Proceedings of the 2023 International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/39/20230570

Research on autonomous mobile robot maze navigation problem based on Dijkstra's algorithm

Qihan Wu

Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT

qxw092@student.bham.ac.uk

Abstract. In recent years, the field of autonomous mobile robotics has garnered significant attention due to its potential applications in various domains such as logistics, surveillance, and search and rescue operations. A crucial challenge in this area is the efficient navigation of robots within complex and dynamic environments, particularly when navigating through maze-like structures. The maze navigation problem involves finding optimal paths for robots to traverse from their initial positions to designated destinations while avoiding obstacles and making intelligent decisions to ensure timely and safe navigation. This study aims to investigate and apply Dijkstra's algorithm to solve the maze navigation problem for autonomous mobile robots. By analyzing the navigation challenges faced by autonomous mobile robots in maze environments, a solution based on Dijkstra's algorithm is proposed. In conclusion, this study contributes to the field of autonomous mobile robotics by proposing and evaluating the application of Dijkstra's algorithm for maze navigation. The experimental results validate its potential to address the challenges of navigating intricate maze environments. However, it is acknowledged that further refinement and innovation are possible to continue improving the performance of autonomous mobile robots in maze navigation scenarios.

Keywords: Dijkstra's Algorithm, Autonomous Mobile Robots, Maze Navigation, Path Planning, Graph Searching.

1. Introduction

The maze navigation problem for autonomous mobile robots is an important research area in artificial intelligence and robotics. Maze navigation requires the robot to find the shortest or efficient path in an unknown maze environment to reach the goal point [1,2].

The main topic of this research is the solution of the maze navigation problem for autonomous mobile robots based on Dijkstra's algorithm.

The research question is how to apply Dijkstra's algorithm to solve the navigation problem for autonomous mobile robots in maze environments. The methods include maze modeling, path planning, and robot control design and implementation.

By studying and applying Dijkstra's algorithm to solve the maze navigation problem for autonomous mobile robots, the path planning and navigation capabilities of robots in unknown environments can be improved, expanding the practical application areas of robots.

© 2023 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/39/20230570

2. Definition and challenges of maze navigation problem

2.1. Overview of the Maze Navigation Problem

In this section, we will introduce the fundamental concepts and challenges of the autonomous mobile robot maze navigation problem. The maze navigation problem refers to the task of an autonomous mobile robot finding the shortest or efficient path from the starting point to the destination point in an unknown maze environment. This problem holds significant research significance and practical applications in the fields of artificial intelligence and robotics.

The maze navigation problem can be represented using graph theory, where the maze's structure can be seen as a directed or undirected graph, with each room or corridor serving as a node in the graph and connected edges between adjacent rooms or corridors [3]. The robot's movement in the maze can be viewed as searching for a path in the graph, allowing it to reach the goal point from the starting point while minimizing the path's length.

The maze navigation problem presents several challenges to the robot [4]:

Unknown Maze Structure: Robots typically navigate in unknown maze environments, requiring them to simultaneously construct a maze map and plan a path while exploring.

Complex Path Planning: The maze may contain multiple paths, but not all of them are optimal. Hence, the robot needs an efficient path planning algorithm to find the best or near-optimal solution within a limited time frame.

Environment Perception and Obstacle Avoidance: robots must use sensors to perceive the maze environment, avoid collisions with obstacles, and make reasonable decisions when encountering complex terrain or topological structures. Real-time Requirements: In some scenarios, real-time navigation is essential for the robot to respond promptly to environmental changes and new navigation demands.

Robustness and Stability: The robot's navigation algorithm must possess a level of robustness and stability, allowing it to reliably complete navigation tasks in different maze environments.

2.2. Existing solutions and their limitations

Currently, various methods are available to solve the autonomous mobile robot maze navigation problem, including:

Random Exploration Method: The robot explores the maze randomly, continuously trying different paths until it finds a route connecting the starting and destination points. While simple, this method has high time complexity and uncertainty, potentially leading to longer navigation times and sub optimal paths.

