
From node A to node B, the cost is 3. Update node B's weight to 3.

From node A to node C, the cost is 9. Update node C's weight to 9. Now A is fully visited.

From node B to node D, the cost is 8. The weight from A to D through B is 3 + 8 = 11. Update node D's weight to 11.

From node B to node C, the cost is 4. The weight from A to C through B is 3 + 4 = 7. Since 7 is less than C's previous weight, 9, update node C's weight to 7.

Fig. 12. The first few steps of Dijkstra's Algorithm through a weighted node graph. In each step, a new node is visited and the corresponding weights of nodes are updated.



Visited Nodes = { A, B, C }

From node C to node D, the cost is 2. The weight from A to D through B, C is 3 + 4 + 2 = 9. Since 9 is less than D's previous weight, 11, update node D's weight to 9. Now C is fully visited.

Visited Nodes = { A, B, C, D }

Since node D has no unvisited neighbors, it is now fully visited. The optimal path is given A -> B -> C -> D.

Fig. 13. The last few steps of Dijkstra's Algorithm through a weighted node graph. The final step shows the optimal path given by the weights of the nodes.

It is the best fit if we are trying to find multiple destinations. However, since Dijkstra's algorithm is focused on exploring every possible state, it is not the most efficient possible approach to pathfinding [3]. The basis of pathfinding is finding the shortest path between two nodes, whereas Dijkstra's algorithm is the most optimal approach for finding paths between multiple nodes. The aforementioned tweak in Dijkstra's algorithm solves this by offering a heuristic function and is called A* (A-Star).

## V. A Star (A*) Algorithm

What's interesting about Dijkstra's is that, with a small tweak in the original 1956 algorithm, we can prioritize the direction we head in. This variant is called A star, or commonly abbreviated as A*.

A* is widely considered the most efficient pathfinding algorithm to find the shortest path between two nodes. Dijkstra's Algorithm calculates distance from the root node to calculate cost, which made it very effective when we have multiple weighted destinations on a graph or grid. However, it can often end up exploring in directions that may not be promising, just like BFS. A* takes an additional step by using the distance from the root node as well as the estimated distance to the end node to calculate its heuristic node. This means that A* will actively seek out the direction in which the end node exists, which reduces valuable computation time by searching less promising directions than, say, BFS or DFS.

In our application, since we only have 4 possible directions, our basic heuristic will be the Manhattan Distance. If we were working with more directions on our grid, A* could make use of heuristics such as Diagonal Distance or Euclidean Distance to take different approaches. We can also tweak the heuristics to match certain scenarios more efficiently as well.

