Robot Motion Planning using TSP and A* Search Algorithm

Leen Ghattas and Ola Ghattas

Boston University College of Engineering
Department of Mechanical Engineering
110 Cummington Mall, Boston, MA 02215

***Abstract*— Five different algorithms that can plan a path for a robot to navigate from a start point to a goal point while passing through multiple predefined locations will be presented in this paper. This problem is similar to the Travelling Salesman Problem which is finding the optimal route that passes through multiple locations only once and returns to the start point. The exact solution to TSP is to generate the sequence that yields the optimal path by calculating the cost of all possible permutations and choosing the one with the minimum cost. However, the cost of the paths is calculated using the Euclidean distance between the points which makes it unreliable in case there are obstacles in the environment. Another possible way to find the route is to use the A* search algorithm since it is guaranteed to get the optimal path between 2 points. However, the robot is required to visit multiple points before reaching its goal, so A* alone is not guaranteed to generate the optimal route due to the fact that it only considers two points at a time. This paper proposes combining both algorithms, TSP and A*, to generate an optimal path passing through all the points once and taking into consideration all the obstacles in the environment. The results will be demonstrated through simulations of the paths generated and the performance of each method will be evaluated based on the computational time, length of the path, and the number of examined cells.**

I. INTRODUCTION

Path planning and trajectory planning are crucial issues in the field of robotics [12]. Path planning is an important primitive that is defined as the process of finding a path for a robot from its initial position to the goal point by avoiding collisions with obstacles or other agents present in its environment [30]. The path is said to be optimal if the sum of its transition costs is minimal across all possible paths leading from the initial position to the goal position [13].

A* search algorithm is one of the best and most popular optimal heuristic searches used in path-finding and graph traversals. The algorithm finds the shortest path, if it exists, between an initial and a final node, and it makes its decision based on an evaluation function, that is the sum of the distance from the initial node to another one, and the heuristic distance from that node to the final node [24], [19]. Heuristic functions can be either consistent or inconsistent [11], but to guarantee that we're getting the optimal path using A* an admissible heuristic function, which doesn't overestimate the cost of the actual path between two points, should be used like Manhattan, Chebyshev and Euclidean [23].

However, in cases where multiple points are required to be visited before reaching the final destination, A* does not always yield the shortest route since it can only compute the shortest path between two points without taking into considerations all the other points that should also be visited. To tackle this issue, the Travelling Salesman Problem (TSP) can be used along with A*. TSP is the challenge of finding the shortest yet most efficient route to take passing through multiple locations only once and returning to the start point [26]. It is an extension of the Hamiltonian Circuit Problem and an NP-Complete problem [5] which implies that a polynomial-time algorithm that can exactly solve the problem in a reasonable amount of time does not exist when then number of locations is large [21].

A. Prior Work

Prior work discussed in this section are are exact and approximate solutions to the Travelling Salesman Problem. To find the find optimal path using TSP, exact algorithms that are typically derived from the integer linear programming (ILP) formulation of the TSP [25] are used when the small number of nodes is small. However, when the number of nodes become larger, the solution of TSP becomes out of bounds of most powerful computers [28]. As a result, researchers have developed several heuristic and approximation algorithms to provide sub-optimal solutions to TSP within a reasonable amount of computing time [15], [20], [22]. One of the oldest and simplest heuristics to implement in solving TSP is the Nearest Neighbor algorithm [3]. It finds the shortest path, not necessarily optimal, by always visiting the nearest unvisited node until all nodes have been reached [4], [17], [18]. Another heuristic was introduced by Christofide in 1976 to solve TSP with a worst-case ratio strictly less than 3/2 [6]. The algorithm involves creating a minimum spanning tree (MST), adding links, creating Euler tour, and removing redundant paths by shortcuts [16]. Furthermore, Neural Networks have been also proposed to solve TSP. JJ Hopfield et al presented a neural network that was capable of solving TSP [10]. In 1998, J. C. Fort found sub-optimal tours for the TSP by using the Kohonen algorithm [14], but that algorithm is not guaranteed to converge, so a the Kohonen Network Incorporating Explicit Statistics(KNIES) was introduced [2] which performed better in finding path for TSP. Moreover, Potvin introduced the usage of Genetic algorithms (GA) to solve TSP and presented various extensions to it [25]. Genetic algorithms (GA) are evolutionary techniques used for optimization purposes according to survival of the fittest idea. The results indicate that genetic algorithms are competitive

with the best-known heuristics for the TSP for medium-sized TSPs with a few hundred cities. However, to achieve good results they require large computation.

