Primal-Dual-Based Algorithms for a Directed Network Design Problem

Vardges Melkonian • Éva Tardos

Department of Mathematics, Ohio Universtiy, Athens, Ohio 45701, USA
Department of Computer Science, Cornell University, Ithaca, New York 14853, USA
vardges@math.ohiou.edu • eva@cs.cornell.edu

We present efficient algorithms for a special case of network design problems, the *strong connectivity problem*. Given a directed graph G, the strong connectivity problem seeks a minimum-cost strongly connected spanning subgraph of G. Our algorithms include the primal-dual method, penalty algorithms, and drop algorithms. Primal-dual methods have been quite successfull for developing algorithms for undirected network design problems. However, no results are known for extending them to the directed counterparts. We apply the primal-dual method to the strong connectivity problem, and show that the new algorithm has an approximation guarantee of three. Our computational results over randomly created instances show that the primal-dual is practically very efficient. We develop two improved algorithms, penalty and drop, building on primal-dual as a subroutine.
(*Networks-Graphs; Integer Programming, Heuristics; Analysis of Algorithms*)

---

1. Introduction

We consider the following network design problem for directed networks. Given a directed graph with nonnegative costs on the arcs, find a minimum-cost subgraph where the number of arcs leaving set S is at least $f(S)$ for all subsets S. Formally, given a directed graph $G = (V, E)$ and a requirement function $f : 2^V \mapsto Z$, the network design problem is the following integer program:

$$
\begin{aligned}
(IP) \qquad & \min && \sum_{e \in E} c_e x_e && && (1)\\
& \text{s.t.} && \sum_{e \in \delta^+(S)} x_e \geq f(S), && \text{for each } S \subseteq V, && (2)\\
& && x_e \in \{0,1\}, && \text{for each } e \in E, && (3)
\end{aligned}
$$

where $\delta^+(S)$ denotes the set of arcs leaving S.

The following are some special cases of interest. When $f(S) = 1$ for all $\emptyset \neq S \subset V$ that contain (don't contain) a given node r the problem is that of finding a minimum-cost *rooted out(in)-branching.* Edmonds (1967) showed that this problem can be solved optimally in polynomial time. When $f(S) = k$ for all $\emptyset \neq S \subset V$, the problem is that of finding a minimum-cost k-connected subgraph. The directed Steiner tree problem is to find the minimum-cost directed tree rooted at r that contains a subset of vertices $D \subseteq V$. This problem is a network design problem where $f(S) = 1$ if $r \notin S$ and $S \cap D \neq \emptyset$ and $f(S) = 0$ otherwise. These last two problems are known to be NP-complete. In fact, the directed Steiner tree problem contains the set-cover problem as a special case, and hence no polynomial time algorithm can achieve an approximation better than $O(\log n)$ unless P=NP (Raz and Safra 1997).

In the last ten years there has been significant progress in designing approximation algorithms for *undirected* network design problems, the analog of this problem where the graph G is undirected. General techniques have been developed for the undirected case, e.g., primal-dual algorithms (Agrawal et al. 1995, Goemans and Williamson 1995, Williamson et al. 1995, Goemans et al. 1994, Rajagopalan and Vazirani 1999, Jain and Vazirani 1999) and LP rounding techniques (Jain 1998).

There has been very little progress made on *directed* network design problems. The main general technique used for approximating undirected network design problems, the *primal-dual method* does not have a simple extension to the directed case. In this paper, we extend the primal-dual to a special case of directed network design problems, the *strong connectivity problem.* Given a directed graph $G = (V, E)$, the problem is to find a minimum-weight *strongly connected spanning subgraph.* In terms of our integer-programming formulation, this is the special case when all subset requirements are one. We will refer to that special integer program as (IP_{SC}).

This problem is interesting in the following sense: though it is the most special and the simplest case among NP-hard directed network design problems, the approximation factor

is not better than that for the general crossing supermodular case (Melkonian and Tardos 1999). So it is natural to seek improved algorithms for this special case first.

The strong connectivity problem was first studied by Frederickson and Jaja (1981), and an algorithm achieving an approximation factor of two was obtained. It is obtained by taking the union of a minimum-weight in-branching and a minimum-weight out-branching, rooting at an arbitrary node. Recall from that the branching problem can be solved optimally by Edmonds' (1967) algorithm. We will refer to the algorithm of Frederickson and Jaja as the *branchings algorithm*.

The branchings algorithm has the shortcoming that its in-branching and out-branching subroutines might have a "myopic view" when choosing arcs and thus omit some arcs that have high "common" value. On the other hand, the primal-dual does not have that myopic view. It takes a more holistic approach when choosing arcs, and thus presumably could lead to a good algorithm for the strong connectivity problem as it does in the case of undirected network design problems. In this paper we discuss the difficulties of extending the primal-dual for undirected networks to the strong connectivity problem. Straightforward extensions could lead to algorithms that do not guarantee constant approximation. However, by doing some modifications we give a version of primal-dual that has a constant approximation factor and performs well in practice. On average, its output is within 20% of optimum, and it almost always delivers a better solution than does the branchings algorithm. We prove that the primal-dual is a three-approximation algorithm. Note that the factor of three is not known to be tight, but rather it is the result of our proof technique. In fact, in the worst-case example we could get (given in Section 3.4) the primal-dual output is only $2-2/n$ times more expensive than is the optimum. Our conjecture is that $2-2/n$ is the real approximation factor, and computational results support this conjecture. In another example given in the same Section 3.4, the branchings algorithm delivers a solution that is exactly twice more expensive than the optimum.

Along with the primal-dual we present several improved algorithms for the strong connectivity problem. Most of these algorithms use primal-dual as a subroutine.

Suppose a given solution to a network design problem is not optimal. Intuitively, we might have the following two reasons for that: (1) there are some "good" arcs that are not in the solution; (2) there are some "bad" arcs (or combinations of arcs) that for some reason were included in the current solution. Normally both (1) and (2) are true for non-optimal solutions. How can we fix this situation? In Section 4 we add penalties to suspiciously bad

arcs to deal with type (2) situations. In Section 5, we will give a procedure to discover "good" arcs mentioned in (1). This procedure leads to so-called drop algorithms that were first proposed by Zhu et al. (1999).

While it is hard to give any theoretical guarantees for most of these algorithms, the computational results are promising for some of them.

The paper is structured as follows. We start Section 2 with a general discussion about the primal-dual method, and how it is used to get approximation algorithms. In Section 2.2, we discuss why the technique for undirected networks does not extend easily to directed counterparts; then we give modifications to primal-dual that allow us to get a good approximation guarantee for directed networks. In Section 3, we prove the main theorem of this paper, which states that our implementation of primal-dual for the strong connectivity problem yields a three-approximation algorithm. The penalty algorithms are covered in Section 4, and in Section 5 we discuss the drop algorithms. We speak about computational results in Section 6.

2. Primal-Dual Algorithm

First we will present the general primal-dual technique based on Goemans and Williamson (1995). Then we will specialize it to the strong connectivity problem and will give a performance guarantee for the algorithm in the next section.

2.1 General Primal-Dual Technique

Consider the LP relaxation of (IP_{SC}):

$$
\begin{aligned}
(LP_{SC}) \qquad & \min \quad \sum_{e \in E} c_e x_e & & (4)\\
& \text{s.t.} \quad \sum_{e \in \delta^+(S)} x_e \geq 1, & \text{for each } S \subseteq V, \quad & (5)\\
& \qquad\qquad x_e \geq 0, & \text{for each } e \in E. \quad & (6)
\end{aligned}
$$

and its dual linear program:

$$(DP_{SC}) \qquad \max \sum_{S\subseteq V} y_S \tag{7}$$

$$\text{s.t.} \sum_{S:e\in\delta^+(S)} y_S \leq c_e, \qquad \text{for each } e \in E, \tag{8}$$

$$y_S \geq 0, \qquad \text{for each } S \subseteq V. \tag{9}$$

In the primal-dual method for approximation algorithms, an approximate solution to (IP_{SC}) and a feasible solution to the dual of LP relaxation are constructed simultaneously; the performance guarantee is proved by comparing the values of both solutions.

Consider the complementary slackness conditions for (LP_{SC}) and its dual (DP_{SC}). There are *primal complementary slackness conditions* corresponding to primal variables:

$$x_e > 0 \Rightarrow \sum_{S:e\in\delta^+(S)} y_S = c_e \tag{10}$$

and there are *dual complementary slackness conditions* corresponding to dual variables:

$$y_S > 0 \Rightarrow \sum_{e\in\delta^+(S)} x_e = 1 \tag{11}$$

The general outline of primal-dual method for approximation algorithms is given in Algorithm 2.1.

