Travelling Salesman Problem

CS271P

Imlementation & Analysis using:

1. Branch and Bound Depth First Search.
2. Stochastic Local Search.

Team 14

Team Members:

1. Megha Sri Satya Sai Devineni, mdevinen, 56969644.
2. Nyamagoudar Namrata, nyamagon, 30921588.

Abstract: This project focuses on solving the Traveling Salesman Problem (TSP) through two different approaches: Branch and Bound with Depth-First Search (DFS) and Stochastic Local Search. The Branch and Bound DFS technique aims for optimal solutions by intelligently exploring potential paths, while the Stochastic Local Search employs randomness to efficiently navigate state spaces. By comparing these methods, the project seeks to provide insights into their effectiveness, trade-offs, and applicability in solving the TSP problem.

Travelling Salesman Problem:

The Traveling Salesman Problem (TSP) is a classic optimization challenge where a salesman must visit a set of cities, minimizing the total distance traveled, and returning to the starting city. For instance, if cities A, B, and C are to be visited, the TSP seeks the shortest path, like A -> B -> C -> A.

Solving the Travelling Salesman problem: As the above problem statement states inorder to solve the TSP, we have to explore all the nodes/cities exactly once before reaching the source node. The chosen path should give us the minimum possible distance. There are various possible approaches to solve TSP, like brute force, dynamic programming, greedy algorithms, etc.. The ones that we use in this project are:

1. Branch and bound depth first search: BnB divides the problem into subproblems, systematically exploring and pruning the search space to find an optimal solution. Depth first search approach is used to explore the nodes depth wise. Minimum distance of a path is stored and later used to prune the lower bounds whenever the current distance exceeds lowest distance.
2. Stochastic Local Search: Stochastic Local Search (SLS) is a general optimization technique used to find approximate solutions to difficult combinatorial optimization problems. It falls under the broader category of metaheuristic algorithms, which are high-level strategies for exploring and exploiting solution spaces. SLS methods are particularly useful when the search space is large and complex, making it difficult to find an exact solution in a reasonable amount of time.

Branch and Bound Depth First Search Algorithm

Explanation:

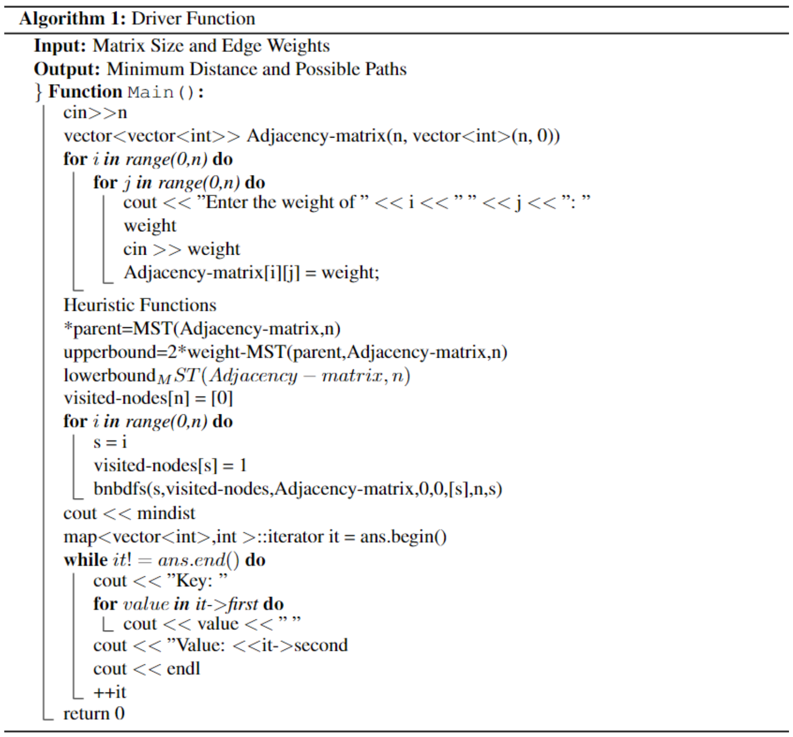
We are following a recursive approach to explore all possible permutations and keep track of the minimum weight along the way. The bnbdfs function takes the adjacency matrix as input along with the source, a tracker for visited nodes, and a heuristically improved list.

Visited nodes' tracker: This list is updated with the nodes that are visited, which is later appended to the ans list so that we can display the path followed to achieve the minimum distance.

Working: We start at the first node called source and mark it as visited, then iterate through a loop that allows us to pick the child node. Once we pick the child node, we mark it as visited and recursively expand the tree. While doing all of this, we append the current distance with each edge's weight and increment the counter that denotes the number of nodes visited so far. This process continues until we exhaust the entire search space. At the end, when the counter matches the total number of nodes, we compare the distance so far with the minimum distance and take the minimum value. Both the minimum distance and the corresponding paths are stored in the tracker list data structure. Later, the process is returned back to the previous step but this time we explore another node and it's corresponding depth.

Branch & Bound part: As we explore the first complete path, we obtain its total distance, since it would be less than positive infinity, and store it in the minimum distance variable. Now, whenever we explore other paths, we use the minimum distance variable to compare it with the current distance. If the latter exceeds the minimum distance, we abruptly stop the recursion and return to the previous state, giving us a chance to exhaust another path.

