Data and Info Visualization - Pathfinding Algorithms Visualized

Andrew Struthers

March 2023

Introduction

In this project, I will be creating a visualization for two different grid-based pathfinding algorithms operating on a grid of walkable and unwalkable tiles. The pathfinding algorithms I will be implementing are A* and Breadth-First Search (BFS) algorithms. Both of these pathfinding algorithms are very common in the game development world, and each has it's own benefits and drawbacks. My goal is to create an executable that allows the user to specify the dimensions of the grid, mark the start and finish for the algorithm, and mark certain cells as "walls", making those "walled" cells unwalkable. The appropriate pathfinding information would update on the visualization as the user is stepped through the pathfinding execution.

A* pathfinding calculates heuristic scores for each cell, and those scores determine the optimal path from start to finish. BFS doesn't use any heuristic calculations and instead checks the neighbors of each cell, adding the new cells to a queue, and creating a linked list able to be traversed to find the shortest number of cells from start to finish once the end has been found. Both algorithms will be discussed at length in this paper, as well as a showcase of the results from the software I developed.

This software was made entirely inside of the Unity game engine. I chose to use Unity because it provided a stable and user-friendly environment for graphically displaying a "game" to a screen. I didn't have to worry about making my own Time class, Camera and screen renderer classes, or any of the other backend framework required to put something on a screen. This allowed me to focus on the actual algorithms and visualization, and not focus so much on having to build a robust system for users to interact with.

Theory of A* Pathfinding

The goal of the A* pathfinding algorithm is to heuristically determine the shortest path between two nodes in a graph. In this case, our map is grid based, but A* pathfinding can work on any weighted or unweighted graph. This is one of the huge benefits of A* over BFS, which only works with unweighted graphs. A* can be applied to any weighted graph, such as a network of streets and highways on Google Maps, where the length of the street serves as the weight in this algorithm. Each node in A* pathfinding has 3 values, a G cost, an H cost, and an F cost. The G cost is the walking cost from the start node. We calculate the horizontal or vertical movement cost to be 1.0 because we move one unit in either direction, and the diagonal cost to be 1.4, since $\sqrt{1^2 + 1^2} = \sqrt{2} \approx 1.4$. We want to work with integers because they are nicer, so we multiply both of these movement costs by 10.

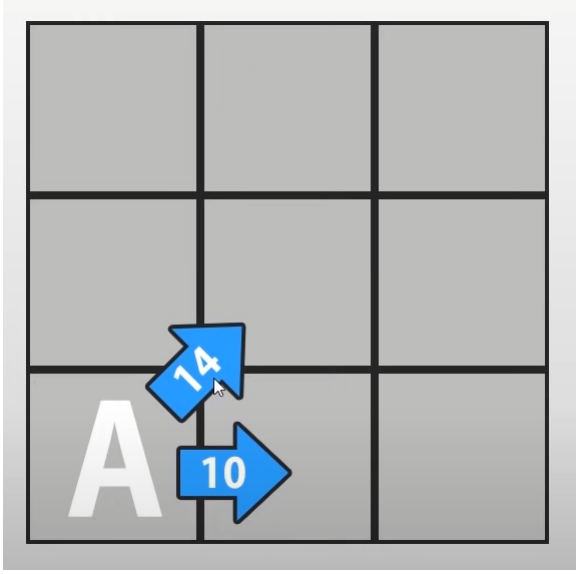

Figure 1: A* movement cost explained

Now that we understand the G cost, we have the H cost. The H cost is the heuristic cost to reach the end node, assuming we can move from our current node directly to the end node without dealing about walls or other unwalkable nodes. This is an estimate to reach the goal without worrying about any walls. The H cost is a simple estimate to figure out what nodes we should prioritize first. Finally, the F cost is simply our G cost plus our H cost. The A* algorithm will prioritize nodes with lower F costs, because those are more likely to be closer to our goal node.

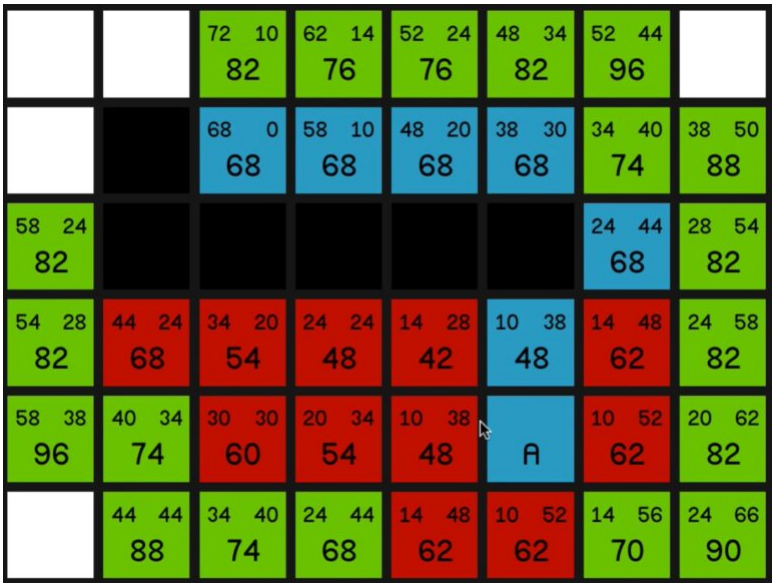

Figure 2: A* pathfinding algorithm example

We can see these concepts in action in the figure above. We start at the node marked A. The top left value in each node is the G cost, which we can see for every vertical or horizontal node from A, the G cost is 10, and each diagonal node has a G cost of 14. The A* algorithm will check the lower F cost neighbors first, which is why there are multiple red nodes on the left of the A node. The algorithm knew that those tiles were closer to the goal, and tried to find a way around the wall by moving left first. Notice that, on the path that the algorithm eventually found, the G cost constantly increases whole the H cost constantly decreases. This means that the algorithm

found a path where each node was closer to the goal than the previous node, which is exactly the behavior you would expect when finding the shortest path between two nodes. Once the goal node is reached, the path is traced back from final node to origin node. Each node knows the previous path node, so we can easily traverse this linked list to find our optimal path.

