ROBOTICS: EXPLORING PATH FINDING ALGORITHMS

ILYAS IBRAGIMOV
NIKOLAOS MANGINAS

I. INTRODUCTION

The aim of this report is to offer an in depth exploration of various path finding algorithms. Although the context of this report is primarily confined to robotics applications, these are general path finding algorithms with further applications in game theory, data structure searching, and many other domains. The algorithms to be explored are the *Best-First Algorithm*, *Dijkstra's Algorithm*, and the *A* Algorithm* implemented with various heuristics. We leverage these famous computer algorithms in an attempt to find the shortest path between two points in a known grid representation of a map with various obstacles.

The grid representation of the map has a finite number of discrete states. The set of all the discrete states is denoted as X_m and if x is a state in the world then it must be that $x \in X_m$. We can denote each cell, representing a state, as a set of Cartesian coordinates. So say we have a 5×5 map, then X_m has a total of $5 \times 5 = 25$ discrete states. Within X_m each state x can be denoted as a set of Cartesian coordinates, i.e. $x = (r, c)$ where r denotes the row and c the column of the cell. Therefore, we can represent the states within the the state-space as $x \in X_m = \{(0,0), (0,1), ..., (rows, columns)\}$ where $rows$ and $columns$ are predefined. We now have a unique set of states, x, in the state-space, X_m.

The goal is to get from a source point, $x_i \in X_m$ to a target point $x_G \in X_m$. The objective then becomes finding the possible set of target actions U_n which will cause a transition from state x_i to state x_G with intermediate states x_{inter}. For simplicity, we have designed the simulations where the agent can only take a discrete number of actions, U. The actions that the agent can take in a given state x are given by $U(x)$, since different states allow for different actions. The collection of all possible actions U is defined as movement to any adjacent cell. In the case where we want to implement only orthogonal movement, U can be limited to $U = \{\text{UP}, \text{RIGHT}, \text{DOWN}, \text{LEFT}\}$. The set of legal actions within a state is a subset of the total collection of actions. That is, $U_x \subseteq U$. Given a state $x \in X_m$ and an action $u \in U_x$, there exists a function $f(x, u)$ that can map x and u to the next state x', yielding $x' = f(x, u)$. If a path exists between two points in a map there must be a collection of actions $U_n = \{u_i, u_{i+1}, ..., u_n\}$ for which that path can be executed. We can define:

$$f_i(u_i) = f(x_i, u_i)$$
$$f_{n+1}(u_{n+1}) = f(u_n{+}1, f_{n-1}(u_{n-1})$$

This recursive definition allows us to prove the statement above. If there exists a path between to points x_G and x_i, then there is a sequence of actions U_n for which $f_n(u_n) = x_n = x_G$.

The task therefore reduces to searching the state space to find whether there exists a path between the source and target points, and if so determine the optimal path between them. Note that many algorithms do not search the entire state space, rather they limit the number of cells searched based on an estimated cost function which determines which cells are prioritized during searching. This estimated cost function is known as a heuristic, which gives an approximate minimum cost to move from any point to the goal. Depending on the heuristic measure used, too few cells may be searched and the algorithm may not find the real optimal path. Therefore there is a trade-off between search efficiency and guaranteeing we obtain the best path [1]. The different forms a heuristic can take and the compromise between speed and accuracy will be discussed further in the report alongside the A* and Greedy algorithms.

II. ALGORITHM ANALYSIS

This section outlines the methodology used in each algorithm as well as critically evaluates the performance of the algorithms. All algorithms use a priority queue. A priority queue is a data structure used for ordered storage. The place in the queue in which a new element is inserted depends on what the queue is ordered by. In this case the queue for all three algorithms is ordered depending of the score of a cell, which differs between the algorithms. Therefore adding a new element, $E = (10.2, (2, 2))$ to a priority queue $Q = [(12.3, (0, 0)), ((14.2), (1, 1))]$ will result in $Q' = [(10.2, (2, 2)), (12.3, (0, 0)), ((14.2), (1, 1))]$, since $10.2 < 12.3 < 14.2$.