The problem of finding optimal routes from a start to the goal passing through multiple points is also tackled in the Vehicle Routing Problem (VRP). VRP is an important optimization problem that is concerned in determining the optimal route that a fleet of trucks should take to deliver goods from a central or multiple depots to a set of customers or demand points [29]. This problem was first introduced by Dantzig and Ramser in 1959 [8] as the truck dispatching problem which is a generalization of the travelling salesman. They added an additional constraint to TSP that limits the number of locations the truck can visit per trip and proposed the first mathematical programming formulation and algorithmic approach. In 1962, Clarke and Wright improved on the Dantzig–Ramser approach by proposing a effective greedy heuristic. They developed an iterative procedure that enables the rapid selection of an optimum or near-optimum route from a start point to a number of locations. [7], [27].

Both TSP and VPR are relevant to the problem tackled in this report however they are not sufficient to solve it due to the presence of obstacles in the environment that are not considered. However these solutions inspire the introduction of another heuristic to solve TSP that accounts for the obstacles.

II. Problem Formulation

The path planning of a robot that has to visit multiple predefined locations before reaching its final destination is discussed in this report. As mentioned in section I, using each of A* algorithm and TSP alone is not sufficient to tackle the problem presented. Each algorithm satisfies one part of our objective so we combined these two algorithms to get the optimal path from a start point to a goal point while passing through multiple predefined nodes and avoiding obstacles. So we proposed 5 different ways of combining these 2 algorithms to do that. These methods were discussed in greater details in the section V and were tested through simulations in different environments and the results obtained were used to evaluate the performance of each method based on the following criterion:

1) Computational Time
2) Length of the Paths
3) Number of Examined Cells

In the environment that we're going to use we placed the predefined nodes that the the robot has to pass through before reaching the goal and split them into two categories A and B, and they were represented as x and o respectively. The robot has two options, either to pass through the points in category A and reach goal A then to pass through the points in category B to reach goal B or pass through the points in category B first before passing through the nodes in Category A.

Since TSP's computational time increases exponentially as the number of nodes increase, we decided to split the route into four different path:

1) Path 1 which starts at the starting point, passes through the points in category 1 and ends at category 1's goal.
2) Path 2 which starts at the stating point, passes through the points in category 2 and ends at category 2's goal.
3) Path 3 which starts at category 1's goal, passes through the points in category 2 and end at category 2's goal.
4) Path 4 which starts at category 2's goal, passes through the points in category 1 and ends at category 1's goal.

The final path is going to be a combination of either Path 1 and Path 3 (Fig. 1) or Path 2 and Path 4 (Fig. 2), whichever give a shorter path.

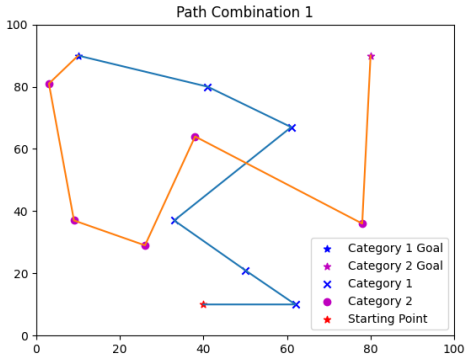


Fig. 1: Combination of Path 1 and Path 3

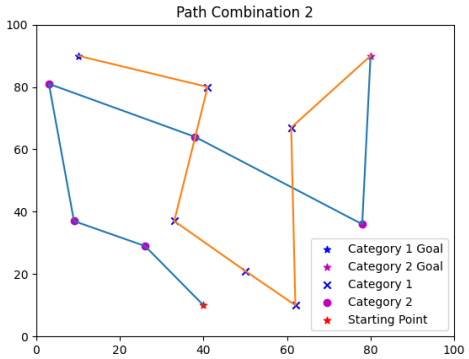


Fig. 2: Combination of Path 2 and Path 4

III. A* Search Algorithm

A* is one of the most successful informed search algorithms used to find an optimal path between nodes or graphs since its flexible and can be used in a wide range of contexts. The algorithm uses a distance-plus-cost heuristic function 1 where G(n) is the actual cost from the initial point to node n and H(n) is the admissible heuristic estimate of the distance from node n to the goal point. To guarantee that we're getting the shortest and most optimal path, we used the admissible Euclidean heuristic function [23].

$$F(n) = G(n) + H(n) \quad (1)$$

A* algorithm begins at the start point n_{start} and then picks the neighbor $n_{neighbor}$, that has the lower cost , which is the estimated $F(n)$, by expanding and evaluating its subsequent nodes and filtering out inaccessible ones. This process is recursively repeated until the shortest path to the goal, n_{goal}, has been found or when O is empty, where O is the list of unvisited nodes [9].

