CS5800 – Final Project Report

Self-Driving Car Navigation System

Rohit SunilKumar Hooda, Sindhu Krovvidi, Tanvi Prashant Magdum

Video Link: https://www.youtube.com/watch?v=ZrCyUQaBUxo

Context

Problem Statement

Self-driving car technology has the potential to revolutionize the way we travel and commute. As computer science students we are passionate about using technology in everyday life to improve the lives of people, we think self-driving car technology will be a potential game-changer for travel and commuting. For self-driving cars, the navigation systems that these vehicles rely on are very crucial, as they ensure both safety and efficiency. These systems must quickly and accurately determine the best route to take, considering factors such as traffic, obstacles, and pedestrian activity. The presence of multiple factors involved in decision-making can make the task overly complex. We aim to develop a technique that can effectively solve this problem.

One of the most used pathfinding algorithms in self-driving car navigation is the A* algorithm, which is an efficient algorithm for finding the shortest and most efficient route from one point to another. To get a better understanding of the advantages and limitations of the A* algorithm in self-driving car navigation, we need to compare it with other pathfinding algorithms that are also commonly used in this context, such as Dijkstra's algorithm and the Floyd-Warshall algorithm. We can examine various aspects of these algorithms, such as their accuracy, efficiency, flexibility, scalability, and overall safety, to determine why A* is the best-suited algorithm in this case.

Project Motivation

Tanvi: Self-driving cars have been a topic of significant interest and discussion in recent years, as the technology behind them continues to evolve and improve. For me personally, this topic is particularly relevant because I am a computer science student with a strong interest in artificial intelligence and machine learning. Self-driving cars represent a challenging problem in these fields, requiring the development of sophisticated algorithms and models that can accurately interpret and navigate complex environments. It is remarkable to observe how these vehicles can negotiate complex road networks and make real-time decisions by utilizing sophisticated algorithms. By exploring the use of advanced algorithms like A* in self-driving car navigation, we can contribute to the development of this exciting and rapidly evolving field, while also helping to make transportation safer, more efficient, and more accessible for people around the world.

Sindhu: Self-driving navigation cars play a vital role with many benefits. Amongst them safety and convenience are at the top of the list. Human errors are the primary reasons for accidents and with the help of self-driving navigation cars there can be reduced in number of accidents as the cars react more quickly and take decisions based on sensors and data. Also, it is more preferrable in long journeys as one can rest, and it also helps in finding the path on the most confused roads. They can also help reduce traffic congestion by optimizing routes and reducing the number of cars on the road. As a computer science student, I am personally interested in self-driving cars as I am fascinated by their advanced technology and real-world impact.

Rohit: The development of self-driving cars is a complex and fascinating field, with the potential to revolutionize the way we think about transportation. Not only can these vehicles improve safety by reducing the number of accidents caused by human error, but they can also enhance convenience by allowing passengers to relax and enjoy the ride. Additionally, self-driving cars can optimize traffic flow and reduce congestion on roads, making transportation more efficient and eco-friendlier. As the technology behind self-driving cars continues to evolve, I am excited to think about the many ways in which these vehicles can improve our daily lives and make travel more accessible to people around the world.

Analysis

Comparison between different pathfinding algorithms and why A* is the most suitable algorithm for our use-case

Pathfinding algorithms are used to find the shortest and most efficient route between two points on a graph. In the context of self-driving car navigation, pathfinding algorithms are crucial for ensuring the safety and efficiency of the vehicle. There are many pathfinding algorithms that can be used for this purpose, but some of the most used algorithms are Dijkstra's algorithm, Floyd-Warshall algorithm, and A* algorithm.

Dijkstra's algorithm is well-known for finding the shortest path between two points in a graph. The algorithm works by starting at the initial vertex and visiting its neighbors. The algorithm assigns a tentative distance to each vertex and then selects the vertex with the smallest tentative distance as the next vertex to visit. The algorithm then updates the tentative distances of the neighboring vertices and repeats this process until the destination vertex is reached. Dijkstra's algorithm is guaranteed to find the shortest path if all edge weights are non-negative. However, it can be computationally expensive on large graphs with negative edge weights.

The Floyd-Warshall algorithm is another pathfinding algorithm used in a self-driving car navigation. This algorithm is used to find the shortest path between all pairs of vertices in a graph. The algorithm creates a matrix of distances between all pairs of vertices and then iteratively updates this matrix until the shortest path between all pairs of vertices is found. The Floyd-Warshall algorithm is particularly useful for dense graphs but can be computationally expensive on large graphs. It is also not suitable for real-time applications.

