DAA Assignment No. : 8
SOLVING THE TRAVELLING SALESMAN PROBLEM USING THE BRANCH AND BOUND METHOD

Lovely Digra
IIT2017090

Abhishek Vishwakarma
IIT2017091

Anuj Raj
IIT2017092

***Abstract*—The goal of this paper is to optimize the delivery of packages to appointed addresses in some city. This problem is also known as the Travelling Salesman Problem and it is an NP hard problem. To achieve this goal, the concepts of a Hamilton path and cycle, as well as a Hamilton graph are used. While designing the problem time complexity for the algorithm is calculated which comes to be $O(n!)$ in worst case but is better than brute force method in all real life examples which is observed from the comparison plots.**

***Keywords*—** Travelling Salesman Problem, Branch and Bound Method, Hamilton path, Hamilton cycle, NP complete problem, NP hard problem

I. INTRODUCTION

The Travelling Salesman Problem is one of the most studied problems in mathematical optimization. The possibility to apply this problem to various human activities is what it makes one of the most recognizable mathematical problems without an ideal solution so far.

The formulation of the problem is simple. The travelling salesman needs to pass through several towns and return to the starting point(hometown) of his travel, making the shortest trip possible. By its nature, the TSP falls into the category of NP-complete problems; meaning there is no algorithm that would provide a solution for the problem in terms of polynomial time but if the solution appears, one could test it and conclude whether it is an optimal one.

The main objective of this problem is to find the shortest possible route that visits every city exactly once and returns to the starting point(hometown) where given a set of cities and distance between every pair of cities.This problem is to be solved by Branch and Bound approach.

Branch and bound method is based on the idea that the sets are divided into two disjoint subsets at every step of the process - branching. Furthermore, one subset contains the path between the two selected towns and the other subset does not. For each of these subsets the lower restriction (bound) is calculated for the duration or for travel expenses. Finally, the subset which exceeds the estimated lower bound is eliminated.

The procedure of branching is represented by a tree where the top is marked by the branching points of the set of solutions, and the edges mark the path between the two contiguous vertices in the graph which are used to model the problem and inside which the shortest Hamilton cycle is pursued.

II. ALGORITHM DESIGN AND EXPLANATION

While solving the Travelling Salesman Problem three approaches can be used. Either one can use Naive approach or one can solve the problem by using Dynamic Programming or Branch and Bound.

Approach 1: Naive Approach

Initially consider city 1 as the starting and ending point(hometown). As there are n number of cities given in the problem generate all (n-1)! possible permutations of cities. Basically generate all possible ways salesman can travel to deliver it's product to different appointed addresses in the city. Next step is to calculate cost of every permutation that is of every route differently and keep track of minimum cost route. Now, Return the permutation with minimum cost so that salesman only need to travel shortest path to deliver it's product to appointed addresses.

Approach 2: Optimized Solution

The branch and bound method provides a optimized solution for the given problem. In Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all.

Let S be some subset of solutions. Let L(S) = a lower bound on the cost of any solution belonging to S Let C = cost of the best solution found so far If C≤L(S), there is no need to explore S because it does not contain any better solution. If C>L(S), then we need to explore S because it may contain a better solution.

A Lower Bound for a TSP

Note that:
Cost of any tour

$$= \frac{1}{2}\sum_{v \in V} \text{(Sum of the costs ofthe two tour edges adjacent to } v\text{)}$$

Now:
The sum of the two tour edges adjacent to a given vertex v
$\geq$ sum of the two edges of least cost adjacent to v
Therefore:
Cost of any tour

$$\geq \frac{1}{2}\sum_{v \in V} \text{(Sum of the costs of the two least cost edges adjacent to } v\text{)}$$

Example: See Figure 1

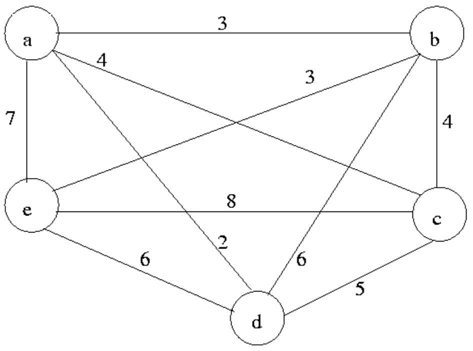

Fig. 1: Example of a complete graph with five vertices

Node	Least cost edges	Total cost
a	(a, d),(a, b)	5
b	(a, b),(b, e)	6
c	(c, b),(c, a)	8
d	(d, a),(d, c)	7
e	(e, b),(e, f)	9