Algorithm 1 A* Search

Require: n_{start}, n_{goal}
$G \leftarrow \{\}$
$F \leftarrow \{\}$
$C \leftarrow \{\}$ ▷ Visited Nodes
$O \leftarrow \{\}$ ▷ Unvisited Nodes
$B \leftarrow \{\}$ ▷ Backpointers
Add n_{start} to O
repeat
  Pick n_{best} from O such that $F(n_{best}) \leq F(n)$ for all $n \in O$
  if n_{best} = n_{goal} **then**
    Exit
  end if
  Remove n_{best} from O and add it to C
  neighbors ← getNeighbors(n_{best})
  for each $n_{neighbor}$ in neighbors **do**
    if $n_{neighbor} \in C$ **then**
      continue
    end if
    cost ← $G(n_{best})$ + getCost($n_{neighbor}$)
    if $n_{neighbor} \notin O$ **then**
      add $n_{neighbor}$ to O
    else if cost $\geq G(n_{neighbor})$ **then**
      continue
    end if
    $B(n_{neighbour}) \leftarrow n_{best}$
    $G(n_{neighbor}) \leftarrow$ cost
    $H \leftarrow$ getHeuristic($n_{neighbour}, n_{goal}$)
    $F(n_{neighbor}) \leftarrow G(n_{neighbor}) + H$
  end for

until O is empty

IV. Travelling Salesman Method (TSP)

The travelling salesman problem is the challenge of finding the shortest route passing through specified locations. TSP belongs to the class of combinatorial optimization problems known as NP-complete [5]. As the number of nodes increases the complexity of the the shortest path increases as well. The exact solution for TSP can be calculated only when the number of nodes is small, otherwise an approximate solution can be calculated using different heuristic and approximation algorithms.

To solve the TSP we used both and an exact and approximate method. In the exact method, we used TSP for each path to generate all the possible permutations and then filtered them to keep the ones that start at the starting point and ends at the goal point. The permutation with the lowest cost was chosen. As for approximate method to solve TSP, the Greedy algorithm was used due to its simplicity. The algorithm always selects the unvisited node having shortest link to the current tour. It is similar to Nearest Neighbor algorithm discussed in the prior work except that it allows to add the new node to both ends of the existing path [1].

Algorithm 2 Travelling Salesman Method

Require: distance_matrix
permutations ← getAllPermutations()
path ← []
minPath ← number of nodes
for each permutation **do**
  if permutation starts at start_point and ends at goal_point **then**
    currentPath ← 0
    for point in permutation **do**
      currentPath ← currentPath + distance_matrix(point, nextPoint)
    end for
    if currentPath ≤ minPath **then**
      path ← permutation
      minPath ← currentPath
    end if
  end if
end for

V. Methods of Combining A* and TSP

As we mentioned previously in section II, the robot can take four different paths and in this paper our goal is to find the combination, Path 1 and Path 3 or Path 2 and Path 4, with the lowest cost that guarantees that the robot pass through all the points in both categories A and B while avoiding obstacles. We implemented five different algorithms that tackle this problem and they're going to be discussed in details below.

A. Method 1

This method is the naive approach and the simplest one among all the methods discussed in this section. We used the exact solution to solve TSP and get the shortest path that goes from the initial to the final position while passing through all the nodes between them only once. After that, we used the A* search algorithm to plot the path between the points while using the same sequence of nodes we got from TSP to guarantee that no collision occurs.

The path obtained from TSP does not necessarily represent the actual path that the robot has to take since it uses the Euclidean distances between the points to calculate the shortest path without taking into consideration the obstacles in the environment. In case an obstacle is present between two successive nodes, A* will lead the robot around the obstacle which results in a longer path, thus making the obtained TSP path unreliable. This inspired the development of the 4 algorithms discussed below.

