Navigation of Mobile Robots

By: Keezhan Hamasoor, Jared Peterson, Priscilla Bunday

December 18th, 2024

In today's age, many people witness mobile robots in their everyday lives, whether outdoors acting as a delivery robot, or inside acting as a waiter. However, the question is how? A mobile robot navigating its environment can be a difficult problem to approach, especially when avoiding obstacles. Multiple factors have to be taken into account, such as various sensors, making it dynamic in unknown environments, but more specifically, what algorithm to choose to navigate obstacles. Depending on what algorithm one picks for their mobile robot, it can drastically change the way the robot navigates through obstacles. One notable factor to take into account is efficiency and speed. This paper explores different approaches to search based on A*, using heuristics such as Manhattan distance and Euclidean distance. Furthermore, this paper looks at how a priority queue or stack approach for these algorithms will increase the performance of the robot's mobile navigation skills in a maze world. The results reveal that a stack-based approach is the most efficient way for all three algorithms tested.

1 Short Description of Problem

The problem that this paper is trying to solve is mobile robots and path navigation, specifically when there are obstacles. This problem was inspired by the Starship Delivery Robots as seen on campus. These mobile robots have to navigate through many obstacles, such as college students, vehicles, and bicycles, all while following traffic rules. However, the overlying question is how these mobile robots safely navigate their obstacles to reach their destination. This is what this paper is trying to solve, and the reason why this is so interesting is because of how complex this problem can be. Multiple factors go into solving this problem. One of them is the external aspects of the robot, such as the sensors mounted on them (infrared, ultrasonic, laser range, touch, and camera sensors) to detect its environment and surroundings /citeKeezhanPaper1. Another big aspect of this problem is which algorithm to use, such as A*, and Dijkstra. However, it is important to choose the right approach depending on the problem, which makes this problem even more interesting. A risk if choosing the wrong approach will result in "high computational power" being produced for the "navigation algorithms" [12]. This also comes with a high risk of energy consumption due to the continuous computation when the robot navigates its environment [12]. Another risk is the "excessive amount of data" that needs to be "interpreted" from the algorithms [12]. Lastly, picking the right algorithm where it will allow the robot to reach its destination optimally, efficiently, and fast is important. Of course, there are ways to combat this depending on the approach one takes. Using reinforcement learning and machine learning approaches are ways to approach this problem, along with using an A* search algorithm. The reason why A* is so efficient is because it uses heuristics to prioritize the most optimal and efficient path depending on the problem. It is also a fairly simple algorithm to implement, which results in lower computational requirements compared to other complex algorithms. Additionally, data structures can be used to optimize path-finding algorithms for mobile robots. However, a concern for this can be allocating memory for each new node in an A* search, which is time-consuming [3]. Allocating a large array of nodes before starting the program can help combat this [3]. In deciding to use a priority queue or stack approach, utilizing a stack approach will allow the results to be "twice as fast as using priority queues" [3]. The approach of using a stack-based or priority-queue approach is interesting because it highlights how the choice of a data structure can change how efficient and complex the algorithm can be. This is especially important for path-finding algorithms that a mobile robot, such as the Starship delivery robot, where it is crucial for deliveries to be delivered on time. Different data structures such as priority queues and stacks can cause different time complexities for operations such as insertion and extraction, which can overall affect how fast the path is found. Because of these factors, this paper explores different search algorithms such as A*, manhattan distance, and Euclidean search algorithms. Additionally, this paper looks into how a queue or stack-based approach for these algorithms will increase the performance of the robot's mobile navigation skills. Because robot mobile navigation is complex, this research has scaled down the problem to creating a maze map where the robot has a starting position and a destination to analyze which algorithm and whether a stack or queue approach will lead to the most efficient and optimal solution. There were 10 trials for each algorithm and the map was generated by using a recursive backtracking algorithm on a 401x301 pixel grid to generate the maze. Typescript was chosen due to it being able to add type definitions on top of JavaScript, which was simpler and easier to use due to it being able to warn and show the user if there are errors, while JavaScript alone would not give enough detailed information to fix any error warnings. TypeScript compiles to JavaScript, which results in the benefits of using a web-based language due to generating a clear image of the robot navigating through the different path-finding algorithms, and allowing simple deployment, allowing anyone able to run the software they currently have installed.

2 Background

To explore mobile robot navigation, it is important to look into the background and different approaches to this problem.

In "Navigation of Mobile Robots in the Presence of Obstacles," they explore the task of a navigation system where the robot is guided to its goal while avoiding obstacles. In general, robots gather their information about their surroundings through sensors mounted on them (such as infrared, ultrasonic, laser range, touch, and camera sensors) [1]. Obstacle avoidance is divided into two categories, "global path planning" and "local motion planning" [1]. Global path planning is where the environment surrounding the robot is known in advance, and local motion planning is more dynamic where the robot can navigate obstacles with much less prior knowledge.

In Ozdemir's and Tuncer's "Navigation of autonomous mobile robots in dynamic unknown environments based on dueling double deep q networks," they explore the navigation of robots based on "dueling double deep Q networks (D3QN)" [12]. This approach is the most optimal when it comes to "Autonomous navigation capabilities" [12] so that the robots can safely reach their goal. The D3QN algorithm was "introduced by Lopez-Martinez" that was structured on using "Double DQNs" and "Dueling DQNs as a hybrid" [12]. This approach is proven to be more successful than using those algorithms separately, which allows for "location nor map information" to not be needed to avoid obstacles [12]. Instead, the robot independently learns about its environment. When they tested their algorithm, the results showed that the algorithm performed well and was able to learn "from scratch in unknown environments" and "detect predefined targets" [12] which is important in mobile robot navigation. This approach was considered in this paper's testing, but overall was not chosen due to wanting to take a different approach from machine and reinforcement learning. Additionally, it was out of scope for this paper due to none of the members in this group having vast experience with this topic.

