Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2004-74

2004-12-09

# Improved Local Search Algorithms with Multi-Cycle Reduction for Minimum Concave Cost Network Flow Problems

Ruibiao Qiu and Jon Turner

The minimum concave cost network flow problem (MCCNFP) has many applications in areas such as telecommunication network design, facility location, production and inventory planning, and traffic scheduling and control. However, it is a well known NP-hard problem, and all existing search based exact algorithms are not practical for networks with even moderate numbers of vertices. Therefore, the research community also focuses on approximation algorithms to tackle the problems in practice. In this paper, we present an improved local search algorithm for the minimum concave cost network flow problem based on multi-cycle reduction. The original cycle reduction local search algorithm... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

## Recommended Citation

Qiu, Ruibiao and Turner, Jon, "Improved Local Search Algorithms with Multi-Cycle Reduction for Minimum Concave Cost Network Flow Problems" Report Number: WUCSE-2004-74 (2004). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1044

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1044

# Improved Local Search Algorithms with Multi-Cycle Reduction for Minimum Concave Cost Network Flow Problems

Ruibiao Qiu and Jon Turner

Complete Abstract:

The minimum concave cost network flow problem (MCCNFP) has many applications in areas such as telecommunication network design, facility location, production and inventory planning, and traffic scheduling and control. However, it is a well known NP-hard problem, and all existing search based exact algorithms are not practical for networks with even moderate numbers of vertices. Therefore, the research community also focuses on approximation algorithms to tackle the problems in practice. In this paper, we present an improved local search algorithm for the minimum concave cost network flow problem based on multi-cycle reduction. The original cycle reduction local search algorithm as proposed by Gallo and Sodini considers only negative cost single cycles; however, we find that such cycle reduction is not complete. We show that negative cost multi-cycles may exist in a network with concave edge costs that has no negative cost cycles, and an existing flow can be reduced to an adjacent neighboring flow with lower cost by redirecting flows along these negative multi-cycles. In this paper, we present an improved local search algorithm based on multi-cycle reduction. We evaluate our proposed algorithm in networks with a simple concave edge cost in different topologies and sizes. The experimental results show that the original cycle reduction algorithms can improve the quality of solutions obtained from a simple minimum cost augmentation approximation heuristic (LDF), and that a multi-cycle reduction yields more improvements; however, it reaches a point of diminished returns when we attempt to reduce more than bicycles.

Department of Computer Science & Engineering

2004-74

# Improved Local Search Algorithm with Multi-Cycle Reduction for Minimum Concave Cost Network Flow Problems

Authors: Qiu, Ruibiao; Turner, Jonathan S.

Abstract: The minimum concave cost network flow problem (MCCNFP) has many applications in areas such as telecommunication network design, facility location, production and inventory planning, and traffic scheduling and control. However, it is a well known NP-hard problem, and all existing search based exact algorithms are not practical for networks with even moderate numbers of vertices. Therefore, the research community also focuses on approximation algorithms to tackle the problems in practice. In this paper, we present an improved local search algorithm for the minimum concave cost network flow problem based on multi-cycle reduction. The original cycle reduction local search algorithm as proposed by Gallo and Sodini considers only negative cost single cycles; however, we find that such cycle reduction is not complete. We show that negative cost multi-cycles may exist in a network with concave edge costs that has no negative cost cycles, and an existing flow can be reduced to an adjacent neighboring flow with lower cost by redirecting flows along these negative multi-cycles. In this paper, we present an improved local search algorithm based on multi-cycle reduction.We evaluate our proposed algorithm in networks with a simple concave edge cost in different topologies and sizes. The experimental results show that the original cycle reduction algorithms can improve the quality of solutions obtained from a simple minimum cost augmentation approximation heuristic (LDF), and that a multi-cycle

Type of Report: Other

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

# Improved Local Search Algorithm with Multi-Cycle Reduction for Minimum Concave Cost Network Flow Problems

Ruibiao Qiu and Jon Turner

WUCSE-2004-74

December 9, 2004

Department of Computer Science and Engineering
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899, USA

## Abstract

The minimum concave cost network flow problem (MCCNFP) has many applications in areas such as telecommunication network design, facility location, production and inventory planning, and traffic scheduling and control. However, it is a well known NP-hard problem, and all existing search based exact algorithms are not practical for networks with even moderate numbers of vertices. Therefore, the research community also focuses on approximation algorithms to tackle the problems in practice. In this paper, we present an improved local search algorithm for the minimum concave cost network flow problem based on multi-cycle reduction. The original cycle reduction local search algorithm as proposed by Gallo and Sodini considers only negative cost single cycles; however, we find that such cycle reduction is not complete. We show that negative cost multi-cycles may exist in a network with concave edge costs that has no negative cost cycles, and an existing flow can be reduced to an adjacent neighboring flow with lower cost by redirecting flows along these negative multi-cycles. In this paper, we present an improved local search algorithm based on multi-cycle reduction. We evaluate our proposed algorithm in networks with a simple concave edge cost in different topologies and sizes. The experimental results show that the original cycle reduction algorithms can improve the quality of solutions obtained from a simple minimum cost augmentation approximation heuristic (LDF), and that a multi-cycle reduction yields more improvements; however, it reaches a point of diminished returns when we attempt to reduce more than bicycles.

