university of groningen / faculty of science and engineering

University of Groningen

Clustered Travelling Salesman Problem

Elucidating the Effects of Clustering on the Performance of Non-Deterministic Polynomial Problems

Luke McNaughton (s3595447)

July 21, 2023

Bachelor's Thesis

To fulfill the requirements for the degree of Bachelor of Science
in Computer Science at University of Groningen
under the supervision of:
Prof. K. Bunte (University of Groningen)
and
Prof. F.F. Mohsen (University of Groningen)

Contents

Acknowledgments

I'd like to acknowledge and give credit to the guidance of my supervisor, Professor K. Bunte, for passing down her rich wisdom in the areas of study for this paper was written, as well as providing extremely valuable criticism, support, and motivation which was vital for completing this daunting task during a fairly difficult time. I'd also like to acknowledge my parents, siblings and my girlfriend for their unconditional mental and emotional support throughout the tenure of this project.

Abstract

The Travelling Salesman Problem is a categorically NP-Hard problem with many proposed solutions for Optimization. The use of Approximation is one such way of increasing the speed of the process at the cost of retrieving, at times, sub-optimal results. We want to see what the effects are with the introduction of Clustering to such a process. For this project, we would like to test what effect varying levels of Clusteredness according to multiple Clustering methods have on the Travelling Salesman Problem relative to its derivative, the Clustered Traveling Salesman Problem. We will see if the establishment of well-formed clusters has any particular effect on search performance.

Keywords: Travelling Salesman, Clustering, Clustering Tendency, P vs. NP, Optimization, Clustered Travelling Salesman

Symbols

$\boldsymbol{x} \in \mathbb{R}$	row data vector
$\mathcal{O}$	Big O notation, represents the upper bound of the time complexity of an algorithm
$X \in \mathbb{R}^{n \times d}$	Dataset of n data points, each with d features
$\sum_i^n$	Summation, denoting the addition of a sequence of numbers, from i to n
$\{C_1, C_2, \ldots, C_k\}$	Set of elements
$T \bigcup \{(p_i, p_j), (p_{i+1}, p_{j+1})\}$	Union of two sets, which represents all the elements that are in either set
$T \setminus \{S_1, S_2\}$	Removal of elements from a set
$\arg\min_{p_j \in P \setminus p_1, \ldots, p_i} f(p_i, p_j)$	Argmin, which finds the argument of the minimum value of a function

Chapter 1

Introduction

There are many famous problems in Computing which can be categorised as computationally hard, with The Travelling Salesman Problem (TSP) being one of the more well known. There are various algorithms and methodologies which may be applied to optimise problems like these to come about more efficient solutions. The purpose of the Travelling Salesman Problem is, simply put, to find the shortest path throughout a given set of points and known distances where each point is visited only once [1]. One widely adopted method for optimising this problem is with the use of Clustering, which then spawns a new variant of the problem known as the Clustered Travelling Salesman Problem (CTSP)[2]. In this variant, which was originally proposed by Chisman in 1975, the cities are grouped into clusters, and the cities are visited contiguously within the cluster before moving onto the next cluster (Chisman, 1975).

We want to evaluate the effect of varying Clusters of different shapes, sizes and densities would have on the search performance on the CTSP. This project would undertake this exploration by having an array of datasets with varying Clustering Types and Tendencies and seeing the effects of the CTSP solution relative to TSP algorithms as the Clustering Tendency of the environments increase. Our work builds upon the findings of Y. Lu, J.-K. Hao, and Q. Wu[2], who proposed a clustering approach to solve the TSP for instances with clusters. Their approach involves partitioning the cities into clusters and then finding a tour that visits each cluster in turn. We aim to extend their work by investigating the performance of these heuristics on their clustering approach. By conducting these experiments, we hope to gain a deeper understanding of the Clustered TSP and identify effective heuristic approaches for solving this challenging variant of the TSP.

To evaluate the performance of these heuristics on the Clustered TSP, we will conduct a series of experiments on benchmark instances from the literature. We will measure the quality of the solutions obtained by each heuristic by computing the ratio of the solution cost to the optimal cost found using another method known as the Held-Karp Lower bound. Additionally, we may also compare the computational time required by each heuristic to obtain a solution. We hypothesize that the performance of these heuristics on the clustered variant of the problem will be influenced by the number of clusters and the size of the clusters. To assess the effectiveness of the proposed algorithms, we will compare their performance with that of the standard TSP algorithm and the Clustered Travelling Salesman Algorithm proposed by Lu's team[2]. Our study will also investigate the possible effects of different clustering tendencies on the performance of the Clustered Travelling Salesman Algorithm.

The analysis of these algorithms on datasets with varying clustering states will provide insights into the factors that affect the performance of the algorithms when applied to clustered TSP instances. The comparison of the results obtained by these algorithms with those obtained by the TSP and Clustered TSP algorithms will help to identify the most effective approach for solving the clustered TSP. Finally, the demonstration of different clusters with varying shapes, sizes, and densities will provide a better understanding of the impact of these factors on the performance of the Clustered Travelling Salesman Algorithm.

Overall, the proposed project will contribute to the existing body of knowledge on the clustered TSP and provide valuable insights into the performance of different TSP algorithms on clustered instances. There are several avenues for future work that can build upon our findings. One promising direction is the exploration of parallelization techniques to improve the performance of the algorithms by solving each cluster in parallel, thus drastically improving the solution time for large scale instances.

1.1 Background Information

1.1.1 Travelling Salesman Heuristics

Nearest Neighbour Algorithm

The Nearest Neighbor heuristic is a well-known algorithm for solving the Traveling Salesman Problem (TSP) that dates back to the 1950s [3]. It is a simple and intuitive algorithm that starts from an arbitrary city and iteratively adds the nearest unvisited city to the tour until all cities have been visited. In the proposed project, the Nearest Neighbor heuristic is the first algorithm used to solve the Clustered Traveling Salesman Problem (CTSP) on the generated datasets. This is because the Nearest Neighbor heuristic is a widely adopted algorithm for the TSP, and it provides a benchmark for evaluating the performance of other heuristics.

The Nearest Neighbor heuristic has several advantages, including its simplicity, speed, and ability to find good solutions for relatively small instances. However, it also has several shortcomings, including its tendency to produce suboptimal solutions and the sensitivity to the starting position. The heuristic provides a useful benchmark for evaluating the performance of other heuristics, including the Lin-Kernighan heuristic which is the other Algorithm that will be used in this project. By comparing the performance of other heuristics to that of the Nearest Neighbour, we can determine whether they can provide significant improvements in solution quality and computational efficiency.

The Nearest Neighbor algorithm has a computational complexity of O(n$\hat{2}$), where n is the number of nodes in the TSP instance. The algorithm works by selecting the nearest unvisited node at each step, which requires scanning all the remaining nodes to find the nearest one. Therefore, the overall time complexity of the algorithm is dominated by the pairwise distance computations between nodes, which require O(n$\hat{2}$) operations [4].

Lin-Kernighan Heuristic

The Lin-Kernighan algorithm is another well-known heuristic for solving the Traveling Salesman Problem (TSP) that was first introduced by Shen Lin and Brian Kernighan in 1973 [5]. The algorithm works by iteratively improving an initial tour by exchanging a sequence of edges in the tour with an alternative sequence of edges. This process is repeated until no further improvements can be made.

The Lin-Kernighan algorithm consists of two primary steps:

- Identify a sequence of "t" edges to be removed from the tour.
- Replace the removed edges with a new sequence of "t" edges to form an improved tour.

The algorithm iteratively applies these two steps to obtain a locally optimal tour. The Lin-Kernighan algorithm is considered to be one of the most effective heuristics for solving the TSP and has been the basis for several more advanced TSP heuristics [4] .

One advantage of the Lin-Kernighan algorithm over other TSP heuristics is its ability to find high-quality solutions for relatively large TSP instances. The algorithm's effectiveness can be attributed to its ability to explore a wider range of solution space compared to simpler heuristics like the Nearest Neighbor algorithm [6].

