A Comparison of BFS, Dijkstra's and A* Algorithm for Grid-Based Path-Finding in Mobile Robots

Faza Fahleraz 13516095

Program Studi Teknik Informatika Sekolah Teknik Elektro dan Informatika Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia 13516095@std.stei.itb.ac.id

Abstract—Path-finding plays a very significant role in many applications such as mobile robot navigation. A grid-based environment can be used to represent the navigation space for robots on the real world, to avoid obstacles, or to navigate to a desired position. Unlike many other path-finding problems, gridbased path finding poses some unique challenges. A grid-based search environment can be represented by a graph with a twodimensional grid of nodes with each nodes having a connecting edge to it's neighboring edge. We can also represent some kind of obstacles on the grid-based environment by deleting several nodes. In this paper, I will do some analysis for three different path-finding algorithms namely Breadth-First Search, Dijkstra's, and A* in the context of a grid-based path-finding that is frequently used in mobile robot navigation. I will also present a performance comparison for each of those algorithms.

Keywords—path-finding; grid-based search; heuristic; BFS; Dijkstra; A*.

I. INTRODUCTION

Path-planning plays a very significant role in many applications such as mobile robot navigation. A robot should be move around it's environment. It should be able to move from it's current position to a new desired postition. In order to do this, a mobile robot should be able to model it's environment and the obstacles the robot should avoid in the environment. In addition to modelling it's environment, a mobile robot should be able to plan it's movement to a desired new position inside it's environment by calculating a path that stretches from the robot's current position to the new position. We also prefer the path to be the shortest possible. This can be achieved by a pathplanning algorithm.



Figure 1.1 Several mobile robots playing soccer in the RoboCup MSL. Source: https://www.researchgate.net/figure/A-typical-scenario-ofthe-RoboCup-MSL-competition-a-match-between-TU-e-and-NuBotin fig1 317754606

In this paper, I will discuss how some of the well known path-planning algoritms can be used to tackle this problem namely Breadth-First Search, Dijkstra's, and A* algorithm. I will also do a comparison of the performance for each of these algorithms.

II. GRID-BASED ENVIRONMENT REPRESENTATION

A mobile robot should be able to navigate it's environment autonomously and it should be able to do it in a fast and efficient manner. In order to navigate, a mobile robot should have some kind of a model of it's environment. A grid-based map can be used to represent the robot's environment.



Figure 2.1 A model of the robot's environment as a grid map. Source: author

A mobile robot can move in many direction, therefore we should be able to represent such movement in our model. A grid fits nicely with how a mobile robot can navigate. To model the environment in a grid, we shall devide the environment into a two-dimensional grid of nodes where each nodes represents each of the positions the robot may be. As you may have noticed, the more nodes there is, the more positions of the robot can be represented by the model, hence the greater the resolution or the accuracy of the robot's movement can be achieved. To model the robot's movement, we can add edges that connects each of the nodes to all of it's neighboring nodes.

In addition to space and the robot itself, an environment can also consist of other objects that acts as an obstacle for the robot. Generally, when we tell a robot to move to a new desired position, we also want the robot to avoid obstacles along it's way. This is called obstacle avoidance.

The question now is, how do we represent these obstacle inside our grid-based model of the environment. This can be achieved by deleting the nodes at where the obstacle is. And subsequently deleting all of the edges that connect to those nodes. Therefore we represents the obtacles as an absence of node in the obstacle's position. Signifying that the robot cannot go into that direction.



Figure 2.2 A graph representing the grid-based environment. Source: author

After modelling the robot's environment, we should also be able to model the robot's movement. This involves finding a path that streches from the robot's current position to the robot's goal position. We also would like the path to be the shortest and we should be able to compute such path in a reasonabli fast time. There are several algorithm that we can use to accomplish this task. The most popular algorithms to solve this kind of path-finding problem are the BFS Algorithm, Dijkstra's Algorithm, and A* Algorithm.



Figure 2.3 A graph representing the grid-based environment with obstacles represented as an absence of nodes. Source: author

III. PATH-FINDING ALGORITHMS

A mobile robot should be able to navigate it's environment autonomously. To do that, we need some kind of a pathplanning algorithm for the robot to determine the shortest path from the robot's current position to a new desired position. In this section, I will discuss three of the most common pathfinding algorithms: Breadth-First-Search, Dijkstra's, and A*.

A. Breadth-First Search Algorithm

Breadth-first search (BFS) is an algorithm for traversing or searching a tree or a graph. BFS can be used for many

applications. In this case it can be used to find the shortest path form a source node in a graph to a goal node.