The algorithm operates on two lists of nodes, known as the open list and the closed list. The open list is the list of all nodes queued up for searching, and the closed list is the collection of all nodes that have already been searched. The algorithm will keep going until either the current node is the end node, or the open list is empty, meaning there was no found path from start to finish. This is all the theory we need to know in order to implement the A* algorithm.

Theory of Breadth-First Search Pathfinding

Breadth-First Search (BFS) is a simple graph traversal algorithm that is very common in many problems in Computer Science. This algorithm is introduced very early on in formal Computer Science courses, especially in Data Structure and Algorithms courses, when discussing Linked Lists. BFS algorithm will traverse each node in a graph, so the average runtime of BFS algorithms is $O(n)$. BFS works by traversing each node in a graph until it reaches the goal node, storing a reference in each node to the parent that the current node came from. This type of traversal has a unique property where, when the algorithm reaches the goal node, we can trace back the parent references such that we will have the shortest path from the end node to the origin node. One of the disadvantages of this algorithm over A* is that BFS only works on unweighted graphs. In our case, since we are using a grid for the navigation, this effectively means that BFS can only traverse the graph in horizontal or vertical movements. It cannot traverse diagonally, like A*. In our case, we can represent the grid as a network of nodes and edges, where each tile is a node. In this representation, each walkable tile next to the current tile will create an edge between the current tile and the walkable tile, forming our pseudo linked list. Unwalkable tiles simply won't create an edge to surrounding tiles, preventing the algorithm from traversing into the "wall" tiles.

BFS makes use of a queue data structure. The queue starts with the first node in our graph, then adds in all surrounding nodes to the queue. The algorithm will iterate until the current node is equal to the final node, or all nodes have been searched. Each iteration, the algorithm pops the first element of the queue and checks its surrounding tiles. Any surrounding tiles that aren't already in the queue get added to the end of the queue. The algorithm will continue iterating, popping the first element of the queue, until the end node has been found. Each time a node gets added to the queue, BFS give the newly added node a reference to the parent node, i.e. the node that the current node came from. Once the final node has been found, we can simply trace back the path using the parent references until we get back to the start. Because of the properties of a BFS, this traced back path will be the shortest path possible on an unweighted graph.

Pathfinding Software

Now that we understand the theory behind both pathfinding algorithms, we can discuss the software I've developed for this visualization. I have built out an executable that allows the user to specify the dimensions of the grid being used, mark the start and finish for the algorithms, and mark certain cells as "walls", i.e. unwalkable nodes. Users can then run both algorithms, visualized simultaneously side by side on two separate grids, where the visualization steps through each calculation like the functionality provided by a debugger. Since we are running two different pathfinding algorithms, there will be two grids created, one for A* and one for BFS. Each grid's pathfinding algorithm information would update on the visualization as algorithms run, and some runtime statistics would be displayed to allow a more direct comparison of each algorithm in the

created scenario. Since we know that A* pathfinding calculates scores in each cell, and those scores determine the optimal path from start to finish, whereas BFS checks the neighbors of each cell until it finds the path that goes through the least amount of nodes, the visualizations for each algorithm will need to be slightly different. This is the main reason behind creating two identical grids, in addition to reducing visual clutter. Juxtaposing the two algorithms side by side will help the user to more easily understand how each differs, and how each works.

The software has a home screen which includes sub-menus for references used and a project description. Once the user navigates to the actual visualization, they are met with a screen that informs them of what the tool is and how they can use it, as well as some options for configuring the width and height of the grids. They are then greeted with the page seen in the figure below, which allows them to customize and interact with the grids by placing the start, finish, and any walls anywhere on the grids.

Figure 3: Empty default graph inside of my software implementation

This software has been made with user experience in mind. I tried my best to make it as "unbreakable" as possible, so that anyone who wishes to could download the software from my Github profile and run the visualization, without needing to worry about "pressing the wrong button" or "doing it wrong".

Algorithm Implementations

Below, you will be able to see the two pathfinding algorithms implemented in this project. Both rely on custom *AStar_PathNode* or *BFS_PathNode* objects respectively, which both serve as *TGridObject* generics for my custom generic *Grid* class. The grid a two dimensional list of *TGridObject* objects.

```
public List<BFS_PathNode> FindPath(int startX, int startY, int endX, int endY)
    {
        BFS_PathNode startNode = grid.GetGridObject(startX, startY);
        BFS_PathNode endNode = grid.GetGridObject(endX, endY);

        queue = new Queue<BFS_PathNode>();
        exploredNodes = new List<BFS_PathNode>();

        queue.Enqueue(startNode);

        for(int x = 0; x < grid.GetWidth(); x++)
        {
            for(int y = 0; y < grid.GetHeight(); y++)
            {
                BFS_PathNode node = grid.GetGridObject(x, y);
                node.queueOrder = 0;
                node.cameFromNode = null;
            }
        }

        startNode.queueOrder = queueOrder;

        BFS_PathfindingDebugStepVisual.Instance.ClearSnapshots();
        BFS_PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, startNode, queue,
exploredNodes);

        while(queue.Count > 0)
        {
            BFS_PathNode currentNode = queue.Dequeue();
            if(currentNode == endNode)
            {
                //reached final node
                BFS_PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, currentNode, queue,
exploredNodes);
                BFS_PathfindingDebugStepVisual.Instance.TakeSnapshotFinalPath(grid,
CalculatePath(endNode));
                return CalculatePath(endNode);
            }
            exploredNodes.Add(currentNode);

            foreach(BFS_PathNode neighborNode in GetNeighborList(currentNode))
            {
                if (exploredNodes.Contains(neighborNode) || neighborNode == null) continue;

                queueOrder++;
                neighborNode.queueOrder = queueOrder;
                if (!neighborNode.isWalkable)
                {
                    exploredNodes.Add(neighborNode);
                    continue;
                }

                exploredNodes.Add(neighborNode);
                neighborNode.cameFromNode = currentNode;
                queue.Enqueue(neighborNode);

                BFS_PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, startNode, queue,
exploredNodes);
            }
        }

        //Out of nodes in queue (searched through whole map, couldn't find a path)
        return null;
    }
```