# Improved Local Search Algorithm with Multi-Cycle Reduction for Minimum Concave Cost Network Flow Problems

Ruibiao Qiu Jonathan S. Turner
{ruibiao,jst}@arl.wustl.edu
Applied Research Laboratory
Department of Computer Science and Engineering
Washington University
St. Louis, MO 63130, USA

## 1. Introduction

Given a directed graph $G = (V, E)$ with a source vertex $s$ and a sink vertex $t$, where $V$ and $E$ are the sets of vertices and edges, respectively. Assume each edge $e$ can carry flow $f_e$ subject to some capacity constraint $b_e$. If $b_e$ is $\infty$ for every edge $e$, the network is uncapacitated; otherwise, it is capacitated. In the maximum network flow problem (MNFP), we seek a subset of edges with non-zero flow that have the maximum total flow from $s$ to $t$. When we define a cost function $c_e(f)$ associated with the flow $f$ carried on an edge $e$, the maximum network flow problem can be generalized as the minimum cost network flow problem (MCNFP), which is a problem of finding the maximum flow with the minimum total cost among all other maximum flows. If the edge cost functions in a network are all linear in the amount of flow on edges, this problem is referred to as the minimum linear cost network flow problem (MLCNFP), and can be solved efficiently. In contrast, if the edge cost function $c_e(f)$ on an edge $e$ is a concave function of the flow carried on the edge $f$ (that is, $2c_e(f_1) > c_e(f_0) + c_e(f_2), f_0 < f_1 < f_2$), the problem becomes the minimum concave cost network flow problem (MCCNFP).

The minimum concave cost network flow problem has a variety of real world applications and is also of great theoretical interest. One major application of MCCNFP in practice is to telecommunication network design problems. For example, our previous study showed that the configuration problem for reserved delivery subnetworks (RDS) can be formulated as a MCCNFP [1]. In an RDS, network bandwidth must be reserved carefully on selected links that connect a service provider to access routers at locations with traffic demands to a server location, such that the user demands are satisfied and the network resources are utilized efficiently. Because of the traffic variation in individual flows, it is advantageous to aggregate certain traffic flows together as they go from the server to the remote sites. This benefit of flow aggregation is reflected in a concave edge cost function that grows more slowly as the average aggregated traffic on an edge increases. The RDS configuration problem can be formulated as a MCCNFP as follows: define the backbone network as $G = (V, E)$ with the server location as the source vertex $r$; define the locations with end user demand as a set of sink vertices $S = \{s_1, s_2, \cdots, s_m\} \subset V$, each with a sink demand of $\mu_i (1 \leq i \leq m)$. Add a pseudo sink vertex $t$, and connect only sink vertices to $t$ with zero-cost edges and a capacity equal to the demand of the connected sink. A minimum concave cost network flow in this transformed network corresponds to an optimal RDS because the source vertex connect to all sink vertices, and the path from the source vertex to a sink vertex has sufficient flow to satisfy the demand of the sink. Thus, we can find an optimal RDS by solving the MCCNFP on the transformed RDS configuration problem. In this paper, we focus on the MCCNFP in a uncapacitated network because in an RDS application, the backbone network usually has much greater bandwidth resource than the need of an individual information service provider, and the capacity constraints are not likely an issue. In addition, the optimal flow in a uncapacitated network takes the shape of a tree.

It is well known that MCCNFP is NP-hard [2], and the existing exact algorithms are all search-based algorithms with some intelligent enumeration methods [3, 4]. However, these algorithms do not scale well for networks with even moderate numbers of nodes, and thus are impractical in real applications. In order to provide solutions for MCCNFP in practice, a number of approximation algorithms have been studied and proposed. Among these approximation algorithms, local search algorithms for MCCNFP has enjoyed tremendous success in solving large and complex problems in practice. Given an existing solution, a local search algorithm examines the "neighborhood" of the existing solution, and identifies a solution that is locally optimal within the "neighborhood". The "neighborhood" is defined as a set of solutions that are reachable from an existing solution with a simple operation. In the case of MCCNFP, it is known that the optimal solution is an extreme flow, which is a tree in an uncapacitated network. By taking advantage of this property, a local search algorithm finds a local optimal solution from an extreme flow by examining the adjacent extreme flows reachable from the existing extreme flow with a simple operation, although it may be trapped in a local optimal solution different from a global optimal one.

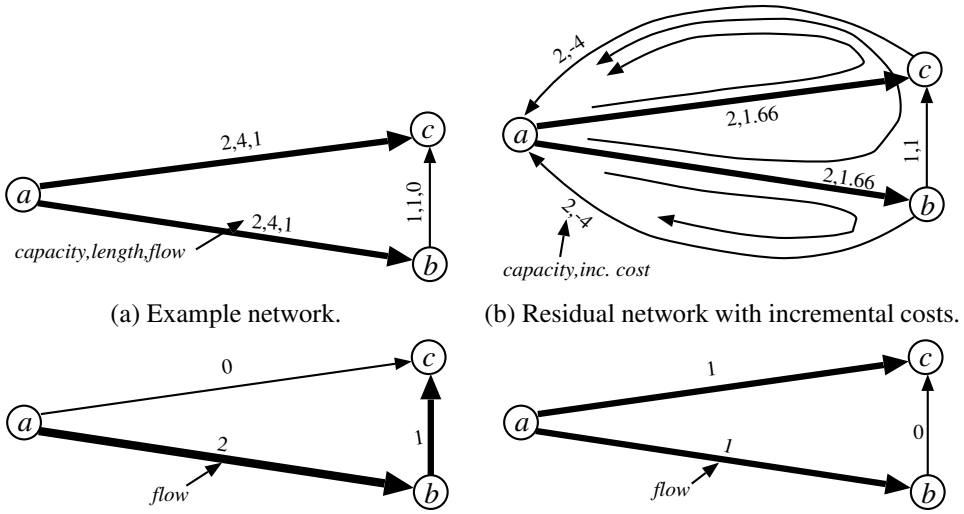
In this paper, we observe the existence of negative cost multi-cycles in a network with concave edge costs. That is, even though there is no negative cost cycle, there could exist a set of cycles with a common path that has a negative total cost. Based on this observation, the cycle reduction algorithm in [5] is not able to include all adjacent extreme flows, and therefore is limited and incomplete. Towards this end, we present an improved local search algorithm for MCCNFP with bicycle reduction method to consider both negative cost single cycles and bicycles. We apply both the original and improved local search algorithms to networks with a simple concave edge cost function in our experiments, and demonstrate the improvement of solution quality. Although we focus on negative cost bicycles in this paper as they are the most likely negative cost multi-cycles, we also show that the bicycle reduction algorithm can be generalized to handle other negative cost multi-cycles too.

