Shortest Path-Finding Robot

Experiment Report & Analysis

Team Members:

Ahmad Waleed, Om Patel

March 18th, 2023

Implementation provided by:

Dr. Adam Gaweda

Ahmad Waleed

Om Patel

Methodology

Measuring Runtimes:

The following code snippet shows how the actual runtime of the robot finding the shortest path
In the following code, the actual runtime of the robot finding the shortest path is being measured. This is done using the Java System.currentTimeMillis() method. At the beginning of the main method, the current time in milliseconds is stored in the startTime variable using System.currentTimeMillis(). Then an instance of RunSimulation is created with the map input and 100 iterations. Then the run method is called using the instance of RunSimulation that was created prior. This will ask the robot to find the shortest path (find the target). After this method is called the current time in milliseconds is stored in the endTime variable using the System.currentTimeMillis() method. The difference between the startTime and the endTime variable will represent the actual runtime for the robot to find the target. This runtime will provide an insight into the algorithm's efficiency and performance which the robot uses. This runtime will be used to determine if the theoretical and practical analyses are similar.

```
public static void main ( final String[] args ) {
    // Currently uses the first public test case
    String mapFile = "maps/public/1250Nodes.txt";
    int iterations = 100;
    double startTime = System.currentTimeMillis();
    RunSimulation sim = new RunSimulation( mapFile, iterations );
    sim.run();
    double endTime = System.currentTimeMillis();
    double elapsedTime = endTime - startTime;
    System.out.println( elapsedTime );

}
```

Algorithm Design

```
/**
* Dijkstra’s algorithm finds the shortest path in a 2D grid. The algorithm starts with the robot’s position
* and maintains a priority queue of positions yet to be evaluated, sorted by cost(g score). It evaluates each neighbor
* updating costs and parent's information and adding them to the queue. The process continues until the target
* position is reached or the queue is empty. The shortest path is constructed by backtracking from the target to start *
using parent pointers
*/
function dijkstra():
    // Get the current position of the robot
    selfPos = getRobotPosition()

    // Get the target position
    targetPos = getTarget()

    // Initialize priority queue for nodes to be evaluated
    yetToBeEvaluated = PriorityQueue()

    // Initialize set for evaluated nodes
    evaluatedAlready = Set()

    // Create starting position node with zero cost
    startingPos = Node(selfPos, 0, null)

    // Add starting position node to priority queue
    add startingPos to yetToBeEvaluated

    // Add starting position node to mapAsNodes
    add startingPos to mapAsNodes

    // Loop until the target is reached or the priority queue is empty
    while yetToBeEvaluated is not empty:
        // Get the node with the lowest cost from the priority queue
        currentPathToTarget = poll from yetToBeEvaluated

        // Check if the current node is the target node
        if currentPathToTarget.pos equals targetPos:
            break

        // Add the current node position to the set of evaluated nodes
        add currentPathToTarget.pos to evaluatedAlready
```

```
        // Get the neighboring positions of the current node
        neighbors = getNeighborPositions(currentPathToTarget.pos)

        // Get the tiles in the environment
        tiles = getTiles()

        // Iterate over the neighbors of the current node
        for each neighbor in neighbors:
            // Get the neighbor's position
            currentPos = neighbor position

            // Get the tile at the neighbor's position
            tile = tiles[currentPos]

            // Get the status of the tile at the neighbor's position
            status = tile status

            // Calculate the cost to reach the neighbor
            cost = currentPathToTarget.g + 1

            // Skip the neighbor if it has already been evaluated
            if currentPos is in evaluatedAlready:
                continue

            // Get the node corresponding to the neighbor's position
            neighborNode = mapAsNodes[currentPos]

            // If the neighbor node does not exist and is passable, create a new node
            if neighborNode is null and status is not IMPASSABLE:
                neighborNode = Node(currentPos, cost, currentPathToTarget)
                add neighborNode to mapAsNodes

            // If the new cost is lower than the previous cost and is impassable, update the neighbor node
            else if cost < neighborNode.g and status is not IMPASSABLE:
                neighborNode.g = cost
                neighborNode.parent = currentPathToTarget

            // Add the neighbor node to the priority queue if it is not already there and is impassable
            if neighborNode is not in yetToBeEvaluated and status is not IMPASSABLE:
                add neighborNode to yetToBeEvaluated

    // Return a "do nothing" action, as the shortest path is stored in the currentPathToTarget
    return Action.DO_NOTHING
```