Another approach for the path-planning of mobile robots is topological navigation, which is explored by Vengatesan and Vasudevan in their paper. Topological navigation uses the environment's "natural structure" to "direct the path planning process" of the robot [2]. The robot is able to navigate "effectively" because of this graph representation, and able to "concentrate on high-level decisions" instead of precise calculations [2]. This is important because sometimes focusing on precise calculations can slow down the performance of the robot. The authors approach was utilizing topology with using Dijkstra's shortest path algorithm. The results showed that the use of topological with an a* approach is efficient, where the landmark recognition range was "98" percent but increased to "99" percent [2]. Additionally, the landmark's identification range was "99" percent but increased to "99.5" percent. Lastly, the "full image detection of vehicles" only took "0.3" seconds, and "localization estimation" only took "0.002" seconds [2]. This approach was also not used due to it being out of scope and real-life images not being used for this research.

Another algorithm for path tracking mobile robots are A* and Dijkstra. In Zixuan's study, the use of Dijkstra in mobile robot path planning was used in a map of "100 pixels" with red lines denoting as "non-passable regions" [4]. However, the issue with the Dijkstra algorithm was that if there were an "abundance of obstacles," then the speed of calculations would experience a "significant slowdown" [4]. To address this, A* was used. In addition to using A*, a SLAM "(Simultaneous Localization and Mapping)" algorithm was used [4]. This helps locate the robot's position that does so by initializing the "laser's sensors" and using the sensor data to "calculate" the robot's position [4]. However, the A* algorithm could only avoid fixed obstacles and tracking. If the obstacle's position was placed in advance, the robot would fail to avoid it, especially in "crowded areas" and "complex traffic situations" [4]. This approach was used in this paper, where an A* search algorithm with a heuristic of zero was used, and instead of Dijkstra, euclidean search was used to combat the significant slowdown due to the abundant of obstacles.

Additionally, efficient movement is an obvious characteristic of any robotic system. Ideally, a minimum-cost path would be followed to conserve the most resources. The now-ubiquitous A* algorithm which can be adapted for path planning was introduced in 1968 in "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", combining aspects of previously used mathematical and heuristic methods into one efficient algorithm [9]. The paper first introduces the mathematical concept of a graph with nodes, connected by edges with different costs, along with the notion of start and goal nodes and a definition for

optimal path cost, meaning no other path can have a lower cost or be more efficient [9]. They are primarily concerned with admissible algorithms, which are guaranteed to find an optimal path. Their A* algorithm is described in only 4 steps, and they go on to prove various properties about it. Namely, they use an evaluation function at each node to provide a better estimate of distance to the goal rather than brute force traversing every single node. Provided that an estimating function $\hat{h}$ used for ordering nodes does not overestimate the remaining optimal path, they prove that A* is actually globally optimal [9]. They go on to note how although any choice of $\hat{h}$ that doesn't overestimate the optimal cost can be used to provide an optimal solution, it may still be useful to choose something besides Euclidean distance, which is calculated using multiplication and a square root. In some cases this may be an expensive calculation, so a simpler estimate of summing the differences on each axis and dividing by two (an efficient bit shift for integers) will still provide optimal results [9]. Finally, they explain how an algorithm that overestimates may not find an optimal solution, but as long as $\hat{h}$ isn't too far off the mark, the solution will still be very efficient and may be found much faster than using Euclidean distance as a heuristic or by using something that uses division [9].

While A* is optimal when the entire environment is known ahead of time, this isn't always a realistic scenario. Objects may not be where a map says they are, or paths may be blocked off. Or a robot may be dropped in a new environment and must rely entirely on its sensors to navigate. These issues were recognized by Anthony Stentz in 1994 in his paper "Optimal and Efficient Path Planning for Partially-Known Environments", in which he described an algorithm known as D* to work around them, for "dynamic A*" [14]. The primary difference from the original A* algorithm is that edge costs in the underlying graph may change over time. This is accomplished mostly by functions called PROCESS-STATE and MODIFY-COST [14]. They are used in tandem to calculate path costs, similar to A*, but also to update the costs. He goes on to give examples of operation for different information sources. A complete map gives a globally optimal solution, never needing to update costs, while a limited radius vision sensor in the same environment is only slightly worse. Similarly, a coarse map of the environment combined with a vision sensor produces a solution close to the global optimum [14]. He goes on to informally prove that these solutions are optimal for the information given to the algorithm, and experimentally compares them to a truly optimal algorithm which replans the entire route as soon as new information is received, showing that all results are the same [14]. This opens up many possibilities for application, letting robots explore new terrain efficiently [14].

When real world constraints apply, some shortcuts may need to be taken. In a fully known environment with many grid cells, it may consume too much time to calculate an optimal solution that steps through one node at a time. To counter this issue, the authors of "Enhanced Robot Motion Block of A-Star Algorithm for Robotic Path Planning" introduce their idea of a Robot Motion Block (RMB) [10]. An RMB is effectively 8 nodes that a robot may move to next (up and left, up, up and right, right, right and down, down, left and down, left). Traditional A* only moves 1 node at a time (n = 1), while an RMB with alternative cost functions may be used to skip evaluating several nodes as long as the path along the same direction is clear (n = 1, 2, 3, ...) [10]. They show that on experimental maps of size 261x261, 462x261, and 462x462, the RMB algorithm with optimally chosen n yields only a slight increase in path cost while searching far fewer nodes and completing the search in a fraction of the time [10]. However, care must be taken when choosing n for an environment, as too high a value may result in no solution being found because a robot may not think it can move between small gaps [10]. Overall this paper presents a useful technique for reducing computational cost while maintaining a respectable path length.

Another important factor to consider is that mobile robots go through three phases, which are "perception, path planning and synthesis, and motion" [13]. It is also important for the robot to pick the most optimal route while avoiding obstacles. A way to navigate this is "reinforcement learning (RL)" [13]. Reinforcement learning allows to "identify routes" based on this history of the robot's past behavior [13]. It also goes through a reward/punishment system depending on the actions it takes, where the higher the points, the robot will adjust its technique so that it can score higher again in the future. Two distinct learning techniques are "Monte Carlo" and Temporal-Difference (TD)" [13]. These two techniques allow "full flexibility of data" and for large use of memory, which is beneficial in a robot's navigation process [13]. Additionally, the robot has to be trained for the reinforcement learning process. This was done so by placing the robot in "several unpredictable scenarios" called "episodes" [13]. Every episode assumes a goal. The more episodes, the more the robot will be "skilled" and "dynamic" [13]. In Ravi's paper, his use of RL

techniques resulted in "better performance," "speed mobility," "substantial target-reach success rate," and "greater extrapolation" relating to "diverse motion situations and surroundings" such as incoming traffic and pedestrians.

