

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“Jnana Sangama”, Belagavi – 590 018



Advanced Java (21CSE644)

Assignment

“A* Algorithm using Collections”

Submitted By

Name: Ananya G

USN: 1GA21CS020

Name: Bindu Madhavi V

USN: 1GA21CS040

Under the Guidance of

Mr. Shyam Sundar Bhushan

Assistant Professor

Dept. of CSE



Department of Computer Science and Engineering

GLOBAL ACADEMY OF TECHNOLOGY

Rajarajeshwarinagar, Bengaluru – 560 098

2023 - 2024



CHAPTER 1

PROBLEM DEFINITION

Imagine you're designing a navigation system for a robot in a maze. The robot's goal is to find the shortest path from its starting point to the exit. The maze is represented as a graph where each intersection is a node and the paths connecting them are edges with associated costs (e.g., distance, time). The challenge is to efficiently determine the optimal path considering both the distance traveled and the estimated distance to the goal.

The A* algorithm is a search algorithm that efficiently finds the shortest path between a starting node and a goal node in a graph. It combines the strengths of Dijkstra's algorithm and Greedy best-first search. This algorithm uses a heuristic function to estimate the cost from the current node to the goal, helping it prioritize exploration of promising paths. By evaluating a node's actual cost from the start (g-score) and the estimated cost to the goal (h-score), A* calculates a total cost (f-score) and explores nodes with the lowest f-score first. This approach effectively balances exploration and exploitation, making it suitable for various pathfinding problems.

CHAPTER 2

IMPLEMENTATION

2.1 COMPONENTS

There are 3 components in this program namely Node, AStar and AStarDemo.

1. Node class

- The Node class represents a point in a graph used by the A* algorithm. It stores its name, neighbors with associated costs and pathfinding data.
- name: Unique identifier.
- neighbors: Adjacency list of connected nodes and their costs.
- gScore: Cost from the start node.
- hScore: Estimated cost to the goal.
- previous: Parent node in the path.
- The class facilitates pathfinding by providing a structure to represent graph nodes and their attributes, essential for calculating the f-score ($\text{gScore} + \text{hScore}$) used to prioritize node exploration in A*.

2. AStar class

- The AStar class implements the A* search algorithm to find the shortest path between two nodes in a graph.
- findPath(Node start, Node goal) method:
 - i) openSet is a priority queue (using a comparator) that prioritizes nodes with a lower f-score while closedSet is a set to keep track of explored nodes.

ii) It sets gScore of the starting node to 0 and calculates its hScore using the heuristic function. It then adds the starting node to the openSet.

- Main Loop:

i) While openSet is not empty:

Get the node with the lowest f-score from openSet. If the current node is the goal, then return the reconstructed path. Add the current node to the closedSet.

ii) Loop through the neighbors of the current node:

If a neighbor is already in the closedSet, skip it.

iii) Calculate a tentative gScore for the neighbor (current's gScore + cost to reach the neighbor).

iv) If the neighbor is not in the openSet or the tentative gScore is lower than the neighbor's current gScore:

Add the neighbor to the openSet. Update the neighbor's previous node (for path reconstruction), gScore, and hScore (using the heuristic function).

