AI Homework Assignment

GISGINIS MICHAIL CEN2.2A

1 Introduction

Github Project: https://github.com/MIKEGISGINIS1/HOMEWORK-ASSIGNMENT/tree/main

2 Problem statement

This project is all about finding the shortest route that visits a given list of cities exactly once and returns to the starting point. We tackle this classic problem using three different algorithms: Breadth-First Search (BFS), Least-Cost (Uniform Cost) Search, and A* Search.

3 Pseudocode of the algorithms

Pseudocode for `tsp_utils.py`

This module contains utility functions used across different algorithms.

```
function calculate_path_cost(path, distances):
    cost = 0
    for i from 0 to length(path) - 2:
        cost += distances[path[i]][path[i + 1]]
    cost += distances[path[length(path) - 1]][path[0]]  # Return to the
starting city
    return cost

function calculate_heuristic(path, distances):
    remaining_cities = set(range(length(distances))) - set(path)
    if length(remaining_cities) == 0:
        return 0
    last_city = path[length(path) - 1]
    min_edge_cost = minimum(distances[last_city][city] for city in
remaining_cities)
    return min_edge_cost

function generate_random_distance_matrix(num_cities):
    distances = random integer matrix of size (num_cities, num_cities)
between 1 and 100
    set diagonal of distances to 0
    return distances
```

Pseudocode for `tsp_bfs.py`

This module contains the BFS implementation for TSP.

```
function bfs_tsp(distances):
    num_cities = length(distances)
    min_cost = infinity
    best_path = None
    queue = [(0, [0])]  # (current_city, path)

    while queue is not empty:
        current_city, path = queue.pop(0)
        if length(path) == num_cities:
            cost = calculate_path_cost(path, distances)
            if cost < min_cost:
                min_cost = cost
                best_path = path
        else:
            for city from 0 to num_cities - 1:
                if city is not in path:
                    queue.append((city, path + [city]))

    return best_path, min_cost
```

Pseudocode for `tsp_ucs.py`

This module contains the Least-Cost (Uniform Cost) Search implementation for TSP.

```
function least_cost_search_tsp(distances):
    num_cities = length(distances)
    min_cost = infinity
    best_path = None
    queue = priority queue with [(0, [0])]  # (cost, path)

    while queue is not empty:
        cost, path = pop from priority queue(queue)
        if length(path) == num_cities:
            total_cost = calculate_path_cost(path, distances)
            if total_cost < min_cost:
                min_cost = total_cost
                best_path = path
        else:
            for city from 0 to num_cities - 1:
                if city is not in path:
                    new_cost = cost + distances[path[length(path) - 1]][city]
                    insert into priority queue(queue, (new_cost, path +
[city]))

    return best_path, min_cost
```

Pseudocode for `tsp_astar.py`

This module contains the A* Search implementation for TSP.

```
function a_star_tsp(distances):
    num_cities = length(distances)
    min_cost = infinity
    best_path = None
    queue = priority queue with [(0, [0])]  # (cost + heuristic, path)

    while queue is not empty:
        cost, path = pop from priority queue(queue)
        if length(path) == num_cities:
            total_cost = calculate_path_cost(path, distances)
            if total_cost < min_cost:
                min_cost = total_cost
                best_path = path
        else:
            for city from 0 to num_cities - 1:
                if city is not in path:
                    new_cost = cost + distances[path[length(path) - 1]][city]
                    heuristic_value = calculate_heuristic(path + [city],
distances)
                    insert into priority queue(queue, (new_cost +
heuristic_value, path + [city]))

    return best_path, min_cost
```

Pseudocode for `main.py`

This script runs the experiments and compares the algorithms.

```
function run_experiments():
    num_cities_list = [4, 8, 12]

    for num_cities in num_cities_list:
        print "Running experiments for {num_cities} cities..."
        distances = generate_random_distance_matrix(num_cities)

        # BFS
        start_time = current time
        bfs_path, bfs_cost = bfs_tsp(distances)
        bfs_time = current time - start_time

        # Least-Cost Search
        start_time = current time
```

```
        ucs_path, ucs_cost = least_cost_search_tsp(distances)
        ucs_time = current time - start_time

        # A* Search
        start_time = current time
        astar_path, astar_cost = a_star_tsp(distances)
        astar_time = current time - start_time

        print "Results for {num_cities} cities:"
        print "BFS Path: {bfs_path}, Cost: {bfs_cost}, Time: {bfs_time:.4f}s"
        print "UCS Path: {ucs_path}, Cost: {ucs_cost}, Time: {ucs_time:.4f}s"
        print "A* Path: {astar_path}, Cost: {astar_cost}, Time:
{astar_time:.4f}s"
        print "-------------------------------------------------"

if __name__ == "__main__":
    run_experiments()
```

4.Overview of the Architecture

This project tackles the Traveling Salesman Problem (TSP) by employing three distinct algorithms: Breadth-First Search (BFS), Least-Cost (Uniform Cost) Search, and A* Search. To maintain a clean and modular design, each algorithm is implemented in its own module, with shared utility functions stored in a separate utility module.