The rest of the paper is organized as follows: Section 2 briefly discusses the local search algorithms using cycle reduction strategy, and explains why a naive cycle reduction approach fails in a network with concave edge costs. The local search algorithm with cycle reduction method proposed by Gallo and Sodini [5] is reviewed in Section 3. We also describe a path compression technique to the original algorithm, reducing the number of shortest path trees computations. We illustrate in Section 4 that how a local minimum can be sub-optimal because of the existence of negative cost bicycles. In Section 5, we describe the improved local search algorithm with bicycle reduction to identify and remove negative cost bicycles. Section 6 outlines the simulation environment and analyzes the simulation results. Section 7 discusses the more general case of negative cost multi-cycles and generalizes the bicycle reduction algorithm to handle negative cost multi-cycles. Some related work is outlined and compared in Section 8, and Section 9 concludes the paper.

## 2. Local Search Algorithms Using Cycle Reduction Strategy

Local search [6] is a well-known approximation method that is applicable to almost all combinatorial optimization problems. Although it can not determine if the best solution found so far is optimal, local search has enjoyed tremendous success in solving large and complex combinatorial optimization problems in practice. When a local search method is applied to a problem, a simple operation is employed to transform an existing feasible solution to a neighboring feasible solution, and a neighboring solution with lower cost is chosen and further explored until no further improvement can be made. For minimum cost network flow problems, a local search method searches for a flow with the least cost among all neighboring feasible flows obtainable from an existing feasible flow with a simple operation. As for the choice of the simple operation in a local search algorithm, the negative cost cycle reduction method is a natural candidate. The negative cost cycle reduction works by "pushing" flows along a negative cost cycle to transform an existing feasible flow to another feasible flow with lower total cost. This operation is exactly what we expect for a local search algorithm. In addition, the negative cost cycle reduction method can form the basis of efficient algorithms for the minimum cost network flow problems in networks with linear edge costs [7].

However, we must be careful when we apply the negative cost reduction method in a network with concave edge costs. The difficulty of applying negative cost cycle reduction in networks with concave edge costs can be illustrated in an example in Figure 1. We show a small network in Figure 1(a) with the capacity, length, and current flow of each




(a) Example network.

(b) Residual network with incremental costs.

(c) A good choice of negative cost cycle.

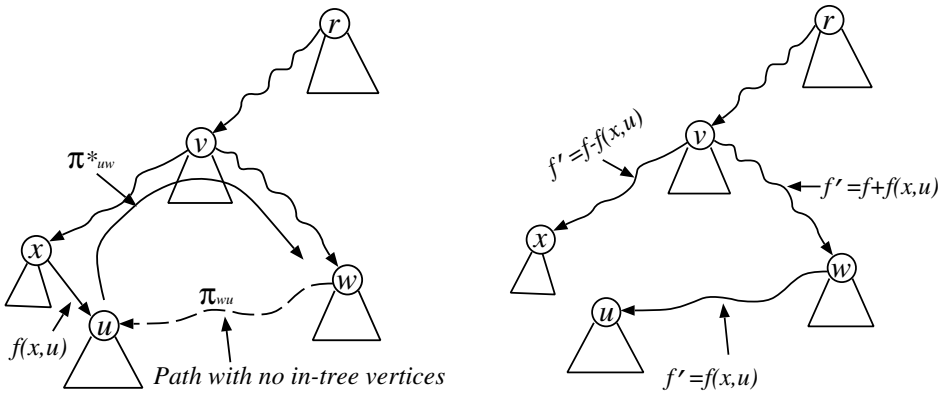
(d) A bad choice of negative cost cycle.

**Figure 1:** Problems of negative cost cycle reduction in a network with concave edge costs.

edge in the labels. In this simple network, $a$ is the source vertex that supplies the sink vertices $b$ and $c$. Currently, there is a unit flow on edges $(a, b)$ and $(a, c)$. For this example, we adopt a simple concave edge cost function: the cost $c_e(\mu)$ of an edge $e$ with an average flow of $\mu$ is defined as $c_e(\mu) = l\mu^{1/2}$, where $l$ is the length of the edge. Thus, the existing flow has a total cost of 8. The incremental cost on $e = (u, v)$ with a flow increment of $\Delta$ is

$$\Delta c_e(\Delta) = \begin{cases} l(\sqrt{\mu + \Delta} - \sqrt{\mu}) & \text{if there is no flow on reverse edge } (v, u) \\ l(\sqrt{\mu - \Delta} - \sqrt{\mu}) & \text{if there is flow on reverse edge } (v, u), \text{ and } \Delta < \mu \\ l(\sqrt{\Delta - \mu} - \sqrt{\mu}) & \text{if there is flow on reverse edge } (v, u), \text{ and } \Delta \geq \mu \end{cases}$$

Figure 1(b) shows the corresponding residual graph of the current flow when a unit of flow will be changed on all edges. The incremental costs for a unit flow increment are shown on the labels of Figure 1(b). Clearly, Figure 1(b) has three negative cycles: namely, $\{(a, c), (c, a)\}$, $\{(a, b), (b, a)\}$, and $\{(a, b), (b, c), (c, a)\}$ with cost of $-2.34$, $-2.34$ and $-1.34$, respectively. With the negative cost cycle reduction method, we attempt to redirect flow along a negative cost cycle such that the negative cost cycle is removed and the total cost is lowered after the flow redirection. If we choose the $\{(a, b), (b, c), (c, a)\}$ cycle, and push a unit flow along it, we could get a new flow with a lower cost of $6.656$, and there is no more negative cost cycle in the residual graph (Figure 1(c)). However, if the $\{(a, c), (c, a)\}$ cycle is picked, and flow is redirected this cycle, it would neither remove the negative cost cycle, nor lower the total flow cost (Figure 1(d)). In fact, any edge in an existing flow has a two-edge negative cycle in the residual graph in such a network with concave edge costs. If any of such two-edge negative cycle is chosen by the cycle reduction algorithm, the local search algorithm is stalled. This is caused by the concavity of the edge cost function because on such an edge the absolute incremental costs of increasing and decreasing the same amount of flow are different, causing the asymmetric incremental costs and a "false" negative cycle in the residual graph. In contrast, in a network with linear edge costs, the absolute incremental costs of the two opposite edges are the same, but with different signs. So, there is no negative cost cycle with only two edges in a network with linear edge cost. Thus, we can not implement negative cost cycle reduction in a local search algorithm if we can not distinguish two-edge negative cost cycles from other legitimate negative cost cycles. In particular, we can not pick an arbitrary negative cost cycle and push flow along it in a network with concave edge costs. There are a number of efficient negative cycle reduction algorithms for minimum cost flow problems, such as the Minimum Mean Cycle Canceling algorithm [8] and the most helpful cycle canceling algorithm [9]. However, because they provide no efficient way to characterize the two-edged negative cycles that we want to avoid, these negative cycle reduction algorithms are not good candidates for local search for MCCNFP.




