The Bounded Cycle Cover Problem

Dorit S. Hochbaum
Department of Industrial Engineering and Operations Research and
Walter A. Haas School of Business
University of California, Berkeley; email: dorit@hochbaum.berkeley.edu

Eli V. Olinick
Computer Science and Engineering Department
School of Engineering and Applied Science
Southern Methodist University
email: olinick@seas.smu.edu

July 26, 2000

Subject classifications: Telecommunications, Networks-Graphs: design and heuristics
Other key words: Chinese postman problem, cycle basis problem, cycle cover, SONET rings

We consider the bounded cycle cover problem, which is to find a minimum cost cycle cover of a two-connected graph such that no cycle in the cover contains more than a prescribed numbered of edges. This problem arises in the design of fiber-optic telecommunications networks that employ multiple self-healing rings to provide routing for communication traffic, even in the event of a fiber cut or other type of link failure. We present this problem, along with several related problems, and develop heuristic algorithms that find near optimal solutions for the bounded cycle cover problem based on solution techniques for these related problems. Empirical results of these algorithms, applied to randomly generated problem instances, are presented and discussed.

In this paper, we consider the *bounded cycle cover problem* (BCCP), which is to find a minimum cost cycle cover of a two-connected graph such that no cycle in the cover contains more than a prescribed numbered of edges. A cycle is said to *cover* the edges it contains and a *cycle cover* of a graph is a set of cycles whose union contains all the graph's edges. If there are costs or weights associated with the edges of the graph, then the cost of a cycle is the sum of the costs of the edges it covers. We give a precise definition of the problem in Section 2 and illustrate it here with an example in Figure 1. Note that in Figure 1, the edge costs are indicated by the numbers next to the edges of the graph. Since edges $(1,4)$ and $(2,3)$ appear in both cycles, they are charged for twice, making the total cost of the solution 130.

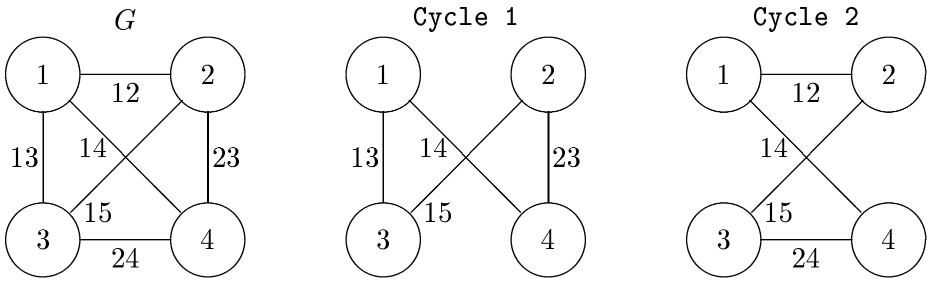

Figure 1: BCCP example with a maximum cycle size of 4 edges.

BCCP is an intractable problem. The number of potential cycles in a graph grows exponentially as a function of the number of nodes in the graph and the allowable cycle size. Our approach is based on finding a good, but not necessarily optimal solution, by first finding a set of candidate cycles that form a cover. The candidate cycles are found by solving optimization problems that are very similar to BCCP, yet can be solved optimally with polynomial-time algorithms. We then find a BCCP solution by selecting a minimum cost subset of the candidate cycles that covers the edges of the graph. Selecting this subset of the candidate cycles is also an intractable problem, but, as described in Section 9, we are able to select the optimal subset of the candidate cycles quickly using standard integer programming techniques.

In Section 1, we describe how BCCP arises in the design of fiber-optic telecommunications networks that employ multiple self-healing rings to provide routing for communication traffic in the event of a fiber cut or other type of link failure. The rest of this paper is laid out as follows. In Section 2, we present a formal definition of BCCP in graph theoretic terms. In Section 3, we discuss the complexity of finding optimal and feasible solutions to BCCP. Related optimization problems that can be solved efficiently to provide lower and upper bounds on BCCP are presented in Section 4. A formulation of BCCP as a Set Cover Problem is given in Section 5, and techniques for finding feasible solutions to the Set Cover Problem are discussed in Section 8. In Sections 6 and 7, we develop heuristic algorithms for BCCP based on solving the problems presented in Section 4. The results of computational experiments with running these algorithms on randomly-generated problem instances are presented in Section 9.

1 BCCP and the Design of Survivable Telecommunication Networks

The reliability of telecommunications networks has become increasingly important as more and more organizations throughout the world have grown to depend on them. An important consideration in the design of such a network is its survivability— that is, its ability to continue to provide communication between the sites it connects in the event that one or more of its links are cut or otherwise fail.

If a particular link, say between nodes i and j, in the network fails, the service provided by the link can be restored by rerouting messages that would normally traverse the link ij around an alternative path. This is referred to as *dynamic service restoration* and takes place on a *logical* level. In other words: changes are made in the routing protocol so that messages reach their destination by other paths, but the network remains physically unchanged. For this scheme to work, the network must be built so that there are at least two independent routes that have no links in common between any pair of nodes. This type of network topology is referred to as a *2-connected* design and is fundamental to the design of networks with high survivability. If the network does not have this feature, then there will be at least one pair, and possibly many more, of nodes that will not be able to communicate with each other if link ij fails.

Along with a 2-connected topology, survivable network designs often exploit a *ring*. A ring in a network is a set of nodes that form a closed loop. In order to take advantage of the ring structure and optimize message rerouting, dynamic restoration schemes may use a control system that refers to a database containing information about the status of all the links in the network. These schemes have the drawback that they are complex and difficult to implement. Furthermore, it may take a long time to detect failures and restore service at the logical level of the network [21]. By using dedicated hardware and simple protocols for detecting and rerouting around failures, *self-healing rings* (SHRs) enable automatic and fast service restoration at the physical layer of the network in an area that is localized around the damaged link. Only the nodes in the SHR need to modify the way they route traffic.

There are several varieties of SHRs (see [18, 21]) with the basic principle that the *working ring*, an existing set of links in a communications network that forms a ring, is backed up by a second, parallel ring of redundant links called the *protection ring*. For example, some SHRs use a *line switch* mechanism, whereby a failure in the working ring is bypassed by switching all communication to the protection ring. We say that the links in the working ring are *protected* by the SHR.

Given an existing network that does not use self-healing rings, we consider the problem of improving survivability by constructing a set of SHRs (a *ring cover*) such that there is at least one SHR containing protection and working copies of link ij if there is a link between sites i and j in the original network. The data for the problem includes a link cost c_{ij} that is incurred by each SHR that has direct links between sites i and j. This cost, defined for all links, may include the cost of purchasing and installing extra fiber material and cable terminating equipment at the network's nodes.

We represent the original telecommunications network as an undirected graph $G = (V, E)$ where the vertices (or nodes) represent the sites connected to each other by the network, and edge (i, j) represents a link between sites i and j. The edge costs in the graph are equal to the corresponding link costs. This gives us a one-to-one correspondence between ring covers of the links in the network and cycle covers of the graph. For example, we can construct a ring cover of the original network G in Figure 1 with two SHRs: one corresponding to Cycle 1 and another to Cycle 2. Note that both SHRs incur costs c_{14} and c_{23} for using links between sites 1 and 4 and between sites 2 and 3. The self-healing ring architecture is associated with the Synchronous Optical NETwork (SONET) standard which limits the number of nodes or sites on a ring to a maximum of 16 [16]. In practice, SHRs typically consist of 10 or fewer nodes [21], and so we assume that a maximum allowable ring size is also given, and this corresponds to the cycle-size limit in BCCP.