Thus a lower bound on the cost of any tour
$= \frac{1}{2}$ (5 + 6 + 8 + 7 + 9) = 17.5

A solution Tree for a TSP instance:

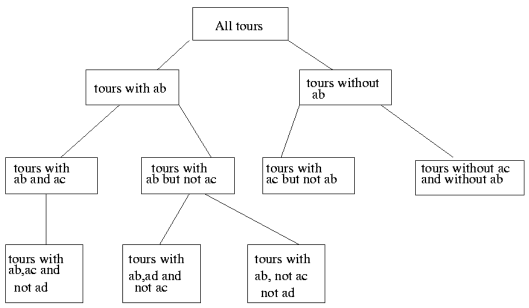
All tours
tours with ab
tours without ab
tours with ab and ac
tours with ab but not ac
tours with ac but not ab
tours without ac and without ab
tours with ab,ac and not ad
tours with ab,ad and not ac
tours with ab, not ac not ad

Fig. 2: A solution tree for a TSP instance

Now we have an idea about computation of lower bound. Let us see how to how to apply it state space search tree. We start enumerating all possible nodes

1 The Root Node: Without loss of generality, we assume we start at vertex 0 for which the lower bound has been calculated above.

Dealing with Level 2: The next level enumerates all possible vertices we can go to (keeping in mind that in any path a vertex has to occur only once) which are, 1, 2, 3 n (Note that the graph is complete). Consider we are calculating for vertex 1, Since we moved from 0 to 1, our tour has now included the edge 0-1. This allows us to make necessary changes in the lower bound of the root.
Lower Bound for vertex 1 = Old lower bound - ((minimum edge cost of 0 + minimum edge cost of 1) / 2) + (edge cost 0-1)
Dealing with other levels: As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, n. Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and alter the new lower bound for this node.

Lower bound(2) = Old lower bound - ((second minimum edge cost of 1 + minimum edge cost of 2)/2) + edge cost 1-2)

Note: The only change in the formula is that this time we have included second minimum edge cost for 1, because the minimum edge cost has already been subtracted in previous level.

When we branch, after making what inferences we can, we compute lower bounds for both children. If the lower bound for a child is as high or higher than the lowest cost found so far, we can 'prune' that child and need not consider or construct its descendants.

If neither child can be pruned, we shall, as a heuristic, consider first the child with the smaller lower bound. After considering one child, we must consider again whether its sibling can be pruned, since a new best solution may have been found.

Further optimization : Parallel Programming model
The proposed algorithm is the sequential approach for solving the recursion tree instance shown in fig 2 and 3.In a given level First the left sub trees are evaluated then followed by the right sub trees.There can be a case where the answer if found in the rightmost sub tree. If each sub trees generated from a given level is executed simultaneously then the algorithm will jump to a conclusion much earlier. This idea can be realised by applying multi-threading to the proposed algorithm. The idea is more profound in a parallel computing environment with multi core processors.

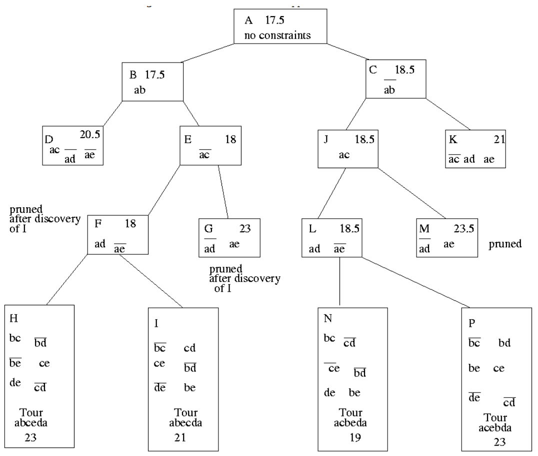

Fig. 3: Branch and bound applied to a TSP instance

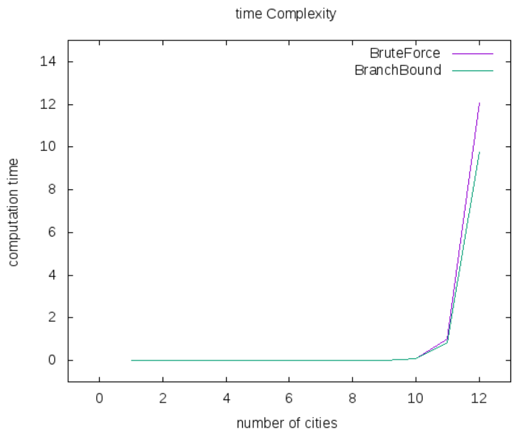

Fig. 4: comparision of computational time a. brute force b. Branch and Bound

III. EXPERIMENTAL STUDY

For the experimental analysis of this algorithm, number of cities(vertices) is varied, to see how the execution time varies with increasing size. The weight that is the cost for each pair of cities are taken from the standard random generator in the range $1 < W < 100$. Where W is the cost of traveling. The graph of length vs execution time is plotted using .dat file generated from the source code. see Fig 4.