(a) Original extreme flow. (b) The neighboring extreme flow obtained.

**Figure 2:** Original cycle reduction algorithm.

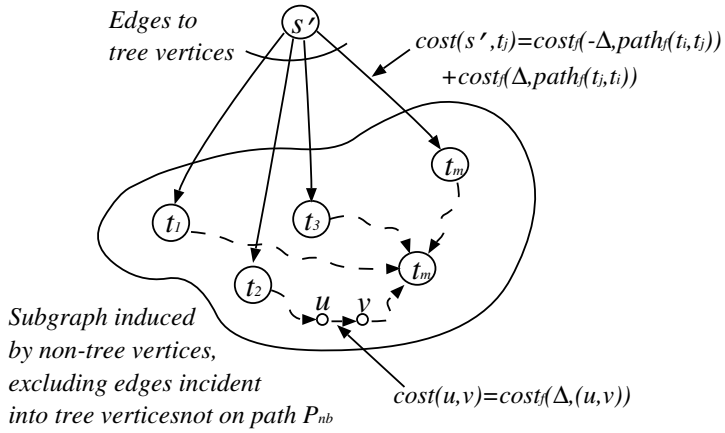



**Figure 3:** Finding the best solution for "target" $t_i$, where $cost_f(x,p)$ = the incremental cost of adding $x$ units of flow along path $p$, relative to existing flow $f$, $\Delta = f(p_f(t_i), t_i)$ is the flow into $t_i$, $p_f(u)$ = the parent of $u$ in the tree defined by $f$, and $path_f(t_i, t_j)$ = the path from the nearest common ancestor of $t_i$ and $t_j$ to $t_i$ in the tree defined by $f$. Note, for the original cycle reduction algorithm, $P_{nb}$ is only the vertex $t_i$. For the improved algorithm with compressed paths, $P_{nb}$ is the longest "non-branching" in-tree path to $t_i$.

# 3. Local Search Algorithm with Cycle Reduction

## 3.1. Gallo-Sodini Cycle Reduction Algorithm

The local search algorithm for uncapacitated networks presented in [5] provides an effective way to implement negative cost cycle reduction that is more efficient than an algorithm that searches for any negative cycles. An extreme flow in an uncapacitated network is a feasible flow in a network that the edges with non-zero flow constitute a tree with the source vertex at the root of the tree and all sink vertices at the leave. The Gallo-Sodini cycle reduction algorithm is based on the idea that an extreme flow $x'$ is adjacent to an existing extreme flow $x$ if and only if all edges that are in $x'$ but not in $x$ constitute a path connecting only two vertices in $x$.

Figure 2 and Figure 3 illustrate the basic operations of the Gallo-Sodini algorithm for finding a neighboring extreme flow from an existing extreme flow. The original extreme flow $f$ is shown as a tree in Figure 2(a). For a vertex $u$ with

an incoming flow of $f(x,u)$ in the tree defined by the existing flow, first find the undirectional path $\pi^*_{uv}$ from $u$ to any other tree vertex $v$ that is not in the subtree rooted at $u$. Notice that $\pi^*_{uv}$ could be composed of two directed paths: one from the nearest common ancestor $c$ of $u$ and $v$ to $u$, and the other from $c$ to $v$. Compute the incremental cost $c^*_v$ of adding $f(x,u)$ units of flow on $\pi^*_{uv}$ as $c^*_v = cost_f(-f(p_f(u),u), path_f(u,v)) + cost_f(f(p_f(u),u), path_f(v,u))$, where $cost_f(x,p)$ is the incremental cost of adding $x$ units of flow along path $p$, relative to existing flow $f$, $p_f(u)$ is the parent of $u$ in the tree defined by $f$, and $path_f(u,v)$ is the path from the nearest common ancestor of $u$ and $v$ to $u$ in the tree defined by $f$. For example, the path from $u$ to $w$ ($\pi^*_{uw}$) is shown in Figure 2(a) as well as a directed non-tree path $\pi_{wu}$ from $w$ to $u$ that connects two tree vertices $w$ and $u$.

Next, a new network is derived for a specific tree vertex $t_i$ as shown in Figure 3. In this transformed network, a pseudo source vertex $s'$ is introduced, and $s'$ connects to every tree vertex $t_j$ defined by the existing flow with a direct edge $(s', t_j)$, except for $t_i$. All tree edges are removed; all non-tree edges incident to any tree vertex in the existing flow except $t_i$ are removed too. It we define $\Delta = f(p_f(p_f(t_i), t_i)$ to be the flow into $t_i$, an edge $(s', t_j)$ is assigned a length of $cost(s', t_j) = c_{t_j} = cost_f(-\Delta, path_f(t_i, t_j)) + cost_f(\Delta, path_f(t_j, t_i))$, and a non-tree edge $(u,v)$ is assigned a length $cost(u,v)$ the same as the incremental cost of adding $\Delta$ units of flow on that edge, $cost(u,v) = cost_f(\Delta, (u,v))$. We then try to find the shortest path from $s'$ to $t_i$ in the transformed network. After the shortest path is found, the last vertex $w$ on the path from $s'$ to $t_i$ is identified, and $\Delta$ units of flow is redirected along the undirectional path $\pi^*_{t_i w}$ and then along the directed path $\pi_{wt_i}$. The resulting flow is a neighboring extreme flow to the original flow, as shown in Figure 2(b). If the modified flow has a lower cost than the original flow, the above procedure is repeated to find a lower cost neighboring flow of the new flow. Otherwise, the original flow is restored.

