Université Jean Monnet

Advanced Algorithms and Programming Project 2024-2025

Comparing Various Algorithms Used to Solve the Traveling Salesman Problem

Professor: Amaury Habrard

Team Members: Patrick Barry
Buu Dinh Ha
Mahima Haridasan Sumathy
Chelsy Mena
Serhii Vakulenko

Saint-Etienne, 12/2024

TABLE OF CONTENTS

CHAPTER 1. Introduction

1.1 Problem Definition

The Travelling Salesman Problem (TSP) is the problem of finding the route that is shortest and most efficient for a person to take, given a list of destinations and the cost of travelling between each pairs.

Stated formally, given a set of n cities and distances between every pair, the problem is to find the shortest route that visits each city exactly once and returns to the starting city. TSP is an NP-Hard problem since there is no polynomial-time solution exists for this problem.

Asymmetric Traveling Salesman Problem (TSP) is an alternative of TSP, but differentiated by the fact that the distance function may not be symmetric. That is, for two locations u and v, it is possible that $d(u, v) \neq d(v, u)$.

The TSP has several applications such as planning, logistics for mail service, and the manufacture of microchips by finding shortest-path for lasers to sculpt microprocessors.

1.2 Objective

Aim of the project is to implement these TSP-solving algorithms: brute-force, branch-and-bound, minimum spanning tree approximation, greedy, dynamic programming, branch-and-bound algorithm based on edge selection and matrix reduction, randomized, genetic and Lin-Kernighan.

The project aim is also to evaluate the solution produced by these algorithms on the instances generated by the random problem generator and TSP, ATSP instances from TSPLIB.

CHAPTER 2. Experimental Setup

2.1 Data Generation and Benchmark Datasets

The problem generators $generate\_tsp$ and $generate\_atsp$ creates instances of the Traveling Salesperson Problem (TSP) and its asymmetric variant (ATSP) based on two parameters: n (the number of cities) and $sparsity$ (a value between 0 and 1 representing the proportion of edges removed from a fully connected graph). Initially, n cities are randomly generated with coordinates drawn from a uniform distribution. The Euclidean distances between all city pairs are calculated and stored in a $distance_matrix$. If $sparsity$ is greater than 0, edges are removed to achieve the desired sparsity level. This is done by first generating a random tour (a random permutation of the cities) to ensure the graph remains connected. The specified percentage of edges not part of this random tour are then removed.

To evaluate the performance of the algorithms against known optima, we also utilized a selection of instances from the well-known TSPLIB benchmark library [1]. This library provides a collection of TSP instances of varying sizes and structures, along with their optimal solutions.

2.2 Evaluation Metrics

To evaluate the performance of the algorithms in general, we consider these metrics: wall-clock time, deviation from optimal solution (in percentage) and memory usage.

Each algorithm has a different method to evaluate the performance, which will be discussed in the next chapter. While the evaluation of individual algorithm was tested on multiple computers, for evaluation in chapter 5, we tested it on the base model MacBook Air (M1, 2020).

CHAPTER 3. General Implementation

3.1 Programming Language and Environment

The implementation of these TSP algorithms was carried out using Python.

3.2 Code Structure and Organization

The project structure is shown in table 3.1. The project follows a clear convention, each algorithm is stored in a separate folder. For each algorithm, the folder, file and main function are named after the algorithm itself. This structure allows easier integration and extension of the project

Table 3.1: Project structure

Folder	**Description**
ant_colony	implementation of ant_colony algorithm
branch_and_bound	implementation of branch_and_bound algorithm
brute_force	implementation of brute_force algorithm
data	store TSPLIB and random generated instances
dynamic_programming	implementation of dynamic_programming algorithm
genetic	implementation of genetic algorithm
greedy	implementation of greedy algorithm
gui	create GUI
lib	library for GUI
lin_kernighan	implementation of lin_kernighan algorithm
mst	implementation of mst algorithm
randomized	implementation of randomized algorithm
utils	functions to plot deviation, generate random problem, visualize problem

3.3 Graphical User Interface

The Graphical User Interface shown in figure 3.1 was developed using the open source python libraries **PyQt5** for the interface and **pyvis** for the visualization of the graph. These libraries together allow the user to deploy locally a window that leverages the native operating system interactive elements, and runs any of our algorithms with a TSPLIB instance or a TSP problem generated with the chosen size and sparsity.

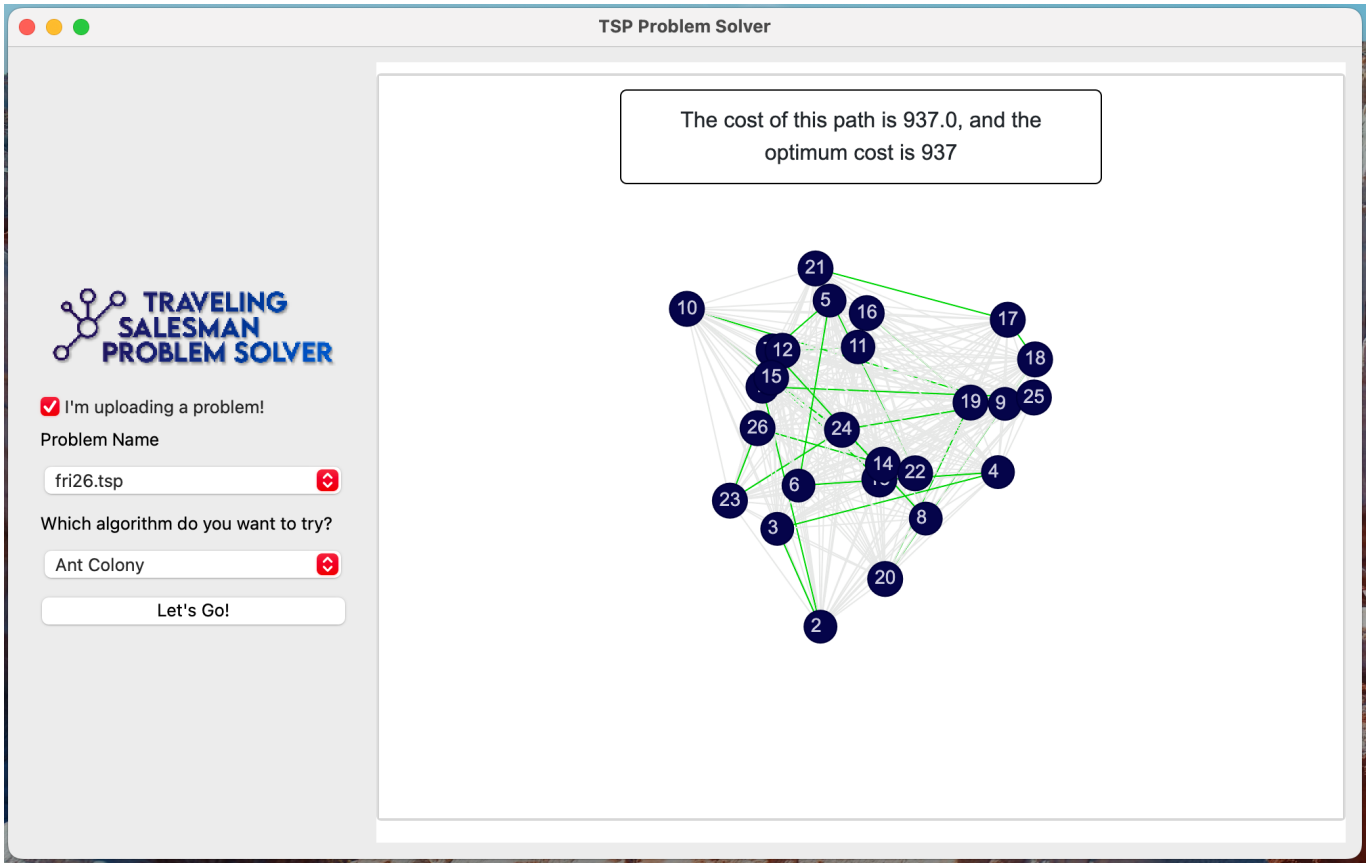


Figure 3.1: Graphical User Interface

4.1 Brute-force Approach

4.1.1 Introduction

The brute-force approach aims to solve a TSP instance by generating and checking all permutations of the path. This process is outlined in Algorithm 1. It is guaranteed to return the optimal solution. It is computationally very expensive with a complexity of $O(n!)$, where n is the number of cities in the given problem. This is because there are $(n-1)!$ possible paths to examine after fixing the starting city.

4.1.2 Algorithm

Algorithm 1 Brute-Force Algorithm for TSP

Input - Distance Matrix: D
Output - Minimum Cost: $cost_{min}$, Optimal Path: $path_{best}$

$n \leftarrow$ number of cities (size of D)
$path_{best} \leftarrow \emptyset$
$cost_{min} \leftarrow \infty$
$cities \leftarrow \{1, 2, ..., n-1\}$ ▷ Exclude starting city (0)
$permutations \leftarrow$ all permutations of $cities$
for each p in $permutations$ **do**
    $path \leftarrow [0] + p + [0]$ ▷ Add start and end city 0
    $cost \leftarrow 0$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        $city_1 \leftarrow path[i]$
        $city_2 \leftarrow path[i+1]$
        $cost \leftarrow cost + D[city_1][city_2]$
    if $cost < cost_{min}$ **then**
        $cost_{min} \leftarrow cost$
        $path_{best} \leftarrow path$
return $cost_{min}, path_{best}$

The itertools library in Python was used to generate and iterate through all permutations.

4.1.3 Results Analysis

This algorithm is one of the simplest but not efficient for problems larger than size 10. As shown in Figure 4.1, the execution time grows factorially with the number of cities. For instance, in our experiments, the average execution time for n=10 was 0.92 seconds, but it increased to 10.17 seconds for n=11.

4.2 Branch-and-Bound

4.2.1 Introduction

The branch and bound approach divides a problem into several sub-problems that can be solved recursively and bounding with a function so as to not compute sub cases which are not optimal. This approach was first devised by Land and Doig [2] in 1960., The approach consists of algorithms that follows the general rule of dividing into sub-problems (branching) and the use of a function to prune off sub-optimal cases (bounding). Here, two implementations using the branch and bound approach have been done.

4.2.2 Algorithm

a, General Branch and Bound Working

The branch and bound algorithm represents a TSP instance as a state-space tree,from a given distance matrix, where each node is a city and its children are unvisited cities with an edge in the graph connected

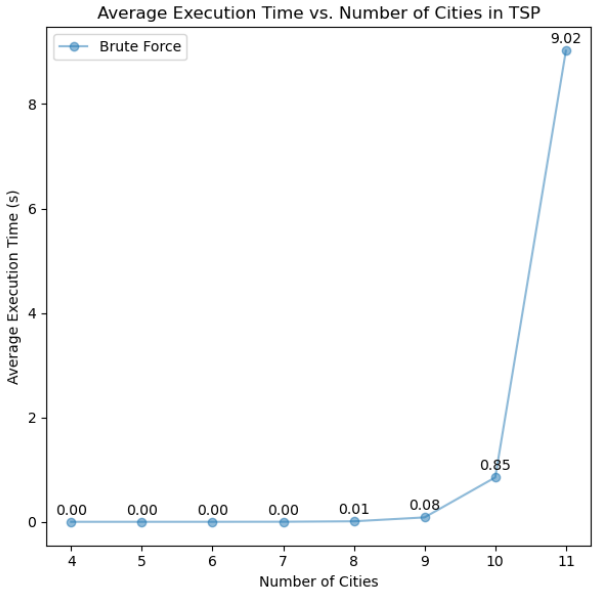


Figure 4.1: CPU Execution Time of Brute-Force Algorithm vs. Number of Cities

to the parent. When the algorithm reaches a vertex i, it chooses the next unvisited city that guarantees an optimal solution. This is done by calculating the lower bound of a solution in each city not visited connected to i and going to the city with the lowest lower bound. The calculation of the lower bound for the root node, the second node and the rest of the nodes is calculated differently. An optimal tour for a given graph would pass through the two lowest weighted edges of each vertices. The lower bound for the second node is calculated by subtracting the average minimum edge weights of the first and second nodes from the original lower bound and then adding the cost of traveling from the first to the second node. For all other levels, the lower bound to the next city is calculated by subtracting the average of the minimum edge weight of the current node and the second lowest edge weight of the previous node from the previous lower bound and then adding the cost of traveling from the previous node to the current node. The minimum edge weight of the previous node was already subtracted to calculate the previous lower bound, so the second lowest edge weight is used instead.

This algorithm recursively traverses the tree until it reaches a leaf, at which point it will stop and return to the root. The nodes that the algorithm passes through will be the optimal path for a TSP instance. The pseudo-code for this algorithm is given in algorithm 2

b, Using Reduction Matrix and Edge Selection

The reduction and Edge Selection process helps in pruning the search tree more efficiently in the branch-and-bound algorithm by reducing the search space tree.

