Traveling Salesman Problem

Seungbae Son
43483449
seungbas@uci.edu

Ju Young Cha
61163075
juyc1@uci.edu

Shawn Lo
94449743
losv@uci.edu

Abstract

The Traveling Salesman Problem, deals with creating the efficient path that a salesman would take while traveling between cities. The solution to any given TSP would be the shortest way to visit a finite number of cities, visiting each city only once, and then returning to the starting point. For the most part, the solving of a TSP is no longer executed for the intention its name indicates. Instead, it is a foundation for studying general methods that are applied to a wide range of optimization problems. We are going to find a solution by using two algorithms, Branch and Bound Depth First Search Algorithm and Stochastic Local Search Algorithm and take an experiment to evaluate the result in the view of performance, accuracy, and efficiency with an appropriate measure factor.

1. Traveling Salesman Problem

We will be given a list of cities and the distances between each pair of cities, and be asked “what is the shortest possible route that visits each city exactly once and returns to the origin city?” This is what the Traveling Salesman Problem is and we need to figure out how to design the optimal route a salesman would follow when moving between locations. TSP would have a solution that involved traveling back to the beginning point using the shortest route possible while stopping in each city just once. In other words, TSP assumes that the distances between N cities are specified in a cost matrix that specifies the non-negative cost between any pair of cities. Each city must be visited exactly once and no cities can be skipped. The goal is to find the sequence of cities that starts and ends with city 1 such that the overall cost of the tour is minimized.

2. Approach

For the approach of TSP, there will be various ways to solve it but in this project we will mainly focus on the following two ways 1) Branch and Bound DFS and 2)Stochastic Local Search and we used Python to implement this.

2-1. Algorithm and Implementation

1) Branch and Bound

CONCEPTS A branch-and-bound algorithm enumerates potential solutions in a methodical manner using state space search; the set of candidates is conceptualized as a rooted tree with the full set at the root. This tree's branches, which stand for subsets of the solution set, are searched by the algorithm. Before listing a branch's potential solutions, it is tested against upper and lower estimates of the optimal solution's bounds (in this case, the lower bound) and eliminated if it is unable to generate a better solution than the one discovered thus far by the algorithm. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in the worst case. Since this algorithm keeps trying to prune the branch rather than just going the full search, the Branch and Bound Algorithm is known to solve these problems relatively quickly.

IMPLEMENTATION The algorithm mainly depends on efficient estimation of the lower bounds of branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search. For this matter, both getting a minimum bound and search path which can be expressed as a data structure is important. We've tried to implement an algorithm to simulate which data structure can improve the performance and also implement a recursive way which may take a way more to see if our main BnB algorithm can reach an optimal solution.

First, we define an object called Node which includes level, path, bound and cost data, and it also can compare its bound with others and also another function which can return these values. The below algorithm uses priority queue as a data structure which has shown better performance than stack. Our idea to compute bounds for this problem, called heuristic function, is to get the sum of the cost of two edges adjacent to every vertex u and in the tour where u is in state space and sum their costs. The overall sum for all vertices would be twice the cost of tour T because we have considered every edge twice. For example, when we are calculating for vertex 1, since we moved from 0 to 1, our tour has included the edge 0-1. This allows us to make necessary changes in the lower bound of the root. To include edge 0-1, we can add the cost of it and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. The edge subtracted can't be smaller than this. Then, if we move on to the next level, we can enumerate possible vertices again.