The Gallo-Sodini cycle reduction algorithm can be briefly described by the following pseudo code:

Find an extreme flow $x^0$.
Repeat
  For each tree vertex $t_i$ with an incoming flow of $\Delta$.
    For each tree vertex $t_j \neq t_i$,
      Compute the incremental cost $c^*_{t_j}$ for redirecting $\Delta$ units of flow along the undirectional tree path $\pi^*_{t_i,t_j}$.
    For each non-tree edge $(u,v)$,
      Compute incremental cost $c_{uv}$ of adding $\Delta$ units of flow.
    Let $G' = (V', E')$ be the subgraph induced by non-tree edges.
    $V' \leftarrow V' \cup \{s'\}$,
    $E' \leftarrow E' \cup \{(s', v) | v$ is a tree vertex, and $v \neq t_i\} - \{(w, v) | v$ is a tree vertex, and $v \neq t_i\}$,
    For each edge $(s', t_j) \in E'$, assign a length of $c^*_{t_j}$, for any other edge $(u,v) \in E'$, assign a length of $c_{uv}$.
    Find the shortest path from $s'$ to $t_i$ in $G'$.
    Let edge $(s', w)$ be on the shortest path from $s'$ to $t_i$.
    Redirect $\Delta$ units of flow along paths $\pi^*_{t_i,w}$ and $\pi_{w,t_i}$, and obtain an updated flow $x'$.
    If the updated flow $x'$ has a higher cost than the current flow,
      restore the original flow.
until no flow with lower cost can be obtained.

Note that this algorithm transforms the flow to the first improved neighboring extreme flow found. An alternative is to check all neighboring extreme flows and then transform to the best neighboring extreme flow. However, as suggested by the empirical results in [3], transforming to the first improved neighbor generally requires $25 - 40\%$ fewer shortest path computations than the best neighbor algorithm, and yields results of comparable quality.

**Complexity Analysis** Let $n$ and $m$ be the numbers of vertices and edges in the network, for each flow, the cycle reduction algorithm may need to check $O(n)$ vertices before it can determine if a neighboring extreme flow with lower cost exist [5]. For the tree defined by an existing flow with $k (k \leq n)$ vertices, we need to solve $k$ nearest common ancestor problems, each taking $O(k)$ time. We also need to compute $2k^2$ incremental path costs. In addition, we need make $k$ shortest path tree computation. The checking procedure is dominated by the single source shortest path computation. If we denote $S(n,m)$ as the time complexity of a single source shortest path algorithm in a graph with

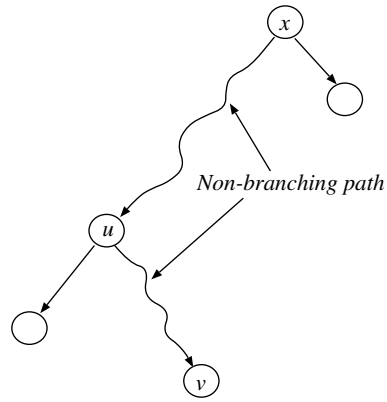


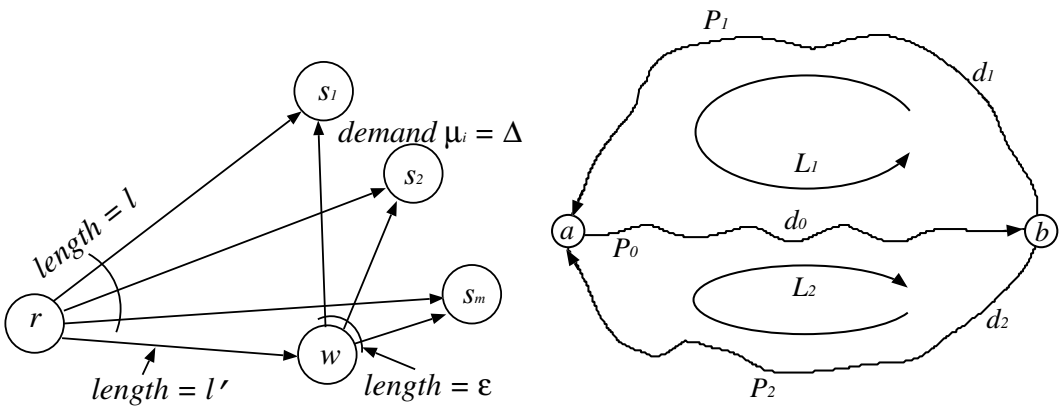

**Figure 4:** Path compression in the cycle reduction algorithm.

$n$ vertices and $m$ edges, then the time complexity for finding a neighboring flow with lower cost in the cycle reduction algorithm is $O(nS(n,m))$, or $O(n(n+m)\log n)$ if the single source shortest path algorithm is implemented efficiently.

