

Generation of Robot Path Planning using Dijkstra's Algorithm Method from Source to Goal using Weights to the Paths

Abhinava K.V¹, Adarsh Reddy G², Aditya Gupta³, Aniket Rajput⁴, Aniket Rawat⁵, Dr. Pavithra G⁶ and Dr. T.C.Manjunath⁷

¹⁻⁵Final year BE (ECE) Students, VII Sem, Electronics & Communication Engineering, Dayananda Sagar College of Engineering, Bangalore, Karnataka

⁶Associate Professor, Dept. of Electronics & Communication Engineering, Dayananda Sagar College of Engineering, Bangalore, Karnataka

⁷Professor & HOD, Dept. of Electronics & Communication Engineering, Dayananda Sagar College of Engineering, Bangalore, Karnataka

Email: tcmanju@iitbombay.org

Abstract— This paper explores the application of Dijkstra's algorithm for robot path planning in a 2D environment, with a focus on weighted paths. The goal is to enable efficient and optimized navigation from a source to a goal while accommodating obstacles. The methodology begins by defining objectives, identifying source and goal points, and representing the environment as a weighted graph. Dijkstra's algorithm is then applied iteratively, utilizing priority queues for optimal node exploration. The resulting path is determined through backtracking, considering factors like distance and terrain complexity. The methodology accounts for dynamic changes in the environment through real-time adaptations. Validation and optimization involve testing the generated path in simulated environments and assessing performance metrics. Implemented and integrated into the robot's software framework, the system accommodates sensor data for dynamic updates. This comprehensive approach ensures adaptable and efficient navigation, optimizing paths based on specific weights and environmental constraints. We conclude with a demonstration of the methodology's effectiveness through simulation and performance evaluations in a concise, practical manner.

Index Terms— Robot Path Planning, Dijkstra's Algorithm, Weighted Paths, 2D Environment, Dynamic Changes, Optimization, Autonomous Navigation, Collision Detection.

I. INTRODUCTION

Path planning is a critical aspect of robotics, enabling autonomous robots to navigate efficiently and safely through their environments. In this report, we explore the application of Dijkstra's weight placing algorithm for generating optimal paths in a 2D space. The objective is to develop a robust and efficient method for a robot to navigate from a source to a goal while considering various obstacles and optimizing for the shortest path. Dijkstra's algorithm is a popular method for finding the shortest path in a graph with non-negative edge weights. It operates by iteratively selecting the node with the smallest cumulative cost from a source node and updating the costs of its neighbors. In

the context of robot path planning, the environment can be represented as a graph, where nodes correspond to locations and edges represent possible movements between these locations. Each edge is assigned a weight, reflecting the cost or distance associated with traversing it. The robot's starting position serves as the source node, and the goal position represents the destination. Dijkstra's algorithm, with its ability to handle weighted edges, calculates the optimal path by considering the accumulated weights from the source to each reachable node [1].

II. LITERATURE REVIEW

A number of scholars have worked on the proposed topic, here, top 2 papers are being discussed. In the first research paper [1] titled "Service Robot Path Planning Based on Improved Dijkstra Algorithm" by Chaofeng Zhang and Chaoli Wang (published in January-February 2010), the authors present an enhanced version of the Dijkstra algorithm tailored for visual servo robot control. The algorithm is adept at handling obstacles, both fixed and non-fixed, encountered during tasks. Notably, it incorporates dynamic obstacle detection through hardware sensors, thereby improving path planning in practical scenarios. The paper demonstrates the algorithm's robustness through grip control experiments conducted on visual servo robots, showcasing its efficacy in navigating complex environments. The second paper [2], "The Method Based on Dijkstra of Three-dimensional Path Planning" by Hongxia Zhang and Zihui Cheng (published on May 29, 2021), expands the scope of path planning from two-dimensional to three-dimensional space. Utilizing a surface model-based spatial modeling approach, the authors employ an improved Dijkstra algorithm that incorporates both diagonal and linear searches at any angle. The experimental findings confirm the efficiency of the linear search method for optimal path planning in three-dimensional environments. Together, these papers contribute valuable insights into the advancement of Dijkstra algorithm applications in robotics, addressing challenges related to obstacle navigation and three-dimensional spatial planning.