Algorithm 2.1 General Primal-Dual Method for Approximation Algorithms

---

Initialize: dual $y \leftarrow 0$; infeasible primal A
while there does not exist a feasible integral solution obeying primal complementary slackness conditions **do**
  Get direction of increase for dual (moving towards feasibility of primal)
end while
Return feasible integral solution x obeying primal complementary slackness

---

The primal-dual method for approximation algorithms differs from the classical primal-dual method in that some of the complementary slackness conditions are not enforced. In most cases the dual conditions are not enforced, and that is the approach we use in this paper; but there are some algorithms that relax the primal conditions. Relaxing the complementary slackness conditions is not a surprise considering that our problems are NP-hard: enforcing both primal and dual complementary slackness conditions would lead to an optimal solution. Though the dual complementary slackness conditions are not enforced, a main idea of the

primal-dual approximation algorithm is to keep those conditions as little violated as possible. As we see later, a successfull realization of that idea helps to get a good approximation factor.

Now we go into detail to see how the ideas of Algorithm 2.1 are implemented for network design problems.

Primal-dual is an iterative algorithm. It maintains a dual solution that is initially zero and an infeasible solution $A \subseteq E$ for (IP_{SC}) that is initially empty. In each iteration the algorithm adds an arc to current A until A gets feasible for (IP_{SC}). The idea is to add these arcs in such a way that the dual solution remains feasible and the primal complementary slackness conditions stay satisfied. The next paragraph elaborates how to achieve this.

If the current A is infeasible for (IP_{SC}), then (2) is not satisfied for some subsets of V. There is a dual variable y_S corresponding to each of those subsets. The idea is to choose a set of subsets violating (2), call it $Violated(A)$, and increase the dual values of all $S \in Violated(A)$ uniformly until a dual constraint gets tight. The arc corresponding to the dual constraint is added to A (ties are broken arbitrarily). Note that this choice of the arc ensures that the dual solution remains feasible and that primal complementary slackness conditions are satisfied.

The above algorithm has the shortcoming that, though at any particular iteration the edge added to A was needed for feasibility, by the time the algorithm terminates it may no longer be necessary. These unnecessary edges increase the cost of A. To solve this problem, the refined algorithm does reverse deletion of the redundant edges in stage 2 (after getting an original feasible A). Note that the famous Dijkstra algorithm for the shortest-path problem also does reverse deletion at the end to get rid of redundant arcs (actually, Dijkstra's algorithm can be interpreted as a special case of primal-dual algorithm). The summary of the primal-dual for general network design problems is given in Algorithm 2.2.

To analyze the performance guarantee of this algorithm, we need the following definition.

Definition 1 *A set $B \subseteq E$ is said to be a minimal augmentation of a set $A \subseteq E$ if:*

- *$A \cup B$ is feasible, and*
- *for any $e \in B$, $A \cup B - \{e\}$ is not feasible.*

Let A_f be the output of the algorithm. Note that if A is the infeasible set in some iteration, then $A_f - A$ is a minimal augmentation of A.

Earlier we mentioned that the dual complementary slackness conditions (11) are not enforced in primal-dual approximation algorithms but they are kept violated as little as

Algorithm 2.2 Primal-Dual

```
Initialize: y ← 0, A ← ∅, l ← 0 (l is a counter)
while A is not feasible do
  l ← l + 1
  𝒱 ← Violated(A) (a subroutine returning a set of violated subsets of V)
  Increase y_S uniformly for all S ∈ 𝒱 until ∃ e_l ∉ A such that Σ_{S:e_l∈δ⁺(S)} y_S = c_{e_l}
  A ← A ∪ {e_l}
end while
for j ← l down to 1 do
  if A − {e_j} is still feasible then
    A ← A − {e_j}
  end if
end for
Return A
```

possible to achieve a good approximation guarantee. What are those conditions in terms of this particular algorithm? Note that $y_S > 0$ means that subset S was in $Violated(A)$ in some iteration. Now $\sum_{e\in\delta^+(S)} x_e$ is the number of $\delta^+(S)$-arcs in the minimal augmentation $A_f - A$ of A (this follows from *reverse* deletion). Thus, (11) says that for any $S \in Violated(A)$ we should have $|\delta^+(S) \cap (A_f - A)| = 1$. We can't hope to have this condition satisfied for NP-hard network design problems. But if, on average, $|\delta^+(S) \cap (A_f - A)|$ is not a big number for $S \in Violated(A)$, that is, if the dual complementary slackness conditions on average are not violated much (for any solution), then it is possible to give a good approximation guarantee for the primal-dual. Formally, this result is stated in the next theorem, which is the main analysis tool for primal-dual approximation algorithms and is due Goemans and Williamson (1995):

Theorem 1 *Suppose for any infeasible A and any minimal augmentation B of A,*

$$\sum_{S\in Violated(A)} |\delta^+(S) \cap B| \leq \alpha |Violated(A)|. \tag{12}$$

Then

$$\sum_{e\in A_f} c_e \leq \alpha \cdot c(y) \leq \alpha \cdot OPT, \tag{13}$$

where $c(y)$ is the cost of the dual solution at the end of the algorithm, and OPT is the optimal value of (IP_{SC}).

Theorem 1 implies that the primal-dual is an α-approximation algorithm. The question is how to achieve a good (small) α.

2.2 Choosing Violated Subsets for Strong Connectivity Problem

In this section, we give those details of the algorithm that are specific to the strong connectivity problem.

For good performance of the algorithm, it is crucial to choose the set $Violated(A)$ properly. For example, for the generalized Steiner tree problem ($f(S)$ is 0 or 1 in *undirected* networks) the primal-dual works the in following way. In each iteration those connected components $S \subset V$ for which $f(S) = 1$ are included in $Violated(A)$. Goemans and Williamson (1995) show that this choice of $Violated(A)$ leads to a two-approximation for the generalized Steiner tree problem.

In the case of the strong connectivity problem, for a current infeasible solution A, the graph (V, A) can be partitioned into a set of maximal strongly connected components. Call those components *cells*. An important observation is that there are A-*arcs* connecting different cells (see Figure 1 for an example) while in the undirected case all A-*arcs* were within the cells. As we can see below, this really causes difficulties. If we didn't have A-*arcs* between different cells then arguments similar to the undirected case would yield the same approximation factor of two.

So how should we choose $Violated(A)$? Obviously, a violated set can't intersect with a cell. Thus, the cells and their unions are the only candidates for being violated subsets. A natural idea for choosing $Violated(A)$ is to include all those cells for which (2) is violated (i.e., there are no arcs leaving them). The example of Figure 1 shows that if $Violated(A)$ is chosen in this way, then using Theorem 1 cannot lead to a good approximation guarantee.

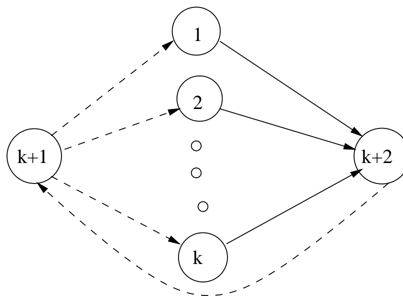

Figure 1: Example motivating our rule

In this example, the circles $1, 2, ..., k+2$ are the cells; the solid-line arcs are in the current infeasible solution A, and the dashed-line arcs form the minimal augmentation B of A. (We will use the same conventions in all our figures).

If we defined violated sets based on the rule above, then

$$|Violated(A)| = 2 \text{ and } \sum_{S \in Violated(A)} |\delta^+(S) \cap B| = k + 1,$$

which leads to an approximation factor of $O(k)$.

This example also suggests what would be a better way of selecting violated subsets. If we also included those cells that don't have A-arcs entering them, then $|Violated(A)|$ would go up to $O(k)$, leading to a better (constant) ratio. But how can we define such cells as violated subsets if they satisfy (2)? Note that if there is no arc entering a set S, then there is no arc leaving its complement, $V - S$. Thus, $V - S$ should be defined as a violated set. Note that if S is a cell then $V - S$ is the union of all the cells but S; call it the *co-cell* of S. Combining these ideas, here is our rule for selecting violated sets:

- include a cell S in $Violated(A)$ if $\delta^+(S) \cap A = \emptyset$;
- include a co-cell of S in $Violated(A)$ if $\delta^-(S) \cap A = \emptyset$.

Note that both a cell and its co-cell can be included in $Violated(A)$. If we apply this rule to the example above, then

$$|Violated(A)| = k + 3 \text{ and } \sum_{S \in Violated(A)} |\delta^+(S) \cap B| = 2k + 2.$$

This gives a ratio of less than two. In the next section we show that, in general, this rule leads to a constant approximation guarantee.

3. Approximation Guarantee for the Primal-Dual

In this section we prove that when the rule of Section 2.2 is applied, the primal-dual is a three-approximation algorithm for the strong connectivity problem. Based on Theorem 1, it is enough to show that

Theorem 2 *For any infeasible A and any minimal augmentation B of A,*

$$\sum_{S_i \in Violated(A)} |\delta(S_i) \cap B| \leq 3|Violated(A)|.$$

The rest of this section gives the proof of Theorem 2.

First we need some definitions and notation. A cell is called a *source* if there are no A-arcs entering it (and so its co-cell is included in $Violated(A)$). A cell is called a *sink* if there are no A-arcs leaving it (and so it is included in $Violated(A)$). In the example of Figure 2, cells 2, 6, 7, and 11 are sources, cells 3, 4, 5, 9, and 10 are sinks, and cells 1 and 8 are both sources and sinks.