Algorithm 1 Branch and Bound Algorithm using Priority Queue

```
# Define Node class with "level", "path", "bound" as variables to express a state of certain vertex
# Used data structure with a PriorityQueue to express this state space
```

```
function BranchAndBound_TSP(matrix)
        # Initialize variables
        n ← length of input matrix, opt_cost ← infinity(default),
        final_path ← none, curr ← ptimal_path ← Node(level=1, path=[0])
        # Calculate initial bound
       for each i in range(n) do
            curr.bound += (firstMin(matrix,i) + (secondMin(matrix,i)
       Let curr.bound as ½*curr.bound

        Curr.path[0] ← 0
        PQ ← Initialized Priority Queue and put node in PQ with bound and k(for tie breaking)

        while not PriorityQueue.IsEmpty() // Do iteration during PQ has a node in it
              curr ← pick a node from PQ
               level ← curr.level
               visited ←  false * n // set visited list
               for j in range(level) do
                 if curr.path is not -1 then
                      let visited current path ← true

      #When node contagion a full path, it checks its cost
      if level == n then
         if res_cost < opt_cost then
            final path ← curr.path and opt_cost ← res_cost

       for i in range(n) do
          k +=1 // k is a parameter for tie breaking
             if matrix[curr.path[level-1][i] != 0 and not visited) do
               temp ← curr
                 curr ← Node(curr.values)
               curr.cost += matrix[curr.path[level-1][i]

               #set current bound value
               if level is 1 set
                 curr.bound -= ((FirstMin(matrix, curr.path[level-1]) + firstMin(matrix,i)) / 2)
               else
                 curr.bound -= ((SecondMin(matrix, curr.path[level-1]) + firstMin(matrix,i)) / 2)

               #compare current cost to optimal cost
               if curr.bound + curr.cost < opt_cost do
                    curr.path[level] ← i, visited[i] ← true, curr_level +=1
                  PQ.put((curr.bound+curr.cost, k, curr))
               else delete curr
                 curr ← temp
        return optimal_path, optimal_cost, count
end function
```

2) Stochastic Local Search Approach

CONCEPTS A stochastic local search algorithm explores neighborhoods of solutions to find the best solution to a combinatorial problem that would be infeasible to search completely for an optimal solution. A stochastic algorithm will probabilistically explore solutions that are worse to avoid local minima, but will tend to improve more than simple local search algorithms overall as they run. Our algorithm will start with an initial greedy tour, then simulated annealing with tabu search to improve the tour, and finally use a traditional local search to ensure that the final produced tour is in a local minima.

Our stochastic search algorithm needs an initial path to iteratively improve, and while our algorithm could use any arbitrary tour, we have chosen to use a nearest neighbor algorithm to generate the initial tour. Nearest neighbor is a greedy algorithm that simply starts at node 0 and picks the shortest edge leading out of every successive node to generate a valid tour. Compared to a completely random initial tour, the greedy algorithm will produce initial tours that are much more fit to begin with.

As an additional step, tabu search can also be used to explore various initial conditions (instead of random restart) by excluding edges that were already in a previous initial tour. Additional runs of the algorithm can be further done on these different initial conditions in an attempt to not run into the same local minima. This would be an additional wrapper around the entire algorithm that could find a lower cost tour, but would involve running the algorithm multiple times, which is expensive.

The initial tour is further improved using a simulated annealing algorithm. Simulating annealing improves a tour by randomly picking moves, accepting ones that make the tour shorter, and occasionally accepting worse solutions depending on how many iterations the algorithm has gone through. Randomly picking moves is usually described as "generating neighbors" to a tour, and can be done in a variety of ways, but we picked a simple method that picks two vertices and swaps their positions in the tour. Another method we tried to generate neighbors is to pick three vertices and do a three-Opt swap, but it was not fully implemented. Occasionally accepting worse solutions to escape local minima is not as simple as setting a constant probability: as the tour improves, making a suboptimal move is less likely to be correct, and depending on how costly the swap is, we also assign it a lower probability. This is done through this function $e^{-costDelta/temp}$ (Allanah, Travling Salesman…) this function describes a curve that is bounded by (0,1) with (costDelta/temp) as a variable. Temperature starts high and gives us a high probability to accept solutions that increase the cost by a high amount, and as temperature decreases costly neighbors are discarded with higher probability.

As an additional step in the stochastic search, we use a tabu list to ensure that edges that were recently changed are not reverted too soon, as that risks falling back into a local minima and increasing the search time. In any fully connected graph there are n(n-1)/2 edges and the size of the tabu list must not be too large so as to include every edge, and cause the algorithm to "seize up". Since our neighbors are generated by picking two vertices and swapping them, we save the pair of vertices in a queue that is the size of ten percent of all the edges and prevent those two vertices being selected together again until they leave the queue.

