Chapter 21
Approximating the Minimum Equivalent Digraph

Samir Khuller * Balaji Raghavachari † Neal Young ‡

Abstract
The MEG (minimum equivalent graph) problem is the following: "Given a directed graph, find a smallest subset of the edges that maintains all reachability relations between nodes." The MEG problem is NP-hard; this paper gives an approximation algorithm achieving a performance guarantee of about 1.64 in polynomial time. We give a modification that improves the performance guarantee to about 1.61. The algorithm achieves a performance guarantee of 1.75 in the time required for transitive closure.

The heart of the MEG problem is the minimum SCSS (strongly connected spanning subgraph) problem — the MEG problem restricted to strongly connected digraphs. For the minimum SCSS problem, the paper gives a practical, nearly linear-time implementation achieving a performance guarantee of 1.75.

The algorithm and its analysis are based on the simple idea of contracting long cycles. The analysis applies directly to 2-EXCHANGE, a general "local improvement" algorithm, showing that its performance guarantee is 1.75.

Keywords: directed graph, reachability, approximation algorithm, strong connectivity, local improvement.

1 Introduction
Connectivity is fundamental to the study of graphs and graph algorithms. Recently, many approximation algorithms for finding subgraphs that meet given connectivity requirements have been developed [1, 9, 11, 15, 16, 24]. These results provide practical approximation algorithms for NP-hard network-design problems via an increased understanding of connectivity properties.

Until now, the techniques developed have been applicable only to undirected graphs. We consider a basic network-design problem in directed graphs [2, 12, 13, 18] which is as follows: given a digraph, find a smallest subset of the edges (forming a minimum equivalent graph (MEG)) that maintains all reachability relations of the original graph.

When the MEG problem is restricted to graphs which are strongly connected, we call it the minimum SCSS (strongly connected spanning subgraph) problem. When the MEG problem is restricted to acyclic graphs we call it the acyclic MEG problem. The MEG problem reduces in linear time [5] to a single acyclic problem given by the so-called "strong component graph", together with one minimum SCSS problem for each strong component (given by the subgraph induced by that component). Furthermore, approximating the MEG problem is linear-time equivalent to approximating both restricted versions.


The related problem of finding a transitive reduction of a digraph — a smallest set of edges yielding the same reachability relations was studied by
Aho, Garey and Ullman [2]. Transitive reduction differs from the MEG problem in that the edges in the transitive reduction are not required to be in the original graph. However, the transitive reduction problem decomposes just like the MEG problem into acyclic and strongly connected instances. For any strongly connected instance, a transitive reduction is given by any Hamilton cycle through the vertices. For an acyclic instance, the transitive reduction is unique and, as Aho, Garey and Ullman observe, is equivalent to the MEG problem: it consists of those edges \((u, v)\) for which there is no alternate path from \(u\) to \(v\). In fact, Aho, Garey and Ullman show that the transitive reduction problem is equivalent to the transitive closure problem. Thus, the acyclic MEG problem reduces to transitive closure.

The acyclic MEG problem can be solved in polynomial time, whereas the minimum SCSS problem is NP-hard [8]. Consequently, this paper focuses on approximation algorithms for the minimum SCSS problem. By the observations of the preceding paragraphs, the performance guarantees obtained for the minimum SCSS problem carry over to the general MEG problem with the overhead of solving a single instance of transitive closure.

1.1 Our Results. Given a strongly connected graph, our basic algorithm finds as long a cycle as it can, contracts the cycle, and recurses. The contracted graph remains strongly connected. When the graph finally collapses into a single vertex, the algorithm returns the set of edges contracted during the course of the algorithm as the desired SCSS.

The algorithm achieves a performance guarantee of any constant greater than \(\pi^2/6 \approx 1.645\) in polynomial time. We give a nearly linear-time version that achieves a performance guarantee of 1.75. We give examples showing lower bounds on the performance guarantees of the algorithm. For the general algorithm, the lower bounds are slightly above 1.5. For the nearly linear-time version, the lower bound is 1.75, matching the upper bound.