III. DIJKSTRA'S ALGORITHM

Given a starting node, Dijkstra's algorithm works by iteratively searching the entire state space for optimal paths between the starting node and all other nodes, or until it reaches a pre-defined goal. Specifically, Dijkstra's first searches for all the neighbouring nodes of the starting node, adding these to a priority queue sorted by their distance to the starting node. Then the algorithm repeats this process by working through the queued nodes and adding *their* neighbours to the queue. As it is working through each node, the path with the lowest cost from the source to that node is recorded. If a goal is defined

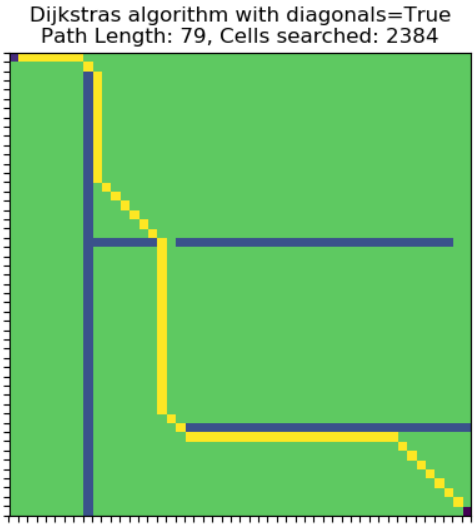


Fig. 1: Dijkstra's algorithm search results from (0,0) to (49,49). The obstacles are represented by blue squares, the searched cells are coloured green and the final path found is in yellow. The optimal path of length 79 was found.

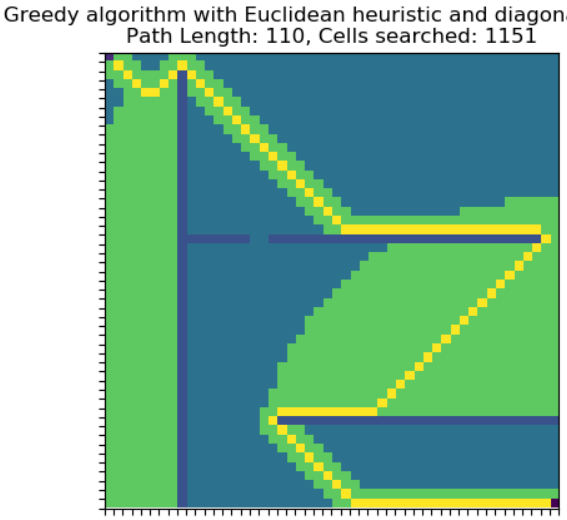


Fig. 2: Greedy algorithm search results from (0,0) to (49,49) with Euclidean distance to goal as the heuristic. The light blue cells are unsearched. A sub-optimal path of length 110 was found after searching 1151 cells.

then the process ends when the goal node is reached. Although computationally inefficient, Dijkstra's algorithm is guaranteed to find the shortest path between any two nodes, given a path exists between them. An example of Dijkstra's algorithm is shown in figure 1. Notice that every cell is searched in order to find the shortest path.

IV. Best-First Algorithm(Greedy)

Dijkstra's Algorithm is expensive both computationally and time-wise. The Best-First Algorithm tries to limit the search performed, however it is bound to only arrive at locally optimal solutions. Hence, although Greedy algorithms search a much smaller part of the state space, and are therefore faster, the final path yielded is not always optimal. Greedy Algorithms use a heuristic to limit their search. A heuristic, in essence an abstract measure of cost, classifies the cells to be searched depending on their distance to the goal. Hence not all cells are searched. Instead every time-step the cell searched is closer to the goal than any other cell that can be searched. More on the different types of heuristics will be discussed alongside the A* algorithm. However it should be noted that, in contrast to Dijkstra's and A*, Greedy Search does not take distance from the source to the cell in to account when ordering the queue of cells to be searched. Rather, it only considers the estimated distance from the cell to the goal given by the heuristic. Referring to the pseudo code for A* in Algorithm 1, to alter the program to have a greedy approach we replace:

$$fScore[child] \leftarrow gScore[child] + \textbf{\textit{heuristic}}(child, map)$$

with:

$$fScore[child] \leftarrow \textbf{\textit{heuristic}}(child, map)$$

Reiterating that in each time step the algorithm searches the cells that are estimated to be closer to the goal first by selecting the cell with the smallest heuristic estimate. This 'greedy' ordering of the search queue is what causes Best-First searches to generally search less of the state space when finding a solution. In many cases, Best-First and Dijkstra's searches can yield the same final path, even though Dijkstra's is much more computationally expensive. On the other hand, although Dijkstra's searches much more, it is guaranteed to find the optimal path every time, in contrast to the Best-First algorithm. When performing a greedy search on the same map as figure 1, we see a good example of the trade-off between search time and better pathing. The results are shown in figure 2, where although roughly half the cells are searched, the final path length was drastically longer at 110 cells compared to a length of 79 found by Dijkstra's.

V. A* Algorithm

Building upon both prior described algorithms,the A* algorithm is essentially a generalized version of Dijkstra's and Best-First algorithms. [4] It orders the nodes to be searched in the priority queue by the lowest estimated cost to reach the goal through the node (i.e. the heuristic). This lowest estimated cost for node n, $f(n)$, is calculated from $f(n) = g(n) + h(n)$, where $g(n)$ is the exact cost to reach the node from the start, and $h(n)$ is the heuristic estimated cost to reach the goal from node n [3]. When $h(n) = 0$, the A* algorithm behaves identically to Dijkstra's, basing its search order only on $g(n)$. This is demonstrated visually in figure 3, where the results are identical to Dijkstra's algorithm.

The three most common heuristic measures used for h(n) are Euclidean, Octile, and Manhattan distance to the goal. Depending on the heuristic measure used, A* can still be guaranteed to give the shortest path while being more efficient than Dijkstra's. In order to be certain a heuristic will give the shortest path, it must be admissible. If the heuristic never overestimates the cost to reach the goal from the current cell, then it is considered admissible. Further, a heuristic may be

Algorithm 1 A* Algorithm [5]

```
function EXPLORE_MAP(source, target, map)
    Q ← []
    gScore ← {}
    fScore ← {}
    prev ← {}
    gScore[source] ← 0
    fScore[source] ← heuristic(x, map)
    Q.put((fScore[source] , source))
    terminate ← False
    while Q.empty() != True and not Terminate do
        parent ← Q.get()[1]
        for child in neighbours(parent) do
            tentative_gScore ← gScore[parent]+distance_between
            if tentative_gScore < gScore[child] then
                gScore[child] ← tentative_gScore
                fScore[child] ← gScore[child] + heuristic(child, map)
            end if
            if heuristic(child, map) then
                Terminate ← True
            end if
            prev[child] ← parent
            Q.put((fScore[child] , child))
        end for
    end while
end function
```

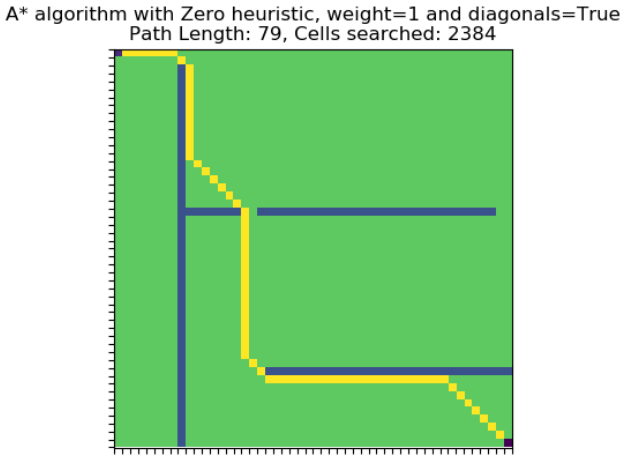


Fig. 3: A* algorithm search results with a constant zero as the heuristic. This is identical to results with Dijkstra's algorithm.

