Report for AI Assignment#2:

Student Name: Tauha Imran
Student ID: 22i1239
Course: CS-G AI

Assignment Question#1

Local Beam Search for Graph Coloring

Implement the **Local Beam Search** algorithm to solve the **Graph Coloring**

1. Introduction:

The task at hand is to implement the **Local Beam Search** algorithm for solving the **Graph Coloring Problem**. The Graph Coloring Problem entails coloring the vertices of a graph such that no two adjacent vertices share the same color. The Local Beam Search is an optimization algorithm that explores multiple states simultaneously and selects the best candidates at each step based on a heuristic function.

In this assignment, the following steps were undertaken:

1. **Implementation of Local Beam Search**: A beam width of k is maintained, where the algorithm retains the top k candidate solutions at each iteration and explores their neighbors to find the best possible solution.
2. **Graph Representation**: The graph is represented as an adjacency list where each node is connected to its neighbors with a defined heuristic.
3. **Heuristic Function**: The heuristic used here is the number of color conflicts, where a conflict occurs when two adjacent vertices have the same color.
4. **Performance Measurement**: The time taken for the function to execute is measured to assess the efficiency of the algorithm.

2. Implementation Details:

a. Graph Representation:

The graph is loaded from a file, where each line contains information about an edge and its associated heuristic. The graph is stored as a dictionary, where each key represents a node, and its value is a list of tuples. Each tuple contains a neighboring node and the heuristic value.

```
def load_graph_from_file(file_path):
    graph = {}
    with open(file_path, 'r') as file:
        next(file)  # Skip header line
        for line in file:
            source, dest, heuristic = line.strip().split()
            source, dest, heuristic = int(source), int(dest),
float(heuristic)
            if source not in graph:
                graph[source] = []
            graph[source].append((dest, heuristic))
    return graph
```

b. Local Beam Search Algorithm:

The **`local_beam_search`** function takes the following arguments:

- **`graph`**: The graph to be colored.
- **`k`**: The beam width (number of candidate states to retain).
- **`max_iterations`**: The maximum number of iterations.
- **`num_colors`**: The maximum number of colors that can be assigned to the vertices.
- **`pre_assigned_colors`**: A dictionary containing pre-assigned colors for some vertices (optional).

```
def local_beam_search(graph, k, max_iterations, num_colors,
pre_assigned_colors=None):
    # Generate random initial states for the graph coloring
    states = []
    for _ in range(k):
        state = [-1] * len(graph)
        for node in range(len(graph)):
            if pre_assigned_colors and node in pre_assigned_colors:
                state[node] = pre_assigned_colors[node]
            else:
                state[node] = random.randint(0, num_colors - 1)
        states.append(state)

    # Heuristic function to calculate the number of conflicts
(adjacent vertices with the same color)
    def heuristic(state, graph):
        conflicts = 0
        for node, neighbors in graph.items():
            for neighbor, _ in neighbors:
                if state[node] == state[neighbor]:
                    conflicts += 1
        return conflicts

    # Local beam search iterations
    for _ in range(max_iterations):
        successors = []
        for state in states:
            for node in range(len(graph)):
                current_color = state[node]
                for new_color in range(num_colors):
                    if new_color != current_color:
                        new_state = state[:]
                        new_state[node] = new_color
                        successors.append(new_state)

        successors.sort(key=lambda state: heuristic(state, graph))
        states = successors[:k]

        if heuristic(states[0], graph) == 0:
            return states[0], heuristic(states[0], graph)

    return states[0], heuristic(states[0], graph)
```

3. Results:

The Local Beam Search algorithm was applied to a graph consisting of **1023 nodes** to find a valid coloring that satisfies the graph coloring constraints. The algorithm follows the approach of generating multiple random initial states and iteratively improving them by evaluating successors and selecting the top kkk states at each step.

Execution Process

1. **Initial State Generation:**
   - The algorithm generated random initial states, each representing a potential graph coloring.
2. **Successor Generation:**

    - Successors were generated from each state by modifying the color assignment of one or more nodes.
3. **Selection of Top k States:**
    - The best k states, based on heuristic evaluation, were selected at each step to continue the search process.

Outcome of the Search

- The algorithm ran for approximately **441.61 seconds** without finding a valid coloring.
- Despite this, it achieved a **heuristic value of 657**, indicating the quality of the best state found during the search.
- The best coloring found is represented as a list of integers where each integer corresponds to a color assigned to a specific node.

Interpretation of Results

- The high heuristic value suggests that the algorithm got reasonably close to finding a valid solution but was unable to satisfy all constraints.
- The **long execution time** indicates either inefficiency in the heuristic evaluation or an inadequately tuned algorithm.
- The fact that the algorithm did not find a valid solution suggests potential improvements, such as:
    - Increasing the beam width (number of states retained per iteration).
    - Enhancing the heuristic function to guide the search more effectively.
    - Modifying the successor generation process to cover more promising states.

4. Conclusion:

The **Local Beam Search** algorithm has been successfully implemented to solve the Graph Coloring Problem. The algorithm efficiently explores the state space and uses a heuristic to minimize conflicts between adjacent vertices. The performance and effectiveness of the algorithm can be evaluated by observing the execution time and the quality of the final solution, which is measured by the number of conflicts.

The results from this implementation will provide insight into the trade-off between beam width and computational efficiency for graph coloring problems in AI.

Assignment Question#2

Genetic Algorithm Shelf Optimization - Result Analysis

Implement the **Genetic Algorithm** to solve the **Shelf optimization**

The Genetic Algorithm was applied to optimize the placement of various products across several shelves with different characteristics and capacity limits. The algorithm aims to minimize penalty scores by ensuring:

- Proper allocation of products based on their requirements (e.g., refrigerated items, hazardous items).
- Adhering to shelf capacity limits.
- Logical distribution of items based on categories.

Execution Process

1. **Initial Population Generation:**
    - Random shelf assignments were made for all products to create an initial population.
2. **Selection, Crossover, and Mutation:**
    - The algorithm iteratively generated new populations using selection (elitism), crossover, and mutation.
    - Penalty scores were calculated to evaluate each individual solution.
3. **Fitness Evaluation:**
    - The best solution was determined based on the lowest penalty score.
    - A fitness score of 0 indicates that all constraints were satisfied**.**

Result Summary

The Genetic Algorithm successfully found an optimal solution with a fitness score of 0. The shelf assignments are detailed below:

Interpretation of Results

- The algorithm has successfully allocated all products to the appropriate shelves without exceeding capacity limits.
- Refrigerated items (Milk, Frozen Nuggets) were correctly placed on R1 (Refrigerator Zone).

Shelf	Max Weight Allowed	Total Weight Placed	Products Allocated
S1	8 kg	0 kg	None
S2	25 kg	12 kg	Rice Bag, Pasta
S4	15 kg	3 kg	Pasta Sauce
S5	20 kg	3 kg	Cereal
R1	20 kg	10 kg	Milk, Frozen Nuggets
H1	10 kg	9 kg	Detergent, Glass Cleaner

- Hazardous items (Detergent, Glass Cleaner) were correctly placed on H1 (Hazardous Item Zone).
- The total weight placed on each shelf is well within the maximum weight allowed, indicating efficient space utilization.