Algorithm Analysis

Algorithm Analysis:

Algorithm:	**Analysis/Runtime Rationale**
/** * Dijkstra's algorithm finds the shortest path in a 2D grid. The algorithm starts with the robot's position * and maintains a priority queue of positions yet to be evaluated, sorted by cost(g score). It evaluates each neighbor * updating costs and parent's information and adding them to the queue. The process continues until the target * position is reached or the queue is empty. The shortest path is constructed by backtracking from the target to start * using parent pointers */ function dijkstra(): 1 // Get the current position of the robot 2 selfPos = getRobotPosition() **-> O(1)** 3 4 // Get the target position 5 targetPos = getTarget() **-> O(1)** 6 7 // Initialize priority queue for nodes to be evaluated 8 yetToBeEvaluated = PriorityQueue() **-> O(1)** 9 10 // Initialize set for evaluated nodes 11 evaluatedAlready = Set() **-> O(1)** 12 13 // Create starting position node with zero cost 14 startingPos = Node(selfPos, 0, null) **-> O(1)** 15 16 // Add starting position node to priority queue 17 add startingPos to yetToBeEvaluated **-> O(log V)** 18 19 // Add starting position node to mapAsNodes 20 add startingPos to mapAsNodes **-> O(1)** 21 22 // Loop until the target is reached	selfPos = getRobotPosition() - This line fetches the current position of the robot, which takes O(1) time targetPos = getTarget() - This line fetches the target position, which takes O(1) time. yetToBeEvaluated = PriorityQueue() - This line initializes an empty priority queue, which takes O(1) time. evaluatedAlready = Set() - This line initializes an empty set, which takes O(1) time. startingPos = Node(selfPos, 0, null) - This line creates a node for the starting position with a cost of zero and no parent, which takes O(1) time. add startingPos to yetToBeEvaluated - This line adds the starting node to the priority queue, which takes O(log V) time. add startingPos to mapAsNodes - This line adds the starting node to the mapAsNodes data structure, which takes O(1) time. while yetToBeEvaluated is not empty: - This while loop runs until the priority queue is empty, which takes at most O(V) iterations. currentPathToTarget = poll from yetToBeEvaluated - This line extracts the node with the lowest cost from the priority queue, which takes O(log V) time.

```
23    // or the priority 23queue is empty
24    while yetToBeEvaluated is not empty: -> O(V)
25        // Get the node with the lowest cost from the
26        // priority queue
27        currentPathToTarget = poll from yetToBeEvaluated -> O(log V)
28
29        // Check if the current node is the target node
30        if currentPathToTarget.pos equals targetPos: -> O(1)
31            break
32
33        // Add the current node position to the set of
34        // evaluated nodes
35        add currentPathToTarget.pos to evaluatedAlready -> O(1)
36
37        // Get the neighboring positions of the current
38        // node
39        neighbors = getNeighborPositions(currentPathToTarget.pos) -> O(E)
40
41        // Get the tiles in the environment
42        tiles = getTiles() -> O(1)
43
44        // Iterate over the neighbors of the current node
45        for each neighbor in neighbors: -> O(E)
46            // Get the neighbor's position
47            currentPos = neighbor position
48
49            // Get the tile at the neighbor's position
50            tile = tiles[currentPos]
51
52            // Get the status of the tile at the neighbor's position
53            status = tile status
54
55            // Calculate the cost to reach the neighbor
56            cost = currentPathToTarget.g + 1
57
58            // Skip the neighbor if it has already been evaluated
59            if currentPos is in evaluatedAlready: -> O(1)
60                continue
61
62            // Get the node corresponding to the neighbor's position
63            neighborNode = mapAsNodes[currentPos]
64
65            // If the neighbor node does not exist and ispassable,
```

if currentPathToTarget.pos equals targetPos: - This if statement checks if the current node is the target node, which takes O(1) time.