Heuristic-based Search Methods: For instance, the A* algorithm considers the cost of paths and heuristic functions (e.g., Manhattan distance or Euclidean distance) for searching, aiming to find the optimal or near-optimal solution. This method can improve path planning efficiency to some extent, but for complex maze structures, it may still yield local optima.

Depth-First Search (DFS) and Breadth-First Search (BFS): These algorithms can be used to traverse the maze graph and find paths from the starting point to the destination point. DFS tends to delve deep into the search, possibly resulting in longer search paths; BFS can find the shortest path, but it consumes significant memory in large-scale mazes [5].

The above methods may perform well in specific scenarios, but they also have certain limitations. For instance, the random exploration method may lead to extended search times, while heuristic search algorithms may be limited by the design of the heuristic function and iteration counts. Additionally, for complex and large-scale maze environments, path planning computational complexity and memory consumption could significantly increase.

To address these limitations, this research will focus on studying and applying Dijkstra's algorithm to solve the autonomous mobile robot maze navigation problem, aiming to optimize the algorithm for improved path planning efficiency and performance [6]. The following chapters will provide detailed

DOI: 10.54254/2755-2721/39/20230570

insights into the principles and applications of Dijkstra's algorithm and how it can be applied to solve the maze navigation problem for autonomous mobile robots.

3. Principles of Dijkstra's Algorithm

Dijkstra's Algorithm is a widely used graph traversal and shortest path-finding algorithm [7,8]. It was developed by Dutch computer scientist Edsger W. Dijkstra in 1956 and is used to find the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights. The algorithm maintains a set of vertices for which it has found the shortest path from the source vertex and a priority queue (usually implemented using a min-heap) to keep track of the vertices with the minimum distance from the source [9].

Here's a step-by-step explanation of Dijkstra's Algorithm:

Initialization: Start by marking all vertices' distances as infinity and the distance of the source vertex as 0. Also, create an empty set to keep track of the visited vertices.

Find the vertex with the minimum distance: Choose the vertex with the smallest distance from the source vertex among the vertices not yet visited. Initially, this will be the source vertex itself.

Relaxation: For the selected vertex, update the distances of its neighboring vertices. To do this, iterate through all the edges connected to the selected vertex. For each neighboring vertex, calculate the total distance from the source vertex passing through the selected vertex. If this distance is smaller than the current distance of the neighboring vertex, update the distance of the neighboring vertex with this smaller value.

Mark the selected vertex as visited: After updating the distances of neighboring vertices, mark the selected vertex as visited by adding it to the set of visited vertices.

Repeat steps 2 to 4: Continue this process until all vertices are marked as visited.

Termination: The algorithm terminates when all vertices have been visited, and the shortest path distances from the source vertex to all other vertices are determined.

Once the algorithm is complete, the shortest distance from the source vertex to any other vertex can be found in the distance array. Additionally, the algorithm can be adapted to keep track of the shortest path itself by using an array of predecessors, which stores the previous vertex on the path to each vertex.

4. Maze Modeling and Path Planning

4.1. Methods for maze map modeling

In this paper, the following data structure will be defined as a modelling representation of the maze[10]:

```
type Maze = [[Block]]
type Block = (Wall, Wall)

data Wall = Wall | None
  deriving (Eq, Show)

type Position = (Int, Int)
type Distance = Int
type Path = [Position]
```

The string representing the maze is parsed into a maze data structure by the following code:

```
parseMaze :: String -> Maze
parseMaze x = map (parseMazeHelper2 . tail) y
  where
y = (tail . lines) x

parseMazeHelper :: Char -> Wall
parseMazeHelper '_' = Wall
```

DOI: 10.54254/2755-2721/39/20230570

```
parseMazeHelper '|' = Wall
parseMazeHelper ' ' = None
parseMazeHelper _ = error ""Invalid maze""

parseMazeHelper2 :: String -> [Block]
parseMazeHelper2 (x : y : xs) = (parseMazeHelper x, parseMazeHelper y) : parseMazeHelper2 xs
parseMazeHelper2 [_] = error ""Invalid maze""
parseMazeHelper2 [] = []
```

4.2. Definition and representation of nodes and edges

In this paper, in order to maintain clarity and simplicity of implementation in the code implementation, a single lattice will be used as a node, and therefore the distance between all lattices will be 1. In the following section the problems arising from using a single lattice as a node will be discussed and the solution to the problem will be analysed.