A collection of arcs entering the same sink is called an *in-comb*; a collection of arcs leaving the same source is called an *out-comb*. In both cases, the number of arcs can also be zero. An infeasible set $A \subseteq E$ is called a *comb-forest* if it is a collection of in-combs and out-combs that don't have any common vertices. In the example below, $\{2 : 2 \to 3, 4, 5\}$ is an out-comb of size 3, $\{9 : 6, 7 \to 9\}$ is an in-comb of size 2, and $\{1 : \quad\}$ is both an in-comb and an out-comb of size 0.

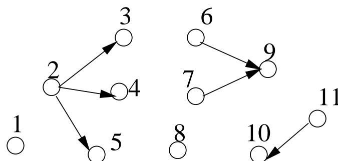

Figure 2: Comb Forest

In all our figures, the arcs of infeasible A will be solid and the arcs of its minimal augmentation will be dashed. Nodes in figures represent the cells of current A.

We will frequently compare $\sum_{S_i \in Violated(A)} |\delta^+(S_i) \cap B|$ with $|Violated(A)|$ in the proof. So let us introduce shortcuts for those expressions:

$$\Lambda_A^B = \sum_{S_i \in Violated(A)} |\delta^+(S_i) \cap B| \; ; \qquad \rho_A = |Violated(A)|.$$

Outline of the proof

The proof is by induction on $|V|$.

In Section 3.1, the theorem is proved for the cases of directed cycles and so-called digon-trees (defined in Section 3.1). These two cases constitute the induction base. If we are in neither of these two cases, an induction step is done by reducing the size of the network. We give the induction step in Section 3.2. Throughout the proof it is assumed that the infeasible set A has a special structure of a comb-forest as defined above. In Section 3.3, we show why this assumption can be made by reducing any A to a comb-forest. In Section 3.4, we give bad case examples for the primal-dual and branchings algorithms.

3.1 Induction Base: Cases of Digon-Trees and Directed Cycles

A *digon* is a pair of opposite arcs. If a digraph consists only of digons, and ignoring the directions will make it an undirected tree, then the digraph will be called *digon-tree*. An example of a digon-tree is given in Figure 3.

In this section, we show that the theorem statement is true for digon-trees and directed cycles. Later, these cases serve as the induction base for more general network structures.

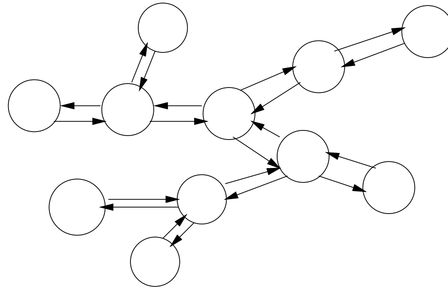

Figure 3: Example of Digon Tree

The Case of Digon-Trees

Lemma 1 *Suppose A is an infeasible set and B is a minimal augmentation of A such that A is a comb-forest and $A \cup B$ is a digon-tree. Then*

$$\sum_{S_i \in Violated(A)} |\delta(S_i) \cap B| \leq 2.5|Violated(A)|.$$

Proof is by induction on $|V|$. Namely, we consider several cases, and in each of these cases a given digon-tree is reduced to a smaller one; then we show that if the statement of the lemma is true for the smaller network, then it is also true for the bigger one.

Call a node v of a tree a *pre-leaf* if, after eliminating all the leaves, v becomes a leaf. Any non-trivial tree has a pre-leaf.

Suppose v is a pre-leaf of $A \cup B$. Consider the cases:

- *Case 1: There is a leaf u of v that is both a source and a sink.*

  Create a reduced instance (A', B') by deleting u and its incident arcs.

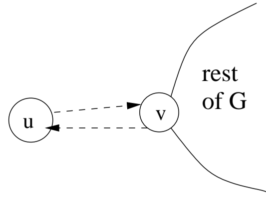

Figure 4: Case 1 Example

Then $\rho_{A'} = \rho_A - 2$. Each of the cell u and its co-cell will contribute one less in $\Lambda_{A'}^{B'}$ than in Λ_A^B; each of the cell v and its co-cell may contribute the same or one less in $\Lambda_{A'}^{B'}$ than in Λ_A^B (this depends on v being a source or a sink or both). Thus, $\Lambda_{A'}^{B'} \geq \Lambda_A^B - 4$. So if the lemma is true for A' then it is also true for A.

- *Case 2: We are not in case 1, and v has at least two leaf-neighbors.*

  Without loss of generality, assume that v is a source and all its leaf-neighbors are sinks (recall that we assume that A is a comb-forest).

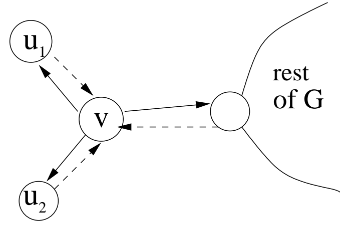

Figure 5: Case 2 Example

  Get a reduced instance A', B' by deleting one of the leaf-neighbors and the arcs incident to it. The deleted leaf-neighbor will contribute one less in $\Lambda_{A'}^{B'}$ than in Λ_A^B; the co-cell of v will contribute one less in $\Lambda_{A'}^{B'}$ than in Λ_A^B. So if the lemma statement is true for A' then it is also true for A.

- *Case 3: We are not in case 1, and v has exactly one leaf-neighbor u.*

  Without loss of generality assume that v is a source, and $v \to u$ is the only A-arc incident to it (see Figure 6, with u_1 as u and v_1 as v). Otherwise, we could apply the technique of case 2.

  Get a reduced instance A', B' by deleting u, v and the arcs incident to them. Then $\rho_{A'} = \rho_A - 2$. The cell u will contribute one less in $\Lambda_{A'}^{B'}$ than in Λ_A^B; the co-cell of v will contribute two less in $\Lambda_{A'}^{B'}$ than in Λ_A^B; each of the cell w and its co-cell may contribute the same or one less in $\Lambda_{A'}^{B'}$ than in Λ_A^B (this depends on w being a source or a sink or both). Thus, $\Lambda_{A'}^{B'} \geq \Lambda_A^B - 5$. So if the lemma is true for A' then it is also true for A. $\square$

Note that in case 3 we have the worst estimate of α, which is 2.5. Consider the following complete example (see Figure 6).

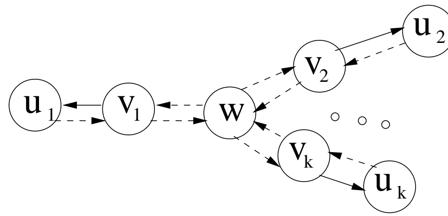

Figure 6: Example with $\alpha = 2.5$

Here, $\Lambda_A^B = 5k$ and $\rho_A = 2k+2$. As k increases, α asymptotically goes to 2.5. Thus, the value 2.5 for α is the best for which we can hope.

The Case of Directed Cycles

Another induction base is the case when $A \cup B$ is a directed cycle.

Lemma 2 *Suppose A is an infeasible set and B is a minimal augmentation of A such that A is a comb-forest and $A \cup B$ is a directed cycle. Then*

$$\sum_{S_i \in Violated(A)} |\delta(S_i) \cap B| = |Violated(A)|.$$

Proof:

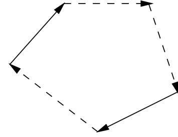

Figure 7: $A \cup B$ is a Cycle

In this case, there is exactly one B-arc entering (leaving) each source (sink); thus, $\Lambda_A^B = \rho_A$, and the lemma is true. □

3.2 Induction Step: Reducing the Size of the Network

In this section we show that if $A \cup B$ is neither a digon-tree nor a directed cycle, then we can apply an induction step by reducing the size of the network. During this reduction, we lose at most a factor of three in the approximation.

The idea of the reduction is the following. Suppose there is a directed cycle $C \subseteq A \cup B$ such that $|C| \geq 3$ and $C \neq A \cup B$. We reduce the size of the network by contracting C to a two-node structure. Here are the rules of reducing the original (V, A, B) to (V', A', B') (see also Figures 8 and 9):

• $V' = V \cup \{u, v\} - \{C\text{-nodes}\}$. In other words, we remove the nodes that are on cycle C and instead add two new nodes u and v.

• If $x \to y \in A, x \notin C, y \in C$, then $\exists\ x \to u \in A'$. In other words, if an A-arc enters C, then it enters u in A'. Note that some of these arcs might merge.

• If $x \to y \in A, x \in C, y \notin C$, then $\exists\ v \to y \in A'$. In other words, if an A-arc leaves C, then it leaves v in A'. Note that some of these arcs might merge.

• B-arcs entering C, enter v in B'. There are no mergers here; otherwise, one of the original B-arcs would be redundant.

• B-arcs leaving C, leave u in B'. There are no mergers here.

• We consider two cases for adding arcs $u \to v$ and $v \to u$.

Case 1: If there are A-arcs both entering and leaving C then $u \to v$ and $v \to u$ are added to B' (see Figure 8).