As discussed in Reignold1997 [3] this method uses the fact that if a constant is subtracted from any row or column of the cost matrix,the optimal solution does not change. In fact,the cost of the optimal solution is diminished by exactly the amount subtracted from the row or column. Therefore, if such a subtraction is done so that each row and column contains a zero and yet the cost from the i to j vertex all remains nonnegative,then the total amount subtracted will be a lower bound on the cost of any solution. For example given in 4.1 , ,the cost matrix given above can be reduced by subtracting 16,1,0,16,5,5 from rows 1 to 6 the same process is done for each columns. Then sum the minimum value for each rows and columns;the initial lower bound is 48 in this case of all possible tours. The reduced matrix represents a lower bound on the remaining path's cost. The algorithm calculates this bound and compares it with the best solution found so far. If the bound is better than the current solution, the node is further expanded. The nodes are added to a priority queue based on the cost (lower bound) of the node. The node with the smallest lower bound is

Algorithm 2 Branch and Bound Algorithm for TSP

Input - Distance Matrix: D
Output - Minimum Cost: $cost_{min}$, Optimal Path: $path_{best}$

$n \leftarrow$ number of cities (size of D)
$path_{best} \leftarrow \emptyset$
$cost_{min} \leftarrow \infty$
Initialize the priority queue (PQ) as an empty list.
$PQ.push((0, [0]))$ ▷ Push initial path (starting at city 0) with cost 0
while PQ is not empty **do**
    Pop the state with the least cost from the priority queue: $(current\_cost, current\_path) = PQ.pop()$
    if length of $current\_path$ equals n **then**
        $final\_cost \leftarrow current\_cost + D[current\_path[-1]][current\_path[0]]$
        if $final\_cost < cost_{min}$ **then**
            $cost_{min} \leftarrow final\_cost$
            $path_{best} \leftarrow current\_path + [current\_path[0]]$
        continue ▷ Skip further processing of this path
    for each city i from 0 to $n - 1$ **do**
        if i is not in $current\_path$ **then**
            $new\_cost \leftarrow current\_cost + D[current\_path[-1]][i]$
            $new\_path \leftarrow current\_path + [i]$
            if $new\_cost < cost_{min}$ **then**
                $PQ.push((new\_cost, new\_path))$
return $cost_{min}, path_{best}$

explored next. If at any point a node's lower bound is greater than the best solution found so far, it is pruned from the search tree. This means that we discard that branch as it is not likely to lead to a better solution.

A relatively successful technique for tree rearrangement in the traveling salesman problem is to split all remaining solutions into two groups at each stage, those that include a particular arc and those that exclude that arc. The arc used to split the solution space is intended to prune the tree as much as possible. For the same example shown in 4.2, if including (1,4), the recalculated bound remains 48. If excluding (1,4), the new bound is 58=48+10+0. Every node in the tree, now obviously a binary tree, will have associated with it a lower bound on the cost of all solutions descended from it. This process of excluding and including an arc or an edge technique is called edge selection.

Repeat the process until each branch represents a single valid tour. At this stage the tour with the smallest total cost is identified as the optimal solution. Any branches with a lower bound higher than this cost are pruned (discarded).

By systematically selecting edges and bounding non-optimal tours, the algorithm determines the optimal tour with the lowest possible cost. For this example 4.4, the optimal tour is found with a cost of 63. The number of nodes examined for this particular problem using our algorithm is a fraction of 325 nodes without pruning, if pruning is effective around 150-200 nodes might be examined.

The pseudo-code for this algorithm is given in algorithm 3

4.2.3 Results Analysis

The algorithm is guaranteed to give the optimal solution but not efficient for problems larger in size. The general execution time is larger, for problems more than 10 cities but after cutting down the cost through row and column reduction the efficiency improves dramatically. The **TimeComplexity** is (N-1) cities possible to visit, O(N^2) cost calculation and priority queue O(MlogM) where M is total number for nodes;

Algorithm 3 Reduction Matrix and Edge Selection Branch and Bound Algorithm

Input: Distance Matrix D
Output: Minimum Cost $costmin$, Optimal Path $pathbest$
$n \leftarrow$ number of cities (size of D)
$pathbest \leftarrow \emptyset$
$costmin \leftarrow \infty$
Initialize priority queue (PQ) as an empty list
PQ.push((0, [0])) ▷ Push initial path (starting at city 0) with cost 0
while PQ is not empty **do**
    Pop the state with the least cost from PQ: $(current\_cost, current\_path) \leftarrow$ PQ.pop()
    Perform row reduction on the distance matrix D to get a reduced matrix
    Row Reduction:
    for each row i from 0 to $n-1$ **do**
        $row[i] \leftarrow \min(D[i][j]$ for all $j)$
        for each column j **do**
            $D[i][j] \leftarrow D[i][j] - row[i]$
    Edge Selection: Choose edges (cities) to add to the path based on the reduced matrix
    if length of current_path equals n **then**
        $final\_cost \leftarrow current\_cost + D[current\_path[-1]][current\_path[0]]$
        if $final\_cost < costmin$ **then**
            $costmin \leftarrow final\_cost$
            $pathbest \leftarrow current\_path + [current\_path[0]]$
        continue ▷ Skip further processing of this path
    for each city i from 0 to $n-1$ **do**
        if i is not in $current\_path$ **then**
            $new\_cost \leftarrow current\_cost + D[current\_path[-1]][i]$
            $new\_path \leftarrow current\_path + [i]$
            Perform row and column reduction on the new matrix after adding the edge $(current\_path[-1], i)$
            if $new\_cost < costmin$ **then**
                PQ.push((new_cost, new_path))
Return $costmin, pathbest$

∞	27	43	16	30	26
7	∞	16	1	30	25
20	13	∞	35	5	0
21	16	25	∞	18	18
12	46	27	48	∞	5
23	5	5	9	5	∞

Table 4.1: Cost matrix for 6-City traveling salesman problem

∞	11	27	0	14	10
1	∞	15	0	29	24
15	13	∞	35	5	0
0	0	9	∞	2	2
2	41	22	43	∞	0
13	0	0	4	0	∞

Table 4.2: Cost matrix after row and column reduction

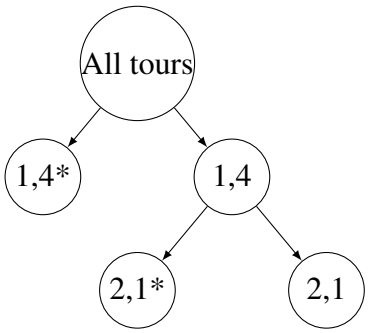
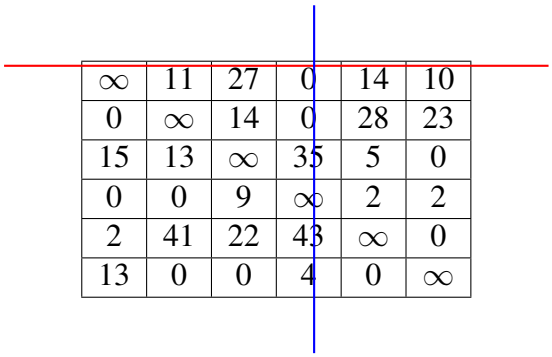


Figure 4.2: Start tree for the traveling salesman problem.



∞	11	27	0	14	10
0	∞	14	0	28	23
15	13	∞	35	5	0
0	0	9	∞	2	2
2	41	22	43	∞	0
13	0	0	4	0	∞

Figure 4.3: Matrix after deletion of row 1 and column 4.

O(N!.N).The **Space complexity** is similar but we consider the path storage therefore **O(N!.**N^2**)**.

4.3 Minimum Spanning Tree

4.3.1 Introduction

The MST-TSP algorithm is a heuristic approach to solving the TSP, which is based on the construction of a minimum spanning tree (MST). The implementation uses Prim's algorithm to construct the MST, after which a depth-first search (DFS) is performed to obtain a Hamiltonian cycle. This approach guarantees an approximation of the optimal solution with a factor of at least 2 (the length of the resulting route does not exceed twice the length of the optimal route). The algorithm is particularly effective for Euclidean TSP problems, where the distances between cities satisfy the triangle inequality, and can serve as a good starting point for further improvements with other optimization methods.

4.3.2 Algorithm

The MST-TSP algorithm consists of two main parts:

1. **Construction of MST - Prim's algorithm in our case:**
   - Initialization of arrays for keys, parent vertices and visited vertices
   - Iterative selection of the vertex with the minimum key among the unvisited ones
   - Update of keys for adjacent vertices
   - Formation of the list of MST edges based on parent links
2. **Creation of a tour using DFS:**
   - Construction of the adjacency list from MST edges
   - Starting DFS from vertex 1
   - Adding vertices to the tour in the order of their visit
   - Closing the tour by returning to the initial vertex

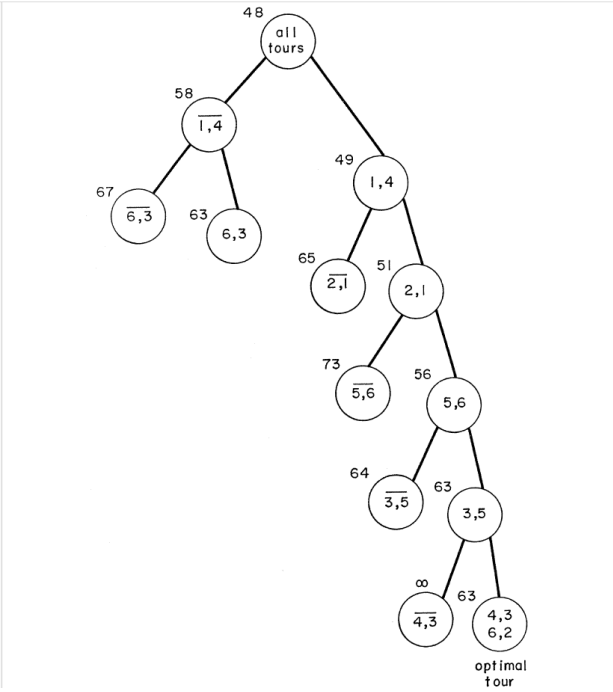


Figure 4.4: Final Tree.

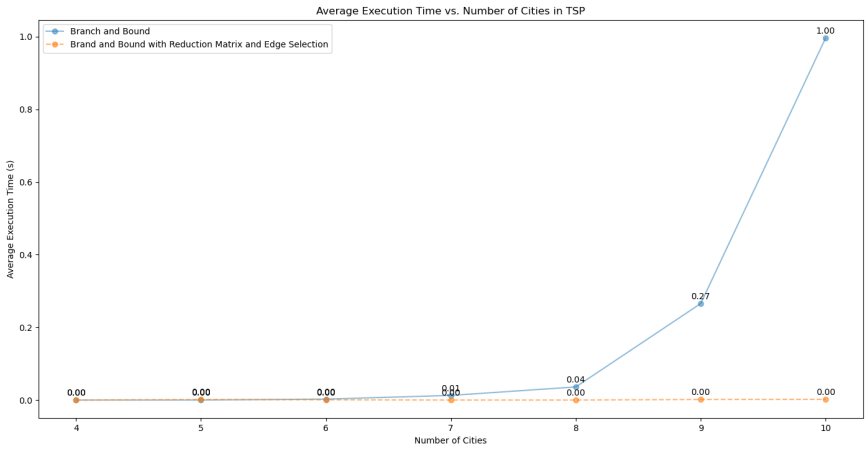


Figure 4.5: Execution time Between Reduction Matrix and General

Cost calculation:

- Calculation of the total length of the route
- Consideration of all transitions between consecutive cities in the tour

Algorithm complexity: $O(V^2)$ for construction of MST (where V is the number of vertices) and $O(V+E)$ for DFS, which in total gives $O(V^2)$ for dense graphs. The algorithm guarantees a 2-approximation of the optimal solution for metric TSP problems.

4.3.3 Result Analysis

Analysis of the MST-TSP test results of the algorithm shows that it provides extremely fast execution time (less than 0.002 seconds for problems up to 100 cities) with an almost constant time up to 40 cities and a moderate increase after 50 cities. However, the quality of the solution deviates significantly from the optimal: the error ranges from 20% for small problems (14-26 cities) to 42% for large problems (100 cities), which makes the algorithm suitable either as a fast initial approximation for more complex algorithms, or for situations where the priority is speed, not optimality of the solution.