The Lin-Kernighan algorithm's computational complexity is typically observed to grow at a rate between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, where n is the number of nodes in the TSP instance [4]. While

this makes the algorithm more computationally expensive compared to simpler heuristics like the Nearest Neighbor algorithm, it is still more efficient compared to exact algorithms like, say, dynamic programming, which have exponential running times.

1.1.2 Clustering

Clustering is a widely adopted method for optimizing the Traveling Salesman Problem (TSP) by grouping the cities into clusters, thus forming a new variant of the problem known as the Clustered Traveling Salesman Problem (CTSP) [2]. In the CTSP, the cities are visited contiguously within the cluster before moving onto the next cluster. This approach can result in more efficient solutions, as it divides the large problem into multiple smaller problems to be handled at a time. This division reduces the total number of possible routes that need to be considered thus an overall improvement [7].

The proposed project aims to evaluate the performance of different TSP algorithms on clustered instances with varying clustering tendencies. Specifically, the project aims to explore the effect of varying clusters of different shapes, sizes, and densities on the search performance of the CTSP. This exploration is conducted by using two clustering techniques:

- Spectral Clustering
- Density-Based Clustering

The performance of TSP algorithms will be compared as the clustering tendency of the environments increases.

The clustering of cities can have a significant impact on the performance of TSP algorithms. For instance, the size and shape of the clusters can affect the total number of possible routes and, consequently, the time required to search for an optimal solution. Similarly, the density of the clusters can affect the efficiency of algorithms that rely on the geometric structure of the problem, such as the Nearest Neighbor algorithm.

Hopkins Statistic

The Hopkins statistic was first introduced by Hopkins in 1954 to measure the clustering tendency of a dataset. It is widely used in cluster analysis to determine the degree of clustering or randomness in a given dataset. The statistic measures the difference between the actual distribution of distances between data points and a uniformly distributed random sample of points. If the distribution of distances between data points is significantly different from that of the random sample, then the dataset is said to have a clustering tendency.

In the proposed project, the Hopkins statistic will be used to evaluate the clustering tendency of the datasets generated using spectral and density-based clustering techniques [8]. By measuring the degree of clustering or randomness in the datasets, we will use this as the gauge for which we'll group or separate the different datasets to be compared.

Spectral Clustering

Spectral clustering is a popular unsupervised clustering algorithm that groups together points in a dataset based on the similarity of their spectral embeddings. Developed by Ng, Jordan, and Weiss in 2002, spectral clustering has been widely used in various domains, including image segmentation, social network analysis, and bioinformatics [9].

The goal of spectral clustering is to partition a dataset into clusters, where each cluster is defined as a set of points that are similar to each other and dissimilar to points in other clusters. The algorithm operates by defining two key parameters: the number of clusters, referred to as n_clusters, and the method of computing the spectral embeddings, which is typically accomplished by constructing a graph from the data and computing the Laplacian matrix.

Spectral clustering works by first constructing a graph from the dataset, where each point is represented as a node in the graph, and the edges between nodes are weighted based on the similarity of the points. The Laplacian matrix is then computed from the graph that serves as a measure of the smoothness of the graph. The eigenvectors of the Laplacian matrix are then used to embed the data into a lower-dimensional space [10], where the points are then clustered using a traditional clustering algorithm, in the case of my implementation, RBF.

The resulting clusters may be of arbitrary shape, and spectral clustering is particularly effective at identifying clusters that are not linearly separable in the original space. Additionally, spectral clustering has been shown to perform well on datasets with varying density and high dimensionality (Luxburg, 2007).

Density-Based Clustering

Density-based spatial clustering of applications with noise, known as DBSCAN, is a popular unsupervised clustering algorithm that was introduced by Ester et al. in 1996. The algorithm groups together points in a dataset based on their proximity to each other, and it has been widely used in various domains, including image segmentation, anomaly detection, and customer segmentation [11].

The goal of DBSCAN is to partition a dataset, where each cluster of points is defined as a set of points that are densely packed together and at the same time separated from other densely packed points by regions that are sparsely populated. The algorithm operates by defining three key parameters:

- the minimum number of points required to form a dense region, referred to as min_samples
- the maximum distance that two points can be from each other to be considered part of the same cluster, referred to as epochs, or eps.
- Points that are not part of any cluster are labeled as noise.

DBSCAN selects a point in the dataset that is unassigned to a cluster or marked as noise, then assesses the number of neighboring points that exceed a specified minimum number of points required to form a dense region with the minimum numbers of neighbours within the set epochs distance of the selected point. If this minimum number of neighboring points is met, the point is designated as a core point and added to a new cluster. Points within epochs distance of the core point are then also included in the cluster. Border points that have fewer than the minimum required neighboring points are added to the cluster of a nearby core point. Any remaining unassigned points are labeled as noise. The process continues until all points

have been assigned to a cluster or marked as noise. The resulting clusters may be of arbitrary shape, and there may be clusters of varying density [11]

Clustered Traveling Salesman Problem and the Single Linkage Clustering Method

In the recent paper by Y. Lu and co. [2], a novel two-stage approach was proposed to solve the CTSP. The first stage involved constructing a TSP tour within each cluster using heuristic algorithms, one of them being the Nearest Neighbor heuristic. The second stage, referred to as "Inter-cluster tour merging", dealt with the process of linking the individual cluster tours into a single tour; A key method proposed for this second stage was the Single Linkage Clustering (SLC) method. In the context of CTSP, SLC is used to iteratively merge intra-cluster tours based on their nearest inter-tour points. This guarantees the minimum possible additional distance when merging two tours, hence optimizing the overall route.

The SLC method is both powerful and flexible in its application to the CTSP, helping to optimize inter-cluster tour merging with computational efficiency. However, it is also important to note that the effectiveness of the SLC can be influenced by the nature of the data set and the distribution of clusters, as well as has, to a certain extent the same limitation as that of Nearest Neighbour, in being that it yields faster results at the expense of slightly sub-optimal results. Despite potential limitations, SLC remains a valuable tool in the CTSP-solving toolkit, offering an innovative approach for inter-cluster tour merging.

Chapter 2

Methods

2.1 Clustering Functions

2.1.1 Hopkins Statistic

The hopkinsStatistic function takes an input Dataframe x and returns the Hopkins statistic. This statistic is used to assess the clustering tendency of the data. If the value of H is closer to 1, it indicates that the dataset has a high clustering tendency and is well clustered. On the other hand, if the value of H is closer to 0.5, it indicates that the dataset does not have a clear clustering tendency and is poorly clustered. If H is closer to 0, it suggests that the dataset is uniformly distributed, and clustering may not be an appropriate technique to analyze the data. This methodology is in reference to that of Lawson and Jurs [12] in conjunction with an implementation found of this found on StackOverFlow.

Initially, a sample size of 5% of X's rows is chosen. Two sets of points are generated: a uniform random sample and a random sample from X. Using scikit-learn's NearestNeighbors class, nearest neighbor distances are computed for both samples. The distance to the first (nearest) neighbor is retained for the uniform random sample, while the distance to the second nearest neighbor is retained for the random sample from X.

The Hopkins statistic is then calculated as the ratio of the sum of distances to the nearest neighbors for the uniform random sample over the sum of distances to the nearest neighbors for the random sample from X. This measure helps determine the degree of clustering in the data, with a higher Hopkins statistic indicating a stronger clustering tendency.

Given a dataset $X \in \mathbb{R}^{n \times d}$:

$$\mathbf{H}(X) = \frac{\sum_{i=1}^{m} u_i}{\sum_{i=1}^{m} (u_i + w_i)} \tag{2.1}$$

where u_i are the nearest neighbor distances for each point in a random uniform sample X_u of size m, and w_i are the nearest neighbor distances for each point in a random sample X_r of size m from the dataset X.

