AN ANT COLONY APPROACH TO THE
BANDWIDTH PACKING PROBLEM

by

Aaron W. Smith

B.S., Eastern Oregon University, 1996

A thesis submitted to the
University of Colorado at Denver
in partial fulfillment
of the requirements for the degree of
Master of Science
Applied Mathematics

2000
This thesis for the Master of Science
degree by
Aaron W. Smith
has been approved
by

Harvey J. Greenberg

S. Anthony Cox, Jr.
Anthony Cox

Richard Lundgren

4/19/00
Date
Smith, Aaron W. (M.S., Applied Mathematics)

An Ant Colony Approach to the Bandwidth Packing Problem

Thesis directed by Professor Harvey J. Greenberg

ABSTRACT

This thesis presents an Ant Colony Optimization (ACO) approach to the Bandwidth Packing Problem (BWP). BWP is a combinatorially difficult problem arising in the area of telecommunications involving selecting and routing a set of calls in a capacitated network. BWP consists of finding an assignment which maximizes profit while satisfying network bandwidth constraints. Several approaches to BWP are described including an implementation of ACO. ACO is a metaheuristic based on the foraging behavior of ant colonies utilizing both local and historic data to develop solutions to the problem at hand. A description of the algorithm as well as empirical results are given. These results compare favorably to the results found by other methods in terms of profit.

This abstract accurately represents the content of the candidate's thesis. I recommend its publication.

Signed

Harvey J. Greenberg

iii
ACKNOWLEDGMENTS

Special thanks to Harvey Greenberg for all of his effort and guidance. Also thanks to Tony Cox and Mark Parker for their assistance.
CONTENTS

Figures ................................................................. vi
Tables ................................................................. vii

Chapter
1  Introduction .......................................................... 1
2  The Bandwidth Packing Problem ................................. 3
3  The Metaheuristic Approach ..................................... 12
   3.1 Genetic Algorithms ........................................... 13
   3.2 Tabu Search ................................................... 17
      3.2.1 Tabu Search I ............................................. 20
      3.2.2 Tabu Search II ........................................... 21
4  Ant Colony Optimization ........................................ 24
   4.1 Evolution of Parameters ..................................... 32
5  Conclusions ....................................................... 43

Appendix
Glossary of Symbols and Notation ............................... 46
FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simple Network</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Roulette Wheel Parent Selection</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Uniform Order-Based Crossover</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Scramble Sublist Mutation</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Simple Neighborhood Search</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Network for Neighborhood Search</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Ants Finding The Shortest Path</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Ant Colony Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Network for Test Problem</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Variation of $m$</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>Variation of $\alpha$</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Variation of $\beta$</td>
<td>39</td>
</tr>
<tr>
<td>4.7</td>
<td>Variation of $(1 - \rho)$</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Variation of $q_0$</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>Variation of $\sigma$</td>
<td>42</td>
</tr>
</tbody>
</table>
TABLES

Table

2.1 Simple Call Table ......................................................... 5
2.2 Arc Definitions ........................................................... 5
2.3 Call Table for IP ............................................................ 9
3.1 Call Table for Neighborhood Search .............................. 19
4.1 Comparison Results ...................................................... 33
4.2 Call Table for Test Problem ........................................ 35
1. Introduction

Communications networks are designed for the transfer of information. This transfer of information must be done efficiently so as to maximize the potential of the network. Telecommunications networks need to efficiently route calls. These calls are generally requested at unknown times and have unknown bandwidth demands. The task of routing calls becomes increasingly difficult as the bandwidth demands of the calls begin to reach or exceed the bandwidth capacity of the network. Decisions must then be made as to which calls to route and which calls to ignore. In a centrally controlled network, call requests are received by a control center which then makes the decision as to whether or not to place the call as well as how to route the calls it decides to place. This process can be done at discrete intervals using the current state of the network and the current call request queue. The results can be implemented and the next state of the network and call queue evaluated. In a non-centrally controlled distributed network, each network node receives call requests and must make the placement and routing decisions.

This thesis considers only the centrally controlled case. The non-centrally controlled case has also been studied [2, 3]. For our case, the network state, as far as the amount of available bandwidth of each edge, as well as the set of call requests, is assumed to be known. The network is defined by its node and edge sets, with each edge having a limited bandwidth capacity and a cost per unit bandwidth used. Each call is defined by its start node, destination node, bandwidth requirement, and revenue associated with placing the call. When attempting to route the calls through the network, two related questions arise that the control center must address: How does one select a subset of calls to place? and once selected, How does one route those calls to maximize profit while satisfying the network bandwidth constraints? Besides a general integer programming approach [20, 21], two metaheuristic approaches have been reported with varied success: tabu search [16, 1] and genetic algorithm [8] techniques. This thesis introduces another metaheuristic approach based on ant colony optimization. While ant colony optimization is based on route finding problems, this thesis shows how the ant colony optimization paradigm can be used to find good solutions to the bandwidth packing problem.

The remainder of this thesis is organized as follows: chapter 2 introduces
the bandwidth packing problem analytically, giving two integer programming approaches to finding an optimal solution; chapter 3 introduces two metaheuristic approaches, one using genetic algorithms, and the other, tabu search; chapter 4 introduces ant colony optimization, describes an implementation as applied to the bandwidth packing problem, and gives empirical support for various parameters used to drive the algorithm.
2. The Bandwidth Packing Problem

In this chapter, the bandwidth packing problem (BWP) is formulated as an integer program (IP). The objective is to maximize total profit, subject to constraints that require calls to be routed using feasible paths and constraints that prevent the bandwidth capacities of each edge from being exceeded. BWP is defined by the following data:

\[ b_e = \text{the bandwidth capacity of edge } e; \]
\[ c_e = \text{the cost for each unit of bandwidth used on edge } e; \]
\[ v_i = \text{the revenue associated with placing call } i; \]
\[ r_i = \text{the bandwidth requirement for call } i; \]
\[ s_i = \text{the start node of call } i; \]
\[ t_i = \text{the terminal node of call } i; \]
\[ N = \text{the set of network nodes;} \]
\[ E = \text{the set of network edges; and} \]
\[ Q = \text{the set of calls.} \]

An IP formulation that does not rely on the paths being generated before hand has not, to our knowledge, been reported previously in the literature. A so called node formulation must include constraints that force the placement of calls using feasible routes. To construct these routing constraints, the notion of a call’s direction across an edge is needed. To accomplish this, we define two arcs for each edge in the network, one for each direction. We use \( e^+ \) and \( e^- \) to denote the two arcs associated with edge \( e \). The additional arc notation is:

\[ \mathcal{E} = \text{the set of all arcs } = \{e^+ \mid e \in E\} \cup \{e^- \mid e \in E\}; \]
\[ \mathcal{H}(j) = \text{the set of arcs that terminate at node } j; \text{ and} \]
\[ \mathcal{T}(j) = \text{the set of arcs that originate from node } j. \]

The decision variables under this formulation are:

\[ \bar{x}_{ia} = \begin{cases} 
1 & \text{if call } i \text{ uses arc } a \\
0 & \text{otherwise,} 
\end{cases} \quad (2.1) \]

\[ x_{ie} = \begin{cases} 
1 & \text{if call } i \text{ uses edge } e \\
0 & \text{otherwise,} 
\end{cases} \quad (2.2) \]
and
\[
y_i = \begin{cases} 
1 & \text{if call } i \text{ is routed} \\
0 & \text{otherwise.} 
\end{cases} \quad (2.3)
\]

The IP can now be formulated as follows:

\[
\text{maximize} \quad \sum_{i \in Q} y_i v_i - \sum_{i \in Q} \sum_{e \in E} x_{ie} r_i c_{ie}, \quad (2.4)
\]

subject to
\[
\sum_{a \in T(j)} \bar{x}_{ia} - \sum_{a \in H(j)} \bar{x}_{ia} = \begin{cases} 
y_i & j = s_i, \quad i \in Q, \\
y_i - t_i & j = t_i, \quad j \in N, \\
0 & j \neq s_i, t_i \end{cases} \quad (2.5)
\]

\[
\sum_{i \in Q} x_{ie} r_i \leq b_e, \quad e \in E, \quad (2.6)
\]

\[
\bar{x}_{ie+} + \bar{x}_{ie-} = x_{ie}, \quad i \in Q, \quad e \in E, \quad (2.7)
\]

\[
x_{ie}, \bar{x}_{ia}, y_i \in \{0, 1\}, \quad e \in E, \quad a \in \bar{E}, \quad i \in Q. \quad (2.8)
\]

To see how this formulation solves BWP, each constraint is examined using the network described in Figure 2.1 together with the call list described in Table 2.1. Table 2.2 illustrates the induced arcs.

![Simple Network](image)

**Figure 2.1: Simple Network**

The objective function (2.4) yields the profit for a given routing schedule: the revenue of all calls placed minus the cost for each edge used to place those calls. For our simple example, the objective function is the following:

\[
200 \times y_1 + 100 \times y_2 - (x_{11} \times 10 \times 2 + x_{12} \times 10 \times 1 + ... + x_{23} \times 20 \times 1). \quad (2.9)
\]
<table>
<thead>
<tr>
<th>Call</th>
<th>From</th>
<th>To</th>
<th>Demand</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>20</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2.1: Simple Call Table

<table>
<thead>
<tr>
<th>Arc</th>
<th>Start Node</th>
<th>Terminal Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2+</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2-</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3+</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3-</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2.2: Arc Definitions

The routing constraint (2.5) requires calls to be placed using a feasible path. This is done by creating a path from $s_i$ to $t_i$. The start node, $s_i$, for each placed call must have one more arc leaving it than entering it. Analogously, the destination node, $t_i$, for each placed call must have one more arc entering it than leaving it. All other nodes in the network must be balanced or, have an equal number of arcs entering and leaving them. In the context of our example, looking at call 1 and node 3, we have

$$ (\bar{x}_{12-} + \bar{x}_{13+}) - (\bar{x}_{12+} + \bar{x}_{13-}) = 0, $$

(2.10)

where $\mathcal{T}(3) = \{2^-, 3^+\}$ and $\mathcal{H}(3) = \{2^+, 3^-\}$. Thus, if call 1 uses arc $2^-$, it must also use one of the arcs $2^+$ or $3^-$. This constraint does not explicitly prevent the occurrence of multiple disjoint loops. These so-called sub-tours, however, would not affect our problem. If a sub-tour exists containing edges with positive costs associated with them, the objective prevents the sub-tour from being generated. If $c_e = 0$ for all edges in the sub-tour, only the bandwidth capacities would be affected. If another profitable call could be placed using this bandwidth, the objective causes the placement of the profitable call instead of generating the sub-tour. Thus, the only case in which sub-tours would be generated is when they have no effect on the objective value, in which case we can simply use the path starting from the start node and ending at the terminal node and ignore the sub-tours.
The network bandwidth constraint (2.6) prevents the network capacity from being exceeded. A network’s capacity is defined in terms of the bandwidth capacities for each of its edges. Each edge, \( e \), has a finite bandwidth capacity, \( b_e \), that limits the sum of the bandwidth of all calls using it. The sum of all calls using each edge must not exceed this bandwidth capacity.