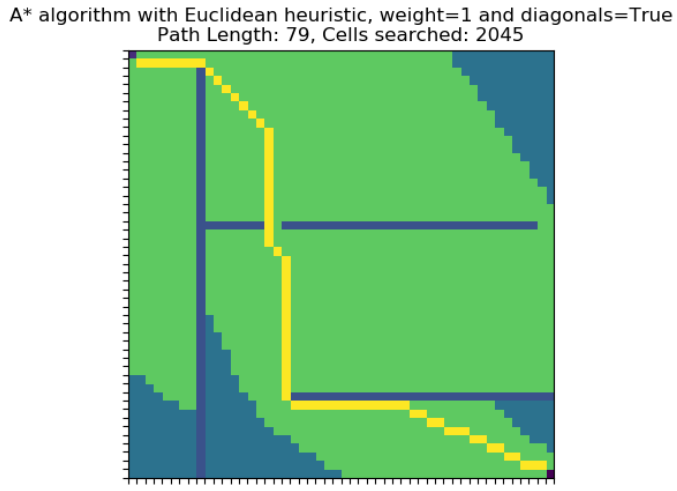
categorized as consistent. This means that the heuristic never overestimates the cost to move to a neighbouring cell added to the estimated cost of reaching the goal from that neighbouring cell. All consistent cells are also admissible, but not vice-versa. The best heuristic to use for A* depends entirely on the scenario, and whether speed or accuracy is more important.

The heuristic discussed above will be analyzed here. Euclidean distance simply refers to straight line distance between the node and the goal, i.e.

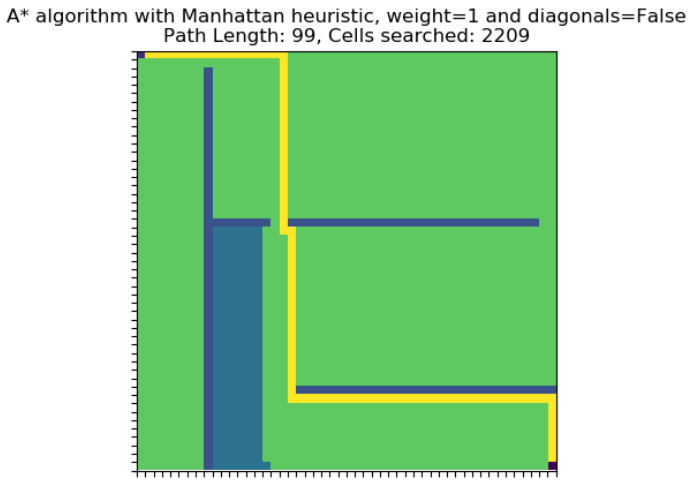
$$d(n,g) = \sqrt{(g_1 - n_1)^2 + (g_2 - n_2)^2}.$$

Since it measures distance in a direct, straight line to the goal, Euclidean distance is a particularly useful heuristic when any direction of movement is allowed [1]. An example is shown in figure 4a, where although fewer cells are searched compared to Dijkstra's, it still finds the optimal path. Manhattan distance differs because it measures distance based on orthogonal grid movement, $d(n,g) = |n_1 - n_2| + |g_1 - g_2|$.Therefore when there is only orthogonal movement, Manhattan distance is usually used as the heuristic. Shown in figure 4b is the shortest orthogonal path found with the Manhattan heuristic. However, it would not be appropriate to use when there is diagonal movement along the grid. Rather, octile distance, the distance to the goal based on diagonal grid movement, would be more suitable. The octile distance formula takes the form $d(n,g) = (dn + dg) + (\sqrt{(2)} - 2) * \min(dn, dg)$ where $dg = |g_1 - g_2|$ and $dn = |n_1 - n_2|$ [1]. A different, but still optimal path found with the Octile heuristic is illustrated in figure 4c. Note that there were less cells searched compared to the Euclidean heuristic, but the path lengths are the same.