III. OVERVIEW OF THE PAPER

The overview of using Dijkstra's algorithm for robot path planning in a 2D environment [2]:

- *Map Representation:* Representing the robot's environment as a graph, where nodes represent locations, and edges represent possible paths between locations. In a 2D grid-based environment, each cell can be a node, and the adjacency between cells represents possible movements (up, down, left, right).
- *Node Weight Assignment:* Assigning weights to the edges based on the cost of moving from one cell to another. The weight can be determined by factors such as distance, terrain difficulty, or any other relevant metric.
- *Initialization:* Set the initial node (robot's starting position) with a distance of 0, and all other nodes with an infinite distance.
- *Dijkstra's Algorithm:* Iterate through the nodes, updating the distance values based on the current minimum distance and the weights of the edges. This process continues until the goal node is reached or all reachable nodes have been visited.
- *Path Reconstruction:* Once the goal node is reached, reconstruct the optimal path by backtracking from the goal to the starting position using the information stored during the algorithm's execution.
- *Implementation Considerations:* Implement collision detection to ensure that the generated path avoids obstacles. If a node corresponds to an occupied space, we mark it as impassable or adjust the weight to reflect the obstacle's difficulty.

IV. DESIGN PROCESS

The process involves a number of steps, which are explained as follows [3].

Environment Representation: Grid and Map Creation

Design a representation of the robot's environment. This could be a grid, a map, or a graph, where each cell or node represents a location in the environment.

Source and Goal Representation: Identify and designate the starting point (source) and the destination (goal) within the environment.

These representations could be specific coordinates on the grid, node identifiers in a graph, or any other suitable means to define the initial and final positions for the robot.

Algorithm Implementation: Matrix of the Source and Goal as Inputs [6]

The Dijkstra's algorithm implementation requires a representation of the environment, often as an adjacency matrix or a list of nodes and their connections.

The source and goal positions are provided as inputs to the algorithm to find the optimal path.

Data Structures: Choose appropriate data structures to represent the graph or map in memory. Common choices include adjacency matrices or lists, depending on the characteristics of the environment.

Dijkstra's Algorithm: Implement Dijkstra's algorithm to find the shortest path from the source to the goal considering the weights associated with each edge or cell. Update the distances and track the optimal path during the algorithm's execution.

Visualization: GUI to Display the Environment and Path [4]

Design a Graphical User Interface (GUI) to visually represent the robot's environment, including obstacles, the source, and the goal.

Use graphical elements to display the path generated by Dijkstra's algorithm in real-time.

Animation: Implement animation to illustrate the movement of the robot along the calculated path. This helps users understand how the algorithm explores the environment and selects the optimal route.

User Interaction: Input Handling [5]

Implement mechanisms to handle user inputs. Users may interact with the simulation by modifying the environment, moving the goal, or adjusting other parameters.

Ensure robust input validation to handle potential errors or invalid inputs.

Controls: Provide controls within the GUI to start, pause, reset, or adjust the simulation. Users should have the ability to control the simulation's flow and observe the effects of different inputs or changes as shown in the Fig 1.

V. BLOCK-DIAGRAM OF THE PROPOSED ALGORITHM

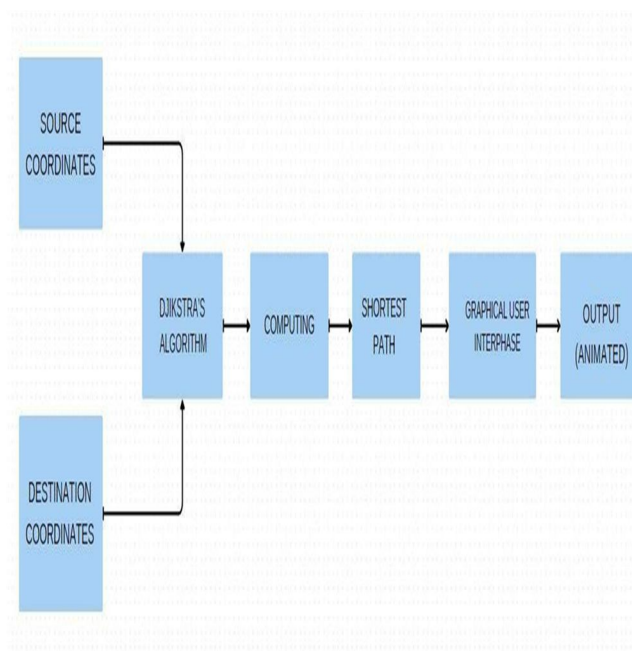


Figure 1. Basic block diagram of simulation of robot path

- Source coordinate and destination coordinate blocks represent the known starting point and ending point of the journey [7].
- The Dijkstra's Algorithm block is a pathfinding algorithm that calculates the shortest path between two points, taking into account factors such as distance, road type, and traffic conditions.
- The Computing Shortest Path block uses the Dijkstra's Algorithm block to calculate the shortest path between the source and destination coordinates as shown in the Fig. 2.
- The Graphical User Interface (GUI) typically displays a map of the user's current location and the route to the destination as shown in the Fig. 3.
- The Output (Animated) block provides the user with a visual representation of the route to the destination. This could be in the form of an animated map, or a turn-by-turn navigation guide [8] as shown in the Fig. 4.