Looking at edge 1, the sum of the bandwidth used by calls 1 and 2 must not exceed 20, or

\[
(\mathit{x}_{11} \times 20 + \mathit{x}_{21} \times 10) \leq 20. \tag{2.11}
\]

Thus, call 1 or 2 can be placed using edge 1, but not both.

Constraint (2.7) forces the variable associated with edge \( e \) to be connected to its associated arcs. Call 1 using arc \( 1^- \) would cause \( x_{11} \) to reflect that placement. The integrality constraint (2.8) prevents the partial placement and splitting of calls.

This formulation, while theoretically valid, has limited practical significance when it comes to actually solving an instance of BWP. This is due to the sheer size of the problem. For the simple example above, the routing constraint (2.5) yields \(|Q| \times |N| = 6\) equations; the network capacity constraint (2.6) yields \(|E| = 3\) inequalities; constraint (2.7) yields \(|Q| \times |E| = 6\) equations; and there are \(|Q| \times |E| + |Q| \times 2|E| + |Q| = 20\) variables, all of which are binary. Thus, even this trivial example requires 15 constraints and 20 variables to solve. As the size of the network and the number of calls in the call table increase, the number of constraints and the number of variables increase rapidly. In problem instances where \(|N|, |E|, \) and \(|Q|\) are not small, it is impractical to solve BWP directly. We look at two IP approaches to BWP that attempt to overcome the size of the problem.

The first, by Parker and Ryan [21], uses column generation and branch-and-bound techniques. They use a path formulation of the problem in their approach.

Two variables are used in this formulation:

\[
x_{ip} = \begin{cases} 
1 & \text{if call } i \text{ uses path } p \\
0 & \text{otherwise,}
\end{cases} \tag{2.12}
\]

and

\[
y_i = \begin{cases} 
1 & \text{if call } i \text{ is routed} \\
0 & \text{otherwise.}
\end{cases} \tag{2.13}
\]
Also, let $\delta_{ep}$ be 1 if edge $e$ is in path $p$ and 0 otherwise. With $P_i$ being the set of feasible paths for call $i$, BWP can be formulated with the following IP model:

\[
\text{maximize} \quad \sum_{i \in Q} v_i y_i - \sum_{e \in E} c_e \sum_{i \in Q, p \in P_i} \delta_{ep} r_i x_{ip} \\
\text{subject to} \quad \sum_{p \in P_i} x_{ip} = y_i \quad i \in Q, \quad (2.14) \\
\sum_{i \in Q, p \in P_i} \delta_{ep} r_i x_{ip} \leq b_e \quad e \in E, \quad (2.16) \\
x_{ip}, y_i \in \{0, 1\} \quad p \in P_i, i \in Q. \quad (2.17)
\]

In words, the IP is to maximize revenue minus cost while placing each call using at most one path and satisfying the bandwidth capacities for each edge in the network. This formulation differs from the first IP formulation in that it does not have a routing constraint. It relies on the set $P_i$ to be generated \textit{a priori}. This, in and of itself, is a difficult computational problem. Removing this process from the IP problem and handling it using an efficient route generating algorithm significantly reduces the size of the problem.

The number of variables $x_{ip}$ can be very large if the number of feasible paths for each call is large. The optimal solution, however, contains only one path for each call placed. Solving the problem considering only these paths, the same solution would be generated but the problem needed to be solved would be greatly reduced. Unfortunately, knowing which paths to omit is impossible until the solution is found. However, the paths that are very costly, or that contain a large number of edges, are less likely to be included in an optimal solution. Solving a problem related to the problem above, one that considers only paths that are likely to be included in the optimal solution, could be used to generate approximations to the original problem. This solution can be tested, using linear programming techniques, as to whether it solves the original problem. If it does not, an iterative process of adding paths and solving related sub-problems can be performed until it does. This is the motivation behind column generation. Column generation is a linear programming technique used to solve problems with a large number of columns (variables) that can be generated (added) as needed.

The first step in this approach to BWP is to solve the linear relaxation of the original IP, denoted LP, using column generation. This column generation implementation is done by solving a related sub-problem of LP, denoted LP',

7
by substituting $S_i$ for $P_i$ for each call $i$ where $S_i \subseteq P_i$. At the first iteration, $S_i = \emptyset$. Paths are added to each $S_i$ after each iteration as described below.

By linear programming duality theory, the solution to LP' solves LP if, and only if, the solution is dual feasible. After solving an instance of LP', feasibility is checked for each dual constraint. The dual constraints associated with $y_i$ are satisfied by the fact that these constraints are included in the dual of LP'. Thus, only the dual constraints associated with the paths in $P_i$ and not in $S_i$ need to be considered for each call $i$. Let $z_i$ and $w_e$ be the dual variables associated with the two sets of constraints in LP, respectively. The dual constraints are then

$$z_i + r_i \sum_{e \in E} \delta_{ep} w_e \leq r_i \sum_{e \in E} c_e, \quad i \in Q, p \in P_i. \quad (2.18)$$

This can be expressed equivalently as

$$\sum_{e \in E} \delta_{ep} (c_e - w_e) \geq z_i / r_i, \quad i \in Q, p \in P_i. \quad (2.19)$$

Feasibility needs to be verified for each dual constraint associated with each $x_{ip}$ not included in LP'. However, if the inequality is verified for the shortest path not in $S_i$, relative to the weights $(c_e - w_e)$, all other paths also satisfy this constraint. By assumption, $c_e \geq 0$, and by duality, $w_e < 0$, which yields $(c_e - w_e) > 0$, so Dijkstra’s algorithm can be used to find the shortest path. If a shortest path for some call $i$ does not satisfy the dual constraint, the path is then added to $S_i$ and LP' is resolved. When all dual constraints are satisfied, the solution to LP' solves LP and branch-and-bound can then be applied to generate a solution to the original IP.

In a typical branch-and-bound approach, a non-integer $x_{ip}$ would be chosen to branch on by solving the two problems associated with forcing $x_{ip} = 0$ and $x_{ip} = 1$, respectively. Forcing $x_{ip} = 1$ in this problem corresponds to placing call $i$ using path $p$. Equivalently, we can remove call $i$ from the call list and remove its associated bandwidth requirement from all edges contained in path $p$. This creates no problems and the column generation technique described above can be used to solve this new problem. Effecting $x_{ip} = 0$ by removing it from $S_i$, on the other hand, creates problems because of the column generation process used to solve LP. The removal of the column associated with path $p$ by removing it from $S_i$ would be counteracted by the column generation process immediately adding the column back. To avoid this, an alternate approach of
creating multiple branches at each step, one for each edge in path \( p \), is used. Each new branch redefines the problem by removing all paths in \( P_i \) that use the edge associated with the branch. This prevents the path from being added to \( S_i \) because it simply does not exist in \( P_i \) for the new problem.

To illustrate this IP approach, consider the problem defined by the network described by Figure 2.1 and the call table defined by Table 2.3.

<table>
<thead>
<tr>
<th>Call</th>
<th>From</th>
<th>To</th>
<th>Demand</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2.3: Call Table for IP

Our decision variables are the following:

\[ x_{11} \] refers to placing call 1 using edge 1;
\[ x_{12} \] refers to placing call 1 using edges 2 and 3;
\[ x_{21} \] refers to placing call 2 using edge 1;
\[ x_{31} \] refers to placing call 3 using edge 2;
\[ x_{32} \] refers to placing call 3 using edges 1 and 3;
\[ y_1 \] refers to call 1 being placed;
\[ y_2 \] refers to call 2 being placed; and
\[ y_3 \] refers to call 3 being placed.

At the first step, \( S_i = \emptyset \), and solving the linear relaxation yields

\[
\begin{align*}
y_1 &= 0 \\
y_2 &= 0 \\
y_3 &= 0.
\end{align*}
\]

(2.20)

(2.21)

(2.22)

Feasibility now needs to be checked for the dual constraints. This corresponds to verifying that the length of the shortest path, relative to the weights \((c_e - w_e)\), for each call, is less than the ratio \( z_i / r_i \). The edge weights are:
<table>
<thead>
<tr>
<th>Edge</th>
<th>$(c_e - w_e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 - 20 = -18</td>
</tr>
<tr>
<td>2</td>
<td>1 - 10 = -9</td>
</tr>
<tr>
<td>3</td>
<td>1 - 10 = -9</td>
</tr>
</tbody>
</table>

All of the edge weights are negative so the length of any path must also be negative. The ratio $z_i / r_i = 0$ for all $i$, so none of the dual constraints are satisfied. The variables corresponding to the shortest path for each call are $x_{12}$, $x_{21}$, and $x_{32}$. These paths are added to the appropriate $S_i$ and the new LP solved. Solving yields:

$$x_{12} = 1$$  
$$x_{21} = 1$$  
$$x_{32} = 0$$  
$$y_1 = 1$$  
$$y_2 = 1$$  
$$y_3 = 0$$

(2.23)  
(2.24)  
(2.25)  
(2.26)  
(2.27)  
(2.28)

There are two variables whose associated dual constraints need to be verified: $x_{11}$ and $x_{31}$. The new edge weights are:

<table>
<thead>
<tr>
<th>Edge</th>
<th>$(c_e - w_e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 - 0 = 2</td>
</tr>
<tr>
<td>2</td>
<td>1 - 0 = 1</td>
</tr>
<tr>
<td>3</td>
<td>1 - 0 = 1</td>
</tr>
</tbody>
</table>

The length of any path must be positive and $z_i = 0$ for all $i$. Thus, all paths satisfy the dual constraints and we have found the optimal solution to the linear relaxation. This solution is also integer so we have found the solution to BWP. If there had been a non-integer variable at this stage, say $x_{32} = 1/2$, we would branch on $x_{32} = 1 \lor x_{32} = 0$. The branch corresponding to $x_{32} = 1$ corresponds to placing call 3 using edges 1 and 3. This is done by removing call 3 from the call table and subtracting $c_3 = 5$ from edges 1 and 3. Two branches are implemented to effect $x_{32} = 0$, one for each edge in the path. The first of these branches would remove all paths for call 3 that use edge 1 from $R_3$. The second branch would remove all paths from $R_3$ that use edge 3. Again, it is important that the paths are removed from $R_3$ and not $S_3$ as the column generation phase would counteract the removal.
The second integer programming approach, taken by Park et al. [20], is similar to the approach described above in that it uses the same formulation and column generation technique, but it uses more complicated techniques to satisfy the integrality constraints. I briefly describe these techniques and provide references for a more detailed explanation.