Machine learning, while harder to implement, has been used as a more computationally effective and dynamic way of dealing with robot path planning [11]. Neural networks are a core part of the machine learning-focused heuristic approaches to robot path planning and can be composed of several variations of neural networks. Neural networks are categorized to deal with "(i) interpreting the sensory data, (ii) obstacle avoidance, and (iii) path planning" [11]. Neural networks work well in non-static environments and can navigate environments with both moving targets and obstacles. One of the methods that does this is the generation of Recurrent Neural Networks that focus on learning navigation skills and localization ability [11]. Other ways of utilizing neural networks include developing principal component analysis networks (unsupervised linear procedure) and multilayer perceptions [11]. Other machine learning methods discussed in "Heuristic approaches in robot path planning: A survey" included neuro-fuzzy logic, which includes elements of neural networks as a preprocessor for fuzzy logic controllers [11]. These alternative methods of heuristic approaches using machine learning can deal with nonlinear mappings and learn better than classical methods, but they involve large numbers of parameters and require significant training to use.

An important risk to be considered in mobile robot path planning is the "high computational power" being produced for the "navigation algorithms" [12]. This also comes with a high risk of energy consumption due to the continuous computation when the robot navigates its environment [12]. A way to combat this is choosing the right algorithms, such as "Dijkstra, A*, and sampling-based planning algorithms such as the rapidly-exploring random tree (RRT) and the probabilistic road map method (PRM)" [12]. Another risk is the "excessive amount of data" that needs to be "interpreted" from the algorithms [12]. Another way to handle this is through a "reinforcement learning (RL)" approach [12]. It is machine learning where it creates a reward-punishment system so that the robot interacting with the environment will take the most efficient action [12].

The focus of the project is finding heuristics that increase the efficiency of path-planning algorithms in robots. The paper, "Evolutionary Heuristic A* Search: Pathfinding Algorithm with Self-Designed and Optimized Heuristic Function" is an exploration of A* heuristics in evolutionary searches. They used a genetic algorithm to optimize the heuristic functions they were using for the A* search, the Evolutionary Heuristic A* search, which effectively worked to create a multi-weighted heuristic function that was selected after iterative sequences of selections, mutation, and crossover evolution on the population of agents [15]. This process allowed them to take admissible heuristics and then iteratively adapt them to be more accurate to the problem. This approach is a subfield of machine learning and showed success at finding optimal solutions to a multiple grid-based pathfinding problem with significant improvements to computational costs and memory efficiency compared to the conventional A* search they tested against [15]. They tested three different models for this, a base model, a Parallel Islands Model, and a Re-Initialization Model; each of these used slightly different methods to evolve the heuristic and created solutions to the benchmark test that had no differences in optimal path length. The Re-Initialization Model was the fastest at optimizing the heuristic however.

Another approach to A* heuristic optimization is based on the article "Time-efficient A* Algorithm for Robot Path Planning" where the heuristic is only used when the robot is about to collide with an object [8]. The basis for this optimization is that the heuristic is one of the most computationally expensive parts of the A* algorithm, so the overall computational complexity of the algorithm can be limited by relying on the simple straight-line distance between the robot's position and the goal until it is about to hit an object. This point is the "switching phase" and the new total path from the starting location to the final goal becomes the addition of the straight line distance from the start to the switch spot and the optimal path from the switch spot to the goal found by the A* algorithm [8]. This process was tested in a simulation in MATLAB and was more time efficient than the original A* algorithm with a 65% reduction in processing time. There was minimal effect on the optimal path length between the two kinds of algorithms so this approach is a comparable way to decrease time complexity.

In "On the Heuristics of A* or A Algorithm in ITS and Robot Path-Planning," the authors looked at Dijkstra's algorithm, another computationally complex algorithm, the A algorithm that does not use admissible heuristics, and several more optimized kinds of heuristics for the basic A* algorithm [7]. The

additional heuristics they considered both involved the Euclidean distance, d(n), between any arbitrary node and the goal node. For the first algorithm, the heuristic is $\hat{d}(n) = d(n) * r(d)$ where r(d) = "((Euclidean distance along the optimal path from one present node n of all start nodes to the goal node) / (Euclidean distance of straight segment be- tween them))" [7]. The second heuristic is $h(d) = r(d) * d(n)$ where $r(d) = 0.01 \log(d) + 1.2$, a heuristic they found to be near optimal. Both of these require pre-computation in order to find the optimal path and then use that to define the optimal heuristic, but they were both more efficient while still selecting paths that were not significantly longer than the optimal path. During this analysis the authors tested on numerous different kinds of graphs, including extremely large graphs, and different kinds of maps and digital configuration spaces.

"Optimizations of data structures, heuristics and algorithms for path-finding on maps" investigated a combination of improvements to heuristics and the data structures that the problems are modeled on. They focused on improving the Manhattan Distance heuristic in two different variations, the ALT and ALTBestp heuristics [3]. The traditional Manhattan Distance Heuristic calculates the difference between the distances of the current position from the goal position with horizontal, vertical, and sometimes diagonal distances (depending on if the implementation allows for diagonal distances). The variations calculate the distances between each node being evaluated, the goal node, and each point in a collection of pre-computed point distances using the triangle inequality to find the shortest distance [3]. The ALTBestp heuristic follows this principle but decreases the complexity by only selecting the point p with the highest heuristic value instead of going through every point p for the distance calculations. This optimization makes it faster than the ALT heuristic while still giving it better accuracy than the Manhattan heuristic. Overall, the ALT heuristic has the highest accuracy but the slowest computation time. All three heuristics are admissible so the biggest limitation is the computational and memory needs for the more accurate heuristics. The authors also explored different ways of setting up nodes on their map implementations and compared the common priority queue implementation with an array of stack implementations. They found that the array of stacks was faster for managing the nodes in the A* searches they were testing while still being implemented in a reasonable chunk of memory [3].