The performance guarantee analysis extends directly to a simple “local improvement” algorithm called 2-Exchange. 2-Exchange starts with the given digraph and performs the following local improvement step as long as it is applicable: find two edges in the current graph that can be replaced by one edge from the original graph, maintaining strong connectivity. Similar local-improvement algorithms are natural candidates for many optimization problems but often elude analysis. We prove that the performance guarantee of 2-Exchange is 1.75.

A natural improvement to the cycle-contraction algorithm is to modify the algorithm to solve the problem optimally once the contracted graph has no cycles longer than a given length \(c\). For instance, for \(c = 3\), this modification improves the performance guarantee to \(\pi^2/6 - 1/36 \approx 1.617\). We use SCSS\(_c\) to denote the minimum SCSS problem restricted to digraphs with no cycle longer than \(c\). The minimum SCSS\(_2\) problem is trivial. The minimum SCSS\(_3\) problem is at least as hard as bipartite matching; in fact we can show that it has a polynomial-time algorithm. However, potential improvement in this direction is limited: we show that the minimum SCSS\(_5\) problem is NP-hard. In fact, we show that the minimum SCSS\(_{17}\) problem is SNP-hard. This precludes the possibility of a polynomial-time approximation scheme, assuming \(\mathsf{P} \neq \mathsf{NP}\) [4].

1.2 Other Related Work. The union of any incoming branching and any outgoing branching from the same root yields an SCSS with at most \(2n - 2\) edges (where \(n\) is the number of vertices in the graph). This is a special case of the algorithm given by Frederickson and JáJá [6] that uses minimum weight branchings to achieve a performance guarantee of 2 for weighted graphs. Since any SCSS has at least \(n\) edges, this yields a performance guarantee of 2 for the SCSS problem.

Any minimal SCSS (one from which no edge can be deleted) has at most \(2n - 2\) edges and also yields a performance guarantee of 2. A linear-time algorithm finding a minimal SCSS is given by Simon [21]. A parallel algorithm is given by Gibbons, Karp, Ramachandran, Soroker and Tarjan [10].

A related problem in undirected graphs is to
find a smallest subset of the edges forming a biconnected (respectively bridge-connected (i.e., 2-edge-connected)) spanning subgraph of a given graph. These problems are NP-hard. Khuller and Vishkin [15] give a DFS-based algorithm that achieves a factor of \( \frac{5}{3} \) for biconnectivity and \( \frac{3}{2} \) for bridge-connectivity. Garg, Santosh and Singla [9] subsequently improve the approximation factors, using a similar approach, to \( \frac{3}{2} \) and \( \frac{5}{4} \), respectively. None of these methods appear to extend to the minimum SCSS problem.

**Undirected** graphs having bounded cycle length have bounded tree width. Arnborg, Lagergren and Seese [3] have shown that many NP-hard problems, including the minimum biconnected-spanning-subgraph problem, have polynomial-time algorithms when restricted to such graphs.

## 2 Preliminaries

To **contract** a pair of vertices \( u, v \) of a digraph is to replace \( u \) and \( v \) (and each occurrence of \( u \) or \( v \) in any edge) by a single new vertex, and to delete any subsequent self-loops and multi-edges. Each edge in the resulting graph is identified with the corresponding edge in the original graph or, in the case of multi-edges, the single remaining edge is identified with any one of the corresponding edges in the original graph. To contract an edge \( (u, v) \) is to contract the pair of vertices \( u \) and \( v \). To contract a set \( S \) of pairs of vertices in a graph \( G \) is to contract the pairs in \( S \) in arbitrary order. The contracted graph is denoted by \( G/S \). Contracting an edge is also analogously extended to contracting a set of edges.

Let \( \text{OPT}(G) \) be the minimum size of any subset of the edges that strongly connects \( G \). In general, the term "cycle" refers only to simple cycles.

## 3 Lower Bounds on \( \text{OPT}(G) \)

We begin by showing that if a graph has no long cycles, then the size of any SCSS is large.