Before we are finished with any tour, we decided to use a local search to refine the tour into a local minima just in case the stochastic search resulted in a tour that could be easily improved. We use a 2-Opt algorithm which checks every pair of vertices, swaps them, reversing the section in between to see if there is improvement, and then stops when no further improvement can be made. This heuristic improves our final tour cost by a small amount.

IMPLEMENTATION

Algorithm 1 *Tabu Initial Conditions Wrapper*

```
function greedyTabuSearch(graph)
      while(not tired of doing it)
            initialTour ← null
            u ← startingNode
            while(initialTour not contains every vertex)
                  v ← min(dist(u,v)) for all v
                        if (u,v) not in TabuEdges then
                              edge ← (u,v)
                              if (degree(tour,u)<2 and degree(tour,v)<2)
                                    insert edge into initialTour
                                    u ← v
                                    push edge into TabuEdgesQueue
                                    insert edge into TabuEdges
                                    if TabuEdgesQueue is full then
                                       pop oldest_edge from TabuEdgesQueue
                                       remove oldest_edge from TabuEdges
            result ← TabuSearch(initial_state,max_iterations)
            if(cost(result)<cost(best_found) then
                  best_found ← result
      return best_found
```

Algorithm 2 *Generate neighbor and get cost*

```
function generateNeighbor(tour,i,j)
      swap(path[i],path[j])
end function

function getCostDifferenceOfNeighbor(tour,i,j):
      difference ← -cost(edge(i-1,i))-cost(edge(i,i-1))
                        -cost(edge(j-1,j)) - cost(edge(j,j+1))
                        +cost(edge(j-1,i)) + cost(edge(i,j+1))
                        +cost(edge(i-1,j)) +cost(edge(j,i+1))
      if i==j+1 or i==j-1
            difference ← difference+2*cost(edge(i,j))
      return difference
end function
```

Algorithm 3 Simulated Annealing with Tabu Search

```
function simulatedAnnealingWithTabuSearch(initialPath,graph)
      initialTemp ← 100*(cost(initialPath)/length(initialPath))
      n ← graph.numNodes
      tempCoefficient ← 10^-(graph.numNodes/5)
      tabu_edges←queue
```

```
      max_tabu_edges←(n*(n-1)/2)/10
      temperature ← initialTemp
      tour←initialPath
      while temperature>0.000001
            randomly pick u,v from tour
            if edge(u,v) is in tabu_edges
                  continue
            cost_delta=getCostDifferenceOfNeighbor(path,i,j)
            if cost_delta<0 or random[0.0,1.0]<e^(-cost_delta/temperature)
                  tabu_edges.push(edge(u,v))
                  if len(tabu_edges) > max_tabu_edges
                        tabu_edges.pop
                  tour ← generateNeighbor(u,v)
            temperature ← temperature*tempCoefficient
      end while
      return tour
end function
```

3. Experiment

We've tried to experiment on the following aspects: performance, accuracy, efficiency. For the performance, what we've mainly tried to work on is finding the best data structure for the BnB algorithm and was able to find priority queue performs a way better than stack. Elements in the priority queue are sorted by the estimated optimal total cost(current cost + estimated cost to the goal) to improve the performance of our algorithm. SLS was run for the same metrics using the same mean and standard deviation

3-1. Performance (Run Time)

We tried to measure the runtime for each algorithm with a different input size and did a simulation(for n=5,7,10 tried 100 times each and for n=15, 20, tried 10 times). Since the performance between BnB and SLS algorithms differ from each other so we did experiment separately with each capacity.

