Application of metaheuristics to a subset selection graph problem

Michael Volz

Follow this and additional works at: https://louis.uah.edu/uah-theses

Recommended Citation

This Thesis is brought to you for free and open access by the UAH Electronic Theses and Dissertations at LOUIS. It has been accepted for inclusion in Theses by an authorized administrator of LOUIS.
APPLICATION OF METAHEURISTICS TO A SUBSET SELECTION GRAPH PROBLEM

by

MICHAEL VOLZ

A THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in
The Department of Computer Science to
The School of Graduate Studies of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2020
In presenting this thesis in partial fulfillment of the requirements for a master’s degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Michael Volz

10/25/2020

(date)
THESIS APPROVAL FORM

Submitted by Michael Volz in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in Computer Science.

Huaming Zhang  10/25/2020  Committee Chair
Dr. Huaming Zhang (Date)

Dr. Grant Zhang  10/26/2020

Dr. Joshua Booth  10/25/2020  Department Chair
Date: 2020.10.28  09:12:57 -05'00'
Dr. Heggere Rangath (Date)

Dr. John Christy  10/30/20  College Dean

David Berkowitz  11/3/2020  Graduate Dean
Dr. David Berkowitz (Date)
ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree Masters of Science College/Dept. Science/
in Computer Science Computer Science
Name of Candidate Michael Volz
Title Application of Metaheuristics
to a Subset Selection Graph Problem

A graph problem is presented in which the objective is to remove links from a graph while maintaining path length and redundancy requirements. A branch-and-bound solution is presented, which is able to deterministically solve the problem for smaller inputs but fails to scale with graph size. An ant colony optimization approach is then presented, which is able to solve the problem stochastically with high probability and low runtime.

Abstract Approval: Committee Chair

Huaming Zhang
Dr. Huaming Zhang

Date: 2020.10.28 09:13:26 -05'00'

Department Chair

H.S. Ranganath
Dr. Heggere Ranganath

Graduate Dean

David Berkowitz 11/3/2020
Dr. David Berkowitz
ACKNOWLEDGMENTS

I would like to thank Neil Sutphin for proposing this project to me and providing advice and feedback along the way. I would also like to thank Dr. Huaming Zhang for his advice on creating this thesis and serving on my thesis committee along with Dr. Grant Zhang and Dr. Joshua Booth.
# TABLE OF CONTENTS

List of Figures ix

List of Tables xi

Chapter

1 **Introduction** 1

1.1 Problem Overview ............................................. 1

1.2 Complexity Analysis .......................................... 3

1.3 Comparison to Similar Problems ............................. 3

1.4 Solution Methods .............................................. 4

2 **Network Requirements** 5

2.1 Overview .......................................................... 5

2.2 Depth-First Search (DFS) .................................... 5

2.3 Chain Decomposition .......................................... 7

2.4 Breadth-First Search (BFS) .................................. 8

2.5 Detour Algorithm .............................................. 10

3 **Branch and Bound** 15

3.1 Introduction .................................................... 15

3.2 Generic Algorithm ............................................ 15
3.3 Our Algorithm .................................................. 17
  3.3.1 Unprunables .................................................. 18
  3.3.2 Minimum Node Degrees ................................. 20
  3.3.3 Check if Link is Prunable ............................... 22
  3.3.4 Check if Potential Solution Exists .................. 23
  3.3.5 Sorting the Links List .................................... 25
  3.3.6 Finding an Initial Solution ............................ 27
  3.3.7 Final algorithm ............................................ 28

4 Ant Colony Optimization ....................................... 31
  4.1 Introduction ................................................ 31
  4.2 Network Solution in Ant Form ............................ 32
  4.3 Generic Algorithm .......................................... 33
  4.4 Algorithm Subprocedures .................................... 33
    4.4.1 Pheromone Update Frequency ......................... 34
    4.4.2 Pheromone Update Method ............................. 36
    4.4.3 Initial Pheromone .................................... 38
    4.4.4 Caching Tested Graphs ................................. 41
  4.5 Final Algorithm ............................................ 42

5 Testing and Results ............................................. 44
  5.1 Input Scenarios .............................................. 44
  5.2 Testing Environment ........................................ 46
  5.3 Branch and Bound Results .................................... 46
5.4 Ant Colony Optimization Results .............................................. 48

6 Conclusions ............................................................................. 51

6.1 Results .................................................................................. 51

6.2 Future Work .......................................................................... 51

6.3 Applications to Other Problems ............................................... 52

APPENDIX A: Scenario Laydowns ................................................. 54

A.1 Alpha .................................................................................... 54

A.2 Bravo .................................................................................... 54

A.3 Charlie ................................................................................... 54

REFERENCES .............................................................................. 61
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Network Requirements Overview.</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Depth-First Search Pseudocode.</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Chain Decomposition Process</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Breadth-First Search Pseudocode</td>
<td>10</td>
</tr>
<tr>
<td>2.5 Detour Algorithm.</td>
<td>13</td>
</tr>
<tr>
<td>2.6 Detour Paths vs. Disjoint Paths</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Generic Branch and Bound Algorithm.</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Finding Unprunable Links.</td>
<td>18</td>
</tr>
<tr>
<td>3.3 Finding Unprunable Pairs.</td>
<td>20</td>
</tr>
<tr>
<td>3.4 Calculating Minimum Node Degrees.</td>
<td>22</td>
</tr>
<tr>
<td>3.5 Check if a Link is Prunable.</td>
<td>24</td>
</tr>
<tr>
<td>3.6 Check if a Potential Solution May Exist.</td>
<td>26</td>
</tr>
<tr>
<td>3.7 Generate an Initial Solution.</td>
<td>29</td>
</tr>
<tr>
<td>3.8 Final Branch and Bound Pseudocode.</td>
<td>30</td>
</tr>
<tr>
<td>4.1 Ant Colony Optimization Initial Algorithm Overview.</td>
<td>34</td>
</tr>
<tr>
<td>4.2 Pheromone Equations.</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Ant Colony Optimization Final Version.</td>
<td>43</td>
</tr>
<tr>
<td>A.1 Alpha Unpruned.</td>
<td>55</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------</td>
</tr>
<tr>
<td>A.2</td>
<td>Alpha Pruned</td>
</tr>
<tr>
<td>A.3</td>
<td>Bravo Unpruned</td>
</tr>
<tr>
<td>A.4</td>
<td>Bravo Pruned</td>
</tr>
<tr>
<td>A.5</td>
<td>Charlie Unpruned</td>
</tr>
<tr>
<td>A.6</td>
<td>Charlie Pruned</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1   Scenario Summary</td>
<td>45</td>
</tr>
<tr>
<td>5.2   Branch and Bound Improvement</td>
<td>47</td>
</tr>
<tr>
<td>5.3   Pruning Results</td>
<td>50</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Problem Overview

Our problem comes from a work requirement posed by our Department of Defense (DoD) customer. The problem deals with radio networks, which are a set of radios that communicate through radio links between the radios. These networks are often modeled as unweighted and undirected graphs, with the radios represented as nodes (or vertices) and the edges represented as links. Thus, a radio network may be defined by $G = \langle N, L \rangle$ such that $N$ is the set of all nodes (radios) and $L$ is the set of all links (edges).

In such networks, some radios are end-user points in the network, while others may only exist to relay messages between radios that cannot form a direct link. We will call these end-user points in the network “relevant nodes”, while the others will be called “relays”. Our DoD customer gave us two primary requirements that exist for a network to effectively transmit messages between relevant nodes: low latency and redundancy. Low latency is important so that any relevant node in the network can quickly transmit a message to any other relevant node. In order to ensure this, the shortest path between any pair of relevant nodes must be kept small. Redundancy
requires that any pair of relevant nodes can still communicate quickly even if a link or a node drops out of the network. To ensure this, relatively short node-disjoint paths must exist between any pair of relevant nodes. We will refer to the combination of these two requirements as the “network requirements”.

