CS271P Project Draft Design

Shengtong Sun (65085562)
Chuqi Wang (79167724)
Zhihang Feng (26827566)
Team 67

I

Pseudocode of Branching and Bounding DFS solving Traveling Salesman Problem:

```
function BranchAndBound-DFS(graph, start_city):
   number_cities ← length(graph)
   #Initialize the node and assign the start city.
   initial_node ← {path: list(start_city), lowerbound: 0, level: 0}
   best_solution ← {path: list(), upperbound: ∞}

   while not STACK.ISEMPTY():
      current_node ← STACK.POP()
      #Pruning: If the lower bound of the current node is greater than or equal to the currently
      known optimal solution, it is no longer expanded.
      if current_node[lowerbound] >= best_solution[upperbound]:
         continue
      #The leaf node has been reached
      if current_node[level] == number_cities - 1:
         #Assign the cost to be the cost of going from the last city in the path back to the start
         city.
         cost ← cost of (current_node[path][last_city] + current_node[path][start_city])
         #If the total cost of this path is less than the upper bound, then add the start city to the
         path and update the upper bound.
         if current_node[lowerbound] + cost < best_solution[upperbound]:
            best_solution[path] ← current_node[path] + current_node[path][start_city]
            best_solution[upperbound] ← current_node[lowerbound]+cost
               continue
      #Expand the nodes from number_cities.
      for city in number_cities do:
         #For cities that haven't been visited, using DFS to go through child nodes, call a
         heuristic function to calculate the lower bound of the cities. Push child nodes into the
         stack.
         if city not in current_node[path]:
            child_path ←current_node[path] + city
            child_bound ← calculate_bound(graph, child_path, city)
            child_node ←{path: child_path, lowerbound: childbound, level: current_node[level] +1
            STACK.PUSH(child_node)
   return best_solution
```

```
function calculate_bound(graph,path,current_city)
        # compute the heuristic lower bound, using the weight of the minimum spanning tree as the lower bound
        return calculate_mst_weight(graph, path) + min_edge_weight_to_unvisited(path, current_city)

function prim_mst_weight(graph, path):
    n ← length(graph)
    unvisited_cities ← set of all cities - set of cities in path
    mst_weight ← 0
    visited ← set containing last city in path
    while unvisited_cities is not empty:
        min_edge ← positive infinity
        min_city ← None
        # Find the minimum edge connected to the current set of visited cities
        for city in visited:
            for neighbor in unvisited_cities:
                edge_weight ← graph[city][neighbor]
                if edge_weight < min_edge:
                    min_edge ← edge_weight
                    min_city ← neighbor
        # Add the weight of the minimum edge to the MST weight
        mst_weight ← mst_weight + min_edge
        # Update the visited and unvisited sets
        visited ← visited + min_city
        unvisited_cities ← unvisited_cities - min_city
    return mst_weight

function min_edge_weight_to_unvisited(path, current_city):
    # compute the minimum edge weight to the unvisited city
    min_weight ← positive infinity
    unvisited_cities ← set of all cities - set of cities in path
    for each city in unvisited_cities:
        weight ← graph[current_city][city]
        if weight < min_weight:
            min_weight ← weight
    return min_weight
```

II

Pseudocode of Stochastic Local Search - Simulated Annealing Algorithm:

```
function simulated_annealing(graph, initial_solution, initial_temperature, cooling_rate):
    current_solution ← initial_solution
    best_solution ← initial_solution
    current_temperature ← initial_temperature

    while current_temperature > 0.1:  # Set termination condition
        # Generate a new solution in the neighborhood
        new_solution ← generate_neighbor(current_solution)

        # Calculate the cost difference between the new and current solutions
        cost_difference ← calculate_cost(graph, new_solution) - calculate_cost(graph,
current_solution)

        # Accept the new solution if it is better or with a certain probability for worse solutions
        if cost_difference < 0 or accept_worse_solution(cost_difference, current_temperature):
            current_solution ← new_solution

        # Update the best solution
        if calculate_cost(graph, current_solution) < calculate_cost(graph, best_solution):
            best_solution ← current_solution

        # Decrease the temperature
        current_temperature ← current_temperature * cooling_rate

    return best_solution

function generate_neighbor(solution):
    # Generate a new solution in the neighborhood of the current solution
    # For example, randomly swap the positions of two cities

    # Copy the current solution to avoid modifying the original solution
    modified_solution ← solution.copy()

    # Randomly select two different cities
    city_index1, city_index2 ← random.sample(range(len(solution)), 2)
```

```
    # Swap the positions of the two cities
    modified_solution[city_index1], modified_solution[city_index2] ←
modified_solution[city_index2], modified_solution[city_index1]

    return modified_solution

function calculate_cost(graph, solution):
    # Calculate the total cost of the solution
    total_cost ← 0
    n ← len(solution)

    # From the starting city to the last city
    for i in range(n - 1):
        current_city ← solution[i]
        next_city ← solution[i + 1]
        total_cost ← total_cost + graph[current_city][next_city]

    # Cost of returning to the starting city
    total_cost ← total_cost + graph[solution[-1]][solution[0]]

    return total_cost

function accept_worse_solution(cost_difference, temperature):
    #Accept worse solutions with a certain probability
    probability ← exp(-cost_difference / temperature)
    random_number ← random()   # Generate a random number between 0 and 1
    return random_number < probability
```