4.3. Path planning algorithm design based on Dijkstra's algorithm

The Path planning algorithm design based on Dijkstra's algorithm is implemented by the following function. This function takes three arguments: the maze data structure, the location of the start point, and the location of the end point. If the shortest path is found, the function returns it; otherwise, the result is Nothing.

```
dijkstra :: Maze -> Position -> Position -> Maybe Path
dijkstra maze start goal = dijkstraHelper maze start goal [] [(start, start, 0)]
```

The definitions and explanations of the remaining functions are listed:

```
dijkstraHelper :: Maze -> Position -> Position -> [(Position, Position)] -> [(Position, Position, Distance)] -> Maybe Path
dijkstraHelper _ _ _ _ [] = Nothing
dijkstraHelper maze start goal visited ((pos, prev, dist) : queue)
  | pos == goal = Just $ reverse $ pos : path
  | otherwise = dijkstraHelper maze start goal visited' queue'
  where
    visited' = (pos, prev) : visited
    queue' = foldl (exploreNext visited pos dist) queue (neighbors maze pos)
    path = backtracking pos visited'
```

- *Input*: `maze` is a maze, `start` is the current search location, `goal` is the goal location, `visited` is a list of visited locations, and `queue` is a pending queue.
- *Output*: `Maybe Path` is a possibly empty path, `Path` is a list representing paths.
- *Function*: The `dijkstraHelper` function iterates using recursion, continually removing locations from the queue and exploring the next step until the target location is found or the queue is empty. In each iteration, it adds the current location to the visited list and updates the queue to continue exploring the next step. If the goal location is found, it returns the reversed list of paths, representing the shortest path from the start to the end.

```
exploreNext :: [(Position, Position)] -> Position -> Distance -> [(Position, Position, Distance)] -> Position -> [(Position, Position, Distance)]
exploreNext visited prev d queue pos
  | pos `elem` map fst visited = queue
  | otherwise = updateDistance pos prev (d + 1) queue
```

DOI: 10.54254/2755-2721/39/20230570

- *Input*: `visited` for the list of visited locations, `prev` for the previous location, `d` for the distance, `queue` for the pending queue, `pos` for the current location.
- *Output*: new pending queue.
- *Function*: `exploreNext` function is used to check the neighbours at the current position and update the pending queue. If the neighbours have already been visited, the original queue is returned directly; otherwise, the `updateDistance` function is called to update the distance and return to the new queue.

```
updateDistance :: Position -> Position -> Distance -> [(Position, Position, Distance)] -> [(Position, Position, Distance)]
updateDistance pos prev d queue
  | null xs = updateDistanceHelper pos prev d queue
  | otherwise =
      if d < (\(_, _, x) -> x) (head xs)
        then updateDistanceHelper pos prev d xs
        else queue
  where
    xs = filter (\(x, _, _) -> x == pos) queue
```

- *Input*: `pos` for current position, `prev` for previous position, `d` for distance, `queue` for pending queue.
- *Output*: new pending queue.
- *Function*: `updateDistance` function is used to update the distance information of the current position in the queue. If the current position does not exist in the queue, a new entry is added; if it already exists, the original entry is updated with the new distance.

```
updateDistanceHelper :: Position -> Position -> Distance -> [(Position, Position, Distance)] -> [(Position, Position, Distance)]
updateDistanceHelper pos prev d queue = takeWhile f queue ++ [(pos, prev, d)] ++ dropWhile f queue
  where
    f (_, _, x) = x <= d
```

- *Input*: `pos` for current position, `prev` for previous position, `d` for distance, `queue` for pending queue.
- *Output*: new pending queue.
- *Function*: The `updateDistanceHelper` function is used to insert new positions and distances into the queue and keep the queue in increasing order of distance.

```
neighbors :: Maze -> Position -> [Position]
neighbors mss (x, y) =
  [ (x', y')
    | (x', y') <- [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)],
      isValidPosition mss (x', y'),
      neighborsHelper mss (x, y) (x', y')
  ]
```

- *Input*: `maze` for the maze and `(x, y)` for the current position.
- *Output*: a list representing the positions of the neighbours.