**Lemma 3.1. (Cycle Lemma)** For any directed graph \( G \) with \( n \) vertices, if a longest cycle of \( G \) has length \( \ell \), then

\[
\text{OPT}(G) \geq \frac{C}{C-1}(n-1).
\]

**Proof.** Starting with a minimum-size subset that strongly connects the graph, repeatedly contract cycles in the subset until no cycles are left. Observe that the maximum cycle length does not increase under contractions. Consequently, for each cycle contracted, the ratio of the number of edges deleted to the decrease in the number of vertices is at least \( \frac{C}{C-1} \). Since the total decrease in the number of vertices is \( n-1 \), at least \( \frac{C}{C-1}(n-1) \) edges are deleted.

Note that the above lemma gives a lower bound which is existentially tight. For all values of \( C \), there exist graphs for which the bound given by the lemma is equal to \( \text{OPT}(G) \). Also note that \( C \) has a trivial upper bound of \( n \) and, using this, we get a lower bound of \( n \) for \( \text{OPT}(G) \), which is the known trivial lower bound.

**Lemma 3.2. (Contraction Lemma)** For any directed graph \( G \) and set of edges \( S \),

\[
\text{OPT}(G) \geq \text{OPT}(G/S).
\]

**Proof.** Any SCSS of \( G \), contracted around \( S \) (treating the edges of \( S \) as pairs), is an SCSS of \( G/S \).

## 4 Cycle-Contraction Algorithm

The algorithm is the following. Fix \( k \) to be any positive integer.

**CONTRACT-CYCLES\(_k\)(G) —**

1. **for** \( i = k, k-1, k-2, \ldots, 2 \)
2. **while** the graph contains a cycle
   - with at least \( i \) edges
3. Contract the edges on such a cycle.
4. **return** the contracted edges

In Section 6, we will show that the algorithm can be implemented to run in \( O(m\alpha(m, n)) \) time for the case \( k = 3 \) and in polynomial time for any fixed value of \( k \). It is clear that the edge set returned by the algorithm strongly connects the graph. The following theorem establishes an upper bound on the number of edges returned by the algorithm.

**Theorem 4.1.** CONTRACT-CYCLES\(_k\)(G) returns at most \( c_k \cdot \text{OPT}(G) \) edges, where

\[
\frac{\pi^2}{6} \leq c_k \leq \frac{\pi^2}{6} + \frac{1}{(k-1)^2}.
\]
7.3.1. Let $n$ vertices remain in the contracted graph after contracting cycles with $i$ or more edges ($i = k, k-1, \ldots, 2$).

How many edges are returned? In contracting cycles with at least $k$ edges, at most $\frac{k-1}{k} \cdot \ell$ edges are contributed. The number of edges returned is thus at most

$$\sum_{i=k}^{k-1} \left( \frac{k-1}{k} \right)^{i-1} (n-i)$$

Using the identity from [17, p. 755],

$$\sum_{i=1}^{\infty} \frac{1}{i} = \ln n + \gamma,$$

we get

$$\sum_{i=k}^{k-1} \left( \frac{k-1}{k} \right)^{i-1} (n-i) \leq \frac{(k-1)^2}{k} \sum_{i=1}^{\infty} \frac{1}{i} \leq \frac{(k-1)^2}{k} \ln n.$$ 

By Lemmas 3.1 and 3.2, $OPT(G)$ is at most $\frac{k-1}{k} \cdot \ell$ edges.

**4.1 Lower Bounds on the Performance.**

In this section, we present lower bounds on the performance ratio of $\text{CONTRACT-CYCLES}(G)$.

The performance ratio of $\text{CONTRACT-CYCLES}(G)$, the algorithm that contracts the graph with only $k$ cycles, is a lower bound on the performance guarantee for $\text{CONTRACT-CYCLES}(G)$. As $k$ approaches $\infty$, the lower bound tends to $1.5$.

Table 1: Bounds on the performance guarantee of the algorithm for small values of $k$ and in the limit as $k \to \infty$.

<table>
<thead>
<tr>
<th>$k$</th>
<th>Upper Bound</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.33</td>
<td>1.75</td>
</tr>
<tr>
<td>4</td>
<td>1.93</td>
<td>1.50</td>
</tr>
<tr>
<td>5</td>
<td>1.75</td>
<td>1.45</td>
</tr>
<tr>
<td>$\infty$</td>
<td>1.25</td>
<td>1.00</td>
</tr>
</tbody>
</table>
of \( E' \), so \( k \)-EXCHANGE runs in polynomial time. The following theorem shows that the approximation factor achieved by 2-EXCHANGE is 1.75.