Having motivated BCCP, we now place it in the larger context of survivable telecommunications network design problems. The volume of traffic in a SHR is limited by the ring capacity, which must be equal for all working and protection links [5, 21]. The capacity of a ring is largely determined by the type of add-drop-multiplexers (ADM) placed at its nodes. ADMs are devices used by the nodes to send and relay traffic to and receive traffic from other nodes on the ring. Since we model the per-node equipment costs as part of the link costs, we do not explicitly address capacity considerations.

Slevinsky et al. [19] proposed the *multi-ring design problem*, which is to find a ring cover of the network, which is optimal with respect to an objective function derived from measures of *capacity efficiency* rather than cost. They present a greedy heuristic for finding a minimum cost cycle cover (without a cycle-size restriction) of a graph where the cost of a cycle is defined as the number of edges in the cycle times the cost of the most expensive edge in the cycle. Kennington et al. [15] present an optimization-based solution procedure for this variant of the cycle cover problem. Kennington and Rahman [16] adapt this approach to the SONET standard where there is a 16-node ring-size limit and the capacity of a ring must be a multiple of 28.

Optimizing the design of survivable telecommunications networks is a very complex problem that is not practically addressed with a single model. As described by Laguna [17] and Cosares et al. [4], successive levels of the design process of SONET networks present different optimization problems. For example, Cosares and Saniee [5] and Dell'Amico et al. [6] consider the problem of partitioning the traffic on a single bidirectional symmetric SHR between clockwise and counterclockwise traffic so as to optimize the ring capacity. [1] The problem addressed in these papers occurs at a "lower" level in the design of the network than the problem discussed in this paper, in the sense that it would be addressed after resolving "higher" level problems, such as forecasting the demand that will be placed on the network and determining which of the possible SHRs will be built.

Typically, problems like BCCP and the cycle cover problems discussed in [19, 15, 16] would be solved as subproblems of the overall network planning process. For example, the BCCP heuristics we propose could be used for finding a set of good rings to cover a so-called "community of interest" (CoI) in the "Ring Routing Problem" described in [4]. A CoI is a set of sites

[1] In a bidirectional ring, traffic is carried by both the working and protection rings and may flow in either direction. SONET is a circuit-based protocol. Thus, a connection between i and j creates an equal amount of traffic from i to j and return traffic from j to i. The line switch mechanism described earlier is used in unidirectional rings, and so the traffic between nodes i and j will take up capacity on the entire working ring. A bidirectional ring can effectively have a greater capacity by limiting this traffic to the clockwise portion of the "working" ring between and i and j and the corresponding counter-clockwise portion on the "protection ring".

that are geographically close to together, have a large amount of demand between them, and are thus ideally suited for a ring-based network. In this setting, one seeks a set of cycles covering the graph's *nodes* rather than its edges. However, a cycle cover of the edges of a graph also covers the nodes and one could easily adapt the set-covering formulation of BCCP presented in Section 5 for the node-covering version of the problem.

Network planners often conduct scenario analyses. They estimate the cost of building a new network or expanding an existing one under a variety of conditions involving different demand patterns and equipment costs. Such studies are short-term in nature and may involve solving a series of related problems. In this setting, BCCP would need to be solved many times, and the speed of the heuristics may be as important, or perhaps more important, than the quality of the solutions they produce.

2 Notation and Formulation

Throughout this paper, $G = (V, E)$ is a graph with $n = |V|$ nodes and $m = |E|$ edges. The cost or weight of an edge is c_e. A *path* between node i and node j is a sequence of adjacent nodes and edges with no repeated nodes. The notation P_{ij} indicates the set of edges in a path between i and j. A *cycle* on nodes i and j is a path between i to j followed by the edge (i, j). C_{ij} denotes the set of edges in a cycle on nodes i and j. The *cost*, or *weight*, of a cycle is defined as the sum of the costs of the edges it contains and the *size*, $|C_{ij}|$, of a cycle as its number of edges. The cost, weight, and size of a path are defined in a similar manner. The term *shortest path* between i and j is generally used to indicate a path between i and j that minimizes the sum of the edge costs, $\sum_{e \in P_{ij}} c_e$. In some cases, however, we use the term to indicate a path that minimizes the *number* of edges or the *length* of the path (i.e. $|P_{ij}|$). We will indicate whether we mean a shortest path with respect to cost or number of edges whenever we use the term (although it is generally clear from the context).

If cycle C contains edge e, we say that C *covers* or *spans* e. A set of cycles that cover all the edges in the graph G is called a *cycle cover* of G. The cost of a cycle cover is the sum of the costs of its cycles. We assume that the network is 2-connected (i.e, the graph may not contain a cut-edge - an edge whose removal destroys all paths between some pair of nodes).

The parameter k is the maximum number of nodes allowed in a cycle. A path containing k or fewer edges is a *k-path*, and a *k-cycle* is a cycle of size at most k. A *k-cycle cover* of a graph G is a cycle cover of G consisting entirely of k-cycles. A cycle cover necessarily contains each edge once, but any given edge may be covered by more than one cycle. In other words, the cover may contain "extra" copies of some edges. The bounded cycle cover problem may be stated as that of duplicating a minimum cost set of edges, such that the edges of the resulting multi-graph may be partitioned into k-cycles.

Let the decision variable x_e be the number of extra copies of link e built to create self-healing rings and E_e be the set of x_e copies of the link. The bounded cycle cover problem (BCCP) is defined formally as follows.

Problem Name: *Bounded Cycle Cover Problem*

Instance: *Given a 2-connected graph $G = (V, E)$, a cost c_e associated with each edge $e \in E$ and an integer k.*

Optimization Problem: *Minimize $\sum_{e \in E} c_e x_e$ such that $E \cup \bigcup_{e \in E} E_e$ can be partitioned into a $k-$cycle cover of G or show that the problem is infeasible for the given value of k.*

We may consider finding a cycle cover that minimizes the sum of the sizes of the cycles rather than the cost. This is equivalent to cases where $c_e = 1$, or any fixed constant, for all e. We refer to this special case as the *unweighted* version of the problem. Alternatively, when reliability of the nodes rather than the links is a major concern, we can choose to find a set of cycles that spans the nodes (but not necessarily the edges) of G. We mention this version of the problem for the sake of completeness, but our focus is on the edge covering version. Any set of cycles that spans the edges of G also spans the vertices of G. Thus, the algorithms we present for the edge-covering version of BCCP also solve the node covering problem.

3 Complexity of the Bounded Cycle Cover Problem

3.1 Finding a Feasible Bounded Cycle Cover

While finding an optimal k-cycle cover for a given graph G is an inherently difficult problem, finding a feasible one, or determining that none exists, is not. A necessary condition for the existence of a k-cycle cover for G is that each edge must belong to at least one cycle with k

or fewer edges. This implies that for each edge (i, j) there exists a path of length at most $k-1$ from i to j which does not contain (i, j). Such a path can be found by conducting a breadth-first-search (BFS) from node j in the graph $G_{ij} = (V, E \setminus \{(i, j)\})$. If node i is found in the first $k-1$ levels of the BFS tree, then we have found a $k-1$ path between i and j. If not, then there is no $k-$cycle in G containing edge (i, j). The BFS procedure requires $O(m)$ steps per edge. Thus, a feasible k-cycle cover can be found in $O(m^2)$ time.

Define $|P^*_{ij}|$ as the length of the shortest path (with respect to the number of edges) between nodes i and j in the graph $G = (V, E \setminus \{(i, j)\})$. The minimum length of a cycle containing edge (i, j) is thus $|P^*_{ij}| + 1$. By solving m shortest path problems, we may determine the minimum cycle length $k^* = 1 + \max_{(i,j)\in E} |P^*_{ij}|$ for which BCCP is feasible. Using a Fibonacci heap implementation of Dijkstra's algorithm, we can find each path $|P^*_{ij}|$ in $O(m + n \log n)$ time [9]. Thus, the complexity of determining k^* is $O(m^2 + mn \log n)$.