The BFS algorithm was first invented in 1945 by Konrad Zuse and Michael Burke, in their (rejected) Ph.D. thesis on the Plankalkül programming language, but this was not published until 1972. It was reinvented in 1959 by Edward F. Moore, who used it to find the shortest path out of a maze, and later developed by C. Y. Lee into a wire routing algorithm in 1961.

The BFS algorithm start from a single source node and subsequently search over all the neighboring nodes and add it into a queue. After fisiting each node, the algorithm will flag that node as visited and will not add that node into the queue if it is visited again in the future. The algorithms visits nodes in the graph according to the queue and will add the all the neigboring nodes of the currently visited node into the back of the queue. The algorithm stops if the queue is empty or the goal node is the currently visited node. Here is a general pseudo-code for BFS:

- problem.get_successors(subtree_root):
 if child in closed_set:
 continue
 - if child not in open_set:
 path[child] = (subtree_root, action)
 open_set.enqueue(child)

```
closed set.add(subtree root)
```

In the case of robot navigaion, we cannot use the output of the algorithm straight away, we need an algorithm to reverseiterate from the goal to the source node in order to know the shortest path from the source to the goal node:

```
def reconstruct_path(state, path):
    action_list = list()
    while True:
    row = path[state]
```

```
if len(row) == 2:
    state = row[0]
    action = row[1]
    action_list.append(action)
    else:
        break
    action_list.reverse()
    return action_list
```

B. Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later in 1959.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other node in the graph. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

In the case of path-finding for robot navigation, we of course only need the shortest-path from the source node to a single destination node because a robot only moves from a single current location to a desired new location. Also, the search space will be too big if we also search for the shortest path for all the nodes since in a grid-based world, the number of nodes will be very big. To do this we can slightly modify the algorithm. This variant is the one that we will use in the case of robot navigation. The general pseudo-code for Dijkstra's Algorithm is:

```
function Dijkstra(Graph, source, goal):
  create vertex set Q
  for each vertex v in Graph:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
    add v to Q
  dist[source] \leftarrow 0
    while Q is not empty:
      u \leftarrow vertex in Q with min dist[u]
      remove u from Q
      if u = goal:
        break
      for each neighbor v of u:
        alt \leftarrow dist[u] + length(u, v)
        if alt < dist[v]:</pre>
           dist[v] ← alt
           prev[v] ← u
    return dist[], prev[]
```

Just like in the case of BFS, we cannot use the output of the algorithm straight away, we need an algorithm to reverseiterate from the goal to the source node in order to know the shortest path from the source to the goal node:

```
function reconstruct_path(target)
S ← empty sequence
u ← target
while prev[u] is defined:
    insert u at the beginning of S
    u ← prev[u]
insert u at the beginning of S
    return S
```

C. A* Algorithm

A* algorithm is another algorithm for finding the shortest path between nodes in a graph. This algorithm was first described by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) in 1968.

A* is an informed search algorithm, or a best-first search, which means that it solves the path-finding problem by searching among all possible paths to the solution (all the possible solutions in the search space) for the one that has the smallest cost based on the heuristic used (in the case of path-finding, the obvious heuristic is distance from the goal), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is an improvement over Dijkstra's algorithm by using a heuristic to guide it's search faster towards the goal.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Formally, A* selects the path that minimizes:

$$f(n) = g(n) + h(n)$$

where n is the last node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A^* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic. The general pseudo-code of the A* Algorithm is:

```
function A*(start, goal)
closedSet := {}
openSet := {start}
cameFrom := an empty map
gScore := map with default value of
    Infinity
gScore[start] := 0
fScore := map with default value of
    Infinity
fScore[start] :=
    heuristic_cost_estimate(start, goal)
```

```
while openSet is not empty
  current := the node in openSet having the
    lowest fScore[] value
```

```
openSet.Remove(current)
closedSet.Add(current)
```

```
for each neighbor of current
if neighbor in closedSet
continue
```

```
if neighbor not in openSet
   openSet.Add(neighbor)
```

```
tentative_gScore := gScore[current] +
    dist_between(current, neighbor)
if tentative_gScore >= gScore[neighbor]
    continue
```

```
cameFrom[neighbor] := current
gScore[neighbor] := tentative_gScore
fScore[neighbor] := gScore[neighbor] +
    heuristic_cost_estimate(neighbor,
    goal)
```

```
return failure
```

Much like the previous algorithms, we cannot use the output of the A^* algorithm straight away, we need an algorithm to reverse-iterate from the goal to the source node in order to know the shortest path from the source to the goal node:

```
function reconstruct_path(cameFrom, current)
  total_path := [current]
  while current in cameFrom.Keys:
    current := cameFrom[current]
    total_path.append(current)
```

return total_path

IV. PERFORMANCE ANALYSIS

Among the three algorithms to test, one can safely assume that in finding the shotest path, A* will be the fastest since it's search will be guided by a heuristic and Dijkstra's and BFS will both be slower than A* and have more or less the same execution time because the exhaustive search nature of both of those algorithms.

