Spring 2023

A hierarchical approach to improve the ant colony optimization algorith

Bryan J. Fischer

A HIERARCHICAL APPROACH TO IMPROVE THE ANT COLONY OPTIMIZATION ALGORITHM

A Thesis

Presented To

Eastern Washington University

Spokane, Washington

In Partial Fulfillment of the Requirements

for the Degree

Master of Science in Computer Science

By

Bryan J. Fischer

Spring 2023

THESIS OF BRYAN J. FISCHER APPROVED BY

[illegible] DATE 6/14/2023

NAME OF CHAIR, GRADUATE STUDY COMMITTEE

[illegible] DATE 6/14/2023

NAME OF MEMBER, GRADUATE STUDY COMMITTEE

Jason W Ashley DATE 6-14-2023

NAME OF MEMBER, GRADUATE STUDY COMMITTEE

ABSTRACT

A HIERARCHICAL APPROACH TO IMPROVE THE ANT COLONY OPTIMIZATION ALGORITHM

by

Bryan J. Fischer

Spring 2023

The ant colony optimization algorithm (ACO) is a fast heuristic-based method for finding favorable solutions to the traveling salesman problem (TSP). When the data set reaches larger values however, the ACO runtime increases dramatically. As a result, clustering nodes into groups is an effective way to reduce the size of the problem while leveraging the advantages of the ACO algorithm. The method for recombining groups of nodes is explored by treating the graph as a hierarchy of clusters, and modifying the original ACO heuristic to operate on a hypergraph. This method of using hierarchical clustering is significantly faster than the original ACO algorithm, even when normal clustering techniques are applied, while producing improved tour lengths.

ACKNOWLEDGMENTS

Thank you to Dr. Bojian Xu, whose insights and thoughtful exploration helped make this project interesting and insightful.

Thank you to Dr. Stu Steiner, who gave endless encouragement and resources, and was always willing to help.

TABLE OF CONTENTS

List of Figures..5

1 Introduction..1

1.1 Organization of this Paper..1

1.2 The Traveling Salesman Problem..2

1.3 Simple TSP Solving Algorithms..2

1.4 Heuristic-based TSP Solver..4

1.5 Applications of TSP Solvers..5

1.6 Problem Statement..5

2 Ant Colony Optimization Algorithm (ACO)..6

2.1 ACO Overview..6

2.2 Parameter Tuning..11

2.3 Path Stagnation..12

2.4 Multi-Agent Communication Via Pheromones..13

3 Related Work..13

3.1 Definition of an Ant Colony Optimization Algorithm..14

3.2 MAX-MIN Ant System..14

3.3 K-means Clustering..15

3.4 Improvement in path stagnation and cycle computations..17

4 Methodology..17

4.1 The Algorithm..18

4.2 Clustering..31

4.3 Hierarchical approach and Top-Down Strategy..32

4.4 An Alternative Technique to Improve Hierarchical ACO..32

5 Experiments and Results..34

5.1 Experiment 1: Normal v. Hierarchical ACO on Clustered Graph, k = 2........35

5.2 Experiment 2: Hierarchical vs. Normal ACO, k = 4..38

5.3 Experiment 3: Hierarchical vs. Normal ACO, k = 6..41

5.4 Experiment 4: Hierarchical vs. Normal ACO, k = 16..44

5.5 Experiment 5: Hierarchical vs. Normal ACO, eil76, k = 4..47

5.6 Experiment 6: Hierarchical vs. Normal ACO, att532, k = 4, k = 16..51

6 Discussion and Conclusion..57

7 Future Work..59

Vita..62

List of Figures

Figure 1: greedy algorithm....4
Figure 2: transition probability.... 8
Figure 3: pheromone updates....9
Figure 4: addition of pheromones to edge.... 10
Figure 5: hypergraph traversal start.... 20
Figure 6: hypergraph traversal step two....22
Figure 7: hypergraph traversal step 3....23
Figure 8: hypergraph traversal step 4....24
Figure 9: hypergraph tour highlighted.... 25
Figure 10: hypergraph cycle is complete.... 26
Figure 11: local tours completed....28
Figure 12: hypergraph edges removed....29
Figure 13: clustered graph k=2.... 35
Figure 14: tour-length vs cycle count, normal ACO....35
Figure 15: lower-left region.... 37
Figure 16: upper-right region....37
Figure 17: whole-graph combined ACO....37
Figure 18: generated graph, k=4.... 38
Figure 19 and 20: k=4 normal ACO.... 39
Figure 21 and 22: k=5 hierarchical.... 40
Figure 23: generated graph, k=6.... 41
Figure 24 and 25: normal aco tour results, k=6.... 42
Figure 26 and 27: hierarchical tour results, k=6.... 43
Figure 28: generated graph, k=16.... 44
Figure 29: normal aco tour results, k=16.... 45
Figure 30: hierarchical tour results, k=16.... 46
Figure 31: eil76 graph....47
Figure 32 and 33: eil76 normal aco tour results.... 48
Figure 34: cluster data from eil76 k = 4....50
Figure 35: nodes per cluster....50
Figure 36: att532 graph....51
Figure 37 and 38: att532 normal aco tour results.... 52
Figure 39 and 40: hierarchical ACO on att532, k = 4, k-means clustering.... 53
Figure 41: cluster data att532, k=4.... 54
Figure 42: notes per cluster att532, k=4.... 54
Figure 43 and 44: hierarchical ACO on att532, k = 16, k-means clustering.... 55
Figure 45: cluster data, att532, k=16.... 56

Figure 46: nodes per cluster, att532, k=16...56

1 Introduction

The Ant Colony Optimization algorithm takes inspiration from the natural behavior of an ant colony foraging for food and applies this behavior to approximate a solution to the Traveling Salesman Problem. This type of nature-inspired behavior is a heuristic that follows a set of rules to probabilistically traverse a complete graph, completing tours, and returning the shortest tour found. It has been shown to be very effective in producing a good approximate solution in most cases, but is slow to complete for large graphs. Since its inception in the early 1990's, the ACO has been iterated on using various strategies, improving both its quality of solution and running time [7]. This thesis proposes a new improvement to the ACO that is shown to provide better quality results at a significantly lower time cost on large graphs.

1.1 Organization of this Paper

Section 1 will discuss the Traveling Salesman Problem, why it's important, and issues with solving it outright. Section 2 explains how the original Ant Colony Optimization operates. Section 3 will give a brief survey about other ways in which the ACO has been modified to improve its performance. Section 4 will describe the implementation details of the hierarchical version of the ACO and explain how it functions. Section 5 gives the results of the implementation proposed in this thesis compared to the original ACO described in section 2, and section 6 will conclude with discussions about the results. Finally section 7 will propose ideas for future work and areas to explore using this new alteration of the ACO.

1.2 The Traveling Salesman Problem

The Traveling Salesman Problem can be described as finding the shortest tour in a graph starting from a home location, visiting every other location exactly once, and then returning back home [6]. An example that is keeping in spirit with the algorithm's namesake: a salesman must leave the sales office and visit several client locations before returning back to the sales office. What is the most efficient way to order each sales meeting such that distance traveled is minimized?

1.3 Simple TSP Solving Algorithms

If the number of locations is small, it is trivial to produce a solution to this problem. Consider a set of locations *n* in a complete graph, the solver must brute-force check every permutation of tours starting and stopping from some home location, and keep track of the shortest tour [3]. The brute-force algorithm is defined roughly as picking a starting home location, determining each tour length from that location to all other locations, then returning the shortest tour found [9]. Three discrete examples are given below:

$n = 2$

{a, b}

a, b, a
b, a, b

$n = 3$

{a, b, c}

a, b, c, a
a, c, b, a
b, a, c, b
b, c, a, b
c, a, b, c
c, b, a, c

$n = 4$

{a, b, c, d}

a, b, c, d, a	b, a, c, d, b	c, a, b, d, c	d, a, b, c, d
a, b, d, c, a	b, a, d, c, b	c, a, d, b, c	d, a, c, b, d
a, c, b, d, a	b, c, a, d, b	c, b, a, d, c	d, b, a, c, d
a, c, d, b, a	b, c, d, a, b	c, b, d, a, c	d, b, c, a, d
a, d, c, b, a	b, d, a, c, b	c, d, a, b, c	d, c, a, b, d
a, d, b, c, a	b, d, c, a, b	c, d, b, a, c	d, c, b, a, d

Each increase in location count, *n*, causes the number of potential tours to increase dramatically. Solving the traveling salesman problem (TSP) using this method has a time complexity of O(n!) [7]. While the advantage to this method is that we are guaranteed to find the shortest tour possible, the disadvantage is that as *n* grows larger and larger, it becomes too costly to solve. For example, if *n* = 100, there are greater than 9.3326×10^{157} possible tours to check.

It's worth noting that going forward, we are assuming symmetric graphs, where the distance from a to b is the same as from b to a, in which case the number of tours to check is halved, but this does not affect the overall time complexity.

Another method of solving the TSP is by using a greedy algorithm, where each next location is chosen based solely on its distance. The greedy algorithm states that we always choose the nearest location next. This poses an advantage in that it is quick to solve the TSP, the disadvantage is that the solution can be very poor; the shortest cycle found using a strictly greedy algorithm has no guarantee about its quality.

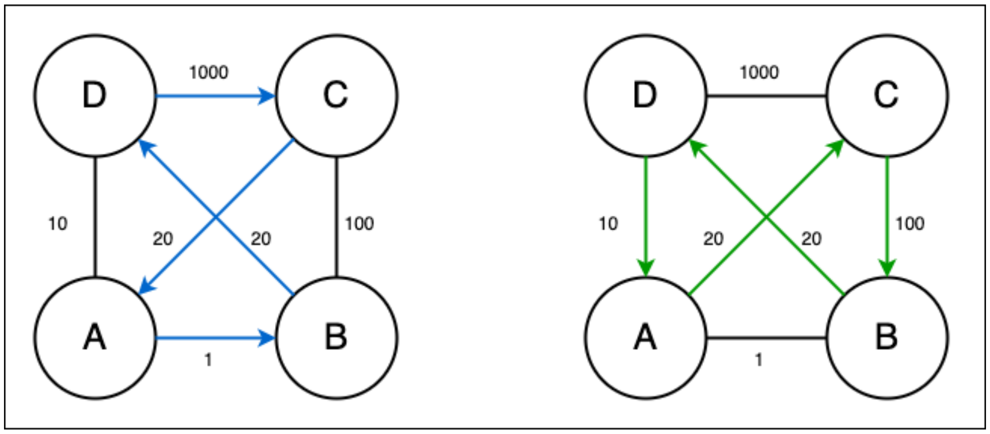


Figure 1: greedy algorithm

An example of where the greedy based approach breaks down is exposed in figure 1. Assuming a starting location at vertex A in a complete graph, always picking the closest vertex next would result in a tour length of 1041 (shown in blue (left): A-B-D-C-A). This is far from an optimal route of length 150 (shown in green (right): A-C-B-D-A). The only way to guarantee we have obtained the shortest cycle in a symmetric graph is to brute-force every possible pathway through the graph and report back the shortest cycle found. There is no known polynomial time algorithm to solve the TSP optimization problem [9].

1.4 Heuristic-based TSP Solver

A heuristic based approach is often employed to approximate a solution to the traveling salesman problem. The Ant Colony Optimization (ACO) algorithm is a heuristic that finds a good approximate solution to the TSP in a relatively short amount of time [5]. It does this by taking inspiration from nature, specifically the emergent behavior of ants foraging for food. It relies on a global communication system via pheromone trails, as well as bias parameters that guide agents, referred to as ants, as they traverse the graph. The general ACO algorithm will be explained in depth in section 2. Heuristic based TSP algorithms are employed because they are significantly faster to find a solution without sacrificing too much quality in their resulting tour.

1.5 Applications of TSP Solvers

The TSP can be applied to any optimal route finding problem that requires traversal of a graph and as such can be useful in many situations. It has been adopted to find the most efficient routing of power lines, wiring in circuits, determining the most efficient route for delivering goods between multiple locations, and flight paths for aircraft [9]. TSP algorithms are popular due to their applicability to many real-world scenarios.

1.6 Problem Statement

We have so far explored what the TSP is and how to solve it using an exhaustive search of every potential solution in factorial time. We then were introduced to the ACO, a generalizable heuristic to approximate an optimal solution to the TSP. The problem this thesis aims to address is how to further improve the general ACO algorithm using top-down hierarchical techniques that will provide both quicker results and shorter tour lengths.

2 Ant Colony Optimization Algorithm (ACO)

M. Dorigo is the research director for the Belgian Funds for Scientific Research, and is credited as the inventor of the Ant Colony Optimization algorithm. In his seminal work titled "The Ant System: Optimization by a colony of cooperating agents," Dorigo describes a heuristic based approach to approximate a solution to the TSP [5]. The following subsections describe his work and the ACO algorithm known as the Ant-System, or AS for short.

2.1 ACO Overview

Taking inspiration from nature, the ACO uses many individual agents, known as ants, to traverse a graph completing cycles. To relate this using similar nomenclature to the TSP, imagine we have a certain number of cities we must visit and, instead of one salesman traveling to every city by himself, we have multiple salesmen all trying different paths looking for the shortest tour. A tour in this context indicates that a salesman has traveled from their starting location, visited each city exactly once, and then returned home. Now imagine that we have a number of salesmen equal to the number of cities, and one salesman is placed in each city. This is something akin to the starting condition of the ACO algorithm. Note that depending on the literature cities, nodes, and vertices are interchangeably used. Also, a tour refers to the path an ant has taken when it completes a route through the graph and returns back home. When all ants have completed their respective tours, this is known as a cycle.

In the ACO, each ant takes a different route through the graph, adhering to the rules of the TSP (visiting each vertex only once, then returning home to complete a cycle). Once all the ants have completed their respective tours, they update the paths they took with a metric called pheromones which increase the desirability of that path relative to the shortness of their cycle. This means that each edge traversed by an ant agent will have some amount of pheromones deposited on that edge that is proportional to that ant’s tour length: shorter paths get more pheromones deposited on each edge in the path. At the end of each cycle, when every ant has concluded their movements and returned back home, the pheromones for each edge are evaporated by some amount, rho, and then a new amount of pheromone is added based on the ants that traversed that edge. The result of this action is that a more desirable path, denoted by the pheromone intensity on edges, emerges after several cycles of the ACO have completed. An edge with a higher value of pheromones means it is very often used by ants to construct their tours.

Ants choose their next movement based on two factors: desirability and distance. These factors are then given weights, alpha and beta, to influence the ant agent's decisions. The alpha weight is related to the pheromone importance, and the beta weight is related to the distance, or visibility importance. We can then change the values of the alpha and beta weights to make the ants behave differently–favoring closer vertices similar to a greedy algorithm, or favoring more popular paths that contain higher concentrations of pheromones.

When an ant makes a decision to move to the next vertex it examines several circumstances. First, an ant cannot travel to a city it has already visited. This can be kept track of via a list of visited cities. Second, the ant will examine all possible next movements and calculate the probability of visiting each city using the following formula from Dorigo [7]:

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{\left[\tau_{ij}(t)\right]^{\alpha} \cdot \left[\eta_{ij}\right]^{\beta}}{\sum\limits_{k \in allowed_k} \left[\tau_{ik}(t)\right]^{\alpha} \cdot \left[\eta_{ik}\right]^{\beta}} & \text{if } j \in allowed_k \\ 0 & \text{otherwise} \end{cases}$$

Figure 2: transition probability

The probability an ant will choose to travel to city j from city i is determined by the strength of the pheromones present on edge (i,j) raised to the power of alpha (the bias for desirability), multiplied by the distance to city j raised to the power of beta (the bias for visibility). This is then divided by the sum of all possible movements using the same calculation. In other words, the numerator from figure 2 is computed individually for every potential choice. All those choices are then added together to form the denominator, and then each choice is evaluated to form a set of probabilities for each potential choice. If, for example, some choices are more favorable than others, they will have a much higher chance of being chosen by a given ant. Ants then roll a random number to determine which path to take, with some paths being more likely to be picked than others. The "t" represents time, which is discrete in this model and is iteratively increased after every ant completes a full cycle. The benefit of choosing movements in this fashion is that ants have the potential to deviate from the path and explore new pathways, even if it has a small chance. This exploration v. exploitation behavior allows the ants to search for better pathways throughout the graph at random, while converging on pathways that are favorable. The datasets are complete graphs, meaning each vertex in the graph has an edge to every other vertex. This means that each ant has the possibility to visit any location in the graph from any other location, but using this heuristic makes it less likely for ants to pick far away locations or edges that are seldom traveled.

Initially, the number of ants is equal to the number of vertices (or cities) in the graph, with one ant being placed at each vertex. This has been shown experimentally to be an ideal starting condition, and in fact using more ants will not necessarily create better solutions due to stagnation behavior: where ants become entrenched in a pathway and seldom deviate [7]. Similarly, more ants will not necessarily improve the outcome because at the start of each cycle, the state of the graph remains unchanged in terms of edge pheromone level and distances between cities. Then each ant completes their entire tour independent of other ants, and updates the graph for the next cycle. Ants are running individually, either successively or concurrently, but do not affect each other until the ants complete a cycle.

After each cycle, the pheromone values of each edge are reduced by the evaporation rate, **ρ** < 1.0, and the pheromones from each ant are added to each edge they traversed based on their route length [7].

$$\tau_{ij}(t+n)=\rho \cdot \tau_{ij}(t)+\Delta\tau_{ij}$$

Figure 3: pheromone updates

The pheromone intensity, **τ**, of edge (i,j) at time **t** is first evaporated by some proportional amount, **ρ**, and then given additional pheromones depending on the quality of the tour found by each ant that visited that edge, indicated by **$\Delta\tau$** [5].

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{Q}{L_k} & \text{if k - th ant uses edge } (i, j) \text{ in its tour (between time t and t + n)} \\ 0 & \text{otherwise} \end{cases}$$

Figure 4: addition of pheromones to edge

Figure 4 showcases how, if an ant k moved along edge (i, j), it will add some amount of pheromone to that edge depending on the length of its tour. Q represents a constant amount of pheromone that could be applied to an edge, and L_k represents the overall length of the tour found by k^{th} ant. For example, if ant k found a tour length of 50, and we have a constant Q value of 100, every edge traversed by ant k_1 for cycle t will first have it's existing pheromone level evaporated by some amount **ρ**, then have 2 added to it. Similarly, if another ant k_2 found a tour length of 75, every edge visited by ant k_2 will have it's pheromone levels decreased by evaporation **ρ,** then increased by 1.5. It becomes clear to see that for multiple ants over multiple cycles, the evaporation effect will cause unused trails to disappear while more popular trails will become stronger. The rate at which a trail becomes more popular is based on the quality, or shortness, of the tour length. This is how the emergent behavior of the ACO occurs.

Dorigo explains that the time complexity for running the ACO is $O(NC \cdot n^2 \cdot m)$ where NC is the number of cycles we choose to run the algorithm, n is the number of cities, and m is the number of ants [7]. Dorigo further explains that there is a linear relationship between the number of ants and the number of towns (vertices) for optimal results, which makes the overall time complexity $O(NC \cdot n^3)$ when the number of ants equals the number of towns.