3.2 BCCP is $\mathcal{NP}$-hard

The decision version of BCCP has a threshold parameter, L. The problem is to decide whether there are values for x_e such that $E \cup \bigcup_{e\in E} E_e$ can be partitioned into a k-cycle cover of G and $\sum_{e\in E} c_e x_e \leq L$. Thomassen [20] showed that the *minimum cost cycle cover problem* (MCCP), which is just BCCP with no bound on the number of edges in a cycle, is $\mathcal{NP}$-complete. Given a graph $G = (V, E)$ and integer L, the decision version of MCCP is to decide whether there are values for x_e such that $E \cup \bigcup_{e\in E} E_e$ can be partitioned into a cycle cover of G with cost at most L. Since an instance of MCCP is also an instance of BCCP with $k = n$, it follows that BCCP is $\mathcal{NP}$-hard.

4 Related Problems

In this section, we present graph optimization problems related to the bounded cycle cover problem. Two of these problems, the Chinese postman and minimum cost cycle basis, can be solved in polynomial time and provide efficient means for finding lower and upper bounds on the cost of an optimal k-cycle cover for a given graph.

4.1 The Minimum Unbounded Cycle Cover Problem

One way of finding a lower bound on the cost of feasible k-cycle covers for a given graph G is to relax the constraint on the size of the cycles used. In this case, we are simply looking for a minimum cost cycle cover of G. We refer to this problem as the *minimum cost cycle cover problem* (MCCP).

Itai and Rodeh [14] considered the unweighted case of MCCP which arises from the analysis of irrigation systems and electrical circuits. They showed that every 2-connected, undirected graph has a cycle cover with total cost at most $m + 2n \log n$ and give a randomized algorithm with an expected running time of $O(n^2)$ which finds such a cover. Itai et al. [13] presented an $O(n^2)$ algorithm which finds a cycle cover with length no more than 3 times that of a minimum cost cover.

4.2 The Chinese Postman Problem

As noted in Section 3, MCCP is $\mathcal{NP}$-complete. There is a connection, however, between this problem and the well known *Chinese postman problem* (CPP), which can be solved efficiently for undirected graphs [7]. A *tour* in a graph $G = (V, E)$ is a sequence of adjacent edges which starts and ends with the same node. For example, the sequence $(1,2),(2,3),(1,3),(1,4),(2,4),(1,2)$ is a tour in the graph in Figure 1. A tour which contains every edge of a graph at least once is called a *postman tour* of G. CPP seeks a postman tour of minimum cost. The CPP solution is also a minimum cost set of cycles that cover the edges of G; however, the two problems (MCCP and CPP) are not identical. The distinction is that CPP allows cycles of length 2. Such cycles are referred to as "degenerate" cycles and are not allowed in MCCP solutions. The following example illustrates this point.

One can often construct a cycle cover of the graph given a postman tour. For example, a CPP solution for graph G in Figure 1 is $(1,4)$, $(2,4)$, $(2,3)$, $(1,3)$, $(1,4)$, $(3,4)$, $(2,3)$, $(1,2)$. This tour can be decomposed into the subtour $(1,4)$, $(2,4)$, $(2,3)$, $(1,3)$ followed by the subtour $(1,4)$, $(3,4)$, $(2,3)$, $(1,2)$. These subtours of G in turn give the cycle cover shown in Figure 1.

This tour decomposition process alone, however, is not sufficient to find a cycle cover of a given graph G. The postman tour may go "back and forth" over the same edge. For example, another CPP solution for the graph in Figure 1 is the tour $(1,2)$, $(2,4)$, $(1,4)$, $(1,3)$, $(2,3)$, $(2,3)$, $(3,4)$, $(1,4)$. This tour can be decomposed into subtours $\{(1,2),(2,4),(1,4)\}$, $\{(1,3),(3,4),(1,4)\}$ and a "degenerate" subtour from node 2 to node 3 and back to node 2. Thus, the postman tour of G does not produce a valid cycle cover. There are some graphs, for example the famous Petersen graph, for which no optimal postman tour gives a valid cycle cover [14].

For an arbitrary graph G, a postman tour may not even solve MCCP, let alone BCCP, as the subtours of CPP are not restricted in length. Guan and Fleischner[11] showed that MCCP is equivalent to the Chinese postman problem for planar graphs. The equivalence follows from a result due to Fleischner [8], which states that if G is planar, then there exists a postman tour of G which is free of degenerate cycles. One may make use of this result if the graph for the BCCP instance happens to be planar, and construct an additional set of candidate rings with a polynomial-time algorithm that involves applying the four color theorem and is non-trivial to implement ([1, 22]). Even if the graph is not planar, CPP provides a lower bound for MCCP, and hence BCCP, and thus plays an important role in the construction and evaluation of the algorithms we present in Section 7.

4.3 Minimum Cycle Basis Problem

In this section, we describe an optimization problem related to BCCP called the *minimum cycle basis problem* (MCBP). Briefly stated, a cycle basis for a given graph $G = (V, E)$ is a set of cycles which can be used to generate, in a manner described below, all the cycles in G. Any cycle basis of G is also a cycle cover for the graph, but not vice versa; so the set of feasible solutions for MCBP is a subset of the solutions for MCCP. The two problems have the same objective function: to minimize the cost of the edges participating in each cycle. Thus, the solution to MCBP provides an upper bound on MCCP.

A cycle in G can be expressed as a $0-1$ m-dimensional vector with a 1 in each position that corresponds to an edge in the cycle (i.e., a 1 in position i indicates that the cycle contains edge i). Using addition modulo 2, the vectors v_1 and v_2, corresponding to cycles C_1 and C_2, can be added to form a new vector v_3, which represents a third cycle,

$$C_3 = C_1 \oplus C_2 = \{C_1 \setminus C_2\} \cup \{C_2 \setminus C_1\}.$$

The set of vectors representing cycles in G, together with the $\oplus$ operation, form a vector space known as the *cycle space* of G. A basis for the cycle space, that is, a set of cycles which can be "added" together to generate any cycle in G, is called a *cycle basis* of the graph. It follows from the definition of the $\oplus$ operation that any cycle basis of G is also a cycle cover. We now state the problem more formally.

Problem Name: *Minimum Cycle Basis Problem (MCBP)*

Instance: *Given a graph $G = (V, E)$, a cost $c(e)$ associated with each edge $e \in E$.*

Optimization Problem: *Minimize* $\sum_{C_i \in \mathcal{C}} \sum_{e \in C_i} c_e$
such that $\mathcal{C} = \{C_1, C_2, \ldots, C_i\}$ *is a cycle basis of* G.

Horton [12] gives an $O(m^3 n)$ algorithm for finding a minimum cost cycle basis of a graph with n vertices and m edges. Chickering et al. [3] show that for the unweighted case, Horton's algorithm also solves the problem of finding a minimum cost cycle basis with the additional objective that the cost of the maximum cost (i.e., the longest) cycle in the basis is minimized. The cost of a cycle is equal to its size in the unweighted case. Hence, the result in [3] means that if an unweighted graph has a minimum cost cycle basis consisting entirely of k-cycles (a k-cycle basis), Horton's algorithm will find it.

Since any cycle basis of a graph must cover all the edges, the MCBP solution returned by Horton's algorithm is a cycle cover. The cycles in MCBP are not restricted in *size*, however, and so it is necessary to modify Horton's algorithm so that it returns a set of k-cycles in order to use it to obtain an upper bound on weighted instances of BCCP. The details of this modification are given in section 7. We do not claim that this algorithm produces a minimum cost k-cycle basis, but it does give a feasible solution for BCCP, and hence provides a valid upper bound.