## 3.2. Performance Improvement in Cycle Reduction Algorithm with Compressed Paths

The original cycle reduction algorithm by Gallo and Sodini has to check every tree vertex for negative cost cycles. If there are $k$ vertices in the tree defined by the existing flow, it requires $k$ shortest path tree computation. However, as Guisewite and Pardalos noted in [3], it is not necessary to check every tree vertices. Instead, we can check all the vertices on a non-branching path in the tree simultaneously. Figure 4 shows some non-branching paths in the tree defined by an existing flow. In this figure, $x$ and $u$ are two branching vertices in the tree, and $v$ is a sink vertex. $u$ and $x$ are possible sink vertices too. To determine the best alternative path into a tree vertex, it is sufficient to construct a shortest path tree for every non-branching path in the tree defined by the existing flow. For the example in Figure 4, instead of computing a shortest path tree for every vertex on the $x \to u$ and $u \to y$ paths, we only need to compute two shortest path trees. Because we consider all vertices on a non-branching path simultaneously, we refer to this improvement heuristic as path compression. If there are $m$ sink vertices, we only need to compute at most $2m-1$ shortest path trees because there are at most $2m-1$ non-branching paths in a tree. In contrast, if there are $n$ tree vertices, the original cycle reduction algorithm has to compute all $n$ shortest path trees to find the local optimal. It is easy to see that $n > 2m-1$, and the performance improvement could be very substantial. The following pseudo code outlines the IMPROVED cycle reduction algorithm with path compression:

Repeat
  For each leaf or branching tree vertex $t_i$ with an incoming flow of $\Delta$,
    For each tree vertex $t_j \neq t_i$,
      Compute $c^*_{t_j}$ the incremental cost of redirecting $\Delta$ units of flow along the undirected tree path $\pi^*_{t_i,t_j}$.
    For each non-tree edge $(u,v)$,
      Compute incremental cost $c_{uv}$ of adding $\Delta$ units of flow on $(u,v)$.
    Let $b(t_i)$ be the nearest branching ancestor.
    $G' = (V', E')$ is the subgraph induced by non-tree edges and edges on the non-branching path $(b(t_i), t_i)$.
      $V' \leftarrow V' \cup \{s'\}$
      $E' \leftarrow E' \cup \{(s',v) | v \text{ is a tree vertex, and } v \neq t_i\} - \{(w,v) | v \text{ is a tree vertex, and } v \notin (b(t_i), t_i)\}$,
    For each edge $(s', t_j) \in E'$, assign a length of $c^*_{t_j}$; for any other edge $(u,v) \in E'$, assign a length of $c_{uv}$.
    Find the shortest path from $s'$ to $t_i$ in $G'$.
    Let edge $(s', w)$ be on the shortest path from $s'$ to $t_i$ in $G'$.
    Redirect $\Delta$ units of flow along paths $\pi^*_{t_i,w}$ and $\pi_{w,t_i}$, and obtain an updated flow $x'$.




(a) An example network.

(b) A general negative cost bicycle.

**Figure 5:** A simple negative cost bicycle example.

If the update flow $x'$ has a higher cost than the current flow,
restore the original flow.
until no flow with lower cost can be obtained.

**Complexity Analysis** Let $n$ be the number of vertices in the network, and $k$ be the number of sink vertices. The improved cycle reduction algorithm only needs to compute at most $2k - 1$ shortest path tree for the non-branching paths in the tree defined by the existing flow to find the local minimal, as in contrast with $n$ shortest path computations in the original cycle reduction algorithm. This improvement speeds up the search for each flow derived from the initial flow, and therefore the whole local search procedure. Because the shortest path tree computation is the dominating factor of the time complexity of the cycle reduction algorithm, the path compression heuristic greatly improves the performance of the local search.

These local search algorithms essentially enumerate all neighboring extreme flows reachable from an existing extreme flow by redirecting flows along a negative cost cycles. In a network with linear edge costs, the resulting flow has no neighboring flow that has lower cost because negative cost cycles are sufficient to find local optimal in such a network. However, in a network with concave edge costs, such as an RDS, the result from the original cycle reduction algorithm does not necessarily include all possible neighboring extreme flows with lower costs as we demonstrate in the next section.

# 4. Negative Cost Bicycles in Concave Cost Networks

In this section, we show that the cycle reduction algorithm can be sub-optimal in a network with concave edge costs. Consider the example network shown in Figure 5(a). The source vertex $r$ connects to $m$ sink vertices with edges of length $l$. $r$ also connects to an intermediate vertex $w$ with an edge of length $l'$ that is slightly shorter than $l$. $w$ also connects to each sink vertex with an edge of length of $\epsilon$. Each sink vertex is associated with a demand of $\Delta$. For simplicity, we still adopt the simple concave edge cost function $cost(x, (u, v)) = l(u, v)x^{1/2}$ of a flow $x$ on an edge $(u, v)$ with a length of $l(u, v)$. With this cost function, the optimal cost is $\epsilon\Delta^{1/2}m + l'(\Delta m)^{1/2}$, while the result based on the shortest path tree (which is the local optima) has a cost of $l\Delta^{1/2}m$. So, if $\epsilon \leq l/m^{1/2}$, the cost ratio of the two solutions is no less than $m^{1/2}/2$, which can be arbitrarily large. This example suggests that reduction on negative cost cycles alone does not guarantee a results with sufficient quality, and better solutions can be reached by redirecting flow along subgraphs with special structures. For the example network in Figure 5(a), we notice that we can reach a

neighboring flow with lower cost by adding $2\Delta$ units of flow along $(r, w)$, and $\Delta$ units of flow along $(w, s_1)$, $(s_1, r)$, $(w, s_2)$, and $(s_2, r)$. The paths we redirect flow on constitute a subnetwork with special structures that we will explore in this section.

In a network with concave link costs, there could exist negative bicycles such as the one in Figure 5(a) that could transform an existing flow to a flow with lower cost. We define a *negative cost bicycle* as a pair of directed cycles that share a common segment, with the remainder of the cycles edge disjoint. When we add flow along the two cycles, the total cost of the resulting flow is lower than the original flow. A general negative cost bicycle is illustrated in Fig. 5(b) that has a pair of vertices $a$ and $b$. There is a common path $P_0$ from $a$ to $b$, and two paths $P_1$ and $P_2$ from $b$ to $a$. The sum of the cost of all these path is negative. Let $d_0$ be the length of the common segment of the negative cost bicycle, and $d_1$ and $d_2$ be the length of the disjoint segments. Then, the incremental cost of adding a unit flow along the bicycle could be expressed as $(1+\epsilon)d_0 + d_1 + d_2$, where $0 \leq \epsilon \leq 1$. If $\epsilon = 1$, the cost of the bicycle is equal to the sum of the cost of both cycles with the usual definition of flow costs. If $\epsilon = 0$, the bicycle is only charged once for the shared segment. Any other $0 < \epsilon < 1$ would result in a cost falls in between, showing the benefits of path sharing. It is easy to see that negative cost bicycle is just one subnetwork structure that can lead to lower cost neighboring flows. However, we first focus on finding negative cost bicycles only, because they are more likely to appear in an existing flow, and therefore have greater effects on final costs. We will generalize to deal with more complex subnetwork structures than bicycles in a later section. However, the cost benefits could be offset by the computational complexity of exploring more complicated structures.