Case 2: If no A-arc enters C or no A-arc leaves C then $v \to u$ is added to A' and $u \to v$ is added to B' (see Figure 9).

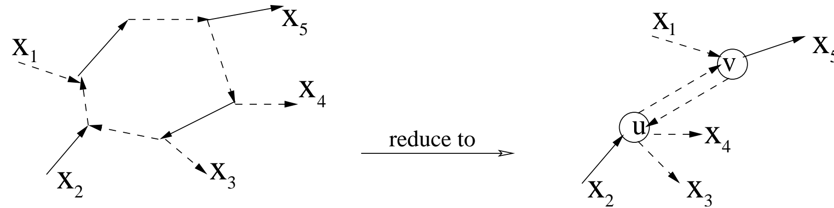

Figure 8: Cycle Reduction (Example of Case 1)

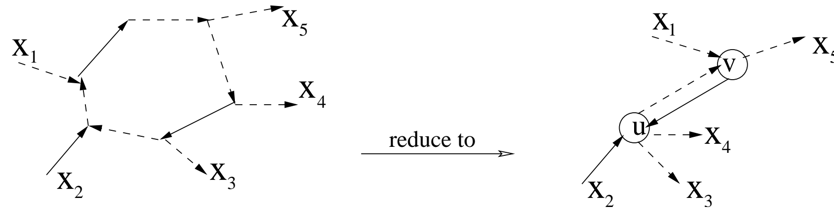

Figure 9: Cycle Reduction (Example of Case 2)

B' is an augmentation of A'. Suppose nodes y and z in $(V, A \cup B)$ were connected with a path that went through cycle C. In $(V', A' \cup B')$, y and z will be connected with the same path, but this time instead of traveling on C, it might go through cycle $u \to v \to u$.

B' might be not a minimal augmentation of A'. The only redundant arcs in B' could be $u \to v$ and $v \to u$. Assume the opposite: there is another redundant arc $s \to t \in B'$, that is, $B' - \{s \to t\}$ is still an augmentation of A'. Then there is a path $P = s \rightsquigarrow t$ in $A' \cup B'$ such that $s \to t \notin P$. But when some portion of cycle C is added to P (if necessary), the modified path would connect s to t also in $A \cup B$. Thus, $s \to t$ should have been redundant for B (if one of s and t is a newly-added node then the redundant arc is not $s \to t$ itself but rather its corresponding arc in B). This contradicts the fact that B was a minimal augmentation of A.

Based on the arguments above, B' (maybe after deleting arcs $u \to v$ and $v \to u$) is a minimal augmentation of A'.

Let ρ_C be the number of sources and sinks on cycle C. There is exactly one $B \cap C$-arc entering each source on C; there is exactly one $B \cap C$-arc leaving each sink on C. Thus, Λ_A^B is reduced exactly by ρ_C as the result of eliminating $B \cap C$-arcs. For any other B-arc we have a corresponding B'-arc in the reduced instance. Based on these observations,

- $\rho_{A'} = \rho_A - \rho_C + 2$ (cycle C was eliminated; the newly-added node u is a sink and not a source; the newly-added node v is a source and not a sink).
- $\Lambda_{A'}^{B'} \geq \Lambda_A^B - \rho_C$ ("$\geq$" because could have or not $u \to v$ and $v \to u$ in B').
- Also, by induction $\Lambda_{A'}^{B'} \leq 3\rho_{A'}$. Thus,

$$\Lambda_A^B \leq \Lambda_{A'}^{B'} + \rho_C \leq 3\rho_{A'} + \rho_C \leq 3(\rho_A - \rho_C + 2) + \rho_C = 3\rho_A + (6 - 2\rho_C) \leq 3\rho_A$$

The last inequality is true because $|C| \geq 3$ and so C has at least 3 sources and sinks on it (considering that A is a comb-forest). Thus, the statement of Theorem 2 is true by induction.

Note that if $|C| \geq 4$ then $\rho_C \geq 4$, and the same analysis would yield a factor of two. Thus, the worst case happens when $|C| = 3$ and we cannot reduce that cycle without losing a factor of three.

Yet we don't have a complete example where the value of α from Theorem 1 is exactly or asymptotically three (like the example of Figure 6 for $\alpha = 2.5$). Thus, α has a room to be improved from three but the best we can hope is 2.5. But even 2.5 is the best hope only when we are using the proof technique, which is based on Theorem 1. In reality, the algorithm might have a better approximation guarantee. In Section 3.4 we give an example where the algorithm output is $2-2/n$ times costlier than the optimum. This is the worst-case example that we have at this point, and so $2-2/n$ is the best hope for the approximation factor.

3.3 Reducing Infeasible Set A to Comb Forest

Throughout Sections 3.1 and 3.2 (the proof by induction) we assumed that the infeasible set A is a comb-forest. In this Section we get a comb-forest from any A using the following three reduction techniques:

Reduction Technique 1:

Feature of A: Removing an arc $a \in A$ doesn't change sources and sinks (see Figure 10). Then define the reduced instance by $A' = A - \{a\}$.

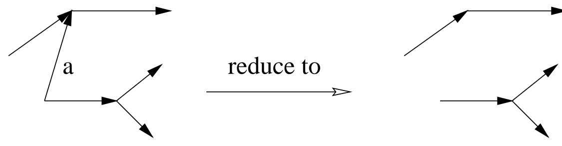

Figure 10: Example of Technique 1

If B is a minimal augmentation of A then $B' = B \cup \{a\}$ is an augmentation of A', maybe not a minimal one. The only arc that can be redundant in B' is a.

Assume the opposite: there is another redundant arc $u \to v \in B'$, that is, $B' - \{u \to v\}$ is still an augmentation of A'. Then there is a path $P = u \rightsquigarrow v$ in $A' \cup B'$ such that $u \to v \notin P$. But P is totally in $A \cup B$ too. Thus, $u \to v \in B$ should have been redundant for B either. This contradicts the fact that B was a minimal augmentation of A.

Based on the arguments above, either $B' = B$ or $B' = B \cup \{a\}$ is a minimal augmentation of A'. In either case, $\Lambda_A^B = \Lambda_{A'}^{B'}$ and $\rho_{A'} = \rho_A$. So if the theorem is true for A' then it is also true for A. Thus, A can be reduced to A'.

Reduction Technique 2:

Feature of A: There is a cell u that is neither a source nor a sink, and the number of A-arcs incident to u is at least 3 (see Figure 11).

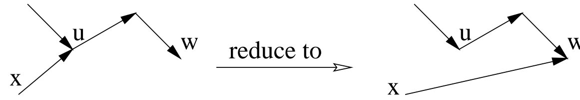

Figure 11: Example of Technique 2

Without loss of generality, consider the case when at least two arcs enter u. Suppose $x \to u$ is one of them. Since u is not a sink then there is a path $P = u \rightsquigarrow w \in A$ such that w is a sink. Create a reduced instance the following way:

$$A' = A \cup \{x \to w\} - \{x \to u\}.$$

If B is a minimal augmentation of A then $B' = B \cup \{x \to u\}$ is an augmentation of A', maybe not a minimal one. The only arc that can be redundant in B' is $x \to u$. The arguments for this are the same as those for the similar situation of technique 1.

Thus, either $B' = B$ or $B' = B \cup \{x \to u\}$ is a minimal augmentation of A'. In either case, $\Lambda_A^B = \Lambda_{A'}^{B'}$ and $\rho_{A'} = \rho_A$. So if the theorem is true for A' then it is also true for A.

Thus, A can be reduced to A'.

Reduction Technique 3:

Feature of A: For a cell u, there is exactly one A-arc $v \to u$ entering u and exactly one A-arc $u \to w$ leaving u (see Figure 12).

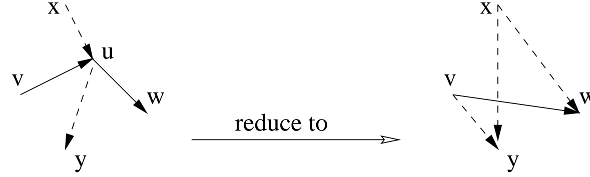

Figure 12: Example of Technique 3

Create the reduced instance by taking a shortcut from v to w, namely, by defining

$$V' = V - \{u\} \text{ and } A' = A \cup \{v \to w\} - \{v \to u, u \to w\}.$$

Suppose B is a minimal augmentation of A. First create an augmentation $\tilde{B}$ of A' (which might be not minimal yet) by changing the arcs entering or leaving u:

- if $x \to u \in B$ then $x \to w \in \tilde{B}$
- if $u \to y \in B$ then $v \to y \in \tilde{B}$
- if $x \to u, u \to y \in B$ then $x \to y \in \tilde{B}$ (note that w and v can also be considered as x and y respectively).