5 Formulation of BCCP as a Set Covering Problem

Suppose we are given all possible k-cycles and their associated costs in a particular graph. Then the bounded cycle cover problem has a natural formulation as a *Set Cover Problem* or *Set Covering Problem* (SCP):

Problem Name: *Set Covering Problem*

Instance: *Given a set $\mathcal{S}$ and set $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\ell\}$ of subsets of $\mathcal{S}$ and a weight $w(\mathcal{T}_i) \geq 0$ associated with each subset $\mathcal{T}_i \in \mathcal{T}$.*

Optimization Problem: *Find a set of subsets $\mathcal{T}_{i_1}, \mathcal{T}_{i_2} \ldots, \mathcal{T}_{i_j} \in \mathcal{T}$ such that $\bigcup_{k=1}^{j} \mathcal{T}_{i_k} = \mathcal{S}$ and $\sum_{k=1}^{j} w(\mathcal{T}_{i_k})$ is minimized.*

In the context of BCCP, $\mathcal{S} = E$, $\mathcal{T}$ is the set of all k-cycles in G, and $w(\mathcal{T}_i) = \sum_{e \in \mathcal{C}_i} c_e$. In order to solve BCCP problems as instances of SCP, we must address two issues. The first is that the number of potential k-cycles in a graph grows exponentially as n and k increase, so the problem of just entering the data into an algorithm for SCP given a particular graph is intractable.

As an alternative to listing all the k-cycles of G, we present polynomial-time heuristic algorithms for generating a set of *candidate* cycles $\mathcal{C}$, which is guaranteed to contain a k-cycle cover, not necessarily optimal, of G. We then apply set covering algorithms to $\mathcal{C}$. When $k = n$, for example, we may be able to construct a cycle cover for G from a subset of the $m - n + 1$ cycles in the minimum cycle basis of G returned by Horton's algorithm. By solving the set covering problem, we can extract a minimum cost subset of the basis cycles that span the edges of G.

Given a set $\mathcal{C}$ of candidate cycles, the second issue we must address is that SCP is $\mathcal{NP}$-hard ([10]). As discussed in Section 9, however, we are able to solve SCP optimally for the candidate sets generated by the algorithms in Section 7 relatively quickly.

6 The Generic k-Cycle Cover Algorithm

Our approach to finding "good" solutions to BCCP is a two-step process. First, we generate a set of *candidate* cycles $\mathcal{C}$, which is guaranteed to contain a k-cycle cover (although not necessarily an optimal one) of G. We then apply set covering algorithms to $\mathcal{C}$ to extract a minimum cost subset of $\mathcal{C}$ that spans the edges of G. Thus, the generic k-cycle cover algorithm can be described as follows:

Algorithm *Generic k-Cycle Cover*
Input A graph $G = (V, E)$ and an integer $k \geq k^*$
Output A set of k-cycles spanning E
(1) Generate a set of *candidate* cycles $\mathcal{C}$,
(2) Solve the corresponding SCP and return solution.

We present two algorithms for step (1) in Section 7 and describe two methods for step (2) in Section 8.

7 Algorithms For k-Cycle Covers

In this section, we present algorithms for generating a set of *candidate* cycles $\mathcal{C}$ for building a k-cycle cover of a given graph. The first algorithm uses a k-cycle basis, the second augments the basis with additional cycles, and the third algorithm starts with a cycle cover derived from a solution to the Chinese postman problem on G and adds k-cycles as necessary.

7.1 The Cycle Basis Heuristic

The following heuristic algorithm is a modification of Horton's algorithm for finding a minimum cost cycle basis. The first phase generates a set of mn candidate cycles by creating one cycle for each edge (i, j) and node v. The candidate cycle is found by taking a shortest path between i and v and another between v and j. The cycle formed by combining these paths with the edge (i, j) meets certain necessary conditions for a cycle to be in a minimum cost cycle basis. In order to ensure that the candidate cycles are all k-cycles, we use a different procedure than Horton to generate a cycle from (i, j) and v. This procedure finds a shortest k-cycle containing edge (i, j) and node v (or determines that no such cycle exists).

The second phase of the algorithm selects a minimum cost set of independent cycles from the candidate set. A set of cycles are *independent* if no cycle in the set can be generated by using a series of $\oplus$ operations on a subset of the other cycles in the set. For a given graph, the number of independent cycles in any basis is $m - n + 1$. Since there may not be a minimum cost k-cycle basis cover for a given graph, we add a third phase, which adds cycles to cover any edges that are not covered by the set of independent cycles from the second phase of the algorithm. If the algorithm does not execute the third phase, then $\mathcal{C}$ is a basis for the cycle space of G. We refer to the modified version of Horton's algorithm shown in the following figure as the *bounded cycle basis algorithm.*

Algorithm *Bounded Cycle Basis*
Input A graph, $G = (V, E)$, and an integer $k \geq k^*$
Output A candidate set of k-cycles spanning E

Phase 1
  $\mathcal{C} \leftarrow \emptyset$
  For all $(i, j) \in E$
    For all $v \in V \setminus \{i, j\}$
      $l \leftarrow \infty$
      For $x = 1$ to $k - 2$
        Find a minimum cost $x-$path, P_{vi}, between v and i
        Find a minimum cost $((k-1)-x)-$path, P_{jv}, between v and j
        If $\exists P_{vi}, P_{jv}$ **Then**
          $C \leftarrow P_{vi} \cup \{(i, j)\} \cup P_{jv}$
          If $\sum_{e \in C} c(e) < l$ **Then**
            $C_{vij} \leftarrow C$
            $l \leftarrow \sum_{e \in C} c(e)$
      If ℓ is not infinite (i.e. there is k-path between i and j containing v) **Then**
        $\mathcal{C} \leftarrow \mathcal{C} \cup C_{vij}$

Phase 2
  $\mathcal{C}' \leftarrow \emptyset$, $\mathcal{C}'' \leftarrow \mathcal{C}$
  While $|\mathcal{C}'| < m - n + 1$ **And** $|\mathcal{C}''| > 0$
    Choose a minimum weight cycle $C^* \in \mathcal{C}$
    If the cycles in $\mathcal{C}'$ do not generate C^* **Then**
      $\mathcal{C}' \leftarrow \mathcal{C}' \cup C^*$
      Mark the edges of C^* as covered
    $\mathcal{C}'' \leftarrow \mathcal{C}'' \setminus C^*$

Phase 3
  If $|\mathcal{C}'| < m - n + 1$ **Then**
    For each edge e not marked as covered
      Choose a cycle $C \in \mathcal{C} \setminus \mathcal{C}'$ that covers e
      $\mathcal{C}' \leftarrow \mathcal{C}' \cup C$

For a given edge (i, j) and node v, the first phase of the algorithm finds a shortest k-cycle containing (i, j) and v. The algorithm finds this cycle by finding a shortest ℓ-path between i and v and then a shortest $(k - 1 - \ell)$-path between j and v where $\ell < k - 1$. By combining these two paths with (i, j), the algorithm constructs a k-cycle containing (i, j) and v. Using the dynamic program described in [2] to solve the *hop-constrained shortest path problem* (i.e., the shortest ℓ-path problem), this cycle can be constructed in $O(km)$ time. In order to find a shortest k-cycle containing a given edge and node, the algorithm compares cycles created by letting ℓ take on all values $1, 2, 3, \ldots, k - 2$. Therefore, the algorithm requires $O(k^2 m)$ time for each edge-node pair. Since there are $O(mn)$ such pairs, the first phase of the algorithm runs in $O(k^2m^2n)$ time.

The second phase of the algorithm reduces the set of candidate cycles $\mathcal{C}$ to a set of independent cycles $\mathcal{C}'$. Horton [12] recommends treating the cycles in $\mathcal{C}'$ as the rows of a $0-1$ matrix where the columns correspond to edges of the graph. The cycle C^* can then be tested for independence by adding it to the matrix for the current $\mathcal{C}'$ and using Gaussian elimination with arithmetic modulo 2. A row of the matrix can be processed in $O(mr)$ operations where r is the number of rows above the row being processed. The maximum number of rows in the matrix is $m - n + 1 = O(m)$ and there are $O(mn)$ cycles in $\mathcal{C}$. Thus, finding the minimum cost independent set requires $O(m^3n)$ time. If for each edge we maintain a list of the cycles generated in the first phase of the algorithm, we can execute the third phase, if necessary, in $O(m)$ time.