	0000
	00.00
	0
************	0
	_

Figure 4.1 Visualization of The Search History of Dijkstra's and A* Algorithm on a Grid Map with an Obstacle Source: https://commons.wikimedia.org/wiki/ File:Dijkstras_progress_animation.gif and https:// commons.wikimedia.org/wiki/File:Astar_progress_animation.gif

To test the performance of those algorithm against each other, in this paper I propose a test that consist of three different grid maps differing in configurations. Please notice that the grid map is a model of the robot's environment that can be represented as a graph that was discussed in chapter two of this paper. Each of the grid maps are of the same size but with different configuration of obstacles that hopefully can test the algorithm's characteristics. Each of the algorithms are tested on these three grid maps to find the shortest path of each of the maps so that we can compare the execution time of each of the algorithms. For the A* algorithm, we will use the manhattan distance to the goal node as the heuristic for each node. The start node of the map is colored in green and the goal node is colored in yellow. Following are the test results:

```
A. Test map 1 (20 x 20)
```



Figure 4.2 Visualization of The Test Grid Map 1 Source: author

Test results:

- BFS: 4.10330891609 seconds
- Dijkstra's: 4.21949696541 seconds
- A*: 2.08329987526 seconds
- B. Test map 2 (20 x 20)



Figure 4.3 Visualization of The Test Grid Map 2 Source: author

Test results:

- BFS: 5.44553685188 seconds
- Dijkstra's: 5.63682413101 seconds
- A*: 3.04203510284 seconds

C. Test map 3 (20 x 20)



Figure 4.4 Visualization of The Test Grid Map 2 Source: author

Test results:

- BFS: 5.00317502022 seconds
- Dijkstra's: 5.385323044764 seconds
- A*: 0.587857961655 seconds

We can clearly see that A^* consistently comes up as the fastest just as predicted earlier. And we can also see that BFS and Dijkstra are both much slower than A^* and more or less have the same execution time. If you look at the test map 3, you can see that it has quite a bit more obstacles (and also configured in a more complex way) than the other two test set. This difference is reflected in the test results as the A^* execution time is significantly faster than the other two algorithm's execution time (around 9x). This is probably because the BFS and Dijkstra's algorithm was having a harder time figuring the shortest path since it had to search multiple sections of the map that are not connected to the goal node because of the complex and deceptive shape of the map's obstacles.

V. CONCLUSIONS

A mobile robot's environment as well as obstacles in the environment can be modelled as a grid that can be represented computationally as a graph. In order to navigate that environment, the mobile robot can use a path-finding algorithm to find the shortest path from the robot's current position to the desired new position while also avoid obstacles along it's way. Among some of the path-finding algorithms that are tested on this paper, the A* algorithm is consistently the fastest by a signifincant margin.

VI. ACKNOWLEDGMENTS

I would like to thank Dr. Nur Ulfa Maulidevi, S.T., M.Sc, Dr. Rinaldi Munir, and Dr. Masayu Leylia Khodra S.T., M.T. as the lecturers of this amazing class and also giving me this chance to explore the interesting applications of the things that I learnt during this class which is to design and implement various algorithms in an efficient manner. I would also like to thank my families and friends to keep me motivated during this chaotic times. As someone who loves robotics, I really appreciate this opportunity to write something that connects my favourite class in this semester and my passion for robotics.

VII. REFERENCES

- [1] Felner, Ariel. Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm. 2011.
- [2] Bondy, Adrian; Murty, U.S.R. (2008). Graduate Texts in Mathematics 244 Graph Theory. Springer.
- [3] Zeng, W.; Church, R. L. Finding shortest paths on real road networks: the case for A*. International Journal of Geographical Information Science. 2009.
- [4] Anbuselvi, R. Path Finding Solutions for Grid Based Search. 2013.
- [5] Saranya, C. *Real Time Evaluation of Grid Based Path Planning Algorithms: A comparative study.* IEEE. 2013.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Mei 2018

Faza Fahleraz 13516095