For a general negative cost bicycle in a network with the simple concave edge cost function as defined in the example network, when we push flow $\Delta$ along the negative cost bicycle, we add $2\Delta$ flow on the common segment of the bicycle $P_0$, and $\Delta$ on the disjoint paths $P_1$ and $P_2$. Thus, the incremental cost $C_\Delta$ for a flow increment $\Delta$ can be expressed as

$$\begin{aligned} C_\Delta &= \sum_{i \in P_0} l_i(\sqrt{\mu_i + 2\Delta} - \sqrt{\mu_i}) \\ &- \sum_{i \in P_1^-} l_i(\sqrt{\mu_i} - \sqrt{\mu_i - d}) + \sum_{i \in P_1^+} l_i(\sqrt{\mu_i + d} - \sqrt{\mu_i}) \\ &- \sum_{i \in P_2^-} l_i(\sqrt{\mu_i} - \sqrt{\mu_i - 2\Delta + d}) + \sum_{i \in P_2^+} l_i(\sqrt{\mu_i + 2\Delta - d} - \sqrt{\mu_i}) \end{aligned}$$

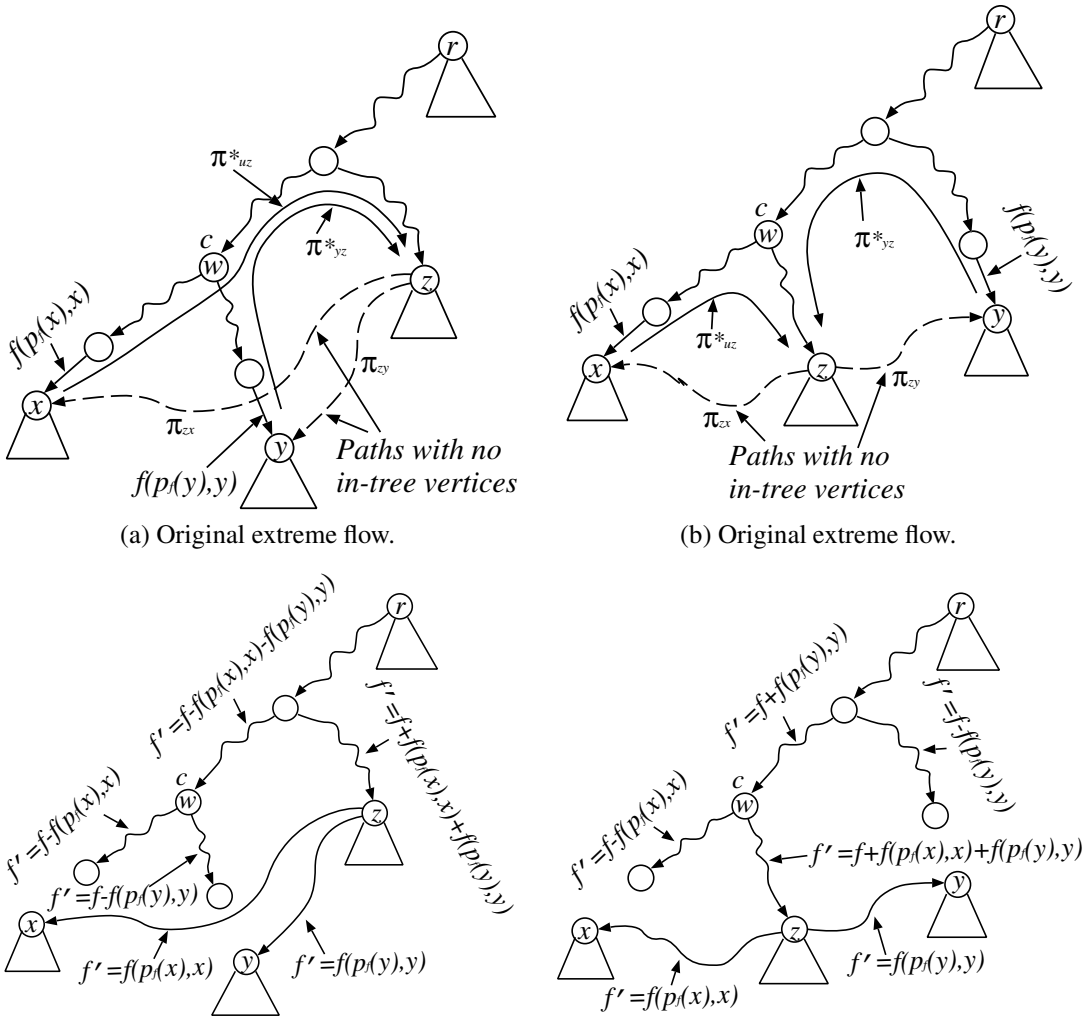
where $d$ is the flow added on the path $P_1$, $P_1^+$ is the set of links in path $P_1$ that have flows in the same direction as $P_1$, and $P_1^-$ is the set of links in path $P_1$ that have flows in the opposite direction of $P_1$; $P_2^+$ and $P_2^-$ are the sets similarly defined on $P_2$. Because this is a negative cost bicycle, $C_\Delta < 0$ for some $\Delta$. We must identify these negative cost bicycles with specific flow increment $\Delta$ to reduce the cost of an existing flow.

# 5. Bicycle Reduction Algorithm

As we described previously, an adjacent extreme flow with lower cost can be reached by redirecting flow along a negative cost bicycle. Thus, in order to extend the cycle reduction algorithm, we must efficiently identify these negative cost bicycles after the negative cost single cycles are all removed by the original cycle reduction algorithm.