Pseudo code of the driver function:



```
Algorithm 1: Driver Function
  Input: Matrix Size and Edge Weights
  Output: Minimum Distance and Possible Paths
  } Function Main():
      cin>>n
      vector<vector<int>> Adjacency-matrix(n, vector<int>(n, 0))
      for i in range(0,n) do
          for j in range(0,n) do
              cout << "Enter the weight of " << i << " " << j << ": "
              weight
              cin >> weight
              Adjacency-matrix[i][j] = weight;
      Heuristic Functions
      *parent=MST(Adjacency-matrix,n)
      upperbound=2*weight-MST(parent,Adjacency-matrix,n)
      lowerbound$_M ST(Adjacency - matrix, n)$
      visited-nodes[n] = [0]
      for i in range(0,n) do
          s = i
          visited-nodes[s] = 1
          bnbdfs(s,visited-nodes,Adjacency-matrix,0,0,[s],n,s)
      cout << mindist
      map<vector<int>,int >::iterator it = ans.begin()
      while $it! = ans.end()$ do
          cout << "Key: "
          for value in it->first do
              cout << value << " "
          cout << "Value: <<it->second
          cout << endl
          ++it
      return 0
```

Fig 1

Pseudo code of the BNBDFS function:

```
Algorithm 1: BNBDFS
  Input: Path, Adjacency Matrix
  Output: Adjacency matrices after swapping 2 edges
  map<vector<int>, int> ans
  mindist=1e9
  } Function bnbdfs (indicator, visited − nodes[], < vector < vector < int >>
   Adjacency − matrix, current − distance, counter, vector < int > paths, n, source − node):
    visited-nodes[indicator]=1
    counter++
    if counter == n then
        if current-distance+Adjacency-matrix[indicator][source-node] <
          mindist and current-distance+Adjacency-matrix[indicator][source-node] <=
          upperbound*2 and current-distance+Adjacency-matrix[indicator][source-node] >= lowerbound
          then
            mindist = curdist + Adjacency-matrix[u][indicator]
            vector <int> optimizedTour = twoOptHeuristic(paths, Adjacency-matrix)
            currentDistance = calculateTourDistance(optimizedTour, Adjacency-matrix)
            if mindist <= currentDistance then
                ans[paths] = curdist + Adjacency-matrix[u][indicator]
            else
                mindist=currentDistance
                ans[optimizedTour]=mindist
    for i in range(0,n) do
        if visited-nodes[i] == 0 then
            paths.push-back(i)
            if current-distance+Adjacency-matrix[indicator][i] <
              mindist and current-distance+Adjacency-matrix[indicator][source-node] <= upperbound*2
              then
                bnbdfs(i,visited-nodes,Adjacency-matrix,current-distance+Adjacency-
                  matrix[indicator][i],counter,paths,n,source-node)
            paths.pop-back()
    visited[indicator]=0
```

Fig 2

Source: Since we can't stick to a single node as the source and destination, instead we have to explore paths starting and ending at all the nodes. That's why I am looping through the initial node and calling the recursive function n (number of nodes) times.

Heuristics: We are using two heuristics to add intelligence to our BNB DFS algorithm to make it even faster.

1. ***Minimum Spanning Tree:*** An MST is a tree-like subgraph of a connected, undirected graph that spans all its vertices with the minimum possible total edge weight. In an MST, each vertex is included, and the edges form a tree without creating cycles. The primary objective is to connect all vertices with the least overall weight, ensuring efficient connectivity.

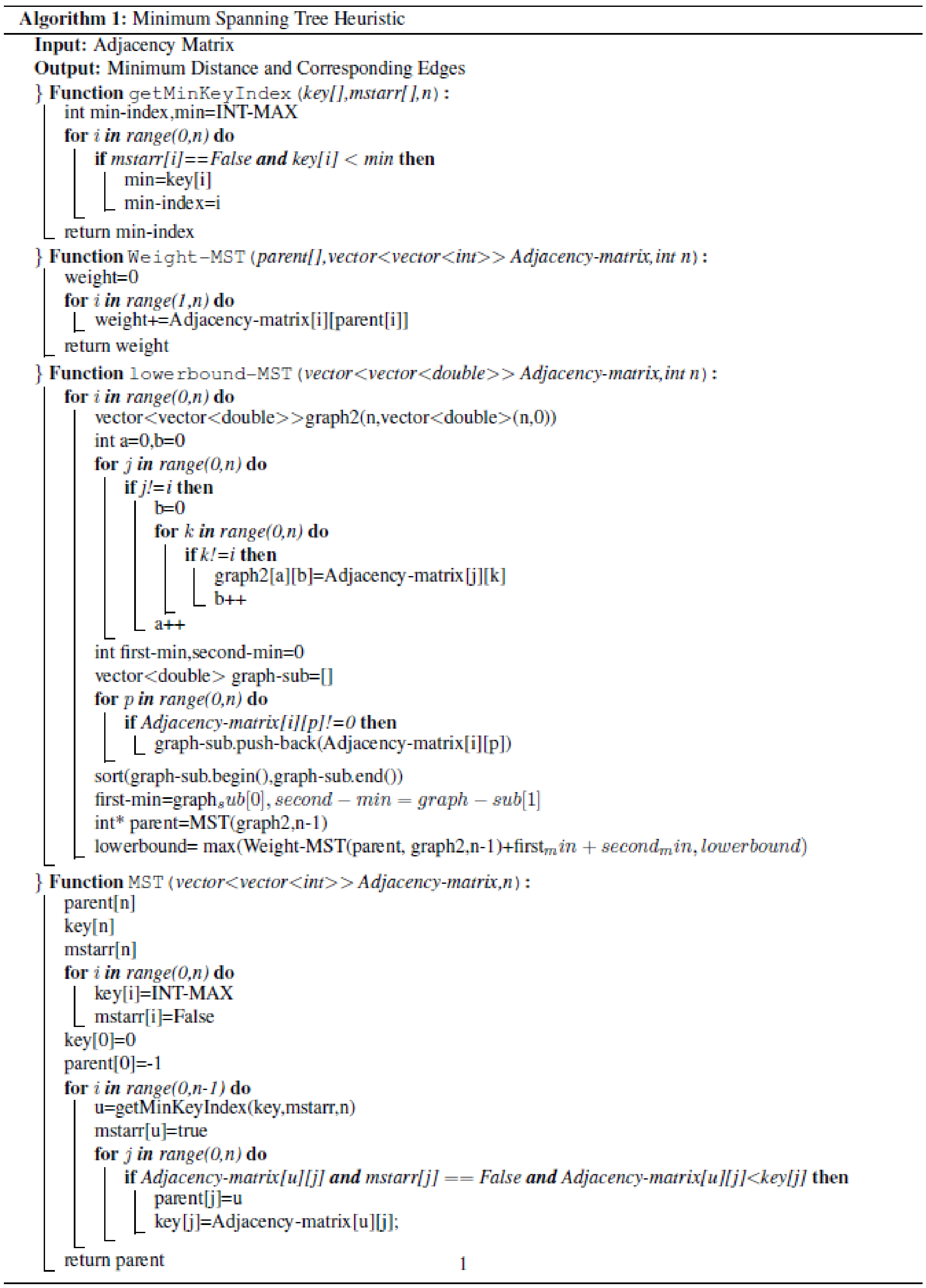
Working: The MST function takes the adjacency matrix and maintains three lists: parent, key, and mstarr. The parent list stores the parent node of each node. For the first node, the parent is always set to -1, initialized accordingly. The key list stores the distance between the current node and its parent node. The mstarr is a boolean list used to track visited nodes. Initialization starts at the first node, considering it as the first unexplored one [mstarr=False]. We mark all other nodes as its children if there is a node between them [adjacency_matrix[u][j]=True]. We then traverse to the next minimum node using the getMinKeyIndex function. This provides the next unexplored node [mstarr[v]==false] with the lowest edge value [key[v]<min]. In each iteration, the current node becomes the parent, and we check for its children by updating their edge weights in the key list and parent details in the parent list. The key list is updated at every iteration with the minimum possible edge values, allowing us to choose the lowest edge. Similarly, the parent list is updated whenever we find a new parent that provides the lowest edge