Algorithm 1 Simplified pseudocode for calculating the Hopkins statistic

function CALCULATEHOPKINS(X) **returns** a value H
    $X \leftarrow X$.values
    $X\_sample \leftarrow X$[random_indices]
    neigh $\leftarrow$ NearestNeighbors($n\_neighbors = p$)
    nbrs $\leftarrow$ neigh.fit(X)
    $u\_distances, u\_indices \leftarrow$ K-Neighbours from Uniform Samples
    $u\_distances \leftarrow u\_distances[:, 0]$
    $w\_distances, w\_indices \leftarrow$ nbrs.kneighbors($X\_samples$)
    $w\_distances \leftarrow w\_distances[:, 1]$
    $u\_sum \leftarrow \text{sum}(u\_distances)$
    $w\_sum \leftarrow \text{sum}(w\_distances)$
    $H \leftarrow \frac{u\_sum}{u\_sum + w\_sum}$
    return H
end function

2.1.2 Spectral Clustering

Spectral Clustering can be defined as: Given a set of points $X \in \mathbb{R}^{n \times d}$, number of clusters $n\_clusters$, and γ:

$$\mathbf{S}(X, n\_clusters, \gamma) = KMeans(n\_clusters).fit(\phi(L, n\_clusters)) = \{C_1, C_2, \ldots, C_k\} \quad (2.2)$$

where $\phi(L, n\_clusters)$ denotes the matrix containing the first $n\_clusters$ eigenvectors of the Laplacian matrix L, and $L = D - A$. The adjacency matrix A is computed as $A_{ij} = \exp(-\gamma||x_i - x_j||^2)$, and D is a diagonal matrix with $D_{ii} = \sum_j A_{ij}$.

Algorithm 2 Pseudocode for the Spectral Clustering algorithm

function SPECTRALCLUSTERING(locations, $n\_clusters, \gamma$)
    $X \leftarrow$ extract_coordinates(locations)
    $distances \leftarrow$ calculate_pairwise_distances(X)
    $\gamma \leftarrow$ compute gamma($distances$)
    $spectral \leftarrow$ spectral_clustering($n\_clusters, \gamma$)
    $cluster\_labels \leftarrow$ spectral.fit_predict(X)
    $clusters \leftarrow$ initialize_clusters()
    for i, label **in enumerate**(cluster_labels) **do**
        $x, y \leftarrow$ get_coordinates(X, i)
        add_point_to_cluster($clusters$, label, x, y)
    end for
    return $clusters$, cluster_labels
end function

Despite its effectiveness, spectral clustering has several limitations, including its sensitivity to the choice of the number of clusters and the method of computing the spectral embeddings. The choice of these parameters may require some domain expertise to determine appropriate

values. Additionally, spectral clustering may be computationally expensive on large datasets, and the resulting clusters may be highly dependent on the quality of the graph construction.

2.1.3 DBSCAN Clustering

Given a set of points $X \in \mathbb{R}^{n \times d}$, distance threshold e, and minimum number of points $min\_samples$:

$$\mathbf{D}(X, e, min\_samples) = DBscan(X, d, N, e, min_samples) = \{C_1, C_2, \ldots, C_k\} \tag{2.3}$$

where d denotes $d(x_i, x_j)$, which computes the distance between points x_i and x_j in X, a neighborhood function N is such that $N(x_i) = x_j \in X \mid d(x_i, x_j) \leq e$, which returns the set of points within the distance threshold e from the point x_i, a core point as a point $x_i \in X$ with at least $min_samples$ points in its neighborhood, i.e., $|N(x_i)| \geq min_samples$, and $\{C_1, C_2, \ldots, C_k\}$ are the clusters formed by connecting core points and their reachable neighbors, and a core point is defined as a point with at least $min\_samples$ neighbors within a distance of e.

Algorithm 3 DBSCAN clustering algorithm

function DBSCANCLUSTER($locations, n\_clusters, targetHop, n, nPerCluster, gauss, e$)
    $X \leftarrow$ np.array(locations)
    $distances \leftarrow$ pairwise_distances(X)
    $\gamma \leftarrow \frac{1}{2 \cdot (\text{np.median}(distances))^2}$
    $dbscan \leftarrow$ DBSCAN(eps=e, min_samples=5).fit(X)
    $cluster\_labels \leftarrow$ dbscan.labels_
    for $i, label$ **in enumerate**($cluster\_labels$) **do**
        $x, y \leftarrow X[i]$
        clusters[label].append((x, y))
    end for
    return $clusters, cluster\_labels$
end function

DBSCAN has several advantages over other clustering algorithms, including its ability to identify arbitrary shaped clusters and its resistance to noise. However, the choice of epochs and min_samples can greatly affect the resulting clusters and may require some domain expertise to determine appropriate values. Additionally, DBSCAN may struggle with datasets of varying density or datasets with high dimensionality.

2.2 TSP Functions

2.2.1 Held-Karp Lower Bound

The held_karp_lower_bound function aims to calculate a lower bound on the total distance for the Traveling Salesman Problem (TSP) using the Held-Karp algorithm. The TSP is a

well-known combinatorial optimization problem, where given a list of cities and their pairwise distances, the goal is to find the shortest possible tour that visits each city exactly once and returns to the starting city. The Held-Karp algorithm computes a lower bound on the optimal tour length, which can be used to prune the search space when searching for the optimal solution.

The input parameter points is a list of 2D coordinates representing the locations of the cities in the TSP. The function begins by calculating the pairwise distances between all cities, resulting in an n x n distance matrix, where n is the number of cities.

The minimum spanning tree (MST) is then calculated from the distance matrix using the minimum_spanning_tree function from the scipy.sparse.csgraph module. The MST is a tree that connects all vertices in a graph such that the sum of the edge weights is minimized. The sum of the edge weights in the MST provides a lower bound for the optimal TSP tour length since the optimal tour must also connect all cities.

To refine the lower bound further, the function finds the minimum distance from the starting city to any other city (min_start_edge) and the minimum distance from any other city back to the starting city (min_end_edge). The lower bound is then calculated by summing the MST's total weight, min_start_edge, and min_end_edge.

Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$:

$$\mathbf{HK}(P) = \text{MST}(P) + \min_{j>1}(d(p_1, p_j)) + \min_{j>1}(d(p_j, p_1)) \tag{2.4}$$

Where MST(P) is the sum of edge weights of the minimum spanning tree of P, and $d(p_i, p_j)$ denotes the Euclidean distance between points p_i and p_j.

Algorithm 4 Held-Karp Lower Bound Function

```
function HELD_KARP_LOWER_BOUND(points)
    n ← len(points)
    dist ← Euclidean distance matrix for points
    mst ← sum of edge weights of minimum spanning tree of dist
    min_start_edge ← np.min(dist[0, 1 :])
    min_end_edge ← np.min(dist[1 :, 0])
    lower_bound ← mst + min_start_edge + min_end_edge
    return lower_bound
end function
```

2.2.2 Distance Functions

Euclidean Distance between points defined as:

$$\mathbf{D(p1,\ p2)} = \sqrt{(p1_x - p2_x)^2 + (p1_y - p2_y)^2} \tag{2.5}$$

Tour Distance throughout all points can be defined as:

$$\mathbf{TotalDistance(tour)} = \sum_{i=0}^{\text{len(tour)}-1} \mathit{distance}(\mathit{tour}_i, \mathit{tour}_{i-1}) \tag{2.6}$$

2.2.3 Lin-Kernighan Heuristic

The lin_kernighan_tsp function is an implementation of the Lin-Kernighan heuristic for solving the TSP. The Lin-Kernighan heuristic is a local search algorithm that attempts to improve an initial tour iteratively by swapping pairs of edges to reduce the tour length. The algorithm has been shown to produce high-quality solutions for the TSP in practice.

Given a set of points $P = p_1, p_2, \ldots, p_n$, the Lin-Kernighan algorithm constructs a tour by applying the 2-opt local search algorithm to a random initial tour. The 2-opt algorithm iteratively swaps two edges in the tour if the resulting tour has a shorter total distance. The LK algorithm can be represented as follows:

1. Generate a random initial tour T.
2. Perform 2-opt swaps on T until no further improvement is possible.
3. If an improved tour is found, go to step 2; otherwise, return the best tour found.

