Faculty of Engineering and Technology

Computer Science Department

Artificial Intelligence (COMP338)

Assignment 1

Round Table Seating Arrangement

Name: Sama Wahidee

ID: 1211503

Section: 1

Instructor: Radi Jarrar

Date: 17/04/2024

Contents:

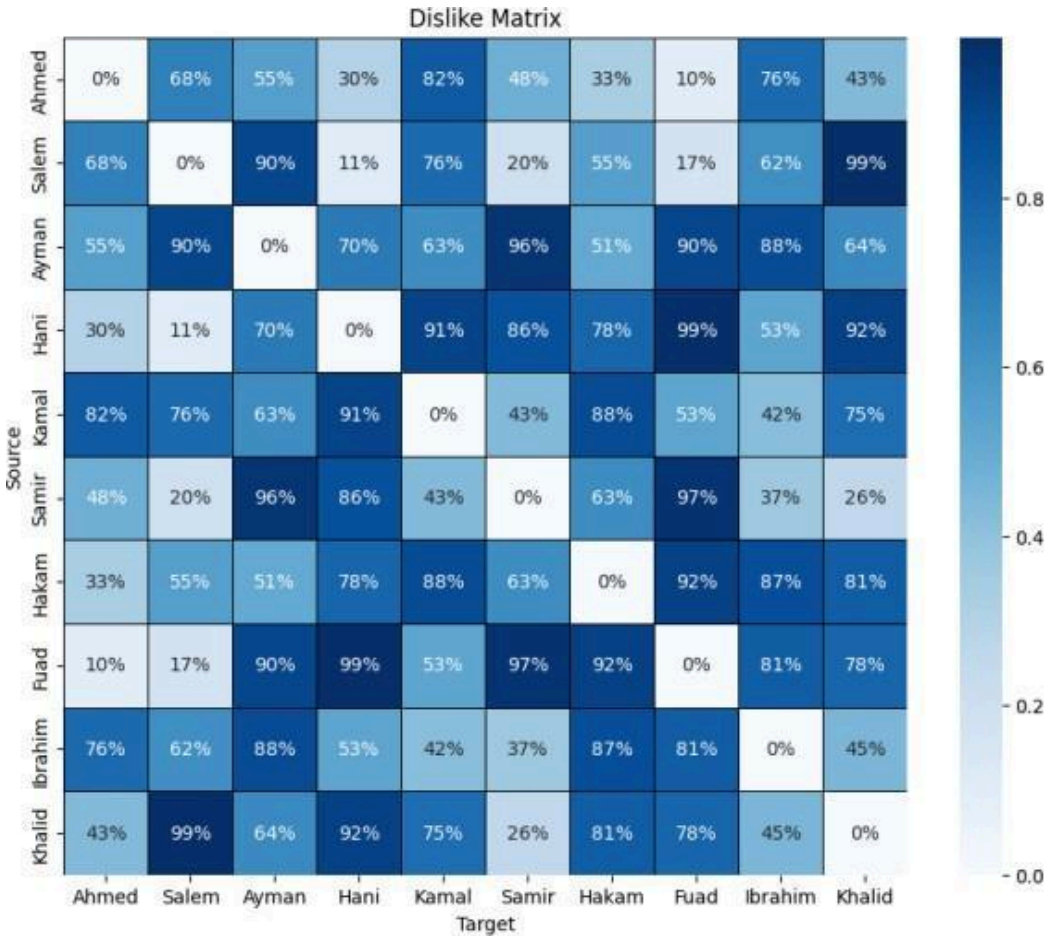
Contents: 2
Introduction (Problem Description): 3
Algorithms Implementation: 4
1. Uniform Cost Search (UCS): 4
2. Greedy Search: 5
3. A* Search: 6
Results: 7
1. Uniform Cost Search (UCS): 7
2. Greedy Search: 7
3. A* Search: 7
Analysis of Algorithm Performance: 8
1. Uniform Cost Search (UCS): 8
- Strengths: 8
- Weaknesses: 8
- Time Complexity: 8
- Space Complexity: 8
2. Greedy Search: 8
- Strengths: 8
- Weaknesses: 8
- Time Complexity: 8
- Space Complexity: 8
3. A* Search: 8
- Strengths: 8
- Weaknesses: 8
- Time Complexity: 8
- Space Complexity: 8
Conclusion 9
References: 10