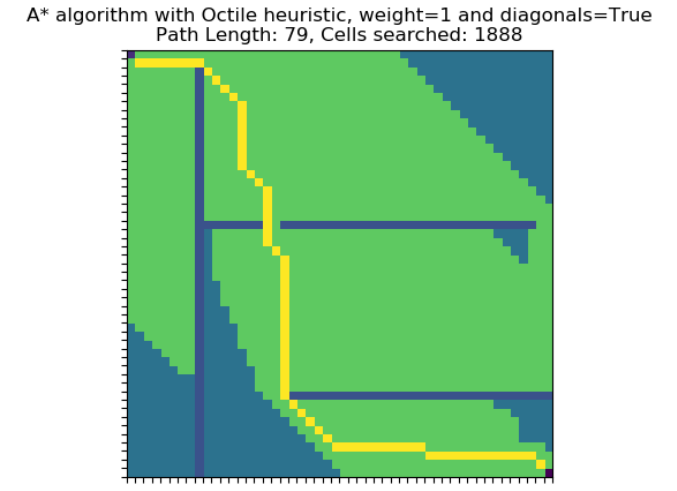
Further expanding on the A* algorithm, the heuristic measure h(n) can also be weighted. By multiplying h(n) by a weight greater than 1, we make h(n) dominant when calculating f(n). When a very large weight is applied, then g(n) becomes negligible when calculating f(n), essentially turning A* into greedy best-first search. After applying this weight, our heuristic is no longer admissible as it now overestimates the cost to get to the goal. An illustration of the effect a weighted heuristic can have on the A* algorithm is shown in figure 5. With a weight of 10 applied to the Euclidean



(a) A* algorithm search results with Euclidean distance to goal as the heuristic. Finds the optimal path length of 79.



(b) A* algorithm with only orthogonal movement with Manhattan distance to goal as the heuristic. Finds the optimal path length of 99 when no diagonal movement is allowed.



(c) A* algorithm search results with Octile distance to goal as the heuristic. Finds the optimal path length of 79.

Fig. 4: A* search results with various heuristic measures. Each sub-figure shows the algorithm found the optimal path.

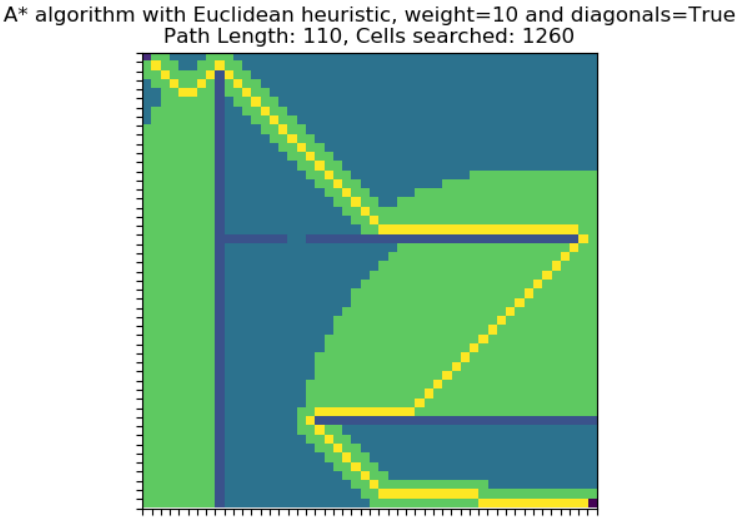


Fig. 5: A* algorithm search results with Euclidean distance to goal multiplied by 10 as the weighted heuristic. These results are similar to what greedy best-first search would output because the heuristic $h(n)$ is dominant over the real cost from the source to the cell $g(n)$.

heuristic, the results become almost identical to those from greedy best-first search, with a sub-optimal path of 110 being selected again. This and the previous examples should demonstrate the flexibility of the A* algorithm.

A* performs very differently depending on the choice of heuristic. We know that Dijkstra's will always yield optimal paths, and A* using a heuristic $h(n) = 0$ reduces to Dijkstra's. Therefore, the question becomes whether A* can use a heuristic different than $h(n) = 0$ to limit the search, while being guaranteed to yield the same(optimal) path. Indeed, A* when equipped with an admissible heuristic is guarantee to yield the optimal path. We can refer above, as to what an admissible heuristic is, but it basically is any heuristic that never overestimates the distance to the goal. Given a grid map, in which we allow diagonal movement the only heuristic excluded is Manhattan distance since it can overestimate the distance to the goal. Given that with either Octile or Euclidean Distance as a heuristic the search is limited while the final path generated is maintained optimal it must be that A* with Euclidean or Octile Distance is better performing than Dijkstra's. In the examples presented above we can see that A* with Octile and Euclidean give final paths of 79 just as Dijkstra's. This map is generally a good representation of a typical map. Therefore, we should calculate the total amount of cells visited and the execution time, which should be correlated, to determine which of the three heuristics performs best. The results can be seen below: This of course, is dependent on various things including, the hardware it is run on for the time, and most importantly the map. However, there is no doubt as to the fact that generally Euclidean and Octile will give results faster. From the two Euclidean performs better here and generally will and therefore is probably the best heuristic to use in A* search.