Figure 4: Breadth-First Search algorithm implemented in Unity

```
public List<AStar_PathNode> FindPath(int startX, int startY, int endX, int endY)
    {
        AStar_PathNode startNode = grid.GetGridObject(startX, startY);
        AStar_PathNode endNode = grid.GetGridObject(endX, endY);

        if(startNode == null || endNode == null)
        {
            return null;
        }

        openList = new List<AStar_PathNode> { startNode };
        closedList = new List<AStar_PathNode>();

        for (int x = 0; x < grid.GetWidth(); x++)
        {
            for (int y = 0; y < grid.GetHeight(); y++)
            {
                AStar_PathNode pathNode = grid.GetGridObject(x, y);
                pathNode.gCost = int.MaxValue;
                pathNode.CalculateFCost();
                pathNode.cameFromNode = null;
            }
        }

        startNode.gCost = 0;
        startNode.hCost = CalculateDistanceCost(startNode, endNode);
        startNode.CalculateFCost();

        AStar_PathfindingDebugStepVisual.Instance.ClearSnapshots();
        AStar_PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, startNode, openList,
closedList);

        while(openList.Count > 0)
        {
            AStar_PathNode currentNode = GetLowestFCostNode(openList);
            if(currentNode == endNode)
            {
                //reached final node
                AStar_PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, currentNode,
openList, closedList);
                AStar_PathfindingDebugStepVisual.Instance.TakeSnapshotFinalPath(grid,
CalculatePath(endNode));
                return CalculatePath(endNode);
            }

            openList.Remove(currentNode);
            closedList.Add(currentNode);

            foreach(AStar_PathNode neighborNode in GetNeighborList(currentNode))
            {
                if (closedList.Contains(neighborNode) || neighborNode == null) continue;
                if (!neighborNode.isWalkable)
                {
                    closedList.Add(neighborNode);
                    continue;
                }

                int tentativeGCost = currentNode.gCost + CalculateDistanceCost(currentNode,
neighborNode);
                if(tentativeGCost < neighborNode.gCost)
                {
                    neighborNode.cameFromNode = currentNode;
                    neighborNode.gCost = tentativeGCost;
                    neighborNode.hCost = CalculateDistanceCost(neighborNode, endNode);
                    neighborNode.CalculateFCost();

                    if (!openList.Contains(neighborNode))
                    {
                        openList.Add(neighborNode);
                    }
                }
                AStar_PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, currentNode,
openList, closedList);
                nodesVisited++;
            }
        }

        //Out of nodes in openList (searched through whole map, couldn't find a path
        return null;
    }
```

Figure 5: A* algorithm implemented in Unity

Results and Software Demonstrations

Now, we can see what the software actually outputs and how we can gather meaningful results. Below we can see two simple 10 by 10 grids, where the left grid is configured to use A* pathfinding and the right grid is configured to do BFS. We can see that there is a little hotbar on the bottom where the user can select if they want to place the start, finish, or some walls, as well as some buttons to start or reset the simulation. We can also see that, under the titles of each grid, there is a place for some stats for each of the algorithms. In this simple example we can see that the start has been placed in the bottom left corner of the grid, and the finish has been placed in the top right of the grid.