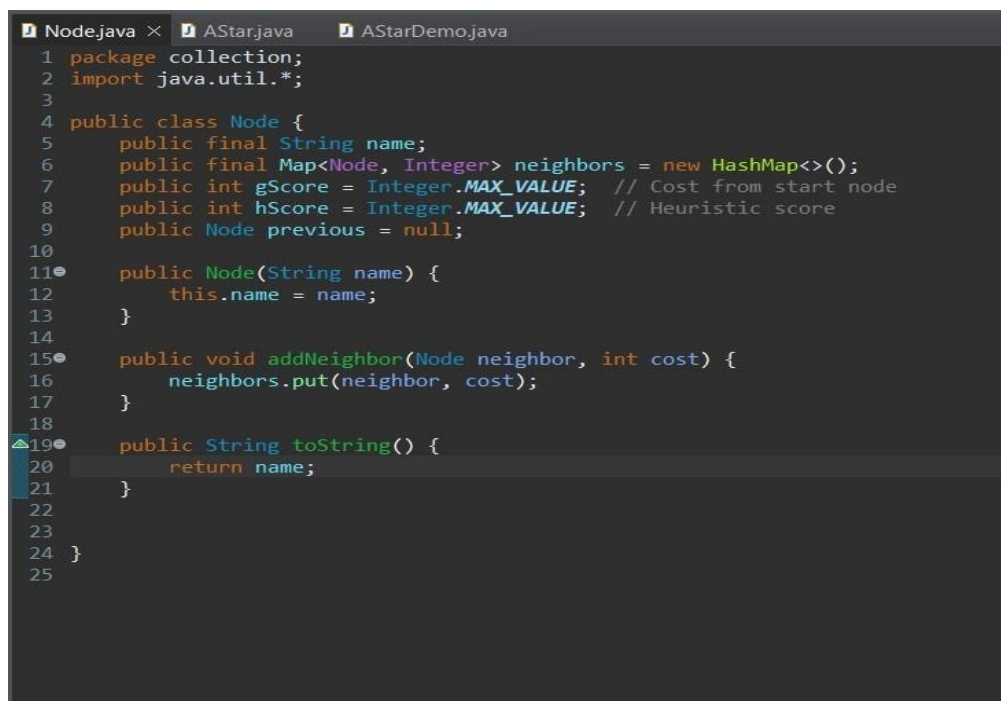
- No Path Found: If the loop finishes without finding the goal, return an empty list indicating no path was found.
- reconstructPath(Node goal) method: Traces back from the goal node, following the previous pointers in each node to create the final path as a list.
- This class essentially performs an informed search through the graph, prioritizing nodes closer to the goal (based on the heuristic) while keeping track of the actual travel cost (gScore).

3. AStarDemo class

- The AStarDemo class shows how to use the AStar class to find a path between two nodes in a graph.
- Create Nodes: Six Node objects are created with names from 'A' to 'F'. These represent the nodes in the graph.
- Create Edges: addNeighbor method is used to establish connections between nodes. For example, a.addNeighbor(b, 1) creates an edge from node 'A' to node 'B' with a cost of 1. This defines the structure of the graph.
- Find Path: The AStar.findPath(a, f) method is called to find the shortest path from node 'A' to node 'F' using the A* algorithm.
- Print Path: The found path is printed to the console, showing the sequence of nodes from the start to the goal.

2.2 PROGRAM CODE

1. Node.java



```
1 package collection;
2 import java.util.*;
3
4 public class Node {
5     public final String name;
6     public final Map<Node, Integer> neighbors = new HashMap<>();
7     public int gScore = Integer.MAX_VALUE; // Cost from start node
8     public int hScore = Integer.MAX_VALUE; // Heuristic score
9     public Node previous = null;
10
11     public Node(String name) {
12         this.name = name;
13     }
14
15     public void addNeighbor(Node neighbor, int cost) {
16         neighbors.put(neighbor, cost);
17     }
18
19     public String toString() {
20         return name;
21     }
22
23 }
24
25
```

2. AStar.java

```
AStar.java ×
1 package collection;
2 import java.util.*;
3
4 public class AStar {
5     public static List<Node> findPath(Node start, Node goal) {
6         PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(n -> n.gScore + n.hScore));
7         Set<Node> closedSet = new HashSet<>();
8
9         start.gScore = 0;
10        start.hScore = heuristic(start, goal);
11        openSet.add(start);
12
13        while (!openSet.isEmpty()) {
14            Node current = openSet.poll();
15
16            if (current.equals(goal)) {
17                return reconstructPath(goal);
18            }
19
20            closedSet.add(current);
21
22            for (Map.Entry<Node, Integer> neighborEntry : current.neighbors.entrySet()) {
23                Node neighbor = neighborEntry.getKey();
24                int cost = neighborEntry.getValue();
25
26                if (closedSet.contains(neighbor)) {
27                    continue;
28                }
29
30                int tentativeGScore = current.gScore + cost;
31
32                if (!openSet.contains(neighbor)) {
33                    openSet.add(neighbor);
34                } else if (tentativeGScore >= neighbor.gScore) {
35                    continue;
36                }
37
38                neighbor.previous = current;
39                neighbor.gScore = tentativeGScore;
40                neighbor.hScore = heuristic(neighbor, goal);
41            }
42        }
43
44        return Collections.emptyList(); // Path not found
45    }
46
47    private static int heuristic(Node a, Node b) {
48        // Simple heuristic function, you may use more complex heuristics
49        return Math.abs(a.name.hashCode() - b.name.hashCode());
50    }
51
52    private static List<Node> reconstructPath(Node goal) {
53        List<Node> path = new ArrayList<>();
54        Node current = goal;
55
56        while (current != null) {
57            path.add(current);
58            current = current.previous;
59        }
60
61        Collections.reverse(path);
62        return path;
63    }
64 }
65 }
66 }
```