Algorithm 4 MST-TSP Algorithm

Input - Distance Matrix: $distance\_matrix$
Output - Tour: $tour$, Cost: $cost$

$n \leftarrow$ number of cities (size of $distance\_matrix$)
$key \leftarrow [\infty, \infty, \ldots, \infty]$
$parent \leftarrow$ [null, null, . . . , null]
$mst\_set \leftarrow$ [false, false, . . . , false]
$key[0] \leftarrow 0$, $parent[0] \leftarrow 0$
▷ Step 1: Construct MST using Prim's Algorithm
for $i \leftarrow 1$ to $n - 1$ **do**
    Select u with minimum key where $mst\_set[u] =$ false
    $mst\_set[u] \leftarrow$ true
    for $v \leftarrow 1$ to n **do**
        if $distance\_matrix[u][v] > 0$ **and** NOT $mst\_set[v]$ **and** $distance\_matrix[u][v] < key[v]$ **then**
            $key[v] \leftarrow distance\_matrix[u][v]$
            $parent[v] \leftarrow u$
$edges \leftarrow$ List of MST edges from $parent$
▷ Step 2: Create Tour using DFS
Build adjacency list adj from $edges$
$visited \leftarrow$ [false, . . . , false], $tour \leftarrow$ []
DFS from vertex 1, appending visited vertices to $tour$
Append 1 to $tour$ ▷ Close the tour
▷ Step 3: Compute Tour Cost
$cost \leftarrow 0$
for $i \leftarrow 0$ to length$(tour) - 2$ **do**
    $cost \leftarrow cost + distance\_matrix[tour[i]][tour[i + 1]]$
return $tour, cost$

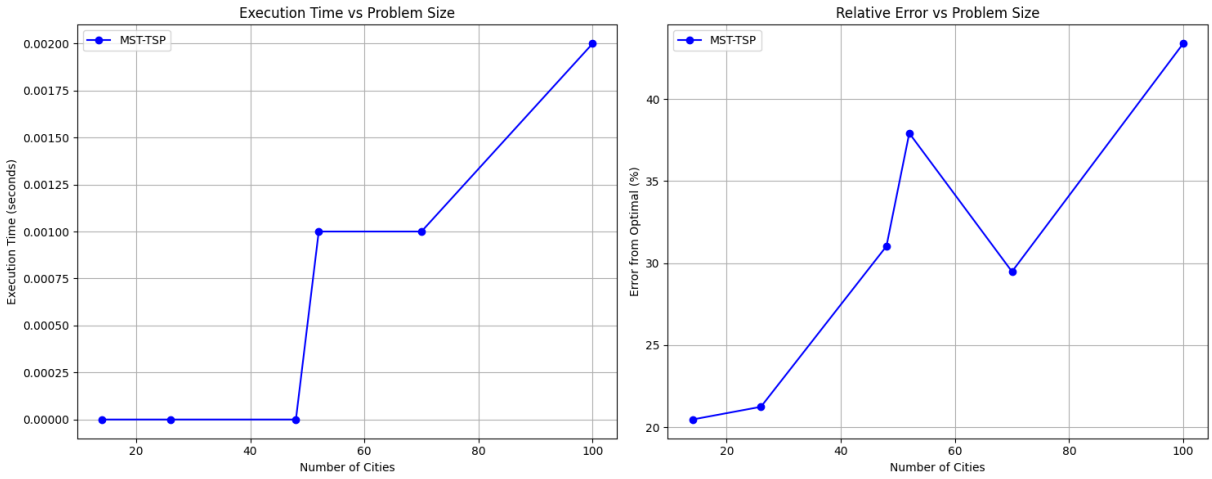


Figure 4.6: MST-TSP Performance Analysis: Execution Time vs Problem Size (left) and Relative Error vs Problem Size (right).

4.4 Greedy Approach

4.4.1 Introduction

We showed a brute force or dynamic programming approach for solving the TSP problem. It suffices to produce an optimal solution with these approaches, but they are also infeasible. There is no polynomial-time solution available for the problem; thus, it is NP-Hard. This problem can be solved by using Greedy approach, which quickly finds an approximate solution to the TSP by prioritizing the minimal distance at each step. A downside of this approach is that it does not guarantee the optimal tour.

4.4.2 Algorithm

A greedy algorithm constructs a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

The solution provided below is the basic weight-based greedy algorithm. The idea is to use a heuristic approach, which constructs a tour by starting from one city and iteratively building the solution. At each step, it selects the unvisited city with the smallest edge weight (shortest distance) from the current city, ensuring a local optimal choice. Then it is marked as visited; the algorithm moves to this city and continues the process. After visiting all the cities, we complete the tour by returning to the starting city.

The pseudo-code for this algorithm is shown in algorithm 5.

4.4.3 Results Analysis

Let us evaluate the performance of Greedy algorithm in term of runtime and deviation from known optimal solution, as shown in Figures 4.7 and 4.8. The average deviation is 21.96%, which shows that, on average, the solutions provided are 21.96% higher than the known optimal solutions. We can conclude that Greedy algorithm does not guarantee optimality. However, its limitation makes up with its efficiency, as it computes a solution for an instance with 7397 nodes in just 4.84 seconds, making it practical for large-scale problems.

One notable observation in Figure 4.7 is that instance pla7397, which is significantly larger than the others (e.g. pr1002), results in a solution with a lower deviation. This counterintuitive result can be explained by examining Figure 4.9, which visualizes the node distributions for these instances. The pla7397 instance exhibits a more symmetrical structure with two distinct clusters, allowing the Greedy algorithm to make better local decisions. In contrast, the pr1002 instance appears more chaotic and irregular, lacking clear patterns for the algorithm to exploit, making it more challenging to approximate an optimal solution effectively.

Algorithm 5 Greedy Algorithm for TSP

Input - Distance Matrix tsp_matrix
Output - Minimum Cost $total\_cost$ and Path Sequence $route$

$num\_nodes \leftarrow$ length of tsp_matrix
$total\_cost \leftarrow 0$, $step\_counter \leftarrow 0$, $current\_city \leftarrow 0$, $min\_distance \leftarrow \infty$
$visited\_cities \leftarrow []$, $route \leftarrow []$, $route[0] \leftarrow 1$, $visited\_cities[0] \leftarrow 1$
while $step\_counter < num\_nodes - 1$ **do**
    for $next\_city \leftarrow 0$ to $num\_nodes - 1$ **do**
        if $next\_city \neq current\_city$ **and** $next\_city$ is unvisited **then**
            if $tsp\_matrix[current\_city][next\_city] < min\_distance$ **then**
                $min\_distance \leftarrow tsp\_matrix[current\_city][next\_city]$
                $route[step\_counter + 1] \leftarrow next\_city + 1$
    $total\_cost \leftarrow total\_cost + min\_distance$
    $min\_distance \leftarrow \infty$
    $visited\_cities[route[step\_counter + 1] - 1] \leftarrow 1$
    $current\_city \leftarrow route[step\_counter + 1] - 1$
    $step\_counter \leftarrow step\_counter + 1$
$last\_city \leftarrow route[step\_counter] - 1$
$total\_cost \leftarrow total\_cost + tsp\_matrix[last\_city][0]$
$route[step\_counter + 1] \leftarrow 1$
return $route$, $total\_cost$

4.4.4 Limitations

As mentioned above, the limitation of greedy comes from its lack of global perspective and inability to produce the optimal solution, making it not suitable for applications requiring high accuracy. To overcome these issues, refinement techniques such as 2-Opt, 3-Opt, or other metaheuristic approaches (e.g., Genetic Algorithms or Ant Colony Optimization) can be applied to improve the initial solutions generated by the greedy algorithm.

4.5 Dynamic Programming Approach

4.5.1 Introduction

As is commonly known, dynamic programming solves problems by finding solutions to sub-problems appearing in the wider problem, given these sub-problems can be solved independently and their solutions then aggregated to find the optimal solution.

For the TSP, the algorithm starts at any city and works its way up by considering every possible ordering of each possible size of subset, it chooses the best one, and moves up to add a new city to the tour.

4.5.2 Algorithm

The algorithm is explained in the pseudo-code block 6, partially inspired by the work done by Bouman et. al in [4].

4.5.3 Results Analysis

It is possible to obtain the optimal cost with this algorithm, but the running time increases exponentially with each city added to the problem. However, for the computing power available for this project, this was not the main issue. As we can see in figure 4.10, the running times of the algorithm were generally low and as we will see in the comparison of algorithms they can drop even lower with a newer machine. The main issue was the memory usage; it was not possible to calculate any problem beyond 22 cities due to a lack of random access memory to store all the possible subsets and their respective costs. It is also interesting how the results are the same for the one atsp problem in this range for which we had the optimum cost in the

Algorithm 6 Dynamic Programming Approach for TSP

Input - Distance Matrix $distance\_matrix$
Output - Minimum Cost $cost$ and Path Sequence $route$

Initialize $first\_city$ as an arbitrarily chosen city from V
Initialize dictionaries $Distances$ and $Paths$
Initialize list $route$
for $u \in V$ **do**
    $Distances[(\{u\}])) \leftarrow distance(u, first\_city)$
for $i = 2, ..., |V|$ **do**
    for $S \subseteq V$ where $|S| = i$ **do**
        for $w \in S$ **do**
            $Distances[(S, w)] \leftarrow \inf$
            for $u \in S \setminus \{w\}$ **do**
                $dist \leftarrow Distances[(S \setminus \{w\}, u)] + distance(u, w)$
                if $dist \leq Distances[(S, w)]$ **then**
                    $Distances[(S, w)] \leftarrow dist$
                    $Paths[(S, w)] \leftarrow u$
$full\_set \leftarrow V \setminus \{first\_city\}$
$min\_cost \leftarrow \inf$
for $w \in V$ **do**
    $cost \leftarrow Distances[(full\_set, w)] + dist(w, first\_city)$
    if $cost \leq min\_cost$ **then**
        $min\_cost \leftarrow cost$
        $final\_city \leftarrow w$
$current\_city \leftarrow final\_city$
$current\_set \leftarrow full\_set$
while $|route| \leq |V|$ **do**
    $route \leftarrow route + current\_city$
    $previous\_city \leftarrow Paths[(current\_set, current\_city)]$
    $current\_set \leftarrow current\_set \setminus \{current\_city\}$
    $current\_city \leftarrow previous\_city$
$route \leftarrow route + current\_city$
$route \leftarrow route + first\_city$
return $route$, min_cost

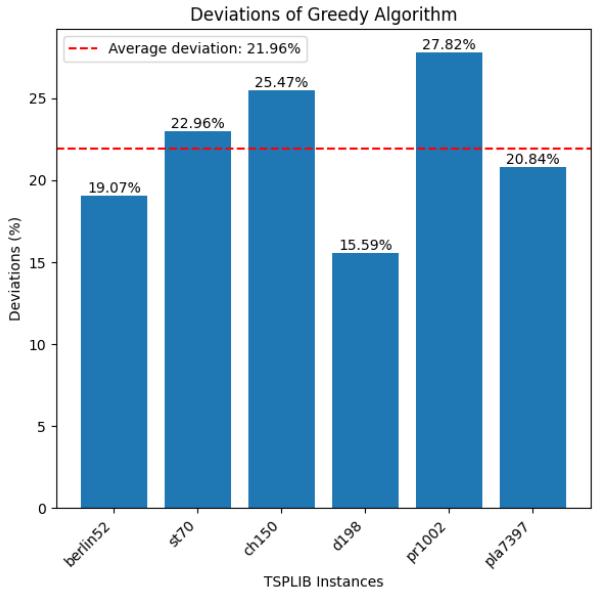


Figure 4.7: Deviation of Greedy algorithm from optimal solution

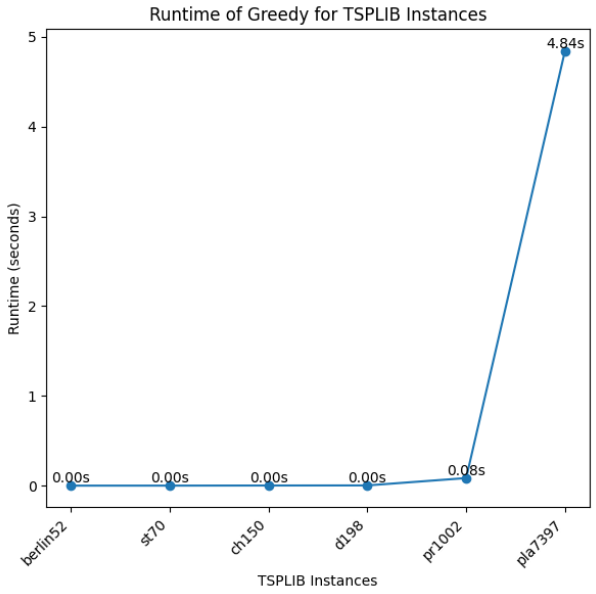
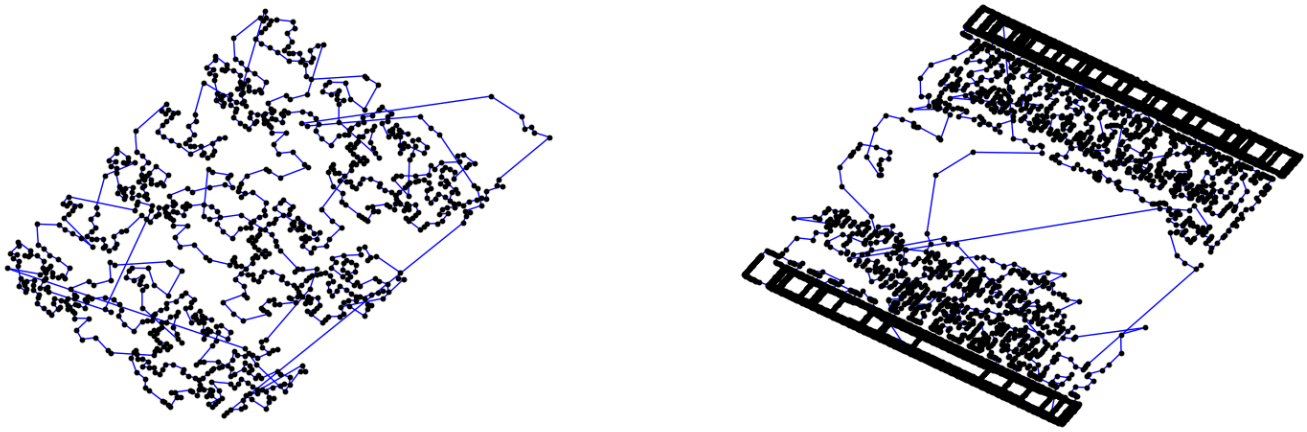


Figure 4.8: Runtime of Greedy algorithm



(a) pr1002

(b) pla7397

Figure 4.9: Visualization of solution given by greedy algorithm

TSPLIB; we can intuit that the driving force between the time and memory usage is simply the number of cities to solve.