The Park-Kang-Park algorithm begins by solving LP using the column generation technique described above. Generate minimal cover inequalities that are violated by the solution to LP. These minimal cover inequalities are valid inequalities that are added to LP to remove the incumbent fractional solution from the feasible region while retaining the integral solution to the original IP problem. For explicit details on how these minimal cover inequalities are generated, refer to Park et al. [20]. For a complete discussion on coverings, minimal cover inequalities and valid inequalities, refer to Nemhauser and Wolsey [18]. This new augmented linear problem is referred to as ALP. ALP is then solved using column generation. Note that new dual variables now exist for the added minimal cover constraints. When ALP is solved, new minimal cover inequalities may result and are added to ALP. This process of augmenting then solving ALP is continued until no minimal cover inequalities can be found. If this solution is integral, it also solves BWP. If not, branch-and-bound is implemented to satisfy the integrality constraints. If this integer solution yields the same objective value as the solution to ALP, the integer solution is optimal. If not, a branch-and-cut procedure is implemented either to prove the branch-and-bound result is optimal or to provide the optimal solution. This branch-and-cut routine is similar to branch-and-bound except that it uses strong cutting-planes and column generation to solve the subproblem associated with each node. Describing the cut used is beyond the scope of this thesis and I refer readers interested in seeing this cutting process to the original paper by Park et al. [20]. The problem of the column generation step adding the column removed by the cutting-plane step is handled by storing variables that keep track of whether or not the path associated with that column has been previously generated. If it has, the column associated with the shortest path that has not been previously generated is added. After this final step, the solution to BWP will have been found.
3. The Metaheuristic Approach

As a first approach in moving from IP techniques, which can produce optimal solutions, to approximation methods, which may not, the idea of generating lists in which to place the calls is a natural direction to take. A method that produces solutions in this manner is considered a order-based heuristic. One order-based heuristic is to order the calls based on the ratio of each call’s revenue to its bandwidth requirement and sequentially attempt to place each call in the list. This would tend to place the calls with high revenues relative to the cost of their placement.

After a list is generated, a method to choose the routing for each call must be constructed. This routing heuristic could range from considering only the shortest path to using a order-based heuristic to order the feasible paths for each call. The more elaborate the heuristic, the more costly its implementation, with the hope that better solutions are generated. The greedy order-based heuristic used for calls and the shortest-path-only routing heuristic generates a solution with very little effort but generally with low profit as well. For any heuristic to be capable of generating the optimal solution for any problem instance, it must be capable of generating all possible call orderings as well as allowing for each call to be placed using any of its feasible paths. Heuristics that neglect certain paths by considering only a subset of a calls feasible paths run the risk of eliminating the optimal solution from its searchable region.

Considering again the problem described by Figure 2.1 and Table 2.1. An optimal solution is achieved by placing call 1 using edges 2 and 3, and call 2 using edge 1, with a profit of 240. Generating a call ordering using the revenue to bandwidth requirement ratio described above yields the ordering (1, 2). This ordering, along with the shortest-path-only path heuristic, generates a solution to place call 1 using edge 1. After call 1 has been placed, call 2 cannot be placed due to its bandwidth requirements and our profit is 180. No ordering can produce the optimal solution using the shortest-path-only heuristic. No heuristic that excludes the routing of call 1 using edges 2 and 3 can find the optimal solution. Although it is conjectured that the ratio of the best solution generated by a shortest path permutation to the optimal solution goes to one with probability one [8], problems of any size can be generated where the best
permutation schedule performs arbitrarily poorly when only the shortest paths are considered. Approximation methods that have been applied to BWP are now considered.

The bandwidth packing problem is a member of a class of problems that are considered \textit{NP-complete}. The term NP-complete comes from computational complexity theory, which was invented by Edmonds with the first major theorem by Cook [6]. This theory categorizes problems in terms of the complexity of algorithms used to solve them. It is conjectured that any algorithm designed to solve an NP-complete problem, including BWP, requires a number of computational steps that grows exponentially with the size of the problem. Increasing the number of calls in the call list or increasing the size of the network to support the calls, increases both the number of variables and the number of constraints in either IP formulation of BWP. Thus, both approaches described above, as with all algorithms designed to find the optimal solution to BWP, require an exponential amount of time, in terms of the problem size, to guarantee that an optimal solution is found. In cases where the optimal solution is not needed, and a trade-off between computation time and quality of solution can be made, metaheuristics can be employed. Metaheuristics comprise a class of approximation methods that apply to combinatorial optimization problems, such as BWP, when exact algorithms are impractical.

\begin{definition} [19] A \textit{metaheuristic} is an iterative generation process that guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search spaces using learning strategies to structure information in order to find efficiently near-optimal solutions.
\end{definition}

Two metaheuristic approaches to BWP are reported in the literature. The next two sections present a brief introduction to each metaheuristic, followed by a description of the implementation as applied to BWP.

\subsection{3.1 Genetic Algorithms}

A Genetic Algorithm (GA) is a metaheuristic that mimics the observed ability of life to adapt successfully to its environment through generational evolution. John Holland, the field's inventor, took this observed ability and turned it into the basis for this highly successful metaheuristic. Life, through its reproductive mechanisms, is able to change over generations to adapt to its environment.
While this process is not fully understood, some features that are generally accepted and form the basis for GA are as follows:

- Living beings are based on encoded *chromosomes*. Evolution takes place on these encoded structures.

- Natural selection is the process by which the chromosomes that develop into successful members of the population pass on the traits that made them successful while less successful member are unable to pass on their traits.

- Reproduction is the mechanism through which natural selection operates. Children possess chromosomes based on the recombination or crossover of the chromosomes of their parents. Mutation can occur where pieces on the child’s chromosome are generated randomly instead of derived from its parents.

- Evolution has no explicit memory of past generations.

At the most basic level, GA takes a population of chromosomes and evolves them through a process of natural selection. Each chromosome represents an encrypted solution to the problem at hand with a certain *fitness*. This fitness is used by the algorithm to determine the probability that a chromosome will reproduce, and consequently pass on the traits that contributed to its high level of fitness. The hope is that a relatively nondescript initial population evolves into a population of encrypted solutions that yield competitive solutions to the problem being solved.

Each GA is defined by its construction of chromosomes, a metric for evaluating a chromosome’s fitness, and the means by which chromosomes reproduce to form subsequent populations. There are no hard and fast rules for defining these characteristics, and two different definitions could yield two drastically different results. While many facets of GA exist, this section describes only the mechanisms used to solve the bandwidth packing problem as described in [8].

The chromosome used in this application of GA is simply an ordering of the calls. This ordering is used to generate solutions to BWP by attempting to place each call sequentially in the call list using the shortest feasible path for that call. In this context, the shortest path is the path that is the least costly
1. Sum the fitnesses of all population members, call it \( F \).

2. Generate a uniform random number, \( n \), between 0 and \( F \).

3. The first population member whose fitness, added to the fitnesses of all preceding population members, exceeds \( n \) is used to parent the next generation.

Figure 3.1: Roulette Wheel Parent Selection

in terms of the sum of the costs on its edges. If there is enough bandwidth to support the placement of the call, it is placed, if not, the next call in the list is attempted until the end of the list is reached.

A chromosome’s fitness is used to determine whether or not it is used to parent the next generation. This GA implementation assigns fitnesses to each chromosome in the following way. For a given parameter \( \pi \), a list of fitness values is generated of the form \((1000, 1000\pi, 1000\pi^2, \ldots)\). These values are assigned to the schedules so that the schedule that generates the greatest profit receives the greatest fitness and the schedule associated with the least profit is assigned the least fitness. These fitness values determine the probability that a schedule is used to generate offspring in the next generation by means of roulette wheel parent selection, which is described in Figure 3.1. As an example of this selection process, consider a population of 10 members with \( \pi = .9 \). The sum of the fitness values is approximately 6,513. If the number 2,319 is randomly generated, the third most fit chromosome would be chosen (\(1000 + 1000 \times .9 + 1000 \times .9^2 = 2710 > 2319\)).

Once the parents have been chosen, the children are generated by means of one of the two reproduction mechanisms used in this GA. The two reproduction mechanisms are uniform order-based crossover and scramble sublist mutation, both of which are described shortly. To decide which mechanism is applied, each is assigned a fitness as a parameter. This fitness is used in the same way a chromosome’s fitness is used in the parent selection phase. A roulette wheel selection scheme is utilized to determine the mechanism to use. The fitness values can be fixed over the course of the run or can be adjusted dynamically. This implementation employs a dynamic scheme.
1. Generate a bit string, at random, that is the same length as the two parents. For BWP this is the number of calls in the list.

2. For each position in the bit string containing a 1, the first child is identical to the first parent.

3. For each position in the bit string containing a 0, permute the calls to the order the calls appear in parent two.

4. Repeat steps 2 and 3 to generate child two, reversing the roles of the parents.

Figure 3.2: Uniform Order-Based Crossover

Uniform order-based crossover is designed to create children with traits similar to their parent’s traits. The idea is that traits that tend to create better solutions are passed on to future generations while maintaining enough diversity that each generation also differs from its predecessors. Uniform order-based crossover achieves this as described in Figure 3.2

As an example, consider the two parents:

Parent 1 1 2 3 4 5 6 7 8 9 10
Parent 2 10 8 6 4 2 9 7 5 3 1

Suppose, at step 1, the following bit string is generated:

Bit String 1 0 0 0 1 1 0 1 1 0

Then, step 2, for both children, produce

Child 1 1 _ _ _ 5 6 _ 8 9 _
Child 2 10 _ _ _ 2 9 _ 5 3 _

Step 3, for both children, produce

Child 1 1 10 4 2 5 6 7 8 9 3
Child 2 10 1 4 6 2 9 7 5 3 8

The order in which calls are placed is crucial to the quality of a solution to BWP. Uniform order-based crossover attempts to capture pieces of the
1. Pick two elements of the parent chromosome.
2. Randomly permute all of the calls between the elements chosen.