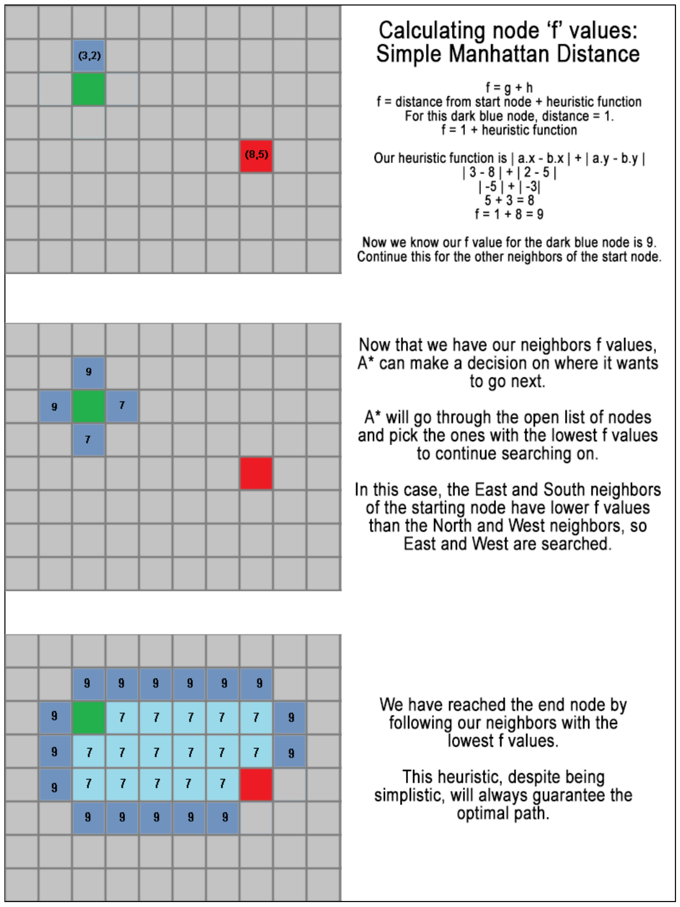
$$d(a,b) = \sum_{i=1}^{n} |a_i - b_i|$$

Fig. 14. The Manhattan distance (d) between our two nodes a = $(a_1, a_2, ..., a_n)$ and b = $(b_1, b_2, ..., b_n)$ is the sum of the distances between each node's coordinates. In simpler terms, | node_a.x - node_b.x | + | node_a.y - node_b.y |.

Every node has 3 values that A* will use: f, g, and h. Value f is used to determine which node to visit - A* wants to pick the node with the lowest f cost, since in theory the closer the current node is to the end node, the lower the f value is. [2] The value f is set to g + h where g is the current node's distance from the start node and h is the heuristic that estimates how far the current node is from the end node.

Node n's f value: f(n) = g(n) + h(n)

Value g is fairly simple to calculate - every turn we move, we add 1 to g since the movement cost to any adjacent node in a grid is 1. Value h is dependent on the heuristic that we decide on. In our app, there are two possible heuristics to work with. Let's start with the basic Heuristic Distance outlined in Figure 14.



Calculating node 'f' values: Simple Manhattan Distance

f = g + h
f = distance from start node + heuristic function
For this dark blue node, distance = 1.
f = 1 + heuristic function

Our heuristic function is | a.x - b.x | + | a.y - b.y |
| 3 - 8 | + | 2 - 5 |
| -5 | + | -3|
5 + 3 = 8
f = 1 + 8 = 9

Now we know our f value for the dark blue node is 9. Continue this for the other neighbors of the start node.

Now that we have our neighbors f values, A* can make a decision on where it wants to go next.

A* will go through the open list of nodes and pick the ones with the lowest f values to continue searching on.

In this case, the East and South neighbors of the starting node have lower f values than the North and West neighbors, so East and West are searched.

We have reached the end node by following our neighbors with the lowest f values.

This heuristic, despite being simplistic, will always guarantee the optimal path.

Fig. 15. An in-depth look at how each node gets assigned its 'f' value and how A* decides to move in relation to it.

This heuristic works well and will always find the optimal path as it is underestimating the heuristic.

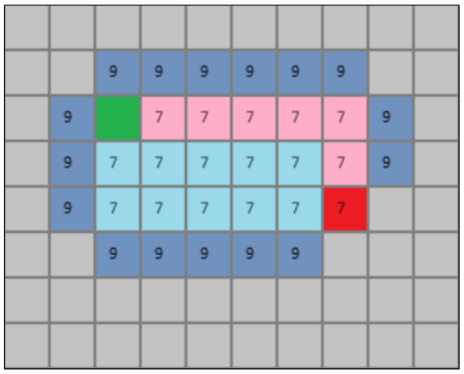


Fig. 16. A* running on the basic Manhattan Distance heuristic and generating the same f values as Figure 15.

The other possible heuristic available in our app is the basic Manhattan Distance that squares the absolute values of the differences between our x and y coordinates. This gives us $|a.x - b.x|^2 + |a.y - b.y|^2$. This will give us a greater difference between a node's f values, which hypothetically will narrow our search tree down.