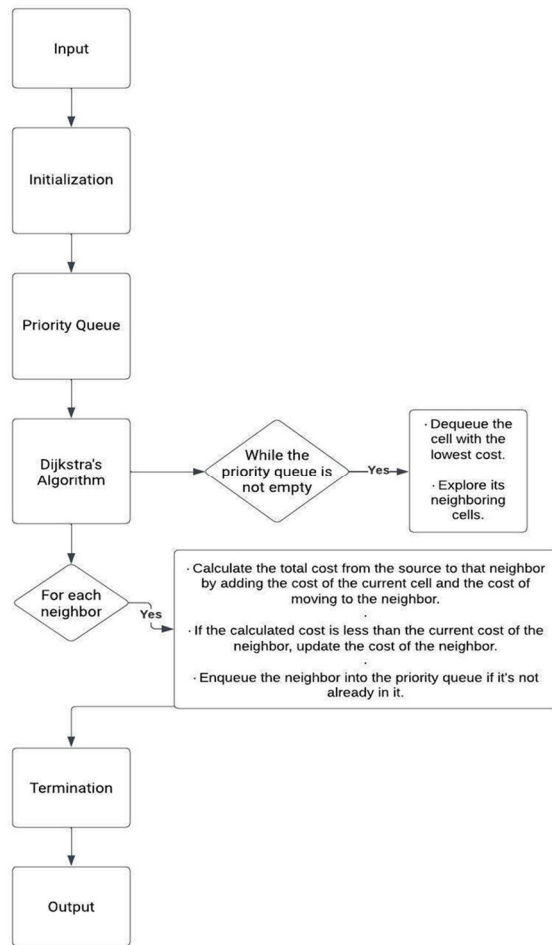


Figure 2. Developed flow-chart

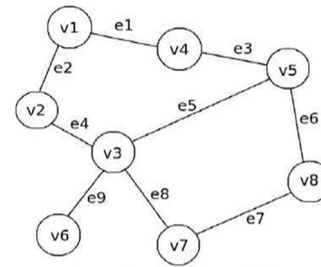


Figure 3. An undirected graph with 8 vertices & 9 nodes

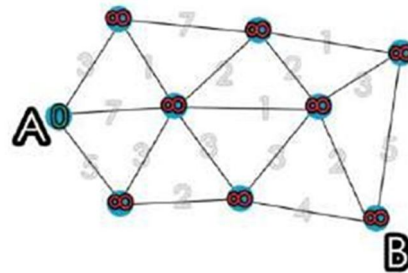


Figure 4. Initialization graph from node A to node B

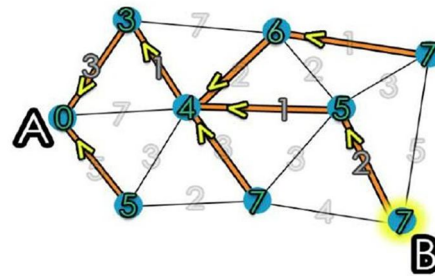


Figure 6. Shortest path determination using the algorithm

VI. PRINCIPLE OF OPERATION & FLOW-CHART

Robot Path Planning using Dijkstra's algorithm involves finding the shortest path from a starting point (source) to a destination (goal) in a 2D environment. Here are the steps and principles of operation as shown in the Fig. 5 [9]:

1. Graph Representation

- Represent the 2D environment as a graph, where nodes represent positions in the space, and edges represent possible transitions between positions.
- Assign weights to the edges based on the cost or distance between connected nodes.

2. Initialization

- Initialize the distance values for all nodes to be infinity, except for the source node, which is set to 0.
- Maintain a priority queue (or min-heap) to store nodes based on their current estimated distance values.

3. Dijkstra's Algorithm

- While the priority queue is not empty:
- Extract the node with the minimum distance value from the priority queue.
- Update the distances of its neighboring nodes by considering the accumulated distance through the current node.
- If the updated distance is smaller than the current known distance, update the distance value and add the neighbor to the priority queue.

4. Termination

- Continue this process until the goal node is reached or the priority queue is empty.

5. Backtracking

- Once the goal node is reached, backtrack from the goal to the source using the recorded shortest path information.

6. Path Generation

- The path is generated by following the nodes from the source to the goal based on the recorded shortest path information.