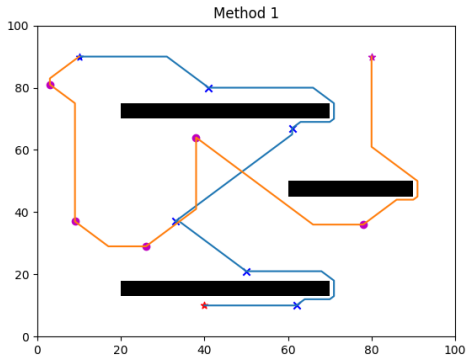


Fig. 3: Method 1 Simulation

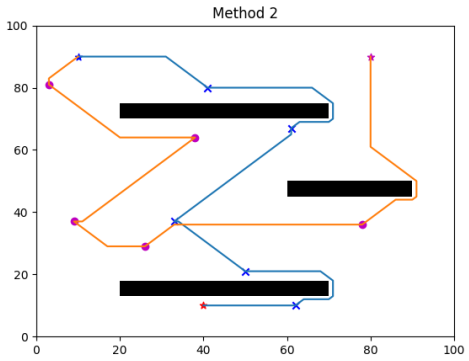


Fig. 4: Method 2 Simulation

Time(s)	Path 1 Length	Path 2 Length	Total Path Length
0.28	181	204	385

TABLE I: Results of Method 1

Time(s)	Path 1 Length	Path 2 Length	Total Path Length
5.93	181	201	382

TABLE II: Results of Method 2

Algorithm 3 Method 1 Algorithm

Require: paths
1: costs ← {}
2: **for** each path in paths **do**
3: distanceMatrix ← getEuclideanDistanceMatrix(path)
4: route, cost ← TSP(distanceMatrix)
5: costs[path] ← cost, route
6: **end for**
7: finalPath, totalCost ← getPathCombinationWithLowestCost(costs)

B. Method 2

In this method, we calculated A* heuristics and passed them to TSP to get the path instead of using the Euclidean distances as we did in Method 1. To guarantee that we do not overestimate the real path length, we used the Euclidean distances as the A* heuristics since they are admissible.

Before we calculated TSP for each path, we calculated the distances between each node in each path using A* and stored the values in a distance matrix. We then calculated the TSP route using the distance matrix obtained. This will result in a better path than Method 1 because it will be calculated based on the real distances, taking into consideration obstacles, between the nodes and setting the sequence accordingly. However, this comes at the cost of computational time since the A* algorithm will be run multiple times and this will increase the code execution time.

This method is expected to find the best path while still executing in a reasonable amount of time. However, this is only true when the number of nodes is low because we used the exact solution to solve TSP.

Algorithm 4 Method 2 Algorithm

Require: paths
1: costs ← {}
2: **for** each path in paths **do**
3: distanceMatrix ← getDistanceMatrixFromAStar(path)
4: route, cost ← TSP(distanceMatrix)
5: costs[path] ← cost, route
6: **end for**
7: finalPath, totalCost ← getPathCombinationWithLowestCost(costs)

C. Method 3

This method is a little different and more complicated than the previous ones. Our aim here is to handle the case where the Euclidean distance between 2 consecutive points in the TSP route is less than their actual distance. This case occurs when there's an obstacle between the two points, so the robot would not be able to take the straight path from one point to the other and is forced to take the longer path to move around the obstacle.

Our proposed solution is to calculate two distance matrices one filled with A* heuristics and the other filled with Euclidean distances and use them to compare the distances between successive nodes in the generated TSP route and the A* heuristics and take actions accordingly.

What we did is that we first used the Euclidean distance matrix to generate the exact path using TSP. Then, we looped over the points where at every point we checked if the Euclidean distance between it and the subsequent one is greater than their corresponding A* heuristics. If so, we replaced the values in the Euclidean distance matrix with the A* heuristics values for the point and its unvisited neighbors and we recalculated the route using TSP starting from that point (Alg 5 lines 11-24).

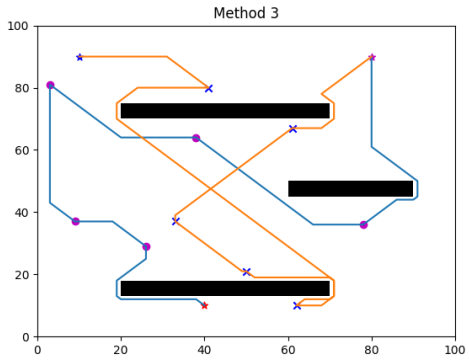


Fig. 5: Method 3 Simulation

Time(s)	Path 1 Length	Path 2 Length	Total Path Length
5.3	232	237	469

TABLE III: Results of Method 3

Algorithm 5 Method 3 Algorithm

Require: paths
costs ← {}
for each path in paths **do**
    AStarDistances ← getDistancesFromAStar(path)
    distanceMatrix ← getEuclideanDistances(path)
    route ← TSP(distanceMatrix)
    finalPath ← route
    totalCost ← 0
    for node in route **do**
        next ← getNextNodeInTSPRoute()
        euclideanDistance ← distanceMatrix(node, next)
        aStarDistance ← AStarDistances(node, next)
        if aStarDistance > euclideanDistance **then**
            distanceMatrix ← Replace euclidean distance with A* heuristic for this node
            updatedRoute ← TSP(distanceMatrix)
            if updatedRoute ≠ route **then**
                Re arrange nodes in the distance matrices
                aStarDistance ← AStarDistances(node, next)
                totalCost ← totalCost + astarDistance
                finalPath ← updateFinalPath()
            else
                totalCost ← totalCost + aStarDistance
            end if
        else
            totalCost ← totalCost + aStarDistance
        end if
    end for
    costs[path] ← totalCost, finalPath
end for
finalPath, totalCost ← getPathCombinationWithLowestCost(costs)

D. Method 4

This method is similar to Method 3, the only difference is how we handled the case when the A* heuristic is greater than the Euclidean distance between 2 consecutive points in the TSP route. Instead of updating the values in the Euclidean distance matrix with A* heuristics and recalculating TSP, we directly got the nearest unvisited node and swapped that with the next node (Algorithm 6 lines 11-23).