Figure 3.3: Scramble Sublist Mutation

ordering from both parents to create their children.

Scramble sublist mutation is designed to introduce an element of randomness into the population. This may help prevent the algorithm from converging to suboptimal solutions by generating populations that would be unlikely to be generated by the order-based crossover mechanism. While uniform order-based crossover attempts to retain characteristics of previous generations, scramble sublist mutation is designed to disrupt this process. This is done as described in Figure 3.3.

Again, because order is essential to the quality of a solution to BWP, scramble sublist mutation is designed to disrupt the ordering contained by the chromosome. This may help to search new areas of the search space and prevent the convergence to a suboptimal solution.

3.2 Tabu Search

Tabu Search (TS) is a metaheuristic that uses adaptive memory to guide its underlying heuristic. This adaptive memory is used to encourage the heuristic to search in some areas of the solution space while discouraging the heuristic's search in others. The goal is to spend more effort searching promising areas, while not excluding seemingly unpromising areas altogether. A brief introduction and explanation of the features utilized by two TS approaches to BWP follows. For a complete description of TS refer to [14] by Glover and Laguna, or to [12] and [13] by Glover.

Tabu search is a neighborhood search heuristic. The idea behind any neighborhood search heuristic is to iterate from solution to solution considering only solutions that are neighbors of the current solution at each iteration. The metric which determines whether or not two solutions are neighbors is the basis for any neighborhood search. A neighborhood for BWP could be as follows: given a call list, $x$, a neighborhood of $x$, denoted $\mathcal{N}(x)$, could be the set of
1. Generate a permutation of the calls, call it \( x \).

2. Calculate the profit of \( x \) by attempting to place each call using its shortest path. If the network can support the call's bandwidth requirement, place the call, if not, move to the next call in the list.

3. Build \( \mathcal{N}(x) \) by generating all solutions created by switching the first call in \( x \) with all other calls in the list.

4. Compute the profit associated with each element of \( \mathcal{N}(x) \). Set \( x \) to be the element with the greatest profit.

5. Repeat until 10 successive iterations have not resulted in an improvement over all of its predecessors.

Figure 3.4: Simple Neighborhood Search

solutions constructed from \( x \) by switching the first call in \( x \) with any other call in the list. Defining \( \mathcal{N}(x) \) in this manner yields a subset of the solution space, at any given solution, of size \(|Q| - 1\). This greatly reduces the number of solutions that the heuristic must evaluate before it makes its next move. This definition also has the property that every permutation can be reached by any other permutation given enough neighborhoods through which to transition. For BWP, a simple neighborhood search heuristic could be defined as described in Figure 3.4.

Notice that there is nothing to preclude this heuristic from visiting the same solutions over and over again. Thus, there is motivation to diversify the solutions in some way by forcing them into new regions of the solution space. To see how this might be achieved, consider Figure 3.5 and the calls in Table 3.1.

If we let \( x = (1, 2, 3, 4) \), \( \mathcal{N}(x) \) becomes \{\( (2,1,3,4), (3,2,1,4), (4,2,3,1) \}\}. Every element in \( \mathcal{N}(x) \) yields a profit of 210, the best of the set. Our heuristic might well choose \( (2, 1, 3, 4) \) as its next move. \( \mathcal{N}(x) \) now becomes \{\( (1,2,3,4), (3,1,2,4), (4,1,3,2) \}\}. Nothing would prevent our heuristic from simply choosing its next move to be \( (1, 2, 3, 4) \) and we would be back where we started. If our heuristic could remember that \( (1, 2, 3, 4) \) had already been visited, it might
choose \((3, 1, 2, 4)\) as its next move. This would cause the optimal solution of \((4, 1, 2, 3)\) to show up in the next \(\mathcal{N}(x)\). This method of remembering past solutions is called explicit memory. Explicit memory can create immense demands on storage because entire solutions must be remembered. A more efficient scheme is to remember defining characteristics of visited solutions. This is called attributive memory. For example, if we had simply remembered that the calls in the first and second positions had been swapped, and that the swap should not be repeated, the same result of moving to a new neighbor would have been achieved. TS keeps a list, known as the tabu list, to prevent areas of the solution space that have recently been visited from being revisited. By using attributive memory instead of explicit, some solutions that have not been visited also end up as tabu. If these are optimal, or would lead to an optimal solution, our memory has backfired on us. TS provides two mechanisms to overcome this.

The first of these two mechanisms is tabu tenure. Each element of the tabu list is resident for only some specified number of moves, its tabu tenure. When this number of moves has been performed, the element is removed from the tabu list and the heuristic is free to make those moves again. This allows TS to visit regions of the solution space more than once. While the solutions in the region are the same, the structure of TS may be different with each visit.
because its tabu list may be different. This may cause different solutions to be generated each time a region is visited.

The second mechanism used by TS is the use of aspiration criteria. This simply provides TS the ability to override the tabu status of a move if it meets some specified criteria. This criteria can be anything from achieving the best solution so far to making large impacts on the TS structure. The only aspiration criteria used by the two TS implementations below is achieving the best solution so far.

There are many facets to TS not included here. I merely introduce the aspects that aid in the understanding of the TS implementations described in the following two sections.

3.2.1 Tabu Search I

The approach taken by Anderson et al. [1] uses a permutation of calls to represent each solution. To evaluate a given solution, the first call in the list is placed using its shortest feasible path. The available bandwidth is updated for each edge used to place the call by subtracting the bandwidth requirement for the call. An attempt to place each subsequent call is made using the shortest feasible path. This TS implementation considers only instances of BWP where the cost of using bandwidth on each edge is negligible, that is, $c_e = 0$ for all $e \in E$. Under this constraint, the notion of the shortest feasible path is simply the path with the fewest edges instead of the path with the lowest cost.

There are three types of moves defined for this implementation. Each has its own notion of a neighborhood. The first type of move is a partial 2-opt, done by considering all the permutations generated by swapping the first call with every other call. When a move of this type is made, it becomes tabu to repeat the swap until its tabu tenure has expired.

When 10 moves are performed in succession without an improvement in objective value, a cut move is performed. This is done by defining $\mathcal{N}(x)$ to be the lists formed by placing the calls 1 through $i$ after the calls $i + 1$ through $|Q|$. A move of this type erases the tabu list.

If no cut move produces an improvement in objective value, a full 2-opt is performed. Notice that a swap of position $i$ with $j$ is equivalent to swapping $i$ with 1 then 1 with $j$. Thus, we can update the tabu list as was described above for the partial 2-opt move.
3.2.2 Tabu Search II

The second TS implementation, by Laguna and Glover [16], is a more complex implementation in that it uses dynamic tabu tenure values and two types of memory: long-term and short-term. These new features are described below.

This implementation starts by generating a specified number, denoted $k_i$, of shortest feasible paths for each call $i$. A parameter, $K$, is used as an upper bound for this number. The length of a path $p$, denoted $l_p$, is defined

$$l_p = \sum_{e \in E_p} c_e,$$  \hspace{1cm} (3.1)

where $E_p$ is the set of edges in path $p$. The calls are then ordered in decreasing order of the ratio of their revenue to their bandwidth demand ($v_i/r_i$). An attempt to place each call is made starting with the first call and its shortest path down to the last call and its longest path. This constitutes the first solution. A move, $(i, h)$, indicates call $i$ moved from its current path, $g$, to path $h$. Once a move is made, TS prevents call $i$ from being assigned to path $g$ in the near future by placing move $(i, g)$ on the tabu list. Each call’s path list is augmented with a null path, used to indicate that a call is not placed.

At each iteration, all non-tabu moves are considered and the move with the greatest value chosen. The value of a move is a function of local network information and the contents of its long term memory structures. The local information component of a move’s value is,

$$V(i, h) = r_i (l_g - l_h).$$  \hspace{1cm} (3.2)

Looking closer at $V(i, h)$, notice that it does not consider revenue, $v_i$, as a part of the move’s value. It considers only the bandwidth costs associated with the feasible paths. The null path, however, has no cost and therefore always have the greatest local component. When $V(i, h)$ is employed, it encourages the use of the null path for all placed calls. The only way for a call to be placed is if the null path is contained on the tabu list and thus cannot be selected. This would seem to discourage the placement of calls and is seems contrary to the objective of maximizing profit, which can be generated only by the placement of calls. It is not clear to the author how this method would yield positive results without handling the local component of the null path’s move value as a special case. Using $V(i, null) = 0$ would seem an intuitive
value as \( V(i, h) \) is a measure of how a call's bandwidth requirement affects the state of the network. Not placing a call has no effect on the network capacities so 0 would be a logical choice.

While the short-term memory is affected by the recency of a move, the long-term memory is affected by the frequency of a move. Each move has an associated value, \( \omega(i, h) \), that keeps track of how many times that move has been made. If a move has been made many times over the course of the run, it is penalized to create diversity. Combining \( V(i, h) \) and \( \omega(i, h) \) gives us a move's penalized move value, denoted \( \phi(i, h) \), which is how TS selects the next move to make. \( \phi(i, h) \) is defined:

\[
\phi(i, h) = \begin{cases} 
V(i, h) & \text{if } V(i, h) > 0; \\
V(i, h) - \theta \times \omega(i, h) & \text{otherwise}, 
\end{cases}
\]

where \( \theta \) is a parameter.

A move's tabu tenure in this implementation is dynamic, varying among three values: short, medium, and long. The values used for each setting are \( k_i \), \( 1.5k_i \), and \( 2k_i \). This requires that each member of the tabu list keep track of the number of iterations required before it can be removed from the tabu list. TS iterates through the three values by the sequence \( \{k_i, 1.5k_i, k_i, 2k_i, 1.5k_i, 2k_i\} \). A change in tabu tenure is done at the number of moves equal to twice the value of the current tabu tenure.

Consider the simple example described by Figure 2.1 and Table 2.1. For this simple example, we have 3 paths for call 1 and 2 paths for call 2. Use \( p_{10} \) to denote the null path for call 1, \( p_{11} \) to denote the route using edge 1 for call 1, \( p_{12} \) to denote the route using edges 2 and 3 for call 1, \( p_{20} \) to denote the null path for call 2, and \( p_{21} \) to denote the route using edge 1 for call 2. A solution to this problem consists of a list of length 2, consisting of the routing information for each call.