Note that we didn't eliminate any path connecting two nodes; we merely took a shortcut wherever node u was on our way. Thus, $\tilde{B} \cup A'$ is strongly connected, and $\tilde{B}$ is really an augmentation of A' but might be not a minimal one. Let D be the set of newly-added $\tilde{B}$-arcs. The only arcs in $\tilde{B}$ that might be redundant are those from D. But we have the following important observation:

Lemma 3 *Suppose $B' \subseteq \tilde{B}$ is a minimal augmentation of A'. If x is a sink of A and $x \to u \in B$ then $\exists\ x \to y \in B' \cap D$, i.e., at least one newly-added arc leaving x is in B'. Similarly, If y is a source of A and $u \to y \in B$ then $\exists\ x \to y \in B' \cap D$, i.e., at least one newly-added arc entering y is in B'.*

Proof: We will prove only the first part: similar arguments apply to the second part.

Suppose the statement is not true, i.e., no newly-added arc leaves x in B'. There is some path going from x to v in $A' \cup B'$. Since there is no $x \to y \in B' \cap D$, the same path could be used as a sub-path to reach from x to u in $A \cup B$ without using the arc $x \to u$. Thus, B is not a minimal augmentation of A, which is a contradiction.

Note that in the special case when $x = w$ and $w \to u \in B$, there is an arc $u \to y \in B$. If we assume the opposite, then there would be a path P from w to v in $A \cup B$ without using $w \to u$; $P \cup \{v \to u\}$ would be a path from w to u in $A \cup B$ without using $w \to u$; and thus $w \to u$ would be redundant in B. And if there is an arc $u \to y \in B$ then the arguments of the general case also apply here. □

Lemma 3 implies that if $B' \subseteq \tilde{B}$ is a minimal augmentation of A' then $\Lambda_{A'}^{B'} \geq \Lambda_A^B$. On the other hand, $\rho_{A'} = \rho_A$. So if the theorem is true for A' then it is also true for A. Thus, A can be reduced to A'.

After applying techniques 1-3 several times we can reduce any infeasible set A to a comb-forest. Techniques 2-3 provide that, in the reduced instance, there is no node that is neither a source nor a sink. And technique 1 eliminates an arc if it doesn't change the sources and sinks.

Note that the reduction techniques of this section would work not only for $\alpha = 3$ but for any other approximation factor.

3.4 Bad Case Examples

At the end of Section 3.2 it was mentioned that we don't have a tight example for our primal-dual algorithm. That is, we don't have an instance of the strong connectivity problem for which the algorithm returns a solution three times more expensive than the optimum. So how bad can the algorithm be? In Figure 13 we give a pathological example for which the algorithm output is $2 - 2/n$ times costlier than the optimum:

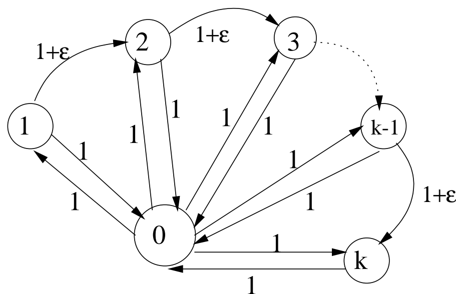

Figure 13: Pathologically Bad Example for the Primal-Dual

In this example, the numbers on the arcs are the costs, and ϵ is a very small positive number. When applying the primal-dual to this instance, all the arcs with cost one will be saturated first. Thus, the algorithm's output will consist of all the arcs incident to node zero

with total cost $2k$. On the other hand, it is easy to see that the optimal solution to this instance is the cycle $0 \to 1 \to 2 \to \cdots \to k-1 \to k \to 0$ with total cost $2 + (k-1)(1+\epsilon) = k + 1 + (k-1)\epsilon$. So the approximation ratio for this instance is $\leq 2k/(k+1) = 2 - 2/n$. This ratio tends to two as the problem size increases. This is the worst example for the performance of the primal-dual we could get theoretically. Our conjecture is that the primal-dual for the strong connectivity problem is really a $2 - 2/n$-approximation algorithm. Our computational results (see Section 6) also support this conjecture.

Note that the output of the branchings algorithm for the example of Figure 13 is the same as the output of the primal-dual. However, this example was not the worst case for the branchings algorithm. Consider the pathological example of Figure 14.

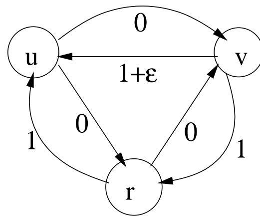

Figure 14: Pathologically Bad Example for the Branchings Algorithm

Again ϵ is a very small positive number. If node r is chosen as root, then the optimal in-branching is $\{u \to r, v \to r\}$ and the optimal out-branching is $\{r \to u, r \to v\}$. The total cost of the output of the branchings algorithm is two. On the other hand, the optimal solution is the cycle $r \to v \to u \to r$ with total cost $1 + \epsilon$. The approximation ratio asymptotically tends to two as ϵ goes to zero even though the problem size remains the same.

Note that the primal-dual returns the optimal solution for this example. It first includes all zero-cost arcs in the solution, then increases simultaneously the dual values of two violated subsets: $S_1 = \{u\}$ and $S_2 = \{v\}$. In the result, $v \to u$ gets saturated and is the last arc that is included in the solution. After reverse deletion, we just get the optimal solution.

4. Penalty Algorithms

The penalty algorithm discussed in this section uses the idea of adding penalty functions to arc costs to improve the performance of the primal-dual algorithm. First we give some motivation for why the idea of penalty functions makes sense for our problem. Then we discuss a classical example of using penalty functions in the traveling salesman problem

(Held and Karp 1970). Finally, we give the penalty algorithm for the strong connectivity problem.

4.1 Why Penalize?

Recall the worst-case example we have in Figure 13. In this example, the output of the primal-dual algorithm is twice as expensive as the optimal solution. The reason is that the primal-dual solution has "too many arcs:"

- all the arcs in the original network have roughly the same cost;
- the optimal solution is a cycle and thus contains the minimum possible number of arcs;
- the primal-dual solution, on the other hand, contains almost twice as many arcs as the optimal solution.

So having many arcs in the solution might be the reason for getting a bad solution. In order to overcome this problem we need to characterize the number of arcs in terms of node degrees. Note that the number of arcs in a directed network is:

$$(1/2)\sum_{u\in V}(indegree(u) + outdegree(u)).$$

In any solution to the strong connectivity problem, both the indegree and the outdegree of any node should be at least one. In case of a cycle solution, the indegree and the outdegree are exactly one. If either the indegree or the outdegree of a node is going up, this automatically increases the number of arcs in the solution. Say, in the example of Figure 13, the reason for having too many arcs is that node zero has indegree and outdegree both equal to $n-1$ in the primal-dual solution. So to decrease the number of arcs, we shall try somehow to prevent a node getting too many arcs incident to it. The technique that allows us to achieve that is adding *penalty functions* to the arc costs.

Note that this problem of having "too many arcs" might arise in almost all network-connectivity problems, not only for directed but also for undirected graphs. An exception is the minimum spanning tree problem where any solution has exactly $|V|-1$ arcs.

Adding penalty functions to costs is one of the most commonly-used techniques in optimization. One of the first applications, which is also related to our problem, is due to Held and Karp (1970). They applied the technique to the traveling salesman problem (TSP). In the TSP, it is required to find a minimum-cost tour (i.e., a cycle passing through each vertex exactly once). While in the strong connectivity problem it is just the assumption of our heuristic that a solution with low-degree nodes is a good one, in the TSP, nodes should be

low-degree in any feasible solution. Namely, any node is required to have degree two in any feasible solution to the symmetric (undirected) TSP, and any node is required to have both indegree and outdegree 1 in any feasible solution to the asymmetric (directed) TSP. Due to this restriction on node degrees, a transformation of arc costs $c_{ij} \leftarrow c_{ij} + \pi_i + \pi_j$ by adding penalties π's to the nodes leaves the TSP invariant, and this is the theoretical basis of the Held-Karp algorithm.

4.2 Penalty Algorithm for the Strong Connectivity Problem

In our penalty algorithm, the idea of using penalty functions is similar to that used by Held and Karp: the arcs incident to high-degree nodes are penalized more. Our main differences from the Held-Karp scheme are the following: (i) in the Held-Karp scheme, the solution costwise is improved in each iteration, *but* the current solution in each iteration (except maybe the last) is infeasible, and there is no guarantee that the algorithm will find a feasible solution. (ii) in our algorithm, the current solution is feasible in all iterations *but* there is no guaranteed way for changing the arc costs such that the current suboptimal solution is improved. This means that our algorithm is a local-search heuristic: in each iteration, we try to get a better solution in the neighborhood of the current solution, but there is no guarantee that we can do that.

The schematic outline of our penalty algorithm for the strong connectivity problem is given in Algorithm 4.1.

Our penalty algorithm is an iterative algorithm that uses the primal-dual as a subroutine. At the beginning of each iteration, we have a current best solution that was obtained by applying the primal-dual with respect to current arc costs. Suppose in the current best solution, node v has $indegree(v)$ and $outdegree(v)$. Then we get a modified instance by adding penalty functions to arc costs in the following way. For arc $i \rightarrow j$, the modified cost is:

$$c_{ij} + \Delta([outdegree(i) - 1] + [indegree(j) - 1])$$

That is, if the tail i has more than one outgoing arc in the solution, then in the modified instance we increase the costs of all outgoing arcs by some factor Δ of $outdegree(i) - 1$. Similarly, if the head j has more than one incoming arc in the solution, then in the modified instance we increase the costs of all incoming arcs by a factor Δ of $indegree(i) - 1$.