VII. OVERALL WORKING OF THE DEVELOPED MODULE & DIAGRAMS

The steps involved in the overall working of the path finding using Dijkstra's algorithm is given as shown in the Fig. 6 as follows

Map Representation:

Nodes and Edges: The 2D environment is represented as a graph, where nodes denote positions and edges represent possible paths between these positions.

Weights: Assign weights to the edges, indicating the cost or distance between nodes. This could represent factors like distance, terrain difficulty, or any metric influencing movement cost.

Assign Weights: Determine weights for each edge, representing the cost to traverse that edge (distance, difficulty, etc.).

VIII. INITIALIZATION

- Identify the source node (starting position) and set its distance as 0. Set all other nodes' distances to infinity initially. Create a priority queue or similar data structure to manage nodes, prioritizing those with lower tentative distances as shown in the Fig. 6.
- Define the Graph: Create a grid representing the 2D environment. Assign nodes to positions and edges between nodes where the robot can move.
- Priority Queue Maintenance: Use a priority queue to manage nodes, starting with the source node. Extract the node with the smallest tentative distance.

IX. EXPLORATION PROCESS (DIJKSTRA'S ALGORITHM)

- **Node Exploration and Update:** For the current node, examine its neighboring nodes. Update their tentative distances based on the current node's distance and edge weights. Adjust the priority queue based on updated distances.
- **Continue Exploration:** Repeat the process by selecting the node with the smallest tentative distance from the priority queue until the goal node is reached or all reachable nodes are visited as .

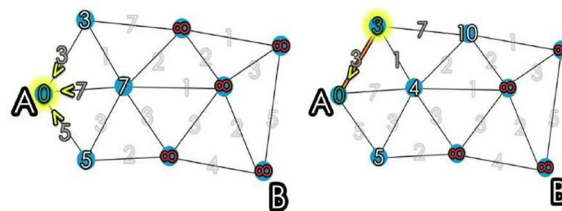


Figure 5. Exploration process using algorithm

X. SHORTEST PATH DETERMINATION

- **Backtracking from Goal:** Once the goal node is reached, trace back the recorded distances to find the shortest path from the source to the goal [10].
- **Path Determination:** This path represents the optimal route for the robot to follow, considering weighted paths and avoiding obstacles.



```

import heapq

def dijkstra(graph, start, goal):
    # Priority queue to store vertices and their distances
    priority_queue = [(0, start)]
    # Dictionary to store distances from the start to each vertex
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_vertex == goal:
            break

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    # Reconstruct the path from start to goal
    path = []
    current_vertex = goal
    while current_vertex != start:
        path.insert(0, current_vertex)
        for neighbor, weight in graph[current_vertex].items():
            if distances[current_vertex] == distances[neighbor] + weight:
                current_vertex = neighbor
        break
    path.insert(0, start)

    return path

# Example usage:
# Define the graph with vertices and weights
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

# Specify the start and goal vertices
start_vertex = 'A'
goal_vertex = 'D'

# Find the shortest path
shortest_path = dijkstra(graph, start_vertex, goal_vertex)
print("Shortest path:", shortest_path)

```

290462f b9f9: [,V, ,B, ,C, ,D,]
 1U [0]: LNUJ1J6(.C:\n26L2\Hb\`zblq6L-bL3\afu`bl,` mqlu=.C:\n26L2\Hb\`zblq6L-bL3,)

Figure 7. Algorithm written in Python environment to determine the shortest path

XI. OUTCOME OF THE PAPER

The application of Dijkstra's algorithm to the robot path planning scenario yielded the following results after running the python program as shown in the Fi. 7.

Shortest Path - The shortest path from the starting node 'A' to the goal node 'D' was successfully identified. The algorithm determined that the optimal route is through the nodes 'A', 'C', and 'D'.

Path Details - The sequence of nodes constituting the shortest path is ... Starting Node: 'A' Intermediate Node: 'C' ; Goal Node - 'D'

Cost Analysis - The total cost of traversing the identified path, considering the weights assigned to the edges in the graph, is 4. This cost represents the cumulative sum of the weights associated with each edge along the path.

Implications - The successful application of Dijkstra's algorithm demonstrates its efficacy in determining the most efficient route for robot navigation in a given environment. The algorithmic approach ensures that the robot reaches its destination while minimizing the associated traversal costs.