A restriction on these networks exists which makes the above requirements more difficult to meet. Our radios can only support a limited number of timeslots, and each link attached to the radio consumes some of those timeslots. A radio with too many links will not be able to support them all. Therefore, the number of links per radio must be kept low. Specifically, we want to decrease the max degree of the network, which refers to the number of links attached to the node with the most links. This restriction competes with the aforementioned requirements since restricting the number of links reduces the options for forming shortest paths between nodes.

We will be taking graphs that include every possible link included as our input. Our goal, as specified by our customer, is to reduce the max degree, number of nodes with the max degree, and number of total links in the graph while still maintaining the shortest path and disjoint path requirements. The exact limits on the length of shortest paths and node-disjoint paths can be modified to produce a graph that meets a user’s latency desires. Imposing smaller limits will lead to decreased latency, but require more links to be kept. Our specific customer’s desire sets the shortest path requirements to 10 hops (links) and the node-disjoint path requirements to 12 hops.
1.2 Complexity Analysis

Our problem’s solution space can be envisioned as a full binary tree in which each node in the tree represents a link in the unpruned graph. The tree contains L levels, where L is the number of links in the unpruned graph. The top node represents the first link in the graph. Each node in the second level represents the second link. Continuing this pattern, each node in the bottom level represents the last link. The edge moving to the left child represents a choice to prune the parent, while the right child represents a choice to keep the parent. In this fashion, a path downward from the top node to one of the bottom nodes represents a keep/prune decision for each link in the graph.

This depiction of the problem space shows the complexity of the problem. Exactly one path exists to each bottom node, and the number of bottom nodes is \(2^L\), where L is the number of links in the graph. Therefore, \(2^L\) potential solutions exist. To evaluate every potential solution, the network algorithms would have to be called for each solution. A problem of this complexity at any non-trivial size cannot be searched quickly using a simple brute force method.

1.3 Comparison to Similar Problems

This problem can be formulated as a feature selection problem. In feature selection, one attempts to identify a subset of features which are most critical for creating an accurate model of a given domain [1] [2]. Feature selection can be framed
as a combinatorial problem, in which one must choose M features from N possible features. This problem has been shown to be NP-hard [3] [4].

In our problem, the full set of features is the full set of links in the input graph. Our objective is to identify the subset of M links from N input links which are most critical for meeting the network requirements. Both problems have $2^N$ possible solutions, where N is the number of items in the input set. Therefore, our problem is equivalent to a feature selection problem in terms of complexity and belongs to NP-hard.

1.4 Solution Methods

Since our problem is in NP-hard, we do not have a deterministic algorithm to solve it in a reasonable amount of time. However, any classical metaheuristic could in theory be applied to this problem. A metaheuristic is a general-purpose algorithm used to find approximate solutions to a range of computationally complex problems [5]. Examples of metaheuristics applied to feature selection problems include genetic algorithm, tabu search, and simulated annealing [6] [7] [8]. We chose to implement two different strategies to solve our problem. The first is a deterministic, exhaustive approach called branch and bound (abbreviated to BnB) and is presented in Chapter 3. The second is a best-effort metaheuristic called ant colony optimization (abbreviated to ACO) and is presented in Chapter 4.
CHAPTER 2

NETWORK REQUIREMENTS

2.1 Overview

This chapter will discuss in detail the ways in which the network requirements introduced in Chapter 1 will be checked. A depth-first search (DFS) algorithm runs to determine the connectivity of the graph and classify each edge. A chain decomposition algorithm is performed to determine edge-disjointedness and node-disjointedness. A breadth-first search (BFS) algorithm determines the shortest path lengths between pairs of relevant nodes. A newly created algorithm coined “detour” determines the longest path lengths in the graph given the removal of any node. A graph must pass each of these checks in order to be considered a solution to our problem. High-level pseudocode outlines this process in Figure 2.1.

2.2 Depth-First Search (DFS)

DFS is a widely known algorithm for exploring a graph [9]. Running a DFS on a graph determines if a graph is connected, i.e., that every node can reach every other node. DFS also creates an ordering of the graph’s nodes by the time of discovery in the search process. With a small modification to the general algorithm, DFS can also
classify each edge in the graph as a tree edge, which is an edge from a current node to a newly discovered node, or a back edge, which is an edge from a current node to a previously discovered node. The general strategy that was implemented for conducting DFS was borrowed from [9], and the pseudocode with the edge classification is shown in Figure 2.2.

The classification of these edges will be used in the chain decomposition algorithm. Note that if the DFS resulted in finding disconnected groups, then we can skip the rest of the algorithms since any potential solution must be connected. The DFS algorithm visits each edge and each node exactly once, so the total complexity for this algorithm is $\Theta(N+L)$, where $N$ is the number of nodes and $L$ is the number of links.
2.3 Chain Decomposition

Chain decomposition is an algorithm that quickly determines if a given graph has two edge-disjoint and two node-disjoint paths for each pair of nodes. We use this algorithm as detailed in Schmidt’s paper [10]. The algorithm starts with the original graph and the classification of tree and back edges found in the DFS. Each tree edge is transformed into a directed edge from the node found later to the node found earlier.
Each back edge is transformed into a directed edge from the node found earlier to the node found later. Note that this makes the edges directed in the opposite direction that the undirected edge was originally found in the DFS.

After transforming the edges, the chain decomposition process is executed on the directed graph. The nodes are looped through in order from the first node found to the last node found. For each node, the node is marked as visited. For each back edge attached to the current node, the path starting with the back edge is traversed and continued until a node that has already been visited has been found. As the traversal is executed, each node that is newly discovered is marked as visited. Each traversal in this manner forms a chain. Once all of the nodes’ back edges have been traversed, the chain decomposition is complete. This process visits each node and each edge exactly once, so the algorithm has the same complexity as DFS: $\theta(N+L)$.

At the end of this chain decomposition process, if an edge was not visited then the graph is not 2-edge-connected, i.e., it is not 2 edge-disjoint. Next, the chains discovered in the previous step must be checked for cycles. Any chain that starts and ends with the same node is a cycle. If more than one cycle exists, then the graph is 2-edge-connected but not 2-vertex-connected. Otherwise, the graph is 2-vertex connected. An example graph undergoing this process is shown in Figure 2.3.

### 2.4 Breadth-First Search (BFS)

The next network requirement is to guarantee that any relevant node can reach any other relevant node in 10 hops or less. To determine this, we must compute the shortest paths between every pair of nodes. Since our graph is undirected, we can
use a BFS to determine every shortest path from a single node. We can repeat this process for each relevant node to get all the shortest paths. We use the standard BFS algorithm shown in pseudocode in Figure 2.4, which was taken from [9].

Running the BFS algorithm on a single relevant source node determines the longest paths to all other relevant nodes from that source node. We must repeat this process then for every relevant node and take the maximum of the longest path counts as the longest path count in the network.

BFS, like DFS, touches each node and edge exactly once, so the complexity to run BFS for a single source node is \( \theta(N+L) \). Running BFS from every relevant node, then, gives us a total complexity for verifying this network requirement as \( \theta(R(N+L)) \),
where $R$ is the number of relevant nodes, $N$ is the number of total nodes, and $L$ is the number of links (edges).

Other all-pairs shortest-paths strategies were studied for this algorithm which boast slightly improved times (as a function of numbers of edges and nodes) than the detour algorithm [11], [12]. However, those algorithms are more beneficial for larger graphs. Our scenarios are relatively small; the issue is that we run BFS a large number of times on them. The overhead costs of the more complex algorithms outweighed using the BFS algorithm for graphs of our size.