The A* algorithm is another widely used pathfinding algorithm for self-driving car navigation. A* algorithm is an informed search algorithm that uses a heuristic function to guide the search towards the goal. The algorithm works by expanding nodes with the lowest estimated total cost to reach the goal. The total cost is the sum of the actual cost to reach the node and the estimated cost to reach the goal from that node. The estimated cost is calculated using a heuristic function. A* algorithm can quickly find the shortest path between two points, especially on large graphs. It is optimized for real-time applications and can be adapted to diverse types of graphs.

When evaluating the effectiveness of pathfinding algorithms for self-driving car navigation, several criteria need to be considered:

1. Accuracy: The algorithm should be able to find the shortest and most efficient path between two points.
2. Efficiency: The algorithm should be able to quickly find the path, especially on large graphs.
3. Flexibility: The algorithm should be adaptable to distinct types of roads and traffic conditions.
4. Scalability: The algorithm should be able to handle large and complex graphs.
5. Safety: The algorithm should be able to avoid obstacles and ensure the safety of passengers and other road users. (Future scope)

Usage of A* algorithm in self-driving cars

Step 1: Defining the problem

First, we clearly define the problem we are trying to solve. For self-driving cars, the problem is to find the shortest and most efficient route from the current location to the desired destination. For now, we are considering traffic conditions. In further scope, we might need to consider factors like road conditions, and potential obstacles to make the software more realistic.

Step 2: Building a graph representation of the road network

We create a graph representation of the road network, with each intersection or node representing a point in the network and each road segment or edge representing a connection between nodes. We can also assign weights to the edges to represent factors such as distance or travel time.

Step 3: Defining the heuristic function

To use the A* algorithm, we define a heuristic function that estimates the remaining cost from each node to the goal node. This function can be based on factors such as distance, travel time, or the presence of obstacles. We are considering the traffic factor for now. While calculating cost, we consider this overhead at each node. Straight-line distance aka. Euclidean distance can be used as a heuristic function for this project.

Step 4: Initializing the algorithm

We start the algorithm by initializing a priority queue and adding the starting node to it. We also create two maps to store the cost and parent of each node, with the cost of the starting node set to 0 and the cost of all other nodes set to infinity.

Step 5: Main loop

In each iteration of the main loop, we take the node with the lowest cost from the priority queue and call it the current node. If the current node is the goal node, we have found the optimal path and can terminate the algorithm. Otherwise, we consider each of the current node's neighbors.

Step 6: Calculating the cost

For each neighbor, we calculate the cost of the path from the starting node to the current node plus the cost of the edge from the current node to the neighbor. We then update the neighbor's cost and parent if the calculated cost is lower than the current cost. We also calculate the neighbor's heuristic value using the heuristic function.

Step 7: Adding neighbors to the queue

Once we have updated the cost and parent of all neighbors, we add each neighbor to the priority queue, considering its combined cost and heuristic value.

Step 8: Repeating loop

We continue the main loop until we have processed all nodes or have found the goal node.

Step 9: Retrieving the optimal path

Once we have found the optimal path, we can retrieve it by following the parent map from the goal node back to the starting node.

Step 10: Navigating the self-driving car

Finally, we use the optimal path to navigate the self-driving car to the destination. We can continually update the path as the car moves and conditions change, using the A* algorithm to calculate the new optimal path.

Time complexity and Space complexity

The time complexity of the A* algorithm depends on the size of the problem space, which is determined by the number of nodes and edges in the graph. In the worst-case scenario, A* will need to visit all nodes in the graph before finding the goal node, which gives a time complexity of $O(b^d)$, where b is the branching factor of the graph (the average number of edges per node), and d is the depth of the goal node.

The performance of A* can be improved significantly by using heuristics that guide the search toward the goal node more efficiently. The time complexity can be reduced to O(n log n) by using a priority queue to keep track of the nodes to be expanded, where n is the number of nodes in the graph. This is because each node is visited at most once, and the priority queue operations take logarithmic time.

The time complexity of A* also depends on the quality of the heuristic function used. The Euclidean distance heuristic is admissible and can be computed quickly, thus making the algorithm efficient. If we consider obstacles or pedestrian conditions, we might need to use different heuristic approaches like the Manhattan distance or the Diagonal distance. A heuristic must be admissible i.e., should not overestimate the cost to reach the goal.