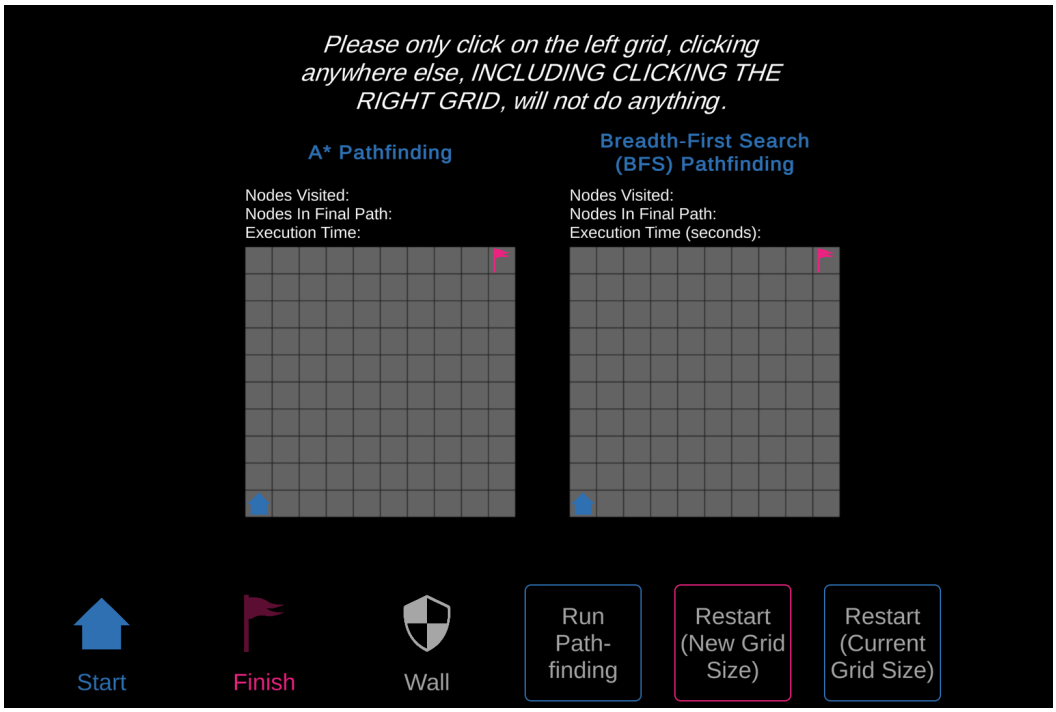

Figure 6: Very simple pathfinding demonstration

When the user clicks run, the pathfinding algorithms calculate their respective paths. The statistics and path were calculated very quickly, but the visualization steps through "snapshots" of algorithm, one snapshot every 50ms. Even though the full visualization isn't complete, we can see that the stats for each algorithm had already been determined, and we can see that the visualization is currently about halfway through showing the user different snapshots of each algorithm's runtime behavior. We can see that the A* algorithm is calculating the G cost and displaying it in the top left corner, the H cost in the bottom right corner, and the F cost in the center. For the BFS visualization, each node has a corresponding number representing the order the nodes were searched in. We can see, for example, that the node directly to the right of the start in the BFS graph was searched 2nd, and the one directly above the start was searched 3rd. The nodes are each color coordinated to inform the user of some additional information. Each blue node is either a node in the open list for A*, or in the ready queue for BFS, i.e. they have yet to be searched. The red nodes are nodes that have been searched by the algorithm already. The green node in A* is the current node being processed, and the green node in BFS represents the starting node, since we can trace the processing of BFS by looking at the order in which the node was searched.

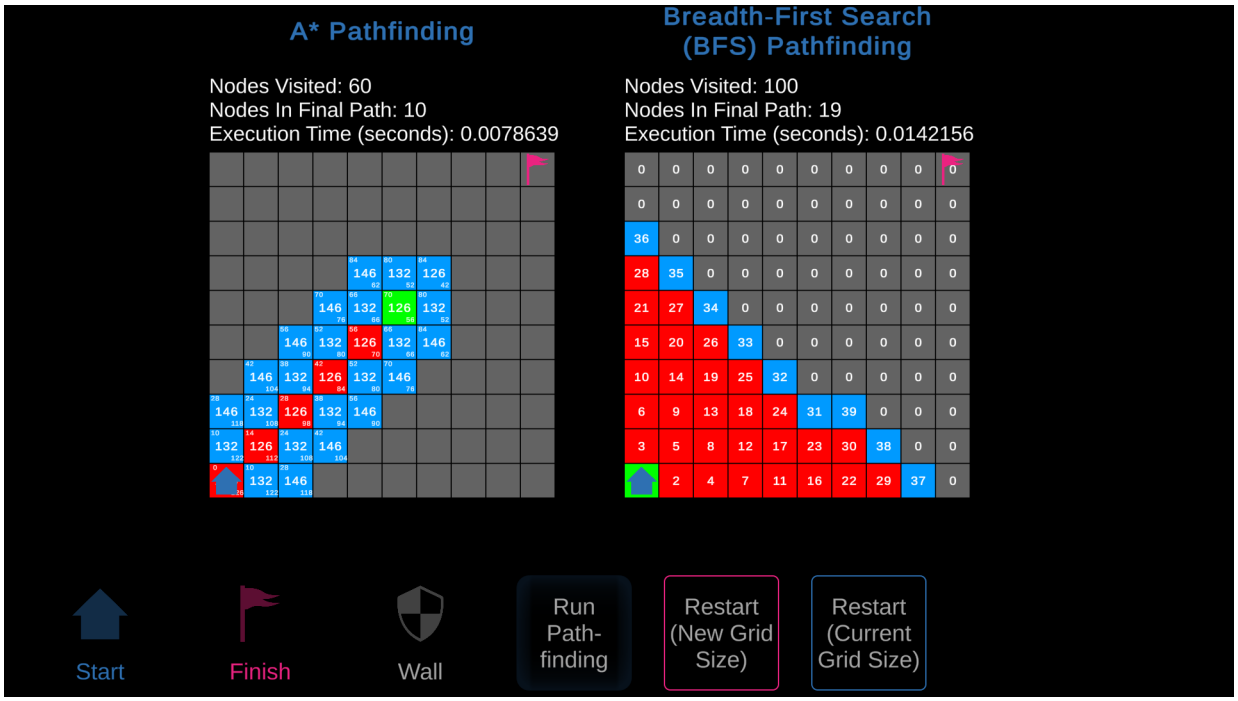

Figure 7: Basic pathfinding demonstration mid-visualization

Here, we can see what the visualization ends up looking like once each snapshot has been displayed. We can see the fastest path each algorithm determined highlighted in green, with the other algorithm-specific information still displayed. As we can see, the BFS algorithm found the shortest unweighted path, since BFS can't process diagonal nodes appropriately. In this simple example, we can see that A* is the clear winner, with fewer nodes visited, fewer nodes in the final path, and a faster execution time.