DOI: 10.54254/2755-2721/39/20230570

- *Function*: The `neighbors` function is used to find the neighbours at the current position and keep only the neighbours that match the requirements of the maze.

```
neighborsHelper :: Maze -> Position -> Position -> Bool
neighborsHelper mss (x, y) (x', y')
  | x' - x == 1 = case p of
      (None, _) -> True
      _ -> False
  | x' - x == -1 = case p' of
      (None, _) -> True
      _ -> False
  | y' - y == 1 = case p of
      (_, None) -> True
      _ -> False
  | y' - y == -1 = case p' of
      (_, None) -> True
      _ -> False
  | otherwise = False
  where
    p = mss !! (x - 1) !! (y - 1)
    p' = mss !! (x' - 1) !! (y' - 1)
```

- *Input*: `maze` for maze, `(x, y)` for current position, `(x', y')` for position of neighbour to be checked.
- *Output*: boolean value indicating whether it is a valid neighbour.
- *Function*: `neighborsHelper` function is used to check if it is possible to move between the current position and the position of the neighbour to be checked.

```
isValidPosition :: Maze -> Position -> Bool
isValidPosition mss (x, y) =
  x > 0 && x <= length mss && y > 0 && y <= length (head mss)
```

- *Input*: `maze` for maze, `(x, y)` for current position.
- *Output*: Boolean value indicating whether the current position is within the maze.
- *Function*: `isValidPosition` function is used to check whether the current position is within the valid range of the maze.

```
backtracking :: Eq t => t -> [(t, t)] -> [t]
backtracking a xs = case lookup a xs of
  Just b -> if a == b then [] else b : backtracking b xs
  Nothing -> []
```

- *Input*: `a` for the current location, `xs` for a list of visited locations.
- *Output*: a list of paths.
- *Function*: The `backtracking` function is used to find the path from the beginning to the end of the path, starting from the end. It looks up the previous location based on the `xs` list, and so on until it gets back to the start. Note that since `xs` stores a mapping from the current position to the previous position, it may form a loop, so this needs to be checked when constructing the path.

5. Experimentation and Evaluation

Use the following maze as the example as shown in figure 1:

DOI: 10.54254/2755-2721/39/20230570

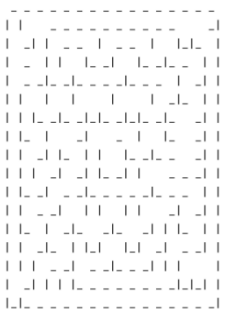

Figure 1. Maze data structure as an example.

By running the above code, the following results can be obtained:

Just

[(16,1),(15,1),(15,2),(14,2),(13,2),(13,3),(12,3),(12,2),(11,2),(11,3),(11,4),(10,4),(10,5),(9,5),(9,6),(8,6),(8,5),(7,5),(7,4),(8,4),(9,4),(9,3),(10,3),(10,2),(9,2),(8,2),(8,3),(7,3),(7,2),(6,2),(5,2),(5,3),(6,3),(6,4),(5,4),(5,5),(6,5),(6,6),(5,6),(5,7),(5,8),(6,8),(6,9),(5,9),(5,10),(5,11),(6,11),(6,12),(5,12),(5,13),(4,13),(4,12),(4,11),(4,10),(3,10),(3,9),(4,9),(4,8),(4,7),(4,6),(3,6),(3,5),(4,5),(4,4),(3,4),(2,4),(2,5),(2,6),(2,7),(3,7),(3,8),(2,8),(2,9),(2,10),(2,11),(3,11),(3,12),(2,12),(2,13),(3,13),(3,14),(3,15),(4,15),(4,14),(5,14),(5,15),(6,15),(6,14),(7,14),(8,14),(8,13),(8,12),(7,12),(7,11),(8,11),(8,10),(7,10),(7,9),(7,8),(8,8)]

```

This path is the shortest path from the starting point to the end of the maze.

Although the above result is correct, the problem of long search paths in real situations needs to be taken into account. In an implementation using the above code, the autonomous mobile robot, after moving to the next node, would have to return along the path to the yet-to-be-visited vertex with the smallest distance from the source vertex. This will consume a lot of time. Consider the following scenario as shown in figure 2:

**Figure 2.** A maze with a looping path.

The length of the path to be travelled from point (1, 1) to point (4, 4) is:

$$2 \times \sum_{i=1}^{5} i + 6 = 36 \tag{1}$$

To optimise this problem, consider only bifurcated intersections, dead-end paths, start points and end points as nodes. A lattice with only one wall or no wall can be considered as a fork, and a lattice with three walls can be considered as the end of a dead end. When the autonomous mobile robot is travelling towards a path, the node is recorded and returned only if it encounters one of the above situations.

However, this method still needs to be improved, as the autonomous mobile robot must continue exploring once it reaches the end point; otherwise, it may not be able to find the shortest path. Consider the following scenario as shown in Figure 3:
```