distance value for each child node. At the end of the process, we obtain the lowest possible total distance in the key list and the connected edges in the parent list.

Integrating the heuristic into the BNB-DFS algorithm:

Without the heuristic, we iterate/exhaust a node, and once we reach the goal, we update the mindistance with that path's value. Later, while exploring other paths, we compare the current distance with the minimum distance to prune the lower nodes. However, consider leveraging edges that are known to provide the lowest possible edge weight. This can be achieved using the Minimum Spanning Tree (MST). The MST serves as a valuable heuristic in addressing the Traveling Salesman Problem (TSP). In this approach, the weight of an MST is used to define the range for the TSP problem. The upperbound of a TSP's total distance= 2*MST weight. Whereas the lowerbound of a TSP can be calculated by removing each node at a time and calculating the MST of the remaining tree and later adding the 2 lowest possible edges of the previously removed node. The TSP's lowest distance will always fall between the lower and upper bounds. This way while iterating the tree we can prune the paths that are exceeding the upperbound. Although the MST solution may not be optimal, it acts as a robust foundation for the Branch and Bound algorithm to systematically examine and improve potential tours. This ultimately aids in discovering near-optimal solutions for the Traveling Salesman Problem. The involvement of heuristic can be found in Fig 1 where we call the MST function to initialize the *parent* list which is later used to calculate the lowerbound and upperbounds.These boundaries are later implemented in the BNBDFS function[Fig 2]

Pseudo code of the Minimum Spanning Tree Heuristic Algorithm:



```
Algorithm 1: Minimum Spanning Tree Heuristic
  Input: Adjacency Matrix
  Output: Minimum Distance and Corresponding Edges
  } Function getMinKeyIndex (key[],mstarr[ ],n) :
      int min-index,min=INT-MAX
      for i in range(0,n) do
          if mstarr[i]==False and key[i] < min then
              min=key[i]
              min-index=i
      return min-index
  } Function Weight-MST (parent[],vector<vector<int>> Adjacency-matrix,int n) :
      weight=0
      for i in range(1,n) do
          weight+=Adjacency-matrix[i][parent[i]]
      return weight
  } Function lowerbound-MST (vector<vector<double>> Adjacency-matrix,int n) :
      for i in range(0,n) do
          vector<vector<double>>graph2(n,vector<double>(n,0))
          int a=0,b=0
          for j in range(0,n) do
              if j!=i then
                  b=0
                  for k in range(0,n) do
                      if k!=i then
                          graph2[a][b]=Adjacency-matrix[j][k]
                          b++
                  a++
          int first-min,second-min=0
          vector<double> graph-sub=[]
          for p in range(0,n) do
              if Adjacency-matrix[i][p]!=0 then
                  graph-sub.push-back(Adjacency-matrix[i][p])
          sort(graph-sub.begin(),graph-sub.end())
          first-min=graph$_s ub[0], second - min = graph - sub[1]$
          int* parent=MST(graph2,n-1)
          lowerbound= max(Weight-MST(parent, graph2,n-1)+first$_m in + second_m in, lowerbound)$
  } Function MST (vector<vector<int>> Adjacency-matrix,n) :
      parent[n]
      key[n]
      mstarr[n]
      for i in range(0,n) do
          key[i]=INT-MAX
          mstarr[i]=False
      key[0]=0
      parent[0]=-1
      for i in range(0,n-1) do
          u=getMinKeyIndex(key,mstarr,n)
          mstarr[u]=true
          for j in range(0,n) do
              if Adjacency-matrix[u][j] and mstarr[j] == False and Adjacency-matrix[u][j]<key[j] then
                  parent[j]=u
                  key[j]=Adjacency-matrix[u][j];
      return parent                                   1
```