We apply the primal-dual to this instance with modified costs. If it returns a better solution in terms of original costs then we update the current best solution and start a new

iteration. Otherwise, the algorithm stops and returns the current best solution.

Algorithm 4.1 Penalty Algorithm

1: Initialize: $COST_0 \leftarrow 0$; $k \leftarrow 1$ (k is a counter);
2: $A_1 \leftarrow PrimalDual(c_{ij})$; $COST_1 \leftarrow$ cost of A_1;
3: **while** $k = 1$ or $COST_k < COST_{k-1}$ **do**
4: $\quad k \leftarrow k + 1$
5: $\quad outd(v) \leftarrow$ outdegree of node v in A_{k-1}; $ind(v) \leftarrow$ indegree of node v in A_{k-1};
6: $\quad c_{ij} \leftarrow c_{ij} + \Delta([outd(i) - 1] + [ind(j) - 1])$ (for some $\Delta \geq 0$);
7: $\quad A_k \leftarrow PrimalDual(c_{ij})$; $COST_k \leftarrow$ cost of A_k in terms of original arc costs;
8: **end while**
9: **return** A_{k-1}

Note that Δ might be changed several times in the same iteration until a better solution is found. Namely, if the current Δ doesn't change the solution, then we increase Δ, supposing that a larger perturbation could improve the solution; if the solution gets worse then we decrease Δ, supposing that a better value lies somewhere between 0 and the current Δ. Of course, we might not be able to improve the current solution at all. Choosing an initial Δ, the strategy of changing it and the stopping criteria for the algorithm are open implementation issues.

Since the penalty algorithm can only improve the solution of the primal-dual then it also has the approximation factor of three. While it is hard to improve this factor theoretically, computational results in Section 6 show that the penalty algorithm can considerably improve the performance of the primal-dual algorithm in practice.

5. Drop Algorithms

Suppose a given solution to a network design problem is not optimal. As we mentioned in Section 1, there are two natural ways to improve it. If the solution is not optimal then we might have the following two reasons for that: (1) there are some "good" arcs that are not in the solution; (2) there are some "bad" arcs (or combinations of arcs) that for some reason were included in the current solution. Normally, both (1) and (2) are true for non-optimal solutions. How can we fix this situation? In Section 4 we added penalties to suspiciously bad arcs to deal with type (2) situations. In this section, we will give a procedure to discover "good" arcs mentioned in (1). This procedure leads to so-called drop algorithms.

The drop algorithm for network design problems was proposed by Zhu et al. (1999). For the strong connectivity problem, it first runs the branchings algorithm to get an initial solution, and then does a series of local improvements to get a better solution.

We will first present Zhu's algorithm for the branchings algorithm and then extend it to the primal-dual case.

5.1 Intuition Behind Drop Algorithms

We will give some intuition behind drop algorithms in the context of the branchings algorithm. But the main ideas of this analysis also apply to the primal-dual case.

Let's reconsider the example from Section 3.4 given in Figure 15.

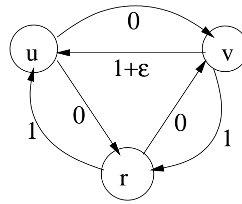

Figure 15: Pathologically Bad Example for the Branchings Algorithm

Recall that this was a worst-case example for the branchings algorithm. Arc $v \to u$ loses the competition to arc $r \to u$ in the out-branching algorithm and to arc $v \to r$ in the in-branching algorithm, in both cases just by a little, namely by ϵ. But overall, $v \to u$ is a very valuable arc because while $r \to u$ is useful *only* for out-branching and $v \to r$ is useful *only* for in-branching, $v \to u$ can be used for both and yield overall a much cheaper solution.

Thus, there are some arcs with a "hidden" value that are "ignored" by the branchings algorithm. What is the numerical procedure that will allow us to fix the situation by including these valuable arcs in the solution?

Note that if we decrease the cost of $v \to u$ just a little, from $1 + \epsilon$ to $1 - \epsilon$, then $v \to u$ will be included in both in-branching and out-branching, and the cost of the solution will decrease dramatically, from 2 to $1 - \epsilon$. Thus, some kind of "sensitivity analysis" could help us find those hidden valuable arcs.

5.2 Zhu's Drop Algorithm

In Zhu's algorithm, the sensitivity analysis is done by computing the following number for each arc e:

$$drop(e) = \frac{(c(B_I) + c(B_O)) - (c(B'_I) + c(B'_O))}{c(e)} \ , \tag{14}$$

where $c(B'_I)$ and $c(B'_O)$ are the minimum-cost in-branching and out-branching if the cost of e is dropped to 0.

The algorithm is iterative. In each iteration it determines the most important arc as the one with the highest drop value (as defined in (14)) and drops its cost to zero. The algorithm terminates when the cost of the branchings output gets to zero. This final output consists of arcs that were chosen as most valuable arcs in some iterations of the algorithm. The outline of this procedure is given in Algorithm 5.1.

Algorithm 5.1 Branchings-Drop Algorithm

1: Pick an arbitrary root node r
2: **while** $c(B_I) + c(B_O) > 0$ **do**
3: Let e^* be an arc with maximum drop
4: Set $c(e^*) = 0$
5: **end while**
6: Compute 0-cost in-/out branchings rooted at r
7: **return** union of edges in the branchings

In Section 5.1 we gave some intuition why the drop algorithm is likely to improve the solution of the branchings algorithm. Zhu et al. (1999) give a theoretical guarantee that drop algorithm's output can't be worse than the output of the branchings algorithm.

Lemma 4 *Branchings-drop is a two-approximation algorithm.*

Proof: Let $D = \{e_1, e_2, ..., e_k\}$ be the set of those arcs whose costs are dropped to zero during the algorithm (the cost of e_i is dropped to zero in iteration i). Then we have the following chain of inequalities and equalities:

$$cost(Drop) \leq \sum_{e \in D} c(e) \leq \sum_{e \in D} drop(e)c(e) = \sum_{i=1}^{k} \frac{(c_i(B_I) + c_i(B_O)) - (c_i(B'_I) + c_i(B'_O))}{c(e_i)} c(e_i)$$

$$= \sum_{i=1}^{k} (c_i(B_I) + c_i(B_O)) - (c_i(B'_I) + c_i(B'_O)) = cost(Branchings) \tag{15}$$

Here for each $e \in D$, $drop(e)$ is the drop value of e in the iteration when its cost was dropped to 0.

The first inequality is true because the output of the drop algorithm consists of D-arcs (plus, maybe some arcs which had zero cost originally). The second inequality follows from the fact that the largest drop value in each iteration is at least one. Really, in each iteration the output of the branchings algorithm has at least one arc with non-zero cost; for that arc the drop value is clearly at least one. The first equality follows from the definition of the drop value. The last equality is true because $c_{i+1}(B_I) + c_{i+1}(B_O) = c_i(B'_I) + c_i(B'_O)$ for each $i \in 1, ..., k-1$, $c_1(B_I) + c_1(B_O) = cost(Branchings)$, and $c_k(B'_I) + c_k(B'_O) = 0$.

Since the branchings algorithm is a two-approximation algorithm, (15) implies that the drop algorithm also has approximation guarantee of two. □

5.3 Extension of the Drop Algorithm to Primal-Dual

In Section 5.1 we discussed why the "cost-dropping" procedure might improve the solution of the branchings algorithm. Would a similar procedure improve the solution of the primal-dual algorithm? For the branchings algorithm, we argued that in-branching and out-branching might have a "myopic view" when choosing arcs and thus omit some arcs that have high "common" value. The primal-dual algorithm does not have that myopic view; it takes more holistic approach when choosing arcs. But it turns out that in this case too, "sensitivity analysis" might discover some valuable arcs that were wrongly "ignored" by primal-dual.

Reconsider the example of Figure 13 from Section 3.4 (see Figure 16) to illustrate the potential benefit of applying the sensitivity analysis for the primal-dual case.

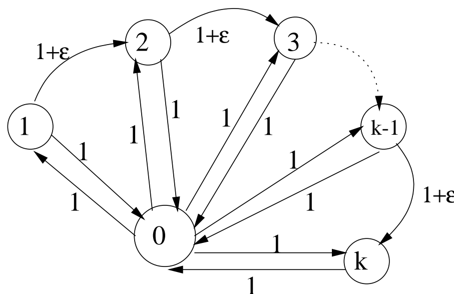

Figure 16: Pathologically Bad Example for the Primal-Dual

This was the worst-case example for the performance of primal-dual that we could get theoretically. Recall that the primal-dual output consists of all the arcs with cost one. Let's do some sensitivity analysis here. If we decrease the cost of arc $1 \to 2$ by 2ϵ, from $1+\epsilon$ to

$1-\epsilon$, then the primal-dual first will include $1 \to 2$ then $0 \to 1$, $2 \to 0$ and $0 \to i$, $i \to 0$ for all $i \in 3, \ldots, k$. In the result the cost of the solution will go down by $2-(1-\epsilon)=1+\epsilon$. the same kind of improvement can be achieved by decreasing the cost of any arc with cost $1+\epsilon$. On the other hand, making any arc with cost one cheaper by ϵ will improve the solution just by the same ϵ. Thus, the sensitivity analysis implies that the arcs with cost $1+\epsilon$ that were ignored by primal-dual have hidden values. So the cost-dropping procedures have the potential of improving the solution of primal-dual too.