### 2.5 Detour Algorithm

The final network requirement is based on finding paths between nodes even when a node in the network is removed. The motivation for this is to be able to
send messages between relevant pairs of nodes with low latency even when any given node drops from the network. In our problem, this represents the ability for a pair of relevant radios to be able to communicate on a short path even when any single radio in the network stops working. Specifically, we want to enforce that the alternate path between two relevant nodes should be less than or equal to 12 hops.

Recall that in the chain decomposition algorithm we determined whether or not the network has two node-disjoint paths. If this algorithm determines that the network does have two node-disjoint paths, then we know that there exists an alternate route between every pair of relevant nodes, since the dropping of one radio could only affect one of the two paths. However, this algorithm does not make any guarantees about the length of the disjoint paths. One could envision a cycle graph with 20 nodes. In this graph, a path exists in 10 hops or less for every pair of nodes. Additionally, each pair of nodes has a node-disjoint path available, which simply requires traversing the opposite way around the cycle. However, if a node drops from the network, the new shortest path between the dropped node’s neighbors becomes 18 hops. The chain decomposition algorithm does not consider the length of this second path. Therefore, we need an alternate algorithm to guarantee the length requirements.

We initially looked at an algorithm published by Bhandari [13] which finds the minimum possible lengths for two node-disjoint paths in a simple graph. However, this algorithm is too restrictive for our purposes. This guarantees that there are two completely node-disjoint paths that both meet the 12-hop requirement. We actually only want to guarantee that a 12-hop path still exists when a single node is removed (not all of the first path nodes removed). It is possible that a graph does not contain
two node-disjoint paths under 12 hops for a pair of nodes, but that a 12-hop path exists given the removal of any single node. Figure 2.6 illustrates this phenomenon. The top three figures from left to right show the full network, the shortest path, and the disjoint path. The four figures on the bottom show the shortest paths available given the removal of a single node along the original shortest path. Note that the removal of any node not on the shortest path would still result in using the shortest path, so we only check the cases in which we remove the shortest path nodes. In this example, we see that the shortest node-disjoint path that exists from the source to destination node is 13 hops, but removing any single node always yields an alternate path of no more than 11 hops.

This result leads us to create a new algorithm, which we’ll call the “detour” algorithm. This algorithm starts by removing a node and its links from the graph. BFS is called from each relevant node remaining in the network to calculate the new shortest path lengths. If any of the new lengths exceed 12 hops, then the requirement is failed and the algorithm terminates. Otherwise, the node that was removed is restored, and the process is repeated until all nodes have been removed and restored. If no path lengths failed, then the graph passes the network requirement. Figure 2.5 provides pseudocode for this process.

The detour algorithm is the most computationally-intensive algorithm in the network requirements. Let N represent the total number of nodes, R represent the number of relevant nodes, and L represent the number of links. As discussed earlier, the complexity of calculating all relevant shortest paths is \(\theta(R(N+L))\). In detour, this process is repeated N times. Therefore, the total complexity for detour is
\[ \theta(RN(N+L)) \]. Every other algorithm’s complexity is negligible compared to that of detour’s, so this also provides us with the complexity for checking all of the network requirements.

**Figure 2.5:** Detour Algorithm.
Figure 2.6: Detour Paths vs. Disjoint Paths. Top left shows a sample network, with two relevant nodes S and D. Top middle shows the shortest path from S to D, and top right shows the 13-hop node-disjoint path from S to D. The four figures on the bottom from left to right show the detour paths for the removal of nodes 0, 1, 2, and 3, respectively. All four detour paths have 11 hops or less.
CHAPTER 3

BRANCH AND BOUND

3.1 Introduction

Branch and bound (BnB) is a popular metaheuristic for generating solutions to computationally complex problems. It has been used to effectively generate solutions to many NP-complete problems, including feature subset selection [14], traveling salesman [15], and clustering [16]. In general, BnB algorithms recursively partition a solution space into smaller subsets, or branches, calculate the bounds on the best possible solution in each subset, and then proceed in only the branches which may contain a solution that beats the currently best-known solution [17]. These bounds calculations are problem-specific; the bounds for a traveling salesman problem are much different than the bounds for our problem. Therefore, while the general strategy of BnB is borrowed from literature, most of the algorithms that will be detailed in the algorithm subsections were innovated for this problem.

3.2 Generic Algorithm

We will begin our BnB algorithm by implementing the generic BnB algorithm. We use a recursive strategy to walk through each potential solution in the potential
solution space. We start by generating a list of all the links in the unpruned graph. We then prune the first link and check if the new graph passes. If it does, and if this graph is our new best graph (as determined by max node degree, number with max node degree, and total number of links), then we store this graph as our new best graph. If it does not pass, then we know that every potential solution in this branch of the tree will also fail, and we can implicitly reject all other potential solutions in the subtree. If it does pass, then we move to the next link and perform the same process recursively. After returning from the checks in the lower levels of the tree, we restore the link to the graph and move to the next link with the current link in the graph instead. Once we return from all of the subtrees, we will have considered (either explicitly or implicitly) every possible solution and identified the optimal one. This algorithm is shown in pseudocode in Figure 3.1.

GIVEN: input_graph

best_graph = input_graph
links_list = list of all links in input_graph
_solve(input_graph, first_link)

def _solve(cur_graph, cur_link):
    delete cur_link from cur_graph
    pass = network requirements(cur_graph)
    if pass:
        if better_solution(cur_graph, best_graph):
            best_graph = cur_graph
        if not at the end of links_list:
            _solve(cur_graph, next link in links_list)
    restore cur_link to cur_graph
    _solve(cur_graph, next link in links_list)

Figure 3.1: Generic Branch and Bound Algorithm.
3.3 Our Algorithm

As discussed in Chapter 1, there are $2^L$ potential solutions to consider, where $L$ is the number of links in the graph. Obviously, not every solution can be checked explicitly for any sizable input graph. Fortunately in a BnB algorithm much of the solution space may be checked implicitly by comparing to the solution bound. The solution spaces that don’t exceed the bound are the only spaces requiring explicit checks. An explicit check means a full evaluation of a potential solution. For our problem, that involves a network requirements check on the potential solution, which is where the majority of computation time occurs. Therefore, one of the main goals of a BnB algorithm is to implicitly check as many potential solutions as possible. In other words, the goal is to impose the tightest possible bounds on the solution spaces at each step.

An implicit evaluation means passing or failing a potential solution without calling the network requirements check. This requires us to be able to definitively eliminate potential solutions based only on some criteria that are more computationally efficient to calculate than the pass/fail algorithm. Many examples exist in literature in which BnB algorithms are able to implicitly check large percentages of the solution space. A K-nearest neighbor BnB algorithm explicitly checks approximately 6 percent of the solution space in order to find the optimal answer [18]. A project-scheduling BnB algorithm shows the ability to solve problems that were previously unsolvable without imposing bounds [19]. The following subsections will detail
the enhancements that we made to the generic algorithm that reduces the number of explicit checks in the solution space.

### 3.3.1 Unprunables

The first major implicit check to add to the base algorithm is the concept of “unprunable” links and unprunable combinations of links. A link is considered unprunable if the removal of that link alone causes the graph to fail the network requirements. We can test which links are unprunable in a simple process. For each link in the graph, remove the link. Check the graph requirements for the full graph minus the removed link, then restore the link. If this graph fails with the single link removed, then every potential solution must include this link. Therefore, when iterating through the $2^L$ potential solutions, where $L$ is the number of links, we can immediately rule out all potential solutions that do not include this link. The algorithm for finding the unprunable links is shown below in Figure 3.2.

```python
GIVEN: input_graph
      links_list

unprunable_links = []
for link in links_list:
    delete link from input_graph
    pass = network_requirements(input_graph)
    if not pass:
        add link to unprunable_links
        add link back to input_graph
return unprunable_links
```

**Figure 3.2:** Finding Unprunable Links.
The next step is to consider sets of links that, when pruned from the graph, cause the graph to fail. This requires a similar process to the above, except we must consider pairs of links instead of single links. To find the unprunable pairs, we need to consider every pair of links in the graph. We have already determined the single links that are unprunable, though, so we don’t need to recheck any pairs that include an unprunable link.