4.5.4 Limitations

As discussed, although giving the optimal every time the dynamic programming approach cannot yet be relied upon to be used in problems with a big number of cities because of memory limitations. In 1962 Bellman [5] asserted that problems with more than 21 cities were beyond the reach of the computing at the time, and we found that number to still ring true for our, admittedly limited personal computers. In this work it is suggested that modifying the storage technique would alleviate this limitation and increase the number of cities that can be solved; but this is beyond the scope of this report.

4.6 Randomized Approach: Markov Chain Monte Carlo

4.6.1 Introduction

Local search algorithms for the TSP are a class of algorithms that attempt to find a low-cost route by partially traversing the solution space. They are often deployed after a reasonable route has been found

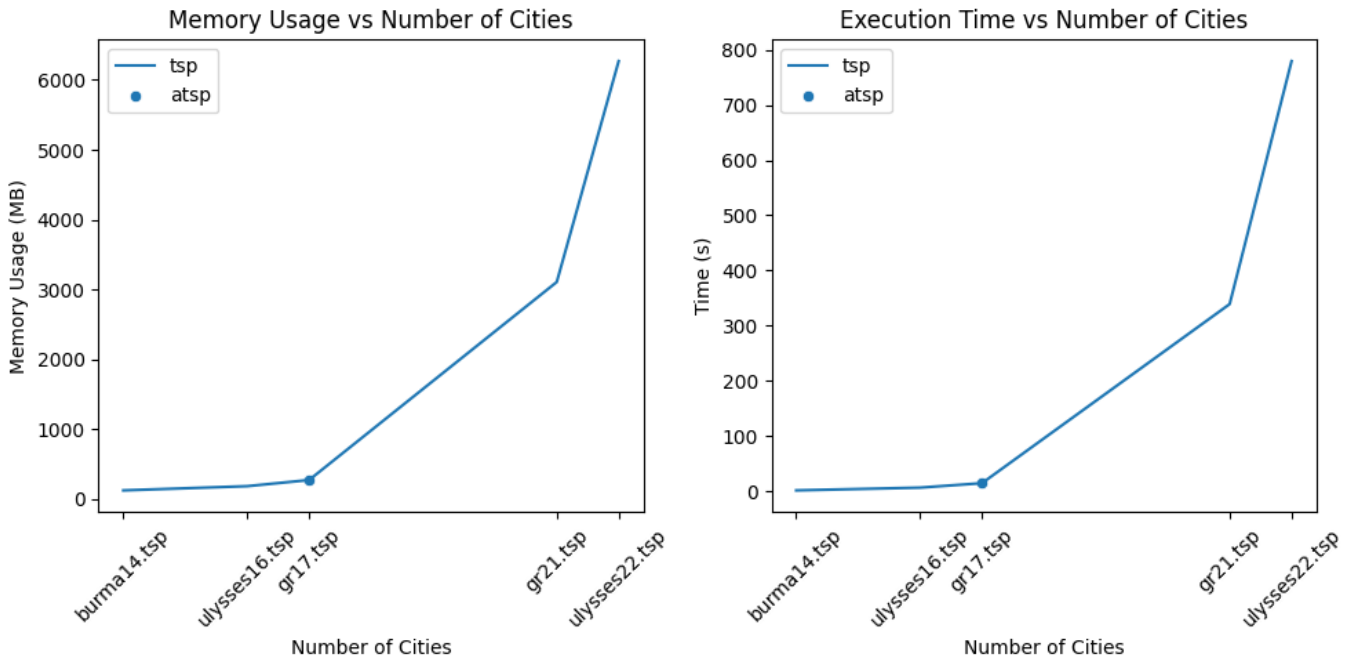


Figure 4.10: Memory Usage and Running Time when finding the optimal cost with the Dynamic Programming Approach

with methods like a greedy approach, and sequentially apply transformations to it that improve the total cost until they cannot anymore. These methods are not expected to find the optimal solution, especially because there is a possibility that even after a marked improvement from the initial proposed path they get stuck in a local minimum from which the algorithm cannot move to another region of the solution space without applying a more radical transformation to the route that would signify a bad change in the given iteration. A proposed tweak to the algorithm then is that it occasionally accepts a route with a higher cost than the one before, to explore another region of the solution space.

This method of constructing a new route by applying a transformation to the current one, instead of choosing another completely random ordering of cities for example, is reminiscent of a Markov chain. These are a sequence of events in which the probability of the next step depends only on the current one, not the ones before; and after a great number of iterations this probability will converge to a value given by the probability of transitioning to any of the possible states in the chain or the possible routes in the case of the TSP. And to have an algorithm that occasionally accepts a new route with a greater cost than the one before, a reasonable threshold of probability for this poor change of route is set and a value is randomly drawn from the uniform distribution to compare to this threshold, thus constituting an arbitrary jump to a different region of the solution space. A technique that is reminiscent of that of the Monte Carlo methods of simulation [6]. This approach also borrows from the method termed a simulated annealing, in which the more iterations pass the less likely it is that you will choose one of these bad configurations as the next step, named after the crystallization phenomena in which the temperature of a crystal will progressively lower and this will cause its atoms to settle into their final configuration.

4.6.2 Algorithm

We begin with a random ordering of our cities, and in each iteration we select a random subsection of this path and reverse it. We compare the cost of this new path with the one before and if it is lower we accept this as our new best route; if not, it is only accepted with the conditional probability given by the temperature. As explained by Martin et. al. in [7], as the temperature decreases, so does the term $e^{-\frac{\Delta}{T}}$ and it is less and less likely that a bad route will be accepted. However, to mitigate the fact that this could see the algorithm stuck in a bad path in late iterations, there is an occasional reheating of the system after 500 iterations where the temperature has changed. The detailed behavior can be seen in the pseudo-code in algorithm 7.

It is necessary then to set several parameters, the ranges of values for them have been studied for the TSP problem [8] and we have done a process of trial an error as well. We chose the following:

- The temperature T, which combined with a random value governs how often the algorithm accepts a non-favorable change in the solution by testing $random\_value \leq e^{-\frac{\Delta}{T}}$ where Δ is the difference in cost between the current best route and the one being evaluated. We set this to 1.
- The tolerance tol_T to ensure that at some point, when the temperature drops sufficiently low, the algorithm ceases to accept these more costly paths at all. Set to 0.000005.
- The number of iterations to run, in this case there is at least 5 times as many cities iterations of different routes before the temperature is lowered, which happens at most 10000 times.
- The number of iterations after which there is no longer any improvement, and the algorithm should stop. In this case, 15000 times where the temperature has been lowered.

Algorithm 7 Randomized Approach: Markov Chain Monte Carlo

Input - Distance Matrix $distance\_matrix$
Output - Minimum Cost $cost$ and Path Sequence $route$

Initialize $route$ as an arbitrarily chosen route
Initialize $dist\_best$ and $dist\_route$ as the cost for this route
Initialize T as the temperature of the system, and tol_T as a tolerance
Initialize $plateau\_counter$ and $plateau\_limit$
while $i \leq 10000$ **do**
    while $j \leq 5 * n\_cities$ **do**
        $modified\_route \leftarrow$ route with random change
        $\Delta \leftarrow cost(modified\_route) - dist\_route$
        if $(\Delta < 0)$ or $(T > tol\_T$ and $random\_value \leq e^{-\frac{\Delta}{T}})$ **then**
            $route \leftarrow modified\_route$
        $dist\_route \leftarrow cost(route)$
        if $dist\_route < dist\_best$ **then**
            $dist\_best \leftarrow dist\_route$
            $plateau\_counter \leftarrow 0$
        else
            $plateau\_counter \leftarrow plateau\_counter + 1$
    if $plateau\_counter > plateau\_limit$ **then**
        break
    $T \leftarrow 100 \times 0.9^i$
return $route$, min_cost

4.6.3 Results Analysis

To understand the behavior of the solutions given by this algorithm, it was tested against instances from the TSPLIB sized at most 150 cities; and since the random nature of it makes it so there is some degree of variation in the answer every time, we ran it 50 times for each problem, ensuring a significant sample.

We noticed that for the TSPLIB instances there were two clusters of behavior, shown in figure 4.11. For the TSP instances, the algorithm consistently delivers answers within 40% of the optimal cost; regardless of the number of cities, while for the ATSP instances the algorithm mostly gets further and further away from the answer as the number of cities increases. This could be due to the fact that when the path is reversed in the local search the new path is so different in cost to the last one that we have left the region of the solution space we were exploring and effectively started over in another one. The algorithm would take much much longer to reach a cost close to the optimum one.

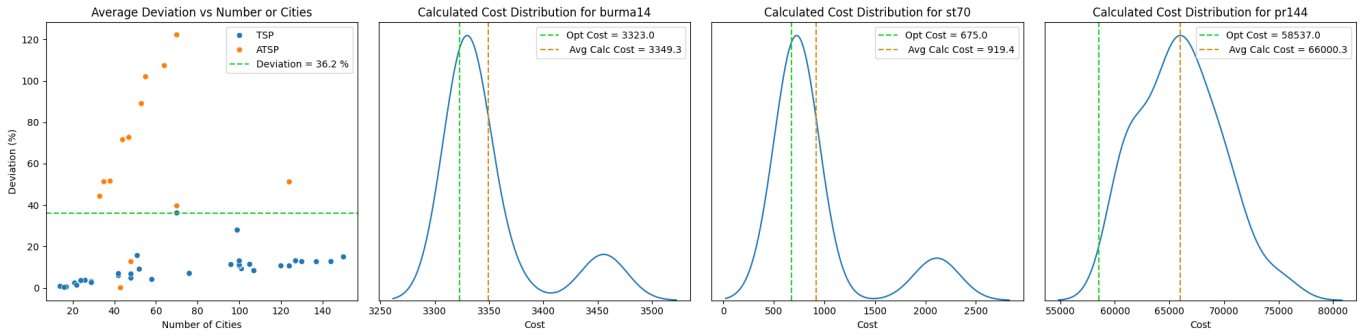


Figure 4.11: Average Deviation for Instances of the TSPLIB with the Markov Chain Monte Carlo Algorithm

4.6.4 Limitations

This algorithm has proven to deliver a good approximation and, with enough tries, even reach the optimum. However the fact that this is left to chance is not ideal. To mitigate this risk one would have to increase the number of iterations and perhaps add a few more reheating moments in the process, but this would of course make the runtime of the algorithm grow exponentially. It is also very sensitive to the chosen route transformation for the local search, but this means that it could also be very powerful if combined with a well-studied process like 2-opt or 3-opt that ensure a methodical traversing of the solution space.

4.7 Genetic Programming

4.7.1 Introduction

Genetic algorithm uses a stochastic approach for solving optimization problem by randomly searching for good solutions. These approaches are based on analogy of natural systems to build solutions, ensuring better solution than completely random search. Its operation is based on the principle of 'survival of the fittest', which implies that the fitter individuals are tend to survive and have better chance of passing their good genetic to the next generation. In genetic algorithm, each individual (chromosome) that is a member of population represents a potential solution to the problem. In TSP, the path (or route) is represented by chromosome.

4.7.2 Algorithm

First, we need to set GA parameters. The parameters defined in the implementation are population size, maximum number of generation, crossover rate and mutation rate. The first step is to generate initial random population. Then, each chromosome (route) in the population is evaluated by a fitness (cost). After evaluation, we select individuals from the population to become parents for the next generation by performing Tournament Selection (check 9). Stronger individuals (high fitness score) has a higher chance to be selected. Then, we perform Order Crossover (check 10) to combine the genetic of parents (which is considered to be good) to produce a new child. Finally, we perform Reverse Mutation (check 11). Its purpose is to introduce randomness in the offspring to prevent premature convergence and avoid getting stuck at local optima. Now a new population is created and this process repeat n times (n is the maximum number of generation that we set). In the end, we get the best chromosome (route) with the best fitness (cost) for TSP. The pseudo-code for this algorithm is shown in algorithm 8.