2.2 Parameter Tuning

One of the biggest challenges with using the ACO is finding the best parameter settings for the algorithm. It is often thought that if you understand the theory of an algorithm you can leverage that knowledge to improve it. Although this is true, the ACO is somewhat unique in that it is widely studied experimentally [4]. Recall we have several parameters when setting up the ACO:

α - Alpha biases the importance of pheromone levels. The higher the alpha bias the more importance the ant places on the popularity, also referred to as desirability, of a given path. A consequence of setting the alpha bias too high is that path stagnation will occur quickly, as ants will not want to leave the most popular paths to explore the graph.

β - Beta biases the importance of visibility, or distance, for a given path. If the beta bias is set to be very large, the ants tend to always want to travel to the nearest city with each movement. A consequence of setting the beta bias too high is the algorithm shifts to be similar to a strictly greedy-based approach. This was outlined in section 1.2 and often doesn't produce the best results.

ρ - Rho is the evaporation rate and directly impacts how quickly ants settle into a convergence. The value of rho is set to between (0.0, 1.0), with higher evaporation rates making less popular trails disappear quicker. Rho directly impacts how much the ants will explore as the cycle count increases. When the evaporation rate is high, cycles will stop producing new routes earlier. When a pathway's pheromone level evaporates to almost zero, it is practically impossible for ants to pick this path anymore and these edges are sometimes removed as candidate choices altogether [7].

Number of ants - Typically used in a linear relationship with the number of cities, adding more ants than cities has a similar effect to running more cycles.

Number of Cycles - A constant value that determines when the algorithm should stop. Depending on the other parameters and the layout of the graph, there is a point of diminishing returns where an increase in cycle count does not yield significantly better results. Since the ACO does not exhaustively search the graph for a known shortest tour, the impetus is placed on the cycle count to determine when to stop searching.

Experimentally, Dorigo has shown that good values for these parameters are α=1, β = 5, ρ = 0.5, with the number of ants equaling the number of cities. The maximum cycle count, at least experimentally, is set to a large value to better observe the behavior of the algorithm and see when it stops producing better results. This value can vary depending on the data set being studied, but typically falls somewhere between 500 and 5000 cycles [3]. The ACO algorithm runs in $O(n^3)$ time so there are practical limits to consider as well. Dorigo, using a 30-city test, used NC = 5000 to conclude that the above values are optimal for this ACO strategy [5].

The main focus of this thesis is to introduce new methods for applying the ACO in a layered, hierarchical way and will adhere mostly to the experimentally optimal parameters found by Dorigo [5].

2.3 Path Stagnation

Although the number of cycles may be set to a very large value, the ants will often find their best shortest tour fairly early. Depending on the size and composition of the graph, this could happen within the first 50 to 100 cycles, or for larger graphs, a few hundred cycles. When ants no longer produce better results, or when it takes a very long time to produce marginal improvements, this is known as path stagnation. The focus of this thesis will be on running a constant amount of cycles with statically set alpha and beta values to showcase the improvements of a hierarchical, cluster based approach over the traditional ACO.

2.4 Multi-Agent Communication Via Pheromones

One of the big advantages of the ACO algorithm is that the ant agents communicate to each other via a global pheromone table. This means that when a cycle starts, each ant looks at the static graph to make all decisions throughout a cycle and does not need to wait on other ants to traverse specific edges of the graph. This allows each ant agent to run autonomously in parallel relative to other ants. Pheromone values are stored in a table whose indices are edges in the graph. This table only gets updated by each ant at the end of each cycle, so during the most computationally expensive task of graph traversal the ants can run in parallel.

3 Related Work

The original ACO was a groundbreaking strategy to approximate a solution to the TSP, and many people have since altered or extended the work of Dorigo in an attempt to improve the algorithm's performance in various ways. Of these improvements I will focus on a few implementations that are related to the work proposed in this thesis, and give a greater context to the landscape of improvements to be found by modifying the original ACO.

3.1 Definition of an Ant Colony Optimization Algorithm

The term Ant Colony Optimization Algorithm has become a fairly broad term that could be applied to many different types of algorithms that involve solving the TSP using multi-agent strategies. Many TSP solvers share a similar foundation of multiple agents traversing a graph, but have been extended to be very different from one another. I will outline three other ACO algorithms in the following subsections: the MAX-MIN Ant System developed by Thomas Stutzle and Holger Hoos, an ACO algorithm that utilizes K-Means clustering written by Yen-Ching Chang, and finally work by Zhang who has proposed a method on reducing computation cost based on pheromone trails.

3.2 MAX-MIN Ant System

The MAX-MIN Ant System (MMAS) is similar to the original ACO presented by Dorigo except that at the completion of each cycle, only the ant with the best solution is allowed to add pheromones to the graph [12]. The benefit to this approach is that the emergence of short tours happens quicker than normal. Since we have already observed that at the beginning of the algorithm the ants have no differentiating information about the cities favorability other than distance, only allowing the best ant to change the pheromone table causes other ants to bias towards that pathway earlier. This also means that if an ant decides to explore outside of this initial pathway, if it is not an improvement in tour length it will be disregarded more quickly than in the standard ACO model. A downside to this method is early stagnation [12]. The MMAS will quickly fall into a tour that happens to have been found early in the cycle count since exploration becomes almost impossible after a few iterations, depending on the value set to rho. The authors of the MMAS combat this problem by introducing limits to pheromone levels within the graph. In the original ACO, if the pheromone levels drop below some minimum threshold that they are so close to zero that there is virtually no chance of an ant choosing that path, then the edge is removed from the graph entirely [5]. This is appropriate in the original ACO algorithm with proper parameter tuning, however in the MMAS this leads to poor results. The MMAS sets a minimum and maximum value for pheromones on each edge in the graph; this ensures that although only the best ant can increase the pheromone levels on a path of edges, it never stops other ants from having the possibility of exploring other pathways [12]. By setting the lower limit of pheromone values to be greater than zero it always allows for exploration among the ant agents. In addition, by setting an upper limit to pheromone values it causes the graph to maintain a constant level of pheromone intensity for good pathways, without allowing a single pathway to dominate too much, while letting bad pathways fade to the minimum. This is advantageous because it encourages ants to explore even after many cycles have progressed and a best tour has likely been found. The authors of the MMAS show that they are able to find a nearly optimal solution to many TSP datasets using this approach and claim that it is a considerable improvement in performance over the original ACO.

3.3 K-means Clustering

In the context of the using cities, or vertices in a graph, K-means clustering is employed as a method of grouping cities together based on their position in the overall graph. Cities that are close to one another are placed in the same group, or cluster, and the clustering produces k number of clusters. This can be beneficial when applied to the

ACO because the time cost is cubic, meaning if we can run the ACO on smaller groups of cities we can potentially save time overall.

In his paper titled “Using K-Means Clustering to Improve the Efficiency of Ant Colony Optimization for the Traveling Salesman Problem” Chang proposes a method of running the ACO on smaller individual clusters and then combines the clusters together to create the overall tour [3]. In this method, Chang uses k-means clustering on the location data of each city, the x and y coordinates, and groups them into several clusters. Then for each cluster, he ran the ACO algorithm to find a local shortest tour. To recombine the clusters into one larger global solution, he finds the two cities, one from each cluster, who are closest to each other and creates an edge. Then checks the neighbors of those two cities within their respective tours to find the next smallest distance between the two clusters and creates another edge. He then cuts the edges out of the local tours to form one larger result. If there are multiple smaller clusters that need to be joined, the Chang suggests repeatedly joining clusters one by one until we are left with a single large global tour.

This method is beneficial in that the time to run the ACO algorithm on multiple smaller clusters is much less than on the single large graph. Also, by using k-means clustering, it is guaranteed that local clusters will never make long traversals to the opposite side of the graph and will instead stay within their region, exhausting all cities to visit before leaving to the next cluster. The author concedes that how they make connections between clusters could be further investigated. The results however, show an improvement over the traditional ACO alone by producing roughly the same tour lengths but running about ten times as fast.

It's worth noting that the benefits for this approach are more apparent when the city count is large. For a city count of 30, the traditional ACO can generally find a better solution in a shorter time, but for a city count of 76 Chang's clustering method performs better in both time and result [3]

3.4 Improvement in path stagnation and cycle computations

There has been some work into figuring out how to prevent path stagnation, such as work by Zhang who introduces an improved form of the ACO specifically designed to reduce the processing costs of ants [8]. Their work introduces two methods for solving this problem. The first approach describes that as soon as a single ant has found a new shortest-path in a cycle, all other ants stop working and the cycle ends. Second, they allow each ant to employ their own route finding strategies that better benefit the changing landscape of pheromones on the graph [8]. They argue that Dorigo's suggestion of α=1, β = 5 is initially a good choice as ants have little knowledge other than distance to cities. But after the graph has been shaped by pheromones over several cycles, Zhang suggests shifting the bias the other direction gradually until α = 5, β = 1. The effect of this change is quicker convergence time and consequently less processing of the algorithm.

4 Methodology

The solution proposed by this thesis involves using the traditional ACO described in section 2, as well as clustering cities using k-mean clustering, and solving the problem of joining clusters together by taking a top-down approach combined with the ACO heuristic to find connections between clusters.

The ACO has been investigated thoroughly since its introduction, the difficulty comes with improving the algorithm using novel methods. Due to the time complexity being cubic, it seems natural to reduce the search space of the ACO as an easy method for increasing performance. The most straightforward way is to group nodes that are close to one another and force a local tour to be found within that cluster before moving on. The k-means clustering work from section 3.3 shows one way to accomplish this and does indeed provide good improvements over the original ACO. This approach belies the difficulty in joining individual clusters of the hypergraph without introducing too much complexity or unnecessary processing, while still gaining performance improvements over the original algorithm. This thesis resolves these difficulties using a top-down approach on a hypergraph created by performing k-means clustering, and joining local tours using a modified version of the traditional ACO algorithm.

4.1 The Algorithm

To properly describe the algorithm we must first give some definitions of terms. A hypergraph is a graph where each pair of vertices is connected via a set of edges, not just a single edge [1]. A hypergraph tour is a tour which visits every vertex in the hypergraph exactly once and returns back to the starting vertex, adhering to the normal TSP rules. Similarly, a local tour is a tour connecting each vertex within a particular cluster, and a global tour is a tour connecting all vertices in the graph.

The algorithm works as follows:

1. Cluster the vertices using k-means clustering.
2. Treat each cluster as a single node in the hypergraph.
3. Run the normal ACO on the hypergraph with the following modifications:

  a. Ants may not visit each cluster more than once in a hypergraph tour, except for the starting cluster to return home.
  b. Upon visiting a hypergraph node, which is a cluster of vertices from the original graph, the ant must enter and exit the cluster using different vertices–it cannot enter and exit the cluster using the same vertex.
  c. Once the ant enters and exits a cluster at the hypergraph level, the local vertices in the cluster that were used to enter and exit the cluster are merged together. This merge happens by forcing the edge connecting the two vertices to be used in the local tour later.

4. Run a specified number of cycles on the hypergraph until we have a satisfactory result at the hypergraph level. At this point, the hypergraph nodes will be connected forming a shortest tour among clusters.
5. Run the normal ACO on each individual cluster. When an ant reaches one of the merged vertices from the hypergraph processing, it will automatically move to its connected vertex on the next move.
6. Once the specified number of cycles has been completed on each local cluster, unmerge the local vertices revealing the overall global cycle.

A trace of the algorithm is shown below:

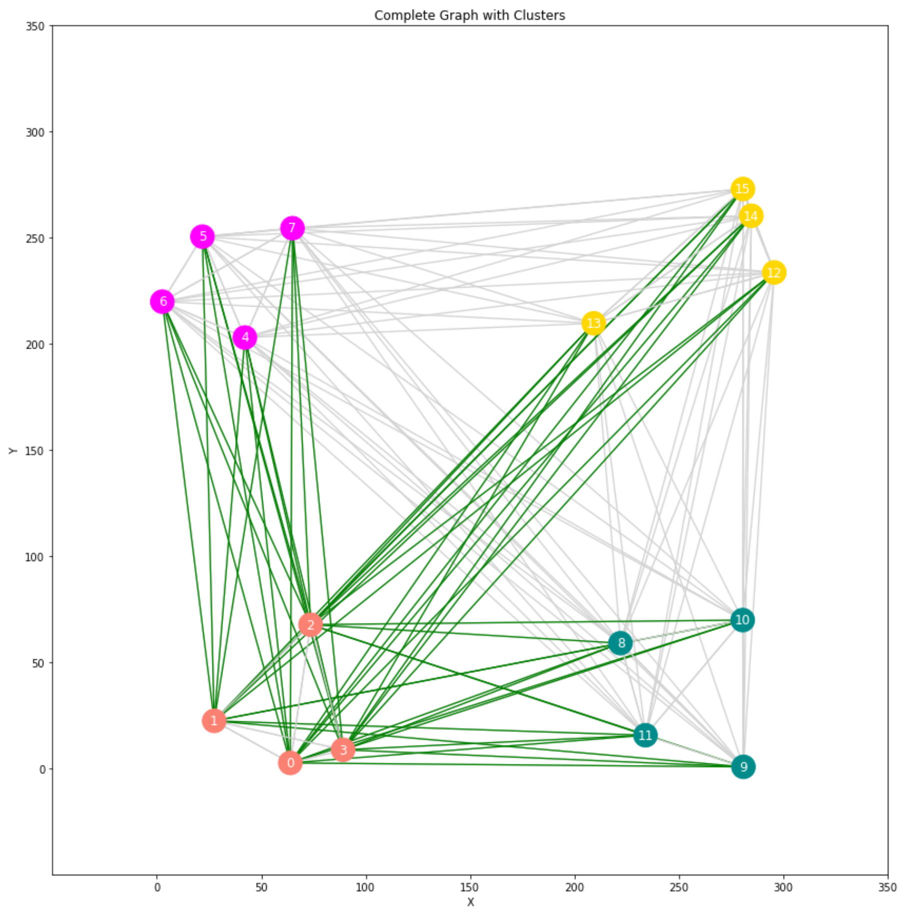


Figure 5: hypergraph traversal start

In this example, the graph consists of 16 nodes grouped into 4 clusters. Each cluster is the same size and is represented by a different color. Nodes 0 through 3 are in the “orange” cluster located at the bottom left, nodes 4 through 7 are in the “magenta” cluster in the top left, nodes 8 through 11 are in the “cyan” cluster in the bottom right, and nodes 12 through 15 are in the “yellow” cluster located in the top right. For exposition purposes, we can assume that we have already performed the clustering step on the graph (step 1) and arrived at these four clusters. Step 2 of the algorithm states we must treat each cluster as its own individual node in the hypergraph. If we view this graph as a hypergraph we will see only 4 nodes, one for each color, with multiple edge connections between them. For step 2, we use one ant per hypergraph node, and traverse the hypergraph.

Arbitrarily lets begin in the orange cluster located at the bottom left, shown in figure 5. The ant will look at all available edges it is allowed to traverse, represented in green, and choose one edge. Notice that the ant cannot move to another local node located in its own cluster, this is because we are operating at the hypergraph level and it must move between two distinct clusters. The normal ACO algorithm is employed to calculate a set of probabilities for the ant to choose its next move among all allowed choices in green. This is step 3a of the algorithm.

Step 3b is represented by figure 6, where our ant has decided to move from node 2 to node 4. Notice, that upon making this traversal, the ants' next traversal choices have changed. It cannot visit any vertex within a cluster it has already visited, nor can it pick any vertex located within the cluster it currently resides. Finally, it cannot choose to leave the magenta cluster using the same vertex it arrived with, eliminating all edges connected to vertex 4. The blue line represents the traversed edge, while the green lines represent possible choices based on the ants current cluster location. Notice that the ant can now choose to visit either the yellow or the cyan cluster from any node within the magenta cluster except node 4.