Although not necessary for real-world applications, procedural world generation can be helpful for testing a model against many unknown data sets to evaluate its performance. In the context of a grid world, maze-generating algorithms can create a random environment that is hard to navigate [6]. Many such algorithms exist, and Peter Gabrovšek provides a helpful overview of 6 of them in "Analysis of Maze Generating Algorithms" [6]. The 6 algorithms are recursive backtracking (modeled after depth-first search), Aldous-Broder, Wilson's Algorithm, Prim's algorithm (follows from breadth-first search), Hunt and Kill, and Kruskal's Algorithm. Several of these are algorithms from graph theory repurposed to generate mazes [6]. Along with the mazes, 4 agents are used to analyze the difficulty of the mazes: a random walk agent, depth-first search, heuristic depth-first search (biased to choose nodes closest to the goal), and breadth-first search [6]. The analysis focuses on several factors: the generating time, a number of intersections (nodes with more than two adjacent nodes), and a number of dead ends, in which Prim's algorithm was ranked first on all fronts and recursive backtracking was ranked last [6]. The agents were then used to compare the number of steps taken, visited intersections, and visited dead ends. By averaging those respective rankings among all agents, it was found that Aldous-Broder produces the most difficult mazes on average, while recursive backtracking produces the easiest [6]. This data can be used to choose how mazes are generated based on the desired characteristics of the environment for an agent to be tested on.

When a maze is not realistic enough, other approaches need to be taken to generate an environment, as shown in "Space Colonisation (sic) for Procedural Road Generation". Traditional approaches adopt techniques from biology introduced by Aristid Lindenmayer called L-systems. L-systems, quite literally intended to model biology, can generate tree structures such as leaves, ferns, and branches [5]. More recent work in biology may use the Space Colonisation (sic) Algorithm (SCA), an alternative to L-systems that can also generate trees. It differs from L-systems by allowing for attraction points that guide the growth of the structure in certain directions [5]. The paper explores related works, many of which use output from L-systems to then generate a road network. The authors give an overview of the SCA, which uses a set of attraction points and initial nodes to then build a tree. This growth can be controlled by several parameters, such as the radius of influence for attraction points, and the kill distance to get rid of nodes [5]. Specifically for road generation, the SCA is used in an initial phase to generate an initial tree guided

by attraction points, which could be random or represent population centers. The initial SCA parameters influence the final shape, such as capillarity (how many roads there are), angle constraints (turning angles), road hierarchy (highways, city streets, side streets, etc.), and snapping and merging (not putting roads too close together) [5]. These are all existing work, but the innovation comes in the second phase of the road generation algorithm, wherein roads are connected to each other, breaking the tree structure. More attraction points may be used to guide this process, and angles are also taken into consideration to avoid triangular intersections. These techniques already produce realistic road maps, but the authors further explore how more information throughout the whole process can affect the results. Areas may be effectively blocked off to create parks, and vortexes can produce a circle of roads around an area [5]. The final result could then be discretized onto a grid world for a path planning algorithm to explore to test robot navigation in a more realistic environment.

3 Approach to Solve Problem

The approach for solving the problem involved two different levels of analysis: the implementation of two different data structures to manage the nodes for the A* search, as well as the implementation of three different heuristics that could be used for the A* search. This section will start with the analysis of the data structure code before going into the heuristics that were used. Below is the code for the stack-based approach:

```
class StackAstar:
    public search(): Node | null {
        this.stack.push(this.start);
        this.nodes_added += 1;
        while (this.stack.length > 0) {
            const n = this.stack.pop();
            this.nodes_removed += 1;
            if (n === undefined) continue;
            n.closed = true;
            if (n.x == this.goal.x && n.y == this.goal.y) {
                return this.goal;
            } else {
                // Weird looking code compared to a priority queue A* but it seems to work
                const neighbors = this.neighbors(n);
                for (const neighbor of neighbors) {
                    if(n.currentMinCost + 1 < neighbor.currentMinCost) {
                        neighbor.currentMinCost = n.currentMinCost + 1;
                        neighbor.parentIdx = n.y * this.grid.width + n.x;
                    }
                    if (!neighbor.closed) {
                        neighbor.f_hat
                        neighbor.parentIdx = n.y * this.grid.width + n.x;
                        neighbor.currentMinCost = n.currentMinCost + 1;
                        neighbor.f_hat = this.f_hat(neighbor);
                    }
                }
                neighbors.sort((a, b) => {
                    if (a.f_hat < b.f_hat) {
                        return -1;
                    } else if (a.f_hat == b.f_hat) {
                        return 0;
                    } else {
                        return 1;
                    }
                });
                neighbors.reverse();
                for(const neighbor of neighbors) {
                    if(!neighbor.closed) {
                        this.stack.push(neighbor);
                        this.nodes_added += 1;
                    }
                }
            }
        }
        return null;
    }

    private neighbors(node: Node): Node[]:
        const ret: Node[] = []
        if (this.boolGrid.isValid(node.x, node.y - 1) && this.boolGrid.get(node.x, node.y -
     1)):
            ret.push(this.grid.get(node.x, node.y - 1))
        if (this.boolGrid.isValid(node.x, node.y + 1) && this.boolGrid.get(node.x, node.y +
     1)):
            ret.push(this.grid.get(node.x, node.y + 1))
        if (this.boolGrid.isValid(node.x - 1, node.y) && this.boolGrid.get(node.x - 1, node
    .y)):
            ret.push(this.grid.get(node.x - 1, node.y))
```

```
        if (this.boolGrid.isValid(node.x + 1, node.y) && this.boolGrid.get(node.x + 1, node
    .y)):
            ret.push(this.grid.get(node.x + 1, node.y))
        return ret
```

Listing 1: Stack-Based Approach

The analysis of the code for the stack-based approach is standard. The algorithm starts by pushing the start node to the stack. While the stack is not empty, it pops a node until the goal is reached. If not, it gets the neighbors and ensures it is not blocked by an obstacle (in this case, the maze's wall) and either goes up, down, left, or right. Additionally, if a node is popped from the stack, it is marked as closed so it does not visit the path again. The best case for this algorithm is O(1) if the goal is found right away. However, this is usually not the case. The average runtime can be O(n), where it entirely depends on how big the world is, and where/how far the goal is.