Fig 3

Time Complexity:

The MST construction has a time complexity of O(V^2), whereas the bnbdfs is a recursive function with a for loop inside it. The loop is executed n * n^n = n^(n+1) times. This is repeatedly done for n sources, so the total time complexity is approximately O(n^(n+2))~~O(n^n)

Space Complexity:

The primary space complexity is due to the recursive call in the bnbdfs function. The depth of recursion[assume the recursive calls are stored in a stack, that is how we return to the previous state by popping the stack] in the worst case can be n. The adjacency matrix stores weights for each edge between nodes, thus the space complexity is O(n^2). Therefore, the space complexity is O(n^2). Visited, paths also contribute to the space complexity, for them it’s O(n). Total space complexity=O(n^2)

2. We explored another heuristic called ***2-OPT heuristic***:

The 2-opt heuristic starts with an arbitrary tour; it repeatedly replaces k edges of the tour with k other edges and continues replacing edges as long as it yields a shorter tour. For our use case, we are sticking with the 2-OPT heuristic. Assume a case where we decide on a path and want to enhance it further, making it even less costly. We achieve this approach from the idea that, to prepare a completely new distinct path that is different from the older one, we need a maximum of n-2 shared nodes among the two paths. Programmatically speaking, to achieve this, we select 2 edges that have different endpoints to break and rearrange them in a way that shortens the overall path. It is possible to generate a total of n*(n-3)/2 distinct nodes after performing 2-OPTs, so we have to generate these new paths and calculate their costs. Once we reach a lower cost, we stop the generation and explore the new path. This step is continuously repeated until no further enhancement can be made. 2-OPT doesn't always give us the best optimal path; in some cases, it might get stuck in a path's children, as we aren't exploring all n*(n-3)/2 possibilities but rather just taking the first 2-OPT path that gives us the lowest distance. We solve this problem by returning back to DFS and exploring a new path altogether. ***Based on our analysis, the 2-OPT heuristic is found to improve the performance of the TSP by giving us the near to optimal path very early into the execution.***

Pseudo code of the 2-OPT heuistic:

```
Algorithm 1: 2-Opt Heuristic
  Input: Path, Adjacency Matrix
  Output: Adjacency matrices after swapping 2 edges
  map<vector<int>, int> ans
  mindist=1e9
  } Function calculateTourDistance( vector<int> path,vector<vector<int>> Adjacency-matrix ):
      distance = 0
      for i in range(0,n-1) do
          distance += Adjacency-matrix[path[i]][path[i + 1]]
      distance += Adjacency-matrix[path.back()][path.front()]
      return distance
  } Function twooptswap( vector<int> path,i,k ):
      newTour = path
      for i < k do
          swap(newTour[i], newTour[k])
          i++
          k- -
      return newTour
  } Function twooptheuristic( vector<int> tour, vector<vector<int>> Adjacency-matrix,n ):
      n = tour.size()
      vector<int> currentTour=tour
      for i in range(0,n-2) do
          for k in range(i+2,n) do
              vector<int> newTour = twoOptSwap(currentTour, i + 1, k)
              currentDistance = calculateTourDistance(currentTour, distanceMatrix)
              newDistance = calculateTourDistance(newTour, distanceMatrix)
              if newDistance¡currentDistance then
                  currentTour = newTour;
      return currentTour
```

Fig 4

Working: The DFS takes an adjacency matrix and generates an initial complete path. This path is then passed down as the input to the 2-OPT heuristic. Later, the 2-OPT heuristic continues swapping 2 edges until the current distance is less than the previous total distance, potentially yielding a maximum of n*(n-3)/2 distinct paths in the worst case. If a swap is successful, meaning it produces a path with a minimum distance, the algorithm stores that path into the minimum distance list. This process continues until a state is reached where no more minimum distance paths can be obtained. If the heuristic is unable to find a minimum path, it returns to the bnbdfs function, which explores a new path altogether, triggering another call to the 2-OPT heuristic function.

Time Complexity:

The bnbdfs is a recursive function with a for loop inside it. The loop is executed n * n^n = n^(n+1) times. This is repeatedly done for n sources, so the total time complexity is approximately O(n^(n+2)). The 2-OPT heuristic is O(n^2) and it is present in each bnbdfs function so the total time complexity is O(n*(n^2+n^n+1))~~O(n^n)

Space Complexity:

The primary space complexity is due to the recursive call in the bnbdfs function. The depth of recursion[assume the recursive calls are stored in a stack, that is how we return to the previous state by popping the stack] in the worst case can be n. The adjacency matrix stores weights for each edge between nodes, thus the space complexity is O(n^2). Therefore, the space complexity is O(n^2). Visited, paths also contribute to the space complexity, for them it's O(n). Total space complexity=O(n^2)

Observations:

1. We initially implemented a plain Branch and Bound Depth-First Search (BNBDFS) algorithm without incorporating any heuristics. Unfortunately, the execution time for this approach was excessively long, requiring 6 hours to reach an optimal solution for a node count (n) of 25.
2. Subsequently, we introduced the Minimum Spanning Tree (MST) heuristic. Despite including both upper and lower bounds derived from the MST heuristic, the performance did not witness significant improvement. Pruning was limited to only 4-5 paths, and the algorithm did not exhibit the desired acceleration.
3. To address this, we integrated the 2OPT heuristic as our primary heuristic. This adjustment yielded highly promising outcomes, allowing us to achieve a near-optimal solution in a matter of milliseconds, a remarkable improvement compared to the 2-hour duration required by the MST heuristic alone.
4. However, a new challenge surfaced when attempting to find subsequent optimal solutions using 2OPT. Although the initial iteration provided a near-optimal solution swiftly, the subsequent iterations proved time-consuming (1-2 hours). This extended duration is attributed to the fact that 2OPT, having already delivered an optimal solution initially, necessitates extensive backtracking to explore alternative optimal solutions. 2-OPT also goes through 2 nested for loops running nearly n^2 times to generate multiple combination.