The algorithm for finding the unprunable pairs involves a nested loop. The outer loop iterates through all the links. The inner loop iterates from the next link after the outer loop’s link to the end of the links. This inner iterator only iterates from the outer link to the end instead of iterating through all the links so that we only consider each pair of links once. In each iteration of the outer loop, we check if the current link is unprunable. If so, there is no more work to do (since we don’t need to record unprunable pairs which include an unprunable link) and we continue to the next iteration. Otherwise, we remove a link, run through the inner loop, and restore the link. For each iteration in the inner loop, we check if the inner link is unprunable, and continue to the next iteration if it is unprunable. Otherwise, we remove a link, check if the network passes with the outer and inner link removed, and then restore the inner link. The pseudocode for finding the unprunable pairs is shown in Figure 3.3.

This pattern of finding unprunable sets of links may be continued for any order of sets. Each additional order requires an additional nested inner loop. The complexity of this preprocessing step becomes \( \theta(L^K) \), where \( L \) is the number of links in the graph and \( K \) is the size of the largest group checked. Clearly the runtime penalty
for checking unprunable sets of links rises exponentially with set size. However, discovering more sets of unprunable links in preprocessing leads to a quicker pruning of the potential solution space during the main algorithm’s solve stage. Therefore, the right balance between increasing preprocessing time and decreasing runtime must be found. Based on the results for our scenarios, the best overall runtime comes from considering sets of up to size three. Considering sets larger than this incurs too large of a runtime penalty in preprocessing for the time saved during the solve stage.

3.3.2 Minimum Node Degrees

As discussed earlier in the problem overview, we cannot explicitly solve this problem by a divide and conquer approach. However, we can gain more insight into limitations on potential solutions by applying divide and conquer. The more that we know about the best possible solution, the faster we can eliminate solutions from the
potential solution space. For instance, if we know that a given node must have at least L links in order to pass network requirements, then we can immediately rule out any potential solutions that do not have at least L links attached to the node. Fortunately, we can solve this subproblem relatively quickly for each node in the graph, and we can use essentially the same algorithm that we have developed to solve the full problem.

To determine the min degree requirements for a given node, we construct a list that only contains the links that include the given node. We begin the solve stage at the first link in the newly constructed list. We remove the link, then check if the graph passes the network requirements. If it passes, then we store this solution as a new best solution if it has a lower degree than the previous best solution. We then solve for the next link in the links list with the current link removed if it passed then network requirements. After recursively iterating through the solutions with the current link pruned, we solve for the next link with the current link remaining in the graph. We repeat this process for each node to determine the min degree required for each node. The pseudocode for this simplified branch-and-bound algorithm is shown in Figure 3.4.

Because we only consider the links attached to a given node in the solution space, this algorithm runs quickly. The max degree that we have on any input graph is 13 (more discussion on these inputs is presented in Chapter 5). This means that the largest solution space for an iteration of this algorithm is $2^{13}$. In practice, this simplified BnB sub-algorithm only has to explicitly check a small fraction of this solution space.
3.3.3 Check if Link is Prunable

The next improvement to the branch-and-bound algorithm is to include a check at each solution step to determine if a link may be prunable based on minimum node degrees and unprunable combinations. At the beginning of each solve step, we call this algorithm to determine if we can rule out the solution branch with the link pruned without calling the network requirements algorithm.

Figure 3.4: Calculating Minimum Node Degrees.

GIVEN: input_graph
nodes_list

min_deg_dict = {}
for cur_node in input_graph nodes:
graph_copy = copy of input_graph
node_links = links in links_list that touch node
node_links_length = length of node_links
min_deg_dict[cur_node] = node_links_length
cur_index = 0
link_stack = [(cur_index, graph_copy)]
while link_stack is not empty:
    pop cur_index, cur_graph off link_stack
    graph_link_on = copy of cur_graph
    cur_link = node_links[cur_index]
    delete cur_link from cur_adj_mat
    pass = network_requirements(cur_graph)
    if pass:
        if len(cur_adj_mat[cur_node]) < min(min_deg_dict[cur_node]):
            min_deg_dict[cur_node] = len(cur_adj_mat[cur_node])
            # recurse with cur_link off
        if cur_index != node_links_length - 1:
            link_stack.append((cur_index+1, cur_graph))
    # recurse with cur_link on
    if cur_index != node_links_length - 1:
        link_stack.append((cur_index+1, graph_link_on))
return min_deg_dict
The first check in this algorithm is to compare the degrees of the two endpoint nodes of the link to the minimum degrees calculated in the preprocessing step. In the current graph, if the degrees of either of these nodes are already at the min degree, then we know that pruning the current link will result in a graph that cannot pass network requirements. This means that we can skip the entire solution branch that prunes the current link.

The other check is to determine if the current link is part of an unprunable group in which all other links in the group have already been pruned. If this is the case, then we know that pruning this link will result in failing network requirements, and we can again rule out the solution branch that prunes the current link. This means that we must loop through each of the unprunable data structures in order to determine if this criterion is met. Fortunately, the runtime for these checks is small compared to the runtime of the entire branch-and-bound algorithm. The pseudocode for this check is shown in Figure 3.5.

3.3.4 Check if Potential Solution Exists

Another enhancement to the branch-and-bound algorithm is to check at each iteration of the solve stage if a potential solution may still exist in the current branch of solutions. This check uses the knowledge gained from calculating the unprunable combinations and the min degrees as explained in earlier sections. At each stage of solving, one of the two possible solve routes is to keep the current link and search for solutions with the current link in. However, sometimes keeping the current link in may make it impossible to reach low enough node degrees to beat the current best
solution found. For instance, consider a case in which the current best solution found has a max node degree of 5. Also assume that our current graph while solving has already kept 5 links attached to a given node, and now we are considering another link attached to that node. In this case, we know that including the current link will push the max degree past our current best solution, and no potential solutions in the current branch will be able to lower the max degree enough to find a new best solution. We can therefore skip the entire solution space at the current solve stage since no following stages could reduce the node degrees.

We can construct a quick algorithm for checking if a potential solution exists by using the results of the unprunable combinations and min degree requirements that were calculated in the preprocessing stage. We create a copy of the current

```python
GIVEN: input_graph
       link
       min_deg_dict

endPt1 = link[0]
endPt2 = link[1]

if endPt1 degree = min_deg_dict[endPt1]:
    return false
if endPt2 degree = min_deg_dict[endPt2]:
    return false

for each set S in the unprunable sets of links:
    if link is in S:
        if each other link in S is not in input_graph:
            return false

return true
```

**Figure 3.5:** Check if a Link is Prunable.
graph so we don’t modify the original. From the current link to the end of the links list we check if the link is prunable using the strategy described previously. If it is prunable, then we remove it from the new graph. We now have a modified graph with every link pruned that could be pruned, as determined by the check described in the previous subsection. We calculate the max degree, number with max degree, and link count for this modified graph. We may need to increment these characteristics if any of the node degrees in the modified graph fall below the min degree requirements as calculated in preprocessing. For each node in the modified graph, if the node’s degree is below the min degree found in preprocessing, we increment the characteristics of the modified graph accordingly. Once we have modified the characteristics, we return a boolean indicating if these graph characteristics represent a better solution than the currently found best solution.