This was done in an attempt to speed up the computational time of the algorithm. The motive is to replace the re-computation of TSP by directly choosing the nearest neighbor to the point using A* heuristics. However, the trade-off here is that choosing the nearest neighbor is not guaranteed to result in an optimal route since the position of all the other nodes needs to be considered when choosing the best overall path.

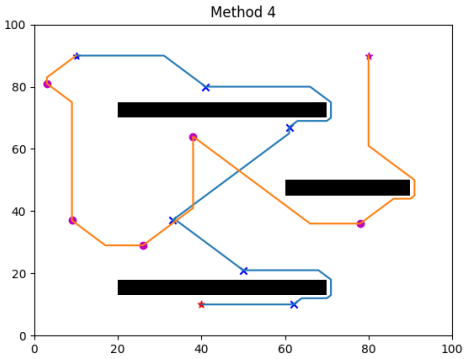


Fig. 6: Method 4 Simulation

Time(s)	Path 1 Length	Path 2 Length	Total Path Length
5.36	282	204	385

TABLE IV: Results of Method 4

E. Method 5

In this method, we decided to use an approximate solution to TSP in contradiction to all the previous methods. Although exact solutions to TSP provide an optimal path, the computational time increases exponentially as the number of nodes increases, as mentioned previously. We added this method to account for cases with a large number of nodes in an environment. There are multiple approximate solutions to solve TSP but the issue is that the goal point cannot be specified. In this method, we chose Greedy TSP as our heuristic and modified it to ensure that the path ends at the specified goal position. Our first approach was to calculate the path and then append the goal point at the end but that resulted in a bad approximation. Instead, we added a dummy node at the end of the distance matrix and we set the distance between the dummy node and both, the start and end points, to zero. This way, we tricked the algorithm to always find

Algorithm 6 Method 4 Algorithm

Require: paths

costs ← {}
for each path in paths **do**
    AStarDistances ← getDistancesFromAStar(path)
    distanceMatrix ← getEuclideanDistances(path)
    route ← TSP(distanceMatrix)
    finalPath ← route
    totalCost ← 0
    for node in route **do**
        next ← getNextNodeInTSPRoute()
        euclideanDistance ← distanceMatrix(node, next)
        aStarDistance ← AStarDistances(node, next)
        if aStarDistance > euclideanDistance **then**
            nearest, minDistance ← getNearestNode()
            if next ≠ nearest **then**
                totalCost ← totalCost + minDistance
                finalPath ← updateFinalPath
            else
                totalCost ← totalCost + aStarDistance
            end if
        else
            totalCost ← totalCost + aStarDistance
        end if
    end for
    costs[path] ← totalCost, finalPath
end for
finalPath, totalCost ← getPathCombinationWithLowestCost(costs)

the path that starts and ends at the start and goal points respectively.

This method is expected to be faster than the previously mentioned methods, especially those where TSP was calculated multiple times, however, it is not expected to find the optimal path. Also, in an environment with a large number of nodes, this is the only method guaranteed to find a solution in a reasonable amount of time.

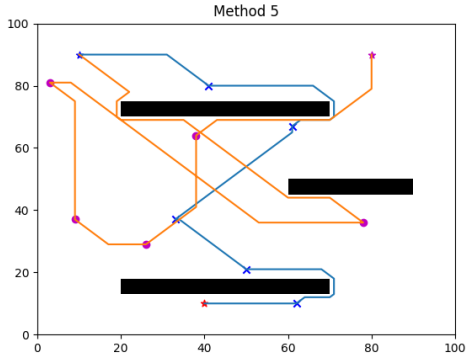


Fig. 7: Method 5 Simulation

Time(s)	Path 1 Length	Path 2 Length	Total Path Length
0.34	181	304	485

TABLE V: Results of Method 5

Algorithm 7 Method 5 Algorithm

Require: paths

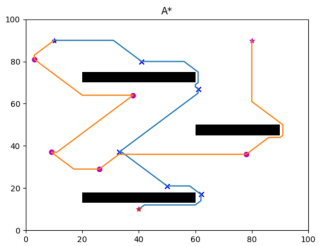
costs ← {}
for each path in paths **do**
    distanceMatrix ← getEuclideanDistances(path)
    distanceMatrix ← addDummyNode(distanceMatrix)
    route, cost ← GreedyTSP(distanceMatrix)
    route ← removeDummyNode(route)
    route ← reverse(route)
    costs[path] ← totalCost, finalPath
end for
finalPath, totalCost ← getPathCombinationWithLowestCost(costs)