4.7.3 Result Analysis

One of the main difficulties in building a practical genetic algorithm is in choosing suitable values for parameters size its performance varies vastly according to the parameters. As mentioned above, the parameters chosen in the implementation are population size, generation threshold, mutation rate and crossover rate. The algorithm is tested on the 3 instances generated by random problem generator (10, 30, 50 cities) and 3 instances in TSPLIB (burma14, bayg29, eil51).

For now, let fix the population size at 200. The range of generation threshold is set according to the instance size. For example, for a small instance (small number of cities) as in figure 4.12a, the range for generation

Algorithm 8 Genetic Algorithm for TSP

Input - $distance\_matrix$, $hyperparams$ (optional)
Output - $cost$, seq

Set default parameters
$nb\_cities \leftarrow$ length of $distance\_matrix$, $best\_fitness \leftarrow \emptyset$
Initialize $population$
for $gen \leftarrow 1$ to GEN_THRESH **do**
    Sort $population$ by fitness
    $current\_best\_fitness \leftarrow$ fitness of best individual
    Append $current\_best\_fitness$ to $best\_fitness$
    $selected\_population \leftarrow$ **perform selection**
    $new\_population \leftarrow \emptyset$
    for $i \leftarrow 0$ to size of $selected\_population$ **do**
        $parent1 \leftarrow selected\_population[i]$
        $parent2 \leftarrow selected\_population[i+1]$
        if $random\_number < crossover\_rate$ **then**
            $child \leftarrow$ **perform crossover**
            Add $child$ to $new\_population$
    Add $selected\_population$ to $new\_population$
    Sort $new\_population$ by fitness
    $new\_population \leftarrow$ **perform mutation**
    $population \leftarrow new\_population$
$best\_individual \leftarrow$ individual with best fitness in $population$
$cost \leftarrow$ fitness of $best\_individual$
$seq \leftarrow$ path of $best\_individual$
return $cost$, seq

Algorithm 9 Genetic Tournament Selection

Input - $population, tournament\_size$
Output - $selected\_population$

$fitness\_list \leftarrow$ get $fitness$ of each $individual$ in $population$
$selected\_population \leftarrow [\,]$
for $individual$ in $population$ **do**
    $competitors \leftarrow$ select $tournament\_size$ random individuals from $population$
    $winner \leftarrow$ select $individual$ with the best (lowest) $fitness$
    append $winner$ to $selected\_population$
return $selected\_population$

Algorithm 10 Genetic Order Crossover (OX1)

Input - $parent1, parent2, distance\_matrix$
Output - $child$

$chrom\_size \leftarrow$ size of $chrom$ in $parent1$
$child\_chrom \leftarrow$ empty $chrom$ of size $chrom\_size$ filled with value -1
$start, end \leftarrow$ select random pos in $range(1, chrom\_size)$ and sort them
copy the genes from $parent1$ between $start$ and end into corres. pos of $child\_chrom$
$unused\_genes \leftarrow$ from $parent2$ that are NOT in $child\_chrom$
insert genes from $unused\_genes$ into empty pos of $child\_chrom$ from L to R
append starting city to the end of $child\_chrom$
$child\_fitness \leftarrow calulate\_fitness(child\_chrom, distance\_matrix)$
return $child\_chrom, child\_fitness$ as $child$

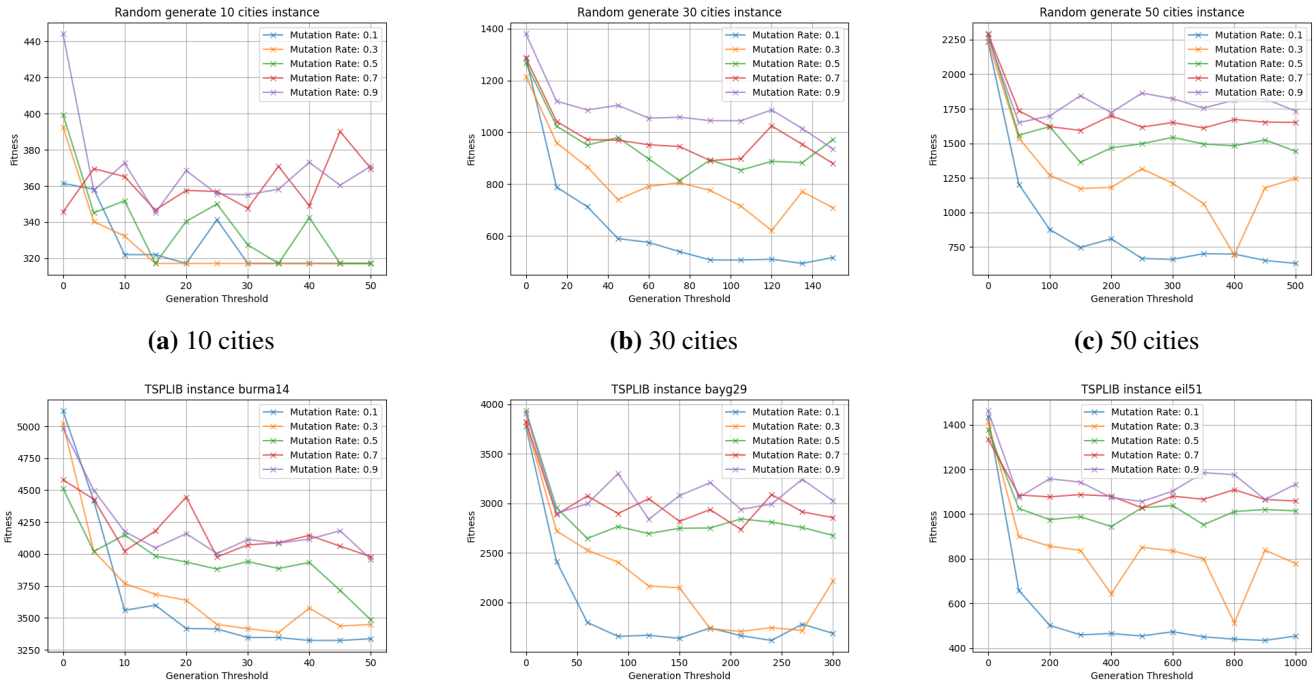
Algorithm 11 Genetic Reverse Mutation

Input - $population, mutation\_rate, distance\_matrix$
Output - $mutated\_population$

for $individual$ in $population$ **do**
    if $random\_prob() < mutation\_rate$ **then**
        $pos1, pos2 \leftarrow$ select 2 random indices from $individual.chrom$
        ensure $pos1 < pos2$ to define a valid range
        reverse the sub-segment of $chrom$ from $pos1$ to $pos2$
        update $individual.chrom$ with the modified $chrom$
        $individual\_fitness \leftarrow calulate\_fitness(individual\_chrom, distance\_matrix)$
        update $individual.fitness$ with $individual\_fitness$
return $mutated\_population$

threshold is set from 0 to 50, and in larger instances, as in figure 4.12c, the range is set from 0 to 500. This is necessary because if the range is insufficient, the result produced by genetic can be stuck in local minima and cannot converge.

After choosing the proper range for the generation threshold, let us choose the mutation rate. To decide the best mutation rate for such small-sized instances, we can analyze the performance of genetic by plotting its fitness scores (y-axis) on a range of generation threshold (x-axis). For now, we are not concerned about the crossover rate yet, thus set it to any value (let say 0.7). The results are the lines correspond to each mutation rate in that range 0.1 to 0.9. The fitness scores are the cost of sequences produced by genetic, thus we want to minimize this value. The result in figure 4.12 implies that, in both random generated and TSPLIB instances, a smaller mutation rate tends to produce a solution with a smaller fitness score. Mutation rates from 0.3 or above generally make genetic solutions stuck in local minima. On the contrary, the mutation rate at 0.1 makes genetic solutions converge in all 6 instances. Thus, we selected a mutation rate of 0.1.



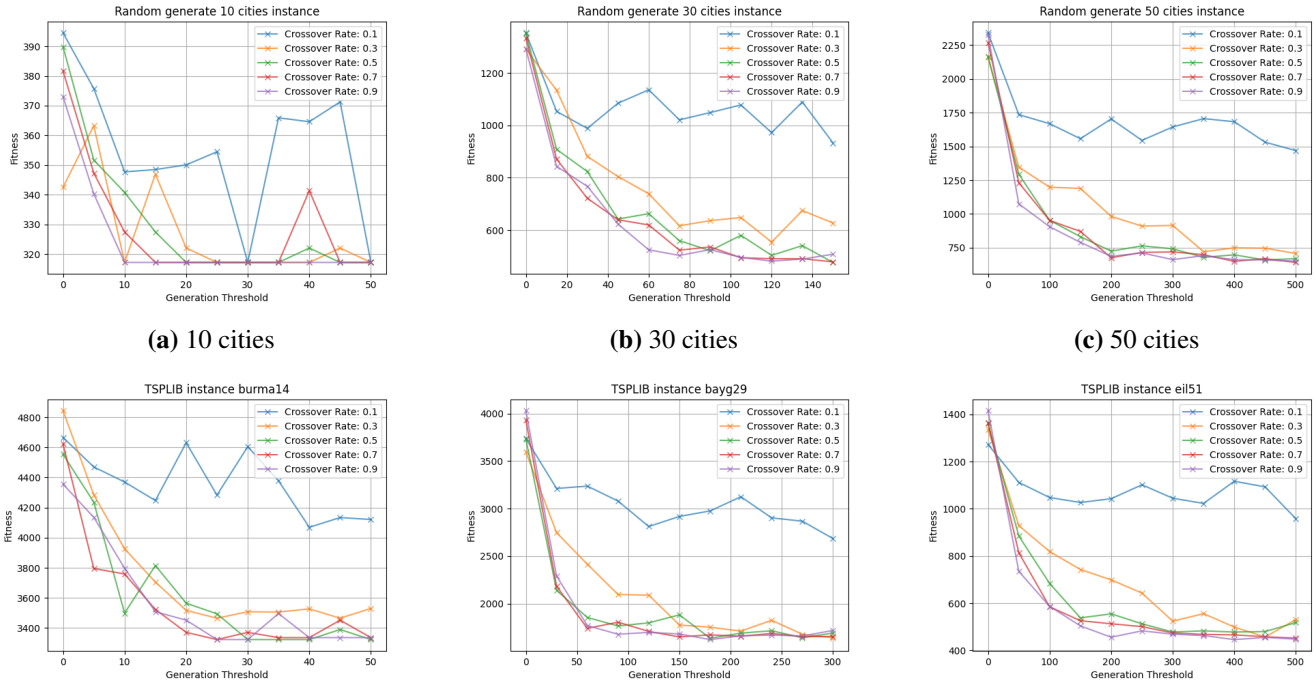
(a) 10 cities **(b)** 30 cities **(c)** 50 cities

(d) burma14 **(e)** bayg29 **(f)** eil51

Figure 4.12: Genetic performance with different mutation rate

Let's redo the same experiments, but instead of doing it with a range of mutation rate and fixed crossover rate, we fix the mutation rate at 0.1 (the best value that we concluded above), and perform genetic on a range of crossover rate from 0.1 to 0.9. The plot in figure 4.14 illustrates that the crossover rate at 0.1 produces worse fitness scores, and it does not converge. Other crossover rates from 0.3 to 0.9 perform better, produce better fitness scores and reach convergence. However, it is obvious that the crossover rate at 0.9 produces slightly better fitness scores and converge faster than others. Thus, we selected the crossover rate of 0.9.

As previously stated, the generation threshold is chosen based on the instance size, and we have not discussed the population size. As shown in the graph 4.14a and graph 4.14b, genetic produced better results with higher generation threshold and population size. Indeed, with a small-sized instance as burma14, there is no problem. However, for bayg29 and eil51, it is clear that these 2 parameters have a strong impact on the performance of genetic. For example, in figure 4.14a, the deviation if the generation threshold is 100 for eil51 is significant, at 48.83%. For population size, if we selected 100, the deviation is 12.68%, also the largest deviation. Thus, we can conclude that it would be preferable to use a large generation threshold and population size. However, this represents a trade-off, as shown in graph 4.14c; a larger generation threshold and population size make genetic computation time sharply increase.