Stochastic Local Search Algorithm

The different types of SLS algorithms include Simulated Annealing, Genetic Algorithms, Tabu Search, and Ant Colony Optimization.

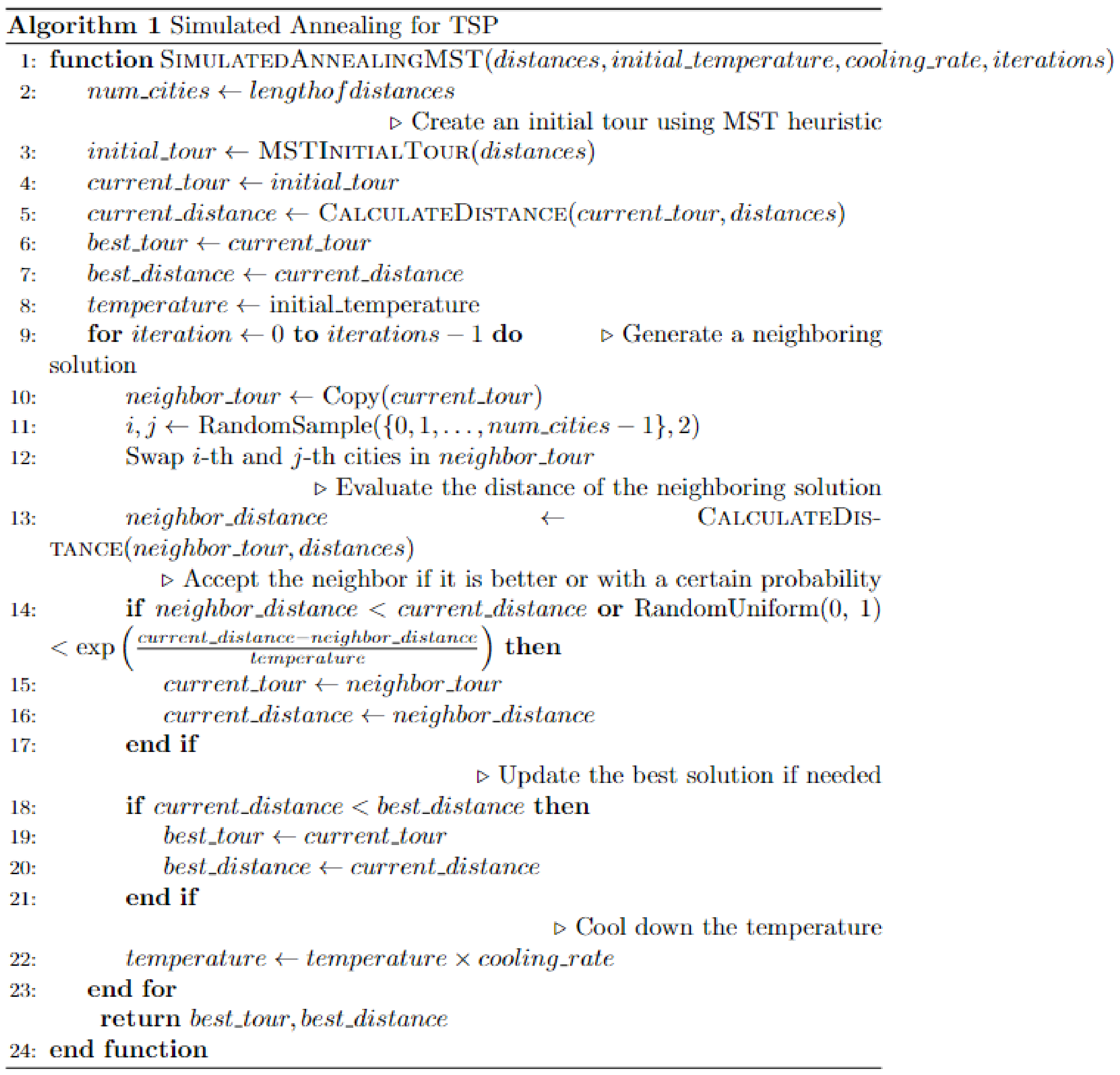
We have chosen Simulated Annealing for solving the Traveling salesman problem.This algorithm is inspired by the annealing process in metallurgy. It starts with a high temperature (allowing for more exploration), and as the algorithm progresses, the temperature decreases, reducing the randomness and focusing on exploitation. The advantages of using Simulated annealing to solve the TSP are:

- Global Optimization: Simulated Annealing is capable of escaping local optima, it explores the solution space globally, increasing the likelihood of finding a near-optimal solution.
- Flexibility and Adaptability: It is well-suited for optimization problems with different characteristics and constraints.
- Acceptance of Worse Solutions: SA's acceptance of worse solutions with a certain probability allows it to explore a broader solution space, preventing it from getting stuck in local optima. This stochastic nature helps SA to escape from suboptimal solutions and explore the search space more effectively.
- Parameter Tuning: SA allows for easy adjustment of parameters such as initial temperature, cooling rate, and number of iterations.
- Robustness to Noise: SA is robust to noisy and imperfect objective functions. This property is beneficial in real-world scenarios where the objective function might have noise or uncertainty.
- Adaptive Exploration-Exploitation Balance: The temperature parameter in SA controls the balance between exploration and exploitation. Initially, it allows the algorithm to explore the solution space widely, and as the temperature decreases, the algorithm focuses more on exploiting promising areas.
- Parallelization: Simulated Annealing can be parallelized, enabling the algorithm to leverage parallel processing capabilities to explore the solution space more efficiently.
- Simplicity in Handling Constraints: SA can be easily adapted to handle constrained optimization problems, such as TSP with additional constraints.

