Application of Dijkstra Algorithm for Robot's Obstacle Avoidance System in A Simulated Environment Using ROS (Robot Operating System) and Gazebo Simulator

Farrel Ahmad - 13520110
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13520110@std.stei.itb.ac.id

Abstract—**Background : Navigation, especially in robotics is important for the safe and efficient movement of the robot. The robot must be able to travel to certain destination as efficient as possible. Safe movement for the robot means that the robot must avoid any obstacle while also finding the shortest path. Dijkstra algorithm is well-known algorithm for finding the shortest path. In addition, Dijkstra is useful for not only the shortest path but also the safest path for robot's path planning system.**

Methods : The implementation is based on the basic graph theory and Dijkstra algorithm. The visibility graph is what will the Dijkstra process to find the shortest path. ROS (Robot Operating System) is the program that will instruct the robot movement and Gazebo Simulator will simulate the movement based on the ROS program.

Result : Dijkstra algorithm successfully process the graph with adjacency matrix as the input of the visibility graph. The robot will travel with the shortest distance from start point to goal point in the simulator according to the generated node sequence.

Conclusion : From the result, Dijkstra is proven to be useful in for Obstacle Avoidance System implementation. That is, finding the shortest path from the visibility graph that shows possible and safe passage for the robot.

***Keywords*—Navigation, Dijkstra, Graph, Path Planning**

I. Background

Navigation is one of the most important aspect in robotics. This is because robot must be able to travel to the certain point or destination quickly and safely. During the process of reaching the destination, the robot should be able to calculate and determine the best route possible. The things that define "best route" are the shortest and safest route. The shortest route means that it has to travel with minimum distance possible while the safest route means it has to avoid any obstacle nearby. This method is called path planning.

Generally, programming in robot is divided into two level. These are low-level implementation program and high-level implementation program. Each has their own purpose and work in parallel. The robot type that will be used is a wheeled robot.

The low-level implementation program is anything that closely related to the robot's hardware like motor, vision camera, sensor, motors, and microcontrollers. It can be divided into two subsystem, perception system and locomotion system.

Perception system in low-level implementation controls the vision camera and sensors to output the data that will be processed later in high-level implementation program. Perception system is not only in low-level implementation, but as mentioned before, it is also in high-level implementation because the robot must translate the data into something that can be interpreted (position, velocity, etc).

Locomotion system is used to control the robot's movement. Robot's movement must be accurate without any overshooting and undershooting. This is usually done by using control system within locomotion system. One of the most common control system is PID (Proportional Integral Derivative) controller. The locomotion system also utilizes microcontroller to control how much PWM (Pulse Width Modulation) needed for the motors to move.

The high-level implementation program is anything that is not closely related to the hardware and mainly used for how the robot thinks. When the robot receives the data from perception system's hardware, the data is then converted into the robot's representation of the world by using world model. In the world model, the robot will be able to know its current position, current velocity, and nearby obstacle. When the robot is commanded to move to the certain position, the path planning system will compute the best route possible by using the world model's information. By combining the path result from path planning system and world model's information, the robot will know how it will move to the goal position.

For example, the robot is commanded to move 1 meter forward from an idle position and there's an obstacle ahead. The perception's system will capture the current position based on view and obstacle ahead. After that the data will be sent to world model to know the robot's coordinate of current position, coordinate of the obstacle, and coordinate of the goal. Then, the path planning system will compute the best route without hitting any obstacle. Further calculation is needed to calculate the speed needed to reach that position. This is

usually done by motion planner system. The calculation will be sent to locomotion system where the motor will rotate the wheel and move the robot to reach the speed needed, then reduce the speed, and eventually stop at the goal position.

Dijkstra algorithm is a well-known algorithm to find the shortest route possible from a certain point to another point by using weighted graph. The idea is by using the graph generated by the robot as the representation of the reachable position, the Dijkstra algorithm can find the shortest route while also avoiding any obstacle. Obstacle avoidance system is a crucial system because if the robot hits obstacle, it can be dangerous not only for the robots, but for the obstacle itself that also might be a human, animals, etc.