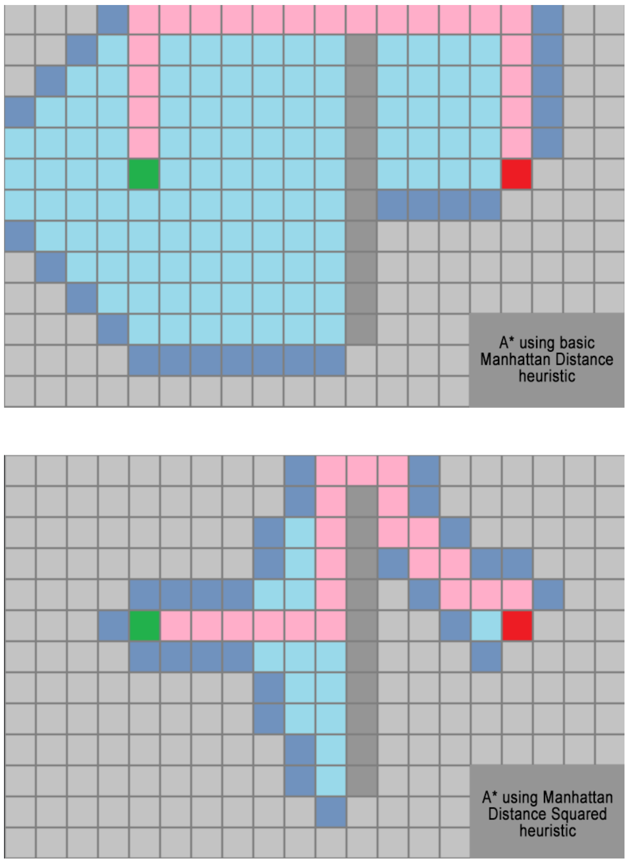

A* using basic Manhattan Distance heuristic
A* using Manhattan Distance Squared heuristic

Fig. 17. A* running on the app. The basic Manhattan Distance heuristic cares more about how close the current node is to the start than the Manhattan Distance Squared heuristic. They generate the same path length, but the basic heuristic searches 95 more nodes.

Changing our heuristic is dangerous, though: if we scale up h in our equation f = g + h, then g will have less and less impact on f, meaning that A* will prioritize how close the node is to the end node over how close it is to the start node. On the other side, if we are underestimating the heuristic by scaling it down, h will have less impact on f and A* will prioritize nodes close to the start node over how close it is to the end node.