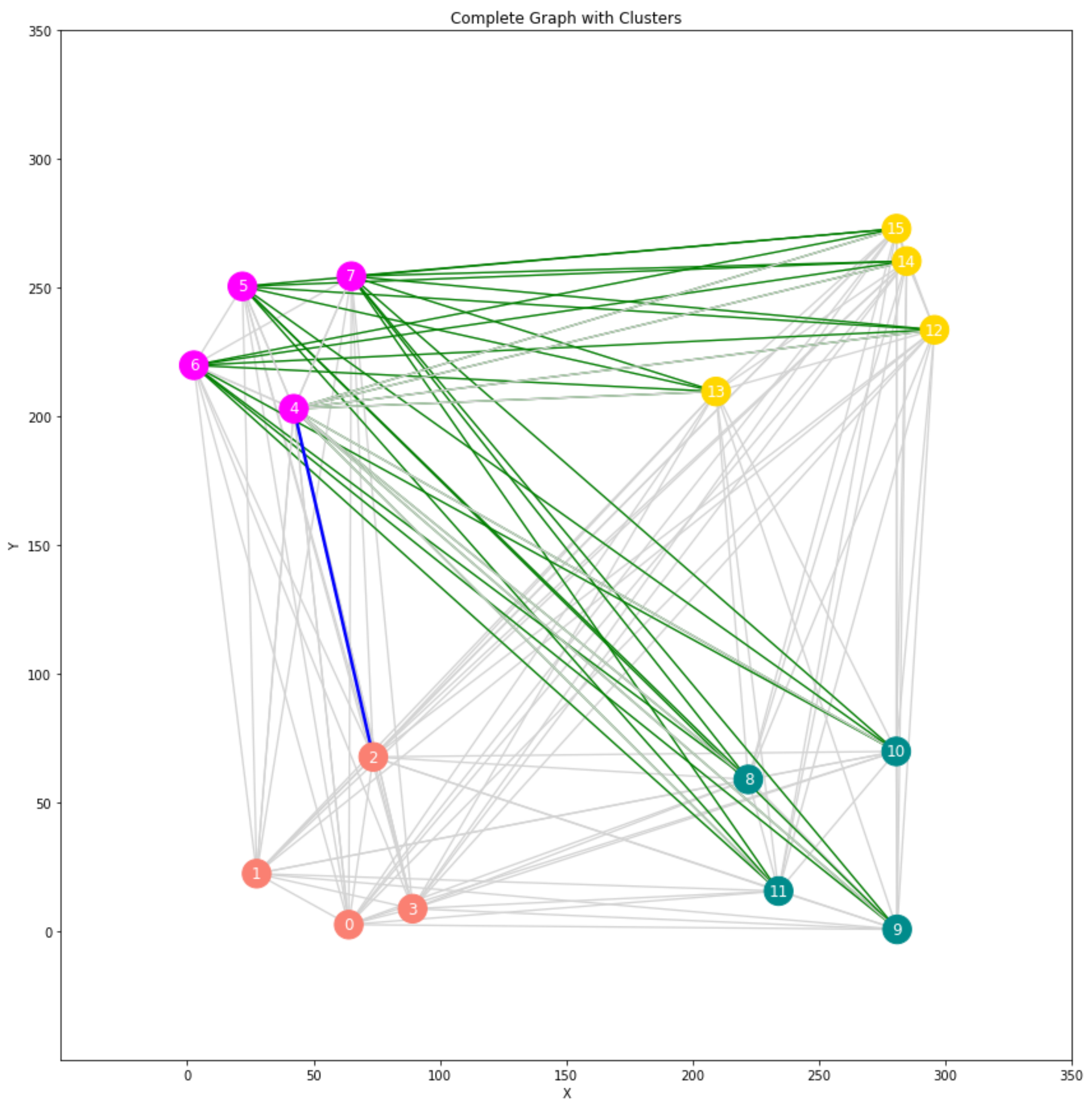


Figure 6: hypergraph traversal step two

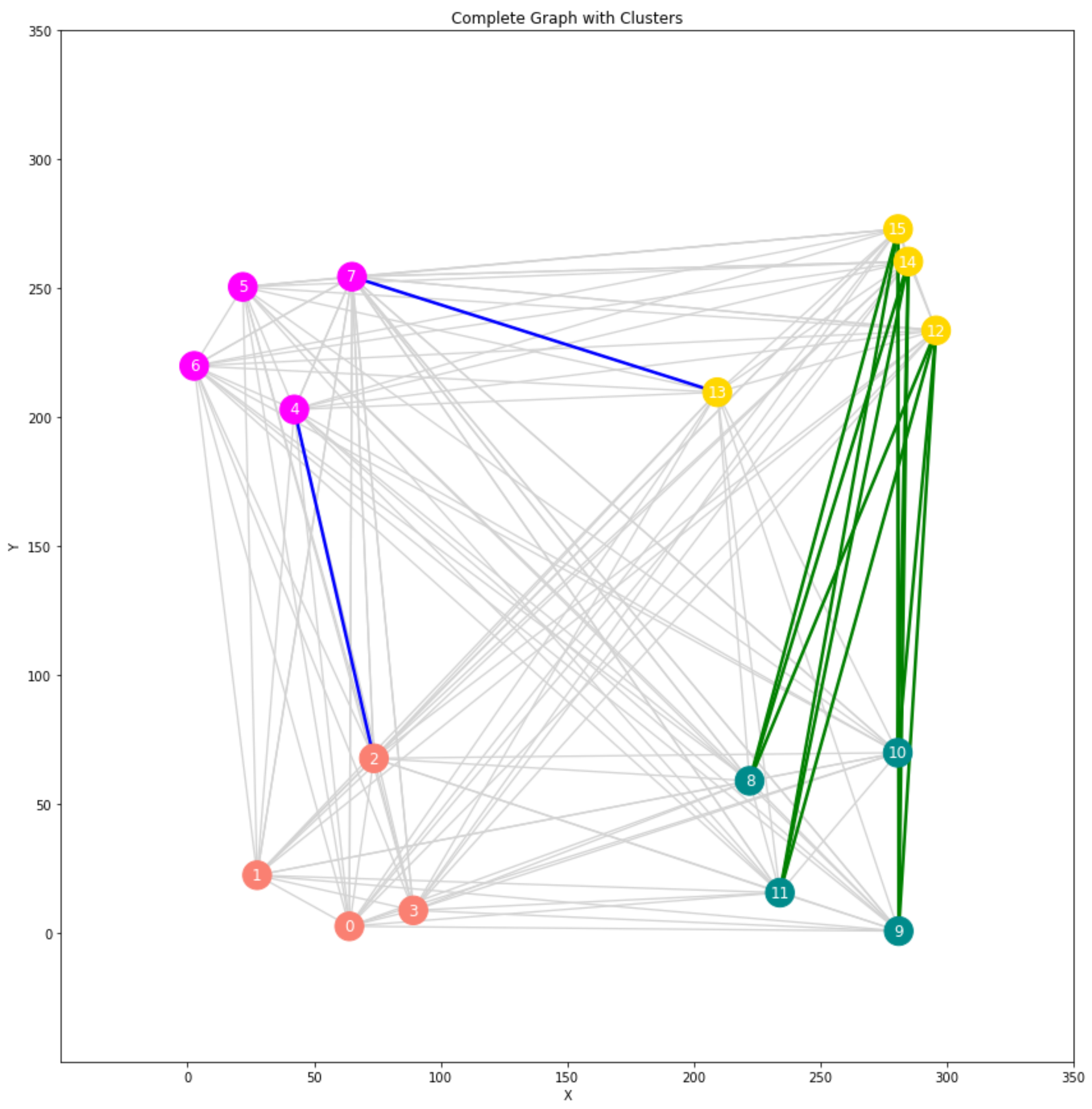


Figure 7: hypergraph traversal step 3

Suppose our ant decides to move to the yellow cluster via edge (7,13). We can see this traversal represented in blue, and see the possible choices for our ant have again changed. Adhering to the rules of the algorithm, the ant cannot visit the magenta cluster, and it can't return home to the orange cluster because it has not traversed all other clusters yet. The only option is for the ant to visit the cyan cluster in the bottom right using any edge whose source vertex is 12, 14, or 15. Again, it generates a set of probabilities based on normal ACO heuristics and makes a choice.

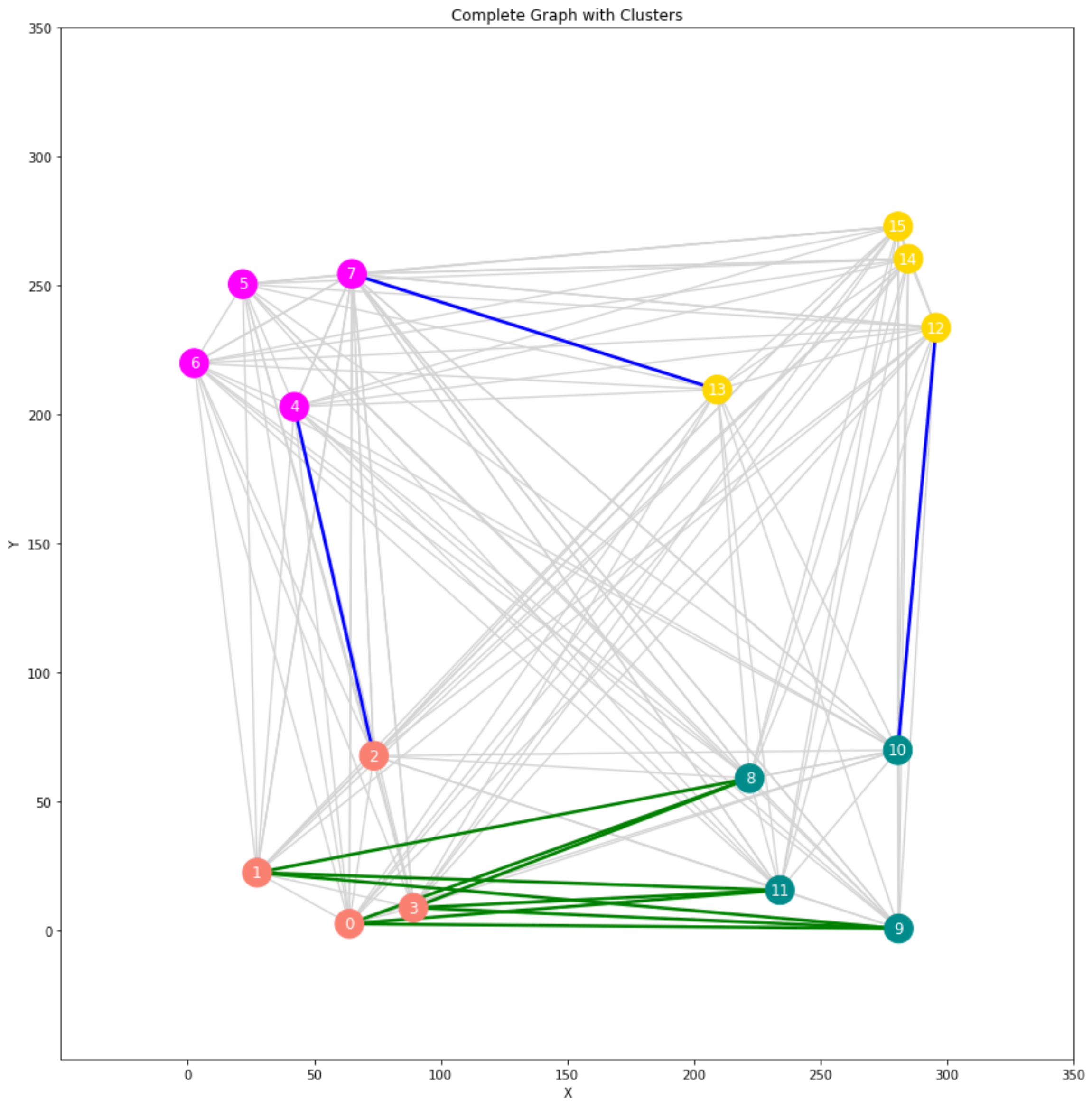


Figure 8: hypergraph traversal step 4

Figure 8 now shows the ant has visited all 4 nodes in the hypergraph and must now return home to the orange cluster. Notice, it must choose an edge that starts at 8, 9, or 11, and ends at either 0, 1, or 3. This is to avoid violating step 3b.

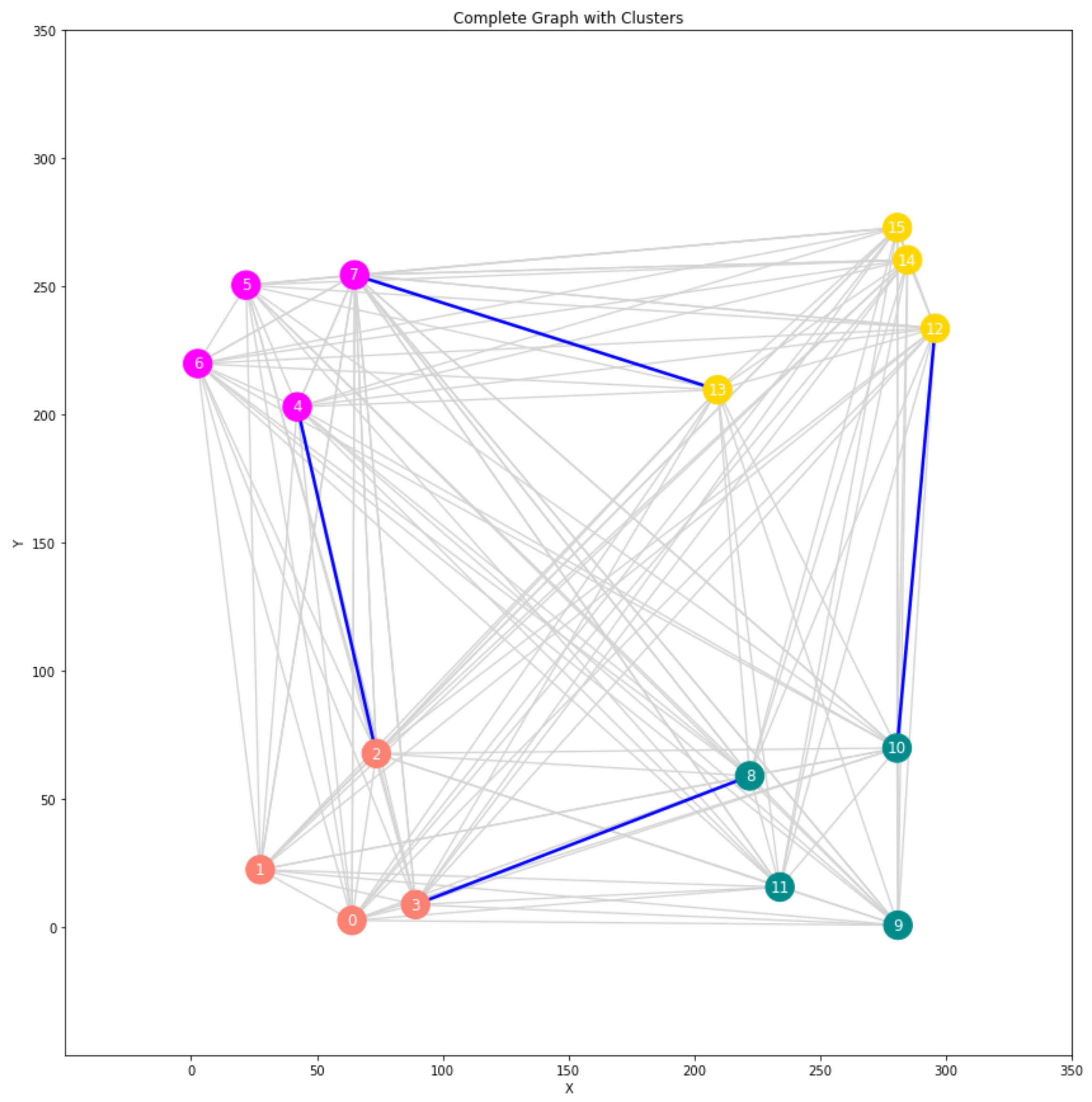


Figure 9: hypergraph tour highlighted

Figure 9 depicts a complete traversal of the hypergraph and the completion of one cycle. The pheromone levels on the blue edges will be increased proportional to the tour length, and all pheromone levels on the gray edges will evaporate by some amount defined by rho. This algorithm already differs from some of the related works discussed earlier in that we have not yet run the ACO on the local clusters and only dealt with the top-level hypergraph. The ant will report back the results of its tour and determine if it has found a new shortest tour in the hypergraph or not. This process will repeat for the specified number of cycles given to the algorithm, just as it did in the original ACO. It's worth noting that the computation cost of operating at the hypergraph level is higher than at the local level.

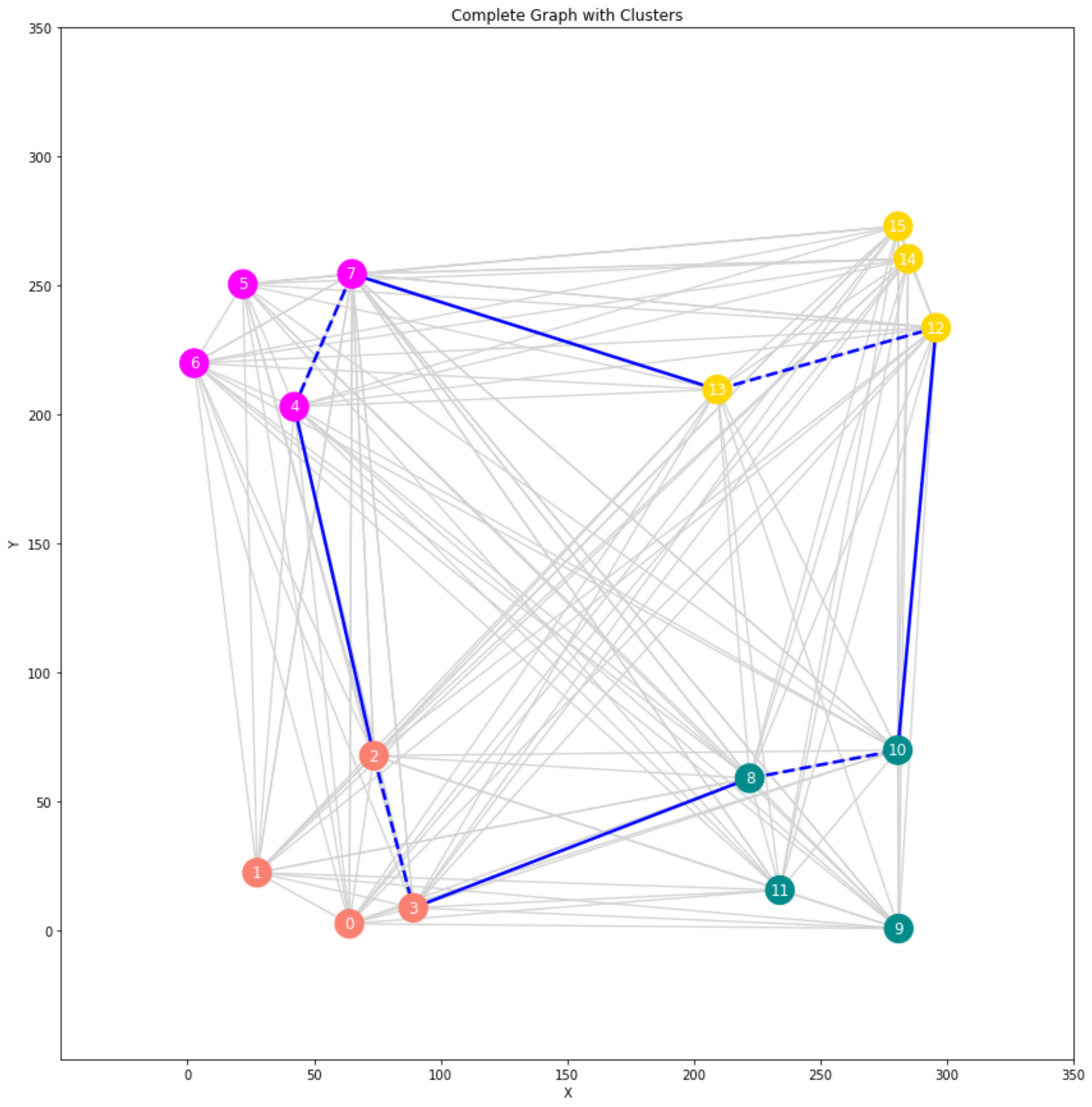


Figure 10: hypergraph cycle is complete

The final step of the hypergraph processing is merging the nodes, represented in figure 10 by the dashed blue line. This is an attribute that will assist the algorithm later when it does the local processing. In step 5, each local cluster is treated individually and the traditional ACO is run to find the shortest tour length among the nodes within that cluster. Before running the ACO on each local cluster, the pheromone levels of all edges inside each cluster are reset to the ACO starting values. The ACO algorithm has not been modified other than the extra logic of dealing with the merged nodes. When an ant reaches a node that has the merged flag set, it does not need to calculate a set of possible locations to move to, it simply chooses the next location based on the node that has been merged. In the example given in figure 10, if an ant is searching for a local tour within the orange cluster and reaches node 3, it will immediately move to node 2 on the next iteration.