II. Methods

A. Graph Theory

Graph is a representation of discrete objects and its relations [1]. Graph is G = (V,E). V is set of vertex and E is set of edges.

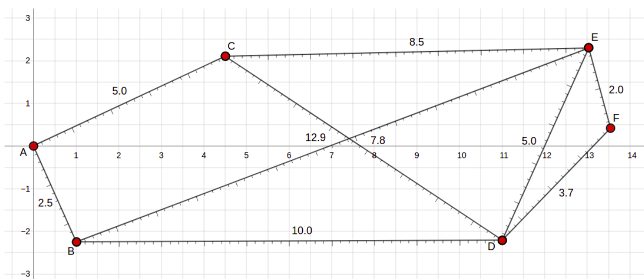


Fig.1 **Route Map Example**

For example, the picture above is a graph that represent a map with routes. Each point (A, B, C, D, E, and F) is a vertex. Vertex is also usually called as node. Each line that connects two vertexes is an edge. If two vertexes are connected by an edge, it means the two vertexes are related. In the picture above, the robot can go from point A to point C because they are connected but cannot go from point A to point F because they are not connected. Because this is weighted graph, each edge is labeled with cost. Because this is a route map, the cost is in form of distance unit (meters)

The graph can be represented in many ways like adjacency matrix, adjacency list, and incidency list. For this experiment, the map will use adjacency matrix.

B. Dijkstra Algorithm

Dijkstra Algorithm is an algorithm to find the shortest path. Using the graph from Picture 1, the shortest path to each point can be computed using this algorithm. The algorithm works as follows (using Picture 1 Graph as example):

1. Make a list of shortest distance of each node and set all of the elements to infinity, with index 0 corresponds to A, index 1 to B, and so on. The length is total nodes. For example,
   dist = [INF,INF,INF,INF,INF,INF].
2. Make a list of visited node boolean and set all of the elements as false. The length is total nodes. For example,
   vis = [false,false,false,false,false,false].
3. For this example, the algorithm will start at point A. Then, distance to point A is set to 0.0 because it is a starting point. Thus dist[0] = 0.0
   dist = [0.0,INF,INF,INF,INF,INF]
   vis = [false,false,false,false,false,false].
4. Find node in dist list with shortest distance and has not been visited. In this case from previous step, is node A.
5. Currently is checking node A. Find nodes that connect to point A and replace the distance value of the corresponding nodes in distance list if the value + node A is less than the value of the node in distance list. After that, mark node A as visited.
   (curr_node : A)
   dist = [0.0, 2.5, 5.0, INF, INF, INF]
   vis = [true, false, false, false, false, false]
6. Repeat like step 4, find node in dist list with shortest distance and has not been visited. In this case from previous step, is node B.
7. Currently is checking node B, find nodes that connect to point B and replace the distance value of the corresponding nodes in distance list if the value+nodeB is less than the value of the node in the distance list. After that, mark node B as visited.
   (curr_node : B)
   dist = [0.0, 2.5, 5.0, 12.5, 15.4, INF]
   vis = [true, true, false, false, false, false]

   note : 12.5 = 2.5 + 10.0 and 15.4 = 2.5 + 12.9
8. Repeat like step 6 until all vis list is true (all node has been visited).
   (curr_node : C)
   dist = [0.0, 2.5, 5.0, 12.5, 13.5, INF]
   vis = [true, true, true, false, false, false]

   note : node E’s distance replaced
9. Next iteration:
   (curr_node : D)
   dist = [0.0, 2.5, 5.0, 12.5, 13.5, 16.2]
   vis = [true, true, true, true, false, false]
10. Next iteration:
    (curr_node : E)
    dist = [0.0, 2.5, 5.0, 12.5, 13.5, 15.5]
    vis = [true, true, true, true, true, false]