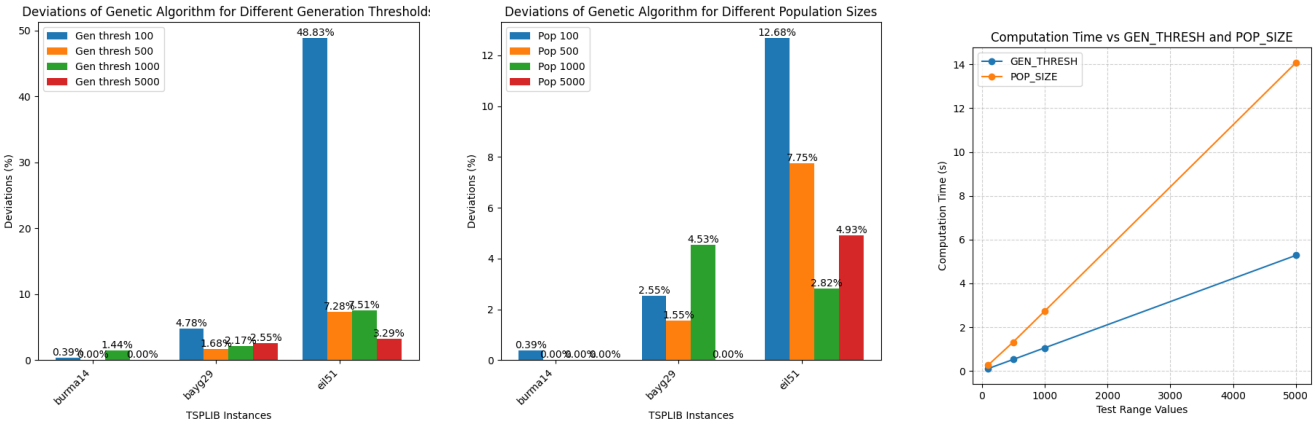


(a) 10 cities **(b)** 30 cities **(c)** 50 cities

(d) burma14 **(e)** bayg29 **(f)** eil51

Figure 4.13: Genetic performance with different crossover rate

Another notable observation is that, for genetic to produce better solution, increasing generation threshold is more favorable than population size. First, just by increase the gen thresh from 100 to 500, in graph 4.14a, the deviation drastically dropped from 48.83% to 7.28%. We can safely conclude that 500 is the acceptable value for the generation threshold, since even if we increase it, the deviation does not decrease, and we have a trade-off for computation time. On the other hand, if we increase the population size from 100 to 500, the deviation drops slightly from 12.68% to 7.75%, much less efficient than adjusting generation threshold. Moreover, the computation time trade-off for increasing population size is significantly larger than the generation threshold, as shown in graph 4.14c.



(a) vs Generation threshold **(b)** vs Population size **(c)** Computation Time

Figure 4.14: Deviation of genetic solution from optimal solution

4.8 Ant Colony Approach

4.8.1 Introduction

Ant Colony Optimisation (ACO) is a bio-inspired algorithm that aims to solve the TSP problem by implementing an algorithm inspired by real-life ant behaviour [9]. The algorithm begins by initializing a set of ants to random tours of the instance. After the first iteration, the ants deposit pheromones on the edges they have traversed. The amount of pheromones they deposit is determined by the length of the tour. The shorter the tour, the more pheromones dropped on each edge of that tour, allowing good solutions to reinforce themselves over time. In subsequent iterations, ants choose their next city from the set of unvisited cities, favoring those connected by edges with higher pheromone levels. After each iteration, the pheromones are multiplied by an evaporation rate, ρ, which helps to prevent the algorithm from getting stuck in a local optima. This process is repeated until convergence or a maximum number of iterations has been reached. A trade-off has to be managed between exploitation and exploration using two parameters, α and β respectively. Exploitation describes how heavily weighted the pheromones are in the ant's decision, while exploration describes the influence of the edge weight (Euclidean distance in our case).

4.8.2 Algorithm

NumPy arrays were utilized wherever possible, particularly for operations involving the probability matrix calculations. This allowed for vectorized computations, ensuring the algorithm performance scales well for large problems.

4.8.3 Parameter Tuning

Figure 4.15 shows the minimum tour length found at each iteration for various problem sizes. The y-axis represents the cost relative to the greedy heuristic, where the greedy solution's cost is set to 100% and the optimal solution's cost is 0%. The algorithm was run on each problem 5 times to alleviate the effect of random initialisation.

Figure 4.16 shows a heatmap of various combinations of α and β. The combination $\alpha = 1$ and $\beta = 4$ yielded the best results over a range of 100 randomly generated TSP problems ranging in size from 10 to 100. Darker regions in the heatmap indicate lower average costs, highlighting better parameter combinations.

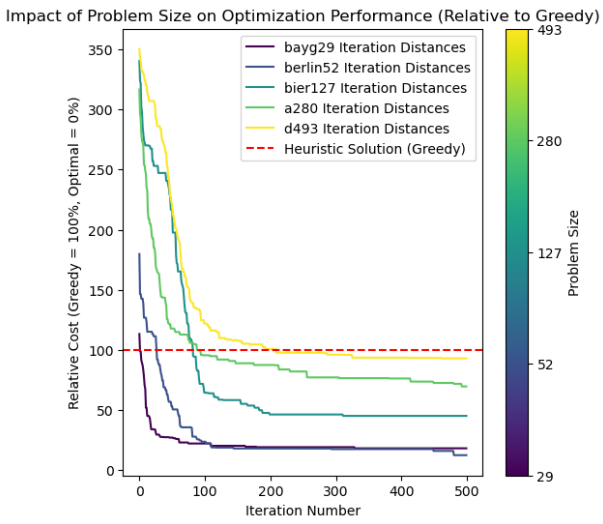


Figure 4.15: Minimum tour length found at each iteration for various problem sizes.

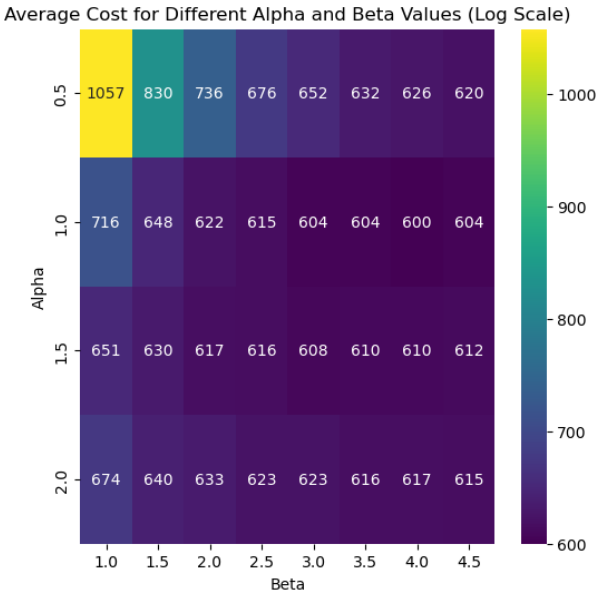


Figure 4.16: Heatmap for various combinations of α and β.

Experimental results indicated that the evaporation rate (ρ), the initial pheromone level (τ), and the number

Algorithm 12 ACO Algorithm for TSP

Input - Distance Matrix: D, Number of Ants: m, Iterations: T, Evaporation Rate: ρ,
Alpha: α (pheromone influence), Beta: β (heuristic influence), Initial Pheromone: τ_0
Output - Best Tour: $tour\_best$, Minimum Cost: $cost\_best$

$n \leftarrow$ number of cities (size of D)
Initialize pheromone matrix: $\tau_{ij} \leftarrow \tau_0$ for all edges (i, j)
$tour_{best} \leftarrow \emptyset$
$cost_{best} \leftarrow \infty$
for $t \leftarrow 1$ **to** T **do**
    Initialize tours for each ant: $tour_k \leftarrow$ empty tour for $k = 1, 2, ..., m$
    for $k \leftarrow 1$ **to** m **do** ▷ For each ant
        Randomly choose a starting city s
        $tour_k \leftarrow [s]$
        $unvisited \leftarrow \{1, 2, ..., n\} \setminus \{s\}$
        while $unvisited \neq \emptyset$ **do**
            $i \leftarrow$ last city in $tour_k$
            Calculate probabilities P_{ij} for all $j \in unvisited$:
            $P_{ij} = \frac{(\tau_{ij})^\alpha \cdot (\eta_{ij})^\beta}{\sum_{l \in unvisited} (\tau_{il})^\alpha \cdot (\eta_{il})^\beta}$ where $\eta_{ij} = 1/D_{ij}$ (heuristic)
            $prob_{selection} \leftarrow$ randomly select a number between 0 and 1
            Select next city j based on first unvisited city j with $\sum_{l=0}^{j} P_{il} \geq prob_{selection}$
            $tour_k \leftarrow tour_k + [j]$
            $unvisited \leftarrow unvisited \setminus \{j\}$
        $tour_k \leftarrow tour_k + [tour_k[0]]$ ▷ Complete the tour
    Calculate tour lengths L_k for each $tour_k$
    for $k \leftarrow 1$ **to** m **do**
        if $L_k < cost_{best}$ **then**
            $cost_{best} \leftarrow L_k$
            $tour_{best} \leftarrow tour_k$
    Update pheromones:
    $\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij}$ for all edges (i, j) ▷ Evaporation
    for $k \leftarrow 1$ **to** m **do**
        for each edge (i, j) in $tour_k$ **do**
            $\Delta\tau_{ij}^k = Q/L_k$ ▷ Q is a constant (set to 10)
            $\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^k$
return $tour_{best}$, $cost_{best}$

of ants had a minimal impact on the algorithm's performance. Increasing the number of ants primarily increased the execution time without significant improvements in solution quality. Consequently, the values chosen were $\rho = 0.1$, $\tau = 0.1$, and the number of ants was set to 10. To evaluate the effect of each of these parameter values, 10 different TSP instances were randomly generated for a range of problem sizes from 10 to 100 cities. The algorithm was then run on each instance using different numbers of ants (also ranging from 10 to 100), and the resulting average distances are presented in Figure 4.17. Similar experiments were conducted for ρ and τ, but the results showed analogous trends: minimal impact on solution quality and execution time. Due to space constraints in this report and the similar nature of these findings, the corresponding graphs for ρ and τ are included in the annexes.

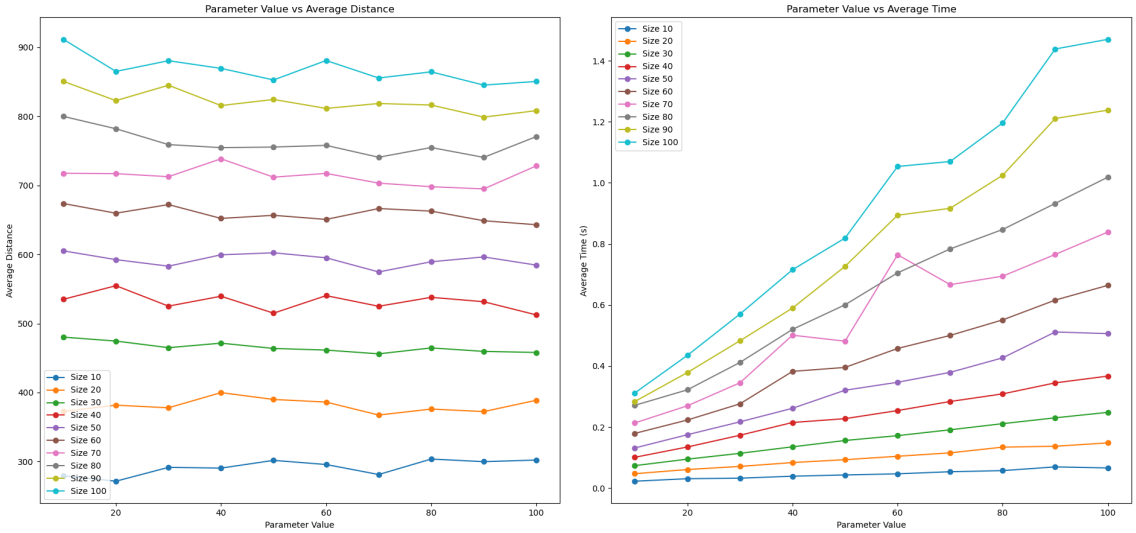


Figure 4.17: ACO Performance vs Number of Ants

4.8.4 Results Analysis

This section analyzes the performance of the Ant Colony Optimization (ACO) algorithm on both symmetric (TSP) and asymmetric (ATSP) TSPs. Test instances were drawn from the TSPLIB benchmark library, and randomly generated problems were also used to evaluate the algorithm's performance and scalability. The ACO algorithm was implemented with the following parameters: number of ants (m) = 10, number of iterations (T) = 100, evaporation rate (ρ) = 0.1, (α) = 1, and heuristic importance (β) = 4. These were chosen based on the parameter tuning section above to minimise execution time and deviation from optimal cost for TSP.

Figure 4.18 shows the convergence behavior of ACO on the 'bier127' TSP instance. The figure plots the best tour length found up to each iteration and the average tour length across all ants in each iteration. The best tour length continues to improve until approximately iteration 450, which suggests ACO is still improving even in later iterations. However, the average tour length stabilizes around iteration 170. This could be explained by the ants getting stuck in a local optimum, which is potentially explained by the low pheromone evaporation rate and a poor initialisation. Despite getting trapped, the continued improvement of the best tour length suggests that the algorithm is still effectively exploring the solution space. Notably, the best tour found by ACO approaches the known optimal solution, and it surpasses the greedy solution after approximately 100 iterations.