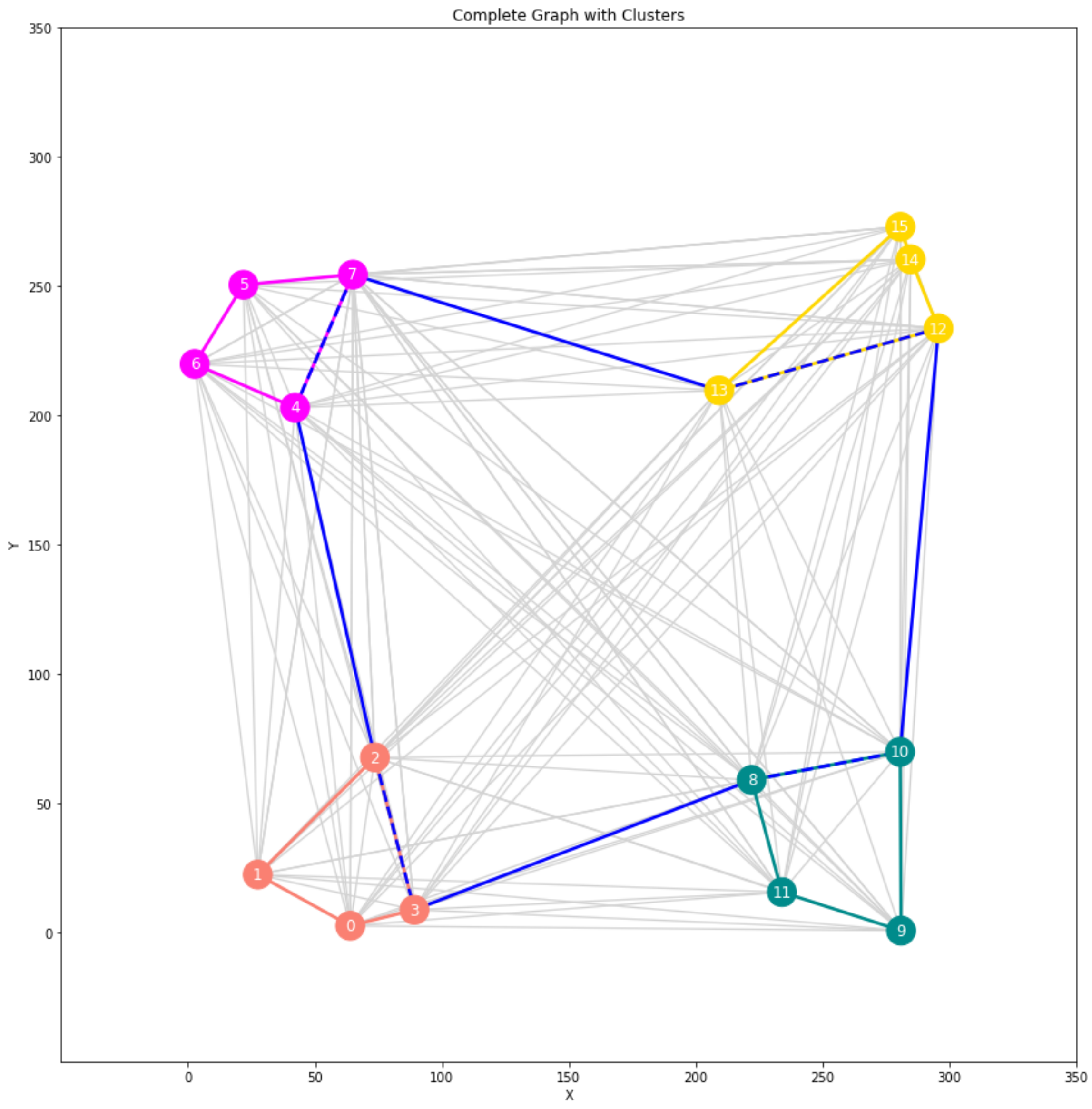


Figure 11: local tours completed

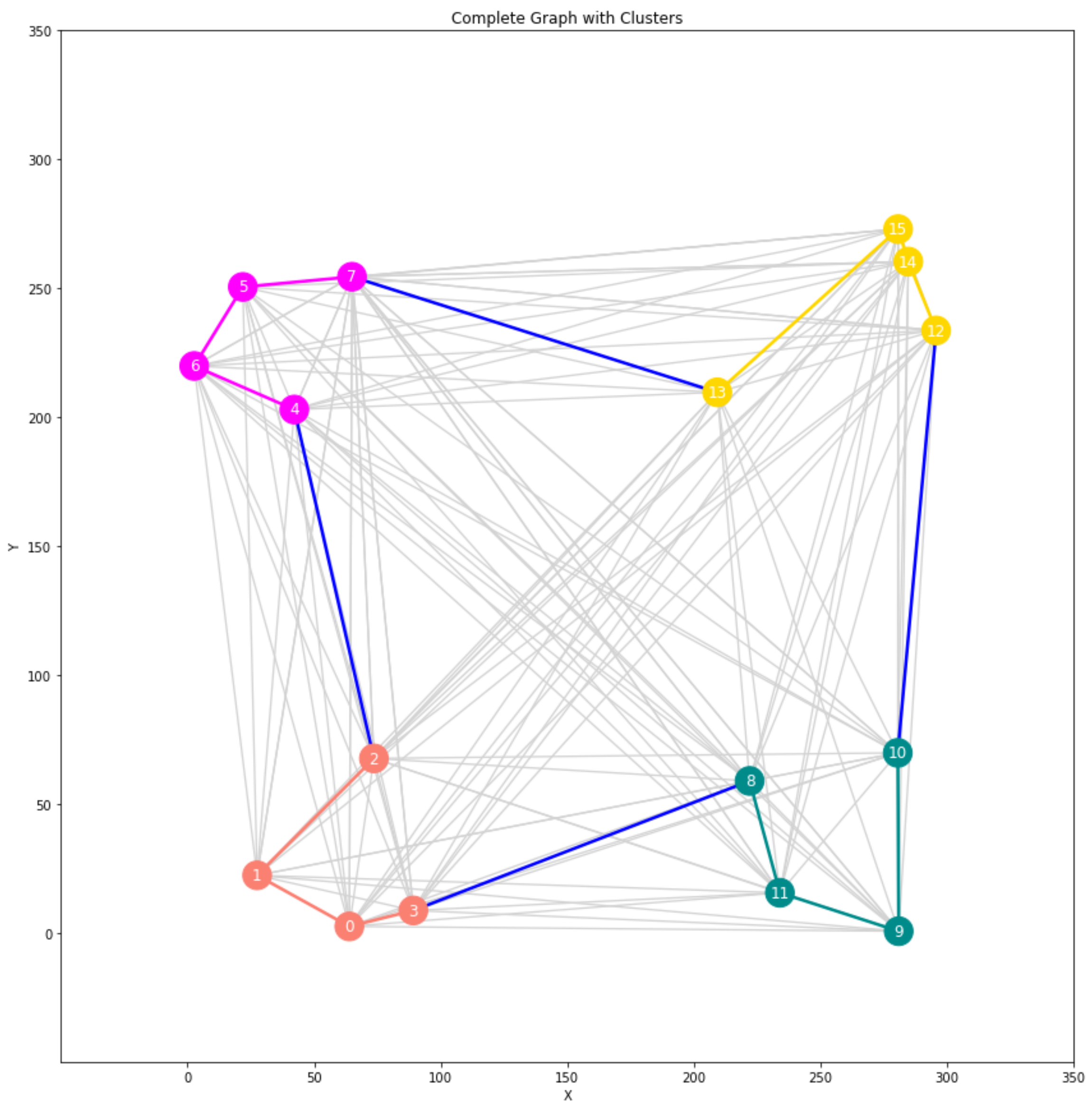


Figure 12: hypergraph edges removed

Figure 11 shows the result after steps 1-5 have completed. We have found a local tour in each cluster representing the shortest tour length discovered by the ants, and we have the shortest tour length connecting each cluster together from the hypergraph ACO work. Finally in step 6 edges connecting the merged nodes are removed from the graph, and we are left with a global tour as our solution, seen in figure 12.

4.2 Node Merging

A very specific and difficult problem arises when trying to merge clusters of a graph together–it's not always clear which edges should be cut to join multiple small tours together. Moreover, to adhere to the rule of the TSP that each node must be visited exactly one time before returning home, you cannot enter and exit a local cluster using the same node. If this occurs, you would visit that node twice: once upon entering the cluster and once upon leaving the cluster. This is solved by forcing the ant agents to enter and exit a cluster using different nodes.

The problem of joining local tours together to form a larger global tour is solved through merging nodes. When we run the ACO at the local level, and it creates a local tour within a cluster, we are forcing the traversal between the merged nodes so that later when we recombine the local tours we can unmerge these nodes without conflicts. Think of merged nodes as temporary bridges that are created for the local cluster processing, but are later destroyed leaving two open “ends” through which we join the global solution. This strategy makes the work of joining multiple clusters together very simple. Note that merging nodes is only a flag on the node object, indicating that the ant’s next traversal choice is already chosen–it will travel to the node it has merged with.

4.2 Implementation

This algorithm was implemented using Java, and all work was tested on a M1 Macbook Pro with 16GB of memory. Originally, it seemed that storing the distances and pheromone data of the edges in 2d arrays would be an obvious choice for implementation, but instead I ended up using a more object-oriented approach. Objects were created for most of the main parts: Ants, Nodes, Graphs, NodePairs, ACO, Parser and Logger. This was beneficial because it allowed the work to be compartmentalized easily. When dealing with multiple clusters, and multiple levels (local level and hypergraph level) it becomes difficult to organize the work. With an OOP approach, I could assign a Node to a specific cluster, then using that cluster attribute run the ACO with relatively minor changes so that it will properly travel from cluster to cluster rather than node to node. Having nodes as objects also allowed the ants to store a candidate list of nodes in an arraylist that could grow and shrink or be cleared as needed. This was immensely helpful in the processing and debugging of the program. The ACO was an object that called the actions of other objects, such as telling ants to make their next movements and running a set number of cycles. The graph object contained, among other things, a hashmap of nodes where their keys were the node ids and the values were the nodes themselves. This allowed for fast lookup of a node's attributes based on its id. Since a lot of the work happens when an ant is trying to determine which next move to make, using a hashmap to lookup node information was a natural choice. Ants also were relatively self-contained, they held all the logic of moving from node to node and had attributes that would record their pathway and the length of their current tour. The parser object allows graph data to be read from a file and proved helpful in benchmarking, and the logging object could record events with fine detail for analysis later.

4.2 Clustering

When testing this version of the ACO, artificial graphs were created using random numbers where clusters were explicitly formed, such as the graph from section 4.1. In addition, the algorithm was tested against various datasets that exist from TSBLib, an online repository of traveling salesman problems [13]. A dataset would be loaded into

RapidMiner and then k-means clustering would be used to process the data and create k clusters for use in the algorithm.

4.3 Hierarchical approach and Top-Down Strategy

This approach uses two levels: the hypergraph level and the local intra-cluster level. Although there are only two levels, the idea could be expanded to be *n* levels, where nodes could be combined to form clusters, then those clusters could be combined to make even larger clusters, etc. By working from the lop level to the bottom level the work could be expanded to any number of levels as might fit best for the dataset being used. The top down strategy avoids nodes being visited more than once, leverages the ACO heuristic to recombine local tours to form larger tours, and is extensible to multiple levels. An example of this could be routing through population centers. Houses could be clustered together into neighborhoods, neighborhoods could be clustered together to form cities. Cities could be clustered within states, and states could be clustered into countries. Using this hierarchical approach, the ACO would first find the optimal route to visit each country, then go down a layer into the state level. Within each state, the aco would find the optimal tour to visit each city, then within each city find an optimal tour to visit each neighborhood and finally each house. The final product would be one large global tour that visits every house in the dataset.

4.4 An Alternative Technique to Improve Hierarchical ACO

To review the problem statement, it is shown that since the ACO algorithm is an n^3 algorithm, by reducing the number of nodes we can greatly decrease the time cost to run the algorithm. A simple way to reduce the number of nodes is by clustering geographically similar nodes together, creating multiple smaller tours, then recombining all smaller tours into one large global tour. As previously discussed, the method used to recombine clusters into one large global tour, without violating the rules of the TSP and visiting a node more than once, is a challenge. This section provides a brief explanation about the alternative methods to solving this problem and why the proposed solution outlined in this thesis is preferred.

The first approach is a “lazy” bottom up approach, where the ACO was run on each individual cluster first, creating multiple local tours, then using the pheromone data from these smaller runs and rerunning the ACO on the global graph. The idea behind this experiment is that by using the pheromone data from each local cluster it would influence the ants to want to remain inside a local cluster before leaving it. The upside of doing this approach is it deviates very little from the standard ACO and is easy to implement. Experimentally, ants did not always stay inside local clusters when forming their global tours and would jump all over the graph. The results between clustered and non-clustered ACO experiments were nearly identical, with no real improvement using the clustered approach.

The second approach is to connect each cluster in the graph using the shortest edges between each cluster. Intuitively it makes sense that if you want to form a global tour you would use the shortest edges between each cluster. Then within each cluster the ACO is run, forming a local tour. The problem with this approach is there is no guarantee that the in-edge and out-edge of a cluster lay on distinctly different nodes in the cluster. Meaning it’s possible for a node to be visited more than once in the process of forming this global tour. Other works have suggested this approach and the solution relies on cutting and joining adjacent edges to make the TSP work [3].

5 Experiments and Results

Experiments were run using graphs that were randomly generated by Java's built-in Random function [10], and graphs that were found from TSPLib [13]. The random graphs were generated such that they produced nodes that were grouped together to form deliberate clusters. This was done to showcase the performance of the hierarchical ACO algorithm under ideal circumstances. In addition, the datasets found from TSPLib were used due to their popularity in benchmarking TSP algorithms [11], and were clustered using RapidMiner and k-means clustering. The results of each experiment are shown below followed by a table of results.

Each of the experiments used the following parameter settings:

$\boldsymbol{\alpha} = 1.0$

$\boldsymbol{\beta} = 5.0$

$\boldsymbol{\rho} = 0.5$

Ant count = node count

Cycle count: listed in each result table

5.1 Experiment 1: Normal v. Hierarchical ACO on Clustered Graph, k = 2

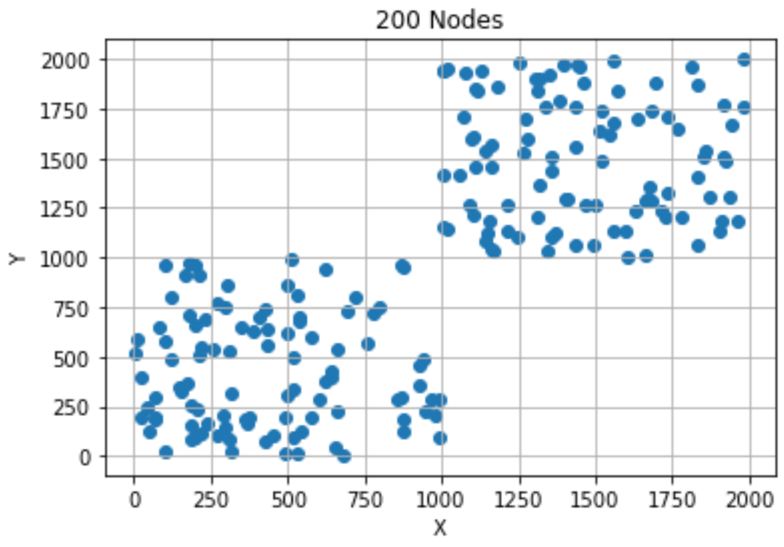


Figure 13: clustered graph k=2

The graph is randomly generated using Java’s built-in Random class to produce two distinct regions of nodes: 100 nodes in each region with a total of 200 nodes. The first test involves running the traditional ACO on the graph as-is, for a total of 1000 cycles (NC = 1000).

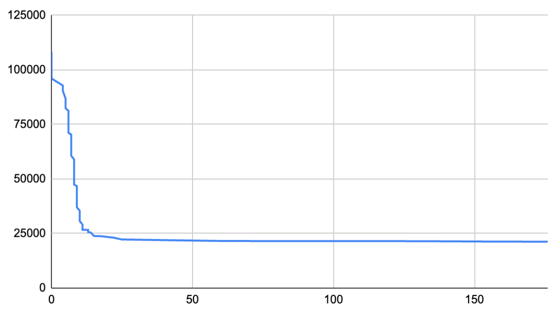


Figure 14: tour-length vs cycle count, normal ACO

Non-clustered ACO	
Shortest Tour Length	21270
Cycle Count	1000 cycles
Elapsed Time	196 seconds

Notice, the ACO algorithm stagnates at around 176 cycles meaning no new paths were found even though 1000 cycles were completed. In an attempt to show more granularity, graphs showing tour-length v. cycle count will omit the trailing regions if no new data is available to be presented.

Next we examine the hierarchical approach. First the ACO is run on the lower-left region for 500 cycles, then the upper-right region is run for 500 cycles. These two sections are run independent of each other. Finally the ACO is run using the overall graph, including the saved pheromone data from each region's previous runs, for 500 cycles.

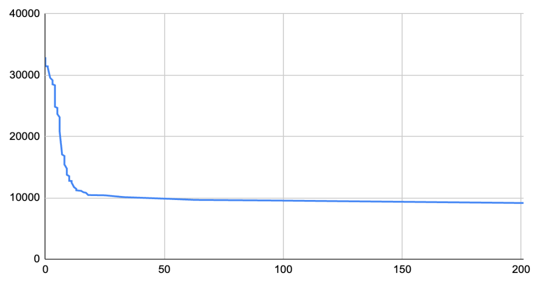


Figure 15: lower-left region
Tour-length vs cycle count
500 cycles x 100 nodes, 25 seconds

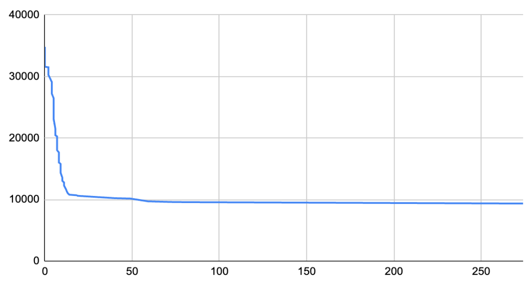


Figure 16: upper-right region
Tour-length vs cycle count
500 cycles x 100 nodes, 34 seconds

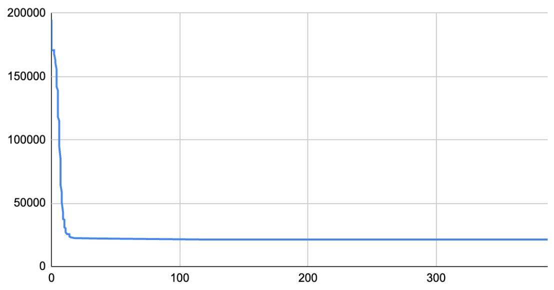


Figure 17: whole-graph combined ACO

Tour-length vs cycle count
500 cycles x 200 nodes, 130 seconds

Clustered ACO - Non-hierarchical Approach		
Shortest Tour Length	21354	**+84 cycle length**
Cycle Count	500 cycles for each cluster 500 cycles for global graph	2(500*100) + (500*200)
Elapsed Time	189 seconds	**-7 seconds, 0.03% less time**

Empirically this experiment reveals that there is not much difference in time or quality using the clustered approach compared to the non-clustered approach with the traditional ACO method.

5.2 Experiment 2: Hierarchical vs. Normal ACO, k = 4

Graph has random points divided into four clusters, with 25 nodes in each cluster.