11. Next iteration:
    (curr_node : F)
    dist = [0.0, 2.5, 5.0, 12.5, 13.5, 15.5]
    vis = [true, true, true, true, true, true]

12. Result :
    dist = [0.0, 2.5, 5.0, 12.5, 13.5, 15.5]
    vis = [true, true, true, true, true, true]

13. Process ends.

Based on the result, each element in dist list is the shortest distance of each node. For example, the shortest distance to point D (Index 3) is 12.5 distance unit. However, with that information, the sequence of which node should be visited to reach the designated final point is still unknown. Thus the list of previous node is needed to find the sequence of the shortest route. The additional list is previous list with initial value as -1 and with the length of total nodes. For example,

prv = [-1, -1, -1, -1, -1 , -1]

Using the steps as mentioned before. The additional step is as follows,

5. (curr_node : A)
   dist = [0.0, 2.5, 5.0, INF, INF, INF]
   vis = [true, false, false, false, false, false]
   prv = [-1, 0, 0, -1, -1 , -1]

   note : assign prev node of B and C as A (index 0)

7. (curr_node : B)
   dist = [0.0, 2.5, 5.0, 12.5, 15.4, INF]
   vis = [true, true, false, false, false, false]
   prv = [-1, 0, 0, 1, 1, -1]

8. (curr_node : C)
   dist = [0.0, 2.5, 5.0, 12.5, 13.5, INF]
   vis = [true, true, true, false, false, false]
   prv = [-1, 0, 0, 1, 2, -1]

9. (curr_node : D)
   dist = [0.0, 2.5, 5.0, 12.5, 13.5, 16.2]
   vis = [true, true, true, true, false, false]
   prv = [-1, 0, 0, 1, 2, 3]

10. (curr_node : E)
    dist = [0.0, 2.5, 5.0, 12.5, 13.5, 15.5]
    vis = [true, true, true, true, true, false]
    prv = [-1, 0, 0, 1, 2, 4]

11. (curr_node : F)
    dist = [0.0, 2.5, 5.0, 12.5, 13.5, 15.5]
    vis = [true, true, true, true, true, true]
    prv = [-1, 0, 0, 1, 2, 4]

12. Result :
    prv = [-1, 0, 0, 1, 2, 4]

From the result, the order of the node to reach the certain final point can be found by backtracking the prv list. This is because the previous node is guaranteed to be in the shortest distance. For example, to find the order of node from point A to point F. The value of point F (index 5) in prv is 4, means that the previous node before index 5 is index 4 (node E). Then the previous of node E (index 4) is node A (index 0) based on prv list. The value of index 0 is -1, means that the backtracking process has finished and reached the starting position. The sequence is $0 \rightarrow 2 \rightarrow 4 \rightarrow 5$. In order to reach node F (Index 5), the system (or robot) will follow the generated sequence.

C. ROS (Robot Operating System)

The Robot Operating System (ROS) is a set of libraries and tools that can help users build robot program mainly in C++ or python or both [5]. Although the name has “Operating System” in it. It is actually not an operating system. It is actually closer to a framework that is commonly used in robotics world.

The program works by having nodes that can communicate to each other by publish-subscribe method and server-client (service-request) method.

D. Gazebo Simulator

Gazebo Simulator is a 3D robot simulator that is usually used for robotics. It can simulate accurate robot movements and simulate as in controlling real robot. During robot development, it is important to use simulator to simulate the performance of the robot first before testing it directly to the robot. Testing it directly in the robot is more risky because if there is a bug, the robot can be in a state of unwanted behavior like going out of control or anything dangerous. The ROS program that has been made can be connected to Gazebo Simulator to test the code.

E. Robot Specification

The robot that will be used in a simulator is based on a real world robot called ROBOTIS TurtleBot3 Burger. This is a small kit robot that is usually used for educational purpose. The robot has a dimension of 13.8 cm x 17.8 cm x 19.2 cm (L x W x H). The length and width size of the robot will determine the offset of the node’s position.