Let's define the drop value of any arc e for primal-dual the same way we did for the branchings algorithm:

$$PDdrop(e) = \frac{c(PD) - c(PD')}{c(e)}\ , \tag{16}$$

where $c(PD')$ is the cost of the primal-dual output if the cost of arc e is dropped to 0.

The drop algorithm for primal-dual is given in Algorithm 5.2.

Algorithm 5.2 Primal-Dual Drop Algorithm

1: **while** $c(PD) > 0$ and $\exists\ e$ s.t. $drop(e) \geq 1$ **do**
2:    Let e^* be an arc with maximum drop
3:    Set $c(e^*) = 0$
4: **end while**
5: **return** union of edges in 0-cost output of primal-dual

Lemma 5 *Primal-dual-drop is a three-approximation algorithm.*

Proof: The proof is similar to that of Lemma 4. We still have (15) but this time for primal-dual, and it proves that the solution of primal-dual-drop can't be worse than the solution of primal-dual.

The only difference is that in this algorithm we explicitly check that the maximum drop value is at least one. Recall that in branchings-drop there was no need for that since the positive-cost solution implied that there is an arc with drop value at least 1. Our conjecture is that it is true also for primal-dual-drop and our computational results support this conjecture; but we could not prove it theoretically because the situation here is not as simple as in the case of branchings-drop. □

Our computational results show that branchings-drop and primal-dual-drop significantly improve the performance of the original branchings and primal-dual algorithms correspondingly. The results of primal-dual-drop are especially promising (see Section 6).

6. Computational Results

In order to evaluate the practicality and the relative performance of our algorithms, we have implemented several versions of them. The implemented algorithms are the branchings algorithm, the primal-dual algorithm, the penalty algorithm, and the drop algorithms (branchings and primal-dual versions). All implementations are for the case of the strong connectivity problem.

Below we discuss how the algorithms were implemented, and the sets of problems on which they were tested. Then we analyze the computational results.

6.1 Implementation Environment

Our goal is to consider 1) the performance of these algorithms relative to the optimal solution in order the compare them with each other, and 2) speed of execution. To achieve the goal, we have chosen to use a programming environment that provides maximum possible speed and good memory management for the algorithms. We have used C as our implementation environment. Adjacency lists are used for graph structures.

The integer programs were solved with CPLEX 6.6.1. We used a modified version of the mipex2.c file of CPLEX to call the MIP solver within our main C code. All experiments were performed on a conventional Sun UltraSparc workstation.

6.2 Problem Instances

There are, to our knowledge, no libraries of public instances for the strong connectivity problem, so we have constructed our own testbed of instances from other sources. We have generated random instances and also have adapted some of the instances from a library of public instances for the asymmetric traveling salesman problem (ATSP) to our problem. We put the randomly-generated instances into different categories based on their *size* and *cost type*.

Randomly-Generated Instances: Problem Sizes.

We distinguish two sizes of networks:

1) Instances for which we *could* find optimal solutions. We could find optimal solutions for networks with up to 50 nodes; we will refer to these networks as "small" networks. The basis for comparison of the algorithms in this case is naturally the optimal cost.

2) Instances for which we *could not* find an optimal solution. We will refer to these

networks as “large” networks. The number of nodes in our large networks are 100, 500, and 1000. What is the basis of comparison in this case?

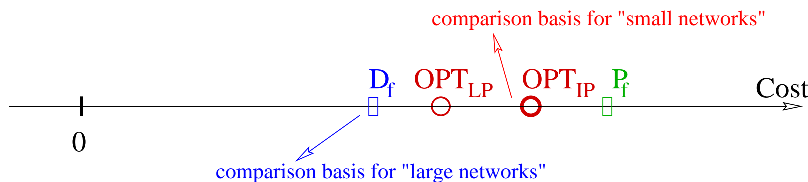

Figure 17: Cost Axis

Consider the cost axis of Figure 17. Here, D_f and P_f are the costs of the dual and primal outputs of the primal-dual algorithm. OPT_{IP} is the optimal cost of our problem (the integer program), and OPT_{LP} is the optimal cost of the LP-relaxation of the integer program. Since we always have $D_f \leq OPT_{LP} \leq OPT_{IP} \leq P_f$, D_f is a lower bound for OPT_{IP}, and we will use D_f as a basis for comparison of the algorithms for large networks. Note that OPT_{LP} is an even better lower bound, but we couldn't use it as a comparison basis because CPLEX had memory problems when setting up LPs for large problems.

Randomly-Generated Instances: Cost Types.

The costs of arcs were chosen uniformly at random in some range $[a, b]$. We distinguish three types of problems based on the choice of cost.

1) Costs don't obey the triangle inequality. In this case $a = 0$; b is 10 for small problems, and 50 or 100 for large problems.

2) The triangle inequality for the costs, $c_{uv} \leq c_{uw} + c_{wv}$ is achieved by narrowing the cost range to $[b/2, b]$. b is chosen the same way as in case 1.

Computational results show that making the coefficient of variation (that is, (standard deviation)/mean) of arc costs smaller as in case 2 really makes a difference.

3) The triangle inequality is achieved by first generating random costs, then assigning the shortest-path distances as arc costs.

Randomly-Generated Instances: Graph Densities.

Most of our problem instances are complete graphs. However, to find out how the graph density affects the performance of the algorithms, we have also run experiments on sparse graphs.

Instances Adapted from the ATSP Library

TSPLIB (2003) is a well-known public library that contains instances for a variety of routing problems, including the asymmetric traveling-salesman problem (ATSP). The instances for the ATSP fit our problem since they give data for real-life directed networks. Some of the instances are rather large, so we use the above-mentioned lower bound when comparing the performance of the algorithms for these instances.

6.3 Results of Experiments

The following five algorithms were tested: branchings, primal-dual, penalty, branchings-drop and primal-dual-drop. For each of the above-mentioned cases, the algorithms were tested on samples of several problem instances. For the randomly-generated instances, the sample sizes vary from 100 problems for small networks to five problems for networks with 1000 nodes. Unless specified otherwise, the results are given for complete graph instances.

Small Randomly-Generated Instances

We got the optimal solutions for all small instances by solving the corresponding integer programs. Table 1 gives average percent cost deviations from optimum for the above-mentioned five algorithms. The first number in each cell is for case 1 costs (without the triangle inequality), the second number is for case 2 costs (narrowed cost range), and the third number is for case 3 costs (shortest path distances). Average running times (in seconds) for the same set of instances are summarized in Table 2.

Table 1: Average Deviations from Optimum

	Branch	PrDual	Penalty	BrDrop	PrDDrop
10 nodes	11/22.8/10.9	4.2/16.8/3.5	1.01/6.7/1.2	2.13/4/2.5	0.1/0.65/0.1
15 nodes	12/27.6/12	3.9/17.5/3.6	1.6/8.1/2	2.8/4.5/3.2	0.27/1.1/0.15
20 nodes	11/25.6/8.8	3.8/16.8/3.3	1.9/7.9/1.4	4.2/4.9/2.7	0.04/1.5/0.08

Table 2: Average Running Times

	Branch	PrDual	Penalty	BrDrop	PrDDrop
10 nodes	0	0.01	0.02	0.55	0.69
15 nodes	0	0	0.15	4.45	8.15
20 nodes	0.2	0.2	0	23.8	46.6

To be consistent when comparing the results for small and large networks, we have also computed the average deviations of algorithm outputs from the lower bound. These results are summarized in Table 3.

Table 3: Average Deviations from Lower Bound for Small Problems

	Branch	PrDual	Penalty	BrDrop	PrDDrop
10 nodes	19.7/25/18.9	12.4/18.9/11.1	9.1/8.6/8.7	10.2/5.8/9.9	8.1/2.4/7.4
15 nodes	20.5/29.6/20	11.8/19.4/11.1	9.3/9.8/9.3	10.6/6.2/10.6	7.9/2.7/7.4
20 nodes	17.6/26.9/17.6	9.9/18/11.7	7.9/9/9.6	10.4/6/11	6/2.5/8.2

Large Randomly-Generated Instances

As mentioned above, the basis for comparison in this case is the cost of the dual output of the primal-dual algorithm. Table 4 compares the performances of the algorithms with respect to this lower bound. Table 5 summarizes the running times in seconds.