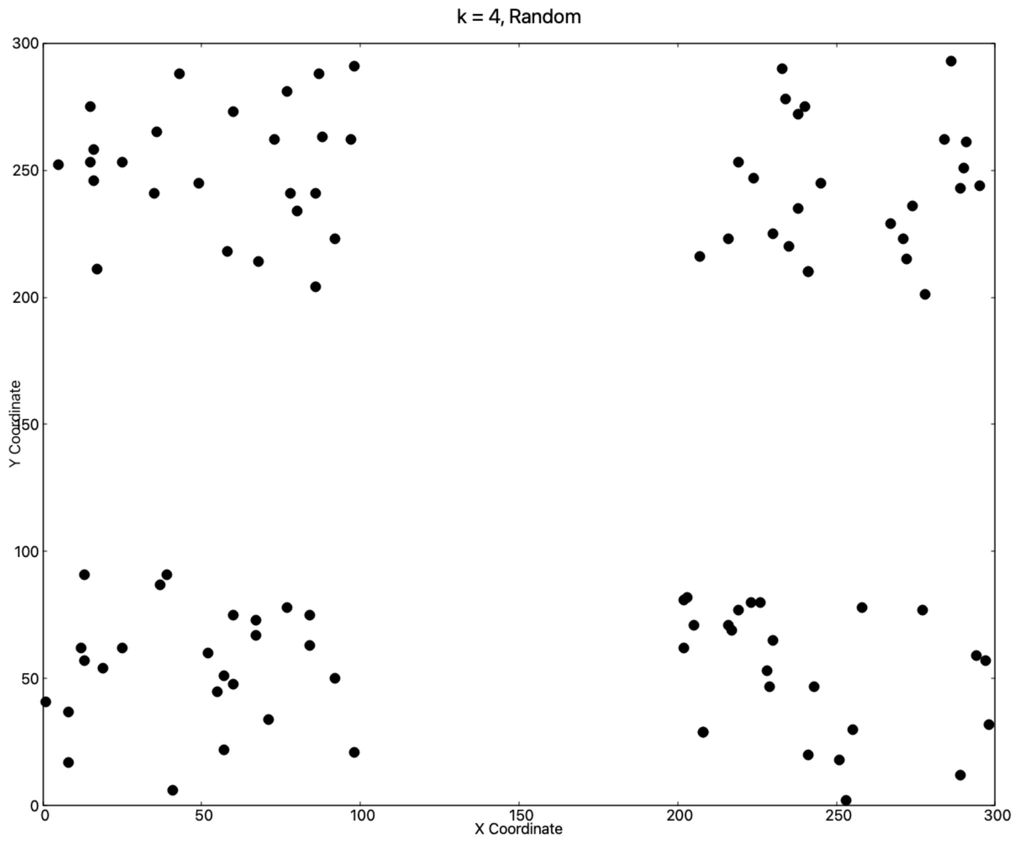


Figure 18: generated graph, k=4

Normal ACO results:

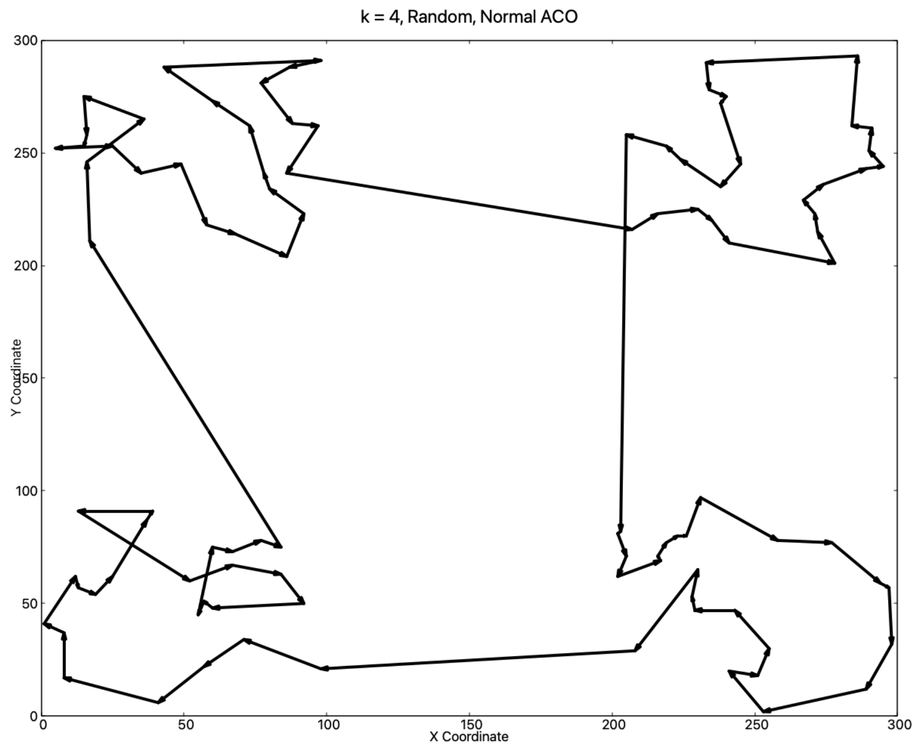
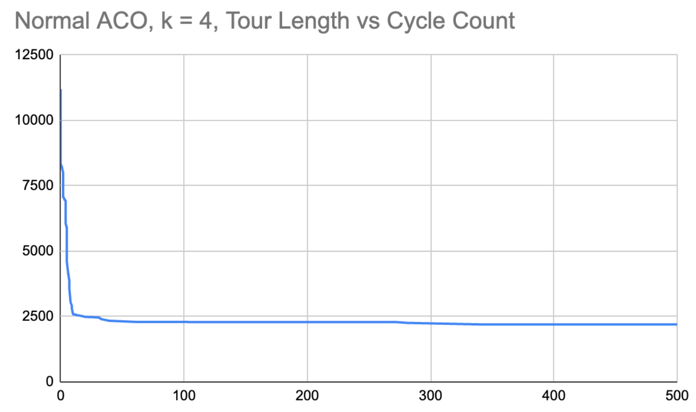


Figure 19 and 20: k=4 normal ACO



Normal ACO, k = 4	
Shortest Tour Length	2198
Cycle Count	500 cycles
Elapsed Time	81.02 seconds

Hierarchical ACO Results

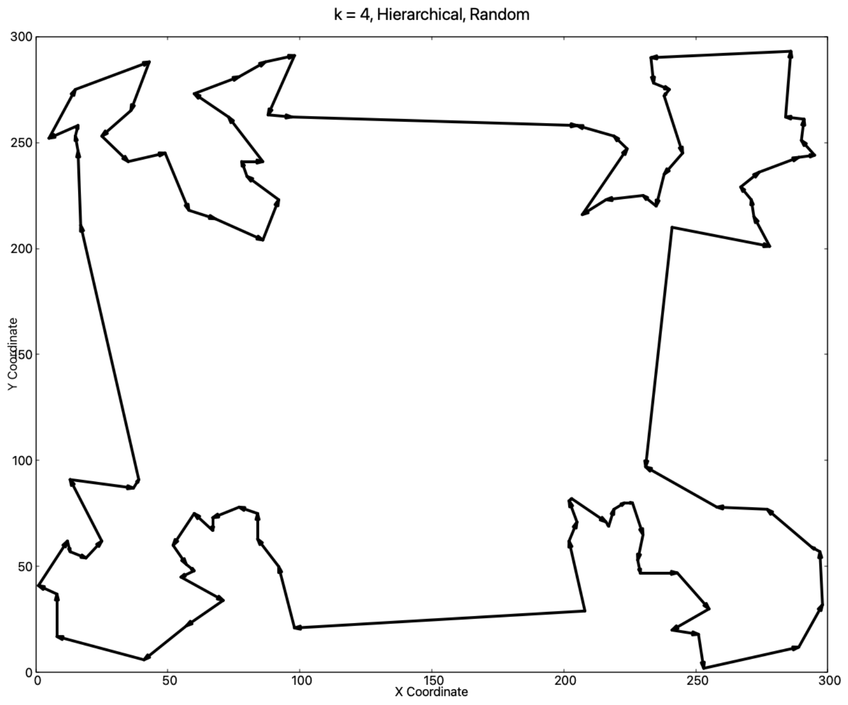
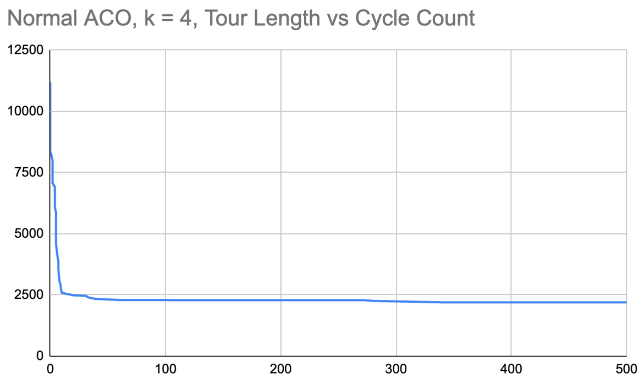


Figure 21 and 22: k=5 hierarchical



Clustered ACO - Hierarchical Approach, k = 4		
Shortest Tour Length	1975	**-223, 10.14% shorter**
Cycle Count	50 at hypergraph level 500 for each local cluster	4(500*25) + (50 * 100)
Elapsed Time	14.90 seconds	**-66.12 seconds, 5.4x faster**

5.3 Experiment 3: Hierarchical vs. Normal ACO, k = 6

Graph has random points divided into 6 clusters, with 20 nodes in each cluster.

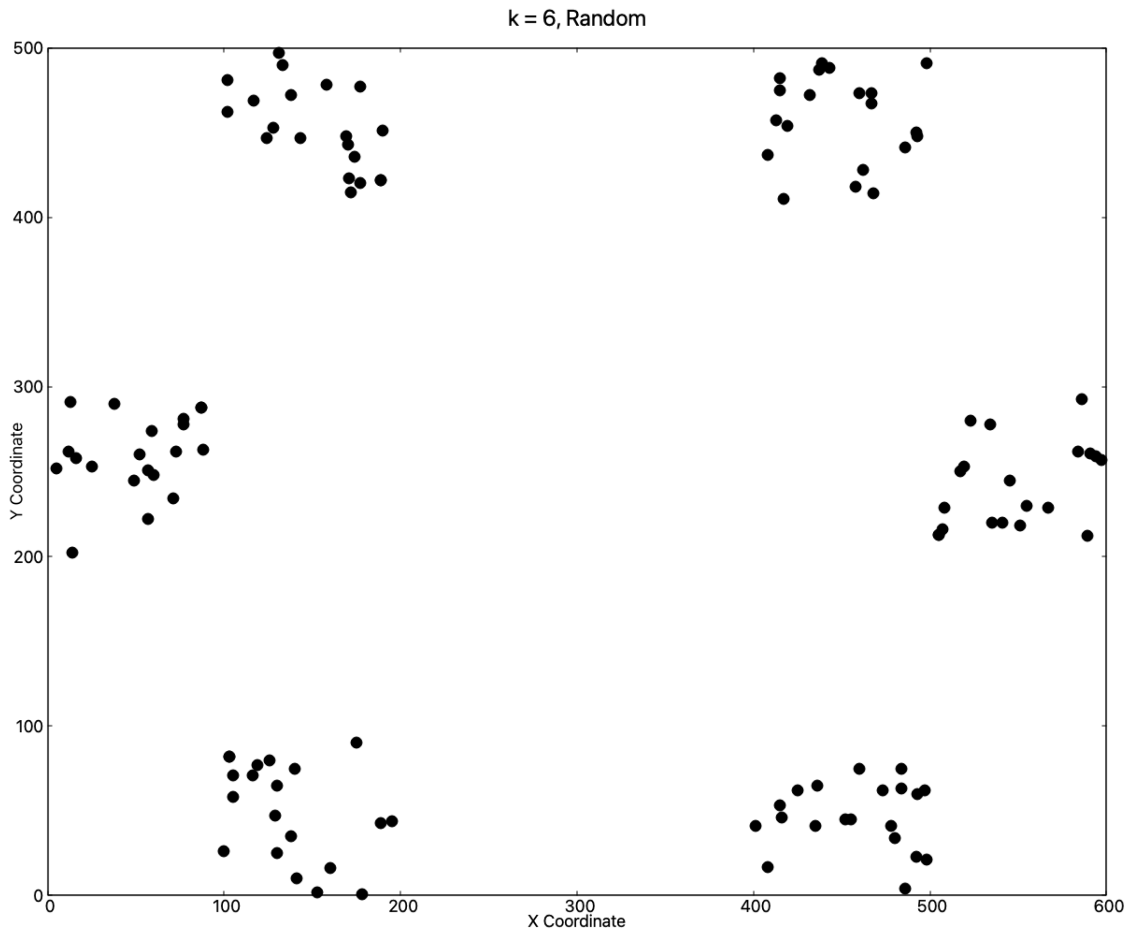


Figure 23: generated graph, k=6

Normal ACO Results

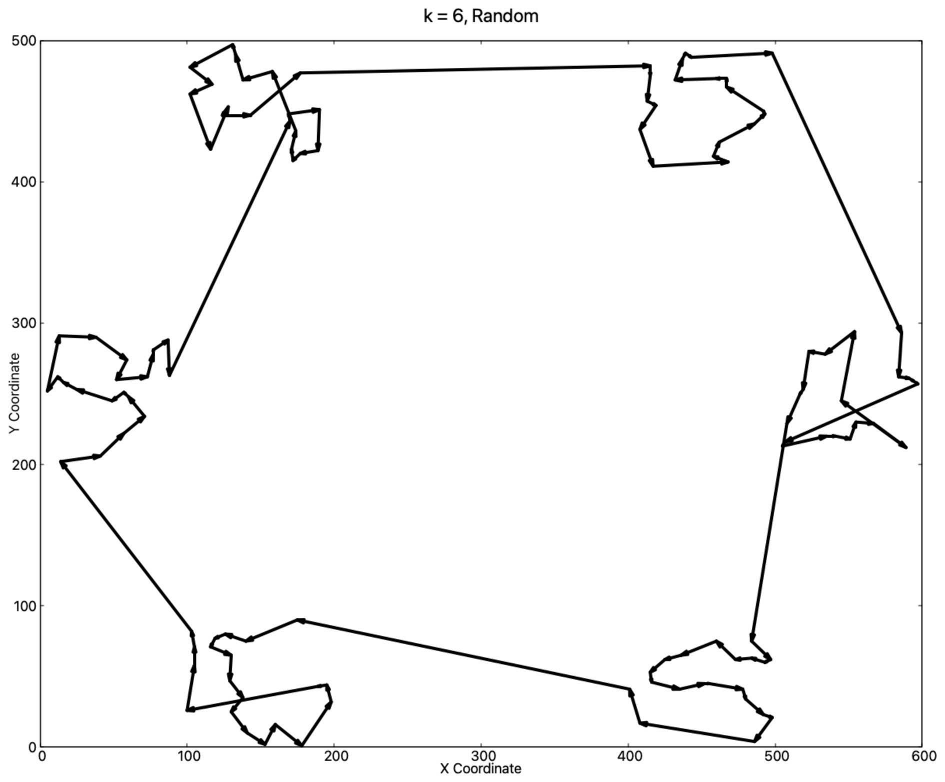
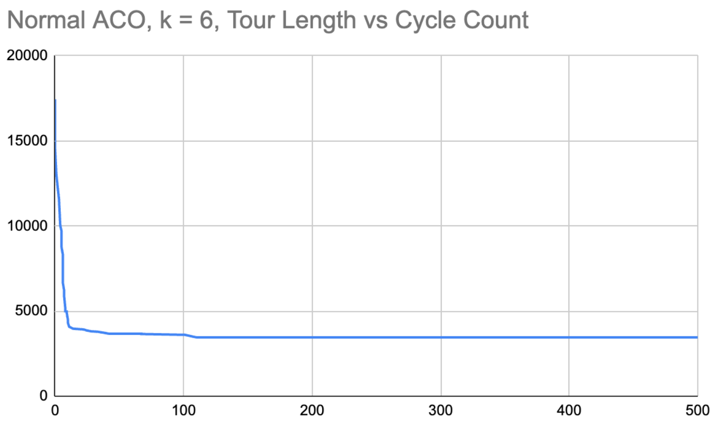


Figure 24 and 25: normal aco tour results, k=6



Normal ACO	
Shortest Tour Length	3456
Cycle Count	500 cycles (500 * 120 nodes)
Elapsed Time	143.11 seconds

Hierarchical ACO Results

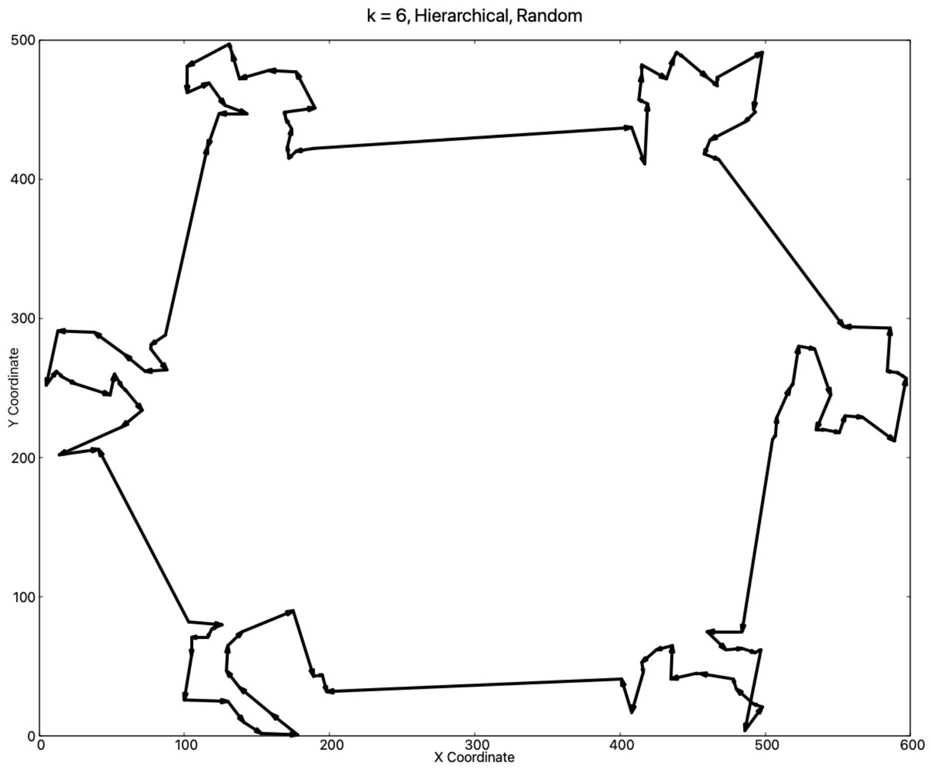
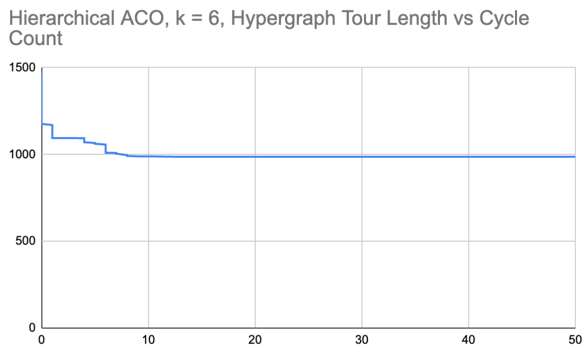


Figure 26 and 27: hierarchical tour results, k=6



Clustered ACO - Hierarchical Approach, k = 6		
Shortest Tour Length	3025	**-431, 12.47% shorter**
Cycle Count	50 at hypergraph level 500 for each local cluster	6(500*20) + (50 * 120)
Elapsed Time	17.29 seconds	**-125.82 seconds, 8.2x faster**