F. Visibility Graph

The idea for robot’s path planner is using visibility graph. That is, each node is the edge of the obstacle and the edge of the graph connects two nodes that are visible. However because robot has size and mass, the path needs to move a little bit away from the obstacle to give spaces for the robot.

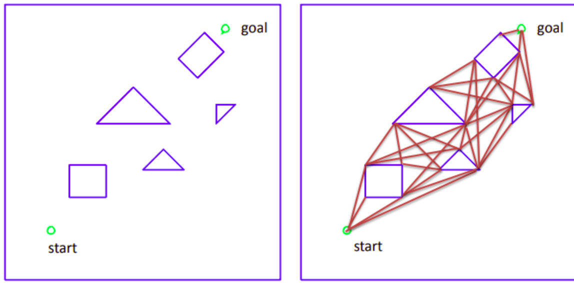


Fig.2 **Visibility Graph Example**
Source : **Columbia University COMS W4733**

G. Route Map Experiment

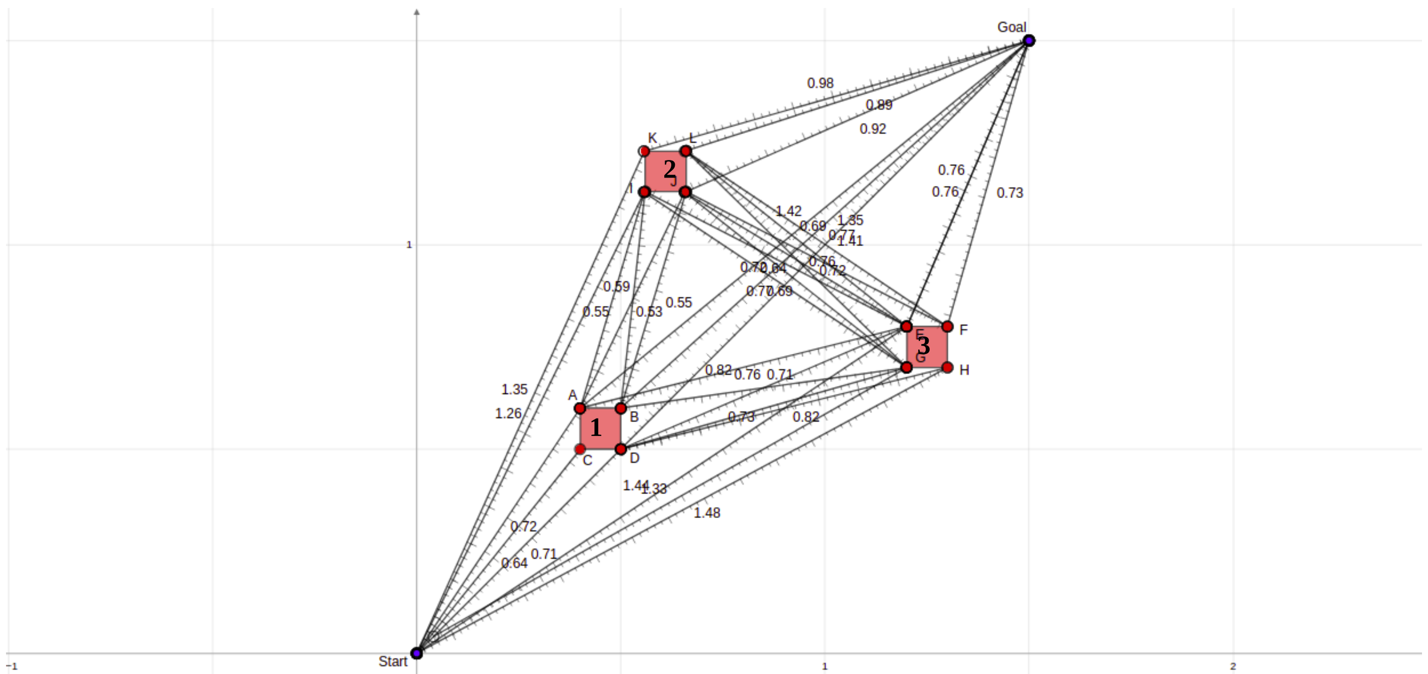
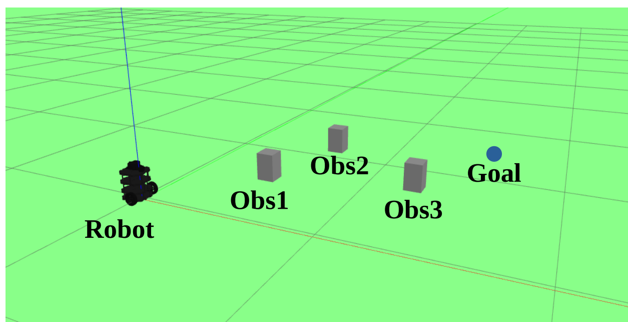