Note that this algorithm only provides a lower bound on the potential of a solution in the current graph. It may return graph characteristics that are actually unattainable. However, the lower bound is often worse than the current best solution, so it still works to trim the potential solution space and reduce explicit solution checks. Pseudocode for this algorithm is shown in Figure 3.6.

3.3.5 Sorting the Links List

Another improvement for the branch-and-bound algorithm is found by properly sorting the links list, which determines the order of links in the solution space that we search. We found that placing the more critical links at the front of the links list leads to fewer explicit solution checks needed. By having critical links up
Figure 3.6: Check if a Potential Solution May Exist.

front, we quickly eliminate large portions of the solution space in the beginning of the algorithm, since we quickly find combinations of pruned links that lead to failing network requirements.

Recall from the unprunables section that we found the links in the links list that, when removed, cause the graph to fail the network requirements on their own. We can delete all of those links from our links list that we consider for pruning so
that every potential solution includes those links. We order the remaining links by the frequency with which they appear in unprunable groups. We calculate the number of unprunable pairs that each link belongs to, and sort the links list such that links that belong to the most unprunable pairs are at the beginning of the list. We then find the number of unprunable triples that each link belongs to and use this number as a secondary sorting comparison after the number of unprunable pairs. Sorting in this manner allows the algorithms for checking for potential solutions and link prunability to rule out many of the early branches, which in turn eliminates large portions of the potential solution space.

3.3.6 Finding an Initial Solution

More potential solutions can be ruled out if a better current solution exists. For instance, if the best current solution has a max degree of 4, we can rule out all potential solutions that have max degrees of 5 or more. Therefore, it is advantageous to find as good of a solution as possible early on, so we do not waste time checking worse solutions. For this reason, we developed a quick algorithm to run in the preprocessing stage that produces an initial solution.

We calculate the initial node degrees for each node in the input graph. We then prune a single link that includes one of the nodes that currently has the max degree in the graph. We check the network requirements, and if the network fails then we try the rest of the links attached to the max degree node until we can remove one and pass the network requirements. Once we find one that passes, we decrement the relevant node degree counts and repeat the process for the new node that has the
max degree. We continue this process until we can no longer prune any links from the max degree node. We then try to prune every other link in the graph one-by-one. The final solution and its characteristics are stored as the initial solution.

In practice, this initial solution step often comes within one or two degrees of the optimal max degree solution. Since the algorithm’s complexity is linear with the number of links in the input graph, this algorithm incurs almost no runtime penalty but saves large amounts of network requirements checks during the solve stage. The pseudocode for this algorithm is shown in Figure 3.7.

### 3.3.7 Final algorithm

We can now combine the algorithmic improvements described in the previous subsections to create our final BnB algorithm, depicted in pseudocode in Figure 3.8.
Figure 3.7: Generate an Initial Solution.

```
GIVEN: input_graph
    links_list
    min_deg_dict

cur_graph = copy of input_graph
degree_dict = dictionary of node to that node's degree in input_graph
cur_unprunable_links = empty set

while true:
    max_node = node in degree_dict with max degree
    successful_prune = false
    for neighbor in cur_graph[max_node]:
        link = (max_node, neighbor)
        if link not in cur_unprunable_links:
            delete link from cur_graph
            decrement appropriate degree_dict counts
            pass = network_requirements(cur_graph)
            if not pass:
                add link to cur_unprunable_links
                restore link to cur_graph
                increment appropriate degree_dict counts
            else:
                successful_prune = true
                break out of for loop
        if not successful_prune:
            break out of while loop

sort links_list so that links with higher degree nodes are first
for link in links_list:
    endPt1 = link[0]
    endPt2 = link[1]
    if link in cur_unprunable_links:
        continue
    if degree_dict[endPt1] == min_deg_dict[endPt1]:
        continue
    if degree_dict[endPt2] == min_deg_dict[endPt2]:
        continue
    remove link from cur_graph
    decrement corresponding degree_dict counts
    pass = network_requirements(cur_graph)
    if not pass:
        restore link to cur_graph
        increment corresponding degree_dict counts

return cur_graph
```
GIVEN:
  input_graph
  relevant_nodes

best_graph = input_graph
links_list = list of all links in input_graph
get_node_min_degrees()
get_unprunable_combos()
sort_links_list()
get_initial_solution()
cur_graph = input_graph
cur_link = first link in links_list
_solve(cur_graph, cur_link)

def _solve(cur_graph, cur_link):
    if not is_link_prunable():
        if potential_solution_exists():
            _solve(next link, cur_graph)
    else:
        delete cur_link from cur_graph
        pass = network_requirements(cur_graph)
        if pass:
            if cur_graph better than best_graph:
                best_graph = cur_graph
            if cur_link not at end of links_list:
                _solve(cur_graph, next link)
            restore cur_link to cur_graph
        if potential_solution_exists():
            _solve(cur_graph, next link)

Figure 3.8: Final Branch and Bound Pseudocode.
4.1 Introduction

Beginning in the early 1990s Marco Dorigo and others began publishing papers on a new metaheuristic called ant colony optimization (ACO) for combinatorial optimization problems [20] [21] [22]. Since then, ACO has been applied to a wide range of complex problems, including transportation problems [23], shortest path problems [24], subset selection problems [25], and more.

ACO algorithms arose out of an observation of ant colonies finding shortest paths between their destinations [20]. As an ant walks it deposits pheromone on its path. Other ants that are nearby can sense this pheromone and are more likely to follow the pheromone trails than to follow a path without any pheromone. The lucky ants who find a short path between their destinations will likely retrace their path to follow their own pheromone, depositing more pheromone as they walk. The unlucky ants that wander without finding a good path spread their pheromone out more thinly than the ones on the shortest paths. Pheromone evaporates over time, so old paths become unused if no new ants follow them. Eventually, the ant colony converges to follow the shortest path with the most pheromone.
As ACO algorithms have evolved they have strayed further from their biological inspiration, and newer models with their own quirks have distinguished themselves to specialize in certain fields [26]. However, they all keep the same principle. Near the beginning of execution, ants are mostly free to explore anywhere in the solution space. Over time, they converge on the most promising solution areas until a good-enough solution is found or an execution limit is reached.

4.2 Network Solution in Ant Form

Before discussing the ACO algorithm form, we need to determine how to represent our graph problem in ACO terms. As discussed before, our problem is essentially a subset selection problem, in which we are attempting to find the optimal subset of links out of the full unpruned set of links. A solution, therefore, consists of a set of links. This means that each ant needs to make a keep/prune decision on each link in the input graph. A solution can then be represented by a keep/prune flag for each link. For ease of discussion, we will assume that we have an array of links. A potential solution then consists of an array of boolean values, each one indicating if the corresponding link is kept in the network or not. In practice, this system may be implemented in a variety of different ways, including an N-bit number, a hex string, or other forms.

Pheromone levels dictate the probabilities with which individual links are kept or pruned in a new solution. We use a decimal value between 0 and 1 to represent the pheromone level for each link. A value of 0.75 means that the corresponding link has a 75 percent chance to not be included in the network, and therefore has a 25
percent chance to be included in the network. Thus these pheromone levels can be thought of as the pruning probability for each link.

To construct a solution we loop through the pheromone values list. In each iteration, we generate a random number between 0 and 1. If this random number is greater than the current pheromone value then we keep the link; otherwise, the link is pruned. In this fashion, we probabilistically generate a new potential solution such that links with high pheromone values are likely to be pruned, and vice-versa.