Consider the solution \( (p_{11}, p_{20}) \). This represents the solution of placing call 1 using path \( p_{11} \) and not placing call 2. From this solution, there are 2 moves that could be made. Move \( (1, p_{12}) \), which switches the routing of call 1 to \( p_{12} \), and move \( (1, p_{10}) \), which results in call 1 not being placed. Call 2 has no feasible moves at this stage. The value of each move is as follows:

\[
\phi(1, p_{10}) = 10 \times (2 - 0) = 2, \quad (3.4) \\
\phi(1, p_{12}) = 10 \times (2 - 2) = 0. \quad (3.5)
\]
Choosing the greater of the two values, move \((1, p_{10})\) is made. After each move is made the tabu list and tenure values must be updated. Only one move has been made at this stage so neither of the calls tabu tenure values need to be changed. Move \((1, p_{11})\) is placed on the tabu list for 2 moves.

We now have two non-tabu moves to evaluate: \((1, p_{12})\) and \((2, p_{21})\). The penalized move values for each of these moves are

\[
\begin{align*}
\phi(1, p_{12}) &= 10 \times (2 - 2) = 0, \\
\phi(2, p_{21}) &= 20 \times (0 - 2) - 0 = -40.
\end{align*}
\]

Move \((1, p_{12})\) has the greatest penalized move value, and the tabu move \((1, p_{11})\) does not meet the aspiration criteria, so move \((1, p_{12})\) is made. We have now reached 2 times the tabu tenure for call 2 which requires it to be changed to \(1.5 \times k_2 = 1.5\). Non-integer values, however, are not allowed. We keep the tabu tenure at 1 but will wait \(2 \times 1.5 = 3\) moves before changing it to \(2 \times k_2 = 2\). Move \((1, p_{10})\) is placed on the tabu list for 2 moves. The state of the tabu list is now

<table>
<thead>
<tr>
<th>Tabu move</th>
<th>Moves remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1, p_{11}))</td>
<td>1</td>
</tr>
<tr>
<td>((1, p_{10}))</td>
<td>2</td>
</tr>
</tbody>
</table>

There is only one non-tabu move available at this stage: \((2, p_{21})\). By default, its penalized move value of -40 is the best move value and is made. The current solution \((p_{12}, p_{21})\) is the optimal solution for this problem. TS has no way of determining this however, and will continue to search for better solutions until some termination criteria is reached.

Both TS implementations for BWP report to generate competitive solutions in reasonable amounts of time. The second implementation contributes support for the use of long term memory in addition to short term memory, as well as the use of a frequency function.
4. Ant Colony Optimization

Ant Colony Optimization (ACO) is a recent addition to metaheuristics. It has been applied successfully to many combinatorially difficult problems, including: the traveling salesman problem [9], the sequential ordering problem [10], the quadratic assignment problem [17], the vehicle routing problem [4], scheduling problems [5], and graph coloring [7]. Ant Colony Optimization is motivated by the observed ability of ants to find the shortest path from their nest to a food source. This is done by means of laying a scent, called pheromone, as they walk. Once a path is established, ants can follow this pheromone trail to and from the food source. The establishment of the shortest path is described in Figure 4.1, taken from [9].

![Figure 4.1: Ants Finding The Shortest Path](image)

In a), the ants have an established path between points A and E. b) depicts an obstacle being placed in the path. The ants are now forced to choose a direction. The ants, having no information as to which direction is better, choose left or right with equal probability. The ants that choose right arrive at point E considerably before the ants that choose left. As a consequence, the ants that chose right arrive back at point B before those that chose left.
This begins to bias the amount of pheromone on the shorter of the two paths, increasing the chances that future ants choose the shorter path. As time progresses, the bias increases until nearly all of the ants choose the shorter path. It is important to note that the pheromone level increases the likelihood of ants choosing a particular path but does not necessarily imply that they will. There are still occasional ants that stray from the established path. This has an important implication on the paradigm; as solutions are found, we want to keep searching for new paths to keep from converging to suboptimal solutions.

This paradigm can be applied to any problem where solutions can be obtained by generating a decision path. Virtual ants begin at some starting state and make decisions, creating a solution path until some terminal state is reached. The ants lay pheromone, proportional to the merit of the solution found, on the decisions they used to encourage future ants to make similar decisions. Pheromone is not our virtual ant’s only factor in making their decisions. A fitness value for each decision is also incorporated based on local information in the network. At any given state where a decision must be made, the ant generates probabilities for each decision based on the pheromone and fitness of that decision. The problem instance dictates the definition of pheromone and fitness.

The BWP decision path must consist of selecting which calls to place and which routes to use when placing the selected calls. This yields two levels of decisions to which ACO can be applied. The first level, selecting calls, is done by generating an ordering in which to attempt to place the calls. The second level, selecting the routing, is done by generating an ordering of feasible paths for the call being placed. Each ant generates a call ordering and a route ordering for each call. For a given network state, this ant decision path generates a unique solution. An outline of the algorithm is as described in Figure 4.2

The algorithm’s first step, finding $k_i$ shortest paths, attempts to find $K$ feasible routes for call $i$ where $K$ is a parameter. $K$ feasible paths may not exist for each call $i$ so $k_i$ is used to designate the number of paths actually found with an upper bound of $K$. This route finding stage can be done using any route finding algorithm. The implementation used to generate the data in §4.1 uses a breadth first search. As a side note, the routes are stored in a linked list that grows as routes are generated then shrinks as they are completed.

For dense networks the number of routes for a given call can be large; up
Outline of ACO as applied to BWP

**Step 1.** Find $k_i$ feasible routings.

**Step 2.** Generate a call ordering for each ant.

**Step 3.** Generate a path ordering for each call in each ants call list.

**Step 4.** Attempt to place each call on each ant's call list with each path in its path list.

**Step 5.** Save the best ant's call and path lists.

**Step 6.** Update pheromone levels on all calls and paths.

**Step 7.** Repeat Steps 1 - 6 until some stop criteria has been reached.

Figure 4.2: Ant Colony Algorithm

to $(n - 2)!$ for complete networks. Thus, there are potentially many routes to store, requiring a great deal of memory. For this reason there is a cutoff at 30,000 entries in the linked list. If this number is reached, the current set of feasible routings is used. For large problems, a different route finding algorithm would need to be used. The test problems used for comparisons done in §4.1 are of the order $n \in (14, 192)$, $|Q| \in (20, 93)$, and $|E| \in (16, 210)$. With these problems, generating 30,000 unfinished paths rarely occurs.

Ants generate the orderings of the calls and routings using a probability function. This function depends on pheromone and fitness values and must be delineated in the context of BWP. Pheromone, generally speaking, gives an ant information about how a decision has performed in past solutions. Decisions that lead to good solutions should accumulate more pheromone than those that produce poor solutions, thus biasing future ants decisions toward decisions that yield good solutions. We must also prevent the pheromone level from becoming too high, even on good decisions, to encourage ants to explore new areas of the solution space. This corresponds to evaporation of the pheromone trail.

Let $\tau(d)$ denote the pheromone level on decision $d$. After each iteration,
\( \tau(d) \) is updated by adding \( \Delta \tau(d) \) to \( (1 - \rho) \times \tau(d) \), where \( \rho \) is a parameter corresponding to pheromone evaporation. \( \Delta \tau(d) \) is based on the merit of the solution found by the ants that made decision \( d \). Note that \( d \) can be a decision of selecting a specific call or a decision of selecting a specific routing for a call. To distinguish between the two, let \( \tau_{\text{call}}(i) \) denote the pheromone accumulated on call \( i \) and \( \tau_{\text{path}}(i, p) \) denote the pheromone accumulated on path \( p \) for call \( i \). We use \( \tau \) with no subscript when referring to both.

Let \( A_{ip} \) be the set of ants that placed call \( i \) using path \( p \) and \( A_i = \bigcup_{p \in P} A_{ip} \). \( A_i \) corresponds to the set of ants that placed call \( i \) using any path. The added pheromone after each iteration becomes,

\[
\Delta \tau_{\text{call}}(i) = \sum_{a \in A_i} \frac{1}{z^* - z_a + 1} \tag{4.1}
\]

for calls, and

\[
\Delta \tau_{\text{path}}(i, p) = \sum_{a \in A_{ip}} \frac{1}{z^* - z_a + 1} \tag{4.2}
\]

for routes, where \( z^* \) is the best profit so far and \( z_a \) is the profit generated by ant \( a \). Notice that \( \Delta \tau \) is always positive and less than or equal to the number of ants. This keeps \( \tau \) from becoming too large and dominating the decision process. The constant 1 is added to the denominator to prevent the possibility of dividing by 0.

A decision’s fitness is designed to encourage ants to make decisions that are likely to generate better solutions. For example, a call with a high revenue and a low bandwidth requirement is more likely to be included in a good solution than a call with a low revenue and a high bandwidth requirement. Using this greedy bias on the ant’s decision greatly reduces the time it takes for ants to find good solutions. A decision’s fitness is denoted by \( \eta(d) \) with the distinction between calls and paths handled in the same manner as was handled in the notation of \( \tau(d) \). This heuristic reasoning motivates the fitness for call \( i \) to be defined as follows;

\[
\eta_{\text{call}}(i) = \frac{(v_i/r_i) - \min_{i \in Q}(v_i/r_i) + 1}{\max_{i \in Q}(v_i/r_i) - \min_{i \in Q}(v_i/r_i) + 2} \tag{4.3}
\]

The constants, 1 and 2, are arbitrary and are simply used to satisfy \( 0 < \eta_{\text{call}}(i) < 1 \).
Paths also need fitness values. Paths that cost less to use tend to generate more profit and should therefore be biased to be included in the solution. There are problem instances, however, when the cost of using an edge is negligible. In this case, paths with fewer edges should be biased as they leave more room for other calls to be placed. Both cases are defined as follows:

\[
\eta_{\text{path}}(p) = \begin{cases} 
1/|E_p| & \text{when neglecting edge costs,} \\
1/\sum_{e \in E_p} c_e & \text{otherwise.}
\end{cases} \tag{4.4}
\]

We assume \(c_e > 0\), so \(0 < \eta_{\text{path}}(p) \leq 1\). This prevents \(\eta_{\text{path}}(p)\) from being too large and dominating the decision process.