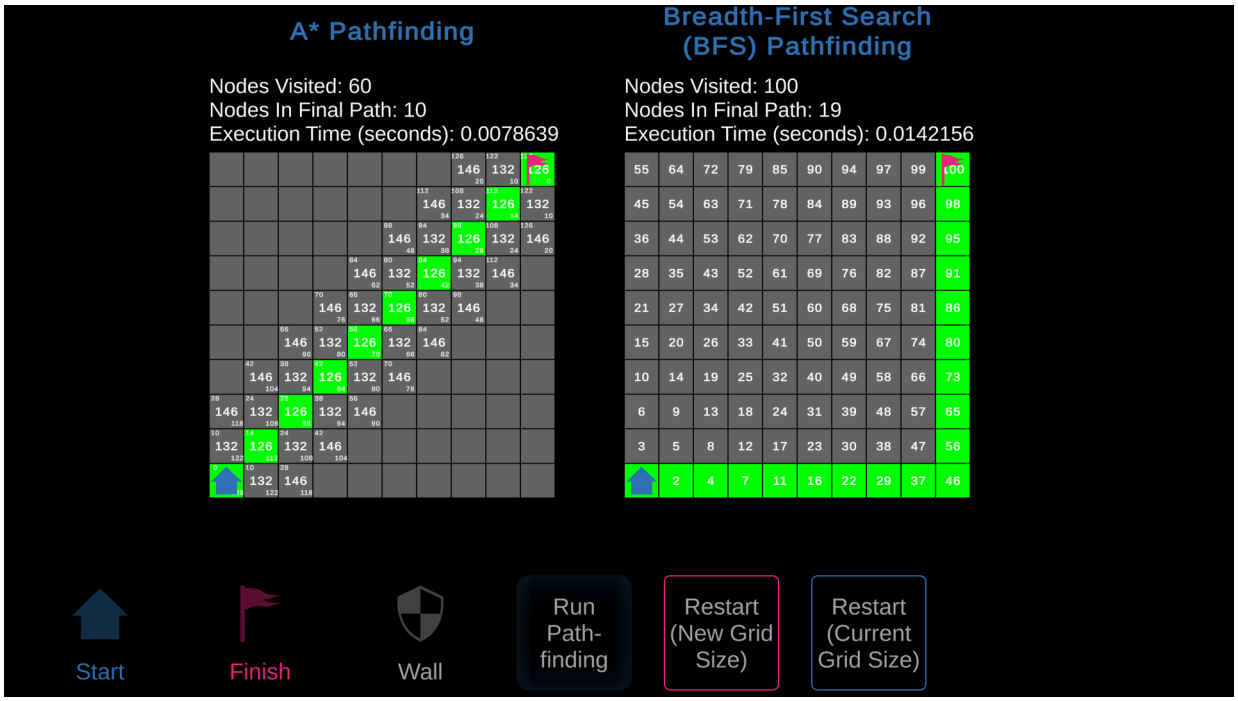

Figure 8: Basic pathfinding at the end of all intermittent snapshots

Below, we can see a much more complex grid of a different size. In this example, we have two grids of 6 by 15. These grids also have numerous walls placed with the intention of "blocking" the straightforward path from start to finish. We can see that, like in the basic example, the pathfinding algorithms execute very quickly, and the path from start to finish is found long before the snapshot visualization is complete.

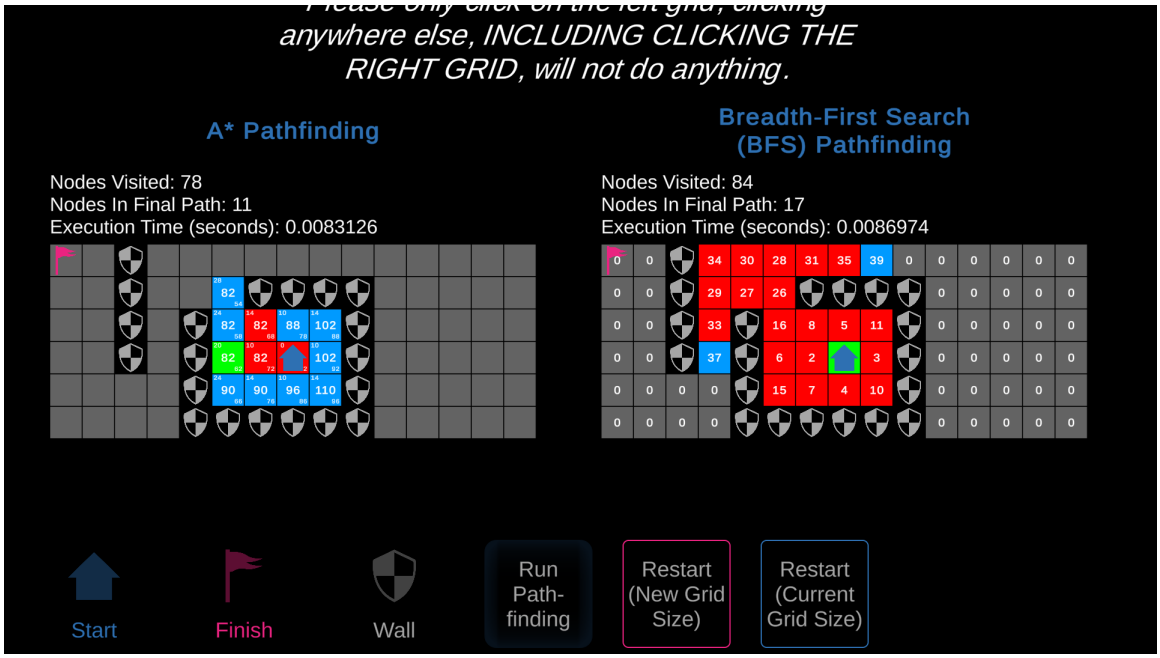

Figure 9: More complex pathfinding example

At the end of this simulation, we can see the visualization at the final snapshot during execution. We can once again see that A* is the clear winner in terms of path length and nodes visited, but the execution time of both algorithms starts to get a lot closer together when we start adding more complex paths.