The 2-opt swap operation is represented by the following formula:

$$\mathbf{LK}(p) = T' = T \setminus \{(p_i, p_{i+1}), (p_j, p_{j+1})\} \bigcup \{(p_i, p_j), (p_{i+1}, p_{j+1})\} \tag{2.7}$$

Algorithm 5 Lin-Kernighan heuristic algorithm

```
function LinKernighan(points)
    function twoOptSwap(tour, i, k)
        return tour[: i] + reversed(tour[i : k + 1]) + tour[k + 1 :]
    end function
    n ← len(points)
    indexTour ← list(range(n))
    shuffle(indexTour)
    improvement ← True
    while improvement do
        improvement ← False
        for i in range(1, n − 1) do
            for k in range(i + 1, n) do
                newTour ← twoOptSwap(indexTour, i, k)
                if tourDistance(newTour) < tourDistance(indexTour) then
                    indexTour ← newTour
                    improvement ← True
                end if
            end for
        end for
    end while
    pointTour ← [points[i] for i in indexTour]
    return pointTour, tourDistance(pointTour)
end function
```

2.2.4 Nearest Neighbour Heuristic

This Nearest Neighbour function takes the input parameter points is a list of 2D coordinates representing the locations of the cities in the TSP. The function initializes an empty set of unvisited cities and a list of visited cities with the starting city at index 0. The algorithm iteratively chooses the nearest unvisited city to the current city, updates the total distance, and moves the chosen city from the unvisited set to the visited list. Once all cities have been visited, the algorithm returns to the starting city and the final tour and its total distance are returned.

Given a set of points $P = p_1, p_2, \ldots, p_n$, the Nearest Neighbor algorithm constructs a tour by iteratively selecting the nearest unvisited point from the current point, starting from p_1. The algorithm can be represented by the following formula:

$$\mathbf{NN}(p) = T = \bigcup_{i=1}^{n-1} (p_i, \arg \min_{p_j \in P \setminus p_1, \ldots, p_i} d(p_i, p_j)) \tag{2.8}$$

where T is the final tour, and $d(p_i, p_j)$ denotes the Euclidean distance between points p_i and p_j.

Algorithm 6 Nearest Neighbour algorithm

```
function NearestNeighbour(points)
    n ← len(points)
    unvisited ← set(range(1, n))
    visited ← [points[0]]
    totalDistance ← 0
    while unvisited do
        minDistance ← ∞
        for i in unvisited do
            distance ← Distance between points[currentIdx] and points[i]
            if distance < minDistance then
                minDistance ← distance
                nextIdx ← i
            end if
        end for
        totalDistance ← totalDistance + minDistance
        currentIdx ← nextIdx
    end while
    totalDistance ← totalDistance + Distance between points[0] and points[currentIdx]
    visited.append(points[0])
    return visited, totalDistance
end function
```

2.2.5 Single Linkage Clustering

Given a set of clusters $C = \{c_1, c_2, \ldots, c_m\}$ where each cluster c_i is a sequence of points from the set $P = \{p_1, p_2, \ldots, p_n\}$, the SLC function merges clusters iteratively until only one cluster remains. This process is expressed as:

$$\mathbf{SLC}(C, T) = \begin{cases} C, & \text{if } |C| = 1 \\ \mathbf{SLC}(C', T'), & \text{otherwise} \end{cases}$$

where

- $C = \{c_1, c_2, \ldots, c_m\}$ is the current set of clusters,
- T is the current distance table,
- $(c_i, c_j) = \arg\min_{i \neq j} D(i, j)$ are the pair of clusters with the minimum inter-cluster distance,
- $C' = (C \setminus \{c_i, c_j\}) \cup \{c'\}$ is the new set of clusters after merging c_i and c_j into a new cluster c', and
- T' is the updated distance table after the merge.

The inter-cluster distance between two clusters c_i and c_j is defined as the minimum Euclidean distance between any pair of points $p_k \in c_i$ and $p_l \in c_j$:

$$D(i, j) = \begin{cases} \min_{p_k \in c_i, p_l \in c_j} dist(p_k, p_l) & \text{if } i \neq j \\ \infty & \text{if } i = j \end{cases}$$

2.2.6 Single Linkage Clustering Optimized wtih Localized Lin-Kernighan

Here, we implemented a modified version of the Lin-Kernighan which now has two changes:

1. No Longer Shuffles the Initial tour
2. Now, using a KD-Tree, performs a localized search in order to optimize nearby areas to the points.

This implementation can be expressed as such:

Given the current tour T and a point p_i in T, let Q be the set of points within a distance r from p_i, efficiently retrieved using the KDTree's 'query_ball_point' function:

$$Q = \text{tree.query\_ball\_point}(p_i, r)$$

For each $p_j \in Q$, the function **LK** improves T by replacing the edges (p_i, p_{i+1}) and (p_j, p_{j+1}) with (p_i, p_j) and (p_{i+1}, p_{j+1}) if the total distance of the tour decreases:

$$\mathbf{LKMod}(p) = T' = T \setminus \{(p_i, p_{i+1}), (p_j, p_{j+1})\} \bigcup \{(p_i, p_j), (p_{i+1}, p_{j+1})\}$$

Which in the end, we can concatenate everything to express it as:

$$\mathbf{SLC}(C, T) = \begin{cases} \mathbf{LKMod}(C), & \text{if } |C| = 1 \\ \mathbf{LKMod}(\mathbf{SLC}(C', T')) \end{cases}$$

Algorithm 7 SLC with Lin-Kernighan algorithm

```
1: function SLCwithLK(solution, table)
2:     clusters ← [s[0] for s in solution]
3:     while len(clusters) > 1 do
4:         min_distance, index1, index2 ← ∞, −1, −1
5:         for i, j in range(len(clusters)) if i ≠ j do
6:             if table[i][j] < min_distance then
7:                 min_distance, index1, index2 ← table[i][j], i, j
8:             end if
9:         end for
10:        entry_point, exit_point ← table[index1][index2]
11:        cluster1, cluster2 ← clusters[index1], clusters[index2]
12:        a ← cluster1[: entry + 1]
13:        b ← cluster2[exit :]
14:        c ← cluster2[: exit + 1]
15:        d ← cluster1[entry + 1 :]
16:        merged ← a + +b + +c + +d
17:        clusters.insert(index1, merged_cluster)
18:        table ← closestPointTable(clusters, range(len(clusters)))
19:    end while
20:    return LinKernighanMod(clusters[0])
21: end function
```

Chapter 3

Experiments

In this experiment, we aim to explore the performance of various TSP algorithms in solving the Traveling Salesman Problem under different conditions. We will focus on four different TSP algorithms, represented as:

1. Nearest Neighbour (NN) algorithm: $\mathbf{NN}(P)$
2. Lin-Kernighan Heuristic (LK): $\mathbf{LK}(P)$
3. Spectral Clustering with Lin-Kernighan Heuristic (SLC): $\mathbf{SLC}(P)$
4. Spectral Clustering with Nearest Neighbour Heuristic (SLCwithLK): $\mathbf{SLCwithLK}(P)$

We evaluate the performance of these algorithms on a set of points (P), which are generated using various target Hopkins Statistic values, Gaussian standard deviation values, and different nPerCluster values around nCluster centres. The points are generated by the function:

$$\mathbf{G}(n\_clusters, \gamma) = \text{generateAround}(nPerCluster, g, centres) \tag{3.1}$$

For each combination of targetHop, Gaussian standard deviation (g), and nPerCluster, we perform the TSP algorithms and calculate the average tour distance:

$$\mathbf{Performance}(avg_HK, avg_dist) = \frac{\text{avg\_hk}}{avg\_dist} \tag{3.2}$$

Where $\mathbf{F}(P)$ denotes one of the TSP algorithms.

In each of our experiments, we adjusted a set of variables to evaluate the sensitivity and adaptability of the different TSP algorithms. These variables were selected to span a range of values to better understand their impact on the resulting tours and to highlight the strengths and weaknesses of each algorithm under varying circumstances. Here is a list of the parameters:

- `Number of Clusters`
- `Hopkins Statistic`
- `Number of Points per Cluster`
- `Gauss`
- `Functions`

The key aspects we'll be examining in the experiments are as follows:

- **Quality of the solutions**: Assess the quality of the solutions obtained from each TSP algorithm by calculating the gap between the generated solutions and the Held-Karp

lower bound **HK**(P). This will help quantify the effectiveness of the algorithms in finding near-optimal solutions to the TSP. This will simply be done by Running each algorithm on the same set of points and looking at the route and the total distance.