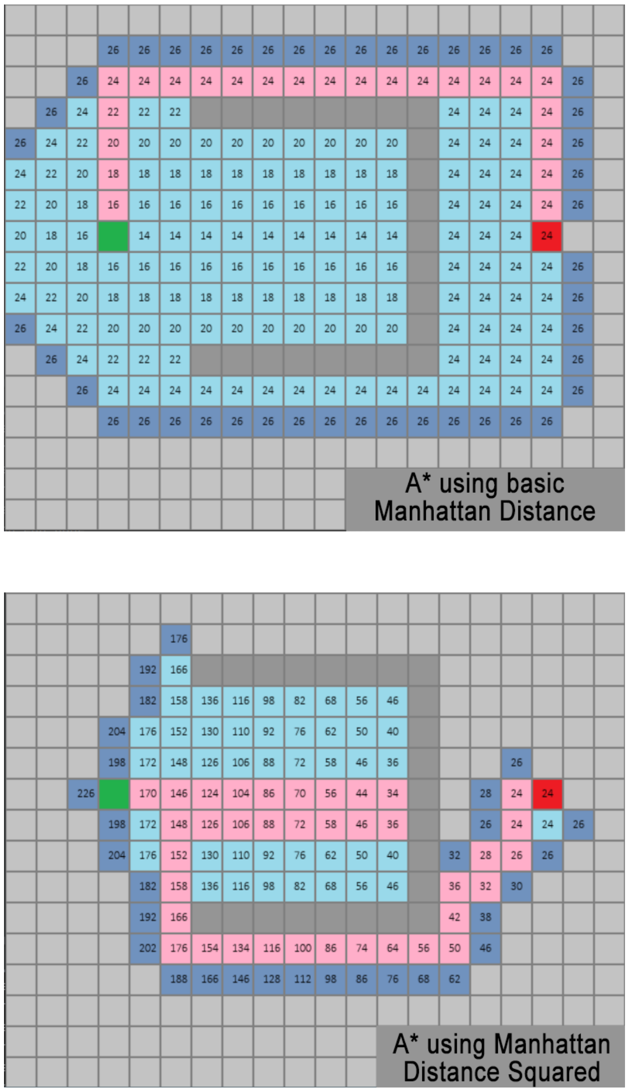

A* using basic Manhattan Distance
A* using Manhattan Distance Squared

Fig. 18. A* running on the app. Here, the overestimating Manhattan Distance Squared gets too greedy and ends up increasing its path length by extending far into the obstacle. The basic Manhattan Distance searches more nodes, but guarantees our optimal path since it is not overestimating.

It becomes clear that A* requires an underestimation of the heuristic to guarantee the shortest path. [4] We can tweak the heuristic to give us either an advantage in speed (nodes searched) or efficiency (shortest path), but not both. Therefore, to ensure the optimal path, we have to prioritize efficiency and potentially give up speed to guarantee optimal path. The heuristic of Manhattan Distance Squared is still in the app to best visualize how tweaking the heuristic affects A*'s performance.

## VI. Summary

Pathfinding is a modern-day adaption of some of computer science's oldest problems and shares algorithms with old and modern concepts alike. BFS is exhaustive but computationally simple, so it will often waste time exploring directions that are in the opposite direction of the end node. DFS can sometimes find end nodes quickly if they are in the optimal directions of its traversal, but due to its dependence on pre-determined direction order, it can often overlook solutions that are nearby. Both of these algorithms have their place in modern computing, but it is clear that DFS is quite inconsistent and not a good solution for simple pathfinding. BFS is better since it guarantees the shortest path, but not cost-effective.

Dijkstra's Algorithm is an improvement in theory since it considers cost, but in practice on a grid it proves similar to BFS since the cost between units on a grid is 1. It remains searching in wrong directions, just like BFS and DFS, and it also comes with higher computational times. A* aims to fix this by giving an estimated cost that includes how far the current node is from the end node and the distance from the start node. A*'s heuristics can be tweaked for individual situations to gather the utmost efficiency or speed which also makes it much more applicable to many modern-day programs thanks to its dynamic nature, truly making it the best choice for any simple pathfinding problem.

## VII. Acknowledgements

I would like to acknowledge Professor Brian Gormanly of Marist College for undertaking this research and app development with me. Professor Gormanly provided resources that proved immensely useful towards understanding and programming Dijkstra's Algorithm and A*. His experience in the field of robotics and knowledge of Computer Science was valuable and appreciated, as well as his feedback on various stages of this paper and the app's development. I would also like to acknowledge Jack Grossman for helping me proofread this paper and for his advice.

## References

[1] Algfoor, Z. A., Sunar, M. S., and Kolivand, H. A comprehensive study on pathfinding techniques for robotics and video games. *Interation Journal of Computer Game Technology 2015* (2015).

[2] Botea, A., Bouzy, B., Buro, M., Bauckhage, C., and Nau, D. Pathfinding in Games. In *Artificial and Computational Intelligence in Games*, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds., vol. 6 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013, pp. 21–31.

[3] Cui, X., and Shi, H. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security 11*, 1 (2011).

[4] Leigh, R., Louis, S. J., and Miles, C. Using a genetic algorithm to explore a*-like pathfinding algorithms. In *2007 IEEE Symposium on Computational Intelligence and Games* (2007), IEEE, pp. 72–79.

[5] Rodriguez, M. A., and Neubauer, P. The graph traversal pattern. In *Graph Data Management: Techniques and Applications*. IGI Global, 2012, pp. 29–46.

[6] Stout, B. Smart moves: Intelligent pathfinding. *Game developer magazine 10* (1996), 28–35.