On table 1.1, you can see the BnB algorithm result of running time depending on the input size. Distinguishing result of this experiment is that we can see the big difference in running time with different data structures. Even though the heuristic is the same, running time can differ by whether to take stack or priority queue. Run Time goes up exponentially when input gets larger and also when input gets larger we can see the difference of run time between algorithms more clearly. For example, when input size is only 5 or 7 there is no difference between the algorithms. However, when it gets to n=15, the difference between performance gets bigger. Also, with the same number of nodes, we were able to see the difference between case1 and case2. ([case 1] μ = 100, σ = 5 [case2] μ = 50, σ = 5). With the table, we can see that when μ gets smaller with the same value of σ, it has a tendency to take more.

In the aspect of performance, we further tried to find the path when input size n=25 (μ = 50, σ = 5) and did iteration to measure how it works in BnB using priority queue. It may take much

longer if it is in a recursive way but by using the BnB PQ algorithm it only took 154 seconds(average).

Table 1.1. Running Time on BnB (s)

([Case 1] μ = 100, σ = 5 [Case2] μ = 50, σ = 5

※ Number of Trials(T) = 100 (n=5,7,10) / T = 10(n=15)

n	BnB Recursive		BnB DFS w/ Stack		BnB DFS w/ PQ	
	case1	case2	case1	case2	case1	case2
5	0.0002	0.0002	0.0002	0.0002	0.0003	0.0003
7	0.0011	0.0012	0.0015	0.0017	0.0016	0.0019
10	0.0264	0.0296	0.0285	0.0304	0.0289	0.0307
15	2.1061	4.9726	1.9167	9.2273	0.9155	2.9737

For the SLS algorithm, we've also measured the runtime with the number of nodes n (n=5,7,10,15,20,25) and the following is the result for this. This algorithm shows the performance doesn't really depending on the μ, σ values and it increases exponentially

Table 1.2. Running Time on SLS

([Case 1] μ = 100, σ = 20 [Case2] μ = 50, σ = 5

※ Number of Trials(T) = 20 (n=5,7,10,20,25)

n	SLS	
	case1	case2
5	0.0040	0.0040
7	0.0010	0.0010
10	0.0418	0.0391
15	0.4130	0.3848
20	4.0950	4.0953
25	41.5286	41.5979

3-2. Accuracy (Optimal)

This is why we've also tried to implement a recursive way to check if our algorithms are optimal. Theoretically, we know that Stochastic Local Algorithm doesn't guarantee to get the optimal answer and get to local minima instead. However, we needed to figure out if the Branch and Bound Algorithm can find the optimal result. We've tested with the same input and see if the BnB Algorithm found the same path with recursive and figured out it gets to the same. We tried the same way and could see that SLS doesn't always get to the optimal path.

Table 2.1 Result in aspect of Accuracy

Recursive	BnB(Stack)	BnB(PQ)	SLS
optimal	optimal	optimal	not always optimal

Table 2.2 Number of Expanded Nodes (μ = 50, σ = 5)

Input matrix 7x7 (μ = 100, σ = 5) =

```
[[  0.00 102.61  94.85  92.01  97.16 103.52 111.12]
 [102.61   0.00  99.38  91.62  99.70  95.35  97.06]
 [ 94.85  99.38   0.00 102.79 103.01  98.57 101.58]
 [ 92.01  91.62 102.79   0.00 103.65 110.84  96.20 ]
 [ 97.16  99.70 103.01 103.65   0.00  99.28 106.50 ]
 [103.52  95.35  98.57 110.84  99.28   0.00  96.29]
 [111.12  97.06 101.58  96.20 106.50  96.29   0.00]]
```

➢ **Test Result**

① **Path with recursive method**
[0, 3, 1, 6, 5, 4, 2, 0]

② **Path with BnB PQ**
[0, 3, 1, 6, 5, 4, 2, 0]

⇒ we were able to see if the algorithm finds the optimal path

When reviewing the aspect of optimal, SLS also seems to run relatively well. As you can see the below table 2.3, the gap for SLS and optimal solution was within 2% of difference. This seems pretty surprising that SLS gets relatively well compared to high performance. (For time limitation, we skipped to get optimal path in large input size)