`Goals/ Variables Measured:`
- Total Distance
- Assessing Solution Quality, i.e., number of loops/crosses

`Static Variables:`
- Number of Points per Clusters
- Number of Clusters
- Hopkins Statistic
- Gauss
- Functions[Nearest Neighbour, Lin-Kernighan, SLC, SLCwithLK]

- **Computational efficiency**: Analyze the computational efficiency of each TSP algorithm in terms of time complexity and memory consumption. This will provide insights into the scalability of the algorithms when applied to large-scale datasets. We do this by simply incrememnting the total number of Points in each map, running each algorithm and recording the amount of time it takes each algorithm to complete.

`Goals/ Variables Measured:`
- Time(ms) to solve each map

`Changing Variables:`
- Number of Points per Clusters
- Number of Clusters

`Static Variables:`
- Hopkins Statistic
- Gauss
- Functions[Nearest Neighbour, Lin-Kernighan, SLC, SLCwithLK]

- **Clustering tendency**: We vary the target Hopkins Statistic values. This allows us to control the degree to which the points are clustered, and observe the impact of different clustering tendencies on the performance of the TSP algorithms. By adjusting the Gauss measurement, we can adjust the density of the clusters, ranging each set from well clustered, to random.

`Goals/ Variables Measured:`
- Time(ms) to solve each map
- Search performance

`Changing Variables:`
- Gauss
- Number of Points per Clusters
- Number of Clusters

`Static Variables:`
- Hopkins Statistic

  - Functions[Nearest Neighbour, SLC, SLCwithLK]

- **Number of clusters vs Number of Points per Cluster**: We also need to verify the effect that's had with the same number of points going from 1 cluster to many. By changing the number of clusters in the dataset, we aim to understand how the TSP algorithms cope with varying levels of granularity in the clustering structure. We do this simply by incrementing the Cluster Size, and dividing all the points among them. Here we take n number of points and m number of clusters. We start at m=1 and increase the number of clusters until 10. Then we observe the the effect on performance and time that incrementing the number of clusters has on our results.

  `Goals/ Variables Measured:`
  - Time(ms) to solve each map
  - Search performance

  `Changing Variables:`
  - Number of Clusters
  - Number of Points in Each Cluster

  `Static Variables:`
  - Gauss
  - Hopkins Statistic
  - Functions[Lin-Kernighan, SLC, SLCwithLK]

- **Varying clustering densities**: Here we'll test with different clusters having different concentration of points than others. To assess the impact of this on the TSP algorithms, we generate datasets with clusters of different densities and analyze the performance of the algorithms under these conditions.

  `Goals/ Variables Measured:`
  - Search performance

  `Changing Variables:`
  - Number of Points per Clusters
  - Gauss

  `Static Variables:`
  - Number of Clusters
  - Functions[SLC, SLCwithLK]

- **Isotropic vs Elongated Clusters**: Finally, we'll study the performance of the TSP algorithms when applied to datasets with more circular clusters relative to elongated ones. This can be achieved by adjusting the Gaussian x and y factors for the clusters, which essentially limits the number of x values for each y value and vice verse, as a result creating an elongating effect.

  `Goals/ Variables Measured:`
  - Search performance
  - Time (ms)

  `Changing Variables:`
  - Gaussian x and y values

`Static Variables:`
- Number of Points per Clusters
- Number of Clusters
- Functions[Lin-Kernighan, SLC, SLCwithLK]

Through this extensive analysis, we aim to gain a better understanding of the performance and robustness of TSP algorithms under various experimental conditions, which can potentially inform the selection of the most appropriate TSP algorithm and parameters for real-world applications involving clustered data.

Chapter 4

Discussion & Results

4.1 Experiment Results

4.1.1 Quality of Solutions

The aim of this experiment was simple; Run all the algorithms on the same group of points, look at the results, and assess the quality of the solutions. Here we have the following clustered group of points:

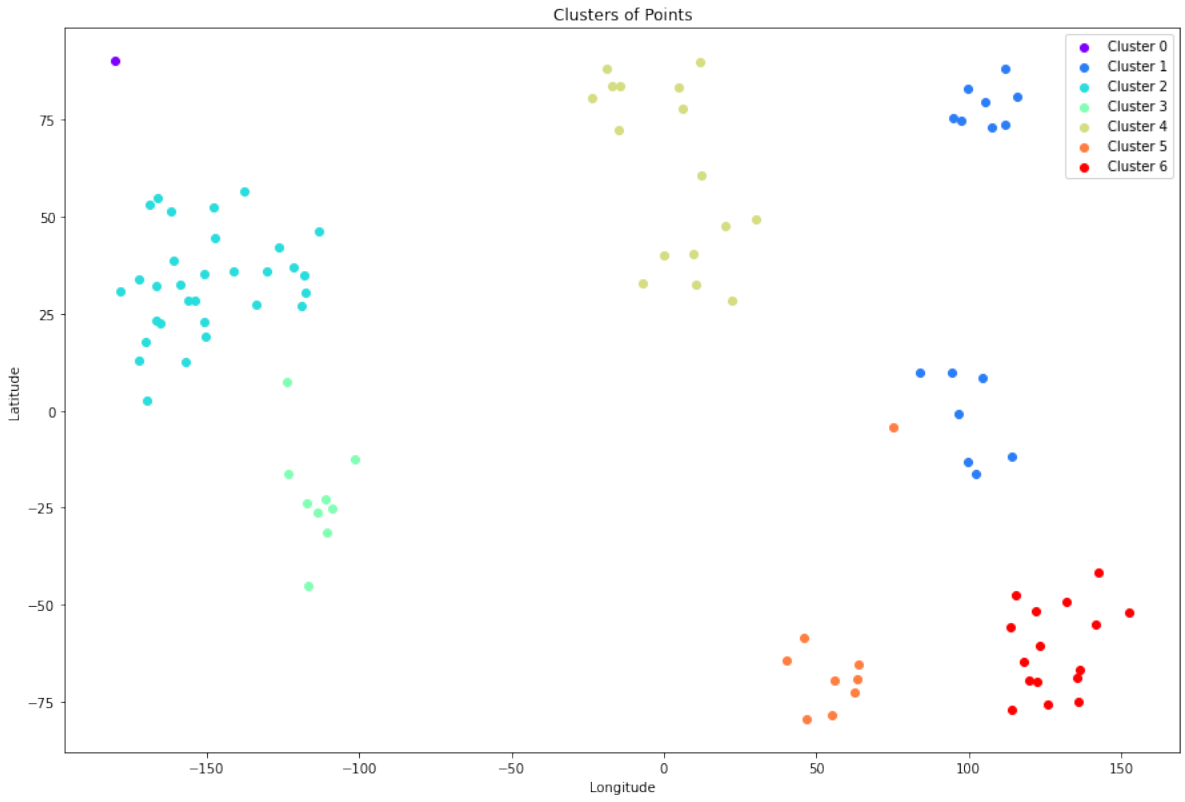


Figure 4.1: Figure showing all the points in their Respective Clusters

From the above points we obtained the following plots from our implemented algorithms:

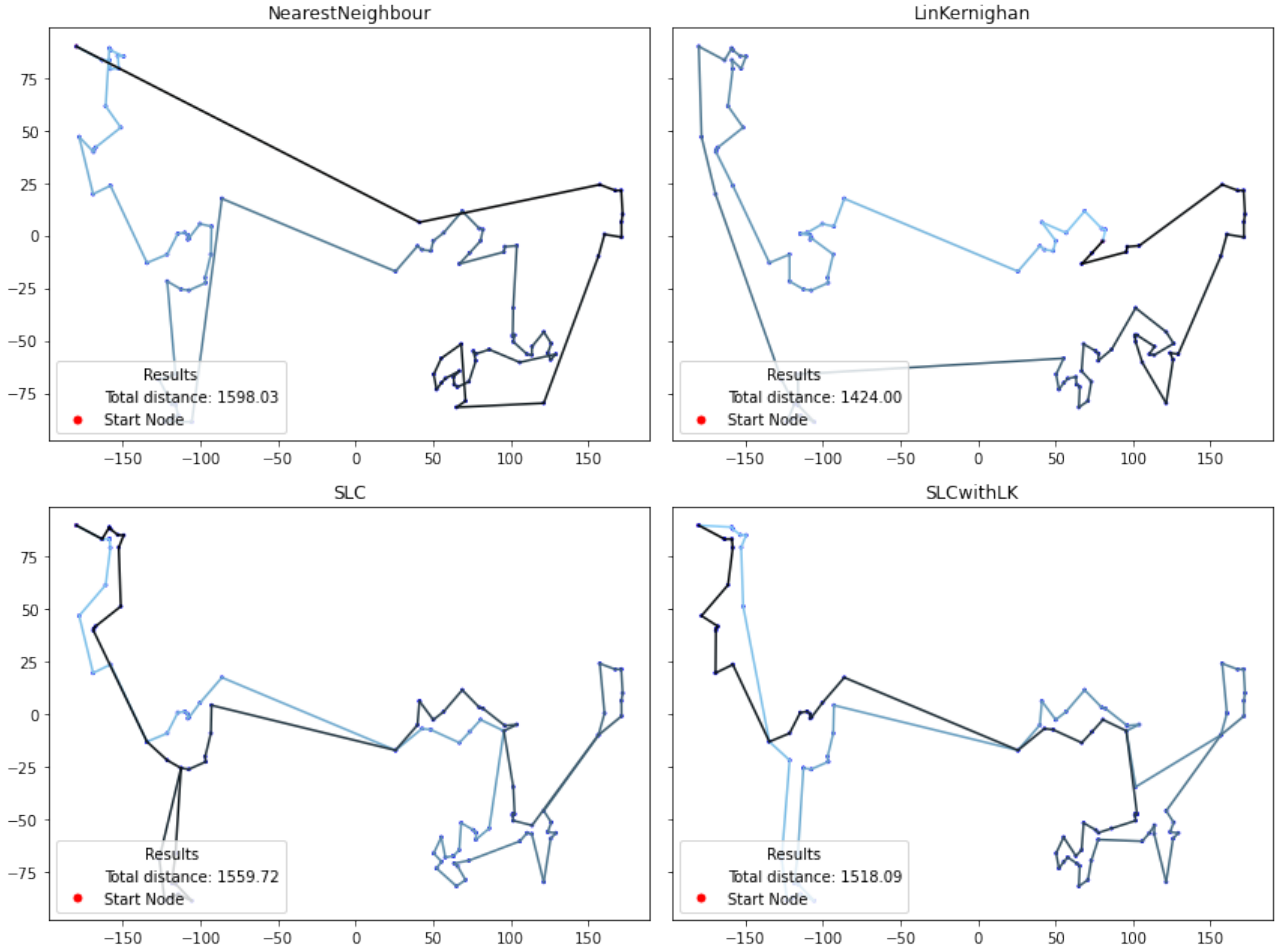


Figure 4.2: Figure Showing all Algorithms Ran on the Same Points

They all start from the same starting point in the top left corner, and the colour of the lines darken from light blue to black to visualize the direction the algorithm is travelling.

Findings

Here we can clearly see the best performing solution is the Lin-Kernighan, as it's meant to be a near optimal solution for the TSP. The key to its success is the way it refines the solution iteratively through a series of 'exchanges' until no further improvements can be made, ensuring that the final route is as optimal as possible. We can see that there are no crossings or intersections, and there's a clear path going around all the points. For the Clustered Algorithms, we can see the path is also quite good but not optimal. They identify and connect clusters efficiently, minimizing travel distance between different clusters, and consequently offering a relatively optimal solution. This can be particularly advantageous when handling complex and large-scale TSP instances, as we will see in the later experiments. On the other hand, the Nearest Neighbour algorithm is a strictly greedy algorithm, which tends to give suboptimal results. It simply selects the closest unvisited node as the next node, leading to fast solutions,

but inefficient routes.

4.1.2 Computational Efficiency

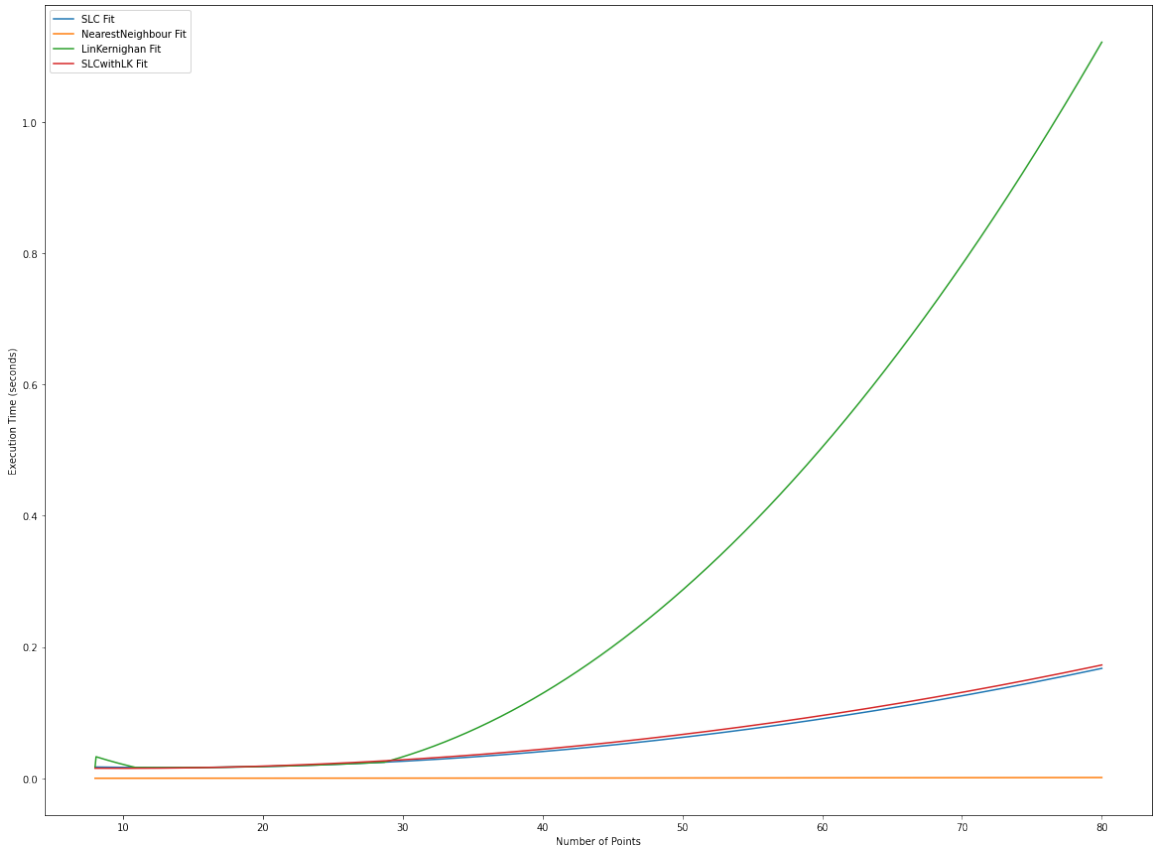


Figure 4.3: Plots of the Change in Distance vs Cluster Tendency

Nearest Neighbour
By far the most computationally efficient of all the algorithms. It simply operates by taking a starting point and then iterating over all remaining unvisited nodes to find the nearest one. This operation, repeated for each node, results in a quadratic time complexity $\mathcal{O}(n^2)$.

Lin-Kernighan
Contrary to the Nearest Neighbour this is the most computationally expensive algorithm. The nature of the algorithm involves substantial computational expense to find the most optimized solution by evaluating all pairs of edges in the path for potential improvements in order to find the most optimal path. Each iteration in this heuristic involves assessing a variety of potential paths, which results in much longer wait times for larger datasets.

Clustered Algorithms
On the contrary, the SLC Clustering TSP algorithm simplifies the problem by breaking down the larger dataset into smaller, manageable clusters. The algorithm then solves the Travelling

Salesman Problem (TSP) within these smaller clusters individually. Given the smaller size of these clusters compared to the overall dataset, the task of solving the TSP becomes relatively quicker. This method reduces the number of 'exchanges' or 'moves' to be evaluated, which ultimately lessens the computational burden. When it comes to forming the final solution by combining all the clusters, the algorithm evaluates 'moves' proportional to the number of clusters, which is again considerably less than the total number of nodes in the dataset.