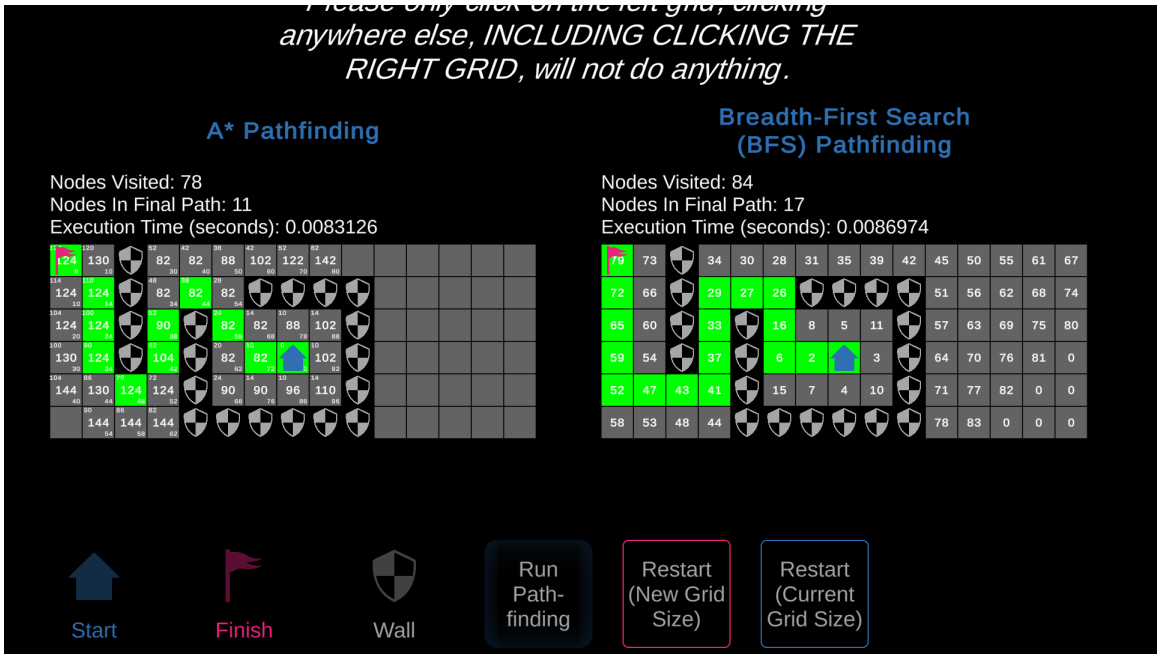

Figure 10: Final visualization of the more complex example

In this final example, we can see a much larger and more complex grid was created specifically with the goal of confusing the A* pathfinding algorithm. We can see from the statistics that the A* pathfinding algorithm took more than twice the time to execute compared to BFS, and searched more than twice the number of tiles. The reason that A* searched more tiles in this case was because A* will recalculate the G, H, and F costs of some neighbor tiles even if the algorithm had already previously visited these tiles. This is because the current implementation of the algorithm doesn't check to see if the tile the neighbor tiles already have an existing F cost. There is more discussion on A* pathfinding optimization later in this report.

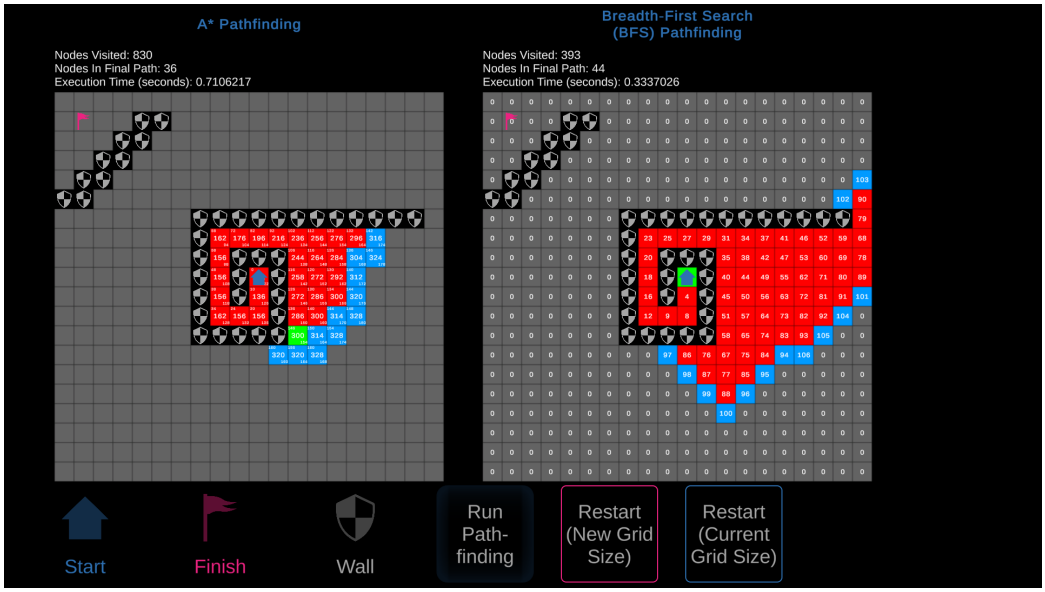

Figure 11: Large, intentionally designed map to punish A*

We can see that BFS has finished calculating and visualizing the path from start to finish before A* is done visualizing, even though each visualization has the same snapshot display timer. Notice, however, that once A* got around the bottom left of the spiral in the center, it stopped checking any of the tiles in the bottom half of the grid, whereas BFS still searched those tiles. This is because of the heuristics that A* calculates for each tile.

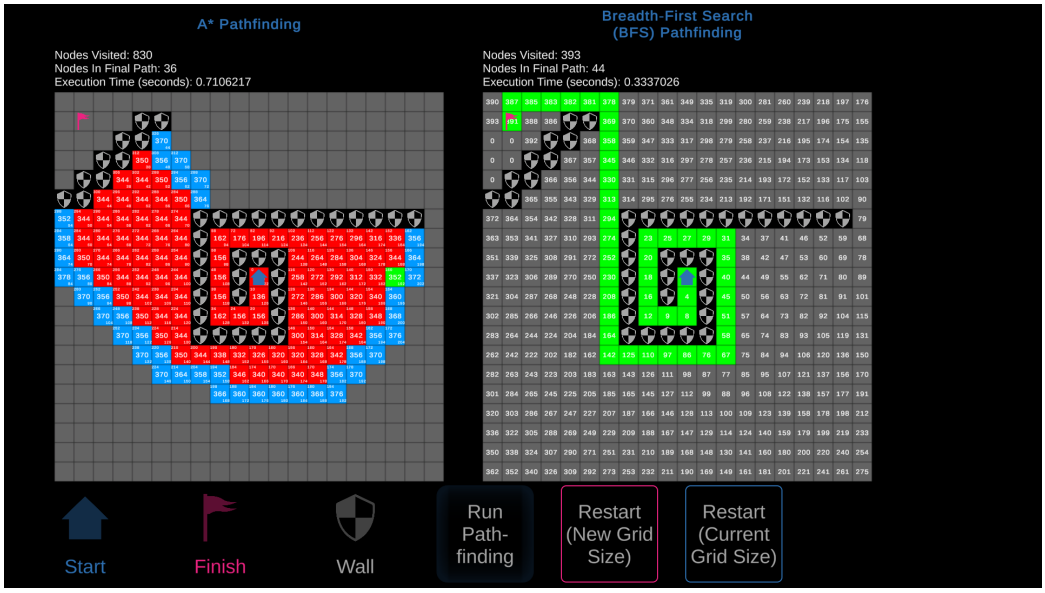

Figure 12: Time difference in the pathfinding visualized

Something interesting happened in this example. At a certain point, A* found that checking to the right half of the grid was technically faster, according to the heuristics it calculated. Us, as human observers, can clearly see that that didn't make any sense, but the heuristic calculations implemented are relatively simple in this case. Again, there is more discussion on A* optimization later in this report.