Input Data Format

The input for our application is a distance matrix representing the distances between each pair of cities. This matrix is a square matrix where each element at position (i, j) represents the distance between city i and city j. Additionally, the initial city (starting point) is specified.

Output Data Format

The output includes the optimal route that visits each city exactly once and returns to the starting city. This route is presented as a list of city indices in the order they are visited, along with the total cost of this route.

Modules Overview

- **BFS Module**: Implements the Breadth-First Search algorithm for solving the TSP.
- **Least-Cost Search Module**: Implements the Least-Cost (Uniform Cost) Search algorithm for solving the TSP.
- *A Search Module**: Implements the A* Search algorithm for solving the TSP.
- **Utility Module**: Provides utility functions used across different algorithms, such as calculating path costs and heuristics.

Functions Breakdown

BFS Module:

- `bfs_tsp(distances)`

- **Purpose**: Solves the TSP using the Breadth-First Search algorithm.
- **Parameters**: `distances` - Distance matrix representing the distances between each pair of cities.
- **Returns**: Optimal route as a list of city indices.

Least-Cost Search Module:

- `least_cost_search_tsp(distances)`
  - **Purpose**: Solves the TSP using the Least-Cost (Uniform Cost) Search algorithm.
  - **Parameters**: `distances` - Distance matrix representing the distances between each pair of cities.
  - **Returns**: Optimal route as a list of city indices.

*A Search Module:**

- `a_star_tsp(distances)`
  - **Purpose**: Solves the TSP using the A* Search algorithm.
  - **Parameters**: `distances` - Distance matrix representing the distances between each pair of cities.
  - **Returns**: Optimal route as a list of city indices.

Utility Module:

- `calculate_path_cost(path, distances)`
  - **Purpose**: Calculates the total cost of a given path based on the distance matrix.
  - **Parameters**: `path` - List of city indices representing the route, `distances` - Distance matrix.
  - **Returns**: Total cost of the path.
- `calculate_heuristic(path, distances)`
  - **Purpose**: Calculates the heuristic value for A* Search.
  - **Parameters**: `path` - List of city indices representing the route, `distances` - Distance matrix.
  - **Returns**: Heuristic value for the given path.
- `generate_random_distance_matrix(num_cities)`
  - **Purpose**: Generates a random distance matrix for a given number of cities.
  - **Parameters**: `num_cities` - Number of cities.
  - **Returns**: A random distance matrix.

5. Experimental Analysis

Experiment Design

To comprehensively test the performance and scalability of our algorithms, we will define input instances of various sizes, ranging from small to large. We will generate random distance matrices with different sizes to represent distances between cities. Additionally, we might use predefined distance matrices for specific scenarios or edge cases.

Metrics for Evaluation

- **Time Taken**: Measure the duration each algorithm takes to find the optimal solution.
- **Quality of Solution**: Compare the cost of the solutions obtained by each algorithm with the known optimal solution (if available).
- **Handling Larger Inputs**: Analyze how each algorithm scales with increasing input size.

Testing Scenarios

- **Small Input Instances**:
    - Use small-sized distance matrices (e.g., 4x4 or 5x5) to test the correctness and efficiency of the algorithms.
    - Compare the solutions obtained by each algorithm and verify if they match the expected optimal solution.
    - Measure and compare the time taken by each algorithm.
- **Medium Input Instances**:
    - Increase the size of the distance matrices (e.g., 8x8 or 10x10) to evaluate the scalability of the algorithms.
    - Analyze the time taken and solution quality for each algorithm.
    - Observe if any algorithm shows significant performance degradation with larger inputs.
- **Large Input Instances**:
    - Use larger distance matrices (e.g., 15x15 or 20x20) to further test scalability.
    - Focus on the time taken and solution quality of each algorithm.
    - Determine if any algorithm becomes impractical for large-scale instances.

Results and Findings

- **Analyze Results**: Evaluate the findings to identify the strengths and weaknesses of each algorithm.
- **Performance Comparison**: Compare BFS, Least-Cost Search, and A* Search in terms of time taken and solution quality.
- **Best Performing Algorithm**: Identify which algorithm performs best under different scenarios, such as small vs. large instances or dense vs. sparse distance matrices.
- **Conclusions**: Based on the experimental findings, conclude the suitability of each algorithm for practical applications.

Optimization Potential

- **Identify Areas for Improvement**: Look for potential areas for optimization within each algorithm, such as data structures and heuristics.
- **Test Optimization Techniques**: Experiment with different optimization techniques to improve efficiency and scalability.
- **Evaluate Impact**: Re-run experiments to assess the impact of optimizations on algorithm performance.

6. Conclusions

In conclusion, the Traveling Salesman Problem (TSP) is a challenging optimization problem that can be approached using various strategies. My implementations of BFS, Least-Cost Search, and A* Search each offer different trade-offs between it being somewhat optimal and efficiencient. While something likes BFS can guarantee optimal solutions,it may not be practical for large problem instances due to high computational cost. Heuristic strategies, such as A* Search, strike a balance between all the above making them suitable for real-world applications. This project demonstrates the implementation and comparison of these algorithms, providing insights into their performance and scalability. Overall, this journey through solving TSP has been both educational and rewarding, revealing the strengths and limitations of each algorithm, and I hope the effort will be appreciated.