Discovering Round Table Seating Arrangements Using Search Algorithms

A Study in Artificial Intelligence

Prepared by: Hanadi Asfour **Date:** 21/4/2024

Problem:

In this assignment, the goal is to determine a seating arrangement for a group of individuals sitting around a circular table, with the aim of minimizing conflict based on dislike percentages between pairs of individuals.

Theory of Search Algorithms:

1. Greedy Search: This algorithm selects the locally next best option at each step using the heuristic function without considering future or past costs. It finds a solution quickly but not necessarily the optimal one.

2. Uniform Cost Search (UCS): This algorithm searches the space while considering the cumulative cost of reaching each state from the start. It ensures finding the solution but can use up more resources.

3. A* Search: This algorithm combines the advantages of Greedy and UCS together. It considers both the actual cost and an estimated heuristic cost. This way there is a balance between finding a good solution quicker and more efficient than UCS.

Solution Approach:

An array of double numbers was populated by the dislike values from the assignment matrix. The first person (Ahmad with id 0) was always seated first to keep the results somewhat consistent. The three searches were implemented as the following:

- Greedy Search:

Each unexplored (not seated) person was iterated to find the one with the least dislike percentage in relative to the last individual inside the seating arrangement. This continued until all individuals were seated around the table. The heuristic function (h(x) = x, where x represents the dislike percentages) was used to evaluate the cost between neighbors.

CODE:

```
public static void GreedySearch(int startState) {

        List<Integer> table = new ArrayList<>();// keep track seating arrangement
        table.add(startState);// seat the first person
        double totalCost = 0;// keep track of total cost of seating arrangement


        // loop until the number of people seated is equal to the total number of people
        while (table.size() < dislikeTable.getNumPeople()) {


        double leastCost = Double.MAX_VALUE;// setting to max to use in finding the min cost
        int nextPerson = 0;// to store the id of the next person to be sited
        int lastPerson = table.get(table.size() - 1);// the last seated person

        for (int i = 0; i < dislikeTable.getNumPeople(); i++) {// loop all people

        // checking if the current person i has a the lowest cost with the last person
        // seated, this person must not be already seated
        if (dislikeTable.getDislike(lastPerson, i) < leastCost && !table.contains(i)) {

                leastCost = dislikeTable.getDislike(lastPerson, i);// save the cost to seat this person
                nextPerson = i;// save person
                }
        }

        table.add(nextPerson);// add person to seating
        totalCost += leastCost;// add the cost to total


        }

        // since the table is circular, we must add to the total cost the dislike
        // between the last and first person seated at the table
        int lastSeated = table.get(table.size() - 1);// last person in array
        totalCost += dislikeTable.getDislike(startState, lastSeated);// adding cost

        printPeople(table, totalCost, "**Greedy Algorithm**");// print

        }
```

- Uniform Cost Search:

A priority queue (min heap) was used to explore the different seating arrangements based on their cumulative dislike costs. At each step, each non-explored individual is separately added to the end of the seating array, along with the cumulative cost between the neighbors. These different arrangement nodes were pushed into the heap waiting to be polled with the minimum cumulative cost. This was implemented using Non-Linear function $g(x) = x^2$, which represents the actual dislike cost.

CODE:

```
public static void UCSAlgorithm(int startState) {

        List<Integer> table = new ArrayList<>();// table to seat people
        table.add(startState);// add first person to seating arrangement

        // minimum heap holding different seating arrangements and costs
        PriorityQueue<NodeState> minHeap = new PriorityQueue<>(new NodeStateComparator());

        // adding the initial table state (one person sitting alone with cost equal t0 0) to the min heap
        minHeap.add(new NodeState(table, 0));

        // enter loop until priority queue is empty or execution is stopped
        while (!minHeap.isEmpty()) {

                // get the minimum seating arrangement in the heap
                NodeState currentState = minHeap.poll();

                // when everyone is seated (done), print solution
                if (currentState.getSeating().size() == dislikeTable.getNumPeople()) {

                double cost = calculateCost(currentState.getSeating(), 1);// calculate total cost
                printPeople(currentState.getSeating(), cost, "**UCS Algorithm**");// print
                return;
                }

                // loop in the people not sitting yet
                for (int i = 0; i < dislikeTable.getNumPeople(); i++) {

                // making sure table doesn't already have the person i sitting
                // equivalent to checking if already explored
                if (!currentState.getSeating().contains(i)) {

                // adding person i to the end of the current table arrangement
                // this forms a new seating arrangement
                List<Integer> newArrangement = new ArrayList<>(currentState.getSeating());
                newArrangement.add(i);

                // getting the last seated person before adding i
                int last = currentState.getSeating().get(currentState.getSeating().size() - 1);

                // calculating accumulative cost of dislike between seated people
                double newCost = currentState.getCost() + Math.pow(dislikeTable.getDislike(last, i), 2);

                // adding the new arrangement to the heap
                minHeap.add(new NodeState(newArrangement, newCost));
                }
        }
}

        System.out.println("Problem?");
}
```

- <u>A* Search</u>:

Almost the same as the UCS, but A*s' state node contained the cumulative cost of reaching that state (g-cost from g(x)) and an estimated cost (h-cost from h(x)) to get closer to the goal by decreasing the conflicts. The heuristic cost estimated the additional dislike between the last seated individual and the next one to be seated, guiding the search towards a more optimal solution.

