Solving the Traveling Salesman Problem Using Clustering Techniques

The sentences are mostly generated by ChatGPT. THIS PAPER IS NOT FOR PUBLICATION.

Hyejun Park
Department of Computer Science and Engineering
Seoul National University
1253spring@snu.ac.kr

September 26, 2024

Abstract

The Traveling Salesman Problem (TSP) is a classic optimization challenge in combinatorial mathematics, recognized for its NP-hard complexity. This paper explores a novel approach that employs clustering techniques to partition cities into smaller, manageable groups, thereby enhancing the efficiency of TSP solutions. We develop an algorithm that clusters cities based on geographic proximity and applies traditional TSP methods to each cluster, ultimately merging the individual tours into a comprehensive route. Our experiments were conducted on graphs with 100 vertices, testing various configurations across three cases to evaluate the impact of the number of clusters, clustering methods, and edge removal criteria on computational efficiency and solution quality. The results demonstrate a significant sensitivity of performance to cluster size, with larger clusters leading to increased execution times and approximation ratios. While our clustering approach shows promise, challenges such as kernel crashes with large clusters highlight the need for improved handling of larger datasets. Future work will focus on developing a clustering algorithm that limits the maximum size of each cluster to mitigate these issues. This study contributes to the growing body of literature on approximation algorithms for TSP and underscores the potential of clustering techniques to address complex optimization problems.

Keywords Traveling salesman · TSP · Clustering

1 Introduction

The Traveling Salesman Problem (TSP) is a well-known optimization challenge in combinatorial mathematics and computer science. Formally, given a set of n cities represented as a complete graph $G = (V, E)$—where V is the set of vertices (cities) and E is the set of edges (paths between cities) with weights d_{ij} representing the distances—the objective is to find the shortest Hamiltonian cycle that visits each vertex exactly once and returns to the starting city. Mathematically, this can be expressed as minimizing the total distance for a permutation σ of the cities:

$$\text{Minimize } \sum_{k=1}^{n} d_{\sigma(k),\sigma(k+1)} \text{ with } \sigma(n+1) = \sigma(1)$$

Classified as NP-hard, TSP lacks known polynomial-time algorithms that can efficiently solve all instances. Traditional methods, including branch and bound and dynamic programming, generally exhibit exponential time complexity, specifically $O(n^2 2^n)$. Consequently, various approximation algorithms have emerged. This paper explores clustering techniques, which partition the set of cities into smaller clusters to facilitate more efficient solutions.

The GitHub repository for this project is available at: `https://github.com/spring1253/TSP_with_Clustering`.

2 Algorithms

The core idea is to cluster cities based on geographic proximity, enabling the application of traditional TSP solutions to each smaller group. The overall algorithm is outlined in **Algorithm 1**.

Algorithm 1 TSP With Clustering

Require: A set of cities C, Number of clusters k
Ensure: A tour that visits all cities

clusters, *clusterCenters* ← *ClusterCities*(C, k) ▷ Divide cities into k clusters and select centers
clusterTour ← empty list
for each *cluster* ∈ *clusters* **do**
    clusterTour.append(*SolveTSPTraditional*(*cluster*)) ▷ Solve TSP for each cluster
end for
centerTour ← *SolveTSPTraditional*(*clusterCenters*) ▷ Solve TSP for cluster centers
finalTour ← *CombineTours*(*clusterTour*, *centerTour*) ▷ Merge tours
return *finalTour*

We first divide the cities into smaller groups, or *clusters*. For each cluster, we solve the TSP using a traditional method. After generating the tours for each cluster, we construct a tour that connects the cluster centers using the same traditional TSP solution method. Finally, we combine the tours of the individual clusters with the tour of the cluster centers into a single comprehensive tour.

2.1 Clustering the Cities

To partition the cities, we utilize a greedy algorithm that iteratively selects cluster centers based on their distance from previously selected centers. This approach ensures the clusters are well distributed. The clustering algorithm is described in **Algorithm 2**:

Algorithm 2 Cluster Cities

Require: A set of cities C, Number of clusters k
Ensure: Cluster assignments and the set of cluster centers

selected ← empty list
pick ← *RandomPick*(C) ▷ Randomly select the first cluster center
C.remove(*pick*)
selected.append(*pick*)
while length of *selected* $< k$ **do**
    furthestCity ← *MaxDistanceCity*(C, *selected*) ▷ Select the furthest city as a new center
    C.remove(*furthestCity*)
    selected.append(*furthestCity*)
end while
for each *city* ∈ C **do**
    assign(*city*, *selected*) ▷ Assign each city to the nearest center
end for
return *selected*

This *k-center clustering* algorithm begins by randomly selecting a city as the first center. It then selects the city furthest from the chosen centers, repeating until k centers are selected. Each city is then assigned to its nearest cluster center. Alternative clustering methods, such as *k-means clustering*, may also be explored.