  The performance of the Simulated Annealing algorithm for the Traveling Salesman Problem (TSP) can be influenced by various parameters. Tuning these parameters appropriately is crucial for obtaining good results. Here are some parameters we adjust to potentially improve the code:

- Initial Temperature: A higher initial temperature allows the algorithm to accept worse solutions with higher probability, promoting exploration of the solution space. However, if the temperature is too high, the algorithm may get stuck in suboptimal solutions.
- Cooling Rate: The cooling rate determines how fast the temperature decreases. A slower cooling rate allows for more exploration early on, while a faster cooling rate emphasizes exploitation. Too fast cooling may cause premature convergence.
- Number of Iterations: The number of iterations determines how long the algorithm runs. Too few iterations may lead to an insufficient exploration of the solution space, while too many iterations might result in excessive computation time.

Pseudocode for Simulated Annealing for Traveling Salesman Problem



Algorithm 1 Simulated Annealing for TSP

```
1:  function SimulatedAnnealingMST($distances, initial\_temperature, cooling\_rate, iterations$)
2:      $num\_cities \leftarrow lengthofdistances$
                                        ▷ Create an initial tour using MST heuristic
3:      $initial\_tour \leftarrow$ MSTInitialTour($distances$)
4:      $current\_tour \leftarrow initial\_tour$
5:      $current\_distance \leftarrow$ CalculateDistance($current\_tour, distances$)
6:      $best\_tour \leftarrow current\_tour$
7:      $best\_distance \leftarrow current\_distance$
8:      $temperature \leftarrow$ initial_temperature
9:      for $iteration \leftarrow 0$ to $iterations - 1$ do        ▷ Generate a neighboring solution
10:         $neighbor\_tour \leftarrow$ Copy($current\_tour$)
11:         $i, j \leftarrow$ RandomSample($\{0, 1, \ldots, num\_cities - 1\}, 2$)
12:         Swap $i$-th and $j$-th cities in $neighbor\_tour$
                                        ▷ Evaluate the distance of the neighboring solution
13:         $neighbor\_distance \leftarrow$ CalculateDistance($neighbor\_tour, distances$)
                                        ▷ Accept the neighbor if it is better or with a certain probability
14:         if $neighbor\_distance < current\_distance$ or RandomUniform(0, 1) $< \exp\left(\frac{current\_distance - neighbor\_distance}{temperature}\right)$ then
15:             $current\_tour \leftarrow neighbor\_tour$
16:             $current\_distance \leftarrow neighbor\_distance$
17:         end if
                                        ▷ Update the best solution if needed
18:         if $current\_distance < best\_distance$ then
19:             $best\_tour \leftarrow current\_tour$
20:             $best\_distance \leftarrow current\_distance$
21:         end if
                                        ▷ Cool down the temperature
22:         $temperature \leftarrow temperature \times cooling\_rate$
23:     end for
         return $best\_tour, best\_distance$
24: end function
```

Initialization:

- The function starts by initializing the number of cities creating an initial tour using the MST heuristic and initializing variables to store the current best tour and its distance.

Iterative Improvement:

- Repeat the following steps until the stopping criterion is met:
    - Perturb the current tour to generate a neighboring solution (by swapping two cities).
    - Evaluate the cost (total distance) of the new tour.
    - Accept the new tour with a probability determined by the Metropolis criterion, which depends on the current temperature and the change in cost.

- Update the temperature according to the cooling schedule.
- Break when the temperature converges to zero.

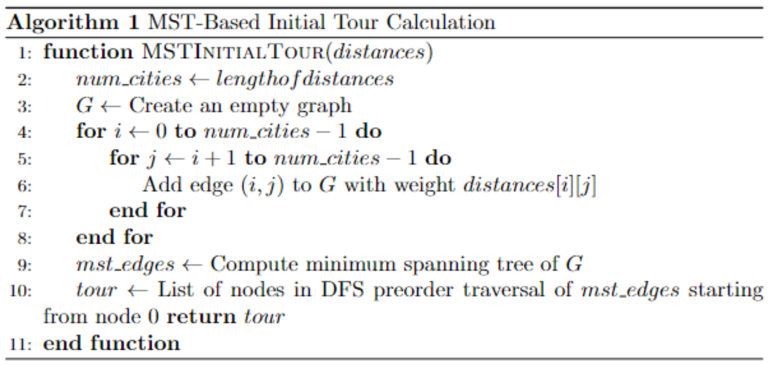
Result:

- The function returns the best tour and its distance found during the simulated annealing process.

Heuristics: We are using Minimum spanning tree heuristic to decide the initial tour and we tune the parameters initial temperature and cooling rate for simulated annealing.