When the algorithm terminates, $\mathcal{C}$ is a set of k-cycles covering the edges of G. The cost of $\mathcal{C}$ provides an upper bound on the cost of an optimal k-cycle cover of G. In Section 9, we discuss the benefits of creating an *augmented bounded cycle basis* by adding the $2n - 1$ shortest cycles from the candidate set found in Phase 1 that were not retained in Phase 2 back to the final candidate set $\mathcal{C}$. We refer to this variant of the bounded cycle basis algorithm as the *augmented basis algorithm*. By expanding the candidate set from $m - n + 1$ to $m + n$ cycles, we obtain better BCCP solutions without significantly increasing the overall running time of the bounded cycle basis heuristic or the time required to solve the resulting set covering problem.

7.2 The Greedy Postman Algorithm

In this section we present an algorithm that derives a set of candidate cycles for a k-cycle cover of a given graph G from a solution to the Chinese postman problem for G. Recall from section 4.2 that a postman tour of a particular graph may contain two types of cycles which cannot be used in k-cycle cover of G: cycles containing more than k edges (long cycles) and "degenerate" cycles of the form $i \to j \to i$. The *greedy postman algorithm* constructs a postman tour of G in a "greedy" manner described below and then generates additional k-cycles as needed to span edges covered by long or degenerate cycles in the postman tour. The k-cycles from the postman tour together with the additional cycles form the candidate set for the k-cycle cover of G. The algorithm can be described in general terms as follows:

> **Algorithm** *Greedy Postman*
> **Input** A graph $G = (V, E' = E \cup F)$ and an integer k
> **Output** A candidate set of k-cycles spanning E
> (1) Find a postman tour T of G using the *greedy tour algorithm*,
> (2) Generate a set of k-cycles $\mathcal{C}_2$ for edges covered by long or degenerate cycles in T using the *partial cover algorithm*
> (3) Return the union of $\mathcal{C}_2$ and the k-cycles of T.

We now describe step (1) above in more detail. CPP is solved by duplicating a minimum cost set of edges of the given graph G such that the resulting graph G' is Eulerian and then finding an Eulerian tour of G' [7]. There may be multiple Eulerian tours of G'. We say that the *greedy tour* algorithm, presented below, is greedy because it creates an Eulerian tour by selecting an edge in G' and taking a shortest (in terms of edge costs) cycle through that edge. All the edges of this cycle are removed from G' and the process is repeated until G' is empty. Thus, step (1) consists of the following parts:

(1a) Solve CPP for G: Given G, find a minimum weight set of edges $F \subseteq E$ that makes the graph $G' = (V, E \cup F)$ Eulerian.

(1b) Find an Eulerian tour of G' using the *greedy tour* algorithm given in the following figure.

> **Algorithm** *Greedy Tour*
> **Input** An Eulerian graph $G' = (V, E)$ and an integer k
> **Output** A set of k-cycles in G'
> $L \leftarrow E, \mathcal{C}_1 \leftarrow \emptyset$
> **While** $L \neq \emptyset$
> Choose any $(i, j) \in L$
> Find a shortest (with respect to cost) path P_{ij} between i and j in $G' \setminus \{(i, j)\}$
> $C_{ij} \leftarrow P_{ij} \cup \{(i, j)\}$
> **If** $3 \leq |C_{ij}| \leq k$ **Then**
> $\mathcal{C}_1 \leftarrow \mathcal{C}_1 \cup C_{ij}$
> $G' \leftarrow G' \setminus C_{ij}, L \leftarrow L \setminus C_{ij}$.
> **Return** $\mathcal{C}_1$

When the greedy tour algorithm terminates, $\mathcal{C}_1$ contains a set of k-cycles covering some of the edges in G. From these edges, form the subset $E_1 \subseteq E$. We apply the *partial cover* algorithm, which is shown in the next figure, to generate a k-cycle for each edge in the set of *uncovered* edges $E_2 = E \setminus E_1$.

The partial cover algorithm generates a k-cycle for each edge (i,j) in E_2. This is done by finding a shortest (with respect to edge costs) $k-1$-path between i and j which does not use the edge (i,j). If there is more than one such path, we select one that maximizes the use of uncovered edges. The aim of this choice is to give us more choices in the final candidate set by generating more than one cycle which contains each edge of E_2. This point is discussed further in Section 9. Such a path can be found by modifying the dynamic program for the hop-constrained shortest path problem (see [2]) so that the length of a path P_{ij} from i to j is defined as $\sum_{e \in P_{ij}} c_e + \frac{1}{k}|P_{ij} \cup E_1|$.

> **Algorithm** *Partial Cover*
> **Input** A graph, $G = (V, E)$, a set of edges $E_2 \subset E$ and an integer k
> **Output** A candidate set of k-cycles spanning E
> $\quad \mathcal{C}_2 \leftarrow \emptyset$
> $\quad$ For all $(i,j) \in E_2$
> $\quad\quad$ Find a shortest (with respect to cost) $(k-1)$–path P_{ij} between i and j in $G \setminus \{(i,j)\}$ that maximizes $|P_{i,j} \cap E_2|$.
> $\quad\quad$ $C_{ij} \leftarrow P_{i,j} \cup \{(i,j)\}, \mathcal{C}_2 \leftarrow \mathcal{C}_2 \cup C_{ij}$.
> $\quad$ **Return** $\mathcal{C}_2$

CPP can be solved in $O(n^3)$ time [7], where the running time is dominated by the time required to solve a minimum weight matching problem in a non-bipartite graph. In the greedy cover algorithm, we seek $O(m)$ shortest paths. Using a Fibonacci heap implementation of Dijkstra's algorithm, we can find each path in $O(m + n \log n)$ time [9]. Thus, the greedy tour algorithm runs in $O(m^2 + mn \log n)$ time.

With the partial cover, we find $O(m)$ shortest k-paths. Since each k-path requires $O(km)$ time to compute (see [2]), the running time for this part of the algorithm is $O(km^2)$. Therefore, we can find $\mathcal{C}_1 \cup \mathcal{C}_2$ in $O(km^2 + mn \log n)$ time.

8 Solving Set Covering Problems

The motivation behind the algorithms presented thus far is to create a set of candidate cycles $\mathcal{C}$ for a BCCP solution, which is used to formulate a set covering problem. We now describe how we solve the set covering problems to obtain a k-cycle cover for a given graph and $\mathcal{C}$.

As noted in Section 5, the set covering problem (SCP) is $\mathcal{NP}$-hard. We use two approaches to solve SCP, both of which are based on formulating the problem as an integer linear program (ILP). Using a straightforward formulation of SCP as an ILP, we define the decision variable x_i to be one if cycle $i \in \mathcal{C}$ is used in the cycle cover, and zero otherwise. If we allow x_j to be a continuous variable such that $0 \leq x_j \leq 1$ for each $j \in \mathcal{C}$, we obtain a linear programming relaxation (LP) of the ILP formulation of the set covering problem. Given an optimal solution $\bar{x}$ to the LP relaxation, we can derive a feasible solution to the ILP by finding a so-called *prime cover*. A prime cover is constructed by rounding up non-integer elements of $\bar{x}$ as necessary, until the vector formed by taking the floor of all the elements of $\bar{x}$ is a feasible solution to the ILP. This approach allows us to find a feasible SCP solution efficiently, but there is no guarantee that this solution is optimal. The results of using this approach are presented in Section 9.2. Since $|\mathcal{C}| = O(m)$ and each cycle contains at most k edges, we can construct the prime cover in $O(km)$ time.

