Homework Assignment

Calin Marius Alexandru

May 2024

1 Introduction

GitHub Repository Link: https://github.com/CalinMariusAlex/AI_Homework

2 Problem statement

TSP - Traveling Salesman problem

Given a list of cities and the distances between each pair of cities, what is the route of minimum cost that visits each city exactly once and returns to the origin city? The cost function of the solution must be the minimization of the longest distance between two consecutive cities.

3 Pseudocode of the algorithms

Uniform Cost Search:

```
function UCS_TSP(distance_matrix):
    n = number of cities in distance_matrix
    priority_queue = [(0, 0, [0], 0)]  // (cost, current_city, path, max_dist)
    best_path = None
    min_max_dist = infinity

    while priority_queue is not empty:
        cost, current_city, path, max_dist = dequeue(priority_queue)

        if length of path == n:
            return_path = path + [0]
            return_dist = max(max_dist, distance_matrix[current_city][0])
            if return_dist < min_max_dist:
                min_max_dist = return_dist
                best_path = return_path
            continue

        for next_city from 0 to n-1:
            if next_city is not in path:
                next_cost = cost + distance_matrix[current_city][next_city]
                next_max_dist = max(max_dist, distance_matrix[current_city][next_city])
                enqueue(priority_queue, (next_cost, next_city, path + [next_city], next_max_dist))

    return best_path, min_max_dist
```

Breath-First Search:

```
function BFS_TSP(distance_matrix):
    n = number of cities in distance_matrix
    queue = [(0, [0], 0)]  // (current_city, path, max_dist)
    best_path = None
    min_max_dist = infinity

    while queue is not empty:
        current_city, path, max_dist = dequeue(queue)
```

```
        if length of path == n:
            return_path = path + [0]
            return_dist = max(max_dist, distance_matrix[current_city][0])
            if return_dist < min_max_dist:
                min_max_dist = return_dist
                best_path = return_path
            continue

        for next_city from 0 to n-1:
            if next_city is not in path:
                next_max_dist = max(max_dist, distance_matrix[current_city][next_city])
                enqueue(queue, (next_city, path + [next_city], next_max_dist))

    return best_path, min_max_dist
```

A* Search:

```
function MST_Heuristic(distance_matrix, remaining_cities):
    if remaining_cities is empty:
        return 0
    sub_matrix = distance_matrix for rows and columns in remaining_cities
    mst = minimum_spanning_tree(sub_matrix)
    return maximum weight in mst

function A_Star_TSP(distance_matrix):
    n = number of cities in distance_matrix
    priority_queue = [(0, 0, [0], 0, set(1 to n-1))]  // (f_score, current_city, path, max_dist, 
remaining_cities)
    best_path = None
    min_max_dist = infinity

    while priority_queue is not empty:
        f_score, current_city, path, max_dist, remaining_cities = dequeue(priority_queue)

        if remaining_cities is empty:
            return_path = path + [0]
            return_dist = max(max_dist, distance_matrix[current_city][0])
            if return_dist < min_max_dist:
                min_max_dist = return_dist
                best_path = return_path
            continue

        for next_city in remaining_cities:
            next_path = path + [next_city]
            next_max_dist = max(max_dist, distance_matrix[current_city][next_city])
            next_remaining = remaining_cities - {next_city}
            heuristic_cost = MST_Heuristic(distance_matrix, next_remaining)
            f_score = next_max_dist + heuristic_cost
            enqueue(priority_queue, (f_score, next_city, next_path, next_max_dist, next_remaining))

    return best_path, min_max_dist
```

4 Application outline.

- The high-level architectural overview of the application

  The application is designed to solve the problem using three algorithms: BFS, UCS and A* Search

- The specification of the input data format

  The input data is a matrix of integers, each element represents the distance between city 'i' and city 'j'

- The specification of the output data format

  The output data consists of:

  - best_path: a list of integers representing the order of cities in the optimal route
  - min_max_dist: An integer representing the minimum of the longest distances between consecutive cities in the best path.

- The list of all the modules in the application and their description

  The modules are:

  - Input Module
  - Output Module
  - Search Algorithms Module
  - Utility Module

- The list of all the functions in the application, grouped by modules; for every function the following details must be provided:

  Input Module:

  - read_distance_matrix:
    - Description: Reads the input distance matrix
    - Parameters: 'filename' is ht name of the file containing the distance matrix
    - return value: the distance matrix

  Output Module:

  - display_result:
    - Description: Formats and displays the output
    - Parameters: best_path, min_max_dist
    - Return value: None

  Search Algorithms Module:

  - bfs_tsp:
    - Description: Solves the TSP using Breath-First Search

- Parameters: distance_matrix
- Return Value: A tuple containing the best path and the minimum of the longest distances

- ucs_tsp:
  - Description: Solves the TSP using Uniform Cost Search.
  - Parameters: distance_matrix
  - Return Value: A tuple containing the best path and the minimum of the longest distances.
- a_star_tsp:
  - Description: Solves the TSP using A* Search.
  - Parameters: distance_matrix
  - Return Value: A tuple containing the best path and the minimum of the longest distances.

Utility Module:

- mst_heuristic:
  - Description: Calculates the MST heuristic for a given set of remaining cities.
  - Parameters: distance_matrix, remaining_cities(set of remaining cities)
  - Return Value: The maximum edge weight in the MST of the remaining cities.

5 Experiments and evaluation

We run 3 different matrices:

- distance_matrix = [
  [0, 12, 10, 19],
  [21, 0, 25, 30],
  [30, 15, 0, 14],
  [10, 10, 23, 0]
  ]

  Best path: [0, 2, 3, 1, 0]
  Minimum of the longest distances: 21

- distance_matrix = [
  [0, 10, 15, 20, 25],
  [10, 0, 35, 25, 30],
  [15, 35, 0, 30, 20],
  [20, 25, 30, 0, 15],
  [25, 30, 20, 15, 0]
  ]

  Best path: [0, 1, 3, 4, 2, 0]
  Minimum of the longest distances: 25

- distance_matrix = [
  [0, 34, 19, 42, 27, 35],

```
    [34, 0, 23, 12, 25, 30],
    [19, 23, 0, 27, 15, 28],
    [42, 12, 27, 0, 19, 24],
    [27, 25, 15, 19, 0, 32],
    [35, 30, 28, 24, 32, 0]
]

Best path: [0, 2, 5, 3, 1, 4, 0]
Minimum of the longest distances: 28
```

6 Conclusions

In conclusion, the Traveling Salesman problem can be solved with the BFS, UCS and A* algorithms.

References

[1] Geeksforgeeks, *Breadth First Search or BFS for a Graph*, https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[2] Geeksforgeeks, *Uniform-Cost Search (Dijkstra for large Graphs)*, https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

[3] Geeksforgeeks, *A* Search Algorithm in Python*, https://www.geeksforgeeks.org/a-search-algorithm-in-python/

[4] Geeksforgeeks, Travelling Salesman Problem using Dynamic Programming, https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/