The space complexity of the A* algorithm for self-driving cars using the Euclidean heuristic is $O(b^d)$, where b is the maximum branching factor of the search tree, and d is the depth of the goal node from the start node. In the worst case, the A* algorithm may need to keep track of all the nodes in the search tree until it reaches the goal node. This can result in a large memory footprint, especially for complex maps or long paths. The space complexity can be reduced by implementing various optimization techniques, such as using a priority queue to store only the most promising nodes or using an iterative deepening approach to limit the depth of the search.

Conclusion

The conclusion of this project is that A* algorithm is a powerful tool for pathfinding in self-driving cars. It provides an optimal path by considering the cost of the path and the estimated cost to reach the goal. This algorithm is an improvement over other popular pathfinding algorithms like Dijkstra's and Floyd-Warshall, as it takes into account the heuristic value. The use of heuristics in A* helps to guide the search process and reduce the computational complexity of the algorithm. The project demonstrates the significance of careful selection and design of heuristic functions. The effectiveness of A* is highly dependent on the accuracy and reliability of the heuristic function used. Choosing an appropriate heuristic is critical to ensure the efficiency and safety of self-driving cars on the road. Inaccurate or poorly designed heuristics can also lead to suboptimal routes and decreased efficiency in the algorithm.

This project highlights the importance of developing effective navigation systems for self-driving cars, and the A* algorithm provides a solid foundation for building such systems.

Takeaways

Tanvi: There are a few key takeaways from this project for me. First, I learned about the A* algorithm, a popular and efficient pathfinding algorithm that finds the shortest path between two points in a graph. We saw how this algorithm can be applied to real-world problems, such as self-driving cars, and how it can be optimized for efficiency and scalability. Second, I gained experience in implementing the A* algorithm in Java, which involved writing code to represent graphs, computing distances, and heuristic values, and traversing the graph using the A* algorithm. This experience was valuable in understanding the intricacies of the algorithm and how it can be customized for specific use cases. I also learned about the importance of heuristic functions in the A* algorithm and how they can impact the efficiency and accuracy of the algorithm as we saw that several types of heuristic functions can be used for different scenarios and that choosing the right heuristic function is crucial for achieving optimal results.

Sindhu: This project enabled me to gain a deeper understanding of shortest-path algorithms, particularly the A* algorithm. I was able to see firsthand how the heuristic function influences the algorithm's decision-making process and how it can be used to efficiently search a graph for the optimal path between two points. It was exciting to see how this algorithm can be applied to solve real-world problems such as finding the

most efficient route for a self-driving car to reach its destination while considering traffic conditions. Shortest path algorithms are essential tools in the field of transportation, and being able to apply them to real-world problems such as optimizing travel time and avoiding traffic congestion was a valuable experience. Overall, this project allowed me to develop my problem-solving skills and gain a greater appreciation for the power of algorithmic thinking. I am confident that the concepts and skills I learned in this project will be applicable to future courses and projects and in my professional career.

Rohit: My favorite aspect of this class has been the graph traversal algorithms, and this project provided me with an opportunity to apply what I learned to a new problem. It allowed me to think creatively about how algorithms can be utilized and reflect on potential improvements and future applications. Furthermore, the project helped me realize that the benefits and drawbacks of algorithms are not fixed, but rather dependent on the problem being addressed. For instance, in our project, while Dijkstra's algorithm can be used, A* is more versatile due to its ability to use heuristics to find a path faster. Working alongside my exceptional team members, we evaluated different approaches to the problems at hand and constantly compared their advantages and disadvantages. This experience has enabled me to better integrate into collaborative environments and approach problem-solving with greater objectivity.

Appendix

Code (in Java)

Node.java

```
public class Node {

  private int x;
  private int y;
  private double cost;

  public Node(int x, int y, double cost) {
    this.x = x;
    this.y = y;
    this.cost = cost;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public double getCost() {
    return cost;
  }

  public double getHeuristic(Node goal) {
    return Math.sqrt(Math.pow((this.x - goal.getX()), 2) + Math.pow((this.y - goal.getY()),
2));
  }

  public double getEstimatedCost(Node goal) {
    return this.getCost() + this.getHeuristic(goal);
  }

  @Override
  public String toString() {
    return "(" + this.x + ", " + this.y + ")";
```

```
  }

}
```