Table 4: Average Deviations from Lower Bound for Large Problems

	Branch	PrDual	Penalty
100 nodes	20.9/37/19	7.7/20/8.4	7.1/6.4/7.8
500 nodes	19.5/37/17.7	7.5/18.6/7.4	7.2/5.7/7.3
1000 nodes	17.6/37.2/17.5	7.2/18.4/7.7	7.2/13.6/7.7

Table 5: Average Running Times

	Branch	PrDual	Penalty
100 nodes	0.5	1.28	8.54
500 nodes	63	160	1561
1000 nodes	370	946	7050

Results on Sparse Graph Instances

For small instances, we have tested the algorithms both on dense and sparse graphs. Table 6 gives average percent cost deviations from optimum for different graph densities. The graph density is measured by the percentage of the number of arcs with respect to the number of arcs in a complete graph (the density of a complete graph is 100%). The costs are of type 1.

Table 6: Average Deviations from Optimum

density	Branch	PrDual	Penalty	BrDrop	PrDDrop
100%	11	4.2	1.01	2.13	0.1
80%	9	3	0.98	1.7	0.09
60%	10.3	4.9	1.3	2.3	0.16
40%	9.1	3.2	1.1	2	0.12
30%	9.4	3.1	0.8	1.4	0.12
20%	6.7	2.2	0.7	1.2	0.05

Instances from the TSPLIB Library

We tested the algorithms on four small problems and six large problems. As in the case of random problems, the basis of comparison is the optimal cost for small problems and the lower bound for large problems.

Table 7: Deviations from Optimum for Small Problems

	Branch	PrDual	Penalty	BrDrop	PrDDrop
br17.atsp	7.7	7.7	7.7	5.1	0
ftv33.atsp	26	22.2	12.5	6.5	3
ftv35.atsp	22.2	14.6	6.8	8.6	0.14
ftv38.atsp	22.9	14.1	8.24	8.17	1.12

Table 8: Deviations from Lower Bound for Large Problems

	Branch	PrDual	Penalty
kro124p.atsp	26.6	25.1	20.3
ftv170.tsp	19.7	17	17
rbg323.atsp	24.1	24.6	24.6
rbg358.atsp	12	9	9
rbg403.atsp	5.7	1.1	1.1
rbg443.atsp	10.1	5	5

6.4 Analysis of Results

Relative Performance of the Algorithms

Computational results show that the three-approximation primal-dual works well in practice. In particular, we didn't get any instance where it returns a solution worse than twice the optimum; this supports our hypothesis that the primal-dual is a two-approximation algorithm. Its performance is also much better than that of the branchings algorithm, while the running times of both algorithms are equally good.

The penalty and drop algorithms offer significant performance improvements over branchings and primal-dual algorithms. In particular, primal-dual-drop delivers impressive results: an optimal solution in most cases and very close to optimum otherwise. However, the penalty algorithm exhibits much better running time than primal-dual-drop; this is due to the fact that the number of primal-dual subroutines within the penalty algorithm is much smaller than the number of those subroutines within primal-dual-drop. Thus, the penalty algorithm can be considered as even a better alternative giving good tradeoff between the performance and running time.

We didn't run the drop algorithms for large networks because of their long running time. It is an open implementation issue how to reduce the running time of drop algorithms while keeping their good performance.

Analysis of Results for Different Cost Types

The algorithms perform equally well on instances with type 1 (without triangle inequality) and type 3 (shortest path distances) costs; but the performance worsens significantly on

instances with type 2 (narrowed range) costs. This concerns especially the branchings and primal-dual algorithms. We explain this by the following observation on primal-dual. For networks with ten nodes, we counted the number of edges in the output of primal-dual and in the optimal solution. The average number of edges in the primal-dual output (for 100 instances) is about 12 for all three cost types. The average number of edges in the optimal solutions is ten for the case of narrowed cost ranges, and about 11.2 for the other two cases.

The reason is the following. For the instances with type 2 costs the cost range is $[5, 10]$. Since the coefficient of variation in arc costs is relatively small, the optimal solution is just a cycle with ten arcs. On the other hand, the primal-dual output has 12 arcs. Since the extra arcs can't be much cheaper than the arcs in the optimal cycle, they push the cost up by 16.7% from the optimum. No wonder the penalty algorithm, whose main feature is reducing the number of arcs, makes significant improvements over primal-dual in this case.

For the instances with cost types 1 and 3, costs are in range $[0, 10]$. Some arcs are much cheaper than the others, and that is why the optimal solution might consist of more than ten cheap arcs. The average number is 11.2 for the optimal solutions and 12 for the primal-dual outputs; and correspondingly the cost deviation from optimum is not so big for the primal-dual.

Sparse vs. Dense Graphs

The performance of the algorithms is generally slightly better on sparser graphs. The improvement becomes significant when the graph density is reduced to 20%. For branchings and primal-dual algorithms, the average deviations from optimum are 6.7% and 2.2% in the case of 20%-dense graphs, while those numbers are 11% and 4.2% for 100%-dense graphs. This is not surprising if we consider that in 20%-dense graphs, the pool of possible solutions is much smaller than in 100%-dense graphs, and thus the algorithms are more likely to return optimal or near-optimal solutions.

7. Future Directions

Here are some problems related to our results that we are planning to consider in the future:

- *Improving the factor of three for primal-dual.* As mentioned before, the factor of three is known to be not tight but rather it is the result of our proof technique. Our conjecture is that $2 - 2/n$ is the real approximation factor, and computational results support this conjecture.

• *Extending the algorithms to other directed network design problems.* It is particularly interesting how our algorithms would extend to higher-connectivity networks and to the case of $\{0,1\}$-connectivity requirements between pairs of vertices. The main difficulty in these extensions is finding appropriate rules for choosing the violated subsets in the primal-dual method. For the strong connectivity problem, the natural choices for violated subsets were the strongly connected components and their complements (see Section 2.2). But it is not clear what should be the right choice when we consider, for example, the most famous $\{0,1\}$-connectivity problem, the directed Steiner tree problem. In this problem, a minimum-cost directed out-tree should be constructed from a root to a set of terminals. The subset requirement is one if and only if the subset contains the root and doesn't contain at least one of the terminals. It is easy to verify that if the strongly connected components and their complements are the only choices for being violated subsets, then after several iterations, no candidate for a violated subset could be left while the solution is still infeasible. So to make the algorithm work, a more sophisticated rule is needed for choosing the violated subsets. But that would also imply more complicated analysis for the approximation factor.

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments and suggestions.

References

Agrawal, A., P. Klein, R. Ravi. 1995. When trees collide: an approximation algorithm for the generalized Steiner problem on networks. *SIAM J. on Comput.* **24(3)** 445-456.

Edmonds, J. 1967. Optimum branchings. *J. of Res. of the National Bureau of Standards B* **71** 233-240.

Frederickson, G. N., J. Jaja. 1981. Approximation algorithms for several graph augmentation problems. *SIAM J. on Comput.* **10** 270-283.

Goemans, M. X., A. Goldberg, S. Plotkin, D. Shmoys, E. Tardos, D. P. Williamson. 1994. Approximations algorithms for network design problems. *Proc. of the 5th Annual Sympos. on Discrete Algorithms.* Association for Computing Machinery, New York. 223-232.

Goemans, M. X., D. P. Williamson. 1995. A general approximation technique for constrained

forest problem. *SIAM Journal on Comput.* **24** 296-317.

Held, M., R. Karp. 1970. The traveling-salesman problem and minimum spanning trees. *Oper. Res.* **18** 1138-1162.

Jain, K. 1998. A factor 2 approximation algorithm for the generalized Steiner network problem. *Proc. of 39th Annual Sympos. on Foundations of Comput. Sci.*, Palo Alto, CA. 448-457.

Jain, K., V. Vazirani. 1999. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and Lagrangian relaxation. *Proc. of 40th Annual Sympos. on Foundations of Comput. Sci.*, New York, NY. 2-13.

Melkonian, V., É. Tardos. 1999. Approximation algorithms for a directed network design problem. *Proc. of the 7th Internat. Integer Programming and Combin. Optim. Conf. (IPCO'99)*, Graz, Austria. 345-360.

Rajagopalan, S., V. Vazirani. 1999. On the bidirected cut relaxation for the metric Steiner tree problem. *Proc. of 10th Annual ACM-SIAM Sympos. on Discrete Algorithms.* Association for Computing Machinery, New York, 742-751.

Raz, R., S. Safra. 1997. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. *Proc. of the 29th Annual ACM Sympos. on the Theory of Comput.*, El Paso, Texas. 475-484.

TSPLIB library. 2003. Traveling salesman problem instances. ftp://ftp.zib.de/pub/Packages/mp-testdata/tsp/index.html.

Williamson, D. P., M. X. Goemans, M. Mihail, V. Vazirani. 1995. A primal-dual approximation algorithm for generalized Steiner network problems. *Combinatorica* **15** 435-454.

Zhu, A., S. Khuller, B. Raghavachari. 1999. A uniform framework for approximating weighted connectivity problems. *Proc. of 10th Annual Sympos. on Discrete Algorithms.* Association for Computing Machinery, New York, 937-938.

Accepted by Jan Karel Lenstra; received April 2002; revised July 2003; accepted November 2003.