CS271P Project Milestone

Travelling Salesman Problem

Group 75
Xiaofei Teng
60042132

Abstract

In this project, we are going to solve the travelling salesman problem (TSP). According to the project requirement, we are supposed to solve the TSP using two different algorithms: the Branch-and-Bound Depth First Search (BnB DFS) algorithm, and the Stochastic Local Search (SLS) algorithm. In this report, I will demonstrate my design, including definition of data structures and functions, pseudocode, and expectation on time and space complexity, for both algorithms.

Introduction

To begin with, I will give a brief introduction to the problem to deal with. The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research [1].

Before I start to apply different algorithms, I want to redefine the problem regarding to this project. The input would be an integer n, which stands for the total number of locations, followed by a n*n diagonal matrix, which represents the distance between each pair of locations. For example, (1,2) of this matrix would be the distance between location 1 and location 2. If there is no path between two locations, it results in a very large number in the corresponding position of the matrix. As you might have noticed, the distance matrix being diagonal meaning that it is a symmetric travelling salesman problem (STSP) that we are dealing with. The expect output is a sequence of the locations that gives the minimum total distance, which is of length n. Because we always have to return to the starting city, in that case, the final optimal tour would be a loop on the location graph, and it does not really matter which city we start from. For simplicity, in part I of this project, we will assume that we start from city 1.

With all that in mind, I will implement 2 algorithms to solve the TSP: BnB DFS algorithm in part I and SLS algorithm in part II.

Part I: BnB DFS Algorithm

To implement the BnB DFS algorithm, the first thing to do would be to understand the state space that we search within. Although the TSP is given in the format of a graph, where vertices are locations and edges are distances between locations, this is not the graph that we apply the DFS algorithm. In fact, this graph should stay static during the whole search process. Instead, we search in a tree of “states”. Here, I will first define state as an object.

```
state{
    linkedlist<node> vis
    set<node> unvis
    curTotDis
}
```

In this definition, I use a link list structure to store the visited nodes because I want to preserve the order that I visited these nodes. This enables me to output the sequence easily if current state is a complete state. Also, I need to query for the current node, which I can achieve by return the last element of the link list, to decide my next move. On the other hand, there is no necessity to maintain order for the unvisited nodes, thus I store them in a set structure. Also, I record the current total distance of any given state, because this is the g function that we use to prune the search tree. Although it is possible to derive the current total distance from the link list of visited nodes, it could use up to O(n) time to compute. I use a structure called node in the definition of state object, so I am going to define node next.

```
node{
    int num
    int[] dis
}
```

The num attribute of node denotes which location this node represents, and the dis[] array store the distance from this node to all other nodes. Notice that using this representation, node.dis[node.num] will represent the distance to itself, thus will always be 0 for any node. All these information we may directly get from input.

Now I will define 3 functions for the state object to facilitate our search: isComplete(), h(), and f().

```
state.isComplete(){
    return (length of link list vis == n)
}

state.h(){
    curNode = vis.tail()
    min = infinity
    for (int i = 1 to n){
        if (i == curNode.num) continue
        if (dis[i] < min AND node[i] is not visited) min = dis[i]
    }
    return min
}

state.f(){
    return state.h()+state.curTotDis
}
```

The isComplete() function checks if current state is a complete state, and a state is complete if all the nodes are visited. The h() function returns the heuristic of the current state, which is defined as “the shortest distance of next possible move”. Because of the format of the input ensures that a number is given even if no path exists between two locations, a possible move is to visit any unvisited node. Because all the distances are non-negative, by defining the heuristic in this way, as long as the current state is not a complete state, which means there is at least 1 more move to be made, h(state) will always be less or equal to h*(state), which is the true cost to reach any complete state from current state. Thus, this heuristic is admissible. The f() function returns the lower bound of current state, which is equal to current total distance + heuristic. If the lower bound of current state is greater or equal to the upper bound, we will prune current state by not pushing its next moves into the DFS stack.

This heuristic design can be interpreted as “the optimal distance of next move”, and compute this heuristic will take O(n) time. In general, we can use “the optimal distance of next k moves”, where k is any positive integer that is less or equal to size(unvis) to be a heuristic. This heuristic would take O(n^k) time to compute and would always be admissible. Note that when k=size(unvis), this heuristic becomes the true heuristic, h*, of current state. For the sake of time complexity, I will use k=1 in my implementation.