With the above definitions it is now possible to construct the probability functions the ants use to generate their call and path lists. We denote ant \(a\)'s call list with \(C_a\) and its path list for call \(i\) with \(L_{ai}\). \(\Psi(a, d)\) is used to denote the probability of ant \(a\) making decision \(d\), with the distinction between calls and paths made as above. Define \(\Psi_{\text{call}}(a, i)\) to be the probability of ant \(a\) choosing call \(i\) to be the next call placed on \(C_a\) as,

\[
\Psi_{\text{call}}(a, i) = \begin{cases} 
\frac{\tau_{\text{call}}(i)^\alpha \eta_{\text{call}}(i)^\beta}{\sum_{u \in Q \setminus i \notin C_a} \tau_{\text{call}}(u)^\alpha \eta_{\text{call}}(u)^\beta}, & \text{for } i \notin C_a \\
0, & \text{otherwise,}
\end{cases} \tag{4.5}
\]

where \(\alpha\) and \(\beta\) are positive parameters.

At iteration one, no ants will have had the opportunity to leave pheromone on any of the calls. To prevent division by zero, \(\tau_{\text{call}}(i)\) is initialized to 0.0001 for each call \(i\). Notice that \(C_a\) changes after each call is added to it. This in turn affects \(\Psi_{\text{call}}(a, i)\), so the probabilities must be regenerated at each selection. After the probabilities have been generated, a call is selected based on these probabilities and added to \(C_a\). This process is repeated until \(C_a\) contains all of the calls.

The probability function that the ants use to choose paths is defined in a similar manner. For each call in the ant’s call list, an associated path list is generated. The probability that ant \(a\) chooses path \(p\) to be the next path in \(L_{ai}\) is
\[
\Psi_{\text{path}}(a, i, p) = \begin{cases} 
\frac{\tau_{\text{path}}(i, p)^{\alpha} \eta_{\text{path}}(p)^{\beta}}{\sum_{w \in P_i \mid w \notin L_{ai}} \tau_{\text{path}}(i, w)^{\alpha} \eta_{\text{path}}(w)^{\beta}} & \text{for } p \notin L_{ai}, \\
0 & \text{otherwise.}
\end{cases}
\] (4.6)

Again we initialize \(\tau_{\text{path}}(i, p)\) to 0.0001 for each path.

Metaheuristics are designed to find high quality solutions in a reasonable amount of time. In an attempt to improve the rate at which good solutions are found, two factors are commonly incorporated.

The first incorporates forcing ants to make their decisions based solely on fitness levels. If all ants make their decisions greedily at every state, the algorithm generates poor solutions, and in all but contrived problem instances, never reach an optimum. If ants make greedy decisions probabilistically, the solutions tend to be higher in profit at all levels of effort. This factor is based on a parameter, \(q_0 \in (0, 1)\). Before an ant makes a decision, a uniform random number between 0 and 1 is generated. If the number is less than \(q_0\), the ant makes a greedy decision, otherwise the above definition of \(\Psi(a, d)\) is used.

The second factor involves adding elite ants, \(\sigma\) of them, to the problem. Elite ants simulate ants using the best known solution. This creates extra pheromone on those decisions used to generate the best solution and therefore encourages future ants to stay closer to them. Since good solutions tend to include similar decisions, this positively affects both the objective value and the rate at which the objective value improves. Computationally, elite ants are very inexpensive because they do not cost anything in the solution generation process. They simply affect \(\Delta \tau\) in the pheromone update stage.

To demonstrate ACO, we walk through one iteration using Figure 2.1 and Table 2.1. The following parameter settings are used: \(q_0 = 0.05\), \(\alpha = 0.25\), \(\beta = 5\), \(\sigma = 2\), \(\rho = 0.3\), and \(K = 15\). We also use only 1 ant.

**Step 1.** Find \(k_i\) feasible paths.

This step attempts to generate \(K\) feasible paths for each call. For our simple example, we have 2 paths for call 1 and 1 path for call 2. Use \(p_{11}\) to denote the route using edge 1 for call 1, \(p_{12}\) to denote the route using edges 2 and 3 for call 1, and \(p_{21}\) to denote the route using edge 1 for call 2.

**Step 2.** Generate a call ordering for each ant.
The first step in generating $C_a$ is to calculate each call’s fitness and pheromone levels. These are used to calculate the probabilities and subsequently, generate the orderings. The fitness values for each of our calls are

$$\eta_{call}(1) = \frac{200/10 - 5 + 1}{20 - 5 + 2} = 16/17 \quad (4.7)$$

and

$$\eta_{call}(2) = \frac{100/20 - 5 + 1}{20 - 5 + 2} = 1/17. \quad (4.8)$$

Since this is the first iteration, $\tau_{call}(1)$ and $\tau_{call}(2)$ are both at their initial values of 0.0001. We can now calculate the probabilities for both calls.

$$\Psi_{call}(a, 1) = \frac{0.0001^{0.25} \times 0.941^5}{0.0001^{0.25} \times 0.941^5 + 0.0001^{0.25} \times 0.059^5} = 99.9999\% \quad (4.9)$$

$$\Psi_{call}(a, 2) = \frac{0.0001^{0.25} \times 0.059^5}{0.0001^{0.25} \times 0.941^5 + 0.0001^{0.25} \times 0.059^5} = 0.0001\% \quad (4.10)$$

Once the probabilities have been calculated, a random number between 0 and 1 is generated to determine if a simple greedy decision is made. With $q_0 = 0.05$, a number between 0 and 0.05 would place call 1 on $C_a$. A number between 0.05 and 1 would require the use of $\Psi_{call}(a, 1)$ and $\Psi_{call}(a, 2)$. This would most likely result in the placement of call 1 on $C_a$. Suppose that call 1 is chosen as the first call on $C_a$. This leaves only call 2 to be placed and it is added to $C_a$ with probability 1.

**Step 3.** Generate a path ordering for each call in each ant’s call list.

Before an ordering can be generated, each path’s fitness and pheromone levels need to be calculated. The fitness values for each of our paths are

$$\eta_{path}(P_{11}) = 1/2, \quad (4.11)$$

$$\eta_{path}(P_{12}) = 1/(1 + 1) = 1/2, \quad (4.12)$$

and

$$\eta_{path}(P_{21}) = 1/2. \quad (4.13)$$

Again, this is the first iteration and the pheromone levels are at their initial values of 0.0001. The path probabilities the ant uses are

$$\Psi_{path}(a, 1, P_{11}) = \frac{0.0001^{0.25} \times 0.5^5}{0.0001^{0.25} \times 0.5^5 + 0.0001^{0.25} \times 0.5^5} = 50 \quad (4.14)$$

30
\[ \Psi_{\text{path}}(a, 1, p_{12}) = \frac{0.0001^{0.25} \times 0.5^5}{0.0001^{0.25} \times 0.5^5 + 0.0001^{0.25} \times 0.5^5} = 50\% \]  
(4.15)

\[ \Psi_{\text{path}}(a, 2, p_{21}) = \frac{0.0001^{0.25} \times 0.5^5}{0.0001^{0.25} \times 0.5^5} = 100\% \]  
(4.16)

As above, a random number must be generated to determine if a greedy decision should be made. A number between 0 and \( q_0 \) would normally result in placing the path with the highest value of \( \Psi_{\text{path}} \). In this case, for call 1, the values are the same so \( \Psi_{\text{path}} \) is used. Suppose that call 1 generates \( L_{a1} = (p_{11}, p_{12}) \) and \( L_{a2} = (p_{21}) \).

In summary, we now have \( C_a = (1, 2), L_{a1} = (p_{11}, p_{12}), \text{ and } L_{a2} = (p_{21}) \). These lists are used to generate a solution in the next step.

**Step 4.** Attempt to place each call on each ant’s call list with each path in its path list.

This step generates the solutions. We iterate through each ant’s lists sequentially attempting to place each call. The first call, call 1, can be placed using its first path, \( p_{11} \). Once the call is placed, the next call in \( C_a \) is considered. In this example, call 2 is the next call in \( C_a \). Call 2 has only one feasible path and after the placement on call 1, \( p_{21} \) cannot support the bandwidth demand of call 2. Thus, call 2 cannot be placed. The solution of placing call 1 using path \( p_{11} \) and not placing call 2 yields a profit of 180.

**Step 5.** Save the best ant’s call and path lists.

After all of the ants have generated solutions, each solution is checked against the best solution so far. In our example, it is the first iteration so the best profit so far is 0. Our ant generated a profit of 180 so its lists are saved.

**Step 6.** Update pheromone levels on all calls and paths.

The decisions made by each ant must now be rewarded based on the quality of the solutions they obtained. The \( \sigma \) elite ants also come into play at this stage. Adding the pheromone from our ant, plus the additional 2 elite ants using the best solution so far yield new pheromone levels of

\[ \tau_{\text{call}}(1) = (1 - 0.3) \times 0.0001 + (1 + 2) \frac{1}{180 - 180 + 1} = 2.00007 \]  
(4.17)

\[ \tau_{\text{call}}(2) = (1 - 0.3) \times 0.0001 + 0 = 0.00007 \]  
(4.18)

\[ \tau_{\text{path}}(1, p_{11}) = (1 - 0.3) \times 0.0001 + (1 + 2) \frac{1}{180 - 180 + 1} = 2.00007 \]  
(4.19)
\[
\tau_{path}(1, p_{12}) = (1 - 0.3) \times 0.0001 + 0 = 0.00007 \\
\tau_{path}(2, p_{21}) = (1 - 0.3) \times 0.0001 + 0 = 0.00007
\] (4.20) (4.21)

This process is repeated until the stop criteria is reached.

4.1 Evolution of Parameters

This algorithm is driven by several parameters. The following is a discussion of each parameter, how it affects the algorithm, and what values tended to produce good results for the problems tested. Each parameter, except \( K \), was examined by varying the parameter under discussion and comparing the results to a base case as applied to a problem representative of the problems used for test. This base case consisted of: \( q_0 = 0.05 \), \( \alpha = 0.25 \), \( \beta = 5 \), \( m = 4|Q| \), \( \sigma = 2|Q| \), \( \rho = 0.3 \), and \( K = 15 \). Various values for \( K \) were not explored because the best value for \( K \) seemed particularly problem dependent. Problems can be generated that produce poor performance results for any value of \( K \). The problems tested rarely contained calls that had more than 15 possible paths to choose from. Setting \( K = 15 \) simply eliminates its effect on the algorithm for the test problems.