Table 1: Heuristic Comparison		
Heuristic	Cells Searched	Time(s)
0	6182	0.39
Octile	4741	0.32
Euclidean	4466	0.27

TABLE I: Heuristic Performance

VI. STDR Integration

This section will outline the integration of the algorithms in a simulation environment called STDR. STDR is a part of the Robotics Operating System(ROS) and is used for robotics simulations. STDR can be loaded with different maps and robots to fit the simulations needs.

In this case we use a map resembling a factory warehouse and a simple robot and try to optimize the robot's pathing as it moves to predefined waypoints in the state space. In order to assess the performance of the robot's pathing we will use three metrics. Distance, will refer to the total distance traveled by the robot during simulation time, total runtime will refer to the total time the robot requires to move from start to goal and though the waypoints and total angle turned will refer to the total amount of degrees the robot turned during simulation. We will use these three metrics to construct a quantitative analysis of the simulation, using all the algorithms described in the first part of the report and using an improved controller which we implemented. We will also perform a comparison between the default controller and the one we constructed using a fixed choice of algorithm and heuristic. Such an analysis will be definitive as to the performance of each algorithm for the given state space.

Our original implementation of the robot controller was very simple and very slow. The controller was given a path calculated by the robot planner, and each graph node in that path was given as a waypoint for the robot to drive to. Before actually moving the robot anywhere, the controller first checked the angle difference between where the robot was facing and where it needed to turn to face the next waypoint. The controller then turns the robot at a speed that decreases linearly as the angle difference decreases to make sure we do not overshoot. Once the robot is facing the correct direction for movement, it was given a positive variable speed that decreases as it approaches the waypoint. The controller repeats this turn-move process for each waypoint the robot was given. Finally, when it reaches the last waypoint (i.e. the goal) it turns itself, once again at variable speed, to the final orientation specified. Figure 6 shows the path taken by the robot to get to one of the waypoints in the factory scenario. Every single node on that path is passed as a waypoint to the robot. Using this implementation of the controller and A* with a Euclidean heuristic as the planner, running the entire set of waypoints for the factory scenario took 954 seconds. Additionally, it traveled a total distance of 157.8 meters and turned a collective 68.3 rad.

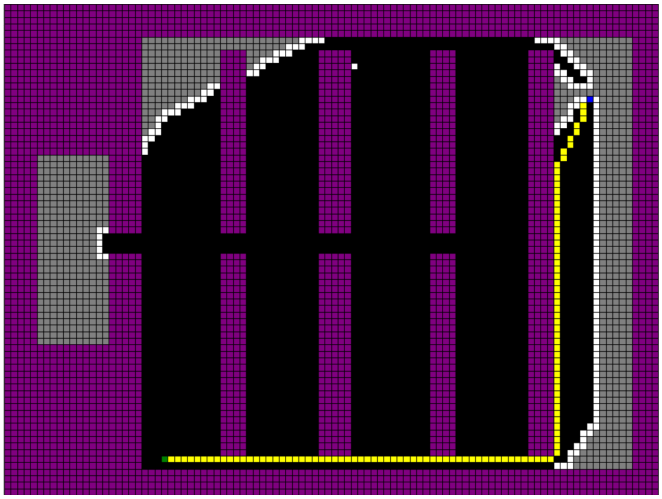


Fig. 6: The path taken by the robot to navigate to one of the waypoints specified in the factory scenario. Path found with A* algorithm with Euclidean heuristic. Yellow represents the path, purple are the obstacles, black are searched nodes, and gray are unsearched.