5.4 Experiment 4: Hierarchical vs. Normal ACO, k = 16

Graph has random points divided into 16 clusters, with 20 nodes per cluster

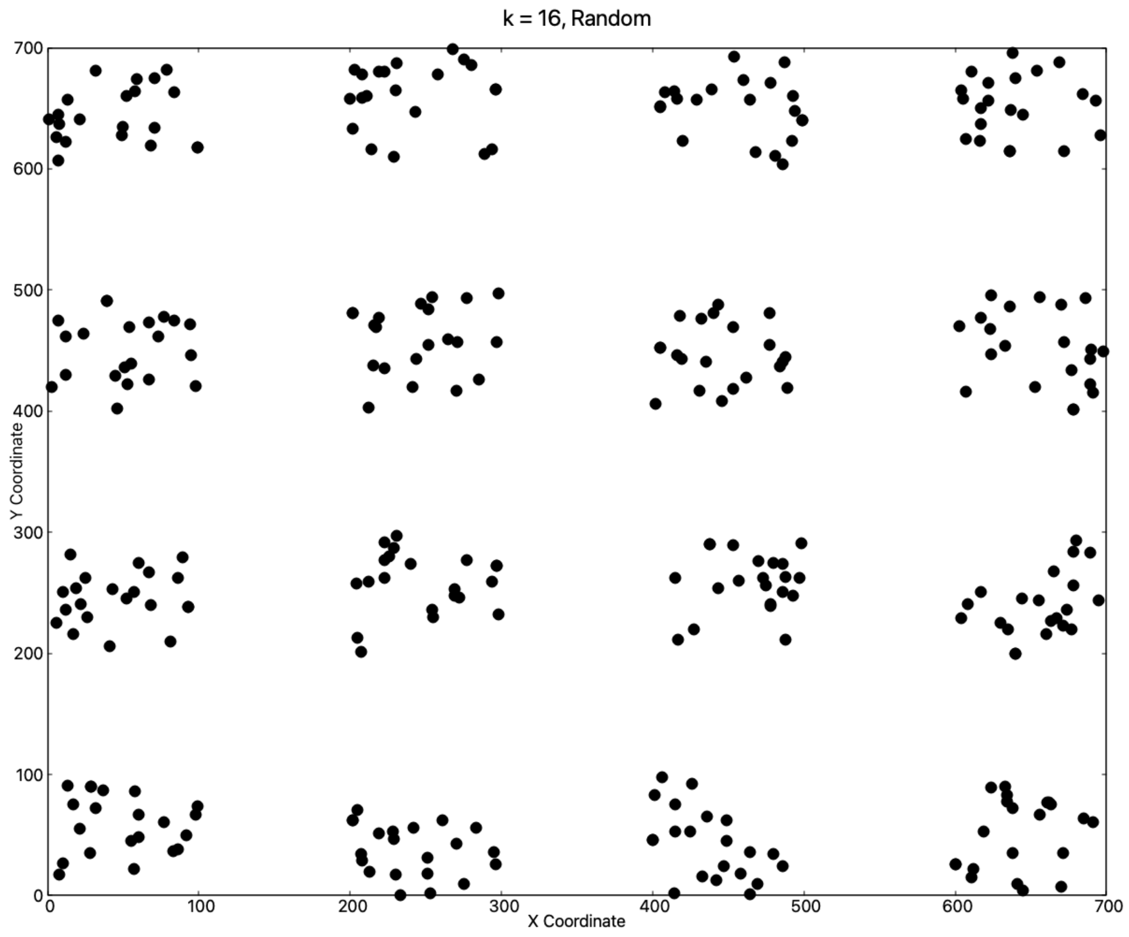


Figure 28: generated graph, k=16

Normal ACO Results

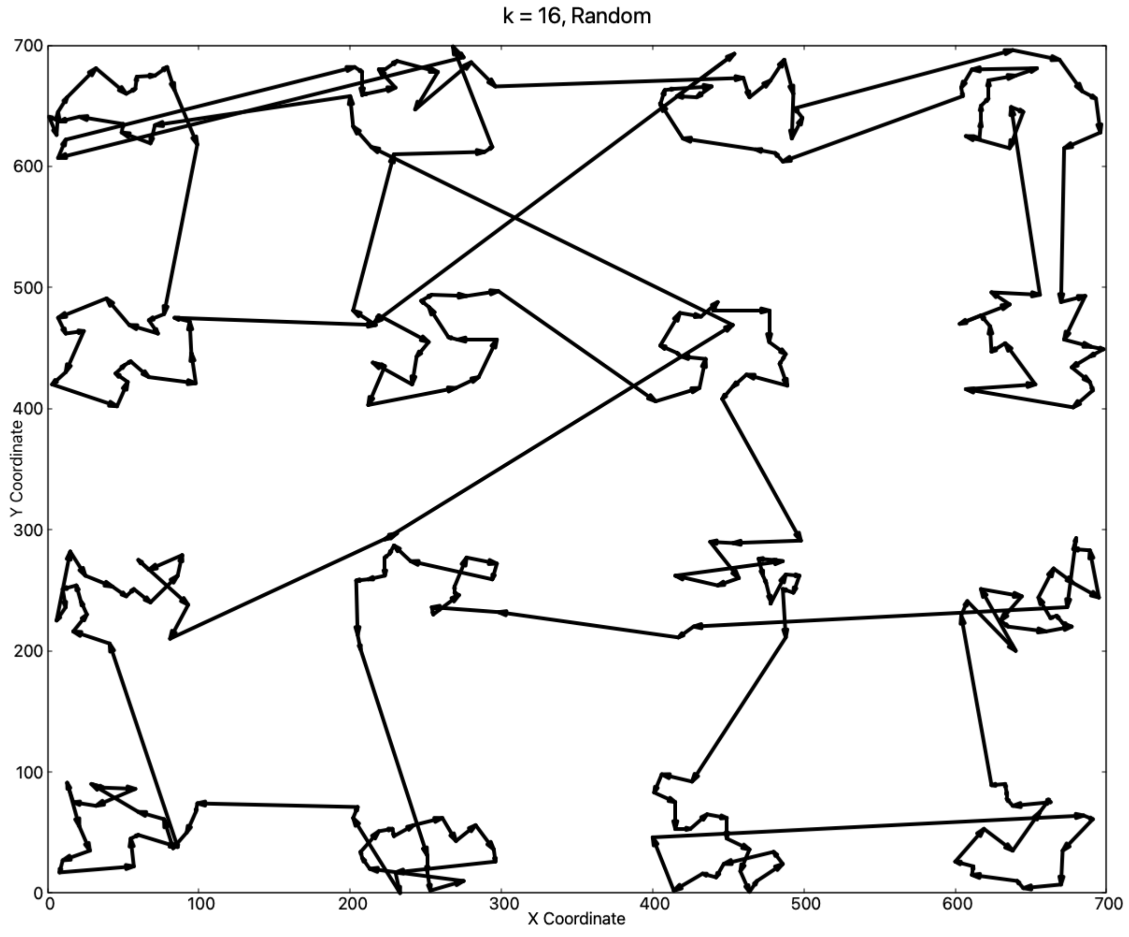


Figure 29: normal aco tour results, k=16

Normal ACO	
Shortest Tour Length	10034
Cycle Count	500 cycles (500 * 320 nodes)
Elapsed Time	4106.62 seconds

Hierarchical ACO Results

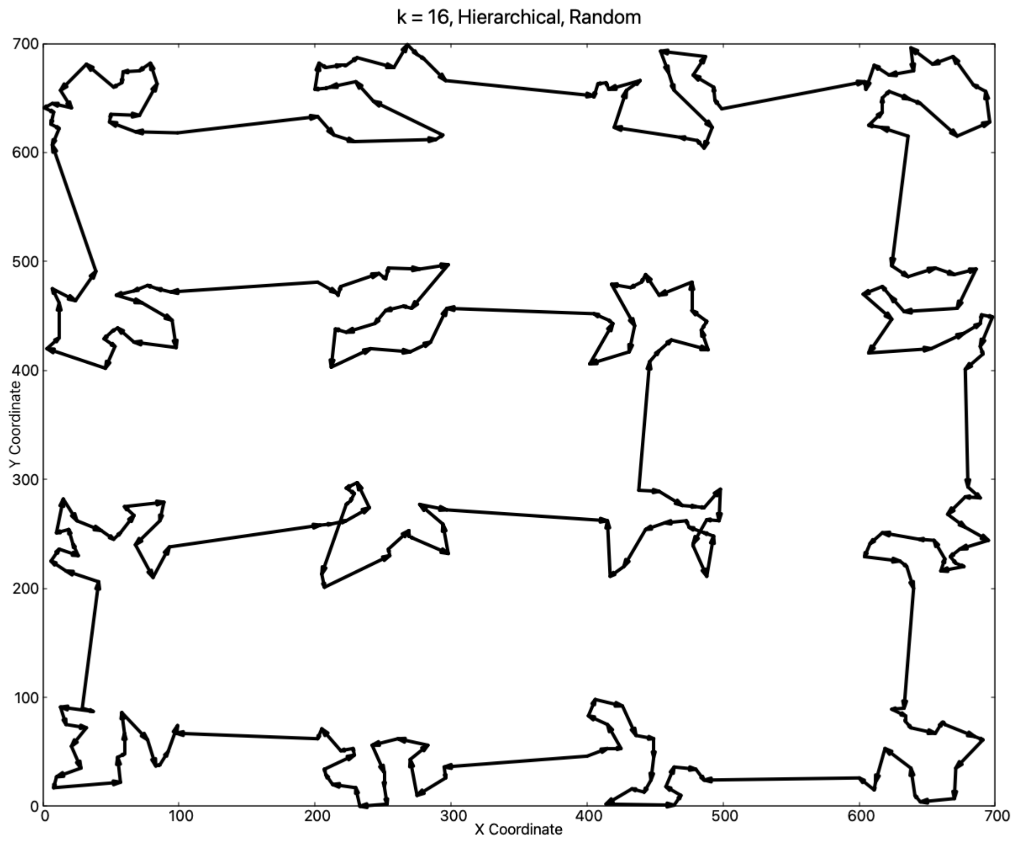
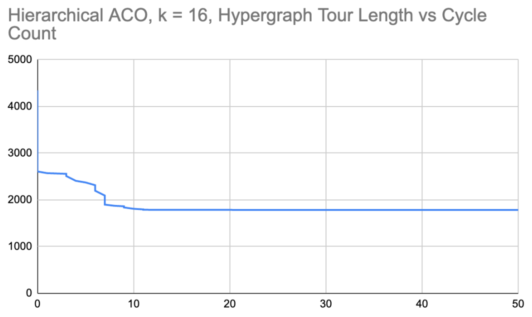


Figure 30: hierarchical tour results, k=16



Clustered ACO - Hierarchical Approach, k = 16		
Shortest Tour Length	7432	**-2602, 25.93% shorter**
Cycle Count	50 at hypergraph level 500 for each local cluster	16(500*20) + (50 * 320)
Elapsed Time	100.84 seconds	**-4005.78 seconds, 40.7x faster**

5.5 Experiment 5: Hierarchical vs. Normal ACO, eil76, k = 4

The eil76 dataset is taken from TSPLib and is often used in ACO papers as a means to benchmark their solutions. Represented in this experiment is the traditional ACO run against the hierarchical ACO with k = 4 clustering. Clusters were created in this dataset by RapidMiner using k-means clustering.

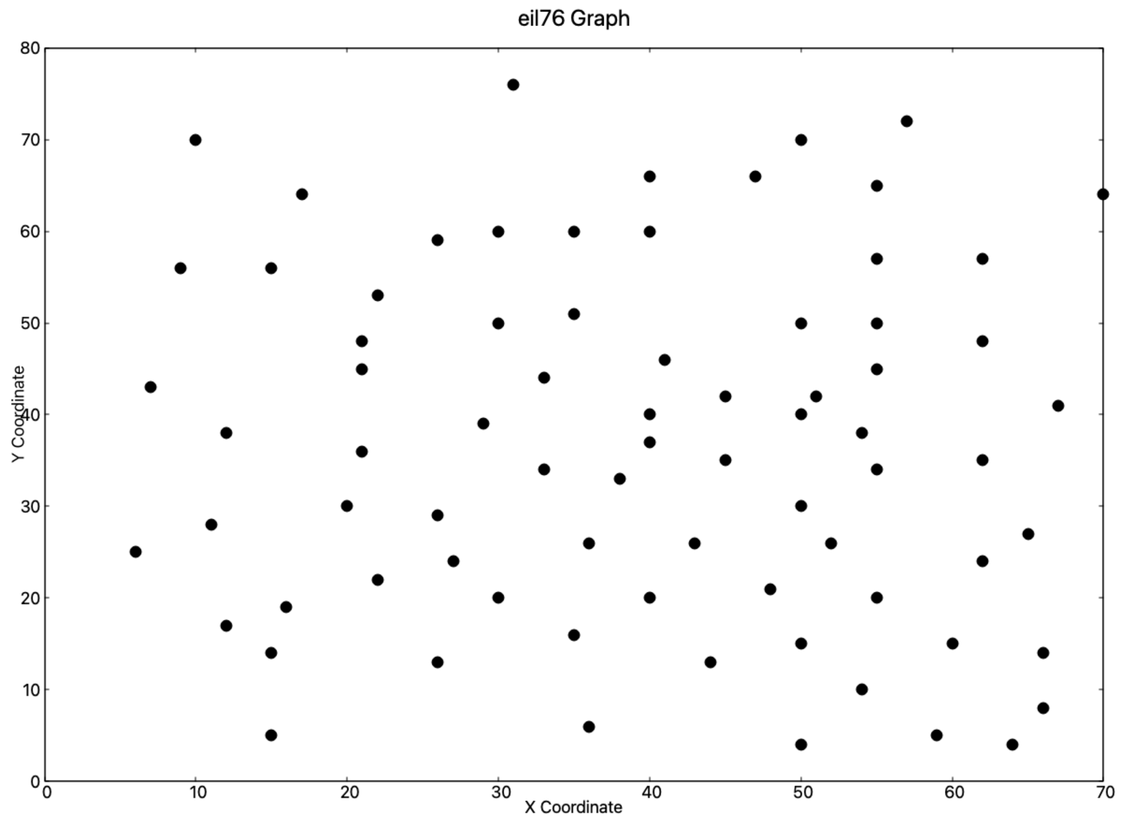


Figure 31: eil76 graph

Normal ACO eil76 Results

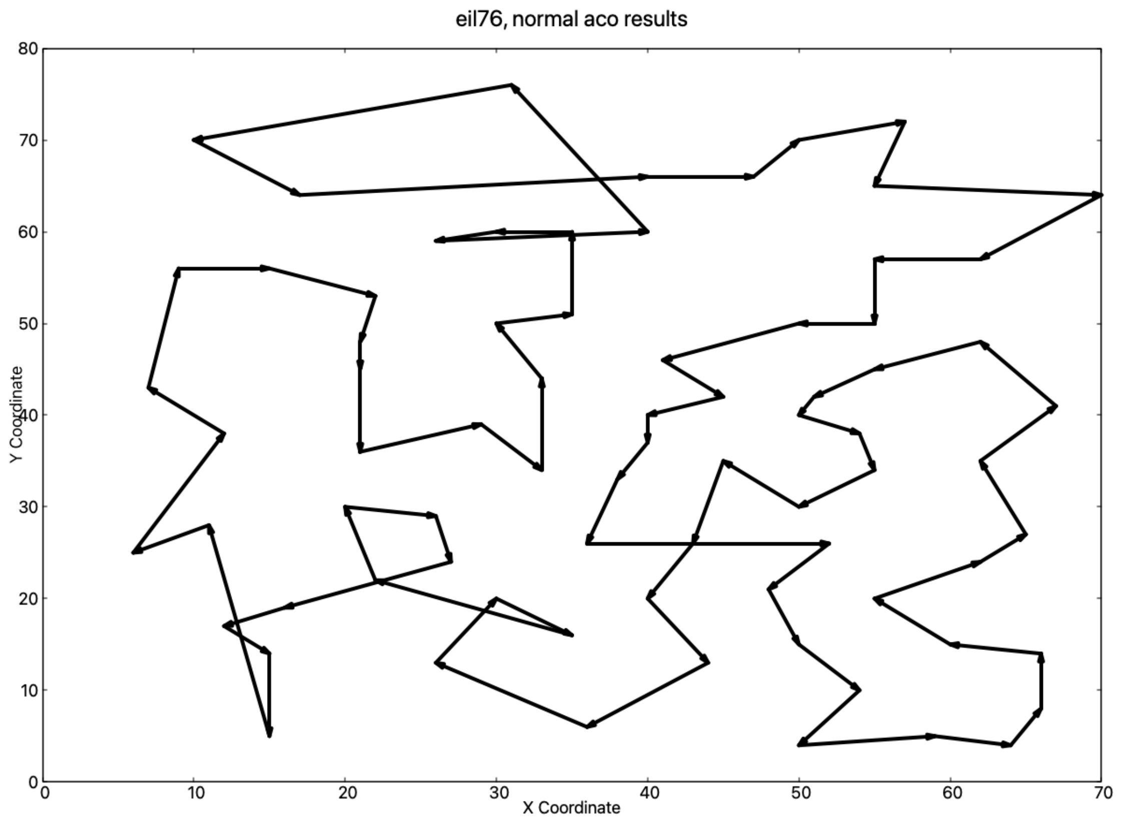
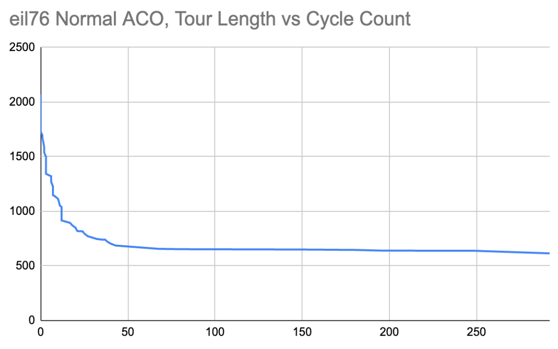


Figure 32 and 33: eil76 normal aco tour results



Shortest Tour Length	615.157
Cycle Count	500
Elapsed Time	31.910 seconds

Hierarchical ACO eil76, k = 4 Results

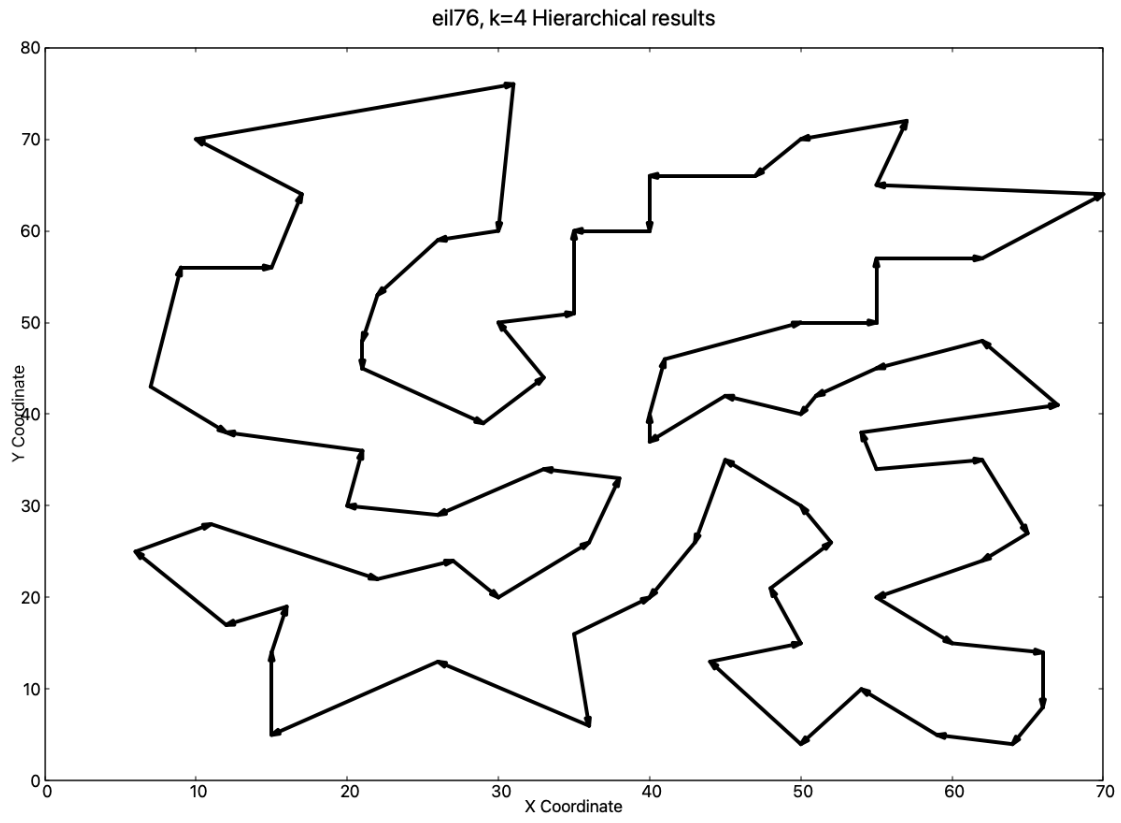
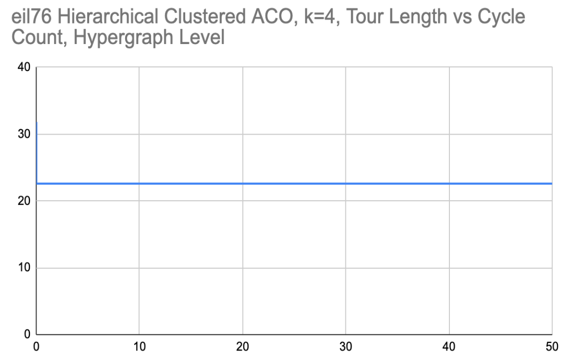