**Theorem 5.1.** The performance guarantee of 2-EXCHANGE is 1.75.

**Proof.** First we show that the performance guarantee is at most 1.75. Let \( E' \) be the set of edges returned by 2-EXCHANGE\((G = (V, E))\). Run CONRACT-CYCLES\(_3\) on the graph \( G' = (V, E') \). Let \( H \) be the set of edges contracted during the first iteration when cycles of at least three edges are contracted. The resulting graph \( G'/H \) is strongly connected and has only 2-cycles. Such a graph has a tree-like structure. In particular, an edge \((u, v)\) is present iff the reverse edge \((v, u)\) is present.

The important observation is that \( G/H \) is equivalent to \( G'/H \). Clearly \( G'/H \) is a subgraph of \( G/H \); to prove the converse, suppose that some edge \((u, v)\) of \( G/H \) was not in \( G'/H \). Consider adding edge \((u, v)\) to \( G'/H \). By the structure of \( G'/H \), \( u \) and \( v \) are not adjacent in \( G'/H \) and for each edge on the path from \( v \) to \( u \) the reverse edge is also in \( G'/H \). If \((u, v)\) is added to \( G'/H \), these (at least two) reverse edges can be deleted from \( G'/H \) without destroying the strong connectivity of \( G'/H \). Consequently, the original edge in \( G \) corresponding to \((u, v)\) can be added to \( G' \) and the original edges in \( G' \) corresponding to the reverse edges can be deleted from \( G' \) without destroying the strong connectivity of \( G' \). This contradicts the fact that \( E' \) was output by 2-EXCHANGE\((G)\), since \( E' \) is eligible for an improvement step.

Next consider executing the procedure call CONRACT-CYCLES\(_3\)(\(G\)). Since \( G/H \) is equivalent to \( G'/H \), the sequence of cycles chosen in the first iteration of CONRACT-CYCLES\(_3\)(\(G'\)) could also be chosen by the first iteration of CONRACT-CYCLES\(_3\)(\(G\)). Similarly, the second iteration in CONRACT-CYCLES\(_3\)(\(G'\)) could be mimicked by CONRACT-CYCLES\(_3\)(\(G\)), in which case CONRACT-CYCLES\(_3\)(\(G\)) would return the same edge set as CONRACT-CYCLES\(_3\)(\(G'\)). Since \( E' \) is minimal (otherwise an improvement step applies), the edge set returned is exactly \( E' \). Thus, the upper bound on the performance guarantee of CONRACT-CYCLES\(_3\) from Theorem 4.1 is inherited by 2-EXCHANGE.

For the lower bound on the performance guarantee, given the graph in Fig. 2, 2-EXCHANGE can choose a number of edges arbitrarily close to 1.75 times the minimum. \( \square \)

**6 Implementation Details**

For any fixed \( k \), CONRACT-CYCLES\(_k\) can be implemented in polynomial time using exhaustive search to find long cycles. For instance, if a cycle of size at least \( k \) exists, one can be found in polynomial time as follows. For each simple path \( P \) of \( k - 1 \) edges, check whether a path from the head of \( P \) to the tail exists after \( P \)'s internal vertices are removed from the graph. If \( k \) is even, there are at most \( m^{k/2} \) such paths; if \( k \) is odd, the number is at most \( n \min(k-1)/2 \). It takes \( O(m) \) time to decide if there is a path from the head of \( P \) to the tail of
For the first iteration of the for loop, we may have $O(n)$ iterations of the while loop. Since the first iteration is the most time consuming, the algorithm can be implemented in $O(n \cdot m^{1+k/2})$ time for even $k$ and $O(n^2 \cdot m^{(k+1)/2})$ time for odd $k$.

### 6.1 A practical implementation

Next we give a practical, near-linear-time implementation of Contract-Cycles$_3$. The performance guarantee achieved is $c_3 = 1.75$. Contract-Cycles$_3$ consists of two phases: (1) repeatedly finding and contracting cycles of three or more edges (called long cycles), until no such cycles exist, and then (2) contracting the remaining 2-cycles.