Table 2.3 Comparison of SLS and optimal path

([Case 1] μ = 100, σ = 20 [Case2] μ = 50, σ = 5 ※ Number of Trials(T) = 20

n	SLS		Optimal		Error rate	
	case1	case2	case1	case2	case1	case2
5	445	239	442	237	0.68	0.84
7	601	321	590	321	1.86	0.00
10	796	454	789	448	0.89	1.34
20	1,433	851	N/A			
25	1,711	1,059				

3-3. Efficiency (# of node expanded or # of iteration)

For the BnB algorithm, we tried to calculate how many nodes are expanded to find an optimal path and figured out that it definitely explores less(which means it runs more efficiently) when using Priority Queue as a data structure.

We put our experiment result in Table 2.1 and this shows the number of expanded nodes between algorithms. When nodes are not big enough(n=5,7), it is not easy to see if there is a difference between the algorithms but when input gets larger, we can say that there is a difference to the number of explored nodes. For example, with n=10, when we use a recursive algorithm, it expands the node up to 1,345 whereas it can be 1,112 nodes with the BnB Priority Queue algorithm. This is approximately a 20% reduction and when input size gets n = 20 we could see

the % of node expansion is highly reduced(-76%) when using the PQ algorithm compared to the recursive method.

Table 3.1 Number of Expanded Nodes (μ = 50, σ = 5)

※ Number of Trials(T) = 100 (n=10) / T = 10(n=15) / T = 5(n=20)

n	BnB Recursive (A)	Priority Queue (B)	% of expansion (B-A)/A
10	1,345	1,112	-17%
15	84,892	46,360	-45%
20	5,103,109	1,208,384	-76%

We've also tried to find out if the algorithm shows any feature when input has different mean and standard deviation and get the following result. "Case 1" is the case with mean = 100, standard deviation = 5 and "case 2" is mean = 50 and standard deviation = 5 and for this cases, it wasn't easy to see the difference when nodes are with 5,7 but when n=10 or 15 there is a tendency that case 2 does less exploration for each algorithm. The difference of expanded nodes gets bigger when n gets bigger. There is approximately a 10% gap when n=10 but it increases to 50% when n=15, and I wanted to figure out why this algorithm shows this with the condition but it wasn't easy to get to the reason.

Table 3.2 Number of Expanded Nodes with different μ and σ

([Case 1] μ = 100, σ = 5 [Case2] μ = 50, σ = 5

※ Number of Trials(T) = 100 (n=5,7,10) / T = 10(n=15)

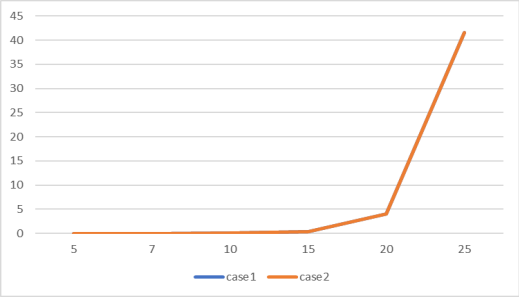
n	BnB Recursive		BnB DFS w/ Stack		BnB DFS w/ PQ	
	case1	case2	case1	case2	case1	case2
5	16	16	20	20	17	19
7	94	101	109	110	96	94
10	1,514	1,345	1,673	1,461	1,393	1,112
15	149,710	84,892	208,028	103,077	99,444	46,360

For the SLS algorithm, we can see the following result for the number of iterations with each case. The number of iterations is not really dependent on the value of μ or σ, the result is related to the input size. We can see it increases kind of exponentially. It gets about 10 times bigger when input size gets 5 bigger than before.

Table 3.3 Number of Iteration with different μ and σ

[Case 1] μ = 100, σ = 20 [Case2] μ = 50, σ = 5 ※ Number of Trials(T) = 20

n	SLS	
	case1	case2
5	400	407
7	831	805
10	2,884	2,858
15	27,491	26,843
20	269,263	263,525



LIMITATION There was also a limitation in some conditions for the algorithm. When σ is too low, like under 1, we found that sometimes BnB may fall the wrong way and tried to solve this but it wasn't that easy to do. Also, for SLS, we could set the parameter by experimenting with a human sense which leads to a better solution however it was hard to be explained mathematically. We expect that we could find why when we've studied this more.