Fig.3 **Route Map Experiment**

For this experiment, the simulator only simulates the robot's shortest path using Dijkstra algorithm and its movement from each node sequentially. In addition, the simulation does not simulate the sensors needed to make a visibility graph. Thus, the graph is assumed ready to use, making the robot able to compute the shortest path.

The map consist of three squares with size 0.1m x 0.1m, the start point (0,0), and the goal point (1.5,1.5). The squares in the map will be a 0.1m x 0.1m x 0.2m cube in Gazebo Simulator to simulate the obstacle. Fig.3 shows the map setup.

H. Simulator World



Fig,4 **Simulator World**

The simulator world is the setup of the experiment. It will spawn the robot at the point of origin (0.0, 0.0), Obstacle 1 at (0.45, 0.55), Obstacle 2 at (0.61, 1.18), and Obstacle 3 at (1.25, 0.75). The goal point will be (1.5, 1.5). The unit of coordinate are in meters. The world in Fig.4 is based on the map in Fig.3.

I. Expanded Obstacle

However, the route map in Fig.3 only treats the robot as a single point or particle. Robot has dimension, in which it needs plenty of space to move around the obstacle. To overcome this problem, the obstacle in the graph in which the robot reads should be expanded. This will give spaces for the robot when going to certain node or around obstacles.

As mentioned before, the robot has a width of 17.8 cm. It is fine to assume that the robot has 17.8 cm in diameter or 8.9 cm in radius. In theory, the obstacle in the graph needs to be expanded to (original_length + 0.089 m + 0.089 m) . It is ideal to round the expansion from 0.089 m to 0.1 m thus each side of the obstacle will be expanded to (original_length + 0.1 m + 0.1 m). Because of the size change in the obstacle, the coordinate of each node were moved making the edges in the graph a little bit different than the original graph. Fig.5 shows the expanded obstacle in the graph.