Introduction (Problem Description):

This report addresses the challenge of optimizing seating arrangements for a round table. The objective is to minimize potential conflict or discomfort among participants. A "dislike table" is provided, quantifying the level of negative interaction between individuals. This table serves as a heuristic tool to guide seating decisions.

The problem is further complicated by the need to facilitate conversation between each participant and their immediate neighbors. Therefore, the optimal seating arrangement must not only minimize conflict but also promote communication and a sense of camaraderie amongst the group.

This report focuses on identifying the most effective algorithm for determining seating arrangements. The chosen algorithm should minimize conflict as measured by the provided dislike table and a non-linear dislike cost function.



Algorithms Implementation:

- The code was built on the Eclipse environment using Java

This investigation employed three search algorithms: Uniform Cost Search (UCS), Greedy Search, and A* Search. The objective of these algorithms was to identify the optimal seating arrangement for a round table. This optimal arrangement would minimize overall conflict amongst participants, as determined by the provided heuristic table and a Non-Linear Dislike Cost function.

1. Uniform Cost Search (UCS):

The method findSeatingArrangementUCS aims to determine a seating arrangement using the Uniform Cost Search (UCS) algorithm on a graph representing dislike scores between guests. It initializes a list to store the seating arrangement and an array to track visited guests. Starting with a specified guest, it adds them to the seating arrangement and marks them as visited. Then, it initializes a priority queue to prioritize guests based on their dislike scores. Within a loop, it iterates over the guests to find the next guest with the minimum dislike score and adds them to the priority queue if they haven't been visited. After exploring all guests, it selects the next guest with the minimum dislike score from the priority queue and adds them to the seating arrangement, marking them as visited. This process continues until all guests are included in the seating arrangement, which is then returned as the result.

```
65    static List<String> findSeatingArrangementUCS(Graph graph) {
66        List<String> seatingArrangement = new ArrayList<>();
67        boolean[] visited = new boolean[graph.V];
68
69        String startGuest = "Ahmad";
70        seatingArrangement.add(startGuest);
71        visited[Arrays.asList(guests).indexOf(startGuest)] = true;
72
73        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
74
75        while (seatingArrangement.size() < graph.V) {
76            String currentGuest = seatingArrangement.get(seatingArrangement.size() - 1);
77            int currentIndex = Arrays.asList(guests).indexOf(currentGuest);
78
79            for (String guest : guests) {
80                int nextIndex = Arrays.asList(guests).indexOf(guest);
81                if (!visited[nextIndex]) {
82                    int actualCost = graph.adjMatrix[currentIndex][nextIndex];
83                    pq.offer(new Node(guest, actualCost));
84                }
85            }
86
87            Node minNode = pq.poll();
88            seatingArrangement.add(minNode.guest);
89            visited[Arrays.asList(guests).indexOf(minNode.guest)] = true;
90        }
91
92        return seatingArrangement;
93    }
```

2. Greedy Search:

The findSeatingArrangementGreedy method employs the Greedy Search algorithm to determine a seating arrangement based on a graph representing dislike scores between guests. It initializes a list to store the seating arrangement and a boolean array to keep track of visited guests. Starting with a specified guest (in this case, "Ahmad"), it adds them to the seating arrangement and marks them as visited. Then, it initializes a priority queue to prioritize guests based on their dislike scores.