Figure 4.19 shows the execution time of ACO as a function of problem size. For each problem size, 30 randomly generated instances were solved, and the average execution time is plotted. The theoretical time complexity of ACO is $O(T \times m \times n^2)$, where T is the number of iterations, m is the number of ants, and n is the number of cities. For each iteration, every ant has to build a tour from $n - 1$ cities. Building the tour involves evaluating all $n - 1$ cities in the worst case. With the fixed values of T (100) and m (10), this

simplifies to $O(n^2)$. Without a reference on the graph, this looks to be roughly reflected by the trend of the graph. This efficient scaling allows ACO to tackle relatively large problem instances within a reasonable timeframe.

Figure 4.20 shows the performance of ACO for both TSP and ATSP instances from TSPLIB, measured as the percentage deviation from the known optimal solutions. Looking at the TSP instances, a clear trend can be seen: the deviation increases as the problem size increases. This is expected given the fixed number of iterations. Larger problems require more iterations for ACO to converge to near-optimal solutions. For ATSP instances, the trend is more complex. While the average deviation is generally higher than for TSP, there is significant variability. This could suggest the set of instances chosen vary in terms of structure, some of which may be more difficult for ACO given the chosen parameters. For instance, the degree of asymmetry or the distribution of edge weights could influence the algorithm. It is possible tuning the values of α and β again for ATSP problems would improve the performance of ACO on ATSP instances.

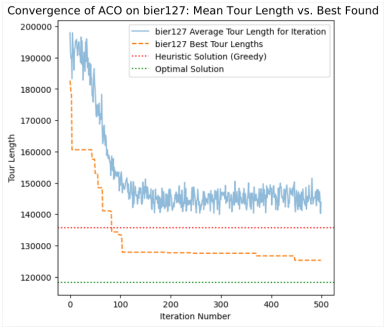


Figure 4.18: Average tour length for 500 iterations vs minimum tour length found so far relative to Greedy and Optimal.

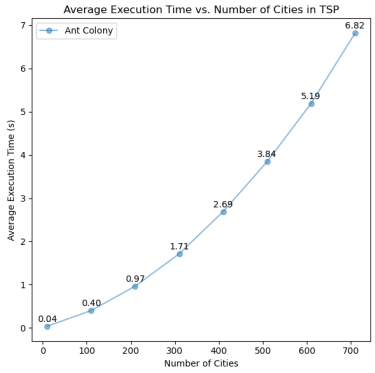


Figure 4.19: Execution Time as Problem Size Increases.

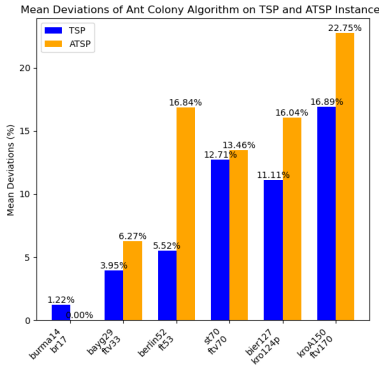


Figure 4.20: Deviation from Optimal Solution for TSP and ATSP Instances from TSPLIB.

In summary, the results demonstrate that ACO is an effective algorithm for solving both TSP and ATSP instances, particularly for smaller to medium-sized problems. While the algorithm's performance degrades with increasing problem size, its scalability allows it to handle relatively large instances. Future work could focus on adaptive parameter tuning, particularly for α and β, and exploring alternative pheromone update strategies to improve performance on larger and more complex instances.

4.9 Lin-Kernighan

4.9.1 Introduction

Lin-Kernighan - a sophisticated heuristic algorithm for solving the TSP, developed in 1973. It performs a variable-depth search by making a sequence of edge exchanges of different types (k-opt moves) to improve the tour. [10] The implemented algorithm can't be called an original, it is more an adaptive version of 2-opt with concepts from the Lin-Kernighan. The true LK algorithm doesn't limit itself to a single 2-opt move at a time. Instead, it systematically considers sequences of up to k edges to remove and another k edges to add, known as a k-exchange. In particular, we limit the search depth, perform periodic random restarts, and use forbidden edges to encourage escape from local optimum and exploration of new regions of the solution space. This approach is simplified compared to the original LK, but aims to maintain a balance between implementation complexity and improved results, allowing for high-quality solutions at a moderate cost of computational resources.

4.9.2 Algorithm

The first step is to randomly choose the initial tour and find its cost. This is followed by the main solution search loop with iterations from 1 to 1000. Each iteration is one complete attempt to improve the route.

At each iteration, the list of forbidden edges (tabu_edges) is cleared - edges that we have already tried to change, and up to 5 improvement attempts are performed (nested loop).

If after all improvement attempts in the current iteration no better solution was found (`improved = False`), the algorithm was set to a 10% chance of performing a complete restart (random restart). When restarting, a completely new random route is created, which allows the algorithm to "escape" from the local optimum and explore other parts of the solution space.

1000 iterations were chosen empirically, but this hyperparameter can be changed depending on the size of the tour. Since we typically use between 14 and 150 cities (for testing), in this case 1000 was chosen as the optimal value, providing a good balance between solution quality and execution time. Similarly, the limit of 5 improvement attempts per iteration is also an empirically determined trade-off - a larger number of attempts significantly increases the running time, and a smaller number may not provide enough opportunities to improve the solution.

Finding the best two edges to replace in the current route is a key element of the algorithm. This is implemented through a system of nested loops, where the outer loop selects the first edge (between cities `city1` and `city2`) and the inner loop selects the second edge (between cities `city3` and `city4`) for potential exchange. The algorithm methodically iterates over all possible combinations of edges, skipping forbidden and adjacent edges, to find the pair whose replacement will result in the greatest improvement in the total length of the route.

The formula for calculating the gain when replacing edges:

$$\textit{gain} = (d_{t_1,t_2} + d_{t_3,t_4}) - (d_{t_1,t_3} + d_{t_2,t_4})$$

where:

- $d_{i,j}$ - distance between cities i and j from the distance matrix,
- (t_1, t_2) and (t_3, t_4) - edges that are removed,
- (t_1, t_3) and (t_2, t_4) - new edges, that we add.

A positive gain value means that the new edge configuration will be shorter than the current one.

So, after finding the "improvement" via `find_best_improvement()` and checking that *gain* > 0, the actual route replacement is performed using the 2-opt operation. This operation removes two edges from the current route (in the example AB and DC), unrolls the part of the route between these edges, and creates two new edges to connect (AC and BD).

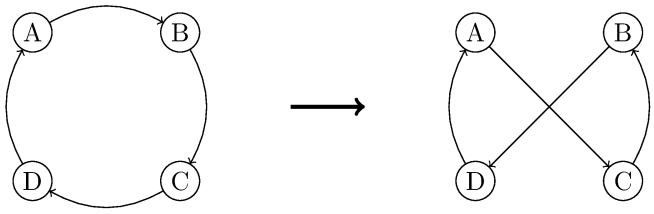


Figure 4.21: 2-opt operation: original route with intersection (left) transformed into an improved route without intersection (right).

That is, if the original route was $A \to B \to C \to D \to A$ (has an intersection), then after applying 2-opt we get $A \to C \to B \to D \to A$, where the intersection is removed. This operation is key to local search in the traveling salesman problem, as it allows us to avoid intersections in the route, which almost always leads to its improvement in Euclidean space.

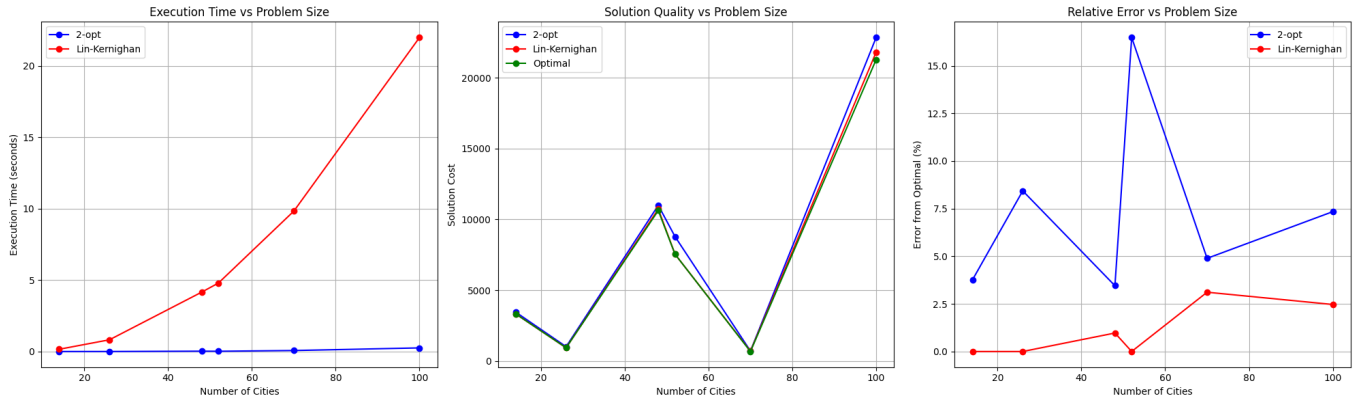


Figure 4.22: Execution Time, Solution Quality, and Relative Error Comparisons between 2-opt and Lin-Kernighan.

The time complexity of the Lin-Kernighan algorithm is $O(\textit{max\_iterations} \cdot n^2)$, where n is the number of cities. This complexity arises because at each iteration (*max_iterations*), we search for the best improvement $O(n^2)$ by checking all possible pairs of edges, and for each improvement, we perform the *two_opt* operation $O(n)$.

Although the theoretical complexity is quite high, in practice, the algorithm works more efficiently due to optimizations (tabu list, search depth limit). These optimizations make it practical for problems with up to approximately 1000 cities. However, the algorithm can become slow for larger TSP instances.

4.9.3 Results Analysis

Testing was performed on 6 standard datasets from the TSPLIB library (burma14, fri26, att48, berlin52, st70, kroA100) with sizes from 14 to 100 cities. Three types of comparative analysis were performed between the 2-opt and Lin-Kernighan algorithms: execution time, solution quality, and relative error from the optimal value. The results show that Lin-Kernighan gives better results with a smaller error (up to 3%), but has an exponential increase in execution time with increasing problem size, while 2-opt works stably fast, but with a larger error (up to 15%). For problems up to 50 cities, both algorithms show acceptable results, and for larger problems, the choice of algorithm depends on the priority between accuracy and speed.

4.9.4 Limitations

Limitations of the current implementation include: fixed search depth (5 levels) regardless of the problem size, which may be suboptimal for different matrix sizes; use of only 2-opt operations instead of k-opt as in the original Lin-Kernighan algorithm; fixed probability (0.1) for random restart.

Possible improvements include:

- Adaptive tuning of the search depth depending on the number of cities;
- Implementation of full k-opt for greater flexibility of the algorithm;
- Dynamic adjustment of the restart probability based on the search progress;
- Addition of parallel computations for large matrices;
- Introduction of adaptive tabu list parameters.

It is also worth considering the possibility of automatic tuning of hyperparameters based on the characteristics of the input problem to optimize the balance between solution quality and execution time.

Algorithm 13 Lin-Kernighan Algorithm - case of 2-opt

Input: $distance\_matrix$ (distance matrix between cities), $max\_iterations$ (maximum number of iterations)

Output: $best\_tour$ (best tour found), $best\_cost$ (cost of the best tour)

Initialization:
$current\_tour \leftarrow$ random permutation of cities
$best\_tour \leftarrow current\_tour$
$best\_cost \leftarrow calculate\_cost(best\_tour)$
for $iteration = 1$ to $max\_iterations$ **do**
    $improved \leftarrow false$
    $tabu\_edges \leftarrow$ empty set
    for $depth = 1$ to $\min(5, n)$ **do** ▷ Depth search (maximum 5 levels)
        $best\_improvement \leftarrow 0$, $best\_swap \leftarrow null$
        for each $edge1$ in $current\_tour$ **do**
            if $edge1 \in tabu\_edges$ **then**
                continue
            for each $edge2$ in $current\_tour$ **do**
                if edges_adjacent **or** $edge2 \in tabu\_edges$ **then**
                    continue
                $gain \leftarrow calculate\_gain(edge1, edge2)$
                if $gain > best\_improvement$ **then**
                    $best\_improvement \leftarrow gain$
                    $best\_swap \leftarrow (edge1\_index, edge2\_index)$
        if $(best\_swap = null)$ **or** $(best\_improvement \leq 0)$ **then**
            break
        $i, j \leftarrow best\_swap$
        if $i > j$ **then**
            Swap i and j
        $current\_tour \leftarrow current\_tour[0 : i + 1] + reverse(current\_tour[i + 1 : j + 1]) + current\_tour[j + 1 :]$ ▷ 2-opt move
        Add new edges to $tabu\_edges$
        $current\_cost \leftarrow calculate\_cost(current\_tour)$
        if $current\_cost < best\_cost$ **then**
            $best\_tour \leftarrow current\_tour$
            $best\_cost \leftarrow current\_cost$
            $improved \leftarrow true$
    if not $improved$ **and** $random() < 0.1$ **then** ▷ Random restart with 0.1 probability
        $current\_tour \leftarrow$ new random permutation
return $best\_tour, best\_cost$

CHAPTER 5. Results and Analysis

5.1 Algorithm Performance

This section compares the performance of our algorithms for solving the TSP across a range of instances. The algorithms are categorized into three groups based on their execution time: slow, medium, and fast. The slow group, comprising Brute Force, Branch and Bound, and Dynamic Programming, is impractical for instances larger than approximately 17 cities due to their exponential time complexity. The medium group, comprising Ant Colony, Genetic, Lin-Kernighan, and Markov Chain, balances between solution quality and scalability, although their performance is sensitive to parameter settings. For instance, the Ant Colony algorithm can quickly generate solutions for large problems, but the solution found improves as the number of iterations increases, also increasing the execution time. Finally, the fast group consists of Greedy and MST Approximation algorithms, which provide rapid solutions but sacrifice solution quality compared to the other groups.