1) Minimum spanning tree: The MST is a tree that spans all the cities in the TSP with the minimum total edge weight. In the context of TSP, the edge weights are the distances between cities. The MST heuristic creates a tree structure that connects all cities while minimizing the total distance traveled.
    - Generating the MST: To apply the MST heuristic, we first construct a complete graph where cities are nodes, and edges represent the distances between cities. We then use Kruskal's algorithm to find the minimum spanning tree.
    - Creating an Initial Tour: The MST is a tree, and a tour for TSP needs to be a closed loop. To convert the MST into a tour, we traverse the tree in a way that minimizes the total distance traveled. This can be achieved using a depth-first search (DFS) preorder traversal starting from an arbitrary node.
    - Simulated Annealing Optimization: We then apply the simulated annealing algorithm to refine and improve the solution. Simulated annealing is a metaheuristic that mimics the annealing process in metallurgy. It starts with a high-temperature phase where the algorithm is allowed to explore the solution space more broadly, and then gradually decreases the temperature, allowing the algorithm to converge towards better solutions.

The following is the pseudocode to find the MST:



Algorithm 1 MST-Based Initial Tour Calculation

```
function MSTInitialTour($distances$)
    $num\_cities \leftarrow lengthofdistances$
    $G \leftarrow$ Create an empty graph
    for $i \leftarrow 0$ to $num\_cities - 1$ do
        for $j \leftarrow i + 1$ to $num\_cities - 1$ do
            Add edge $(i, j)$ to $G$ with weight $distances[i][j]$
        end for
    end for
    $mst\_edges \leftarrow$ Compute minimum spanning tree of $G$
    $tour \leftarrow$ List of nodes in DFS preorder traversal of $mst\_edges$ starting from node 0 return $tour$
end function
```

OBSERVATIONS:

1) First we implemented Simulated annealing which took around 10 seconds just to run for 25 cities.
2) Upon implementing Minimal Spanning Tree heuristic the time reduced to around 2 seconds.
3) In order to further improve the algorithm we included a 2-opt move heuristic to generate tours during each iteration of simulated annealing. As such the combination of MST and 2-opt lead to an increase in the performance of the algorithm for simulated annealing

and the accuracy to find the best optimal solution also improved. Further the execution time was reduced to 0.06 seconds.

The following is the updated algorithm for solving traveling salesman problem using simulated annealing:

```
Algorithm 1: Simulated Annealing
Input: Adjacency Matrix, Maximum Iterations, Initial Temperature, Cooling Rate
Output: Best Solution and Best Distance
} Function simulated_annealing( vector<int> path,vector<vector<int>> Adjacency-matrix) :
    current-solution = generate-mst-solution(distance-matrix,num-cities)
    current-distance = calculate-total-distance(current-solution,
     distance-matrix)
    best-solution = current-solution.copy()
    best-distance = current-distance
    temperature = initial-temperature
    for iteration in range(0,max-iterations) do
        neighbor-solution = generate-neighbor-solution(current-solution)
        neighbor-distance = calculate-total-distance(neighbor-solution,
         distance-matrix)
        probability = acceptance-probability(current-distance,
         neighbor-distance, temperature)
        if random.uniform(0, 1)<sprobability then
            current-solution = neighbor-solution
            current-distance = neighbor-distance
        if current-distance<best-distance then
            best-solution = current-solution.copy()
            best-distance = current-distance
        temperature *= cooling-rate
    distance += Adjacency-matrix[path.back()][path.front()]
    return distance
```

2-opt algorithm:

```
Algorithm 1: Generate Neighbor Solution
Input: current-solution
Output: New Solution
} Function generate-neighbor-solution ( int current-solution ) :
    i, j = sorted(random.sample(range(len(current-solution)), 2))
    new-solution = current-solution[:i] + list(reversed(current-solution[i:j + 1])) + current-solution[j + 1:]
    return new-solution
```