add currentPathToTarget.pos to evaluatedAlready - This line adds the current node position to the set of evaluated nodes, which takes O(1) time.

neighbors = getNeighborPositions(currentPathToTarget.pos) - This line fetches the neighboring positions of the current node, which takes O(E) time.

tiles = getTiles() - This line fetches the tiles from the environment

for each neighbor in neighbors: - This for loop iterates over the neighbors of the current node, which takes O(E) time.

if currentPos is in evaluatedAlready: - This if statement checks if the current neighbor has already been evaluated, which takes O(1) time.

if neighborNode is null and status is not IMPASSABLE: - This if statement checks if the neighbor node does not exist and is passable, which takes O(1) time.

else if cost < neighborNode.g and status is not IMPASSABLE: - This else if statement checks if the new cost is lower than the previous cost and is passable, which takes O(1) time.

```
66            // create a new node
67            if neighborNode is null and status is not IMPASSABLE: -> O(1)
68                neighborNode = Node(currentPos,cost,currentPathToTarget)
69                add neighborNode to mapAsNodes
70
71            // If the new cost is lower than the previous cost and is
72            // impassable, update the neighbor node
73            else if cost < neighborNode.g and status is not IMPASSABLE:
              ↑O(1)

74                neighborNode.g = cost
75                neighborNode.parent = currentPathToTarget
76
77            // Add the neighbor node to the priority queue if it is not 78
// already there and is impassable
79            if neighborNode is not in yetToBeEvaluated and status is not
              IMPASSABLE:
              ↑O(log V)

80                add neighborNode to yetToBeEvaluated
81
82    // Return a "do nothing" action, as the shortest path is stored in
83    // the currentPathToTarget
84    return Action.DO_NOTHING -> O(1)
```

if neighborNode is not in yetToBeEvaluated and status is not IMPASSABLE: - This if statement checks if the neighbor node is not already in the priority queue and is passable, which takes O(log V) time.

return Action.DO_NOTHING - This line returns a "do nothing" action, which takes O(1) time.

Therefore, the total time complexity of the code is O(E + V log V)

Results

Hardware:

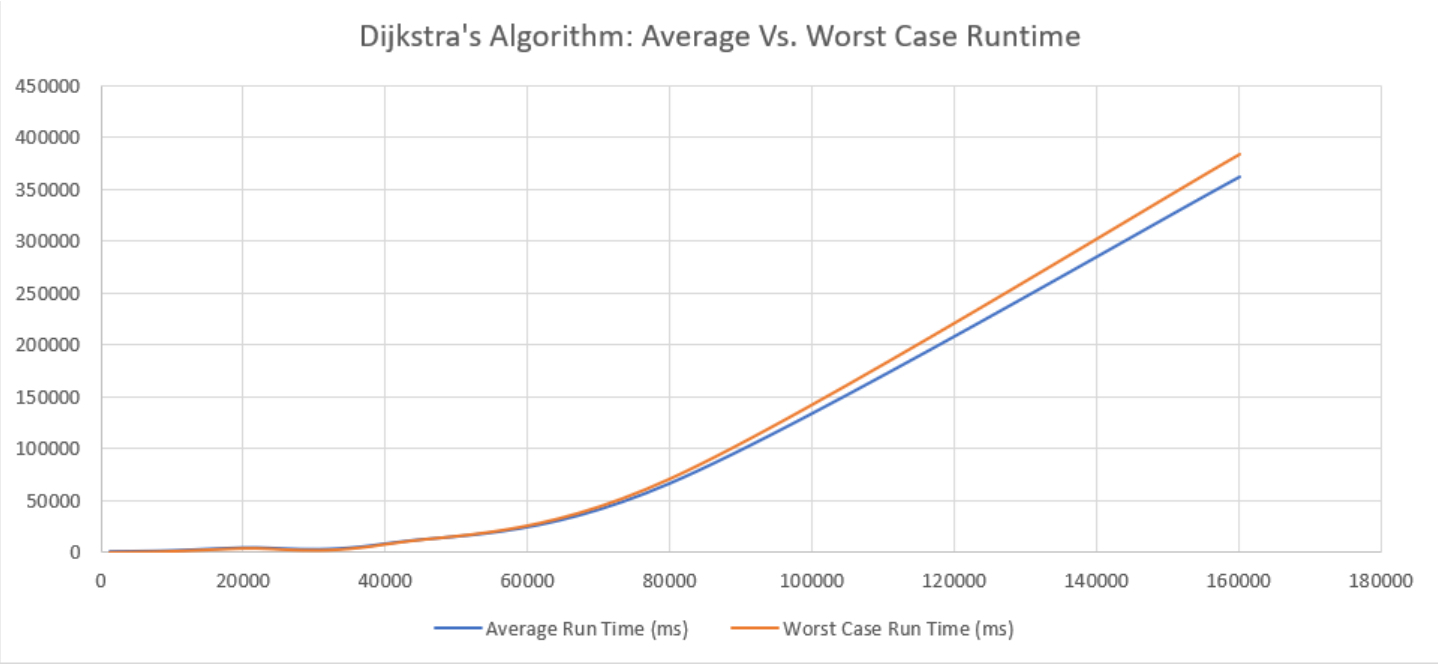
We conducted the experiment using the following hardware:

- Operating system version: macOS Ventura Version 13.0.1
- Amount of RAM: Micron 16 GB LPDDR4
- Processor Type & Speed: Apple M1: 4 cores @ 3.2 GHz, 4 cores @ 2.064 GHz

Table of Actual Runtimes for actionPerformed():

Input Map Size in nodes:	Trial 1 (ms)	Trial 2 (ms)	Trial 3 (ms)	Average Runtime (ms)
1250 Nodes	203.0 ms	199.0 ms	193.0 ms	198.33 ms
2500 Nodes	303.0 ms	303.0 ms	298.0 ms	301.33 ms
5000 Nodes	526.0 ms	497.0 ms	530.0 ms	517.67 ms
10000 Nodes	1155.0 ms	1181.0 ms	1192.0 ms	1176.0 ms
20000 Nodes	3896.0 ms	3950.0 ms	3903.0 ms	3916.33 ms
40000 Nodes	7989.0 ms	7829.0 ms	7865.0 ms	7864.0 ms
80000 Nodes	63895.0 ms	63687.0 ms	71216.0 ms	66266.0 ms
160000 Nodes	383940.0 ms	371364.0 ms	331392.0 ms	362232.0 ms

Chart of Average Vs. Worst Case Runtimes for Dijkstra's Algorithm:



Discussion

Reflection on Theoretical Analysis

Dijkstra's algorithm is an algorithm used for finding shortest paths between two nodes in a weighted graph. The algorithm works by storing the nodes that are yet to be evaluated into a priority queue which are sorted by their cost (g score). This priority queue is then used to evaluate each node which in turn updates the cost of its neighbors and adds them to the queue. The algorithm continues this process until the shortest path between two nodes is found or the queue is empty. In this case the algorithm continues until the target node is found or the queue is empty. The path between the start and target node is created by backtracking from the target node to the start node using parent pointers.

The time complexity of this implementation of Dijkstra's algorithm is O(E + |V| log |V|), where E is the number of edges in the graph (the number of neighbors each node has) and V is the number of nodes in the graph (the number of positions in the map). This runtime is derived from runtime from the combination of insert and get operation in a priority queue O(log |V|) which occurs during each iteration of a node in the map, and iterating through each neigbhor a node has O(E). The total time complexity is therefore the sum of these costs, which is O(E + |V| log |V|).