IV. TIME COMPLEXITY AND DISCUSSION

In the algorithm design section, it is discussed that the brute force method is optimized by enforcing the bound on each recursive call. If the bound condition is not achieved, further exploration of the branch is not required. This is the case when a child node is pruned. Thus in average cases the algorithm is expected to run faster than the brute force. But theoretically, in worst case time complexity of the algorithm will be same as that of brute-force algorithm that is n-1 factorial. Where n is the the total number of cities to be visited.

Algorithm 1 Travelling Salesman

```
procedure TSP_TRAVERSE($c\_p, c\_b, c\_w, level, n$)
    if $level\ == \ n$ then
        if $arr[c\_p[level-1]][c\_p[0]]! = 0$ then
            $result \leftarrow c\_w + arr[c\_p[level-1]][0]$
            if $result < final\_weight$ then
                $final\_weight \leftarrow result$
                $copy(n, c\_p)$
    for $I$ from $0 \rightarrow n$ do
        if $arr[c\_p[level-1]][I]! = 0 and\ vis[I] == 0$ then
            $temp \leftarrow c\_b$
            $weight \leftarrow arr[c\_p[level-1]][I]$
            $c\_we \leftarrow c\_w + weight$
            if $level == 1$ then
                $c\_b \leftarrow c\_b - (first\_min(c\_p[level-1], n) + first\_min(i, n)/2)$
            else
                $c\_b \leftarrow c\_b - (second\_min(c\_p[level-1], n) + first\_min(i, n)/2)$
            if $c\_b + weight < final\_weight$ then
                $c\_p[level] \leftarrow i$
                $vis[i] \leftarrow 1$
                $TSP\_traverse(c\_p, c\_b, c\_w, level+1, n)$
            $c\_w \leftarrow c\_w - arr[c\_p[level-1]][I]$
            $c\_b \leftarrow temp$
            for $I$ from $0 \rightarrow n$ do
                $vis[i] \leftarrow 0$
            for $I$ from $0 \rightarrow level-1$ do
                $vis[c\_p[i]] \leftarrow true$
procedure TSP($int\ n$)
    $c\_b \leftarrow 0$
    $c\_w \leftarrow 0$
    $c\_p \leftarrow Array\ of\ size\ n$
    for $I$ from $0 \rightarrow n$ do
        $c\_b \leftarrow c\_b + first\_min(i, n) + second\_min(i, n)$
    if $c\_b\%2 == 0$ then
        $c\_b \leftarrow c\_b/2$
    else
        $c\_b \leftarrow c\_b/2 + 1$
    $c\_p[0] \leftarrow 0$
    $vis[0] \leftarrow 1$
    $TSP\_traverse(c\_p, c\_b, c\_w, 1, n)$
procedure MAIN
    $Taking\ the\ inputs$
    $Adding\ edges$
    $Printing\ minimum\ cost\ and\ path\ taken$
```

The lower bound after each recursion is updated using the min operation on a given node which finds the minimum cost edge terminating at the given node. If we pre compute this value for each node(city) in the problem and store it for later then this min operation for each recursion is computed in $O(1)$ time. There is a trade-off between the space and time complexity for min operation.

Finally printing the path or the solution takes linear time $O(n)$.

Space Complexity: two arrays of size n is needed for the proposed algorithm thus the complexity has the order $O(n)$. Stack overhead of the recursion is $O(n!)$.

V. CONCLUSION

We designed an algorithm to minimize the travel cost of a salesman who wants to visit all the cities exactly once. We also analyzed the algorithm that resulted in general time complexity less complexity than with brute force with worst case upper bound $O(n!)$ and plotted the graph of number of cities vs time elapsed and found that the the proposed algorithm is faster than the brute force algorithm which consider all possible cases without any bound condition. The bound proposed in the algorithm is a very good bound for faster solution. It can be verified that the algorithm is NP-complete.

REFERENCES

[1] https://www.geeksforgeeks.org/heap-data-structure/
[2] https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/
[3] https://www.geeksforgeeks.org/heap-sort/