Here is the code for the queue-based approach:

```
public search(): Node | null:
        this.start.f_hat = this.f_hat(this.start)
        this.minHeap.insert(this.start)
        while (this.minHeap.size() > 0):
            const n = this.minHeap.extractMin()
            n.inMinHeap = false
            if (n.x == this.goal.x && n.y == this.goal.y):
                n.closed = true
                return this.goal
              else:
                n.closed = true
                const neighbors = this.neighbors(n)
                for (const neighbor of neighbors):
                    if (n.currentMinCost + 1 < neighbor.currentMinCost):
                        neighbor.currentMinCost = n.currentMinCost + 1
                        neighbor.parentIdx = n.y * this.grid.width + n.x
                    const original_f_hat = neighbor.f_hat
                    if (!neighbor.closed):
                        neighbor.f_hat = this.f_hat(neighbor)
                        if (!neighbor.inMinHeap):
                            neighbor.inMinHeap = true
                            this.minHeap.insert(neighbor)
                    else if (neighbor.closed && this.f_hat(neighbor) < original_f_hat):
                        neighbor.closed = false
                        neighbor.f_hat = this.f_hat(neighbor)
                        if (!neighbor.inMinHeap):
                            neighbor.inMinHeap = true
                            this.minHeap.insert(neighbor)
        return null
```

Listing 2: Queue-Based Approach

This algorithm starts by adding the start node to a minimum heap. The minHeap acts as a priority queue implementation because the node that will be popped from the heap will have the minimum cost to get to the goal state. This implementation prioritizes nodes that have the best costs and orders them as such. While the heap is not empty, the minimum node from the heap is extracted and it is set to not be in the heap. The node is checked to see if it is at the goal and true is returned and the node is closed if it is. If not, then the node is closed and each of the neighbors of the node are evaluated to see if they should be placed on the heap.

The computational complexity of this approach is higher than a stack due to using a priority queue, which was implemented with a min heap. Rather than O(1) append and remove last, each insertion and deletion is $O(\log n)$ in the size of the queue. Once a lot of nodes have been added, the overall complexity becomes $O(n \log n)$, much slower than a stack's $O(n)$ complexity. This produces a more efficient solution at the cost of more computation time.

Additionally, here is a snippet of the manhattan distance algorithm this paper used for its problem:

```
const manhattan_times: number[] = []
for (let i = 0; i < NUM_TRIALS; ++i):
```

```
    const m = new Maze(TRIAL_WIDTH, TRIAL_HEIGHT)
    m.backtrack(1, 1)
    // White out top left and bottom right
    for (let y = 0; y < 5; ++y):
        for (let x = 0; x < 5; ++x):
            m.cells.set(x, y, true)
            m.cells.set(m.cells.width - 1 - x, m.cells.height - 1 - y, true)
    // Remove walls by random chance
    for (let y = 0; y < m.cells.height; ++y):
        for (let x = 0; x < m.cells.width; ++x):
            if (Math.random() < 0.2):
                m.cells.set(x, y, true)
    // Change AStar to StackAstar to switch to a stack implementation
    const astar_manhattan = new AStar(m.cells, (n, goal) =>:
        return Math.abs(n.x - goal.x) + Math.abs(n.y - goal.y))
    const start = performance.now()
    astar_manhattan.search()
    const end = performance.now()
    manhattan_times.push(end - start)
```

Listing 3: Manhattan Distance Approach

The astarManhattan function represents the Manhattan Distance heuristic to reach its goal.

```
const zero_times: number[] = []
for (let i = 0; i < NUM_TRIALS; ++i)
    const m = new Maze(TRIAL_WIDTH, TRIAL_HEIGHT)
    m.backtrack(1, 1)
    // White out top left and bottom right
    for (let y = 0; y < 5; ++y):
        for (let x = 0; x < 5; ++x):
            m.cells.set(x, y, true)
            m.cells.set(m.cells.width - 1 - x, m.cells.height - 1 - y, true)
    // Remove walls by random chance
    for (let y = 0; y < m.cells.height; ++y):
        for (let x = 0; x < m.cells.width; ++x):
            if (Math.random() < 0.2):
                m.cells.set(x, y, true)
    // Change AStar to StackAstar to switch to a stack implementation
    const astar_zero = new AStar(m.cells, (n, goal) =>:
        return 0
    const start = performance.now()
    astar_zero.search()
    const end = performance.now()
    zero_times.push(end - start)
```

Listing 4: Zero Heuristic Approach

The Zero Heuristic is just the A* algorithm with an h value of 0.

```
const euclid_times: number[] = []
for (let i = 0; i < NUM_TRIALS; ++i):
    const m = new Maze(TRIAL_WIDTH, TRIAL_HEIGHT)
    m.backtrack(1, 1)
    // White out top left and bottom right
    for (let y = 0; y < 5; ++y):
        for (let x = 0; x < 5; ++x):
            m.cells.set(x, y, true)
            m.cells.set(m.cells.width - 1 - x, m.cells.height - 1 - y, true)
    // Remove walls by random chance
    for (let y = 0; y < m.cells.height; ++y):
        for (let x = 0; x < m.cells.width; ++x):
            if (Math.random() < 0.2):
                m.cells.set(x, y, true)
    const astar_euclid = new AStar(m.cells, (n, goal) =>:
        const xDiff = n.x - goal.x
        const yDiff = n.y - goal.y
        return Math.sqrt(xDiff * xDiff + yDiff * yDiff)
    const start = performance.now()
```

```
    astar_euclid.search()
    const end = performance.now()
    euclid_times.push(end - start)
```

Listing 5: Euclidean Distance Approach

The Euclidean Distance gives the straight-line distance to the goal node from the current node.