In terms of worst case runtime, the worst case scenario for Dijkstra's algorithm occurs when the shortes path has not been found. This means that all of the nodes must be evaluated and iterated through. This situation can happen when a path is not possible to the target node, or if the target node was the very last node to be evaluated. In this case, the algorithm will have to evaluate all V nodes, resulting in a worst case runtime of O(E + |V| log |V|).

In terms of average case runtime, an accurate average case performance can’t be derived because each input map will be different. The runtimes will depend on how the input map is structured as well as the weights of the costs (the g scores) of the nodes. Since around half of the nodes are unreachable in the input map we can say that on average the algorithm will evaluate half of the nodes. This will result in an average case runtime of O(E + (|V|/2) log |V|).

Of course there are other factors that will affect the runtime such as the number of edges (neighbors) that each node has. As each node is evaluated, the algorithm will also iterate over all the neigbhors it has which can increase the runtime if a node has many neighbors.

The space complexity of this implementation of Dijkstra's algorithm is O(|V|), as it stores all the nodes in the graph (the positions in the map) in a map data structure. The map is responsible for mapping each node’s position to the corresponding node.

Reflection on Experimental Analysis:

The experimental analysis we conducted on Dijkstra's algorithm provides strong evidence that the actual runtime aligns with the anticipated runtime. Our above chart shows the relationship between the size of the input and the corresponding increase in runtime. As the input size grows, the graph's curve appears to takes on an exponential pattern, which shows a visual representation of this correlation. The experimental data collected also provides a clearly shows the runtime behavior of Dijkstra's algorithm in relation to the size of the input map. The data shows that as the number of nodes in the map increases, the runtime of the algorithm increases as well, with a clear exponential trend.

The average runtime of the algorithm, for instance, was 198.33 ms when the input map size was 1250 nodes and 38394.0 ms when the input map size was 160000 nodes. This pattern demonstrates the sensitivity of the algorithm's runtime to the input map's size and emphasizes the significance of taking the input's size into account when choosing algorithms for particular problem domains.

The data also demonstrates the consistency of the algorithm's performance because the same outcomes were produced in numerous trials. For instance, the average runtime when the input map size was 5000 nodes was 517.67 ms, which is very similar to the outcomes of the other trials. Its consistency confirms the algorithm's dependability and robustness in tackling issues in many circumstances.

Interestingly, Dijkstra's algorithm is known for exhibiting a similar curve as the input size increases, reaffirming the accuracy of our experimental approach and the consistency between the actual and expected runtimes. This observation underscores the inherent properties of the algorithm under examination, which inherently responds to the challenges posed by increasing input sizes.

Lastly, it can be said that the experimental data provides valuable insights into the performance of Dijkstra's algorithm and its relationship with the size of the input map. The data confirms the expected relationship between input size and runtime and highlights the consistency and reliability of the algorithm's performance. These findings serve as a testament to the algorithm's efficiency and versatility in solving complex problems in various domains.

Reflection On Impact of Map Structure on Dijkstra’s Algorithm:

Dijkstra's algorithm might be affected by the map's structure, thus it's crucial to take into account a number of elements when creating the map. The algorithm's performance can be significantly impacted by the quantity of nodes, edges, and the distribution of costs (the g scores) m.

A map with many nodes and edges, for instance, may require the algorithm to evaluate more nodes and update more costs, which can increase runtime and decrease efficiency. The method will be able to evaluate the nodes and update the costs more quickly on a map with fewer nodes and edges, which will result in a shorter runtime and greater efficiency.

The algorithm's performance can also be affected by how the costs of the map's nodes are distributed. The algorithm will be able to analyze the nodes in a balanced manner and perform a more accurate and efficient search for the shortest path if the costs are evenly distributed. The algorithm may examine some nodes more than once if the costs are unequally distributed, which would lengthen runtime and reduce efficiency.

The findings of the algorithm can also be significantly impacted by the existence of impassable tiles in the map. The method may have to take longer detours to get to the target node if there are many barriers on the map, increasing runtime and decreasing efficiency. On the other hand, the algorithm will be able to reach the target node more rapidly and effectively if the map contains few barriers.