Graph.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class Graph {

  private Set<Node> nodes = new HashSet<>();
  private Map<Node, List<Node>> edges = new HashMap<>();
  private Map<String, Integer> traffic = new HashMap<>();

  public void addNodes(Node... nodes) {
    for (Node node : nodes) {
      this.nodes.add(node);
    }
  }

  public void addEdge(Node source, Node destination, double weight) {
    if (!this.edges.containsKey(source)) {
      this.edges.put(source, new ArrayList<>());
    }
    this.edges.get(source).add(destination);
  }

  public void setTraffic(Node source, Node destination, int level) {
    String key = source.toString() + "->" + destination.toString();
    this.traffic.put(key, level);
  }

  public int getTraffic(Node source, Node destination) {
    String key = source.toString() + "->" + destination.toString();
    if (this.traffic.containsKey(key)) {
      return this.traffic.get(key);
    } else {
```

```
    return 0;
   }
  }

  public List<Node> getNeighbors(Node node) {
   if (!this.edges.containsKey(node)) {
    return Collections.emptyList();
   }
   return this.edges.get(node);
  }

  public double getWeight(Node source, Node destination) {
   return source.getCost() + this.getTraffic(source, destination);
  }

}
```

AStarAlgorithm.java

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

public class AStarAlgorithm {

  private Graph graph;

  public AStarAlgorithm(Graph graph) {
   this.graph = graph;
  }

  public List<Node> shortestPath(Node startNode, Node endNode) {
   Map<Node, Double> distances = new HashMap<>();
   Map<Node, Node> previousNodes = new HashMap<>();
   PriorityQueue<Node> queue = new PriorityQueue<>(
     Comparator.comparingDouble(node -> distances.getOrDefault(node,
Double.POSITIVE_INFINITY)));

   distances.put(startNode, 0.0);
   queue.add(startNode);
```

```
    while (!queue.isEmpty()) {
      Node currentNode = queue.poll();

      if (currentNode.equals(endNode)) {
        return getPath(previousNodes, endNode);
      }

      for (Node neighbor : this.graph.getNeighbors(currentNode)) {
        double tentativeDistance =
            distances.get(currentNode) + this.graph.getWeight(currentNode, neighbor);

        if (tentativeDistance < distances.getOrDefault(neighbor,
Double.POSITIVE_INFINITY)) {
          distances.put(neighbor, tentativeDistance);
          previousNodes.put(neighbor, currentNode);
          queue.add(neighbor);
        }
      }
    }
    return null;
  }

  private List<Node> getPath(Map<Node, Node> previousNodes, Node endNode) {
    List<Node> path = new ArrayList<>();
    path.add(endNode);
    Node currentNode = endNode;

    while (previousNodes.containsKey(currentNode)) {
      currentNode = previousNodes.get(currentNode);
      path.add(0, currentNode);
    }

    return path;
  }
}
```