Each of the three algorithms was tested in 10 trials with a new maze map generated each time. The function AStar represents the queue-based approach, while switching it to StackAstar will provide the stack-based approach. Lastly, the start and end variables record the time it takes for the algorithm to navigate to its goal.

4 Description of Design and Experiments

This project had a few major design components: the generation of the maze, the data structures used for A* searches, the implementations of the A* searches, and the different heuristics the A* searches used.

Before even implementing anything, the language to use was considered. Python allowed for a quick test, but for benchmarking many runs would have been too slow. Java ran fast, but was difficult to get visualizations working for. Finally, a middle ground of JavaScript was decided on to get the best of both worlds. As a scripting language it can be written flexibly, while the JIT in most browsers has been optimized enough to allow for fast benchmarking. Because it was built for the web, visualization is simple whether via the browser console, modifying the DOM, or rendering to a Canvas element.

The generation of the maze was initially supposed to be based on road generation [5], but the procedure for connecting roads didn't leave enough detail to be implemented. Instead, a 2-step process was used. First, a random maze was created out of a graph of nodes using a recursive backtracking algorithm [6]. Then, to make an obstacle problem rather than a maze solving problem, walls were randomly removed with a configurable 20% probability.

The data structures for A* searches were primarily simple grids of cells and a priority queue. The grids were just arrays indexed by width and height. The priority queue was a standard implementation based on a min heap for ordering nodes. A* search was then implemented using a Node class, which had light optimization such as only keeping a node's parent index, not an actual reference, to avoid work for the garbage collector. The $\hat{f}$ function from A* was used to order Nodes in the min heap. $\hat{f}$ is the sum of the current minimum cost found and a heuristic estimating distance to the goal, such as Manhattan distance, Euclidean distance, or no estimate at all (0).

The choice of data structures was influenced by the work done in the paper "Optimizations of data structures, heuristics and algorithms for path-finding on maps" where an array of stacks implementation of the A* search was compared with a priority queue implementation [3]. The priority queue was chosen because it is a very common way of implementing A* searches. Because the goal is to find the nodes that will have the shortest path to the goal node, it makes sense to have a priority queue that will return the minimum distance node each time you pop from it. Priority queues have an O(logn) time for popping the top element off of the queue, which makes it efficient for this application. (Jared add more about this)

The paper originally proposed using an array of stacks as an alternative data structure to more efficiently process nodes for the A* search, but for the final implementation in this project a few modifications were made [3]. The actual implementation instead utilizes a single arrary of Nodes where nodes are added to the stack as they are found and are popped off according to a Depth-First Search algorithm. The stack does not use an ordering function (the heuristic) in order to prioritize which nodes are popped off first, which is different from how the original paper used an f-ordering system for their array of stacks [3]. This was done to investigate the baseline effects of using a stack approach; in the future, further optimizations of the stack ordering should be done to determine if greater time savings would be made.

Three heuristics were chosen to be investigated for this project: the Manhattan distance, Euclidean distance, and the zero heuristic or Dijkstra's algorithm. These three algorithms were chosen because of their versatility for this kind of graph exploration problem and their wide applications. The Manhattan distance in particular was chosen because of its effectiveness on problems with grid-based structures. The distance from the goal node was evaluated by finding the distance in grid squares in the up-down and

left-right directions. It has a very close estimate for the actual distance to the goal node while still being admissible. The way that the heuristic is calculated also closely mimics the way that the path can actually be taken which makes it more accurate to the real path length. This design should in theory provide the best analysis time across both data structure implementations because it will give the A* algorithm a good idea of which nodes to choose next. The Euclidean distance algorithm calculates the straight-line distance from the current node to the goal node. This makes it admissible, but it will generally be a significant underestimate of the true distance to the goal because it does not take into account the obstacles in the way and the way that the path moves will actually be made. Because diagonal moves are not valid in this grid world the Euclidean path could never be truly followed. This heuristic is expected to perform worse than the Manhattan heuristic, but it may be comparable to Dijkstra's algorithm for analysis. The zero heuristic is the A* algorithm without a heuristic or h value which turns the algorithm into Dijkstra's algorithm. Dijkstra's algorithm is admissible and will find the optimal solution for a problem, but it is generally slower because it goes through all possible paths with equal weight. These three algorithms are commonly compared when assessing the effectiveness of A* searches so this project was an effort to benchmark these heuristics while also assessing the impact of different data structures.

The final results for path length and analysis time are shown in the table below. These results are the averages across trials on 10 randomly generated 200x150 mazes, represented on a 401x301 grid. The path lengths were different across the two different data structures, but similar across the heuristics. The average times for analysis are shown in milliseconds for each heuristic and data structure.

Table 1: A* Search Result Averages for Stack and Priority Queue Data Structures

	Stack		**Priority Queue**	
Heuristic	**Analysis Time (ms)**	**Path Length**	**Analysis Time (ms)**	**Path Length**
Euclidean	0.5	1148	58.2	717
Manhattan	1.9	1306	22.7	717
Zero	4.2	2310	47.3	717