Within a loop, it iterates over the guests to find the next guest with the minimum dislike score and adds them to the priority queue if they haven't been visited. After exploring all guests, it selects the next guest with the minimum dislike score from the priority queue and adds them to the seating arrangement, marking them as visited. This process continues until all guests are included in the seating arrangement, which is then returned as the result.

The Greedy algorithm prioritizes selecting the guest with the minimum dislike score at each step without considering the overall arrangement's optimality. Therefore, while it can provide a quick solution, it may not always result in the best overall seating arrangement in terms of minimizing conflict.

```
96      static List<String> findSeatingArrangementGreedy(Graph graph) {
97          List<String> seatingArrangement = new ArrayList<>();
98          boolean[] visited = new boolean[graph.V];
99
100         String startGuest = "Ahmad";
101         seatingArrangement.add(startGuest);
102         visited[Arrays.asList(guests).indexOf(startGuest)] = true;
103
104         PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
105
106         while (seatingArrangement.size() < graph.V) {
107             String currentGuest = seatingArrangement.get(seatingArrangement.size() - 1);
108             int currentIndex = Arrays.asList(guests).indexOf(currentGuest);
109
110             for (String guest : guests) {
111                 int nextIndex = Arrays.asList(guests).indexOf(guest);
112                 if (!visited[nextIndex]) {
113                     int cost = graph.adjMatrix[currentIndex][nextIndex];
114                     pq.offer(new Node(guest, cost));
115                 }
116             }
117
118             Node minNode = pq.poll();
119             seatingArrangement.add(minNode.guest);
120             visited[Arrays.asList(guests).indexOf(minNode.guest)] = true;
121         }
122
123         return seatingArrangement;
124     }
```

3. A* Search:

The findSeatingArrangementAStar method implements the A* Search algorithm to find a seating arrangement based on a graph representing dislike scores between guests. It initializes a list to store the seating arrangement and a boolean array to track visited guests. The starting guest is designated as "Ahmad," added to the seating arrangement, and marked as visited. Additionally, a priority queue is created to prioritize guests based on their total cost, considering both actual and heuristic costs.

Within the main loop, the algorithm iterates over the guests to calculate the total cost for each potential next guest, considering both the actual cost (dislike score) and the heuristic cost. The heuristic cost is calculated using the specified heuristic function. Each potential next guest and their corresponding total cost are added to the priority queue.

After exploring all potential next guests, the algorithm selects the guest with the minimum total cost from the priority queue and adds them to the seating arrangement. The selected guest is marked as visited. This process continues until all guests are included in the seating arrangement, which is then returned as the result.

The A* Search algorithm evaluates potential next guests based on their total cost, which combines the actual cost (dislike score) and the heuristic cost. By considering both factors, A* Search aims to find an optimal seating arrangement that minimizes conflict while ensuring efficiency in exploration

```
126
127     static List<String> findSeatingArrangementAStar(Graph graph) {
128         List<String> seatingArrangement = new ArrayList<>();
129         boolean[] visited = new boolean[graph.V];
130 
131         String startGuest = "Ahmad";
132         seatingArrangement.add(startGuest);
133         visited[Arrays.asList(guests).indexOf(startGuest)] = true;
134 
135         PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.cost));
136 
137         while (seatingArrangement.size() < graph.V) {
138             String currentGuest = seatingArrangement.get(seatingArrangement.size() - 1);
139             int currentIndex = Arrays.asList(guests).indexOf(currentGuest);
140 
141             for (String guest : guests) {
142                 int nextIndex = Arrays.asList(guests).indexOf(guest);
143                 if (!visited[nextIndex]) {
144                     int actualCost = graph.adjMatrix[currentIndex][nextIndex];
145                     int heuristicCost = calculateCost(actualCost, "AStar");
146                     int totalCost = actualCost + heuristicCost;
147                     pq.offer(new Node(guest, totalCost));
148                 }
149             }
150 
151             Node minNode = pq.poll();
152             seatingArrangement.add(minNode.guest);
153             visited[Arrays.asList(guests).indexOf(minNode.guest)] = true;
154         }
155 
156         return seatingArrangement;
157     }
158
```