2.2 Solving TSP with Traditional Algorithm

The traditional TSP algorithm employed here uses dynamic programming with bitmasking, suitable for small to medium-sized graphs, with a time complexity of $O(n^2 2^n)$. Details are found in **Appendix A**.

2.3 Edge Removal for Path Creation

To convert cycles into a directed path that visits each vertex exactly once, we must remove one edge from each cycle. This transformation enables continuous traversal from one cluster to the next. The edge removal decision aims to minimize the total path distance based on the following formula:

$$\text{Distance} = \text{Distance}(P, \text{start}(edge)) + \text{Distance}(\text{end}(edge), N) - \text{Length}(edge)$$

The algorithm for edge removal is detailed in **Algorithm 3**:

Algorithm 3 Find Edge To Remove

Require: A cycle C of a cluster, Center of previous cluster P, Center of next cluster N
Ensure: The edge to remove

```
min_dist ← ∞
for each edge ∈ C do
    dist1 ← dist(P, start(edge)) + dist(end(edge), N) - length(edge)
    dist2 ← dist(P, end(edge)) + dist(start(edge), N) - length(edge)
    if dist1 < min_dist or dist2 < min_dist then
        min_edge ← edge
    end if
end for
return min_edge
```

Once edges from all cycles are removed, a cycle encompassing all cities is generated.

2.4 Results

Figure 1 visually represents the outcomes of our algorithm, illustrating scenarios with 50 vertices organized into 5 clusters.

The length of the cycle generated by our algorithm is $\text{SOL} = 6.739$, while the optimal solution yields $\text{OPT} = 6.019$, resulting in a ratio of $\text{SOL}/\text{OPT} = 1.120$.

3 Performance Evaluation

The objective of this experiment was to evaluate the performance of the clustering-based algorithm designed to solve the Traveling Salesman Problem (TSP). Specifically, we aimed to analyze how variations in **the number of clusters**, **the clustering method used**, and **the criteria for edge removal** affect computational efficiency and solution quality.

The performance was to be evaluated based on:

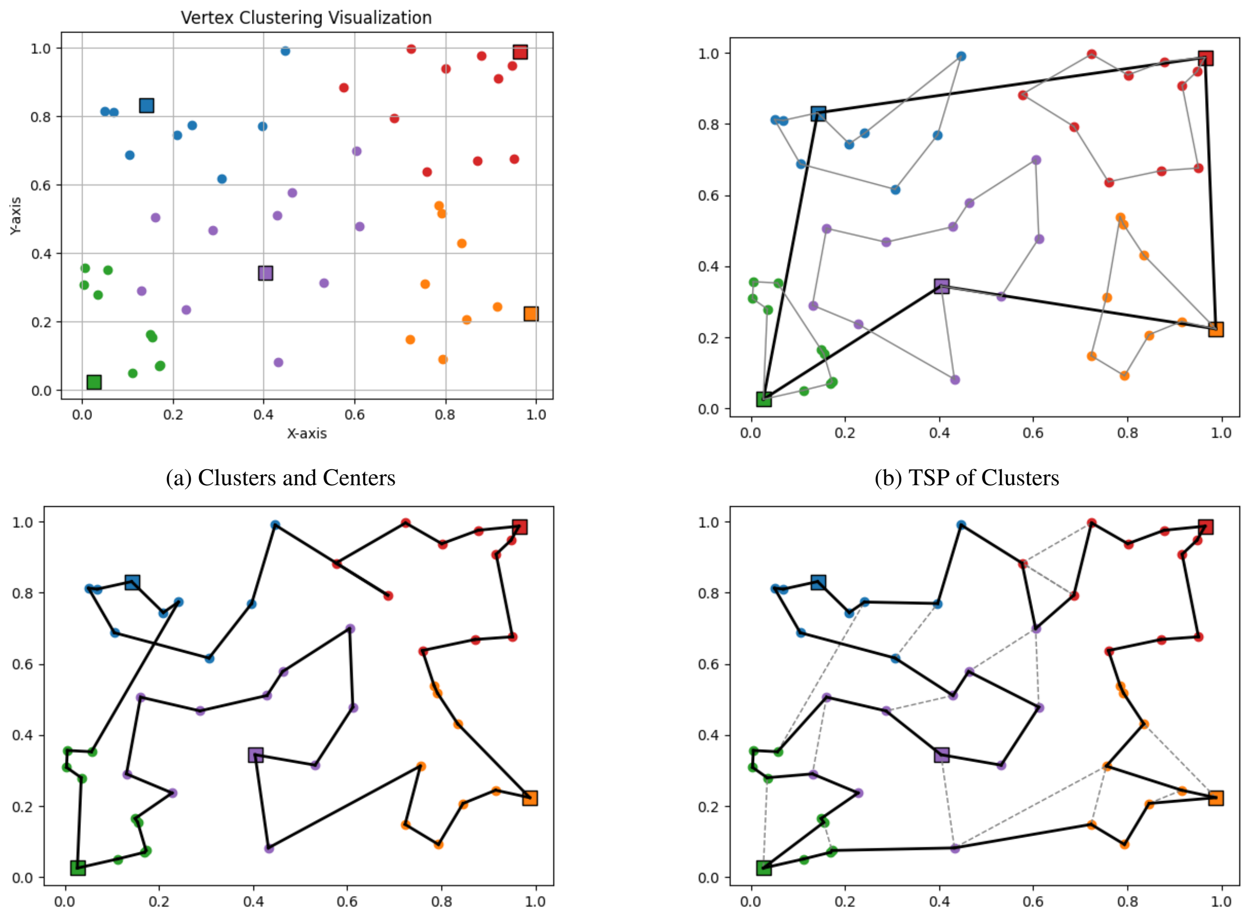
- **Execution Time**: The time taken to compute the TSP for each cluster and the overall tour.
- **Tour Length**: The total distance of the generated tour.