Here, I provide the pseudocode for the complete algorithm:

```
initialstate = new state(vis = new Linkedlist<node>,
                         unvis = new Set<node>, curTotDis = 0)
U = infinity
bestTour = NULL
//assume we start from city 1
initial.vis.add(Nodes[1])
for (node that is not Nodes[1] in Nodes[]) initialstate.unvis.add(node)
BnBDFS = new Stack<state>
BnBDFS.push(initialstate)
while (BnBDFS is not empty){
    curState = BnBDFS.pop()
    //check for completeness
    if (curState.isComplete()){
        if (curState.curTotDis < U)
            bestTour = curState
            U = curState.curTotDis
```

```
                continue
            }
            //check for upper bound restriction
            If (curState.f() > U) continue
            //if current state is incomplete and we do not prune
            for (node in curState.unvis){
                newState = hardcopy(curState)
                newState.unvis.delete(node)
                newState.vis.push(node)
                newState.curTotDis += curState.vis.top().dis[node.num]
                BnBDFS.push(newState)
            }
        }
        return bestTour
```

Notice that bestTour is still a state object. We will have to pop the vis stack of bestTour to get the sequence we want.

Time complexity and space complexity of this algorithm would be both exponential, Because the number of possible states is exponential, and for each possible state, we have to create a new state object for convenience of the search. This algorithm is guaranteed to return an optimal solution.

Part II: SLS Algorithm

To implement the Stochastic Local Search algorithm to sole the travelling salesman problem, we will have to first define the state space to search within. As the result given by the algorithm should be in such a format that all nodes are visited exactly once, if we number the nodes to be 1…n, a valid tour some permutation of 1…n. Because in the original problem, we define a path between every pair of nodes (an extremely large number represents no path exists), every permutation of 1…n could possibly be the optimal solution. Thus, the state search space should be the full permutation space of all the nodes.

To performance SLS, we will generate a random start state within the state space, that is, a random permutation of 1…n. Then we define a trivial objective function to be the total length of such a tour, to evaluate each state. Here are the pseudocodes for these two parts:

```
GenerateRandomState(n){
    return random permutation of n
}

TotalDistance(state[], which is some permutation of 1…n){
    totdis = 0
    for i = 1 to n-1:
        totdis += distance(state[i], state[i+1])
    return totdis
}
```

Then, we define a valid move to be “swap the position of 2 locations within a tour” [2], and thus all possible states that we can reach within one single move is a neighborhood state. To put it in a more formal and mathematical way, for a state[], which is some permutation of 1…n, a valid neighborhood state is what we get from, that, for some 1<=i<j<=n, swap the i-th and j-th element of state[]. If the total distance of a neighboring state is less than what we have for now, we make a move into that neighboring state and record the new total distance to be the current best total distance. The algorithm ends and returns the current state and its total distance if no neighboring state has a better total distance than the current one. Because there are n! total possible states to search within, n! moves could be made at most. This is too much for us to handle with n large enough, thus we set an upper bound for the total number of steps to be made to be m<=n!. Also, to prevent us from stuck in some local maxima, the algorithm will use a random restart mechanism that runs the whole algorithm with different random initial state for k times. The pseudocode of the complete algorithm is here:

```
SLS(k){
    rerun = 0
    while (rerun < k){
        curState = GenerateRandomState(n)
        curBestTotDis = TotalDistance(curState)
        step = 0
        while (step < m){
            for (i = 1 to n-1){
                for (j = i+1 to n){
                    newState = state(swap i-th and j-th element)
                    newDis = TotalDistance(newState)
                    if (newDis>cueBestTotDis){
                        curState = newState
                        curBestTotDis = newDis
                        step++
                        continue to next step
                    }
                }
            }
            //if the for loop runs out, aka. no better neighboring state
            break
        }
        rerun++
        return curState, curBestTotDis
    }
}
```

This pseudocode will return k (tour, total distance) tuples as the results of k reruns. If we only want the best solution we get, we can only record the tuple that gives the best total distance across k reruns. Hyperparameters k and m are set manually.

For any given state, the total number of its neighboring states is (n-1)(n-2)/2, which is O(n^2). Within each search step, we have to calculate the total distance of the given state, this takes O(n) time. Max total step is set to be m, and rerun limit is set to be k. Thus, the total time complexity of this SLS algorithm is O(kmn^3). Because all that the algorithm has to memorize at any given time is the current state and the current best total distance, the space complexity of this SLS algorithm is O(n). However, this algorithm may not always guarantee an optimal solution if k and m are not well-selected.

Reference

[1]: Wikipedia: https://en.wikipedia.org/wiki/Travelling_salesman_problem

[2]: Project Instruction: Project.pdf: CS271P (uci.edu)