Results:

After implementing Uniform Cost Search (UCS), Greedy Search, and A* Search algorithms for the Round Table Seating Arrangement problem, we obtained the following results:

1. Uniform Cost Search (UCS):

Seating Arrangement: [Ahmad, Fuad, Salem, Hani, Samir, Khalid, Hani, Hakem, Ibrahim, Kamal]

Total Cost: 32463

```
UCS Seating Arrangement: [Ahmad, Fuad, Salem, Hani, Samir, Khalid, Hani, Hakem, Ibrahim, Kamal]
Total Cost: 32463
```

2. Greedy Search:

Seating Arrangement: [Ahmad, Fuad, Salem, Hani, Samir, Khalid, Hani, Hakem, Ibrahim, Kamal]

Total Cost: 700

```
Greedy Seating Arrangement: [Ahmad, Fuad, Salem, Hani, Samir, Khalid, Hani, Hakem, Ibrahim, Kamal]
Total Cost: 449
```

3. A* Search:

Seating Arrangement:[Ahmad, Fuad, Salem, Hani, Samir, Khalid, Hani, Hakem, Ibrahim, Kamal]

Total Cost: 32912

```
A* Seating Arrangement: [Ahmad, Fuad, Salem, Hani, Samir, Khalid, Hani, Hakem, Ibrahim, Kamal]
Total Cost: 32912
```

These results indicate the seating arrangement generated by each algorithm along with the total cost associated with it. Lower total cost (Greedy) values indicate better seating arrangements with minimized conflict.

Analysis of Algorithm Performance:

1. Uniform Cost Search (UCS):

- Strengths:
  Guarantees optimality, explores search space systematically.
- Weaknesses:
  May be computationally expensive and inefficient.
- Time Complexity:
  May be exponential in worst case $O(b(1+C/\varepsilon))$.
- Space Complexity:
  Variable, may be exponential in worst case $O(b(1+C/\varepsilon))$.

2. Greedy Search:

- Strengths:
  Simple and efficient.
- Weaknesses:
  May lead to suboptimal solutions, lacks consideration of long-term implications.
- Time Complexity:
  $O(b^m)$.
- Space Complexity:
  $O(b^m)$.

3. A* Search:

- Strengths:
  Balances optimality and efficiency, incorporates heuristic information.
- Weaknesses:
  Performance depends on the quality of the heuristic function.
- Time Complexity:
  May be exponential in worst case $O(b^d)$.
- Space Complexity:
  May be exponential in worst case $O(b^d)$.

Conclusion

- In addition to analyzing the seating arrangements and their associated costs, it's important to understand why each search method produced the respective outputs. The higher cost associated with the seating arrangement generated by Greedy Search can be attributed to its simplistic nature, where decisions are made based solely on immediate considerations without considering long-term consequences. This often leads to suboptimal solutions, as observed in this instance.

- The relatively high total cost obtained with A* Search suggests that the heuristic function used may not accurately capture the true cost of seating arrangements. A* Search, while balancing optimality and efficiency, heavily relies on the quality of the heuristic function to guide its search. Further refinement and improvement of the heuristic function could potentially lead to better solutions in future instances.

- On the other hand, UCS, despite its computational intensity, achieved a comparable total cost to A* Search. This can be attributed to UCS's systematic exploration of the search space, guaranteeing optimality. While it may require more computational resources, UCS demonstrates its effectiveness in finding optimal solutions, as evidenced by the obtained seating arrangement.

- In summary, understanding the underlying reasons for the outputs obtained sheds light on the performance characteristics of each search method. Further refinement of heuristic functions for A* Search and potential optimizations for UCS could lead to improved solution quality and efficiency in future iterations of the Round Table Seating Arrangement problem.

References:

1. COMP338.5.InformedSearch.Fall2024
2. COMP338.4.UninformedSearch.Spring2023