Figure 34 and 35: eil76, hierarchical aco results



Shortest Tour Length	571.432	**-43.725, 7.1% improvement**
Cycle Count	50 on hypergraph 500 on each cluster	50(76) + 4(500 * ~19)
Elapsed Time	4.8 seconds	**-27.11 seconds, 6.6x faster**

Figure 34: cluster data from eil76 k = 4

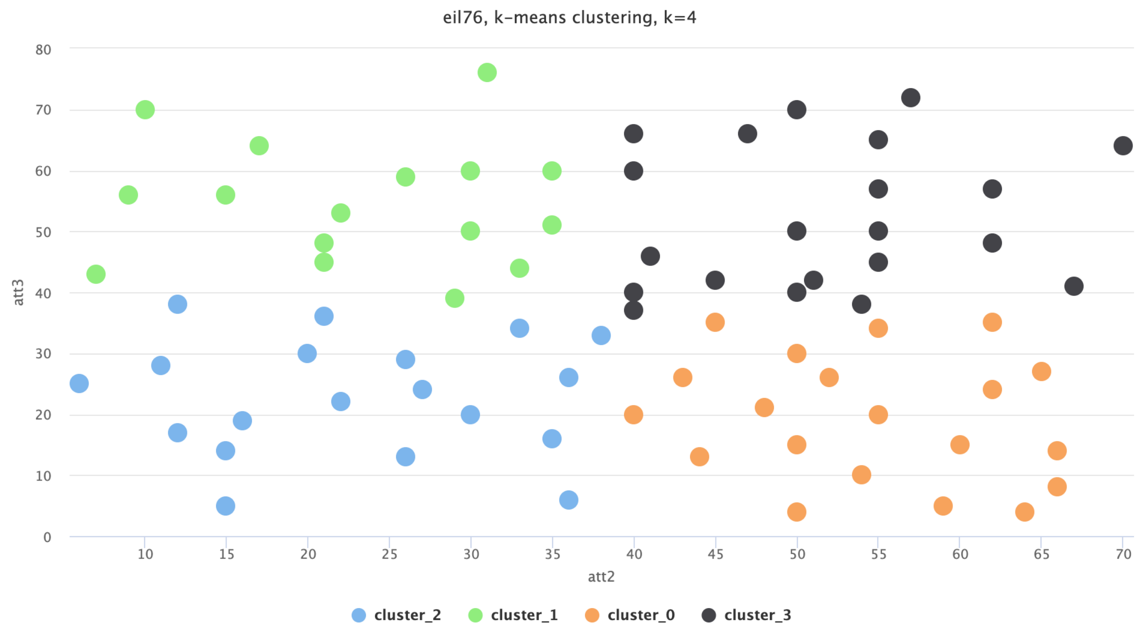
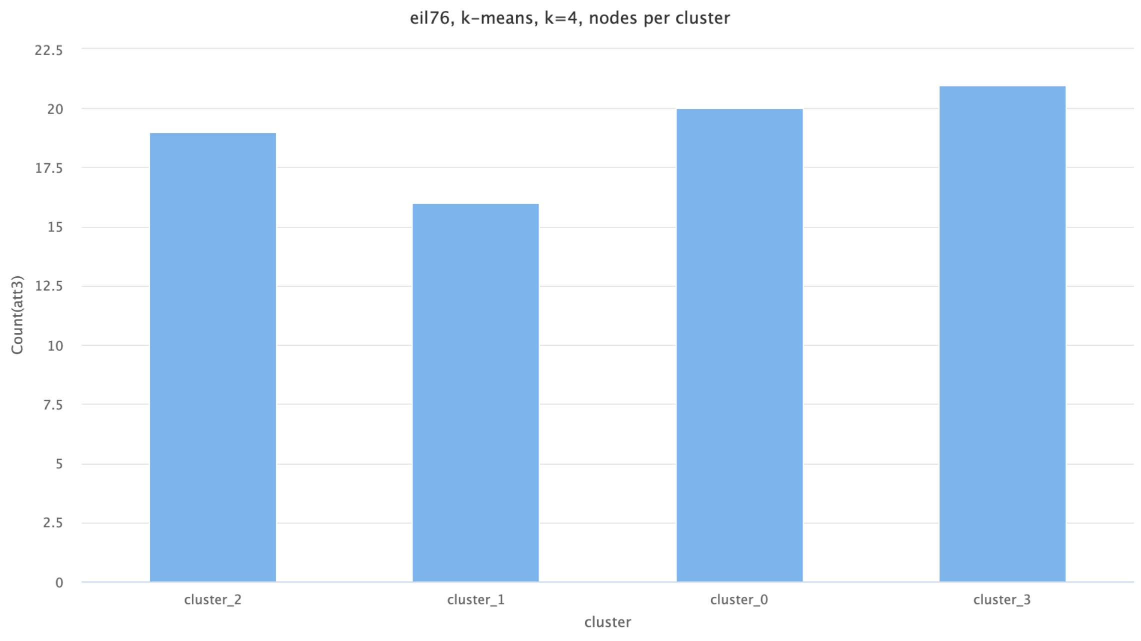


Figure 35: nodes per cluster



5.6 Experiment 6: Hierarchical vs. Normal ACO, att532, k = 4, k = 16

The att532 dataset from TSPLib is another commonly used benchmarking dataset for TSP solvers [11]. It contains 532 data points and since we have a larger number of cities to traverse, the following tests will show k = 4 as well as k = 16 hierarchical results.

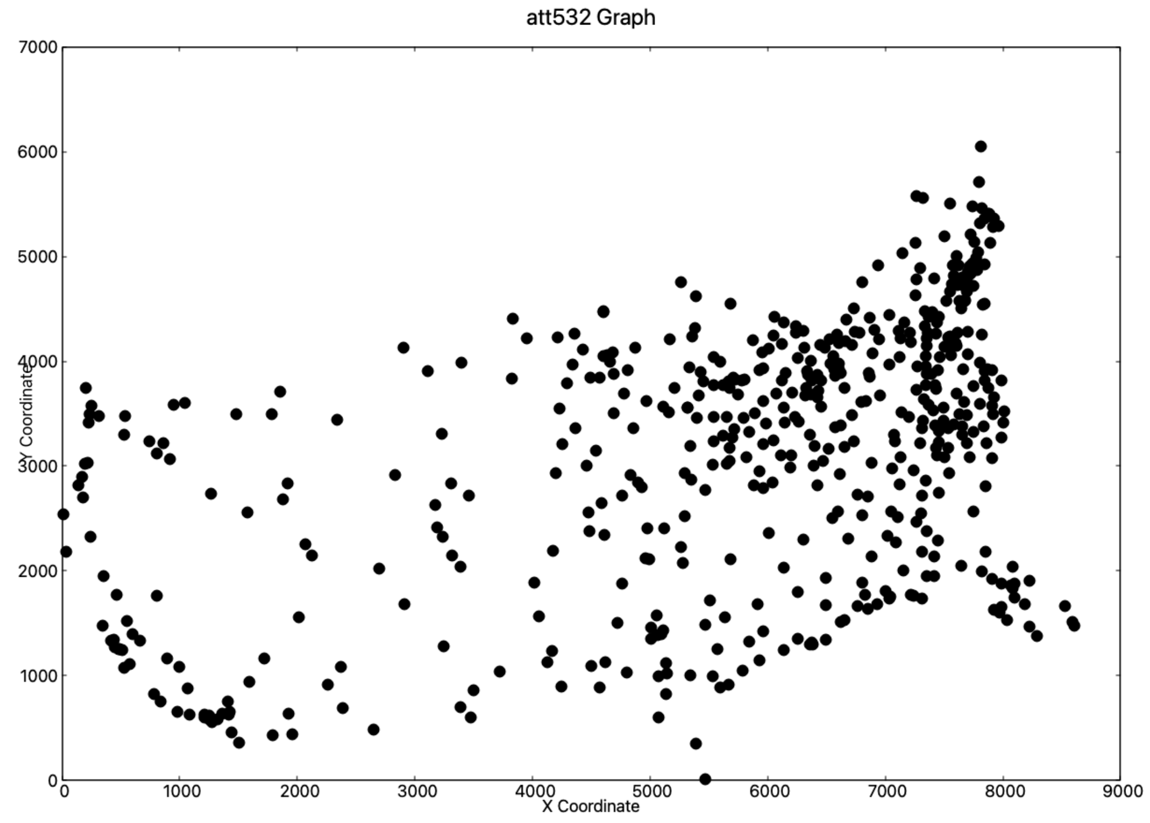


Figure 36: att532 graph

Normal ACO on att532

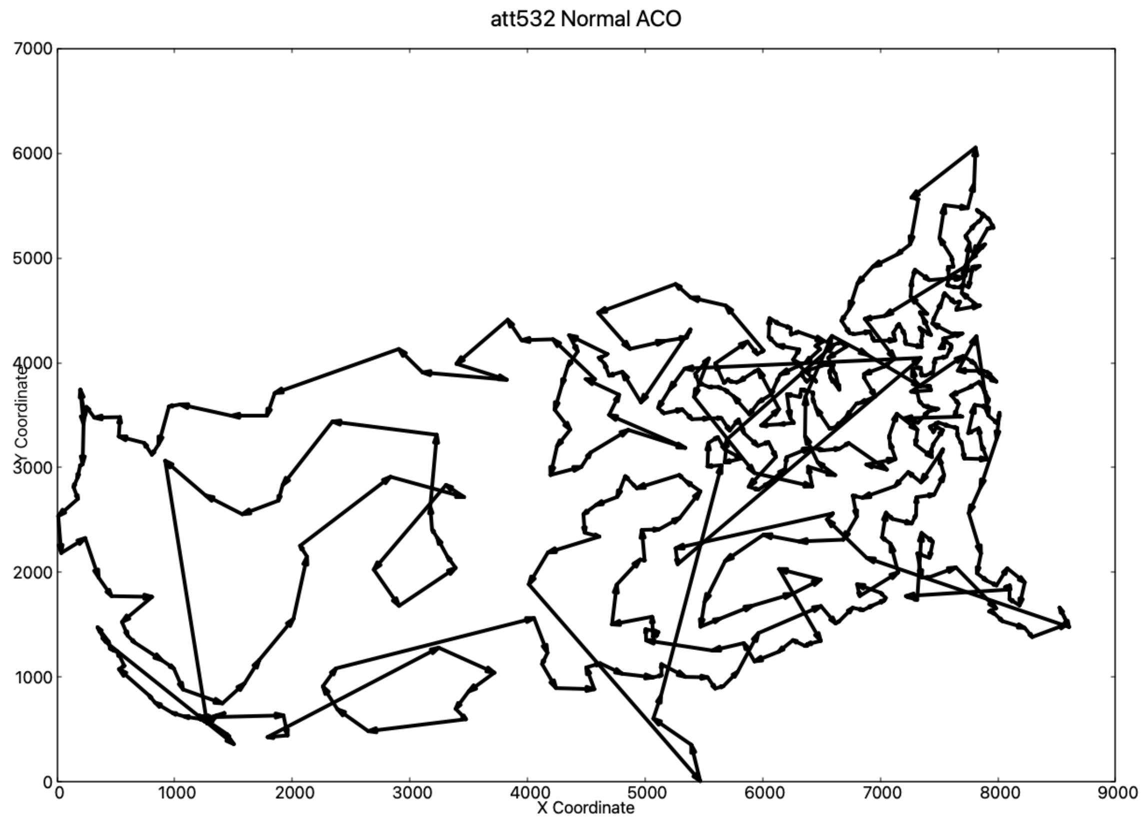
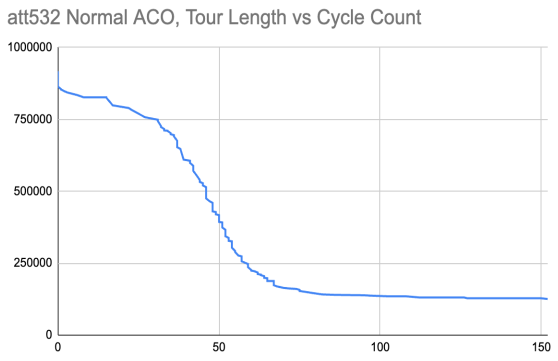


Figure 37 and 38: att532 normal aco tour results



Shortest Tour Length	125574.626
Cycle Count	500
Elapsed Time	6 hours 36 minutes 38 seconds

Hierarchical ACO on att532

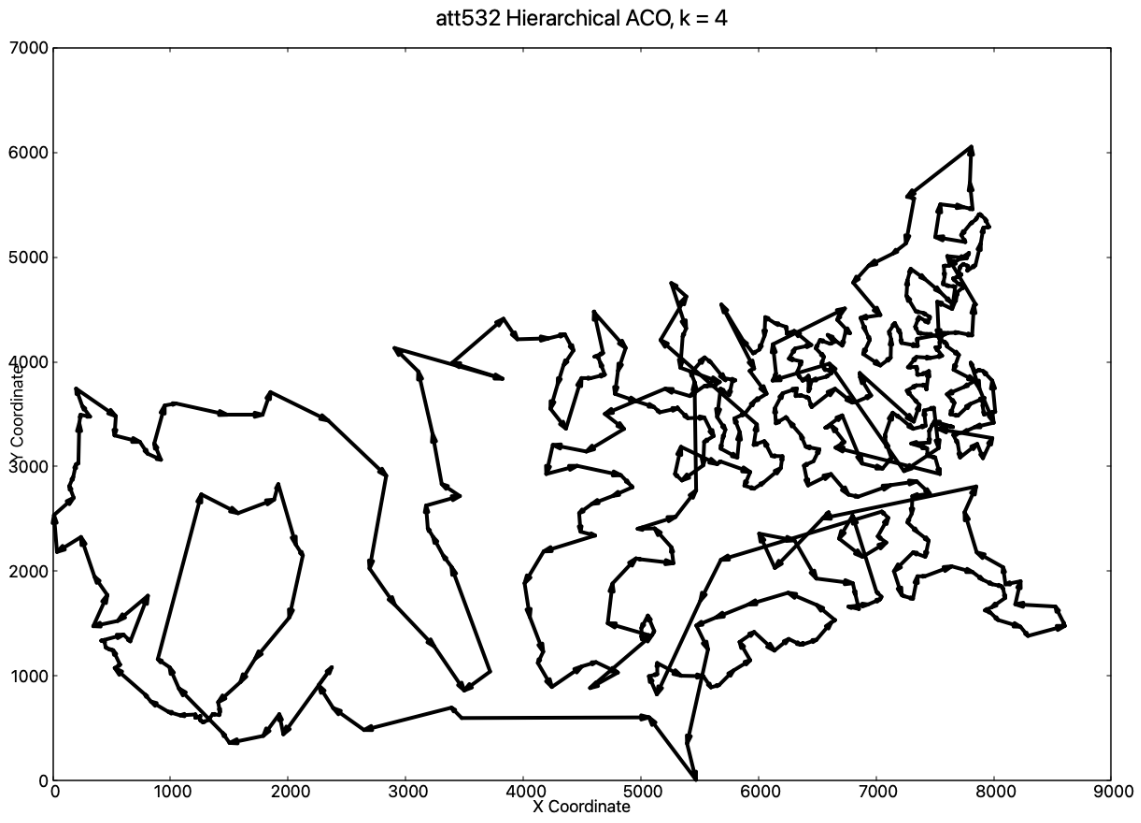
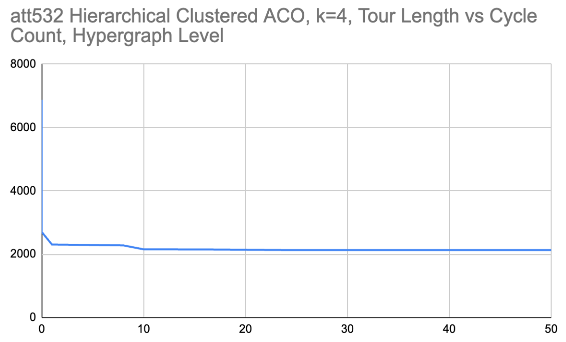


Figure 39 and 40: hierarchical ACO on att532, k = 4, k-means clustering



Shortest Tour Length	108482.3922	**-17091.60, 13.62% improvement**
Cycle Count	50 at hypergraph level 500 on each cluster	(50 * 532) + 4(500 * ~133)
Elapsed Time	32 minutes 46.67 seconds	**-6.06 hours, 12x faster**