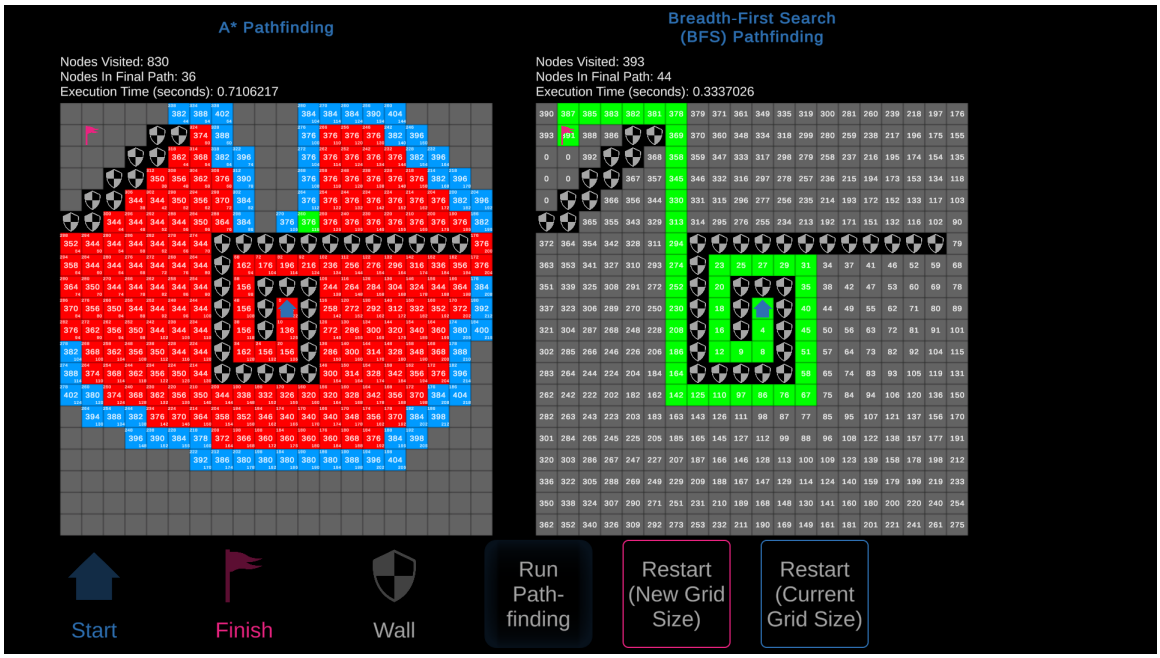

Figure 13: Interesting result of A* heuristics

Finally, we can see the optimal path both algorithms calculated. In this case, BFS clearly won, because the execution time was more than twice as fast, and only has 8 more nodes than A*. That being said, this map was intentionally created to mess with the A* heuristic equations.

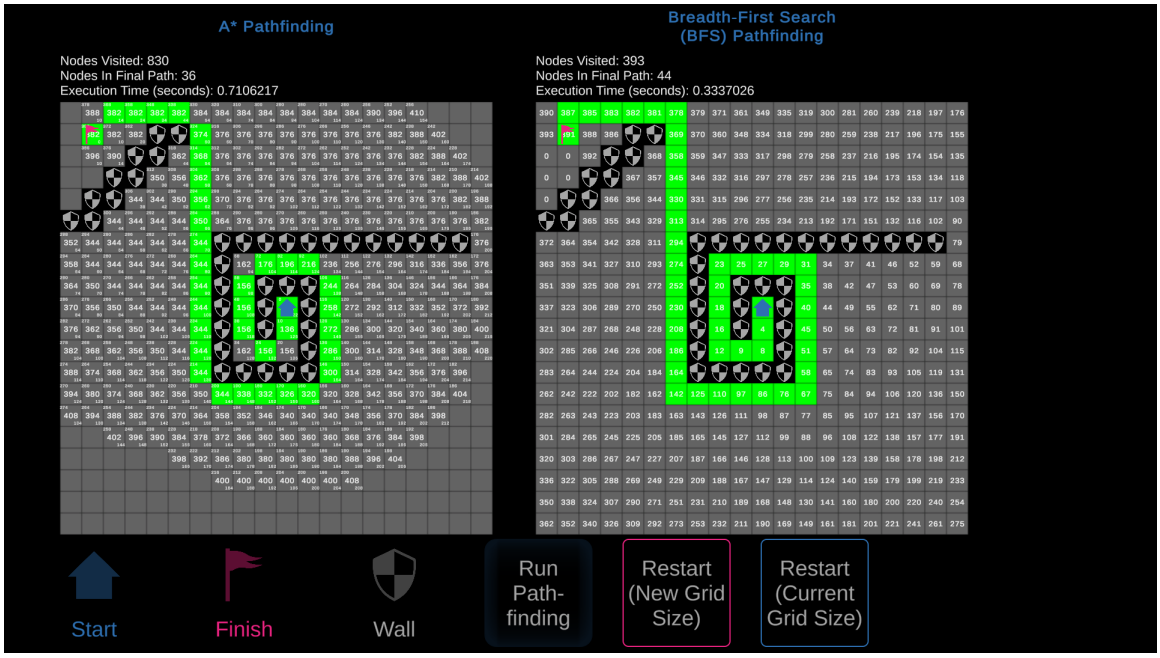

Figure 14: Final visualization of the punishing map

Interesting Findings

One of the most interesting findings of this visualization is how different the two algorithms really are. The heuristic calculations of A* end up pointing to the goal much sooner than BFS, but in most cases the runtime and nodes searched are very similar nonetheless. An interesting property of A* that gives it the edge over BFS is that A* can process and handle diagonal paths, whereas this BFS implementation cannot, which results in A* being much faster on large tiles where the path is relatively unimpeded by walls. A* can traverse very large grids with ease if it has a mostly straight shot. The demonstrations in the previous section were made intentionally to impede A* and show that BFS is still a good pathfinding algorithm despite the lack of heuristics. Despite the vast differences between both algorithms, they almost always find a very similar path, albeit the BFS path is just the A* path without the diagonal movements.