In order to find a minimum cost k-cover contained in a given $\mathcal{C}$, we use a linear programming based branch-and-bound procedure to solve the set covering problems optimally. This approach yields improved solutions over the linear programming relaxation and rounding scheme, but the worst case complexity of the branch-and-bound algorithm is exponential in the number of cycles in $\mathcal{C}$. Section 9.3 gives the results of using this approach.

9 Empirical Results

In this section, we report results obtained by using the algorithms described in Section 7 to find k-cycle covers of randomly-generated graphs. We generated five such graphs for each $n = 10, 15, 20, 25, 30$, and 35. For each graph, we tested the algorithms of Section 7 with values of $k = 6, 7$ and 8. The graphs were generated using the *plane_miles* routine in the Stanford GraphBase. This routine creates planar, Euclidean graphs by randomly selecting cities from the Rand McNally & Company's *Standard Highway Mileage Guide*. Edges between vertices are assigned lengths representing distances between cities in miles.

All computations were performed on a Sun Microsystems SPARCstation 20 Model 16 workstation with a 60 Mhz SuperSparc processor with SuperCache and 32 megabytes of memory. The three algorithms for generating sets of candidate cycles and an algorithm for solving CPP were implemented in C. In Section 9.2 we report the results for each algorithm when the SCP instances were solved via linear programming relaxation. The LP relaxations were solved using CPLEX Linear Optimizer 6.0 and the prime cover algorithm was implemented in MATLAB version 4.2c. In Section 9.3 we report the results of the algorithms when the SCP instances were solved optimally using CPLEX Mixed Integer Solver. Finally, we obtain k-cycle covers by taking the union of the candidate sets produced by the algorithms and solving the resulting set covering problems optimally. This approach produces superior results at the price of increased running times.

9.1 Measuring Solution Quality

Recall from Section 4.2 that a solution to the Chinese postman problem for a given graph G is a lower bound on the cost of an optimal BCCP solution for G. The quality of a solution returned by any of the various algorithms presented in this chapter for a particular instance of BCCP is measured in terms of the relative error with respect to the CPP lower bound. The relative error is computed as follows. Solve CPP for G and let CPP(G) be the value of the objective function. Apply one of the algorithms presented in this chapter to G to generate a set of candidate cycles $\mathcal{C}$ to create an instance of the set covering problem. Find a feasible solution for the SCP instance and let SCP($\mathcal{C}$) be the value of the cost associated with the solution.

This solution could either be an exact solution obtained by a branch-and-bound procedure or an "approximate" solution found by solving the LP relaxation of SCP and then constructing a prime cover. The relative error is given by

$$\frac{\mathrm{SCP}(\mathcal{C}) - \mathrm{CPP}(G)}{\mathrm{CPP}(G)}.$$

9.2 Linear Programming Results

In Table I, we report the average relative error of each of the algorithms for each value of $n = 10, 15, 20, 25, 30$, and 35 when the prime cover procedure is used to solve the SCP instances generated for each problem instance. We present the average relative error for each algorithm for the 5 graphs with $n = 10$, then the average relative error for the 5 graphs with $n = 15$ and so on.

Table I. Results Using Linear Programming Relaxation and Rounding

$k = 6$		Average Relative Error vs. CPP Lower Bound		
n	m	Greedy Heuristic	Cycle Basis	Augmented Cycle Basis
10	6	1.82%	4.57%	13.23%
15	6	6.28%	6.63%	9.72%
20	6	8.52%	22.47%	16.82%
25	6	11.77%	16.12%	11.55%
30	6	5.94%	32.58%	29.28%
35	6	10.29%	42.44%	29.02%

$k = 7$		Average Relative Error vs. CPP Lower Bound		
n	m	Greedy Heuristic	Cycle Basis	Augmented Cycle Basis
10	7	1.82%	4.57%	13.23%
15	7	4.75%	6.63%	9.72%
20	7	8.43%	22.47%	16.82%
25	7	8.95%	16.12%	11.55%
30	7	5.94%	32.58%	29.28%
35	7	10.28%	42.44%	29.02%

$k = 8$		Average Relative Error vs. CPP Lower Bound		
n	m	Greedy Heuristic	Cycle Basis	Augmented Cycle Basis
10	8	1.82%	4.57%	13.23%
15	8	3.56%	6.63%	9.72%
20	8	7.18%	22.47%	16.82%
25	8	6.76%	16.12%	11.55%
30	8	4.95%	32.58%	29.28%
35	8	7.67%	42.44%	29.02%

Using LP relaxation, the greedy algorithm was clearly superior to the cycle basis heuristics in this set of experiments. No rounding was necessary for the LP relaxation of the SCP for the candidate sets generated by the greedy algorithm. In 14 of the 90 problem instances, the greedy algorithm found a k-cycle cover with same cost as the CPP solution. Since CPP provides a lower bound on BCCP, we conclude that an optimal k-cycle cover was found in those 14 cases.

Since the greedy postman algorithm uses a postman tour as a starting point for building a k-cycle cover, its solutions tend to be *minimal* meaning that if any of the candidates are removed, $\mathcal{C}$ no longer contains a cycle cover. Consequently, the candidate sets contained very few, if any, choices of cycles for any given edge in the graph. The cycle basis algorithms produced "richer" candidate sets. For a given graph, the basis tends to provide at least two, and sometimes more, cycles that span any given edge. This leads to a high percentage of variables with non-integer values in the optimal solution of the LP relaxation of the corresponding SCP. Table II shows the average number of cycles covering each edge for the SCP instances generated by the greedy and cycle basis algorithms. From Table II, we can see that the average number of cycles covering each edge in the SCP instances generated by the greedy postman algorithm is close to one, whereas the average number of cycles covering each edge for this algorithm is close to two.

Table II. Average number of Cycles Per Edge

	Greedy			Basis
n	k=6	k=7	k=8	k=6,7,8
10	1.16	1.16	1.16	1.71
15	1.24	1.21	1.20	1.81
20	1.27	1.26	1.24	1.88
25	1.36	1.28	1.22	1.89
30	1.27	1.27	1.22	1.88
35	1.32	1.29	1.25	1.92

Average total running times for the three algorithms, measured in seconds of CPU time, along with running times for the various stages of each algorithm are reported in the Tables III, IV, and V. Note that these tables only report times for $k = 8$. The times were similar for $k = 6$ and 7.

Table III shows running times for the greedy algorithm. As expected from the complexity analysis, this algorithm is much faster than the bounded cycle basis heuristic. Tables IV and V show the running times for the bounded cycle basis and augmented basis algorithms, respectively. These tables include a column indicating the average time required to round the continuous variables from the optimal LP solution to create a prime cover. With both types of algorithms, most of the computation time is spent on generating the candidate sets.