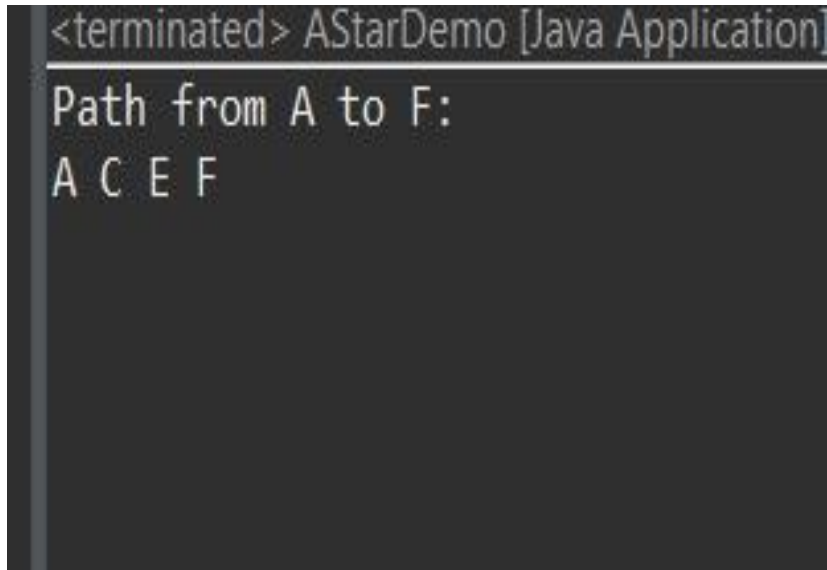
3. AStarDemo.java

```
*AStarDemo.java X
1 package collection;
2 import java.util.*;
3 public class AStarDemo {
4
5     public static void main(String[] args) {
6
7         Node a = new Node("A");
8         Node b = new Node("B");
9         Node c = new Node("C");
10        Node d = new Node("D");
11        Node e = new Node("E");
12        Node f = new Node("F");
13
14        a.addNeighbor(b, 1);
15        a.addNeighbor(c, 4);
16        b.addNeighbor(c, 2);
17        b.addNeighbor(d, 5);
18        c.addNeighbor(d, 1);
19        c.addNeighbor(e, 3);
20        d.addNeighbor(e, 1);
21        e.addNeighbor(f, 1);
22
23        List<Node> path = AStar.findPath(a, f);
24        System.out.println("Path from A to F:");
25        for (Node node : path) {
26            System.out.print(node + " ");
27        }
28
29    }
30
31 }
32
```

CHAPTER 3

SNAPSHOTS

Output:



```
<terminated> AStarDemo [Java Application]
Path from A to F:
A C E F
```

CONCLUSION

The A* algorithm stands as a powerful and versatile tool for efficiently determining optimal paths in graph-based problems. By intelligently combining the concepts of actual cost and estimated distance to the goal, A* excels in finding the shortest path while exploring a limited number of nodes. Its applications span across various domains, including robotics, game development, and transportation systems. While the effectiveness of A* is heavily influenced by the quality of the heuristic function used, its core principles make it a fundamental algorithm in the field of artificial intelligence and pathfinding.