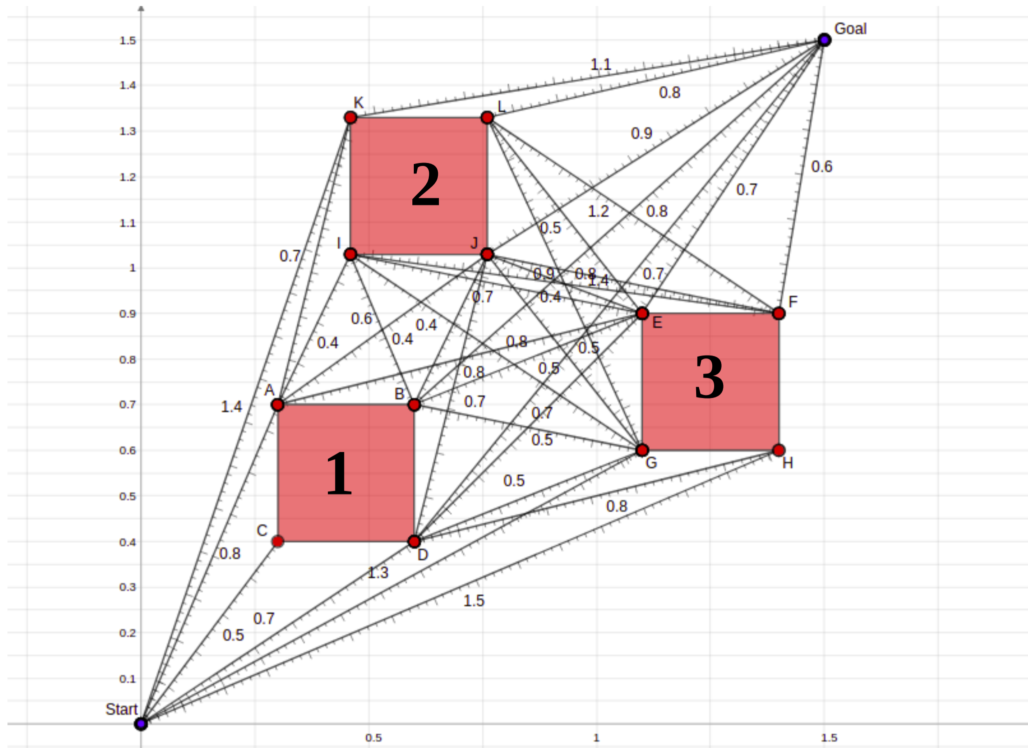


Fig.5 **Route Map (Expanded Obstacles)**

III. Result

A. Adjacency Matrix

```
double graph[14][14]={
/*S*/ {0.0, 0.5, 0.7, 0.8, 0.0, 1.3, 1.5, 0.0, 0.0, 0.0, 0.0, 1.4, 0.0, 0.0},
/*C*/ {0.5, 0.0, 0.3, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
/*D*/ {1.3, 0.3, 0.0, 0.0, 0.3, 0.5, 0.8, 0.7, 0.0, 0.0, 0.7, 0.0, 0.0, 1.4},
/*A*/ {0.8, 0.3, 0.0, 0.0, 0.3, 0.0, 0.0, 0.8, 0.0, 0.4, 0.6, 0.7, 0.0, 0.0},
/*B*/ {0.0, 0.0, 0.3, 0.3, 0.0, 0.5, 0.0, 0.5, 0.0, 0.4, 0.4, 0.0, 0.0, 1.2},
/*G*/ {1.3, 0.0, 0.5, 0.0, 0.5, 0.0, 0.3, 0.3, 0.0, 0.8, 0.5, 0.0, 0.9, 0.0},
/*H*/ {1.5, 0.0, 0.8, 0.0, 0.0, 0.3, 0.0, 0.0, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0},
/*E*/ {0.0, 0.0, 0.7, 0.0, 0.5, 0.3, 0.0, 0.0, 0.3, 0.7, 0.4, 0.0, 0.5, 0.7},
/*F*/ {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.3, 0.0, 0.9, 0.7, 0.0, 0.8, 0.6},
/*I*/ {0.0, 0.0, 0.0, 0.4, 0.4, 0.8, 0.0, 0.7, 0.9, 0.0, 0.3, 0.3, 0.0, 0.0},
/*J*/ {0.0, 0.0, 0.0, 0.6, 0.4, 0.5, 0.0, 0.4, 0.7, 0.3, 0.0, 0.0, 0.3, 0.9},
/*K*/ {1.4, 0.0, 0.0, 0.7, 0.0, 0.0, 0.0, 0.0, 0.0, 0.3, 0.0, 0.0, 0.3, 1.1},
/*L*/ {0.0, 0.0, 0.0, 0.0, 0.0, 0.8, 0.0, 0.5, 0.8, 0.0, 0.3, 0.3, 0.0, 0.8},
/*GL*/{0.0, 0.0, 1.4, 0.0, 1.2, 0.0, 0.0, 0.7, 0.6, 0.0, 0.9, 1.1, 0.8, 0.0}
//      S    C    D    A    B    G    H    E    F     I    J    K    L   GL
};
```

Fig.6 **Adjacency Matrix**

Fig.6 shows the adjacency matrix of the expanded obstacle graph (Fig.5) with zero is not connected and non-zero is connected with the value as the distance of connected nodes.