Fifteen problems were tested in which the best solutions generated by both the GA implementation and the Laguna-Glover TS implementation were known. This data served as a baseline from which some comparisons can be drawn. Comparisons based on the best solutions can be performed. Computational effort required to generate these solutions, however, is not known and only best solutions can be compared. The ACO algorithm matched or exceeded the results in 13 of the 15 problems. These results are summarized in Table 4.1.

The following parameter analysis was done on one of the fifteen problems that previous results are known. The problem used consisted of a network of 27 nodes, 93 calls and 37 edges and is described by Figure 4.3 and Table 4.2. The maximum number of paths for any call was 15 with a mean of 3.86. This problem was chosen because it is representative of a typical network of the problems tested and produced results that varied from run to run. Some of the test problems were easy in the sense that the algorithm converged to the best known solution almost without fail, and did so with very little computational effort under virtually any parameter settings. This test problem was sensitive to parameter modifications and tended to take longer to converge than most
<table>
<thead>
<tr>
<th>(n, q)</th>
<th>GA</th>
<th>Tabu</th>
<th>ACO</th>
</tr>
</thead>
<tbody>
<tr>
<td>(14, 35)</td>
<td>6580</td>
<td>6580</td>
<td>6580</td>
</tr>
<tr>
<td>(24, 68)</td>
<td>7060</td>
<td>7270</td>
<td>7270</td>
</tr>
<tr>
<td>(29, 70)</td>
<td>27710</td>
<td>27340</td>
<td>28250</td>
</tr>
<tr>
<td>(18, 58)</td>
<td>16210</td>
<td>16190</td>
<td>16210</td>
</tr>
<tr>
<td>(19, 47)</td>
<td>7770</td>
<td>7790</td>
<td>7790</td>
</tr>
<tr>
<td>(27, 93)</td>
<td>18900</td>
<td>18550</td>
<td>18910</td>
</tr>
<tr>
<td>(23, 93)</td>
<td>13800</td>
<td>13810</td>
<td>13860</td>
</tr>
<tr>
<td>(28, 41)</td>
<td>8770</td>
<td>8770</td>
<td>8770</td>
</tr>
<tr>
<td>(24, 87)</td>
<td>21360</td>
<td>21340</td>
<td>21250</td>
</tr>
<tr>
<td>(19, 41)</td>
<td>7640</td>
<td>7640</td>
<td>7640</td>
</tr>
<tr>
<td>(14, 23)</td>
<td>6110</td>
<td>6110</td>
<td>6110</td>
</tr>
<tr>
<td>(27, 81)</td>
<td>13330</td>
<td>13290</td>
<td>13290</td>
</tr>
<tr>
<td>(29, 52)</td>
<td>9020</td>
<td>9020</td>
<td>9020</td>
</tr>
<tr>
<td>(20, 46)</td>
<td>7900</td>
<td>7900</td>
<td>7900</td>
</tr>
<tr>
<td>(192, 20)</td>
<td>2070</td>
<td>2070</td>
<td>2090</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison Results

of the problems tested. 100 tests were run with each parameter setting and the average used for comparison.

To compare computational results, a metric needs to be defined to compare the amount of work for a given run. Counting iterations is not sufficient because different parameter settings yield different amounts of work to complete an iteration. A parameter setting that uses 100 ants takes 100 times as much work and produces 100 times as many solutions as a setting that uses only one ant. Thus, the idea of a one-ant cycle is introduced. A one-ant cycle is the amount of effort it takes for one ant to generate one solution. For this implementation, this consists of generating a call ordering, generating a path ordering for each call, and performing a call placement phase. This is not an exact quantity as the call placement phase can take two ants a different number of computations. This difference is minimal for sparse networks, however, and is ignored. All of the fifteen problems tested had an average number of paths in the neighborhood of 1 to 5, so the work expended for each ant was very similar. Each run was given 200 one-ant cycles before being cut off. This translates to 66 iterations for 3 ants per call and 40 iterations for 5 ants per call.
Figure 4.3: Network for Test Problem
<table>
<thead>
<tr>
<th>Call</th>
<th>From</th>
<th>To</th>
<th>Demand</th>
<th>Revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>380</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>190</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>30</td>
<td>940</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>9</td>
<td>20</td>
<td>760</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>19</td>
<td>15</td>
<td>650</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>20</td>
<td>15</td>
<td>590</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>23</td>
<td>25</td>
<td>770</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>24</td>
<td>15</td>
<td>530</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>26</td>
<td>10</td>
<td>420</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>27</td>
<td>10</td>
<td>420</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>8</td>
<td>25</td>
<td>1130</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>10</td>
<td>25</td>
<td>890</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>13</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>20</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>25</td>
<td>10</td>
<td>360</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>300</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>9</td>
<td>5</td>
<td>190</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>20</td>
<td>10</td>
<td>360</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>480</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>230</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>16</td>
<td>10</td>
<td>400</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>19</td>
<td>10</td>
<td>480</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>360</td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>25</td>
<td>5</td>
<td>17</td>
<td>25</td>
<td>1230</td>
</tr>
<tr>
<td>26</td>
<td>5</td>
<td>18</td>
<td>10</td>
<td>380</td>
</tr>
<tr>
<td>27</td>
<td>5</td>
<td>21</td>
<td>15</td>
<td>590</td>
</tr>
<tr>
<td>28</td>
<td>5</td>
<td>22</td>
<td>10</td>
<td>360</td>
</tr>
<tr>
<td>29</td>
<td>5</td>
<td>23</td>
<td>10</td>
<td>360</td>
</tr>
<tr>
<td>30</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>230</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>10</td>
<td>10</td>
<td>440</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>17</td>
<td>10</td>
<td>380</td>
</tr>
<tr>
<td>33</td>
<td>6</td>
<td>27</td>
<td>10</td>
<td>480</td>
</tr>
<tr>
<td>34</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>570</td>
</tr>
<tr>
<td>35</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>460</td>
</tr>
<tr>
<td>36</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td>570</td>
</tr>
<tr>
<td>37</td>
<td>7</td>
<td>18</td>
<td>10</td>
<td>340</td>
</tr>
<tr>
<td>38</td>
<td>7</td>
<td>19</td>
<td>10</td>
<td>300</td>
</tr>
<tr>
<td>39</td>
<td>7</td>
<td>20</td>
<td>15</td>
<td>450</td>
</tr>
<tr>
<td>40</td>
<td>7</td>
<td>22</td>
<td>5</td>
<td>190</td>
</tr>
<tr>
<td>41</td>
<td>7</td>
<td>23</td>
<td>20</td>
<td>620</td>
</tr>
<tr>
<td>42</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>420</td>
</tr>
<tr>
<td>43</td>
<td>8</td>
<td>17</td>
<td>15</td>
<td>630</td>
</tr>
<tr>
<td>44</td>
<td>8</td>
<td>22</td>
<td>30</td>
<td>1020</td>
</tr>
<tr>
<td>45</td>
<td>8</td>
<td>23</td>
<td>25</td>
<td>1110</td>
</tr>
<tr>
<td>46</td>
<td>9</td>
<td>11</td>
<td>20</td>
<td>920</td>
</tr>
<tr>
<td>47</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>650</td>
</tr>
<tr>
<td>48</td>
<td>9</td>
<td>13</td>
<td>10</td>
<td>480</td>
</tr>
<tr>
<td>49</td>
<td>9</td>
<td>14</td>
<td>6</td>
<td>230</td>
</tr>
<tr>
<td>50</td>
<td>9</td>
<td>16</td>
<td>20</td>
<td>990</td>
</tr>
<tr>
<td>51</td>
<td>9</td>
<td>18</td>
<td>5</td>
<td>210</td>
</tr>
<tr>
<td>52</td>
<td>9</td>
<td>22</td>
<td>5</td>
<td>230</td>
</tr>
<tr>
<td>53</td>
<td>9</td>
<td>23</td>
<td>25</td>
<td>930</td>
</tr>
<tr>
<td>54</td>
<td>9</td>
<td>25</td>
<td>10</td>
<td>340</td>
</tr>
<tr>
<td>55</td>
<td>10</td>
<td>12</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>56</td>
<td>10</td>
<td>16</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>57</td>
<td>10</td>
<td>17</td>
<td>25</td>
<td>830</td>
</tr>
<tr>
<td>58</td>
<td>10</td>
<td>21</td>
<td>25</td>
<td>1010</td>
</tr>
<tr>
<td>59</td>
<td>10</td>
<td>26</td>
<td>25</td>
<td>950</td>
</tr>
<tr>
<td>60</td>
<td>11</td>
<td>13</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>61</td>
<td>11</td>
<td>15</td>
<td>15</td>
<td>470</td>
</tr>
<tr>
<td>62</td>
<td>11</td>
<td>19</td>
<td>30</td>
<td>990</td>
</tr>
<tr>
<td>63</td>
<td>11</td>
<td>20</td>
<td>30</td>
<td>1240</td>
</tr>
<tr>
<td>64</td>
<td>11</td>
<td>24</td>
<td>30</td>
<td>1160</td>
</tr>
<tr>
<td>65</td>
<td>11</td>
<td>25</td>
<td>20</td>
<td>920</td>
</tr>
<tr>
<td>66</td>
<td>11</td>
<td>26</td>
<td>5</td>
<td>230</td>
</tr>
<tr>
<td>67</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>650</td>
</tr>
<tr>
<td>68</td>
<td>12</td>
<td>16</td>
<td>5</td>
<td>230</td>
</tr>
<tr>
<td>69</td>
<td>12</td>
<td>22</td>
<td>20</td>
<td>640</td>
</tr>
<tr>
<td>70</td>
<td>12</td>
<td>26</td>
<td>10</td>
<td>440</td>
</tr>
<tr>
<td>71</td>
<td>13</td>
<td>24</td>
<td>5</td>
<td>230</td>
</tr>
<tr>
<td>72</td>
<td>14</td>
<td>26</td>
<td>5</td>
<td>150</td>
</tr>
<tr>
<td>73</td>
<td>15</td>
<td>17</td>
<td>5</td>
<td>210</td>
</tr>
<tr>
<td>74</td>
<td>15</td>
<td>19</td>
<td>10</td>
<td>320</td>
</tr>
<tr>
<td>75</td>
<td>15</td>
<td>21</td>
<td>10</td>
<td>340</td>
</tr>
<tr>
<td>76</td>
<td>16</td>
<td>17</td>
<td>20</td>
<td>900</td>
</tr>
<tr>
<td>77</td>
<td>16</td>
<td>23</td>
<td>10</td>
<td>440</td>
</tr>
<tr>
<td>78</td>
<td>16</td>
<td>26</td>
<td>30</td>
<td>980</td>
</tr>
<tr>
<td>79</td>
<td>17</td>
<td>18</td>
<td>10</td>
<td>380</td>
</tr>
<tr>
<td>80</td>
<td>17</td>
<td>20</td>
<td>25</td>
<td>770</td>
</tr>
<tr>
<td>81</td>
<td>17</td>
<td>21</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>82</td>
<td>17</td>
<td>25</td>
<td>15</td>
<td>450</td>
</tr>
<tr>
<td>83</td>
<td>18</td>
<td>20</td>
<td>5</td>
<td>150</td>
</tr>
<tr>
<td>84</td>
<td>19</td>
<td>24</td>
<td>15</td>
<td>530</td>
</tr>
<tr>
<td>85</td>
<td>20</td>
<td>21</td>
<td>20</td>
<td>780</td>
</tr>
<tr>
<td>86</td>
<td>20</td>
<td>23</td>
<td>10</td>
<td>440</td>
</tr>
<tr>
<td>87</td>
<td>20</td>
<td>25</td>
<td>5</td>
<td>210</td>
</tr>
<tr>
<td>88</td>
<td>21</td>
<td>23</td>
<td>15</td>
<td>650</td>
</tr>
<tr>
<td>89</td>
<td>22</td>
<td>23</td>
<td>15</td>
<td>470</td>
</tr>
<tr>
<td>90</td>
<td>22</td>
<td>24</td>
<td>30</td>
<td>1180</td>
</tr>
<tr>
<td>91</td>
<td>22</td>
<td>26</td>
<td>20</td>
<td>820</td>
</tr>
<tr>
<td>92</td>
<td>23</td>
<td>26</td>
<td>5</td>
<td>170</td>
</tr>
<tr>
<td>93</td>
<td>23</td>
<td>27</td>
<td>10</td>
<td>420</td>
</tr>
</tbody>
</table>