4.3 Generic Algorithm

The ACO algorithm takes as input the full input graph, the list of relevant nodes, and a list of ACO tuning parameters that will be discussed later. In initialization it creates the list of links, a list of initial pheromone values, and stores the input graph and its characteristics as the current best solution. In the pruning stage, a while loop executes until a specified number of iterations has been met. In an iteration, a new potential solution is generated based on the current probabilities of pruning each link. The network requirements are checked against the potential solution. If the solution passes the checks then it is stored as the new best solution. The pheromone levels are updated before proceeding with the next iteration. Pseudocode for the generic algorithm is shown in Figure 4.1.

4.4 Algorithm Subprocedures

The generic algorithm described in the previous section only gives the general form of the ACO algorithm. The implementation of the subprocedures that are
GIVEN:  input_graph
       links_list
       max_iterations

create link_probabilities with initial probability for each link
best_graph = input_graph
best_characteristics = input_graph characteristics
iter = 0

while iter < max_iterations:
    new_solution = generate_new_solution()
    pass = network_requirements(new_solution)
    if pass:
        if new_solution better than best_graph:
            store new_solution as best_graph
        update_pheromone()
    iter++

**Figure 4.1:** Ant Colony Optimization Initial Algorithm Overview.

included in the ACO algorithm determines the algorithm’s performance. This section will discuss the subprocedures included in our algorithm, as well as discussion on how algorithms in literature implement them.

### 4.4.1 Pheromone Update Frequency

Much of the distinction between various ACO algorithms comes down to how often pheromone levels are updated. In the original ant algorithms, called Ant System, every ant deposited pheromone on its path and the pheromone levels were all partially evaporated after each ant [26]. In the Max-Min Ant System, pheromone levels are updated after a set of ants form solutions [27]. Another strategy called Ant Colony System updates pheromone locally as a group of ants form solutions, and then update the overall pheromone level by only the best ant [28]. In the elitist ant system, only the best ant or ants update the pheromone levels [29].
Several different strategies were tested on our problem. We attempted the Ant System approach of updating every pheromone level after every ant. In this system, if the solution that was generated passed the network requirements, then the pruning probability was increased for the links in the current solution that were pruned and decreased for those that were kept in the graph. This performed poorly because many solutions passed the network requirements but were not good solutions because they still had high node degrees. Since graphs with more links kept are more likely to pass the network requirements, more of these were updating the pheromone level. As the algorithm ran, the potential solutions contained more and more links, which converged away from the optimal solution which contains few links.

We then implemented a version of the elitist system. Our strategy for this was to only modify the pheromone levels when a new best solution was found. If the new solution was better in terms of max degree, number with max degree, and number of links, and the solution passed the network requirements, then the pruning probabilities were increased for links pruned from the current solution, and decreased for links still present. This improved the results of the algorithm because the pheromone levels quickly converged to support solutions with low degree counts. However, the drawback of this approach was that it converged too quickly. Since the pheromone levels were only modified when a new best solution was found, the algorithm quickly found a local maximum solution and was unable to reach outside of it.

The final version of our pheromone update frequency is a combination of the two previous strategies. Updating on every solution explored too much, while updating only on new best solutions narrowed in too quickly. The new pheromone strategy
updates only when the conditions numbered below are met. These requirements enforce that bad solutions don’t influence the pheromone levels like in the first system, while also avoiding getting stuck in a local maximum like in the max-min system.

1. The solution passes the network requirements.

2. The solution has an equal or lesser max degree than the current best solution.

3. If the solution has an equal max degree to the current best max degree, then the number of nodes with the max degree in the solution must be less than or equal to the current best number with max degree plus 1.

4.4.2 Pheromone Update Method

Another key distinction between various ant systems is the method by which pheromone levels are updated. Recall that for our problem the pheromone level is a value between 0 and 1 that represents the probability that a given link is pruned from a solution. If we update this value too quickly, then the algorithm converges early on a solution and can get stuck in a local maximum. If the value is updated too slowly, though, the more promising areas of the solution space will not be searched thoroughly enough. Therefore, the right balance must be struck in the weight with which the pheromone levels change.

A wide variety of pheromone update equations exist among the different ant systems. The equations drive the speed with which the pheromone values increase and decrease, as well as the minimum and maximum levels. For our problem, we originally used a simple strategy involving two factors: pheromone factor and evaporation.
factor. Upon finding a new solution, all links that were pruned from the solution have their pheromone multiplied by the pruning factor. All links that were kept in the solution have their pheromone multiplied by the evaporation factor. Additionally, min and max pheromone levels were set. We ran large parametric sweeps to test combinations of evaporation, pheromone, min, and max values to determine what values worked best. Evaporation tended to succeed around 0.7, while pheromone succeeded around 2. Min and max values worked better for larger scenarios if they were closer to 0 and 1, respectively, while smaller scenarios worked better with values closer to 0.1 and 0.9. This meant that upon finding a new solution, links pruned from the solution had their pheromone values doubled (capped by max), while other links were multiplied by 0.7 (capped by min).

This strategy worked well in quickly finding a “good” solution but struggled to improve on that solution as the algorithm continued. One of the faults of this strategy was that the min and max probabilities needed to be manually set differently based on the size of the input graph. Beyond that, we believe that the pheromone factor yielded too steep of an increase in pruning probability. If a link’s pheromone level was relatively low at 0.25, then in only two pheromone increases this level would reach the max. This revelation led us to develop a better system of equations.

In the initialization stage of the algorithm, we added a step to determine the unprunable links, as done in the initialization stage of the branch and bound algorithm in the previous chapter. These unprunable links are assigned a pheromone level of 0, meaning that they will never be pruned. The number of prunable links is calculated as the number of total links in the input graph minus the number of unprunable links.
The min probability is set to 3.0 divided by the number of prunable links. The max probability is set to 1 - the min probability. This allows for the probability limits to be set closer to the extremes for larger networks, and closer to the middle for smaller networks. This effectively means that if every link’s probability is at the min or max probability, then each newly generated solution will swap the status of 3 links on average. The value of 3 was empirically chosen based on large numbers of tests.

Once these new minimum and maximum rules were set, new pheromone and evaporation equations were created to modify the ascent/descent of the pheromone. Our idea was to make a link’s pheromone level move in a magnitude proportionate to the difference between its current level and the limit that it is moving towards. Additionally, the ascent weight should be higher than the descent weight, so that more links are pruned over time. This leads us to our two new equations, with the factors decided by further parametric sweeps over various inputs. Upon a link being pruned from a new best solution, the link’s new pheromone level is incremented by one half of the distance between the current level and the max level. A link that is kept in a new best solution is decremented by one third of the distance between the current level and the max level. Pseudocode for this final version of the pheromone levels is shown in Figure 4.2.

4.4.3 Initial Pheromone

Another point of experimentation in tuning the ACO algorithm was the strategy for setting the initial pheromone levels for each link. From a cursory glance at an input graph we can quickly see that some links will likely be easier to prune from
the graph than others. For instance, if node A has 10 neighbors and node B has 5 neighbors, then it is more likely that several of node A’s links could be pruned and still meet network requirements than it is for node B’s since more redundant paths exist around node A. Our initial idea was to scale the initial pheromone level for each link by the smaller degree of the link’s endpoints. Links attached to two high-degree nodes would have a high probability of being pruned, while links whose nodes have a lower degree would be more likely to be kept. Parametric sweeps were conducted to experiment with these initial pheromone values, giving more initial pheromone to higher degree links and less to lower degree links. However, none of these settings produced as good of a solution as simply setting each link’s probability to the same value. We believe that attempting to guide the algorithm with different initial pheromone levels resulted in converging to local maximum solutions too quickly and discouraged the initial wide search that considers the entire solution space. Most ACO literature uses a similar strategy for initializing pheromone levels [30] [31].