Table III. CPU Times for Greedy Algorithm

Greedy Algorithm					
$k = 8$	Mean Time in CPU seconds				
n	Solve CPP	Generate $\mathcal{C}$	Solve LP	Prime Cover	Total
10	0.000	0.000	0.006	0.038	0.044
15	0.000	0.020	0.002	0.036	0.058
20	0.000	0.054	0.000	0.040	0.094
25	0.000	0.090	0.004	0.042	0.136
30	0.000	0.168	0.006	0.060	0.234
35	0.002	0.264	0.010	0.054	0.330

Table IV. CPU Times for Basis Algorithm

Cycle Basis Algorithm				
$k = 8$	Mean Time in CPU seconds			
n	Find Basis	Solve LP	Prime Cover	Total
10	0.082	0.004	0.036	0.122
15	0.810	0.006	0.036	0.852
20	3.380	0.018	0.046	3.444
25	7.560	0.024	0.056	7.640
30	16.354	0.022	0.084	16.460
35	37.196	0.028	0.080	37.304

Table V. CPU Times for Augmented Basis Algorithm

Augmented Cycle Basis Algorithm				
$k = 8$	Mean Time in CPU seconds			
n	Find Basis	Solve LP	Prime Cover	Total
10	0.086	0.014	0.036	0.136
15	0.824	0.016	0.050	0.890
20	3.412	0.012	0.052	3.476
25	7.644	0.024	0.064	7.732
30	16.528	0.018	0.088	16.634
35	37.586	0.030	0.094	37.710

9.3 Branch-and-Bound Results

In this section, we present results obtained by solving the set covering problem instances to optimality with an LP-based branch-and-bound procedure. Specifically, the set covering problems were solved using CPLEX. Table VI gives the average relative error of each of the algorithms across a range of values of n. The time for the branch-and-bound procedure used by CPLEX is reported in CPU seconds in Table VIII. Use of the optimal SCP solutions produced dramatically improved results for the cycle basis algorithm and led to little or no improvement in the quality of the solutions returned by the greedy postman algorithm.

Table VI. Results Using Branch-and-Bound

$k = 6$	Average Relative Error vs. CPP Lower Bound				
n	Greedy	Cycle Basis (CB)	Augmented Basis (AB)	Greedy + CB	Greedy + AB
10	1.82%	4.57%	4.44%	0.56%	0.56%
15	6.28%	5.18%	5.02%	2.85%	2.71%
20	8.52%	6.35%	5.58%	2.67%	2.20%
25	11.77%	3.34%	2.99%	2.59%	2.59%
30	5.94%	4.82%	4.60%	2.98%	2.71%
35	10.29%	3.69%	3.41%	2.49%	2.38%

$k = 7$	Average Relative Error vs. CPP Lower Bound				
n	Greedy	Cycle Basis (CB)	Augmented Basis (AB)	Greedy + CB	Greedy + AB
10	1.82%	4.57%	4.44%	0.56%	0.56%
15	4.75%	5.18%	5.02%	2.26%	2.12%
20	8.43%	6.35%	5.58%	2.58%	2.11%
25	8.95%	3.34%	2.99%	2.53%	2.53%
30	5.94%	4.82%	4.60%	2.98%	2.71%
35	10.28%	3.69%	3.41%	2.49%	2.38%

$k = 8$	Average Relative Error vs. CPP Lower Bound				
n	Greedy	Cycle Basis (CB)	Augmented Basis (AB)	Greedy + CB	Greedy + AB
10	1.82%	4.57%	4.44%	0.56%	0.56%
15	3.56%	5.18%	5.02%	1.07%	1.07%
20	7.18%	6.35%	5.58%	1.90%	1.67%
25	6.76%	3.34%	2.99%	2.42%	2.37%
30	4.95%	4.82%	4.60%	2.63%	2.37%
35	7.67%	3.69%	3.41%	2.14%	2.03%

As expected, the augmented cycle basis gave better solutions when we solved the SCP ILPs to optimality. We can see from Table VIII that the additional cycles lead to a significant percentage increase in the time required to solve the set covering problems. Since the times shown for the augmented basis SCPs in Table VIII are reasonable, one might consider using more than $m + n$ cycles in the augmented basis. We have observed, however, that the solution times increase dramatically as we increase the number of cycles in the augmented basis. CPLEX required several hours and, in some cases, days to solve the set covering ILPs when we used all mn candidate cycles from Phase 1 of the bounded cycle basis heuristic without a significant improvement in solution quality.

Notice that neither the greedy postman nor the augmented basis algorithms gave the best solution in all cases. By combining the candidate sets from both algorithms, we were able to substantially improve the quality of our solutions by providing a richer set of choices for the set covering problem. The last two columns of Table VI give the results for using the union of the candidate sets produced by the greedy postman algorithm and either the bounded cycle basis or augmented basis heuristic. The improvements in the relative error come at the price of increased running times for the branch-and-bound procedures as indicated by Table VIII.

The bound on cycle length had only a marginal effect, as the overall solution quality changed only slightly as we increased k. As a way of measuring how sensitive the solutions were to the value of k, we ran the heuristics with $k = k^*$; k^* turned out to be three for all 30 graphs. The results from the greedy postman and augmented basis heuristics using the LP relaxation and prime cover to solve SCP and from using the union of the two candidate sets and solving SCP optimally are shown in Table VII.

Table VII. Solution Quality with $k = k^*$

	Greedy LP Relax		Augmented Basis LP Relax		Greedy + AB ILP	
10	11.76%	0.152	1.71%	0.060	1.71%	0.026
15	9.42%	0.068	6.63%	0.118	5.18%	0.114
20	6.99%	0.090	12.20%	0.272	3.04%	0.292
25	7.05%	0.160	28.46%	0.494	2.59%	0.570
30	7.93%	0.246	30.66%	0.842	4.22%	0.962
35	9.05%	0.344	43.71%	1.376	2.61%	1.676

Table VIII. CPU Seconds Using Branch-and-Bound to Solve SCP

$k = 6$	CPU Seconds to Solve SCP				
n	Greedy	Cycle Basis (CB)	Augmented Basis (AB)	Greedy + CB	Greedy + AB
10	0.008	0.008	0.044	0.018	0.044
15	0.000	0.004	0.066	0.016	0.092
20	0.004	0.016	0.060	0.026	0.186
25	0.012	0.018	0.374	0.028	1.210
30	0.004	0.028	0.570	0.058	5.546
35	0.006	0.032	1.846	0.258	104.984

$k = 7$	CPU Seconds to Solve SCP				
n	Greedy	Cycle Basis (CB)	Augmented Basis (AB)	Greedy + CB	Greedy + AB
10	0.006	0.006	0.040	0.012	0.046
15	0.004	0.008	0.054	0.020	0.074
20	0.000	0.010	0.070	0.034	0.198
25	0.004	0.014	0.404	0.058	1.018
30	0.002	0.028	0.596	0.056	5.452
35	0.008	0.032	1.838	0.614	62.634

$k = 8$	CPU Seconds to Solve SCP				
n	Greedy	Cycle Basis (CB)	Augmented Basis (AB)	Greedy + CB	Greedy + AB
10	0.006	0.006	0.036	0.016	0.046
15	0.004	0.010	0.056	0.022	0.046
20	0.008	0.022	0.064	0.024	0.112
25	0.004	0.016	0.398	0.070	0.948
30	0.012	0.022	0.608	0.050	7.018
35	0.004	0.036	1.880	0.480	64.850

9.4 Experiments with Sparser Graphs

In this subsection, we present results from repeating the preceding experiments with a set of sparser graphs. These graphs were generated in a similar manner to those previously described. In this data set, however, the value for k^* ranged from four to eight. Five of the problem instances were infeasible for k less than or equal to 7 and another five were infeasible for $k = 7$. Table IX shows the average relative error with respect to the CPP lower bound and average total solution or greedy postman and augmented basis algorithms using the LP relaxation with rounding to solve SCP, as well as the results for using the union of the two candidate sets and finding optimal SCP solutions.