**High-level description of the algorithm.**

To perform Phase (1), the algorithm does a depth-first search (DFS) of the graph from an arbitrary root. During the search, the algorithm identifies edges for contraction by adding them to a set $S$. At any point in the search, $G'$ denotes the subgraph of edges and vertices traversed so far. The rule for adding edges to $S$ is as follows: when a new edge is traversed, if the new edge creates a long cycle in $G'/S$, the algorithm adds the edges of the cycle to $S$. The algorithm thus maintains that $G'/S$ has no long cycles. When the DFS finishes, $G'/S$ has only 2-cycles. The edges on these 2-cycles, together with $S$, are the desired SCSS.

Because $G'/S$ has no long cycles and the fact that the original graph is strongly connected, $G'/S$ maintains a simple structure:

**Lemma 6.1.** The algorithm maintains the following invariant. (i) The graph $G'/S$ consists of a branching with edges directed outwards from the super-vertex containing the root, together with some reverse edges that are of the form $(W, U)$ such that $(U, W)$ is in the branching. (ii) If such a reverse edge $(W, U)$ is not present in $G'/S$, then the edge $(U, W)$ is on the path in the branching from the super-vertex containing the root to the super-vertex containing the vertex currently being visited by the DFS. This path is called the “active” path.

**Proof.** Clearly the invariant is initially true. We show that each given step of the algorithm maintains the invariant. In each case, if $u$ and $w$ denote vertices in the graph, then let $U$ and $W$ denote the vertices in $G'/S$ containing $u$ and $w$, respectively.

When the DFS traverses an edge $(u, w)$ to visit a new vertex $w$: Vertex $w$ and edge $(u, w)$ are added to $G'$. Vertex $w$ becomes the current vertex. In $G'/S$, the outward branching is extended to the new vertex $W$ by the addition of edge $(U, W)$. No other edge is added, and no cycle is created. Thus, part (i) of the invariant is maintained. The super-vertex containing the current vertex is now $W$, and the new “active path” contains the old “active path”. Thus, part (ii) of the invariant is also maintained.

When the DFS traverses an edge $(u, w)$ and $w$ is already visited: If $U = W$ or the edge $(U, W)$ already exists in $G'$, then no cycle is created, $G'$ is unchanged, and the invariant is clearly maintained. Otherwise, the edge $(U, W)$ is added to $G'$ and a cycle with the simple structure illustrated in Fig. 3 is created in $G'/S$. Dark nodes represent nodes that are still active. The cycle consists of the edge $(U, W)$, followed by the (possibly
empty) path of reverse edges from $W$ to the lowest-
common-ancestor (lca) of $U$ and $W$, followed by
the (possibly empty) path of branching edges from
the lca($U$, $W$) to $W$. Addition of ($U$, $W$) to $G'$/$S$
and contraction of this cycle (in case it is a long
cycle) maintains part (i) of the invariant. If the
"active path" is changed, it is only because part
of it is contracted, so part (ii) of the invariant is
maintained.

When the DFS finishes visiting a vertex $w$: No
dge is added and no cycle is contracted, so part
(i) is clearly maintained. Let $u$ be the new current
vertex, i.e., $w$'s parent in the DFS tree. If $U = W$,.
then part (ii) is clearly maintained. Otherwise,
consider the set $D$ of descendants of $w$ in the DFS
 tree. Since the original graph is strongly connected,
some edge ($x$, $y$) in the original graph goes from
the set $D$ to its complement $V - D$. All vertices in $D$
have been visited, so ($x$, $y$) is in $G'$. By part (i)
of the invariant, the vertex in $G'$/$S$ containing $x$ must
be $W$, while the vertex in $G'$/$S$ containing $y$ must
be $U$. Otherwise the edge corresponding to ($x$, $y$)
in $G'$/$S$ would create a long cycle. □

The algorithm maintains the contracted graph
$G'$/$S$ using a union-find data structure [22] to re-
represent the vertices in the standard way and using
three data structures to maintain the branching,
the reverse edges discovered so far, and the "active
path". When a cycle arises in $G'$/$S$, it must be
of the form described in the proof of Lemma 6.1
and illustrated in Fig. 3. Using these data struc-
tures, the algorithm discovers it and, if it is long,
contracts it in a number of union-find operations
proportional to the length of the cycle. This yields
an $O(ma(m, n))$-time algorithm.