B. Program Execution

Source Code : https://github.com/farrel-a/robot-nav-simulator

Full documentation and source code are available from the link above. The documentation consist of program setup and execution tutorial. Read the "Dijkstra Obstacle Avoidance" by clicking it at the "Table of Contents" part because that is the program (called `robot_2`) that will be used in the Dijkstra Obstacle Avoidance System as this repository also holds another program for Fig.1 implementation (called `robot_1`).

Once the program runs, the terminal will show some informations as follows

```
❯ rosrun navrobot_gazebo robot_2
Prev List: -1 0 0 0 2 2 0 2 7 3 2 0 11 2
Node Sequence : S --> D --> GL
```

Fig.7 **Terminal Log**

Prev List is the prv list as explained in the previous part where every element is the nearest node to that node index. For example Node Goal is node at index 13 according to the documentation. The value is 2 meaning that the shortest edge to node Goal or node 13 is node 2 or node D. The node sequence is the sequence that the robot must travel between nodes. Starting from node S, the robot will travel to node D and then will directly to node GL or node goal. This sequence is the shortest and safest route generated by Dijkstra algorithm.

Fig.8 shows the robot moving to the goal point and avoiding any obstacles by following the generated path. Configuration for the robot's linear speed and angular speed are also configurable in `robot_2.cpp`.

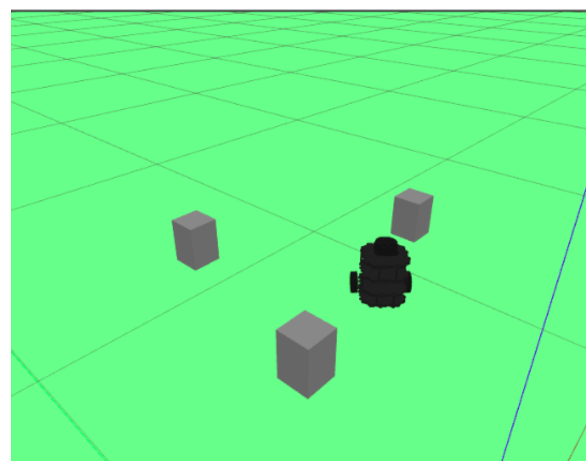


Fig.8 **Robot Navigating through Obstacles**

IV. CONCLUSION

The Dijkstra implementation is useful for navigating through obstacles. In real world application, sensors or vision camera are needed to generate the visibility graph. Using the visibility graph, the algorithm can find the shortest route possible while also avoiding any obstacles as shown in the execution result.

REFERENCES

[1] Munir, Rinaldi. “Graf-2020-Bagian-1”, https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf, Accessed on 12 December 2021.

[2] Educative.io, https://www.educative.io/edpresso/how-to-implement-dijkstras-algorithm-in-cpp. Accessed on 12 December 2021.

[3] Fiset, William. “Dijkstra Shortest Path Algorithm | Graph Theory”. https://www.youtube.com/watch?v=pSqmAO-m7Lk. Accessed on 13 December 2021.

[4] GeeksforGeeks. “Dijkstras Shortest Path Algorithm Greed Algo”. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/ . Accessed on 13 December 2021.

[5] ROS. “Ros Documentation”. http://wiki.ros.org/Documentation. Accessed on 13 December 2021.

[6] CS Columbia University. “Robot Path Planning”. http://www.cs.columbia.edu/~allen/F17/NOTES/lozanogrown.pdf.. Accessed on 13 December 2021.

[7] Robot Advance. “Robotis TurtleBot3 Burger”. https://www.robot-advance.com/EN/art-turtlebot3-burger-1997.htm. Accessed on 14 December 2021.

DECLARATION

I hereby declare that my paper is my own writing, not a summary, nor a translation from other’s writing, and not a plagiarism.

Bandung, 14 December 2021

Farrel Ahmad - 13520110