Main.java

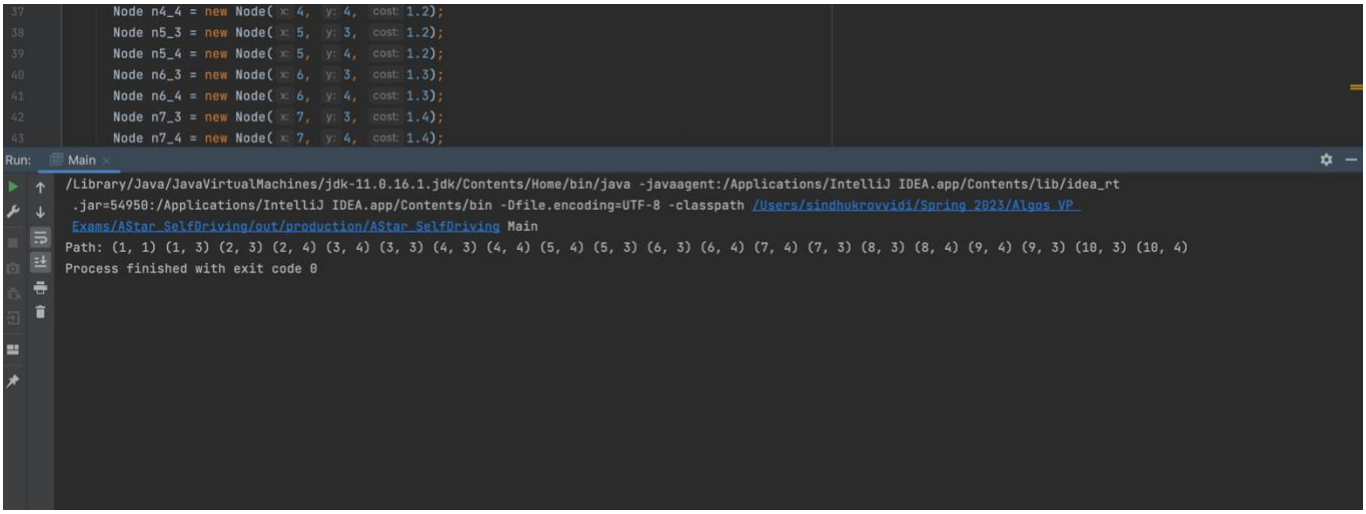
```
import java.util.List;

public class Main {
```

```
public static final Node START = new Node(0, 0, 1.0);
public static final Node GOAL = new Node(10, 10, 1.0);

public static void main(String[] args) {
  // Create nodes for the graph
  Node n1_1 = new Node(1, 1, 1.0);
  Node n1_2 = new Node(1, 2, 1.0);
  Node n2_1 = new Node(2, 1, 1.0);
  Node n2_2 = new Node(2, 2, 1.0);
  Node n3_1 = new Node(3, 1, 1.0);
  Node n3_2 = new Node(3, 2, 1.0);
  Node n4_1 = new Node(4, 1, 1.0);
  Node n4_2 = new Node(4, 2, 1.0);
  Node n5_1 = new Node(5, 1, 1.0);
  Node n5_2 = new Node(5, 2, 1.0);
  Node n6_1 = new Node(6, 1, 1.0);
  Node n6_2 = new Node(6, 2, 1.0);
  Node n7_1 = new Node(7, 1, 1.0);
  Node n7_2 = new Node(7, 2, 1.0);
  Node n8_1 = new Node(8, 1, 1.0);
  Node n8_2 = new Node(8, 2, 1.0);
  Node n9_1 = new Node(9, 1, 1.0);
  Node n9_2 = new Node(9, 2, 1.0);
  Node n10_1 = new Node(10, 1, 1.0);
  Node n10_2 = new Node(10, 2, 1.0);
  Node n1_3 = new Node(1, 3, 1.1);
  Node n1_4 = new Node(1, 4, 1.1);
  Node n2_3 = new Node(2, 3, 1.1);
  Node n2_4 = new Node(2, 4, 1.1);
  Node n3_3 = new Node(3, 3, 1.1);
  Node n3_4 = new Node(3, 4, 1.1);
  Node n4_3 = new Node(4, 3, 1.2);
  Node n4_4 = new Node(4, 4, 1.2);
  Node n5_3 = new Node(5, 3, 1.2);
  Node n5_4 = new Node(5, 4, 1.2);
  Node n6_3 = new Node(6, 3, 1.3);
  Node n6_4 = new Node(6, 4, 1.3);
  Node n7_3 = new Node(7, 3, 1.4);
  Node n7_4 = new Node(7, 4, 1.4);
  Node n8_3 = new Node(8, 3, 1.5);
  Node n8_4 = new Node(8, 4, 1.5);
  Node n9_3 = new Node(9, 3, 1.5);
  Node n9_4 = new Node(9, 4, 1.5);
```

```
        Node n10_3 = new Node(10, 3, 1.5);
        Node n10_4 = new Node(10, 4, 1.5);

        // Add the nodes to the graph
        Graph graph = new Graph();
        graph.addNodes(n1_1, n1_2, n2_1, n2_2, n3_1, n3_2, n4_1, n4_2, n5_1, n5_2, n6_1,
n6_2, n7_1,
            n7_2, n8_1, n8_2, n9_1, n9_2, n10_1, n10_2, n1_3, n1_4, n2_3, n2_4, n3_3, n3_4,
n4_3, n4_4,
            n5_3, n5_4, n6_3, n6_4, n7_3, n7_4, n8_3, n8_4, n9_3, n9_4, n10_3, n10_4);

        // Add the edges to the graph
        graph.addEdge(n1_1, n1_2, 1.0);
        graph.addEdge(n1_2, n2_2, 1.0);
        graph.addEdge(n2_2, n2_1, 1.0);
        graph.addEdge(n2_1, n3_1, 1.0);
        graph.addEdge(n3_1, n3_2, 1.0);
        graph.addEdge(n3_2, n4_2, 1.0);
        graph.addEdge(n4_2, n4_1, 1.0);
        graph.addEdge(n4_1, n5_1, 1.0);
        graph.addEdge(n5_1, n5_2, 1.0);
        graph.addEdge(n5_2, n6_2, 1.0);
        graph.addEdge(n6_2, n6_1, 1.0);
        graph.addEdge(n6_1, n7_1, 1.0);
        graph.addEdge(n7_1, n7_2, 1.0);
        graph.addEdge(n7_2, n8_2, 1.0);
        graph.addEdge(n8_2, n8_1, 1.0);