Since a comparison of Branch and Bound with reduction matrix and edge selection techniques against the standard Branch and Bound algorithm is already detailed in Section 4.2, it will not be repeated here.

On all graphs below, the deviation at 0 indicates the best performing algorithm for that problem size, which is the baseline for all other algorithms. The percentage above that for algorithm j is calculated as: $\frac{Solution_j - baseline}{baseline} \times 100$.

5.2 Algorithm Comparison

As all algorithms in the slow group (brute force, branch and bound, and dynamic programming) guarantee optimal solutions, their performance is differentiated solely by execution time. As expected, brute force is the slowest, followed by branch and bound, and then dynamic programming. The exponential time complexity of these algorithms is evident in their rapidly increasing execution times as the number of cities grows.

The ”medium” group, which includes Lin-Kernighan, genetic, ant colony, and Markov Chain algorithms, has some more variety in terms of performance and execution time. To manage execution time, the Lin-Kernighan iterations were limited to 400, and for the genetic algorithm, the generation and population thresholds were set to 500 and 100, respectively.

These modifications significantly impacted the genetic algorithm's solution quality, leading to a high deviation from the optimal cost above problems of size 30. Lin-Kernighan, ant colony and Markov Chain all performed similarly across all problem sizes, with Markov Chain performing the best overall.

The ant colony algorithm's execution time remained relatively low and increased gradually with the number of cities. Genetic and Markov Chain show similar growth rates, whereas Lin-Kernighan's execution time scaled more steeply.

Comparing the fast group with the best performing algorithm overall from the medium group (ant colony) in Figure 5.3, we find that the deviation MST (around 20 to 25%) is slightly higher than that of greedy (around 10 to 20%). A solution that is only 10 to 20% worse than the best performing algorithm for large problems is very effective and can be very when a heuristic solution is required to compare an algorithm to.

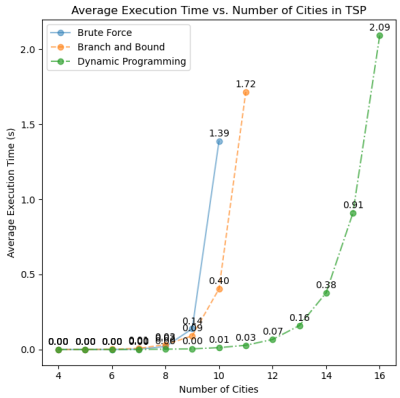


Figure 5.1: Slow Group Execution Time Comparison.

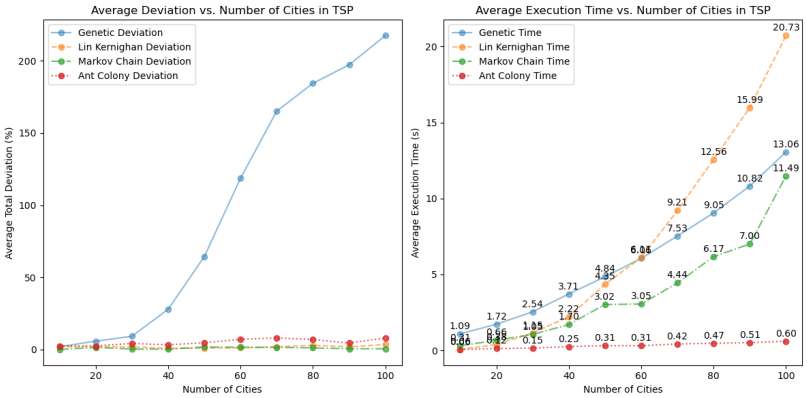


Figure 5.2: Medium Group Deviation from Optimal and Execution Time Comparison.

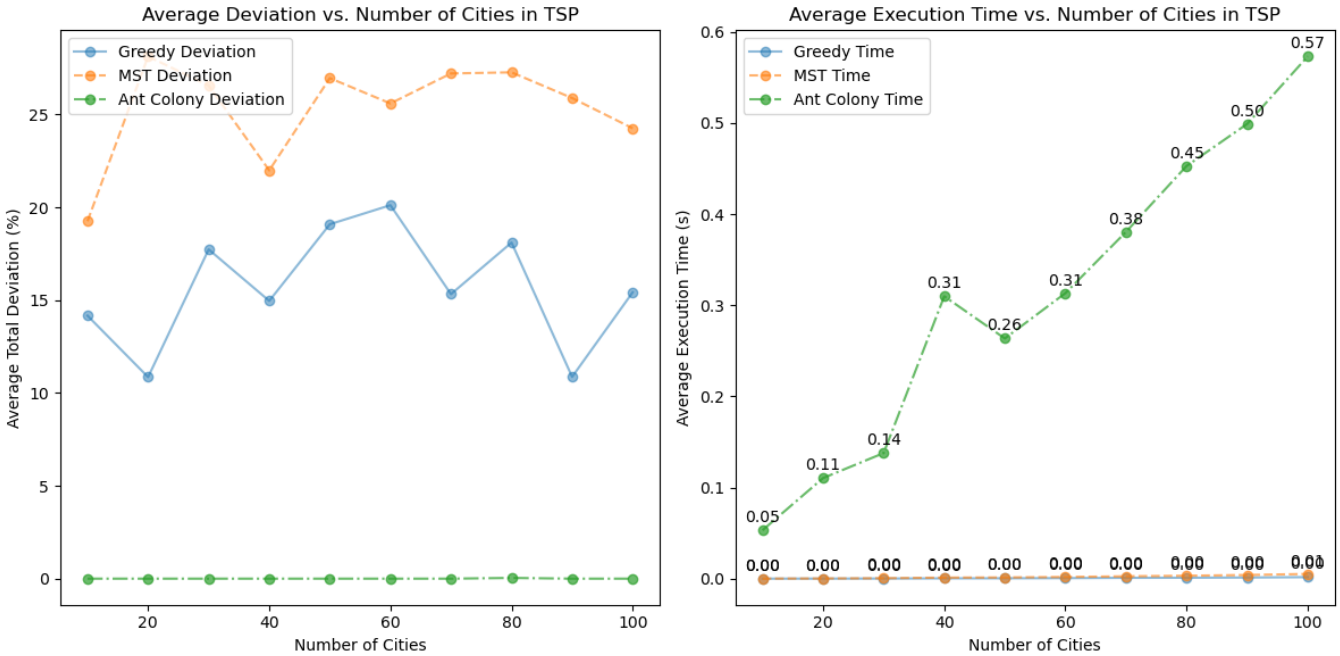


Figure 5.3: Fast Group Deviation from Optimal and Execution Time Comparison.

This problem has given rise to a field of computer science which studies the time and space complexity of algorithms. Of particular interest in this field are algorithms which can run in polynomial steps dependent on its input size. However, these 'fast' algorithms do not always give an optimal solution.

6.1 Challenges and Limitations

While implementing code and during execution phase,certain challenges were encountered.In case of genetic and ant colony as ensuring the hyper-parameters tuning ,and choosing good Parameters for a variety of problem sizes was difficult .In case of randomised and dynamic limitations included it's computing power and memory usage. On the other hand for branch and bound and brute force the execution time with respect to tsp instance took large amount of time and memory space.The reduction matrix algorithm for branch and bound also included matrix manipulation memory space which made it's efficiency a bit slow.

6.2 Algorithm Efficiency

To study the trade-off between efficiency and accuracy, we apply ten algorithms to the famous Traveling Salesman Problem and calculated runtimes for time complexity and average deviation from optimal for certain efficiently working algorithms.We chose to focus on Computation time more than storage requirements. We found Greedy Algorithm produce a solution quickly,but are among the first to diverge from an optimal solution. We also find that algorithms which have exponential theoretical complexities take a long time on a relatively low amount of cities. However, they do converge to optimal solutions.The Ant Colony,Lin Kernighan and Markov Chain works well for more number of cities and also converges to optimal solution,the deviation is seen in Genetic Algorithm.

6.3 Improvement Opportunities

In the future, it would be interesting to investigate further why the algorithms we chose could not effectively compute solutions on sparse matrices. We may need to tweak our code to account for disconnectivity in networks. It would also be interesting to study the effect of parallel computing on run-times for algorithms which can be distributed. Finally, we would like to explore other distributions to generate artificial networks, especially ones which mimic closely the distances between real-world cities.

CHAPTER 7. Contribution

7.1 Patrick Barry

7.1.1 Code

1. Implemented the Brute Force and Ant Colony algorithms.
2. Developed functions to plot deviation and execution time together, generate random TSPs and ATSPs.

7.1.2 Report

1. Wrote chapter 5.
2. Wrote the Brute Force and Ant Colony algorithms in chapter 4.
3. Wrote the random problem generator section in chapter 3.

7.2 Buu Dinh Ha

7.2.1 Code

1. Implemented the Greedy and Genetic algorithms.
2. Developed functions to plot deviation, visualize problems.

7.2.2 Report

1. Wrote chapter 1, 2, 3.
2. Wrote the Greedy and Genetic algorithm sections in chapter 4.

7.3 Mahima Haridasan Sumathy

7.3.1 Code

1. Implemented the Branch and Bound algorithm and the variation Matrix Reduction and Edge Selection.

7.3.2 Report

1. Wrote the Branch and Bound algorithms section in chapter 4.
2. Wrote Chapter 6.

7.4 Chelsy Mena

7.4.1 Code

1. Implemented the Dynamic Programming and Markov Chain Monte Carlo algorithms.
2. Developed the GUI.

7.4.2 Report

1. Wrote the Dynamic Programming and Markov Chain Monte Carlo algorithms sections in chapter 4.
2. Wrote the GUI section in chapter 3.

7.5 Serhii Vakulenko

7.5.1 Code

1. Implemented the Lin-Kernighan and Minimum Spanning Tree algorithms.

7.5.2 Report

1. Wrote the the Lin-Kernighan and Minimum Spanning Tree section in chapter 4.

REFERENCE

[1] G. Reinelt, *TSPLIB 95 – a library of sample instances for the tsp (and related problems) from various sources and of various types*, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html, Universität Heidelberg, Institut für Informatik, 1995.

[2] D. R. Morrison **andothers**, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," *Discrete Optimization*, **jourvol** 19, **pages** 79–102, 2016, ISSN: 1572-5286. DOI: 10.1016/j.disopt.2016.01.005. **url**: https://www.sciencedirect.com/science/article/pii/S1572528616000062.

[3] E. M. Reingold, *TSP BB Edges*. Add Publisher Name Here, 1977, pdf2 version.

[4] P. Bouman, N. Agatz **and** M. Schmidt, "Dynamic programming approaches for the traveling salesman problem with drone," *Networks*, **jourvol** 72, **number** 4, **pages** 528–542, 2018.

[5] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *Journal of the ACM (JACM)*, **jourvol** 9, **number** 1, **pages** 61–63, 1962.

[6] A. Tsun, "Probability & statistics with applications to computing," 2020. **url**: https://www.alextsun.com/files\Prob_Stat_for_CS_Book.pdf.

[7] O Martin, "Large-step markov chains for the traveling salesman problem.," *Operations Research Letters*, **jourvol** 11, **pages** 219–224, 1992.

[8] S. Kirkpatrick, C. D. Gelatt Jr **and** M. P. Vecchi, "Optimization by simulated annealing," *science*, **jourvol** 220, **number** 4598, **pages** 671–680, 1983.

[9] M. Dorigo **and** L. M. Gambardella, "Ant colonies for the travelling salesman problem," *Biosystems*, **jourvol** 43, **number** 2, **pages** 73–81, 1997, ISSN: 0303-2647. DOI: https://doi.org/10.1016/S0303-2647(97)01708-5. **url**: https://www.sciencedirect.com/science/article/pii/S0303264797017085.

[10] S. Lin **and** B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, **jourvol** 21, **number** 2, **pages** 498–516, 1973.