In essence, while Lin-Kernighan's intensive path optimization leads to slower performance for particularly larger datasets, SLC's strategy of breaking down the problem into smaller parts makes it much more computationally efficient, thereby facilitating quicker solutions.

4.1.3 Clustering Tendency

Search Performance:

Theoretical expectations might suggest significant performance variation in algorithms based on the clustering tendency, as quantified by the Hopkins statistic. However, the results from the experiments indicate a different scenario, illustrating that the effect of changes in the Hopkins statistic on the algorithms' search performance fluctuated more than expected, essentially illustrating that change in clustering tendency doesn't particularly lead to predictable alterations in the search performance.

Figure 4.4: Plots of the Change in Distance vs Cluster Tendency

Speed:

Similarly, we predicted that for the Clustering algorithms, we would see an improved performance for this metric than with the basic algorithms, and the results do show that. So while there's a difference between the algorithms, but we don't notice much of a trend in the algorithms within themselves as the tendency changes

Figure 4.5: Plots of the Change in Execution Time vs Cluster Tendency

4.1.4 Number of Clusters vs Size of Clusters

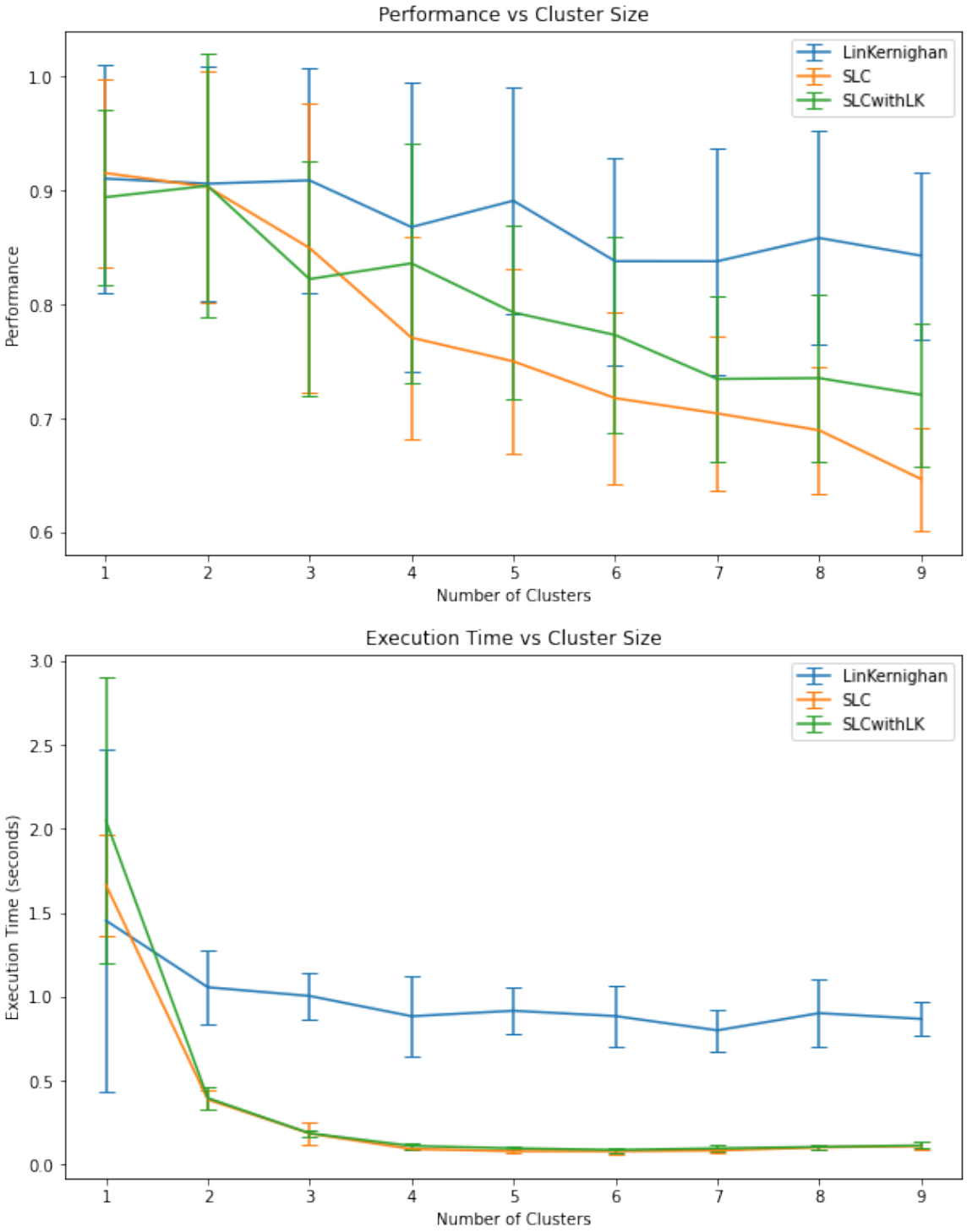


Figure 4.6: Plots of the Change in Performance and Execution Time vs the Number of Clusters

We slowly increased the amount of points in the table to observe the effect of highly clustered data on the performance and efficiency. In the case of the Performance, we measured it, of

course, against the Lower bound of the dataset and found that there was actually a gradual decrease in optimization of the results as the number of clusters increased. It's important to highlight that the observed underperformance is more than likely rooted in the fundamental nature of the clustering-based solutions and their inherent approach to solving the problem. Specifically, these algorithms focus on effectively clustering the data points and then connecting these clusters, which isn't always the most efficient route from a global perspective.

The previous experimental results bear this out. In the first experiment, for instance, the optimal Lin-Kernighan solution shows a path that skirts along the perimeter of all points, demonstrating a global optimization strategy. In contrast, the solutions derived from the clustering algorithms are more focused on connecting points within individual clusters and then linking these clusters together, essentially adopting a local optimization strategy.

While this approach is beneficial in efficiently dealing with and reducing the complexity of large datasets by breaking them down into more manageable subsets, it does not necessarily yield the most optimal path when considering the dataset as a whole. As such, when these results are compared to the optimal solution from the Lin-Kernighan algorithm, they might appear to underperform.