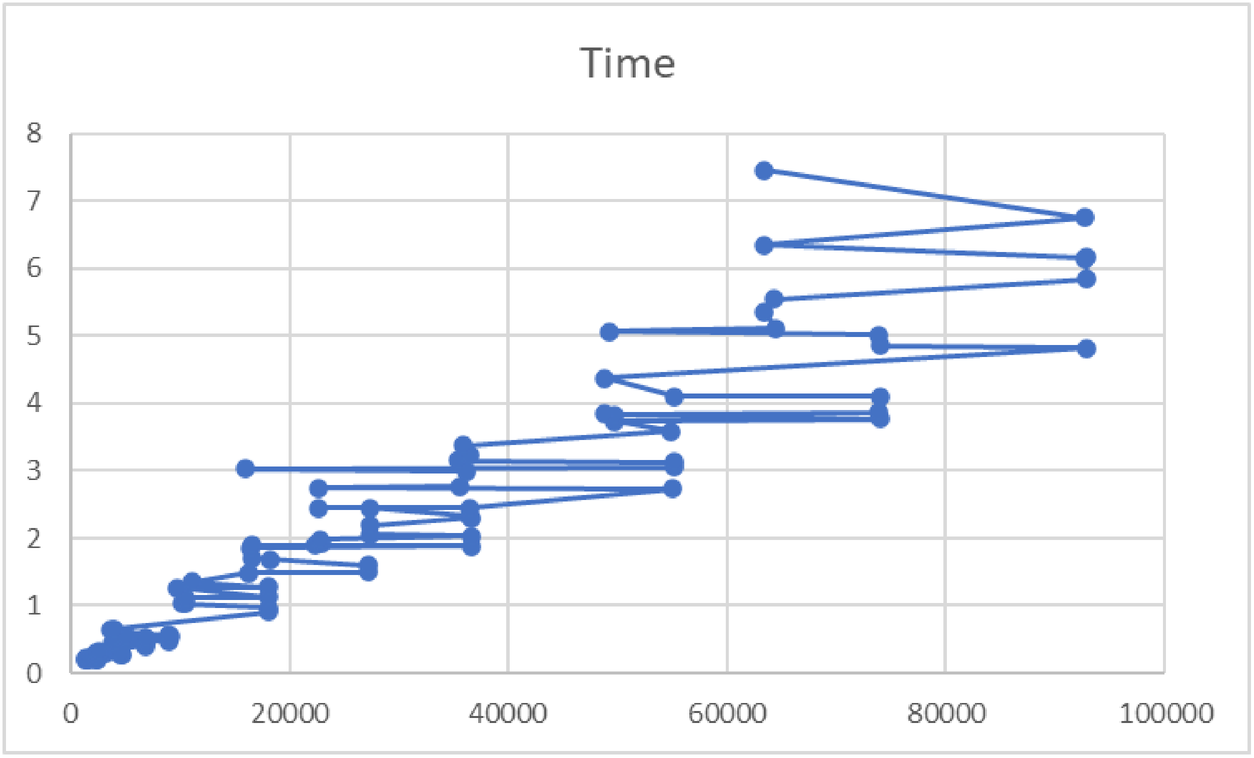


Fig 5

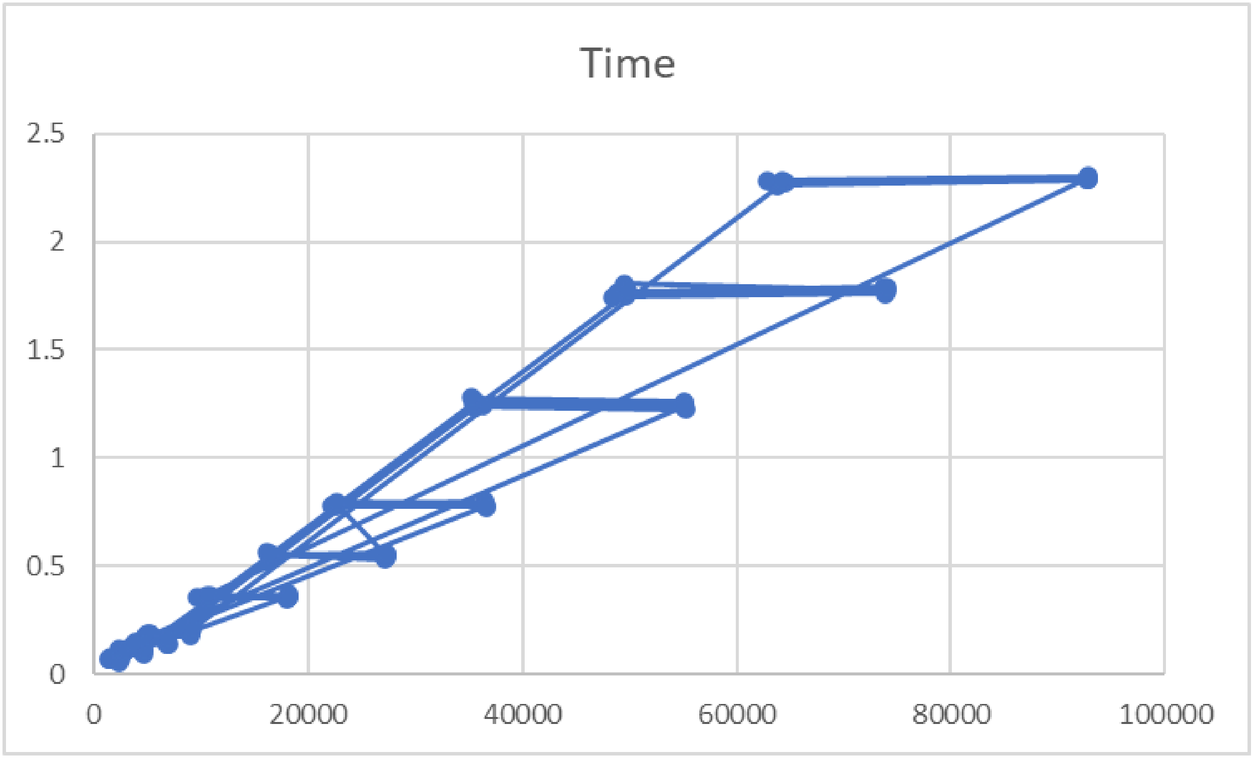


Fig 6

Fig. 5 depicts the distance vs. time graph for the SLS heuristic, while Fig. 6 illustrates the scenario when SLS and MST are considered together. It is evident from these graphs that there

is a drastic change (1/4th reduction) in the time taken to arrive at an optimal distance when using SLS+MST.
Time Complexity: The time complexity is often expressed as O(iterations * N)~O(n^2). The dominant factors are the number of iterations and the cost of generating neighboring solutions and evaluating the objective function. The complexity is also influenced by the complexity of the cooling rate and initial temperature computation. Simulated annealing is a heuristic algorithm, and its time complexity analysis is often more focused on empirical observations and practical performance rather than theoretical worst-case analysis. The actual running time can vary based on the specific problem instances and parameter settings.

Space complexity: The adjacency matrix stores weights for each edge between nodes, thus the space complexity is O(n^2). Therefore, the space complexity is O(n^2). The space complexity of simulated annealing for TSP can be expressed as O(N), where N is the number of cities or nodes in the problem as random tours can be considered to be a list of city indices for N cities. Hence to summarize the Space complexity for the complete algorithm is O(N^2).

References:

- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- https://www.youtube.com/watch?v=8vbKIfpDPJI
- https://imada.sdu.dk/u/marco/Teaching/AY2016-2017/DM841/Slides/dm841-p2-local_search.pdf
- https://link.springer.com/article/10.1023/B:COAP.0000044187.23143.bd#:~:text=The%20classical%20version%20of%20simulated,a%20certain%20value%20%CF%870.
- https://www.researchgate.net/publication/227061666_Computing_the_Initial_Temperature_of_Simulated_Annealing