DOI: 10.54254/2755-2721/39/20230570

Figure 3. A maze with a winding path and a shortest path.

Also from point (1, 1) to point (4, 4), if the autonomous mobile robot starts off to the right, it will return to the right path, but this is clearly not the shortest path compared to the downward path.

For this problem, the autonomous mobile robot can be allowed to continue to complete the exploration, but in order to avoid unnecessary return paths, it can be chosen to continue to complete the exploration with the start point as the end point and the end point as the start point.

6. Conclusion

Through experimentation and evaluation, this study has achieved some success in autonomous mobile robot maze navigation based on Dijkstra's algorithm, enhancing the robot's navigation capabilities and effectiveness in maze environments.

However, the study's limitations lie in the high computational complexity and inadequate path optimization of Dijkstra's algorithm in certain complex maze environments and navigation scenarios, necessitating further improvements and exploration of other algorithms.

Future research directions may include more comparisons and applications of path planning algorithms, improvements in robot perception and decision-making strategies, and navigation capabilities in dynamic and unknown environments.

References

[1] Rudzuan Mohd Nor, "" Developing an autonomous Mobile Robot and a study of navigation towards nonholonomics problems"", in International Conference on Man-Machine System ,2009.

[2] Siegwart R, Nourbakhsh I R, Scaramuzza D. Autonomous mobile robots[J]. A Bradford Book, 2011, 15.

[3] Sadik A M J, Dhali M A, Farid H M A B, et al. A comprehensive and comparative study of maze-solving techniques by implementing graph theory[C]//2010 International Conference on Artificial Intelligence and Computational Intelligence. IEEE, 2010, 1: 52-56.

[4] Alatise M B, Hancke G P. A review on challenges of autonomous mobile robot and sensor fusion methods[J]. IEEE Access, 2020, 8: 39830-39846.

[5] Hidayatullah A S, Jati A N, Setianingsih C. Realization of depth first search algorithm on line maze solver robot[C]//2017 International Conference on Control, Electronics, Renewable Energy and Communications (ICCREC). IEEE, 2017: 247-251.

[6] Permana S H, Bintoro K Y, Arifitama B, et al. Comparative analysis of pathfinding algorithms A*, Dijkstra, and BFS on maze runner game[J]. IJISTECH (International J. Inf. Syst. Technol., vol. 1, no. 2, p. 1, 2018.

[7] Lanning D R, Harrell G K, Wang J. Dijkstra's algorithm and Google maps[C]//Proceedings of the 2014 ACM Southeast Regional Conference. 2014: 1-3.

[8] Salem I E, Mijwil M M, Abdulqader A W, et al. Flight-schedule using Dijkstra's algorithm with comparison of routes findings[J]. International Journal of Electrical and Computer Engineering, 2022, 12(2): 1675.

[9] Javaid A. Understanding Dijkstra's algorithm[J]. Available at SSRN 2340905, 2013.

[10] Hudak P, Fasel J H. A gentle introduction to Haskell[J]. ACM Sigplan Notices, 1992, 27(5): 1-52.