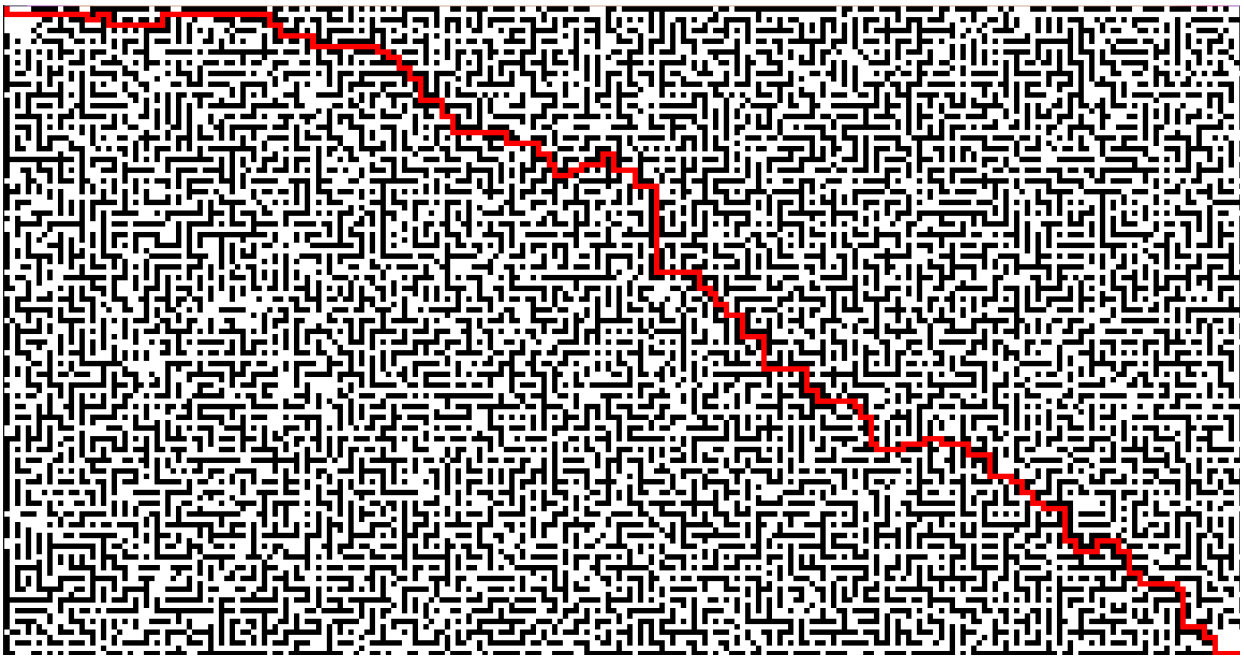

Figure 1: A* using Manhattan distance as heuristic

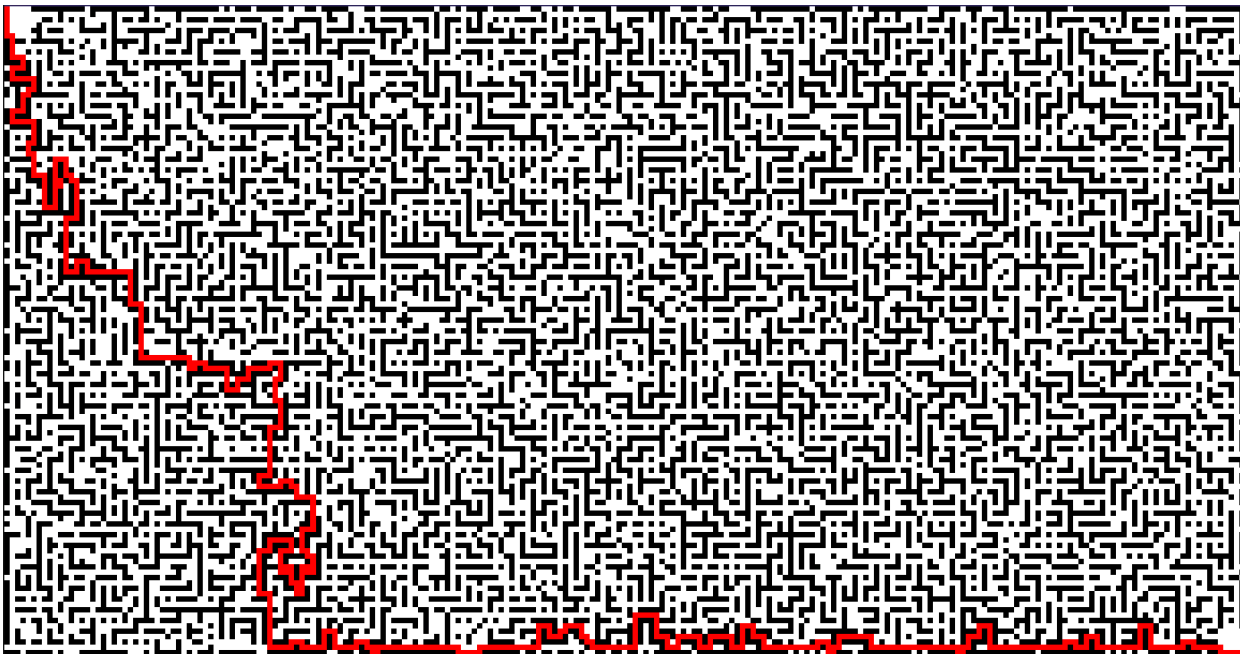

Figure 2: Stack using Manhattan distance as heuristic

5 Analysis

Looking at the analysis time for each of the heuristics first, there are no clear trends that match across both data structures. The times for the priority queue match the original assumptions for how the heuristics should evaluate the path search where the Manhattan is the fastest, and the zero and Euclidean are fairly close to each other. The zero heuristic ended up being faster than the Euclidean distance for the priority queue which could have been due to the Euclidean distance being a significant underestimate of the true distance to the goal when factoring in the maze obstacles. The Manhattan distance performed better than the zero because it was a much more accurate estimate of the true distance and took the ways the search could move through the maze into account, and it also didn't have to explore every possible path equally like the Dijkstra's algorithm (zero heuristic) did. These results were as expected for the priority queue, and because this is the standard implementation for most A* searches it makes sense that these line up with results seen in other studies. The results varied more for the stack implementation because the Euclidean distance performed the best with the Manhattan following and the zero at the worst time. The times might have varied here because the stack itself was not ordered by priority so the differences in heuristics could have been influenced by what nodes happened to be popped first during that search. The times for the stack implementation were significantly shorter than the priority queue with time improvements by 10 fold. One reason is that the stack itself is much faster to handle and no ordering of nodes had to be done. The stack being much faster than the priority queue follows the results that were found for an array of stacks [3]. This implementation was different than what was done here, but the use of an array for a stack still followed the same principle.