4. Assessment

The branch and bound algorithm find a minimal path to reach the optimal solution for a given problem. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in the worst case(when we fail pruning). To reduce time complexity, we can add bounding functions but it requires more computation of bounds so it gets more complicated. However, when there is a huge difference between estimated cost to actual cost it should take many nodes to expand so it may take long. To improve this problem, we took priority queue and this led to reduction of the number of expanded nodes but it still seems to take a lot when the input size gets bigger than 25. Thus, we can say that BnB is not really appropriate with the problem of having a big input size and rather to take SLS is better.

Nearest neighbor is a greedy algorithm that takes $O(n^2)$ time because each node needs to compare the cost of every other node to find the lowest cost edge leading away from the node. Fitness of any tour can be determined by following the tour and adding the distances between the vertices in the order that they are visited; this takes O(n) time. Because 2-Opt only breaks 2 edges in a given instance, it is possible to calculate the difference between two neighbors in constant time using the cost of the four involved edges, and we can add this if calculating cost starts to dominate runtime. Additional memory is required to store all the tabu initial conditions as well as all the tabu tours. The number of edges is n(n-1)/2 and our algorithm saves 10% of those edges, so it shouldn't use more than $O(n^2)$ memory. The runtime of simulated annealing is controlled by an initial temperature variable, a temperature coefficient, and the cost of generating neighbors. The initial temperature is set by multiplying the average cost of an edge in the tour by 100, this is so that the cost of the edges is related to the temperature which makes sense when the probability is calculated for accepting a worse solution. It is reduced every iteration by $10^{-(n/5)}$, which makes it roughly run ten times longer for every five nodes added with the same initial temperature, but we

also capped it at 0.000001 so that it doesn't take too long to run. Even though it can be stopped at any time for an approximate solution, the simulated annealing has $O(10^n)$ iterations.

As can be seen at the result of the experiment, we have tried to experiment how the implemented result looks like and tried to improve by changing the parameter or a data structure. There might be a better heuristic on this problem and that may be more competitive but the whole process to do this project was also meaningful to us because this made us to bring the problem in a "artificial" world in the textbook to the real-life and also experienced a designing the problem, implementing and improving by experimentation.

References

1) "Branch and Bound." *Wikipedia*, Wikimedia Foundation, 23 Aug. 2022, https://en.wikipedia.org/wiki/Branch_and_bound.

2) "Traveling Salesman Problem Using Branch and Bound." *GeeksforGeeks*, 31 Oct. 2022, https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/.

3) "2-Opt." *Wikipedia*, Wikimedia Foundation, 30 Sept. 2022, https://en.wikipedia.org/wiki/2-opt.

4) *"3-Opt." Wikipedia,* Wikimedia Foundation, 15 Sept. 2021*, https://en.wikipedia.org/wiki/3-opt.*

5) *Around the World in 90.414 Kilometers - Towardsdatascience.com*. https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03b8552.

6) Ninad Thakoor, Venkat Devarajan, *2009 IEEE 12th International Conference on Computer Vision*, Computation complexity of branch-and-bound model selection

7) Etykiety. *3-Opt Move*, 6 Mar. 2017, http://tsp-basics.blogspot.com/2017/03/3-opt-move.html

8) Christian Nilsson, *Link¨oping University*, Heuristics for the Traveling Salesman Problem

9) Amanur Rahman Saiyed, *Indiana State University*, 11 April 2012, The Traveling Salesman problem\

10) *Traveling salesman problem using branch and bound*. GeeksforGeeks. (2022, October 31). https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/

Appendix

We've tried to work on a competition problem only with the SLS algorithm. (BnB doesn't seem to handle input size such 1000.(time-out))

For n=1000
Runtime : 41 seconds
Tour length : ~27000
Number of iteration : 2.65 million+

Please see the attached csv file for the result.