        graph.addEdge(n8_1, n9_1, 1.0);
        graph.addEdge(n9_1, n9_2, 1.0);
        graph.addEdge(n9_2, n10_2, 1.0);
        graph.addEdge(n1_1, n1_3, 1.1);
        graph.addEdge(n1_3, n2_3, 1.1);
        graph.addEdge(n2_3, n2_4, 1.1);
        graph.addEdge(n2_4, n3_4, 1.1);
        graph.addEdge(n3_4, n3_3, 1.1);
        graph.addEdge(n3_3, n4_3, 1.2);
        graph.addEdge(n4_3, n4_4, 1.2);
        graph.addEdge(n4_4, n5_4, 1.2);
        graph.addEdge(n5_4, n5_3, 1.2);
        graph.addEdge(n5_3, n6_3, 1.3);
        graph.addEdge(n6_3, n6_4, 1.3);
        graph.addEdge(n6_4, n7_4, 1.3);
        graph.addEdge(n7_4, n7_3, 1.3);
```

```
        graph.addEdge(n7_3, n8_3, 1.4);
        graph.addEdge(n8_3, n8_4, 1.4);
        graph.addEdge(n8_4, n9_4, 1.4);
        graph.addEdge(n9_4, n9_3, 1.4);
        graph.addEdge(n9_3, n10_3, 1.5);
        graph.addEdge(n10_3, n10_4, 1.5);


        // Set the traffic conditions
        graph.setTraffic(n1_1, n1_2, 2);
        graph.setTraffic(n1_3, n2_3, 1);
        graph.setTraffic(n2_2, n2_1, 3);
        graph.setTraffic(n3_1, n3_2, 1);
        graph.setTraffic(n4_2, n4_1, 2);
        graph.setTraffic(n5_1, n5_2, 1);
        graph.setTraffic(n6_2, n6_1, 3);
        graph.setTraffic(n7_1, n7_2, 1);
        graph.setTraffic(n8_2, n8_1, 2);
        graph.setTraffic(n9_1, n9_2, 1);
        graph.setTraffic(n10_2, n9_2, 2);

        // Set the start and end nodes
        Node startNode = n1_1;
        Node endNode = n10_4;

        // Run the A* algorithm
        AStarAlgorithm algorithm = new AStarAlgorithm(graph);
        List<Node> path = algorithm.shortestPath(startNode, endNode);

        // Print the path
        if (path == null) {
          System.out.println("No path found.");
        } else {
          System.out.print("Path: ");
          for (Node node : path) {
            System.out.print(node + " ");
          }
        }
      }
}
```

Output:


Node n4_4 = new Node(x: 4, y: 4, cost: 1.2);
Node n5_3 = new Node(x: 5, y: 3, cost: 1.2);
Node n5_4 = new Node(x: 5, y: 4, cost: 1.2);
Node n6_3 = new Node(x: 6, y: 3, cost: 1.3);
Node n6_4 = new Node(x: 6, y: 4, cost: 1.3);
Node n7_3 = new Node(x: 7, y: 3, cost: 1.4);
Node n7_4 = new Node(x: 7, y: 4, cost: 1.4);
Run:
Main
/Library/Java/JavaVirtualMachines/jdk-11.0.16.1.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt
.jar=54950:/Applications/IntelliJ IDEA.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /Users/sindhukrovvidi/Spring 2023/Algos VP
Exams/AStar_SelfDriving/out/production/AStar_SelfDriving Main
Path: (1, 1) (1, 3) (2, 3) (2, 4) (3, 4) (3, 3) (4, 3) (4, 4) (5, 4) (5, 3) (6, 3) (6, 4) (7, 4) (7, 3) (8, 3) (8, 4) (9, 4) (9, 3) (10, 3) (10, 4)
Process finished with exit code 0