Figure 41: cluster data att532, k=4

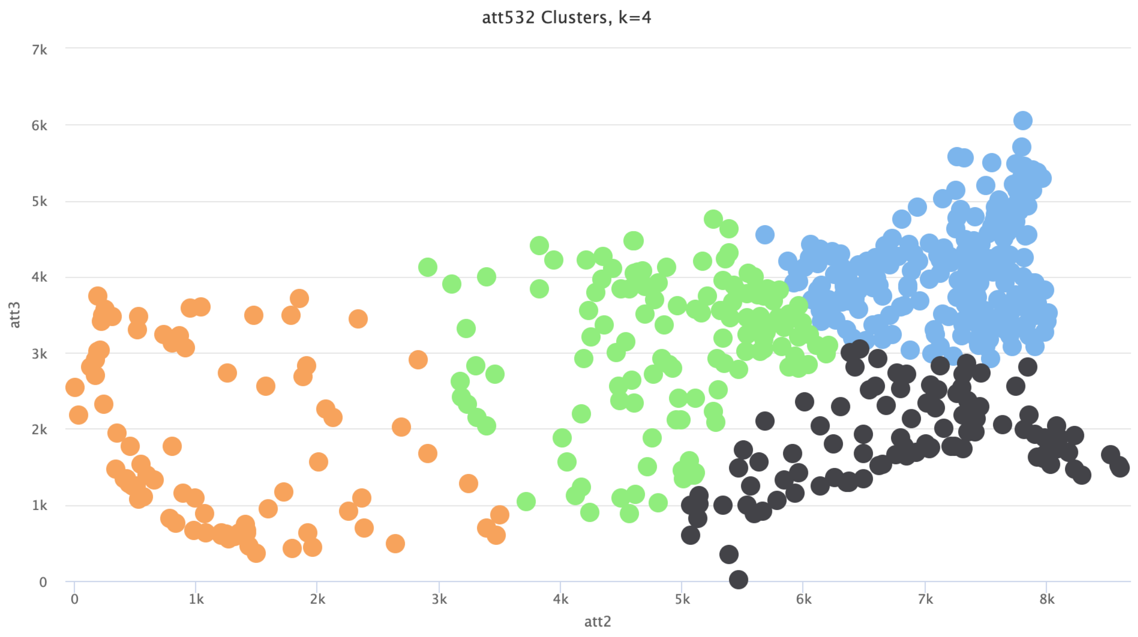
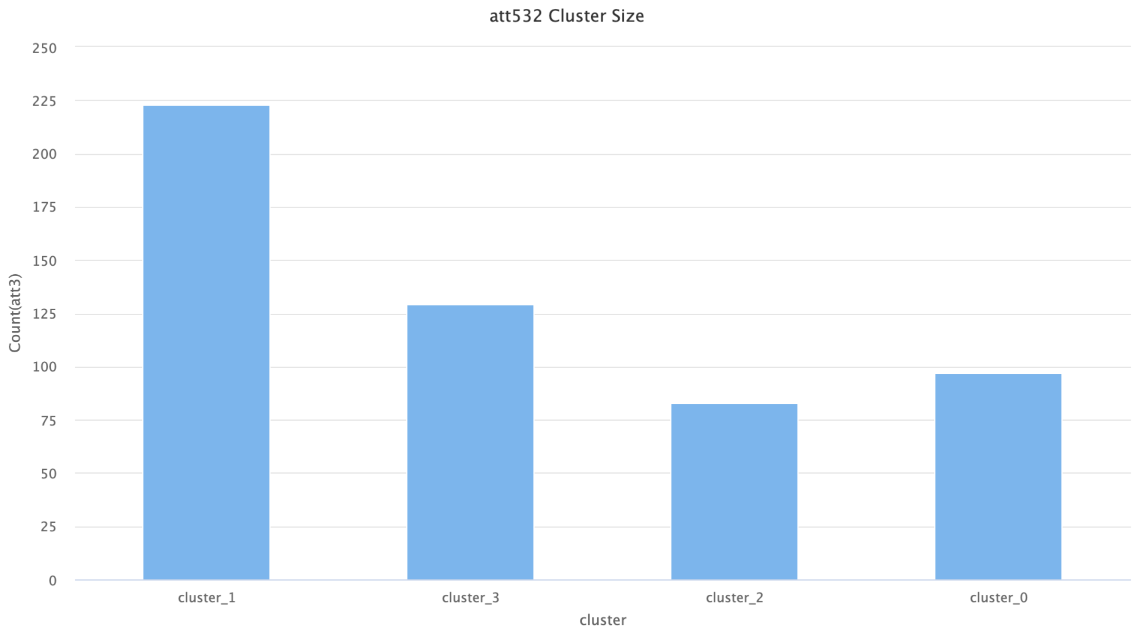


Figure 42: notes per cluster att532, k=4



Hierarchical ACO on att532, k = 16

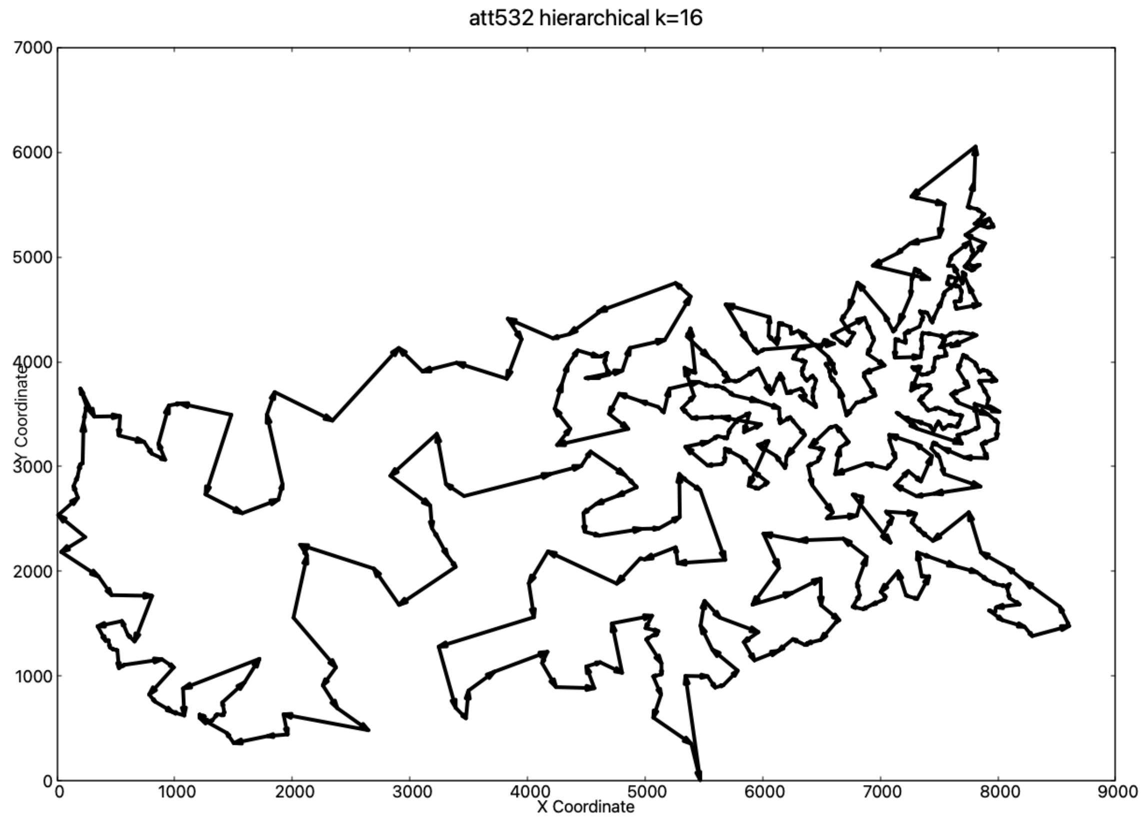
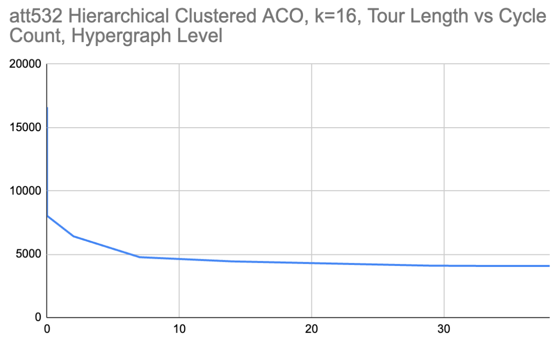


Figure 43 and 44: hierarchical ACO on att532, k = 16, k-means clustering



Shortest Tour Length	96952.9354	**-28621.69, 22.79% improvement**
Cycle Count	50 on hypergraph level 500 on each cluster	(50 * 532) + 16(500 * ~33)
Elapsed Time	3 minutes 56 seconds	**-6.545 hours, 100x faster**

Figure 45: cluster data, att532, k=16

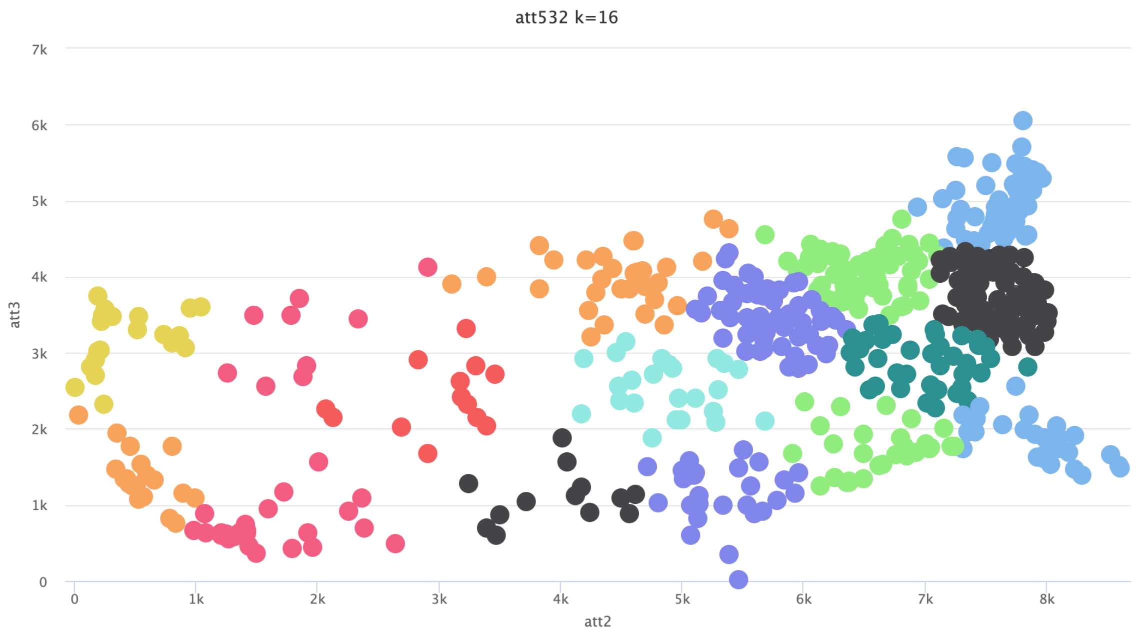
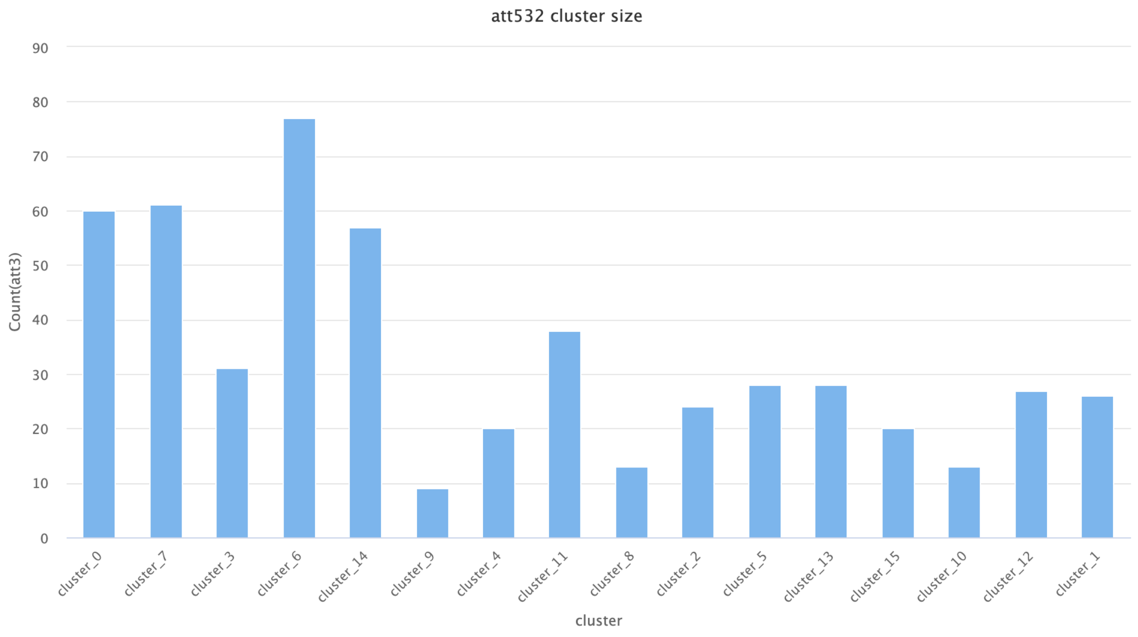


Figure 46: nodes per cluster, att532, k=16



6 Discussion and Conclusion

It is clear that reducing the node count will improve the ACO performance, and the experiments show that we gain dramatic improvements in the time cost of the ACO by utilizing a top-down hierarchical approach. We can see that for smaller graphs the improvements are marginal, but as the graphs get larger we see as high as 99% improvement in time cost while returning a shorter tour length. This is largely due to the fact that O(n^3) algorithms scale so drastically, so when we can chop up the work into smaller portions we end up with better results. If we have a data set of size n, our algorithm running the traditional ACO heuristic will take n^3 time to complete. However, when we use the hierarchical approach and cluster the data into 4 groups, for example, we can see that $n^3 > \left(\frac{n}{4}\right)^3 + \left(\frac{n}{4}\right)^3 + \left(\frac{n}{4}\right)^3 + \left(\frac{n}{4}\right)^3$. We also must add in the overhead cost of processing the graph at the hypergraph level, however this is a function of the number of clusters we have. Since the hypergraph treats each cluster as an individual node, the overall size of the hypergraph will be relatively small. In our experiments, the largest k value used was 16 for a data set of size 532. It is very expensive to process the hypergraph, but since even very large graphs have modest k values, this cost increase is negligible to the overall time savings.

In experiment 1, it is shown how the ACO stagnates relatively early, within 100 to 200 cycles in most cases, at which time no new shortest tours are found. Every experiment was run with a set number of cycles to make comparisons more equitable, however it's worth noting that additional cost savings could be found when terminating the runs earlier due to stagnation [7].

In the later experiments we see the most improvement. When the original graph size is large and the k value is 16, we can achieve a 99% improvement in time cost over the original ACO implementation. The highest potential gains are seen when the graph is very large and each cluster has roughly the same number of nodes. In the att532 test with k=4, notice there is an uneven distribution to the clusters where more nodes resided in the east coast clusters than in the less-dense west coast. This affects the runtime of the algorithm such that it takes longer to process cycles for clusters with higher node counts. Due to the n^3 scaling of the algorithm, if one cluster is twice the size of another cluster, it more than doubles the time cost to process that cluster. It is very important to have equal sized clusters for optimal performance using this approach.

Another observation when looking at the graphs visually, is that the hierarchical approach produces a much cleaner path with less lines crossing each other. Whenever you have two paths that cross one another, there is room for improvement as you can uncross their paths and recombine the edges to improve the tour length [2]. Also notice, that the original ACO does a good job of staying within a local group before leaving, but due to the randomness of the ant movements it will inevitably hop around the entire graph producing long traversals that should not exist in an optimal solution. By clustering the nodes first, and establishing the edges that link clusters together in the beginning at the hypergraph level, we prevent ants from ever making long traversals to opposite sides of the graph and instead force them to stay within their local clusters producing much better results.

Finally, it's worth pointing out that our experiments only used two levels: the hypergraph level and the local cluster level. For a sufficiently large dataset, where the value of k could be very large, we could recursively apply this strategy again and again producing multiple levels in the hierarchy. Then working top to bottom, execute the algorithm to come up with a solution. Graphs of this size would be impractical to run using any n^3 algorithm, but with the improvements seen in this thesis project there is a possibility it could be quick to solve.

7 Future Work

There are many ways to improve the ACO algorithm. Due to how similar this algorithm is to the original ACO in some regards, it would be possible to use other improvements to augment this work. For example, implementing a mechanism to detect path stagnation would significantly speed up the processing time. Also, taking a deeper look into the types of clustering algorithms to ensure a more uniform distribution would be another way to improve the performance. There is also a question of an optimal value for k given specific data sets. Conducting further studies on varying values of k for various types of data might reveal an optimal number that could further help this ACO implementation run faster. Finally, applying the hierarchical strategy to n layers instead of just two would be very exciting to see how well it performs on massive datasets. In the future it would be worth exploring any of these avenues of improvement with further tests and experiments.

References

[1] P. Black. "hypergraph." Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 26 August 2014. https://www.nist.gov/dads/HTML/hypergraph.html. Accessed June 13, 2023.

[2] A. Blazinskas, A. Misevicius. "Combining 2-OPT, 3-OPT and 4-OPT with K-SWAP-KICK perturbations for the traveling salesman problem." 2011, 17th International Conference on Information and Software Technologies.

[3] Y. C. Chang. "Using k-means clustering to improve the efficiency of ant colony optimization for the traveling salesman problem." 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Banff, AB, Canada, 2017, pp. 379-384, doi: 10.1109/SMC.2017.8122633.

[4] M. Dorigo, C. Blum. "Ant colony optimization theory: A survey." Theoretical Computer Science, Volume 344, Issues 2–3, 2005, Pages 243-278. ISSN 0304-3975, https://doi.org/10.1016/j.tcs.2005.05.020.

[5] M. Dorigo, V. Maniezzo and A. Colorni. "Ant system: optimization by a colony of cooperating agents." IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 26, no. 1, pp. 29-41, Feb. 1996, doi: 10.1109/3477.484436.

[6] M. Dorigo, T. Stützle. "ACO Algorithms for the Traveling Salesman Problem." 1999. *Evolutionary algorithms in engineering and computer science 4*: 163-183.

[7] M. Dorigo and T. Stützle, Thomas and Darmstadt, Tu and Group, Intellectics. "The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances." 2001, doi: 10.1007/0-306-48056-5_9

[8] K. Jun-man and Y. Zhang. “Application of an Improved Ant Colony Optimization on Generalized Traveling Salesman Problem.” Energy Procedia, Volume 17, Part A, 2012, Pages 319-325, ISSN 1876-6102, https://doi.org/10.1016/j.egypro.2012.02.101.

[9] A. Neoh, M. Wilder-Smith, H. Chen, C. Chase. “An Evaluation of the Traveling Salesman Problem.” 2020. California State Polytechnic University, Pomona. https://scholarworks.calstate.edu/concern/theses/8g84mp499.

[10] Oracle. *Random Class*. Docs.Oracle.com, (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/Random.html. Accessed June 10, 2023.

[11] A. Saleh, B. Shahadat, M. A. H. Akhand and Md A. S. Kamal. "Visibility Adaptation in Ant Colony Optimization for Solving Traveling Salesman Problem." 2022. Mathematics, MDPI, vol. 10(14), pages 1-24, July.

[12] T. Stützle and H. Hoos. "MAX-MIN Ant System and local search for the traveling salesman problem." Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97), Indianapolis, IN, USA, 1997, pp. 309-314, doi: 10.1109/ICEC.1997.592327.

[13] Universität Heidelberg Institut für Informatik. *Symmetric Traveling Salesman Problem (TSP)*. TSPLIB. comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html. Accessed May 27, 2023.

Vita

Author:

Bryan J. Fischer

Place of Birth:

Spokane, Washington

Undergraduate Schools Attended:

Spokane Falls Community College

Gonzaga University

Eastern Washington University

Degrees Awarded:

Associate of Arts, 2018, Spokane Falls Community College

Bachelor of Arts, 2020, Gonzaga University

Honors and Awards:

Graduate Service Appointment, Computer Science Department, 2021-2022, Eastern Washington University.

Graduated Magna Cum Laude, Gonzaga University, 2020.

Recipient of Gonzaga’s Academic Excellence Scholarship, 2018.

Gonzaga Magis Award, 2018-2019, Math Tutoring.

Professional Experience:

Full-Time Lecturer of Computer Science, Gonzaga University, 2022-2023.

Computer Science Instructor CS1, Fall/Winter/Spring 2021-2022, Eastern Washington University.

Spokane Falls Community College Math Tutor and Supplemental Instruction, Math Department, 2016-2020.