Table IX. Results for Sparser Graphs

		Greedy LP Relax		Augmented Basis LP Relax		Greedy + AB ILP	
n	k^*	Rel. Err.	CPU Seconds	Rel. Err.	CPU Seconds	Rel. Err.	CPU Seconds
10	6	10.78%	0.010	16.48%	0.070	7.34%	0.054
15	6	8.99%	0.014	13.04%	0.088	6.02%	0.136
20	6	10.82%	0.036	15.00%	0.248	4.01%	0.431
25	6	11.37%	0.050	12.59%	0.494	5.89%	1.286
30	6	11.34%	0.070	17.60%	0.638	6.90%	1.424
35	6	9.29%	0.114	32.28%	1.124	5.51%	135.553
10	7	10.78%	0.014	16.48%	0.072	7.34%	0.054
15	7	12.18%	0.016	23.12%	0.144	8.24%	0.217
20	7	5.85%	0.040	15.25%	0.440	2.37%	0.699
25	7	9.68%	0.078	14.83%	0.902	5.78%	1.311
30	7	8.51%	0.106	18.16%	1.778	6.04%	2.696
35	7	9.83%	0.144	31.35%	2.538	5.27%	569.580
10	8	7.53%	0.014	16.48%	0.062	4.09%	0.048
15	8	9.60%	0.024	23.11%	0.156	7.39%	0.210
20	8	7.98%	0.042	19.07%	0.506	4.68%	0.694
25	8	11.82%	0.086	16.33%	1.096	4.63%	1.354
30	8	7.50%	0.128	20.08%	2.332	5.73%	4.626
35	8	8.50%	0.176	31.00%	3.424	5.29%	164.962

The average relative errors for this set of graphs were all less than 9%, but overall significantly larger than the best results for the previous set of graphs. Each "degenerate" cycle in the greedy postman tour is replaced by a proper k-cycle in the greedy postman tour. This leads to an increase in the cost of the greedy tour over the cost of the postman tour from which we get our lower bound. Since there were on average fewer "replacement" cycles to choose from in the sparser graphs, this increase tended to be much greater, 21.15% on average, than for the first set of graphs where the average increase was 5.3%. Thus, the increase in relative error for these graphs may be an artifact of the quality of the CPP lower bound on BCCP.

Table X. Results for Sparser Graphs with $k = k^*$

	Greedy LP Relax		Augmented Basis LP Relax		Greedy + AB ILP	
n	Rel. Err.	CPU Seconds	Rel. Err.	CPU Seconds	Rel. Err.	CPU Seconds
10	10.78%	0.014	18.03%	0.040	7.34%	0.038
15	13.96%	0.026	22.95%	0.084	10.34%	0.106
20	10.39%	0.044	19.69%	0.302	5.19%	0.348
25	8.05%	0.084	15.56%	0.740	4.93%	0.820
30	11.32%	0.132	18.79%	1.214	8.48%	1.002
35	10.65%	0.178	29.03%	1.97	6.81%	1.984

10 Conclusions

This paper details the study of two heuristic algorithms for generating candidate solutions to the bounded cycle cover problem for a given graph. Computational testing indicates that for a modest cost in terms of CPU time, we can obtain near optimal solutions by running both algorithms on a given graph, taking the union of the candidate cycle sets and solving the resulting SCP optimally. In theory, however, the worst case running time of this procedure grows exponentially with the size of the graph. Our experiments indicate there is a tradeoff between solution quality and running time. Larger candidate sets lead to better solutions, but introduce more non-integer values into the solutions of the LP relaxations of the set covering problems, which in turn leads to more branching in the branch-and-bound procedure. In the context of enhancing network survivability with self-healing rings, BCCP does not need to be solved in "real time". If CPU time is a critical factor, for example in a scenario where the cost of building self-healing rings is being used to evaluate different proposed topologies for a network which has not yet been built, our results show that using the candidate set from the greedy postman algorithm with the linear programming relaxation of SCP, followed by the rounding procedure (if necessary) is the best method to generate acceptable solutions quickly.

An interesting direction for further work on this problem would be to attempt to find a better lower bound than CPP for the bounded cycle cover problem (although it appears from Table VI that the CPP bound is quite good for the denser graphs). Further investigation may also reveal a method for determining *a priori* which single algorithm, postman or cycle-basis-based, is likely to give the best result for a particular class of graphs.

Acknowledgments

The authors would like to thank the area editor and two anonymous reviewers for their thoughtful comments and suggestions which have greatly improved the paper, and Herbert Fleischner, Luis Goddyn, and Cun-Quan Zhang for their helpful discussions regarding the complexity of MCCP for planar graphs. We would also like to thank Judy and Michael Olinick for suggesting many stylistic improvements and corrections. This research was supported in part by ONR contract N00014-91-J-1241.

References

[1] B. Alspach, L. Goddyn, and C-Q. Zhang. Graphs with the circuit cover property. *Transactions of the American Mathematical Society*, 344(1):131–153, 1994.

[2] A. Balakrishnan and K. Altinkemer. Using a hop-constrained model to generate alternative communication network designs. *ORSA Journal on Computing*, 4(2):192–205, 1992.

[3] D.M. Chickering, D. Geiger, and D. Heckerman. On finding a cycle basis with a shortest maximal cycle. *Information Processing Letters*, 54(1):55–58, 1995.

[4] S. Cosares, D.N. Deutsch, I. Saniee, and O.J. Wasem. SONET Toolkit: A decision support system for designing robust and cost-effective fiber-optic networks. *Interfaces*, 25(1):20–40, 1995.

[5] S. Cosares and I. Saniee. An optimization problem related to balancing loads on SONET rings. *Telecommunication Systems*, 3(2):165–181, 1994.

[6] M. Dell'Amico, M. Labbé, and F. Maffioli. Exact solution of the SONET ring loading problem. *Operation Research Letters*, 25(3):119–129, 1999.

[7] J. Edmonds and E.L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5(1):88–124, 1973.

[8] H. Fleischner. Eulersche linien und kreisüberdeckungen, die vorgegebene durchgänge in den kanten vermeiden. *J. Combinatorial Theory*, 29:145–167, 1980.

[9] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of ACM*, 34:596–615, 1987.

[10] M. Garey and D. Johnson. *Computers and intractability : a guide to the theory of NP-completeness.* W.H. Freeman, San Francisco, 1979.

[11] M. Guan and H. Fleischner. On the minimum weighted cycle-covering problem for planar graphs. *Ars Combinatorica*, 20:61–68, 1985.

[12] J. Horton. A polynomial-time algorithm to find the shortest cycle basis of a graph. *SIAM Journal on Computing*, 16(2):358–366, 1987.

[13] A. Itai, R.J. Lipton, C.H. Papadimitriou, and M. Rodeh. Covering graphs by simple circuits. *SIAM Journal on Computing*, 10(4):746–750, 1981.

[14] A. Itai and M. Rodeh. Covering a graph by circuits. In *Automata, Languages and Programming, Lecture Notes in Computer Science, no. 62.*, pages 289–299, Berlin, 1978. Springer-Verlag.

[15] J.L. Kennington, V.S.S Nair, and M.H. Rahman. Optimization based algorithms for finding minimal cost ring covers in survivable networks. *Computational Optimization and Applications*, 14(2):219–230, 1999.

[16] J.L. Kennington and M.H. Rahman. The modular SONET network ring cover problem: Algorithms and an empirical analysis. Technical Report 98-CSE-10, Department of Computer Science and Engineering, Southern Methodist University, 1998.

[17] M. Laguna. Clustering for the design of SONET rings in interoffice telecommunications. *Management Science*, 40(11):1533–1541, 1994.

[18] C-C. Shyur, Y-M. Wu, and C-H. Chen. The capacity comparison and cost analyses for SONET self-healing ring networks. *IEICE Transactions on Communications*, E77-B(2):218–225, 1994.

[19] J.B. Slevinsky, W.D. Grover, and M.H. MacGregor. An algorithm for survivable network design employing multiple self-healing rings. In *GLOBECOM '93*, pages 1568–1573, New York, 1993. IEEE Press.

[20] C. Thomassen. On the complexity of finding a minimum cycle covers of graphs. *SIAM Journal on Computing*, 26(3):675–677, 1997.

[21] T-H. Wu. *Fiber Network Service Survivability.* Artech House, Inc., Norwood, MA, 1992.

[22] C-Q. Zhang. *Integer Flows and Cycle Covers of Graphs.* Marcel Dekker, New York, 1997.