VI. SIMULATION & RESULTS

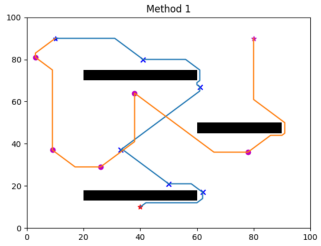
In this section we showcased the results of running the proposed methods in different environments to test their performance and efficiency. The performance of A* was also included as a benchmark.

A. Experiment 1

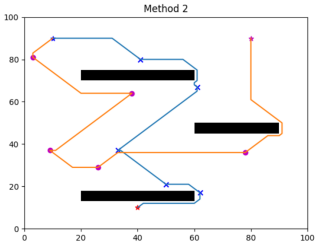
By looking at the results, we can draw the conclusion that Method 2 performed the best in finding the optimal path which has a length of 345. However, as we expected in Section V, the computational time was greater than the other algorithms due to the computation of the A* algorithm several times. Another thing to notice here is that the behavior of all the other algorithms is almost similar. This is due to the spread of nodes away from the obstacles which reduced the need to take a longer path to avoid obstacles. As shown in table VI, the path length and the number of the expanded nodes of all the algorithms are almost the same except in Method 5. In this experiment Method 4 generated a path with the same cost as Method 3 while visiting the same number of cells however it took less time to achieve that. A* and Method 2 also found a path of the same length while expanding the same number of nodes however, Method 2 was faster. This shows that the nodes were distributed in a way that generates an optimal path by following the nearest neighbors. Method 1 found a sub-optimal path with the least amount of time and Method 5 resulted in the longest path. All the proposed methods required less time than A* while generating a path of very similar cost except for Method 5, which proves the effectiveness of these methods.



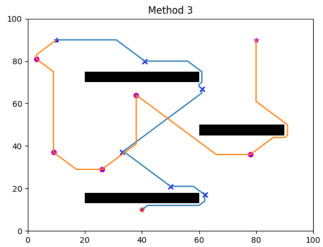
(a) A*



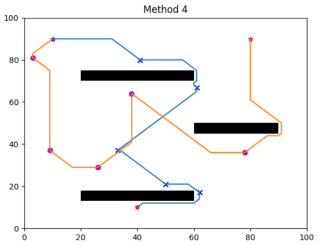
(b) Method 1



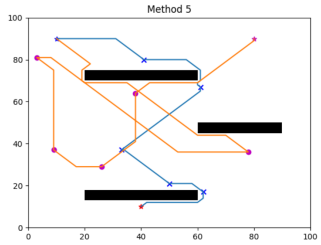
(c) Method 2



(d) Method 3



(e) Method 4



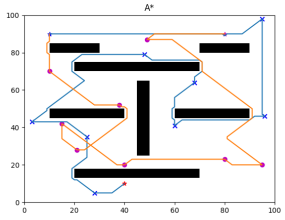
(f) Method 5

	Time(s)	Path Length	Avg Expanded Nodes
A*	4	345	164.28
Method1	0.18	348	163.71
Method2	3.35	345	164.28
Method3	1.33	348	163.71
Method4	1.29	348	163.71
Method5	0.22	438	212

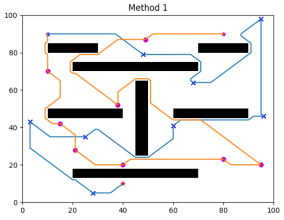
TABLE VI: Results of Experiment 1

B. Experiment 2

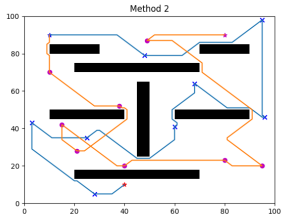
In this experiment, we changed the environment to add more obstacles and nodes. As we expected, the computational time of all the algorithms' increased massively, except for Method 5 in which we used the Greedy TSP instead of the direct solution (0.43 seconds) it increased only minorly. Although Greedy TSP was the fastest, that still didn't make it the optimal algorithm because it had the highest path length (725). Method 2 had the shortest path length (618) and the time it took (3.35 seconds) to execute is reasonable. The method that had the shortest path length was Method 2, which implies that our expectations are once again met and Method 2 was the most optimal in this environment. As for Methods 3 and 4, the latter performed better in terms of time and path length. It was expected from Method 4 to require less time. However, unexpectedly, Method 4 resulted in a more optimal path, and the reason might be that it expanded more than Method 3. The path generated by A* was longer than all the other paths generated except in Method 5 while taking more time to compute it, which proves why combining TSP to A* is crucial for a better path.