Figure 4.2: Pheromone Equations.
The initial pheromone levels should all be the same, but that level still needs to be set. If it is set too high and only changed upon finding a new solution, then too many links will be pruned and the potential solutions will never pass the network requirements. Therefore, we need a strategy that will place the pheromone levels at a point at which potential solutions may still be found. After attempting several different methods, the best approach that we found uses an iterative method. In the initialization stage of the ACO algorithm, each link’s pheromone level is set to 0.5. This means that a candidate solution will randomly choose approximately half of the links to prune from the network. A boolean flag is set to false which indicates whether the first potential solution has been found. In the solve stage, while a first solution has not been found, every pheromone level is multiplied by a factor less than 1 to reduce the pheromone level. After a short number of iterations, the pheromone levels drop enough so that a newly produced solution keeps enough links to pass the network requirements. Experimentation with this value led us to set this factor at 0.97.

Using this dynamic approach to setting the initial pheromone levels greatly improved solution qualities. However, it still converged too quickly on a local maximum, and this maximum usually shared many characteristics with the initial solution found. In other words, the initial pheromone levels had too large of an influence over the direction of the solution space searching. To remedy this, once the initial solution is found, each pheromone level is multiplied by 0.5. This encourages more early exploration and avoids premature convergence.
4.4.4 Caching Tested Graphs

The last improvement to our ACO algorithm was to ensure that we do not call the network requirements multiple times for the same graph. Each potential solution can be represented by an array of booleans, each indicating if the corresponding link in the links list is kept. In the initialization phase, we create an empty set to hold the boolean arrays and then add the input graph’s array, which is an array with all values set to true. In the solve stage, every time that we generate a new potential solution our set cache is checked for that solution’s boolean array. If it is present in the cache, then we can continue to the next iteration of the solve loop. Note that this now means that the max iterations setting only refers to the number of potential networks generated, not the number of network requirements check. As the number of max iterations increases, more and more solutions will fill the cache and the network requirements will be called less often. Since the vast majority of computation time is spent in calculating the network requirements, this caching mechanism saves a large amount of time for larger scenarios.

For the size of our input scenarios, this caching mechanism was effective. However, note that care must be taken if this algorithm is run on larger inputs for long iteration counts. The current implementation uses an ever-growing cache to hold every found solution. With too large of a problem, this could present memory issues as the cache grows. This is an easy problem to work around, though. A user can set a limit for the size of the cache to grow to, dependent on system resources. The cache will then add every potential solution generated until the cache hits the
preetermined limit, and then new solutions will replace the oldest solutions in the cache. If memory is a large concern, the storage of each solution can be efficiently represented as an N-bit array, where N is the number of links. Our biggest scenario, Charlie, contains 257 links, and can therefore be stored in 257 bits, which fits into 33 bytes. A machine with 1GB of memory could store over 30 million such solutions in its cache.

4.5 Final Algorithm

We now have all of the pieces to put together the final version of the ACO algorithm. This algorithm is shown in pseudocode in Figure 4.3.
Figure 4.3: Ant Colony Optimization Final Version.

```plaintext
GIVEN:  input_graph
         links_list
         max_iterations

create link_probabilities with initial probability for each link
best_graph = input_graph
best_characteristics = input_graph characteristics
unprunable_links = get_unprunable_links()
tested_hashes = set()
tested_hashes.add(input_graph)
pheromone_levels = 0.5 for all links
init_evap = 0.97
first_sol_found = false
for link in unprunable_links:
    pheromone_levels[link] = 0
iter = 0

while iter < max_iterations:
    iter++
    new_solution = generate_new_solution()
    if new_solution in tested_hashes:
        continue
    else:
        tested_hashes.add(new_solution)
        new_characteristics = get_characteristics(new_solution)
        if new max degree > best max degree
            || (new max degree == best max degree
            && new num with max degree > best num with max degree + 1):
                continue
        pass = network_requirements(new_solution)
        if pass:
            if not first_sol_found:
                pheromone_levels = 0.5 * pheromone_levels
                first_sol_found = true
            if new_solution better than best_graph:
                store new_solution as best_graph
            for link in links_list:
                old = pheromone_levels[link]
                if old != 0:
                    if link is in current_graph:
                        pheromone_levels[link] = old - (old - min_level) / 3
                    else:
                        pheromone_levels[link] = old + (max_level - old) / 2
                if not first_sol_found:
                    pheromone_levels = init_evap * pheromone_levels
```

Figure 4.3: Ant Colony Optimization Final Version.
CHAPTER 5

TESTING AND RESULTS

5.1 Input Scenarios

We tested our algorithms on hundreds of input graphs that were derived from DoD network scenarios. While our algorithms ran against all of the input graphs, we will focus in-depth on three scenarios: Alpha, Bravo, and Charlie. Visual representations of these three scenarios can be seen in Appendix A.

Alpha is the smallest input graph by both number of nodes and number of links. The max degree in Alpha is only 5, and the number of links per node is just over 3. With relatively few links per node, Alpha does not contain many links that can be pruned while still meeting the network requirements. Due to the small size of the network, the network requirements were easy to meet while keeping the number of links per node low.

Bravo contains more than twice the number of nodes as Alpha and almost four times as many links. It contains two densely clustered groups of nodes, with the largest degree on a node reaching up to 10. The most difficult part of solving Bravo is pruning links from the bottom left cluster while keeping path lengths to the rest of the network under the network requirement limits. With significantly more links
Table 5.1: Scenario Summary

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Alpha</th>
<th>Bravo</th>
<th>Charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio Nodes</td>
<td>13</td>
<td>37</td>
<td>59</td>
</tr>
<tr>
<td>Relay Nodes</td>
<td>9</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>Total Nodes</td>
<td>22</td>
<td>51</td>
<td>69</td>
</tr>
<tr>
<td>Total Links</td>
<td>35</td>
<td>137</td>
<td>257</td>
</tr>
<tr>
<td>Links Per Node</td>
<td>3.18</td>
<td>5.37</td>
<td>7.45</td>
</tr>
<tr>
<td>Max Degree</td>
<td>5</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

overall and more links per node, Bravo poses a much more challenging problem than Alpha.

Charlie presents the largest problem, as it contains more nodes than Bravo and almost twice as many links. It has several clustered node groups with a max degree of 13. The longest paths in this scenario involve traveling between nodes in the top right and bottom right clusters. Even in the unpruned graph, many of the shortest paths between such nodes involve 10 hops. The largest challenge for Charlie is to keep enough links on these nodes to maintain the 10-hop limits while significantly decreasing the max degree. With the highest number of nodes, links, links per node, and max degree, Charlie is the most difficult problem to solve. Table 5.1 summarizes the graph characteristics of the three inputs by which our algorithms were primarily tested against.
5.2 Testing Environment

Our algorithms were written in Python 2.7 and executed on a 64-bit Windows 8 machine with an 8-core Intel i7 CPU and 16GB of RAM. During the testing of our algorithms, we recorded the number of network requirements checks performed and the execution time. Since our BnB implementation is deterministic, we only ran our algorithm once on each input after changing our algorithm. Our ACO algorithm is a stochastic algorithm, though, so each update to the algorithm was followed by a test of 100 runs with different random number seeds on each input graph.