CODE:

```java
public static void AStarAlgorithm(int startState) {

        List<Integer> table = new ArrayList<>();// table to seat people
        table.add(startState);// add first person to seating arrangement

        // heap to hold the state nodes containing the seat arrangements and costs
        PriorityQueue<NodeState> minHeap = new PriorityQueue<>(new NodeStateComparator());

        // creating first node state with h-cost and g-cost as 0 (initial state)
        NodeState first = new NodeState(table, 0, 0);

        // adding initial state to heap
        minHeap.add(first);

        // until priority queue is empty or terminated by another line of code
        while (!minHeap.isEmpty()) {

                // get node with the minimum cost seating arrangement inside heap
                NodeState currentState = minHeap.poll();

                // everyone is seated, print solution
                if (currentState.getSeating().size() == dislikeTable.getNumPeople()) {

                        // calculate the final seating arrangement cost
                        double cost = calculateCost(currentState.getSeating(), 2);
                        printPeople(currentState.getSeating(), cost, "**A* Algorithm**");// print
                        return;
                }

                // loop through everyone not yet seated
                for (int i = 0; i < dislikeTable.getNumPeople(); i++) {
                        if (!currentState.getSeating().contains(i)) {// not seated

                        // adding i to the new seating arrangement
                        List<Integer> newArrangement = new ArrayList<>(currentState.getSeating());
                        newArrangement.add(i);

                        // last person seated before i
                        int last = currentState.getSeating().get(currentState.getSeating().size() - 1);

                        // getting the real dislike cost between i and the previously last person added
                        // adding this cost to the previous accumulated one associated with this seating
                double gCost = currentState.getgCost() + Math.pow(dislikeTable.getDislike(last, i), 2);

                        // the heuristic cost is what gets us closer to a better goal (least conflict)
                        // by adding the heuristic dislike between the last two people
                        double hCost = dislikeTable.getDislike(last, i);

                        // pushing this state into the min heap
                        minHeap.add(new NodeState(newArrangement, gCost, hCost));
                        }
                }
        }
        System.out.println("Problem?");

}
```

- Other Relevant Functions:

  CalculateCost function, which calculates the cost of the seating arrangement depending on the search algorithm mode.

  CODE:

```
public static double calculateCost(List<Integer> seatingArrangement, int mode) {

        double totalCost = 0;// end cost to return
        int numPeople = seatingArrangement.size();

        // looping through all of the seated people
        // saving them as pairs to calculated the dislike between them
        for (int i = 0; i < numPeople; i++) {
                int personA = seatingArrangement.get(i);// first person

                /*
                * setting index as: ((i + 1) % numPeople) wraps around list to include the last
                * person making sure the index never exceeds the list size. In other words, it
                * connects back to the beginning of the list.
                */
                int personB = seatingArrangement.get((i + 1) % numPeople);// second person

                // obtaining the dislike cost between the two selected people
                double costAB = dislikeTable.getDislike(personA, personB);

                if (mode == 0)// heuristic cost for greedy
                        totalCost += costAB;// h(x) = x

                else if (mode == 1)// real cost for UCS
                        totalCost += Math.pow(costAB, 2);// g(x) = x^2

                else// real + heuristic costs for A*
                        totalCost += costAB + Math.pow(costAB, 2);// f(n) = h(x) + g(x)
        }

        return totalCost;
}
```

Results:

Console:

```
**Greedy Algorithm**
Seating Arrangement:
[Ahmad,Fuad,Salem,Hani,Ibrahim,Samir,Khalid,Ayman,Hakam,Kamal,]
Search Cost: 4.390000000000001
Conflict Cost: 2.6529
Execution time: 1 milliseconds



**UCS Algorithm**
Seating Arrangement:
[Ahmad,Fuad,Salem,Hani,Ibrahim,Kamal,Samir,Khalid,Ayman,Hakam,]
Search Cost: 1.5393999999999999
Conflict Cost: 1.5393999999999999
Execution time: 19 milliseconds



**A* Algorithm**
Seating Arrangement:
[Ahmad,Hakam,Ayman,Hani,Salem,Fuad,Kamal,Ibrahim,Samir,Khalid,]
Search Cost: 5.576700000000001
Conflict Cost: 1.7467000000000001
Execution time: 7 milliseconds
```

Analysis:

The Greedy Algorithm provided a solution quickly with the lowest execution time; however, it had the least optimal solution with the largest conflict cost of 2.6529. This algorithm was the best in terms of time but was the least optimal.

UCS Algorithm found a solution with the lowest conflict cost of 1.5394 among the three algorithms. However, it had the longest execution time compared to Greedy and A*. This algorithm was the best in terms of optimality but the worst in execution time.

A* Algorithm had a balance between optimality and efficiency by considering both the actual and heuristic costs. While it had a slightly higher conflict cost compared to UCS with value of 1.7467, it achieved a solution with a lower execution time. For these reasons, A* was the best algorithm in terms of overall performance.

In short, UCS systematically explored the search space, finding the optimal solution with lowest cost, while Greedy prioritizes immediate gains, resulting in higher cost; A* balances optimality and efficiency for moderate cost and low execution time.