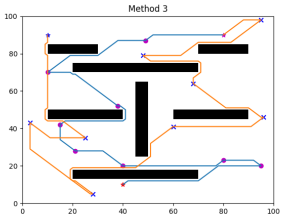
(g) Experiment(2) A*



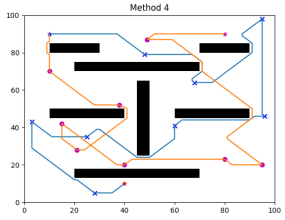
(h) Experiment(2) Method 1



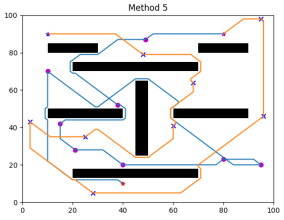
(i) Experiment(2) Method 2



(j) Experiment(2) Method 3



(k) Experiment(2) Method 4



(l) Experiment(2) Method 5

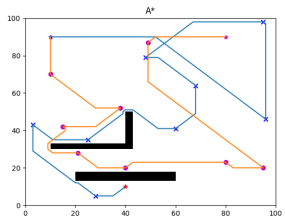
	Time(s)	Path Length	Avg Expanded Nodes
A*	38.05	636	209.39
Method1	1.35	618	233.57
Method2	16.91	583	199.7
Method3	22.61	621	190.7
Method4	18.5	596	208.6
Method5	0.43	725	254.7

TABLE VII: Results of Experiment 2

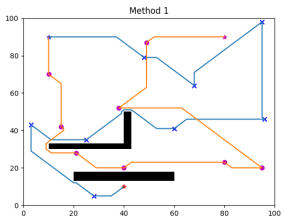
C. Experiment 3

In this experiment, we decreased the number of obstacles in the environment and set the number of nodes as experiment 1. The worst performing methods were Method 5 and A* alone. Although A* showed good performance in Experiment 1, its

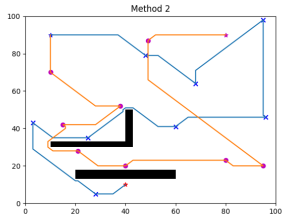
performance is dependant on the environment since it didn't perform well in Experiments 2 and 3. Besides that, we can infer that Method 1 and Method 2 have similar performances when there aren't many obstacles between the nodes, however Method 2 requires much more computational time. As for Methods 3 and 4, unlike experiment 1, Method 3 had better performance in all three criteria. Method 4 explored more of the environment than Method 3, yet it generated a longer path which might be due to it following the nearest neighbor in certain cases. As for Method 5, it generated the longest path with the least amount of time which might be due to the usage of an approximate solution to TSP.



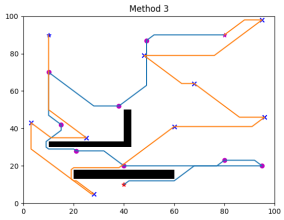
(m) Experiment(3) A*



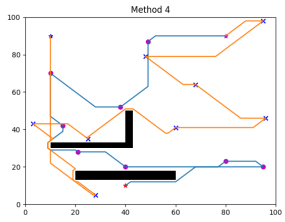
(n) Experiment(3) Method 1



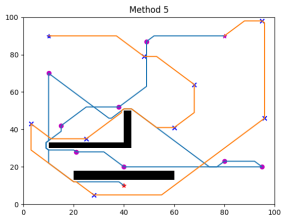
(o) Experiment(3) Method 2



(p) Experiment(3) Method 3



(q) Experiment(3) Method 4



(r) Experiment(3) Method 5

	Time(s)	Path Length	Avg Expanded Nodes
A*	9.4	606	185.8.39
Method1	0.94	560	165.8
Method2	6.08	558	170.6
Method3	4.74	594	177.2
Method4	8.79	606	187.2
Method5	0.23	665	202.6

TABLE VIII: Results of Experiment 3

VII. CONCLUSION

This paper has introduced and detailed five algorithms that can be used to find a path that visits multiple locations while avoiding obstacles. Based on the experiments' results, we realized that each algorithm performs better than the others in a specific environment. Selecting the best performing algorithm is highly dependant on the environment and the criteria that we are most interested in whether it be computational time or total path length. We can also infer that the most optimal method proposed is Method 2 that is guaranteed to produce the shortest path in all experiments however, it requires more computational time than the others. We also concluded that the performance of A* is highly dependant on the environment, which can be solved by combining it with TSP. As for Methods 3 and 4, in environments where choosing the nearest neighbors yields a longer path, Method 3 should be used instead of 4. All in all, this paper extends the functionality of both A* and TSP algorithms and tends to be a promising step towards more advanced extensions of the proposed algorithms to deal with larger environments.