Branch and Bound Depth First Algorithm

Algorithm Design

In this algorithm, we assume that one city can be reached by any other city, that is all the cities are connected. BnB DFS uses regular DFS in finding the first solution and sets its cost as the initial upper bound, and then it will backtrack cities that can find other possible better solutions. The pruning process will be related to the comparison of the lower bound and upper bound, in which the lower bound is evaluated based on heuristics. The algorithm will take a graph as an input, and it will produce the shortest traveling path. The first city in the graph will be chosen at the start.

Algorithm Assessment

The worst run time of BnB DFS is O(b^d). Assuming there are n cities, then there could be (n-1) branching factors and the depth of the search can be n. Thus, the worst runtime of BnB DFS is O($(n - 1)^n$). In addition, since the input is a numpy array that represents the graph, the worst space complexity is O($(n - 1)^n$). The time complexity and space complexity can be improved if a better heuristic is chosen. In a later session, the prim's algorithm for the Minimum Spanning Tree will be discussed, its time complexity is O($n^2 * log(n)$) and its space complexity is O($n^2 + n$).

Data Structures

- Input: numpy array, an integer represents the start city
- Node: a dictionary type with three keys which
    - Path: with a list value that records the current path
    - Lower bound: with an integer value that shows the lower bound of the current node
    - Level: an integer value that records the number of visited cities
- Best solution: a dictionary type with three keys which
    - Path: a list value that represents the current solution or the best solution

    - Upper bound: a float value that sets the cost of the current best solution as the upper bound
- Priority queue: used to select minimum-weight edge in the Prim's algorithm

Heuristic

The chosen heuristic is the prim algorithm for the minimum spanning tree. The basic idea of the algorithm is to find the quickest way to connect each city. Note that this is not a path (it can be a path but not likely unless we have all the cities in a straight line). The algorithm will start with a random node, and the child node with the shortest path will be selected next. In the following steps, the algorithm will examine all the visited nodes and append the child node to a visited node that has the smallest path to it. In the traveling salesman problem, the cost of the last city to the start city should also be calculated. To comprise this condition in the lower bound calculation, the cost of the last connected city to its shortest neighbor will be added.

Stochastic Local Search - Simulated Annealing Algorithm

Algorithm Design

For the SLS algorithm, we will use the simulated annealing algorithm as our random restart wrapper, and we also assume that one city can be reached by any other city. As a type of SLS, the simulated annealing algorithm allows us to improve the initial solution. The costs of each solution will be evaluated and the best solution will be updated if there is one with a lower cost. Through each iteration, the acceptance of the solution with a higher cost is based on two criteria. One is that if the cost difference between the new solution and the current solution is negative, in this case, we keep the new solution and compare it with the best solution. Another case is that if the cost difference is positive, we record the "worse solution" based on a designed probability.

Besides, the randomization of the new solutions could be the position swaps between two cities from the current solution. The algorithm will stop when temperature <= terminal condition and our terminal condition is set to be 0.01.

Algorithm Assessment

Given that the number of cities is N, the runtime of the Simulated Annealing Algorithm is O(N), and the space complexity is also O(N).

Data Structure

- Initial solution: a path with its associated cost that is found by the Depth First Search
- Initial temperature: set to be big value, like 1000
- Cooling rate: a float value that ranges from 0.95 to 0.99
- Probability: sets to be exp(-cost/temperature) based on the Metropolis criterion (Metropolis, n.d.)
- Random _number: generates a random probability (0,1), and we compare it to the probability; the algorithm accepts a worst solution if it is smaller than the probability, rejects otherwise
- New solution: randomly selects two cities and swap their positions

References

Metropolis. Metropolis · Arcane Algorithm Archive. (n.d.). https://www.algorithm-archive.org/contents/metropolis/metropolis.html

Wikimedia Foundation. (2023, October 22). *Branch and bound*. Wikipedia. https://en.wikipedia.org/wiki/Branch_and_bound

Travelling salesman problem. (2023, November 18). Wikipedia. https://en.wikipedia.org/wiki/Travelling_salesman_problem