We were able to drastically improve the performance of our robot as it drives to new waypoints with a couple of key changes to the robot controller. One of the largest bottlenecks on the efficiency of the robot as it followed a path was the number of waypoints it had to go though. Prior to our changes to the controller, the robot kept slowing down for every single point in the path. We found many of these waypoints were unnecessary. For example, if we have the robot going in a straight line, each point in that line would be a waypoint, while we really only needed to consider the first and last points on that line as waypoints to achieve the same result. To fix this, our first improvement allowed the controller to check if there are multiple waypoints along the path in a straight line, and if so it would ignore the unneeded waypoints between the first and last on the line. This sped up the robot significantly as it no longer had to slow down for each point in the path. It could now travel at full speed for significant distances.

Although we achieved great results with our first approach, we noticed another inefficiency in driving the robot to the goal with our algorithms and controller. The various algorithms that have been discussed in this paper are limited to a discrete set of directions of movement along the grid. The path we generate and feed into the robot controller only allows for directly diagonal and orthogonal movement, despite our robot being able to move in any direction. Theta* is an algorithm that is very similar to A*, except it is able to consider paths at almost any direction of movement[6] [7]. Theta* lets the parent of a node be any other node, whereas A* only allows the neighbours of a node to be its parent[6]. Theta* uses a line of sight function, that checks if one node is visible from another node, to determine which path segments it can condense to a straight diagonal line (in any direction) by changing certain nodes' parents.

We used the same line of sight function and principles as

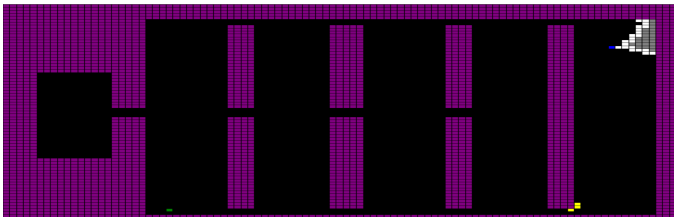
Theta* to tell our robot which waypoints it can safely ignore. As the controller iterates backwards through the generated path by A*, it checks the furthest node along the path that is within the line of sight of the current node. It then ignores the nodes between them, sending the robot only the furthest node within its line of sight as a waypoint. Unfortunately, when implemented in STDR, there were some edge cases when the robot was traveling very close to obstacles that were interpreted as crashes into the obstacles. To solve this we had to also add the second furthest node within the line of sight as waypoint as a buffer for the robot around obstacles. This strategy allows us to use as few as three waypoints for some paths in the factory scenario.

Using both aforementioned changes to the controller, we recorded its performance during the factory scenario with various combinations of algorithms and heuristics shown in table 2. When using A* with a Euclidean heuristic as we did prior to implementing our improvements, our robot only took 138 seconds to finish navigating through all the waypoints. This is an almost seven fold improvement compared to the 954 seconds it took before. Not only did the time decrease, but the distance traveled and total angle turned did too. This is because we now take the (almost) exact shortest path due to our second change, which allows our path to be at any angle. The performance of the Greedy Algorithm is worse then any other,since we only concern ourselves with the time taken by the robot to execute the path and not the time it takes to generate said path, since it is nominal in comparison.

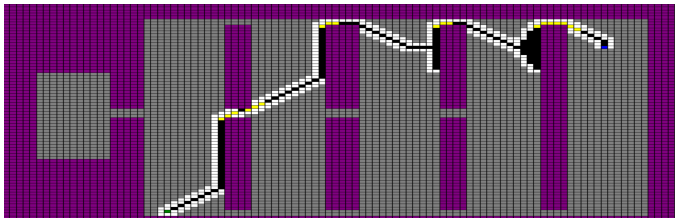
Table 2 Algorithm Performance				
Algorithm	Heuristic	Distance(m)	Time(s)	Angle(rad)
Dijkstra's	–	133.14	140	35.23
Greedy	Euclidean Distance	140.15	221	52.11
Greedy	Octile Distance	139.90	218	52.90
Greedy	Manhattan Distance	140.11	217	52.17
A Star(A*)	Euclidean Distance	132.54	138	36.64
A Star(A*)	Octile Distance	135.31	178	44.24
A Star(A*)	Manhattan Distance	133.16	129	34.11

TABLE II: Algorithm Performance

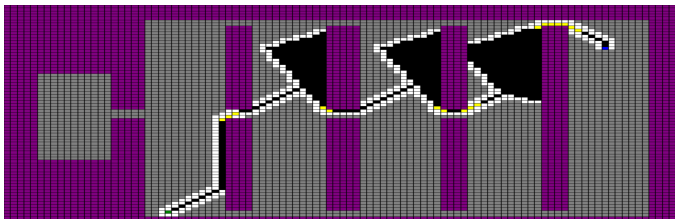
The new path implemented by the robot, with a reduced number of waypoints allowed for higher performance and its difference from the previous approach is apparent when comparing figure 6 with the sub-figures of figure 8. Figure 8 shows the robot navigating to the second waypoint as before, however now only a few waypoints represent the entire path.