In this paper we only considered a limited amount of nodes to test the proposed algorithms. In future work, the performance of these algorithms with a greater number of nodes can be examined. The usage of different heuristics for TSP, other than Greedy that was used here, can be done to provide more optimal paths when dealing with a large number of nodes. Apart from that, ways to optimize the code can be explored to decrease the computation time without compromising the performance. Modifications to the algorithms can also be checked to make them suit real-life applications.

REFERENCES

[1] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The traveling salesman problem*. Princeton university press, 2011.

[2] N. Aras, B. J. Oommen, and İ. Altınel. The kohonen network incorporating explicit statistics and its application to the travelling salesman problem. *Neural Networks*, 12(9):1273–1284, 1999.

[3] K. Arora, S. Agarwal, and R. Tanwar. Solving tsp using genetic algorithm and nearest neighbor algorithm and their comparison. *International Journal of Scientific & Engineering Research*, 7(1):1014–1018, 2016.

[4] K. Arora, S. Agarwal, and R. Tanwar. Solving tsp using genetic algorithm and nearest neighbour algorithm and their comparison. *International Journal of Scientific Engineering Research*, 7, 2016.

[5] S. Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998.

[6] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. 1976.

[7] G. Clarke and J. W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581, 1964.

[8] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[9] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, and L. Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 96, 12 2014.

[10] H. J. DW;. "neural" computation of decisions in optimization problems.

[11] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, and Z. Zhang. Inconsistent heuristics in theory and practice. *Artificial Intelligence*, 175(9):1570–1603, 2011.

[12] D. Ferguson, M. Likhachev, and A. Stentz. A guide to heuristic-based path planning. pages 9–18, 2005.

[13] D. Ferguson and A. Stentz. The delayed d* algorithm for efficient path replanning. In *Proceedings of the 2005 IEEE international conference on robotics and automation*, pages 2045–2050. IEEE, 2005.
[14] J. C. Fort. Solving a combinatorial problem via self-organizing process: An application of the kohonen algorithm to the traveling salesman problem - biological cybernetics.
[15] D. Gamboa, C. Rego, and F. Glover. Implementation analysis of efficient heuristic algorithms for the traveling salesman problem. *Computers Operations Research*, 33(4):1154–1172, 2006. Part Special Issue: Optimization Days 2003.
[16] M. T. Goodrich and R. Tamassia. The christofides approximation algorithm. *Algorithm Design and Applications, Wiley*, pages 513–514, 2015.
[17] G. Gutin and A. Yeo. Anti-matroids. *Operations Research Letters*, 30(2):97–99, 2002.
[18] G. Gutin, A. Yeo, and A. Zverovich. Traveling salesman should not be greedy: Domination analysis of greedy-type heuristics for the tsp. *BRICS Report Series*, 8(6), 2001.
[19] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
[20] Y. Haxhimusa, W. G. Kropatsch, Z. Pizlo, A. Ion, and A. Lehrbaum. Approximating tsp solution by mst based graph pyramid. *Graph-Based Representations in Pattern Recognition Lecture Notes in Computer Science*, page 295–306.
[21] R. M. Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Mathematics of Operations Research*, 2(3):209–224, 1977.
[22] G. Kizilateş and F. Nuriyeva. On the nearest neighbor algorithms for the traveling salesman problem. In *Advances in Computational Science, Engineering and Information Technology*, pages 111–118. Springer, 2013.
[23] N. Kumar, Z. Vámossy, and Z. M. Szabó-Resch. Heuristic approaches in robot navigation. In *2016 IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES)*, pages 219–222, 2016.
[24] M. Likhachev and A. Stentz. R* search. 2008.
[25] J.-Y. Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 63(3):337–370, 1996.
[26] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *Fundamental Problems in Computing*, page 45–69, 2009.
[27] D. Talia, D. Simson, A. Miné, T. Pichpibul, and R. Kawtummachai. A heuristic approach based on clarke-wright algorithm for open vehicle routing problem. *The Scientific World Journal*, 2013:874349, 2013.
[28] M. Worboys. *The Mathematical Gazette*, 70(454):327–328, 1986.
[29] Y. Xiao and A. Konak. A simulating annealing algorithm to solve the green vehicle routing scheduling problem with hierarchical objectives and weighted tardiness. *Applied Soft Computing*, 34:372–388, 2015.
[30] X. Xu, Y. Yang, and S. Pan. Motion planning for mobile robots. *Advanced Path Planning for Mobile Entities*, 2018.