An interesting difference is seen with the path lengths. Within the priority queue implementation the path lengths were all the same. For normal A*, all heuristics are admissible and find the optimal path so it makes sense that all three heuristics for the priority queue implementation would end up having the same path length. What is interesting is that the stack implementation there are significantly longer path lengths and they vary more between heuristics. This could be due to how nodes are added to the queue and the lack of ordering. The neighbors order is not shuffled for this implementation like it is for the priority queue where the best neighbor will always be chosen first. This means more nodes have to be gone through to find the solution when compared to the priority queue. The length of the paths within

the stack implementation mostly make logical sense. Dijkstra's algorithm evaluates all paths equally so it makes sense it would go through more nodes as it finds its way to the goal node and the path length is the longest. The Euclidean distance and Manhattan distance are relatively close to each other in path length, but the Manhattan distance could have taken longer because even though it should be a better distance evaluator it depended on which nodes were evaluated.

The connections between path length and evaluation time for the stacks make sense; the longer the path length is the longer the analysis time is. The priority queues had the same path lengths so the analysis time was dependent on the heuristics' effectiveness.

6 Conclusion

The final results from this experiment provide both insight into how different heuristics can be tested for efficacy and how the choice of data structure impacts the efficiency of the search. The increased speed of analysis from the stack based approach suggests that when this project is scaled for larger implementations it would be important to explore more ways to optimize the use of stacks. This also sheds light on the results of other surveys on heuristics that were based on a priority queue structure and their overall significance. For future work, implementing this algorithm with a real life robot would be interesting to see. That way, external factors can be taken into consideration, such as real life obstacles and how the obstacle's size and shape can affect the robot and its path to reach its destination. Additionally, it would be interesting to see how moving obstacles can affect the robot. In this project, the obstacles did not move (maze walls). By including real-life moving obstacles, such as humans and cars, it would be interesting to see how the algorithm recalculates to account for the new position of the obstacle. Further extensions through this would ideally use the stack based approach for the data structures and the Manhattan distance heuristic for best performance. This exploration into the different data structures and heuristics used for robot path planning was informative and provided hands-on work for implementations of common graph making and artificial intelligence algorithms.

7 Division of work

Everyone edited the paper. Keezhan worked on the abstract, short description, brief review, pseudocode for the stack-based approach, and wrote the code for the StackAstar approach and the approach section for Stack-Based and Manhattan code. Priscilla worked on the pseudocode and explanations for the queue-based approach and heuristics, the Description of Design and Experiments, Analysis, and Conclusion. Jared wrote the code for the queue-based approach and the maze generation, and worked on the approach explanation for the queue-based approach and Description of Design and Experiments.

8 Use of AI

Overleaf's built in grammar assistance was used to edit this paper. Grammarly was also used.

References

[1] Abiyev, R., Ibrahim, D., and Erin, B. Navigation of mobile robots in the presence of obstacles. *Advances in Engineering Software 41*, 10 (Oct. 2010), 1179–1186.

[2] Arumugam, V., and Algumalai, V. Topological Navigation of Path Planning Using a Hybrid Architecture in Wheeled Mobile Robot. In *Advances in Artificial Intelligence and Machine Learning in Big Data Processing* (Cham, 2025), R. Geetha, N.-N. Dao, and S. Khalid, Eds., Springer Nature Switzerland, pp. 32–44.

[3] Cazenave, T. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *2006 IEEE Symposium on Computational Intelligence and Games* (May 2006), pp. 27–33. ISSN: 2325-4289.

[4] Chen, Z., Li, H. J., and Liu, Y. Path tracking robot based on A* and Dijkstra. *AIP Conference Proceedings 3144*, 1 (June 2024), 030013.

[5] Dias Fernandes, G., and Fernandes, A. R. Space Colonisation for Procedural Road Generation. In *2018 International Conference on Graphics and Interaction (ICGI)* (Nov. 2018), pp. 1–8.

[6] Gabrovšek, P. Analysis of maze generating algorithms. *IPSI Transactions on Internet Research 15*, 1 (2019), 23–30.

[7] Goto, T., Kosaka, T., and Noborio, H. On the heuristics of A* or A algorithm in ITS and robot path-planning. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)* (Oct. 2003), vol. 2, pp. 1159–1166 vol.2.

[8] Guruji, A. K., Agarwal, H., and Parsediya, D. K. Time-efficient A* Algorithm for Robot Path Planning. *Procedia Technology 23* (Jan. 2016), 144–149.

[9] Hart, P. E., Nilsson, N. J., and Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (July 1968), 100–107. Conference Name: IEEE Transactions on Systems Science and Cybernetics.

[10] Kabir, R., Watanobe, Y., Islam, M. R., and Naruse, K. Enhanced Robot Motion Block of A-Star Algorithm for Robotic Path Planning. *Sensors 24*, 5 (Jan. 2024), 1422. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

[11] Mac, T. T., Copot, C., Tran, D. T., and De Keyser, R. Heuristic approaches in robot path planning: A survey. *Robotics and Autonomous Systems 86* (Dec. 2016), 13–28.

[12] Ozdemir, K., and Tuncer, A. Navigation of autonomous mobile robots in dynamic unknown environments based on dueling double deep q networks. *Engineering Applications of Artificial Intelligence 139* (Jan. 2025), 109498.

[13] Raj, R., and Kos, A. Intelligent mobile robot navigation in unknown and complex environment using reinforcement learning technique. *Scientific Reports 14*, 1 (Oct. 2024), 22852. Publisher: Nature Publishing Group.

[14] Stentz, A. Optimal and efficient path planning for partially-known environments. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation* (May 1994), pp. 3310–3317 vol.4.

[15] Yiu, Y. F., Du, J., and Mahapatra, R. Evolutionary Heuristic A* Search: Pathfinding Algorithm with Self-Designed and Optimized Heuristic Function. *International Journal of Semantic Computing* (Apr. 2019). Publisher: World Scientific Publishing Company.