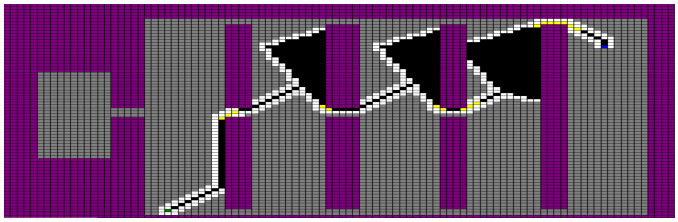
(a) Dijkstra's



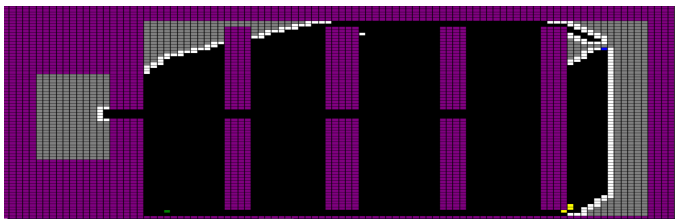
(a) Greedy Algorithm with Euclidean



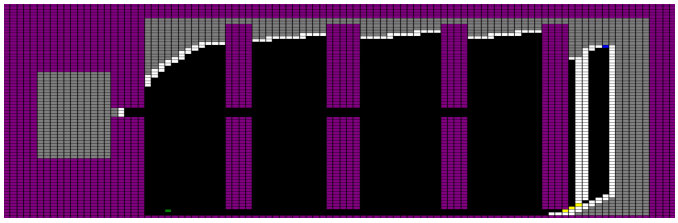
(b) Greedy Algorithm with Octile



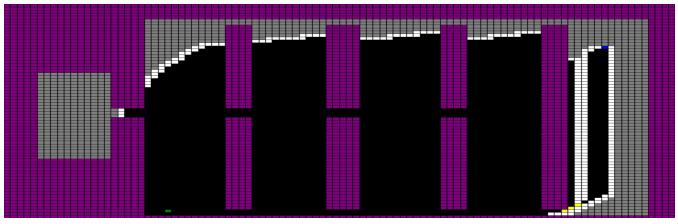
(c) Greedy Algorithm with Manhattan



(d) A* with Euclidean



(e) A* with Manhattan



(f) A* with Octile

Fig. 8: Adapted Paths

REFERENCES

[1] Patel, Amit. “Amit’s Thoughts on Pathfinding: Heuristics.” Stanford.edu, theory.stanford.edu/ amitp/GameProgramming/Heuristics.html.

[2] Tarau, Paul. ”Dijkstra’s algorithm.” cse.unt.edu, http://www.cse.unt.edu/ tarau/teaching/AnAlgo/Dijkstra%27s%20algorithm.pdf

[3] Patel, Amit. “Amit’s Thoughts on Pathfinding: Introduction to A*.” Stanford.edu, http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html

[4] ”Introduction to the A* Algorithm.” mnemstudio.org, http://mnemstudio.org/path-finding-a-star.htm

[5] Paul E. Black, ”Manhattan distance”, in Dictionary of Algorithms and Data Structures, Vreda Pieterse and Paul E. Black, eds. 31 May 2006. https://www.nist.gov/dads/HTML/manhattanDistance.html

[6] Nash, Alex. ”Theta*: Any-Angle Path Planning for Smoother Trajectories in Continuous Environments.” aigamedev.com, http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/

[7] K. Daniel, A. Nash, and S. Koenig, ”Theta*: Any-Angle Path Planning on Grids.” arxiv.org, https://arxiv.org/ftp/arxiv/papers/1401/1401.3843.pdf