The vertices of $G'$/$S$ are represented in union-
find sets as follows:

MAKE-SET($v$): Adds the set $\{v\}$ corresponding to
the new vertex of $G'$/$S$.

FIND($v$): Returns the set in $G'$/$S$ that contains
vertex $v$.

UNION($u$, $v$): Joins into a single set the two sets
 corresponding to the vertices in $G'$/$S$ containing
$G'$'s vertices $u$ and $v$.

The data structures representing the branching,
reverse edges, and the active paths, respectively are:

from-root[$W$]: For each branching edge ($U$, $W$) in
$G'$/$S$, from-root[$W$] = ($u$, $w$) for some $u, w \in
(U \times W) \cap E$.

to-root[$U$]: For each reverse edge ($U$, $W$) in $G'$/$S$,
to-root[$U$] = ($u$, $w$) for some $u, w \in (U \times
W) \cap E$.

to-active[$U$]: For each vertex $U$ on the "active
path" in $G'$/$S$, to-active[$U$] = ($u$, $w$) where
($u$, $w$) $\in (U \times W) \cap E$ and $W$ is the child
of $U$ for which the recursive DFS call is
currently executing, unless no recursive DFS
is executing, in which case to-active[$U$] =
current.

For all other vertices, to-active[$U$] = nil.
Pseudo-code for the algorithm is given below.

CONTRACT-CYCL~S_3(G = (V, E)) —
1 $S \leftarrow \{\}$
2 Choose $r \in V$.
3 DFS($r$)
4 Add 2-cycles remaining in $G'$/$S$ to $S$.
5 return $S$

DFS($u$) —
1 to-active[FIND($u$)] $\leftarrow$ current
2 for each vertex $w$ adjacent to $u$
   — traverse edge ($u$, $w$) —
   if ($w$ is not yet visited)
   3 MAKE-SET($w$)

...
4
9
is
algorithm
13
11
3
2
8
5
7
17
16
15
14
12
19
7
Here
approximation
straightforward
by
achieving
a
preceding
the
discussion,
for
a
graph.
SCSS
of
edges
are
deleted.
SCSS3
problem
is
much
harder
and
has
a
rich
structure.
It
is
not
hard
to
show
that
the
SCSS3
problem
is
as
hard
as
bipartite
matching.
In
fact
we
show
that
the
SCSS3
problem
is
reducible
to
matching
and
hence
has
a
polynomial
time
algorithm.
Due
to
lack
of
space,
we
omit
the
details.

**Theorem 7.1.** There is a polynomial-time algorithm for the SCSS problem.

**Corollary 7.1.** The performance guarantee of the modified Contract-Cycles algorithm is $c_k - 1/36$ (for $k \geq 4$), where $c_k$ is the performance guarantee of Contract-Cycles.

### 7.2 NP-hardness

We make no conjecture concerning the SCSS problem. However, we next show that the SCSS problem is NP-hard, and that for some $c > 0$, the SCSS problem is SNP-hard.

**Theorem 7.2.** The minimum SCSS problem is NP-hard.

**Proof.** The proof is by a reduction from SAT. We omit the proof here. The complete proof is given in the full version of the paper [14].

### 7.3 SNP-hardness

Next we consider the SNP-hardness of the problem.

**Theorem 7.3.** The minimum SCSS problem is SNP-hard.

**Proof.** The proof is by a reduction from the vertex cover problem. Finding a minimum vertex cover is MAX SNP-hard in graphs whose maximum
degree is bounded by seven [19]. The reduction is similar to the reduction from vertex cover to Hamiltonian circuits [8].

Let \( G \) be an undirected graph whose maximum degree is bounded by seven. Let \( G \) have \( m \) edges and \( n \) vertices. We construct a \( 2m + 1 \) vertex digraph \( D \) with root vertex \( r \) and no cycle longer than 17. Any vertex cover of \( G \) of size \( s \) will yield an SCSS of \( D \) of size \( 2m + s \), and vice versa. Since the degree of \( G \) is bounded, \( m = O(n) = O(s) \) and it is easily verified that this yields an L-reduction from degree-bounded vertex cover to the minimum SCSS\(_m \) problem.