During the simulations, we encountered significant challenges:

- **Cluster Size Impact**: The algorithm's performance was highly sensitive to the size of the largest cluster. Given that traditional TSP solvers have a time complexity of $O(n^2 2^n)$, larger clusters led to substantial computational burdens, resulting in kernel crashes.
- **Limited Insights**: When clusters remained small, the results did not provide notable insights, complicating efforts to draw meaningful conclusions about the performance impacts of the various factors being studied.

Our evaluation involved varying the number of clusters on graphs with 100 vertices, with each configuration tested across three test cases. The experimental results are summarized in **Table 1**.

The results indicate that the algorithm's performance is sensitive to cluster size, with larger clusters leading to increased computational burdens and longer execution times.



(a) Clusters and Centers

(b) TSP of Clusters

(c) Combined Cycle

(d) Optimal Solution

Figure 1: Visual Representation of the Algorithm

Table 1: Performance Evaluation Based on Number of Clusters

# of Clusters	**Avg Approx. Ratio**	**Avg Running Time**
13	1.258	0.368 sec
14	1.230	3.173 sec
15	1.203	6.035 sec
16	1.189	1.486 sec
17	1.186	3.401 sec

4 Discussion and Future Work

While the clustering approach shows promise, challenges remain. The kernel crashes encountered with larger clusters indicate a need for improved handling of large datasets. Future work could include experimenting with alternative clustering algorithms, dynamically adjusting cluster size based on city density, and integrating more robust TSP heuristics.

Additionally, comparative analyses with other established heuristics like Genetic Algorithms or Ant Colony Optimization could further contextualize the effectiveness of our method.

A Traditional TSP Algorithm

```
def tsp_traditional(vertices):
    n = len(vertices)
    # Create a distance matrix
    distance = [[0] * n for _ in range(n)]

    for i in range(n):
        for j in range(n):
            distance[i][j] = vertices[i].distance_to(vertices[j])

    # memoization table
    memo = {}
    # To reconstruct the path
    parent = {}

    def dp(mask, pos):
        if mask == (1 << n) - 1:  # All vertices visited
            return distance[pos][0]  # Return to starting point

        if (mask, pos) in memo:
            return memo[(mask, pos)]

        ans = float('inf')
        best_next_city = -1
        for city in range(n):
            if mask & (1 << city) == 0:  # If city is not visited
                new_mask = mask | (1 << city)
                cost = distance[pos][city] + dp(new_mask, city)
                if cost < ans:
                    ans = cost
                    best_next_city = city

        memo[(mask, pos)] = ans
        parent[(mask, pos)] = best_next_city  # Store the best next city
        return ans

    # Start TSP from the first vertex
    min_cost = dp(1, 0)

    # Reconstruct the path
    path = []
    mask = 1
    pos = 0
    for _ in range(n):
        path.append(pos)
        next_city = parent.get((mask, pos))
        if next_city is None:  # In case there's no next city, break
            break
        mask |= (1 << next_city)
        pos = next_city

    # Convert indices to Vertex objects
    path = [vertices[i] for i in path] + [vertices[0]]  # return to start

    return min_cost, path
```