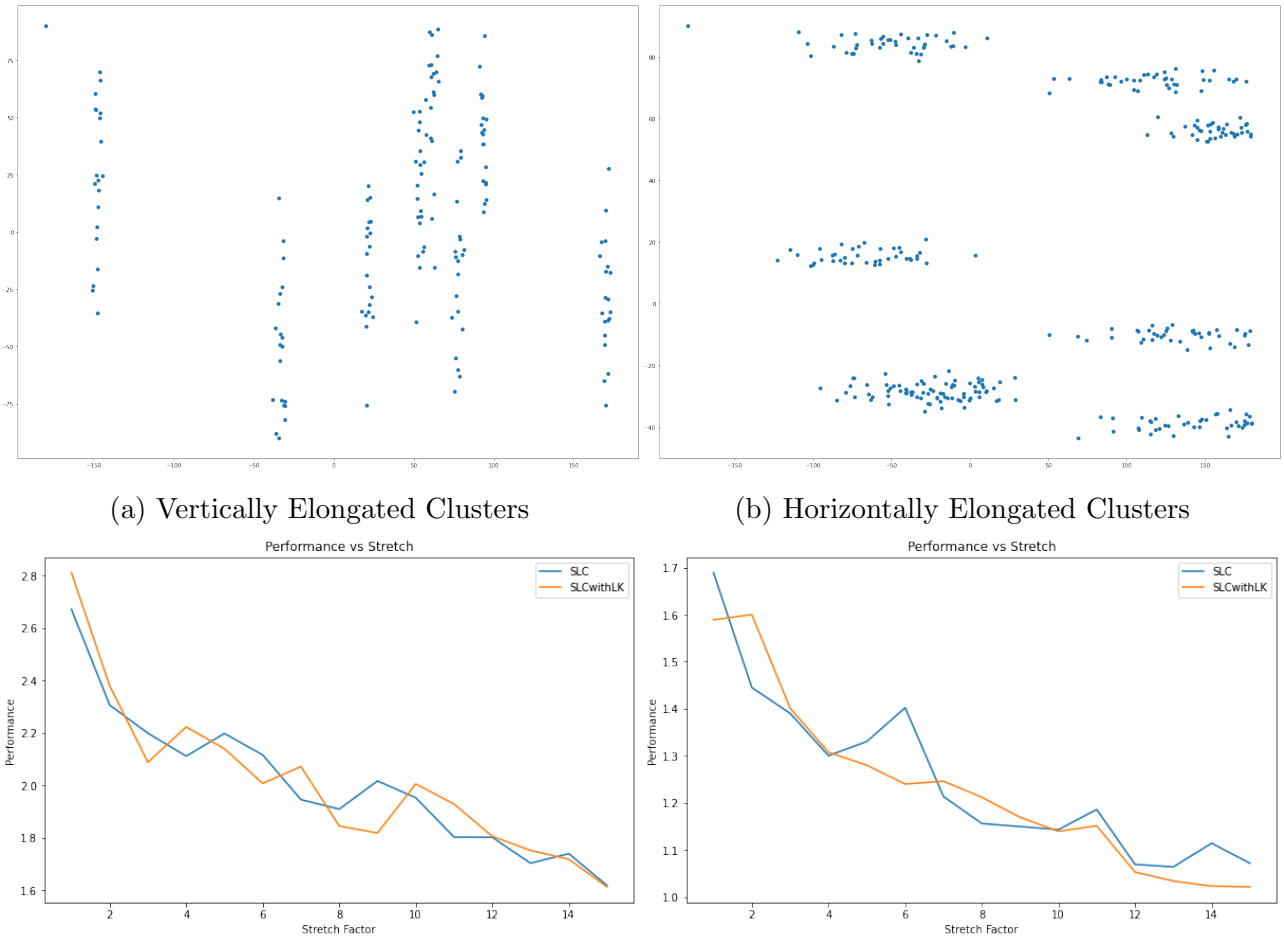
Also measuring the execution time as we increased the number of points, the results came to no surprise. We observe that with respect to the time complexity, increasing the number of clusters trumps the other algorithm in that regard, for fairly obvious reasons. Dividing the larger problem into numerous smaller subproblems, and concatenating them, adopting the local optimization strategy, will intuitively trump any other method for this nature of problem.

4.1.5 Varying Cluster Density

Unfortunately for the varying clustering density, we weren't able to draw a valid conclusion regarding this, with respect to search performance, there seemed to be no real correlation, neither was there for Execution time, the results produced were of course, more sensitive to other parameters, overall Cluster density, number of Clusters, etc.

4.1.6 Isotropic vs Elongated Clusters

Here we analysed whether there would be a noticeable difference between more elongated clusters as opposed to isotropic clusters:



(a) Vertically Elongated Clusters

(b) Horizontally Elongated Clusters

(c) Performance against Vertical Stretching

(d) Performance against Horizontal Stretching

Figure 4.7: Figures Showing Performance vs Elongation of the Clusters

We can see that the more elongated the clusters, the more optimal the results. This makes intuitive sense as the nature of these elongated clusters pairs itself well to the concept of tour construction based on the clustered approach. More specificially, when clusters of points are elongated, an algorithm can traverse through each cluster in a more sequential manner,

systematically moving from one end of the cluster to the other. This pattern can often mirror the optimal route more closely than in isotropic clusters, where the lack of a distinct structural direction can make determining an efficient path more difficult/unlikely.

Chapter 5

Conclusion & Future work

5.1 Conclusion

The central objective of this research is to investigate the effectiveness of different approaches for solving the Travelling Salesman Problem; more specifically with the use of Clustering Methods, creating the Clustered Travelling Salesman Problem, and understanding how differenjt parameters influence the efficacy of these problems. Our experiments yielded some expected results, a couple of surprises, and on a whole granted a lot of insight into the nature of the algorithms.

Looking purely at the solutions themsselves, the CTSP algorithms presented a distinct method of breaking down the problem, providing reasonably optimal solutions while significantly reducing the computational load relative to more exact solutions, thus making it overall more efficient for large-scale TSP instances.

Interestingly, when it came to clustering tendency measured by the Hopkins statistic, the search performance of the algorithms did not follow a predictable pattern. However, the CTSP algorithm did, in fact, demonstrate quicker execution times when data had higher clustering tendencies. This aligns with the fundamental strategy of the algorithm, effectively grouping data points into distinct clusters, thus reducing the complexity of the problem.

Another one of the key insights derived from our investigation is the relationship between the number of clusters and their size. As the number of clusters increased, there was a notable decrease in the optimization of the results compared to the more optimized algorithm. This suggests that while clustering strategies significantly improve the efficiency of handling large datasets, they may not necessarily yield the most globally optimal paths. A similar trend was observed when the clusters were isotropic versus elongated. Elongated clusters, as a result of their structural directionality would yield more optimal results, pointing to the fact that the physical nature of the clusters can impact the efficiency of the solutions.

In conclusion, our research validates the merit in utilizing the CTSP as a strategy for tackling the TSP for certain instances. Despite not providing globally optimal solutions, the value lies in computational efficiency and ability to better deal with larger datasets. Therefore, the choice of algorithm should more be inflienced by the unique characteristics of the problem, such as the number and the type of clusters, as well as the desired balance between computational efficiency and need for the most optimal solution.

5.2 Future Work

Despite thhe work done in this research elucidating the performances and characteristics of the CTSP algorithms, there is still a lot of potential work to be done. One such avenue is with the use of parallelisation or parallel computing being applied to the Clustering Algorithms explored in order to even further speed up the process for larger instances. Intuitively, with the nature of these algorithms involving the splitting up the large instance into multiple smaller instances, and solving them in parallel, would yield an improvement in Computation time. It could also be a means for providing drastic imnprovement for the non-clustered algorithms such as the Lin-Kernighan. Consequentially, the design and implementation of parallel versions of these algorithms should be a major focus of future work.

In addition to the on TSP, the application of clustering methods to other NP-hard problems warrants investigation. Many of these problems share structural similarities with TSP, such as the Job Shop Scheduling Problem, and the Knapsack Problem. The approach of clustering to break down larger problems into more manageable subsets could be an effective strategy to reduce the complexity and improve the efficiency of solving these types problems.

Overall, there is substantial room for future work in terms of enhancing the efficiency and effectiveness of the CTSP and other NP-hard problems, with potential far reaching benefits in numerous fields.

Bibliography

[1] Y. Hultaichuk and A. Matviychuk, "Modern approaches for solving the travelling salesman problem and the examples of their effective application," *Visti Natsionalnoi Akademii Nauk Ukrainy*, vol. 4, pp. 87–97, 2020.

[2] Y. Lu, J.-K. Hao, and Q. Wu, "Solving the clustered traveling salesman problem via tsp methods," *arXiv preprint arXiv:2007.05254*, 2020.

[3] R. Bellman, *Dynamic Programming*. Princeton University Press, 1962.

[4] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

[5] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, vol. 21, no. 2, pp. 498–516, 1973.

[6] K. Helsgaun, "General k-opt submoves for the lin-kernighan tsp heuristic," *Mathematical Programming Computation*, vol. 1, no. 2-3, pp. 119–163, 2009.

[7] J. A. Chisman, "The clustered traveling salesman problem," *Computers Operations Research*, vol. 2, no. 2, pp. 115–119, 1975.

[8] A. Banerjee and R. N. Dave, "Validating clusters using the hopkins statistic," in *Fuzzy Systems, 2004. Proceedings. 2004 IEEE International Conference on*, pp. 149–154, IEEE, 2004.

[9] A. Ng, M. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," *Advances in neural information processing systems*, vol. 2, pp. 849–856, 2002.

[10] U. von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[11] M. Ester, H. P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Kdd*, pp. 226–231, 1996.

[12] B. W. Lawson and P. C. Jurs, "New index for clustering tendency and its application to chemical problems," *Journal of Chemical Information and Modeling*, vol. 30, no. 1, pp. 36–41, 1990.