CS271P Project Report

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

Introduction

In this report, we will produce the solutions to the traveling salesman problem (TSP), and the solutions are the shortest path to visit each city once in the TSP. The report will be divided into two parts that are based on two well-known algorithms - Branch and Bound Depth First Search and Stochastic Local Search. The algorithms will be written in Python, and we assume the inputs are graphs that contain the distances to all cities. Additionally, solving TSP can lead to many meaningful applications in real life, such as transportation planning and goods logistics, or how Amazon can improve their delivery speed for the customers.

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.

Functions: BnB Depth First Search

def BnB_DFS(graph, start_city):

- This function is the core function of our algorithm, it is used to find the shortest possible path that each city can be visited in the graph once and return to the 'start_city'. The input "graph" represents a 2-D matrix, where 'graph[i][j]' is the weight of the edge between cities i and j. We design to use the "dfs" function recursively to explore all possible paths from the ‘start_city’, and it also keeps track of the current path and its total weight.

def calculate_bound(graph, path, current_city):

- This function calculates a lower bound on the shortest path that includes the current path plus the minimum additional weight required to complete all the visits.

def prim_mst_weight(graph, path):

- This function is designed to create the Prim’s algorithm that was discussed in the heuristic part. Specifically, it computes the weight of the Minimum Spanning Tree (MST) for the unvisited cities in the graph, which is a part of the lower bound calculation.

def min_edge_weight_to_unvisited(path, current_city, graph):

- This function finds the minimum weight edge that connects the ‘current_city’ to any unvisited city, contributing to the lower bound calculation. It iterates over all unvisited cities to find the minimum weight edge from the ‘current_city’.

Possible improvements

1. **Heuristic Guided Search**
    - Current Approach: The search follows a DFS pattern without any initial guidance.
    - Improvement: Incorporate heuristic information to guide the search more effectively. For example, starting exploring paths that seem promising based on heuristic estimates like nearest neighbor distances. This can help in reaching good solutions faster.
2. **Heuristic Initial Solutions**
    - Current Approach: Starts from scratch without any initial solution.
    - Improvement: Use a heuristic method (like Nearest Neighbor, Christofides, or Greedy algorithms) to find an initial feasible solution. This solution can provide an early upper bound, helping to prune the search space more effectively.
3. **Memory Optimization**
    - Current Approach: May consume significant memory for large graphs.
    - Improvement: Optimize memory usage by using more efficient data structures or by storing only necessary information in each node of the search tree.

4. **Parallel Processing**
   - Current Approach: The algorithm likely runs on a single thread or process.
   - Improvement: Implement parallel processing techniques. Since BnB DFS inherently explores different branches, these branches can be processed in parallel to speed up the computation, especially on multi-core systems.
5. **Dynamic Ordering of Cities**
   - Current Approach: Cities are explored in a static order.
   - Improvement: Dynamically determine the order in which to explore cities based on certain criteria, like the proximity of the next city or the lower bound of the path through that city. This could lead to faster discovery of promising paths.

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

Functions:

def generate_neighbor(solution):

- This function is designed to generate a new solution by swapping two random cities in the current solution.

def calculate_cost(graph, solution):

- The input graph is the 2-D matrix representing the distances of all cities, and the function calculates the total cost of the input solution based on the graph.

def accept_worse_solution(cost_difference, temperature):

- This function decides whether to accept a worse solution based on the cost difference and the current temperature. It calculates the acceptance probability using an exponential function of the cost difference and the temperature. Then, it can accept the worse solution with this probability.

def simulated_annealing(graph, initial_solution, initial_temperature, cooling_rate):

- This function implements the Simulated Annealing algorithm to find an approximate solution to the TSP. It Starts with an initial solution and temperature. Then iteratively generates neighboring solutions and decides whether to accept them, based on their cost and the current temperature. Finally ,it gradually reduces the temperature at each iteration.

def random_dfs_tsp(graph, start_city):

- This function generates an initial random solution for the TSP using Depth First Search. It starts from a ‘start_city’ and explores the graph in a DFS manner, randomly selecting the next city to visit. Then, If it reaches a dead end or completes a tour, it backtracks.

Possible improvements

1. **Enhanced Neighborhood Generation**
    - Current Approach: The neighborhood is generated by simply swapping two cities.
    - Improvement: Implement more sophisticated methods for generating neighbors. For instance, consider using a variety of moves such as swap, reverse, and insert. This can provide a richer set of neighboring solutions to explore.
2. **Multiple Start Points**
    - Current Approach: The algorithm starts with a single initial solution, which might limit its exploration capability.
    - Improvement: Run the Simulated Annealing multiple times with different initial solutions. This is also known as multi-start Simulated Annealing, which can explore various regions of the solution space, increasing the likelihood of finding a global optimum.
3. **Hybrid Approaches**
    - Current Approach: Solely relies on Simulated Annealing (SA) algorithm.
    - Improvement: Combine SA with other optimization algorithms like Genetic Algorithms (GA) or Ant Colony Optimization (ACO). For example, GA can be used to generate a diverse set of initial solutions for SA, or use ACO to refine solutions found by SA.
4. **Dynamic Cooling Schedule**
    - Current Approach: The cooling rate is constant, which might not be optimal under all conditions.
    - Improvement: Implement a dynamic cooling schedule that adjusts the cooling rate based on the progress of the algorithm. For example, a slower cooling rate initially (to explore more of the solution space) and then speeding up as the algorithm converges.

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