Table 4.2: Call Table for Test Problem
The first parameter is the number of ants, \( m \). A small number of ants can search only a small subset of the solution space before pheromone is added. This causes misleading information in the pheromone trail. Because of this, ants tend to converge on suboptimal solutions and never venture far from them. Large values for \( m \), on the other hand, generate a large set of solutions before pheromone levels grow. This causes the pheromone trail to provide more accurate information as to how a particular decision affects the solution. On the down side, each ant can be costly if the problem is large. A balance must be found as to how many ants are adequate to find a good solution without becoming too costly computationally. In general, tests using a small number of ants found better solutions early on but were eventually outperformed by runs that used more ants. The average values over 100 runs for \( m \) set at 3, 4, and 5 are displayed in Figure 4.4. Notice that smaller values of \( m \) tend to perform better early on, less than 50 one-ant cycles, while larger values tend to perform better in the long term. When the test was stopped after its allotted 200 one-ant cycles, \( m = 4 \) was the best performing value. Larger values exceeded these results when more computational time was allowed.

The basis of ACO is its probability function, \( \Psi(a, d) \). This function governs the ant’s decisions, and thus, build their solutions. The two factors that drive this function, \( \tau(d) \) and \( \eta(d) \), are controlled by means of two parameters, \( \alpha \) and \( \beta \), respectively.

The parameter \( \alpha \) is the exponent of \( \tau(d) \) in \( \Psi(a, d) \). As \( \tau(d) \) tends to be small, values of \( \alpha > 1 \) significantly reduce the affect the pheromone has on the decision process. This causes ACO to become nothing more than a greedy algorithm, using only local network information when constructing solutions. The results for the problems tested showed higher profits for values of \( \alpha = 0.25 \). The results for \( \alpha \) set at 0.15, 0.25, and 0.4 are shown in Figure 4.5. \( \alpha = 0.4 \) significantly dominates both of the other parameter settings for about the first 100 one-ant cycles or so, at which point \( \alpha = 0.25 \) surpasses it. This type of behavior leads me to think that a dynamic value for \( \alpha \) may prove successful. For example, interpolating between 0.4 and 0.25 over the course of the run would have the advantage of finding good solutions early but exploring more as time proceeds.

The parameter \( \beta \) is the exponent of \( \eta(d) \) in \( \Psi(a, d) \). Early tests for various values of \( \beta \) showed drastically different results. As a result of this, a value for \( \beta \) that changed over the course of the run was explored. The value used for the exponent of \( \eta(d) \) was interpolated from 0 to the parameter value \( \beta \) over the course of the run. The results of such a dynamic \( \beta \) produced significant
Figure 4.4: Variation of $m$
Figure 4.5: Variation of $\alpha$

$\triangle: \alpha = 0.15 \quad \square: \alpha = 0.25 \quad \diamondsuit: \alpha = 0.4$
improvements not only in profits but also in the size of the stability region for the parameter. Figure 4.6 shows the test results for values of 0, 5, and 10. These settings are drastically different, considering they are used as exponents, yet the profits yielded by both non-zero values produced results in the same neighborhood. $\beta = 0$ removes the fitness aspect from $\Psi(a, d)$ completely and performs very poorly. This is encouraging in that it validates the use of $\eta(d)$ in the ant’s decisions.

The pheromone trail, $\tau(d)$, has two forces affecting it: Accumulation and evaporation. Accumulation is controlled by $\Delta \tau(d)$ which is driven by the quality of solution the decision has helped to generate. Evaporation is controlled by the parameter $(1 - \rho)$. This evaporation factor is used to prevent the level
of pheromone on decisions becoming too large, dominating the decision process and discouraging the ants from searching previously unexplored regions of the solution space. A large evaporation factor prevents this. Too much evaporation, however, disguises the pheromone trail and not provide the ants information as to how the decision has performed historically. The results for values of \((1 - \rho)\) of 0.8, 0.7, and 0.6 are displayed in Figure 4.7. The results for these three settings are all similar in the early iterations. As time progresses, the amount of pheromone has been allowed to accumulate and the effect of \((1 - \rho)\) becomes significant.

The results of the test problems showed improvements when a greedy aspect was added. The parameter \(q_0\), is used to control this greedy component.
\( q_0 = 0 \) provides no greedy aspect while \( q_0 = 1 \) makes the ant's decision purely greedy. Results for values of \( q_0 \) are shown in Figure 4.8. Values for \( q_0 \) that were much greater than zero did not perform well for this parameter. Even small values improved performance only slightly. This may warrant discarding this parameter altogether.

The number of elite ants, \( \sigma \), is used to encourage the decisions used to generate the best solution. Results of tests using the addition of elite ants were observed to strongly dominate tests in which the elite ants were not used. They generated better solutions at every stage of the tests as long as the number of elite ants was not so great as to cause all of the ants to make the same decisions. Results for values of \( \sigma \) set at 0, 2, and 4 are shown in
Figure 4.9: Variation of $\sigma$

Figure 4.9. Notice how drastic the performance increases when elite ants are used. Both non-zero settings strongly dominate the solutions generated by using no elite ants at all.
5. Conclusions

This thesis has presented the bandwidth packing problem and several approaches that have been applied to it. In addition, we have presented a new approach using Ant Colony Optimization. BWP can be formulated as an integer program but becomes too large to solve directly. Metaheuristics have become an increasingly accepted approach to solving problems that are not suited to direct methods. Genetic algorithms and tabu search have both been applied to BWP. ACO is a novel approach to solve combinatorially difficult problems ranging from the traveling salesman problem to adaptive routing in communications networks. Fifteen problems were tested in which the best solutions generated by GA and the Laguna-Glover TS implementation were known. The ACO results performed well against these other approaches, matching or exceeding the results in 13 of the 15 problems. Computational comparisons in terms of processor time was not available however.

The ACO approach taken here differs from standard ACO implementations in that it uses a dynamic value for $\beta$ instead of the usual static parameter value. This approach proved very successful for this implementation and warrants further study on other ACO applications. Test results also showed that dynamic values for other parameters may also be beneficial. The ACO metaphor could be further implemented by allowing the values of the parameters to change probabilistically based on past performance. This would allow the heuristic to adapt to the specific problem instance it is solving.
References


44


Glossary of Symbols and Notation

\begin{itemize}
\item $b_e$ The bandwidth capacity of edge $e$
\item $c_e$ The cost for each unit of bandwidth used on edge $e$
\item $v_i$ The revenue associated with placing call $i$
\item $r_i$ The bandwidth requirement for call $i$
\item $s_i$ The start node for call $i$
\item $t_i$ The terminal node of call $i$
\item $N$ The set of network nodes
\item $Q$ The set of calls
\item $P_i$ The set of feasible paths for call $i$
\item $E$ The set of network edges
\item $\mathcal{E}$ The set of network arcs
\item $\mathcal{H}(j)$ The set of arcs terminating at node $j$
\item $\mathcal{T}(j)$ The set of arcs originating from node $j$
\item $\mathcal{N}(x)$ The set of solutions neighboring $x$
\item $K$ The maximum number of feasible paths to search for
\item $l_p$ The cost of path $p$
\item $E_p$ The set of edges in path $p$
\item $V(i, h)$ The value of move $(i, h)$
\item $k_i$ The number of feasible paths considered for call $i$
\item $\omega(i, h)$ The number of times move $(i, h)$ has been made
\item $\phi(i, h)$ The penalized move value for move $(i, h)$
\item $\tau(d)$ The pheromone on decision $d$
\item $\Delta \tau(d)$ The amount of pheromone added to decision $d$
\item $(1 - \rho)$ The evaporation factor
\item $A_{ip}$ The set of ants that placed call $i$ using path $p$
\item $A_i$ The set of ants that placed call $i$
\item $\eta(d)$ The fitness of decision $d$
\item $\Psi(a, d)$ The probability associated with ant $a$ making decision $d$
\end{itemize}
\(\alpha\)  The parameter that determines the level which pheromone governs an ant's decision
\(\beta\)  The parameter that determines the level which fitness governs an ant's decision
\(q_0\)  The parameter that determines the greedy component of an ant's decisions
\(\sigma\)  The number of elite ants
\(m\)  The number of ants