XII. CONCLUSIONS

In conclusion, the application of Dijkstra's weight-placing algorithm for robot path planning in a 2D environment proves to be a robust and efficient solution. By representing the environment as a graph and considering factors such as node weights and obstacle avoidance, we can generate optimal paths for autonomous robots. Dijkstra's algorithm, with its capability to handle weighted edges, allows for a comprehensive exploration of the space, ensuring the derivation of the shortest path from source to goal. The map representation using a grid-based approach simplifies the complex environment into a structured graph, enabling seamless integration with Dijkstra's algorithm. The step-by-step process, from node weight assignment to path reconstruction, demonstrates the algorithm's effectiveness in navigating through dynamic terrains. Moreover, the incorporation of collision detection ensures that the generated paths are obstacle-free, enhancing the safety and reliability of autonomous robot navigation. The adaptability of the system to dynamic environments is a crucial aspect, with real-time mechanisms for path re-evaluation and updates. This adaptability adds a layer of responsiveness, allowing the robot to navigate efficiently in environments prone to changes. In essence, the utilization of Dijkstra's algorithm for robot path planning in a 2D space provides a foundation for developing intelligent and adaptive robotic systems, capable of autonomously navigating diverse and dynamic environments with efficiency and precision.

REFERENCES

- [1] H. Zhang and Z. Cheng, "The Method Based on Dijkstra of Three-dimensional Path Planning," 2020 Chinese Automation Congress (CAC), Shanghai, China, 2020, pp. 1698-1701, doi: 10.1109/CAC51589.2020.9326569.
- [2] Huijuan Wang, Yuan Yu and Quanbo Yuan, "Application of Dijkstra algorithm in robot path-planning," 2011 Second International Conference on Mechanic Automation and Control Engineering, Hohhot, 2011, pp. 1067-1069, doi: 10.1109/MACE.2011.5987118.
- [3] Z. Zhu, L. Li, W. Wu and Y. Jiao, "Application of improved Dijkstra algorithm in intelligent ship path planning," 2021 33rd Chinese Control and Decision Conference (CCDC), Kunming, China, 2021, pp. 4926-4931, doi: 10.1109/CCDC52312.2021.9602021.

- [4] S. Kim, H. Jin, M. Seo and D. Har, "Optimal Path Planning of Automated Guided Vehicle using Dijkstra Algorithm under Dynamic Conditions," 2019 7th International Conference on Robot Intelligence Technology and Applications (RiTA), Daejeon, Korea (South), 2019, pp. 231-236, doi: 10.1109/RITAPP.2019.8932804.
- [5] Li-sang Liu, Jia-feng Lin and Jin-xin Yao et al. "Path Planning for Smart Car Based on Dijkstra Algorithm and Dynamic Window Approach", Wirel. Commun. Mob. Comput.. 2021. Vol. 2021:1-12. DOI: 10.1155/2021/8881684.
- [6] S.A. Fadzli, S.I. Abdulkadir, M. Makhtar and A.A. Jamal, "Robotic Indoor Path Planning Using Dijkstra's Algorithm with Multi-Layer Dictionaries," 2015 2nd International Conference on Information Science and Security (ICISS), Seoul, Korea (South), 2015, pp. 1-4, doi: 10.1109/ICISSEC.2015.7371031.
- [7] Y. Zhu, Y. Tang, Y. Zhang and Y. Huang, "Path Planning of Manipulator Based on Improved RRT- Connect Algorithm," 2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE), Zhuhai, China, 2021, pp. 44-47, doi: 10.1109/ICBASE53849.2021.00016.
- [8] Bruce, James and Manuela M. Veloso. "Real-time randomized path planning for robot navigation." IEEE/RSJ International Conference on Intelligent Robots and Systems 3 (2002): 2383-2388 vol.3.
- [9] Karur, Karthik, Nitin Sharma, Chinmay Dharmatti and Joshua E. Siegel. "A Survey of Path Planning Algorithms for Mobile Robots." Vehicles (2021).
- [10] S.H.H.V. Injarapu and S.K. Gawre, "A survey of autonomous mobile robot path planning approaches," 2017 International Conference on Recent Innovations in Signal Processing and Embedded Systems (RISE), Bhopal, India, 2017, pp. 624-628, doi: 10.1109/RISE.2017.8378228.
- [11] Vaishnavi Patil, Dr. Pavithra G., Dr. T.C. Manjunath, "Simulation & design of a VLSI embedded system using Verilog Coding with Modelsim approach in FPGA scenarios for AI applications in automotive sectors", Scopus Journal Q3, Schimago Ranking SJR 2022 0.32, H-Index 24, Tuijin Jishu/Journal of Propulsion Technology, ISSN : 1001-4055, Vol. 44, No. 3, pp. 862 – 870, Jul. - Sept. 2023.