5.3 Branch and Bound Results

Over the course of developing the algorithm, the Alpha scenario was primarily used as the benchmark to determine algorithmic improvements. The generic branch-and-bound implementation took several hours to find the optimal answer and performed over a million calls to the network requirements check. The final version runs in under 20 seconds and calls the network requirements less than 10,000 times. Recall that Alpha contains 35 links, which leads to a total solution space of \(2^{35}\) links. The final algorithm explicitly searched about \(2^{13}\) potential solutions. This means that the algorithm only needed to explicitly check about 0.00002 percent of the total solution space, and all the other solutions were implicitly checked. The improvement in explicit checks and run-time is shown in Table 5.2. Appendix A shows both the pruned and unpruned versions of Alpha.
Table 5.2: Branch and Bound Improvement

<table>
<thead>
<tr>
<th>Improvement</th>
<th>Network Requirements Checks</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic Algorithm</td>
<td>1,194,301</td>
<td>9,149</td>
</tr>
<tr>
<td>Unprunables</td>
<td>289,049</td>
<td>1,773</td>
</tr>
<tr>
<td>Minimum Node Degrees</td>
<td>173,482</td>
<td>1,282</td>
</tr>
<tr>
<td>Is Link Prunable</td>
<td>112,294</td>
<td>702</td>
</tr>
<tr>
<td>Potential Solution Exists</td>
<td>38,023</td>
<td>142</td>
</tr>
<tr>
<td>Sorting Links List</td>
<td>25,930</td>
<td>39</td>
</tr>
<tr>
<td>Initial Solution</td>
<td>12,815</td>
<td>23</td>
</tr>
<tr>
<td>Final Algorithm</td>
<td>9,503</td>
<td>18</td>
</tr>
</tbody>
</table>

The branch-and-bound algorithm was unable to solve the Bravo and Charlie laydowns. The final algorithm was run for two weeks on both Bravo and Charlie, and neither run came close to identifying the ideal solution for either run. The best max degree that had been identified after the two weeks for both Bravo and Charlie was 5, which is worse than the best solutions found by our ACO solution (whose results will be described in the next section). The algorithms were even run with the ideal starting solution set to the best solution found by the ACO algorithm but were still unable to finish. The exponential growth in problem complexity as the number of links grows outweighs any algorithmic improvements that we were able to make over the course of developing the algorithm. While this algorithm worked quickly for smaller graphs, it was unable to solve the larger problems.

One drawback of using the branch-and-bound algorithm is that parallelism provides little benefit. While the solution space can easily be split into equal parts by assigning different subtrees to different processors, the performance improvement
is small compared to the complexity of the algorithm. With an L-link input graph, the complexity of the problem is $2^L$, and splitting the problem into n subtrees only reduces the complexity for each processor to $2^{L-1}$. Additionally, some parts of the solution space can be easily ruled out, while others require many more explicit checks. This means that in addition to needing an unreasonable number of processors to significantly reduce complexity, it is a difficult task to evenly split the actual work.

Part of the difficulty in solving this problem via BnB is that the necessity of a link being included in a graph is highly dependent on what other links are included or pruned. When considering a given link to prune or keep, the number of links currently attached to the considered link’s two endpoints heavily influence whether or not this link can be pruned. As more links are removed, it becomes more difficult to prune a link and still maintain the path length requirements. The majority of links in a dense network like Bravo or Charlie are attached to nodes with high degrees, which means that many of these links’ abilities to be pruned depend on how many links around them have already been pruned. BnB algorithms work more effectively on subset selection problems in which the items’ benefit in the set are largely independent of the other items in the set [32].

5.4 Ant Colony Optimization Results

Since ACO is a stochastic algorithm, running the algorithm with a different random number seed results in a different enumeration of the solution space. To determine the algorithm’s effectiveness, the results of the algorithm need to be measured across many different seeds. Fortunately, one of the large advantages of using a
metaheuristic like ACO is that differently-seeded runs are independent of each other. This allows a user to run as many seeds as they desire in parallel, capped only by hardware constraints. This is in sharp contrast to the BnB algorithm, which cannot be easily split into equal parallel sections.

Experimenting with different levels of max iterations and numbers of seeds found that a surprisingly small number of iterations are needed if the number of seeds is large. For Alpha, most seeds found the optimal solution in several thousand iterations, with each seed running less than a second. For Bravo, most seeds found a good solution within 20,000 iterations, with each seed running around 20 seconds. For Charlie, most seeds found a good solution with 50,000 iterations, with each seed running around 70 seconds. The pruned versions of these scenarios shown in Appendix A were the best results of running the ACO pruning algorithm on the inputs. Table 5.3 shows the final pruning results of the three scenarios.

We considered this algorithm to be a large success for all input scenarios. The best solution found for Alpha matched the optimal solution found by the BnB algorithm and ran in a similar amount of time. The best solutions found for Bravo and Charlie, which took 20 seconds and 70 seconds respectively, both beat the best solutions found after two weeks of running the BnB algorithm.

Recall that the primary objective set by our DoD customer was to reduce the max degree of the networks. We were able to prune Alpha down to a max degree of 3, and both Bravo and Charlie to max degrees of 4. This was a marked improvement, considering that Bravo and Charlie’s initial max degrees were 10 and 13 respectively. Note especially how the pruned versions of Bravo and Charlie in Appendix A have
Table 5.3: Pruning Results

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Alpha</th>
<th>Bravo</th>
<th>Charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Links Before Pruning</td>
<td>35</td>
<td>137</td>
<td>257</td>
</tr>
<tr>
<td>Links After Pruning</td>
<td>25</td>
<td>69</td>
<td>92</td>
</tr>
<tr>
<td>Links per Node Before Pruning</td>
<td>3.18</td>
<td>5.37</td>
<td>7.45</td>
</tr>
<tr>
<td>Links per Node After Pruning</td>
<td>2.27</td>
<td>2.71</td>
<td>2.67</td>
</tr>
<tr>
<td>Max Degree Before Pruning</td>
<td>5</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>Max Degree After Pruning</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

removed the large majority of links in the dense node clusters in the input graphs. Our customer was happy with these results and said that these max degrees would enable the radios to properly support all of their links.
CHAPTER 6

CONCLUSIONS

6.1 Results

We were successful in developing a solution to solve our problem for the given inputs in a short amount of runtime. While the branch and bound algorithm is unable to solve the problem for larger inputs, it does work quickly for smaller networks. Additionally, the work on this algorithm revealed insights that helped develop the ant colony algorithm, most notably the concept of unprunable links. The ant colony optimization algorithm successfully generates quality solutions in little runtime for all three of our scenarios. Our DoD customer stated that our solution pruned all of our input networks to a sufficient degree in a reasonable amount of runtime.

6.2 Future Work

One possible approach to generate faster solutions with our BnB algorithm is to configure it to find a “good enough” solution instead of the optimal solution. Since we know what the likely optimal solutions are from our ACO algorithm, we could modify our BnB algorithm to search for a solution that comes close to our ACO solution. Many examples exist that use BnB algorithms to quickly find near-
optimal solutions, including algorithms in the fields of density estimation [33] and single machine tardiness scheduling [34].

Additional work could be put into the algorithmic enhancements that we created for our BnB algorithm. While our enhancements dramatically cut down the explicit searches of the solution space, there are likely further improvements that could be made. One potential area for improvement is to add additional preprocessing steps to determine larger groups of unpromising links. For instance, the min node degree check could be executed on pairs of nodes in addition to single nodes.

6.3 Applications to Other Problems

Both the branch and bound algorithm and ant colony algorithm can be applied to any problem that can be represented as a simple graph. Most of the problem-specific features are captured in the network requirements algorithm. By replacing the network requirements with a different set of requirements, one could reuse the same algorithms developed here to solve a different problem. As our problem is in the NP-hard domain, our algorithms for solving it could likely be adapted to different NP-hard problems like the traveling salesman, graph partitioning, and Boolean satisfiability.
APPENDICES
APPENDIX A

SCENARIO LAYDOWNS

A.1 Alpha

A.2 Bravo

A.3 Charlie
Figure A.1: Alpha Unpruned.
Figure A.2: Alpha Pruned.
Figure A.3: Bravo Unpruned.
Figure A.4: Bravo Pruned.
Figure A.5: Charlie Unpruned.
Figure A.6: Charlie Pruned.
REFERENCES