Another interesting finding during the research of the theory behind both pathfinding algorithms is that BFS can actually handle diagonal movement without any issues. I could have easily added the four diagonal nodes to the function that gets all neighbor nodes, just like in A*, but there is a slight problem with that. BFS evaluates the movement between nodes as though every movement cost the same amount. Because of BFS being unable to weight the movement costs and differentiate between diagonal and horizontal or vertical movements, if BFS was allowed to see the diagonal neighbors as well, it would not guarantee the shortest path. In most cases, BFS would still find a path, but it won't necessarily find the shortest path any more. Since this is the case, BFS is only typically used in unweighted graph traversal, which is completely fine in most cases. Tile and grid based games that only allow movement to the left, right, up, or down can still benefit from implementing BFS, especially if the maps are complex. We saw that BFS worked better than A* on very complex maps.

Hardships and Lessons Learned

One of the biggest lessons learned throughout this project was how much a simple A* pathfinding implementation can be improved. One of the big strengths of using A* over other pathfinding algorithms is the heuristics involved. The heuristic equation we used here was the simplest version of A*, with a very simple heuristic distance cost serving as the main driver for what node A* looks at next. There are many other, more complicated heuristic models that I could have implemented instead, that all greatly increase the performance of A* in the more complex graphs which we saw that algorithm struggling. Additionally, the implementation relies on finding the next lowest F cost node, which I used a simple linear search for. This is fine on small grids, but we can clearly see the performance difference between A* and BFS become more noticeable on the larger graphs. Implementing a binary search or other algorithm could have helped reduce some of the runtime inefficiencies. I also never accounted for nodes that weren't yet in the closed list, but already had the cost calculations completed. This is why, in all of the A* examples, we can see the "Nodes Visited" statistic is higher than the sum of all of the nodes that have the calculations visualized. In those cases, I recalculated all of the costs, which involved using the linear smallest F cost algorithm previously described, in addition to some other algorithms, for each node. Finally, the algorithm I implemented relied on objects. Each node was a reference to a TGridObject. I could have implemented the pathfinding algorithm in a data oriented way to take advantage of the unity DOTS (Data-Oriented Technology Stack), where each of the nodes could have been represented by a struct instead of an object. Inside of Unity, there are two optimization methods known as "Jobs" and the "Burst Compiler", which a purely data oriented approach to the A* algorithm could have benefit from. The data oriented approach makes use of high speed data types inside of Unity's namespace that would have allowed me to decrease the speed drastically. The implementation that I have is a very basic, zero optimization, baseline implementation of A*. Even with the object ori-

ented approach, there is a lot of optimization that I could implement to decrease the runtime of the algorithm. With these optimizations, A* will run much faster than BFS on any size or complexity of grid. In a real game scenario, this high-speed optimization would allow massive grids with many agents navigating throughout a complex world with very little runtime performance impact.

The biggest hardship I faced while working on this project was the actual visualization and snapshots of the pathfinding algorithms. Finding a way to dynamically and effectively visualize these algorithms took a lot of research and work, but it ultimately provided what I believe to be a nice result. One of the main reasons I used Unity over doing this in a more "pure" environment was so that I could easily visualize and display data to a screen without worrying about how that's actually implemented too much. I ended up struggling to find a method of visualization inside of Unity for much longer than any other part of this project. The actual implementation of the generic grid, and the implementation of both algorithms, was much faster than the work required to allow the user to customize the grid and then visualize their simulations. All in all, I think I was able to come up with a nice result that allows the user to play around and experiment while still displaying the proper information in a nice and clean way.

Conclusion

Throughout this report, you have learned about some of the theory behind both the A* an the BFS pathfinding algorithm, learned about the software solution I created inside of Unity, saw some results that showcased the strengths and weaknesses of both algorithms as well as showcasing the flexibility and user customization of my software, and discussed some ways to optimize A* from the baseline implementation I provided to something that you would see in large games. I learned that even the basic A* algorithm with zero optimization is faster and provides a shorter path than the BFS algorithm, but A* isn't perfect and can be tricked easily by creating algorithms that take advantage of the simple heuristics of A*.

One direction that I would take this project if I had more time is to implement more complicated weighted algorithms like Dijkstra's pathfinding algorithm. It would be easy enough to extend the graph visualization and customization to handle more algorithms, so this might be a fun project I continue working on in my free time. I would also like to implement the data oriented approach to A* pathfinding to see the speed benefits for myself. I am working on a large grid based combat game similar to XCom on my free time, so the lessons learned and experience gained from working on this project will directly impact in my game development efforts as well.

References

[1] Code Monkey. A* pathfinding in unity. *YouTube*, Oct 2019.

[2] Nicholas Swift. Easy a* (star) pathfinding. *Medium*, May 2020.

[3] Amit J. Patel. Introduction to a*. *Amit's Thoughts on Pathfinding*, 2020.

[4] Omar Elgabry. Path finding algorithms. *Medium*, Oct 2017.

[5] Written by: Milos Simic. Tracing the path in dfs, bfs, and dijkstra's algorithm. *Baeldung on Computer Science*, Nov 2022.

[6] Raymond Kim. Bfs and dfs - path finding algorithms. *Raymond Kim BFS DFS Pathfinding Algorithms*, 2020.

[7] TheHappieCat. How to do pathfinding: The basics (graphs, bfs, and dfs in unity). *YouTube*, May 2017.