![Figure 4: A cover testing component.](image)

Applying Vizing’s theorem [23], color the edges of \( G \) in polynomial time with at most eight colors so that no two edges incident to a vertex share the same color. Label the edges \( \{1, 2, \ldots, 8\} \) corresponding to the coloring.

As the construction proceeds, each vertex in \( G \) will have a “current vertex,” initially the root vertex, in \( D \). For each edge \( (u, v) \), in order of increasing label, add a “cover-testing gadget” to \( D \), as illustrated in Fig. 4. Specifically, add two new vertices \( x \) and \( y \). Add two edges into \( x \): the first, labeled \( u^+ \), from the current vertex of \( u \); the second, labeled \( u^- \), from \( y \). Similarly, add two edges into \( y \): the first, labeled \( v^+ \), from the current vertex of \( v \); the second, labeled \( v^- \), from \( x \). Make \( y \) the new current vertex of \( u \); make \( x \) the new current vertex of \( v \). Finally, after all edges of \( G \) have been considered, for each vertex \( v \) in \( G \), add an edge labeled \( v^+ \) from its final current vertex to the root. The gadgets are implicitly layered, with each gadget being assigned to a layer corresponding to the label of the associated edge in \( G \). Except for the 2-cycle edges and edges incident to the root in \( D \), the edges go forward in this layering. It is easily verified that due to this layering \( D \) has no cycle with more than 17 edges.

Given a vertex cover of size \( s \) of \( G \), construct an SCSS of \( D \) of size \( 2m + s \) as follows. For each vertex \( u \) in \( G \), let \( d \) be the degree of \( u \) in \( G \). If \( u \) is in the vertex cover, add the \( d + 1 \) edges labeled \( u^+ \) in \( D \) to the SCSS. Otherwise, add the \( d \) edges labeled \( u^- \) in \( D \) to the SCSS. It is easy to verify that the resulting SCSS is in fact an SCSS and has \( 2m + s \) edges.

Conversely, given an SCSS in \( D \) of size \( 2m + s \), construct a vertex cover of size \( s \) as follows. First, as long as some non-root vertex \( y \) has both of its incoming edges in the SCSS, modify the SCSS as follows. Let \( (x, y) \) be the edge labeled \( v^- \) for some \( v \). Remove the edge \( (x, y) \) and add the other edge out of \( x \), if it is not already present. Alternatively, if some non-root vertex \( x \) has both of its outgoing edges in the SCSS, remove the edge \( (x, y) \) and add the other edge into \( y \). Repeat either modification as long as applicable.

By the layering of \( D \), each modification maintains the strong connectivity of the SCSS. Clearly none of the modifications increases the size. Each step reduces the number of edges labeled \( u^- \) for some \( u \) in the SCSS, so after at most \( 2m \) steps, neither modification applies, and in the resulting SCSS every non-root vertex has exactly one incoming edge and one outgoing edge in the SCSS.

An easy induction on the layering shows that for any vertex \( v \) in \( G \), either all of the edges labeled \( v^+ \) in \( D \) are in the SCSS or none are, in which case all of the edges labeled \( v^- \) are in the SCSS. Let \( C \) be the set of vertices in \( G \) of the former kind. It is easy to show that the size of the SCSS is \( 2m + |C| \), so that \(|C| \leq s \). For every edge \( (u, v) \) in \( G \), the form of the gadget ensures that at least one of the two endpoints is in \( C \). Hence, \( C \) is the desired cover. \( \square \)

8 Open Problems

An obvious problem is to further characterize the various complexities of the minimum SCSS\(_s \) problems.

The most interesting open problem is to obtain a performance guarantee that is less than 2 for the weighted strong connectivity problem (as mentioned earlier, the performance factor of 2 is
due to Frederickson and JáJá [6]). Such an algorithm may have implications for the weighted 2-connection problem [15] in undirected graphs as well.

The performance guarantee of $k$-EXCHANGE probably improves as $k$ increases. Proving this would be interesting — similar “local improvement” algorithms are applicable to a wide variety of problems.

References