In the original cycle reduction algorithm, for a vertex $u$ in the existing flow, we find another vertex $v$ in the existing flow, where the undirectional path in the existing flow $\pi^*_{uv}$ and the path $\pi_{vu}$ not used by the existing flow form a minimum cost cycle. If it is a negative cycle, a neighboring extreme flow with lower cost can be reached by redirecting flow along this cycle.

In contrast to a negative cost single cycle, a negative cost bicycle consists of two cycles with a common path segment. Therefore, in order to find a negative cost bicycle, we start with two vertices, $x$ and $y$, neither of which is a non-branching vertex in the tree defined by the existing flow. We then search for a third tree vertex $z$, through which




(a) Original extreme flow.

(b) Original extreme flow.

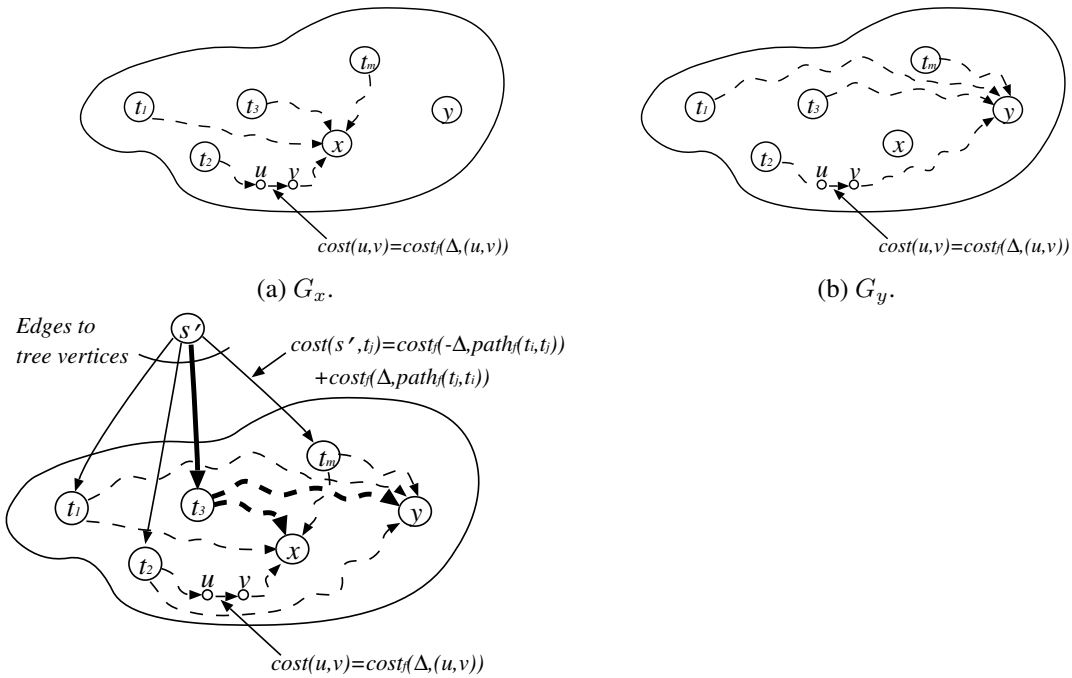
(c) The neighboring extreme flow obtained.

(d) The neighboring extreme flow obtained.

**Figure 6:** Bicycle reduction algorithm.

there is a pair of directed non-tree paths $\pi_{zx}$ and $\pi_{zy}$. We define $z$ as the *optimal split point*. In addition, there is another vertex $w$ on both undirectional tree paths $\pi^*_{xz}$ from $x$ to $z$ and $\pi^*_{yz}$ from $y$ to $z$. We define $w$ as the *optimal merge point*. The tree path $\pi^*_{wz}$ is the common segment of the bicycle, and the two cycles are $(\pi^*_{xw}, \pi^*_{wz}, \pi_{zx})$ and $(\pi^*_{yw}, \pi^*_{wz}, \pi_{zy})$. Figure 6(a) and Figure 6(b) show two example negative cost bicycles. Let $f_x$ and $f_y$ be the amount of flow into $x$ and $y$ respectively in the existing flow, after redirecting $f_x$ units of flow along $(\pi^*_{xw}, \pi^*_{wz}, \pi_{zx})$ and $f_y$ units of flow along $(\pi^*_{yw}, \pi^*_{wz}, \pi_{zy})$, the adjacent extreme flows are shown in Figure 6(c) and Figure 6(d).

Given such a pair of vertices $x$ and $y$ in a tree defined by an existing flow, we first observe that the optimal split point $z$ can not be on the undirectional tree path $\pi^*_{xy}$ between $x$ and $y$. Because if $z$ is on $\pi^*_{xy}$, the optimal merge point $w$ must be the same vertex as $z$. This means that there is no common path segment in the two cycles $(\pi^*_{xz}, \pi_{zx})$ and $(\pi^*_{yz}, \pi_{zy})$, and thus not a bicycle. In addition, it is clear that the optimal split point can not be in the subtrees rooted at neither $x$ nor $y$, because it will not lead to a bicycle either. Based on these observations, we limit our search for the optimal split point in the vertices of the existing flow that are neither in the subtree rooted at $x$ or $y$ nor on the undirectional path $\pi^*_{xy}$.




(a) $G_x$.

(b) $G_y$.

(c) Combination of $G_x$ and $G_y$.

**Figure 7:** Negative bicycle reduction algorithm.

If we define the nearest common ancestor of $x$ and $y$ as $c$ as in Figure 6, the bicycle reduction algorithm can be described as follows:

First, compute the incremental cost of redirecting $f_x$ units of flow from $x$ and $f_y$ units of flow from $y$ to all potential split points in the existing flow as follows: for each tree vertex $u$ that is either in the subtree rooted at $x$ or $y$ or on the undirectional path $\pi^*_{xy}$, define the incremental cost $c^*_u$ to be $\infty$. For any other tree vertex $u$, if $u$ is in the subtree rooted at $c$, let $v$ be the nearest ancestor of $u$ on $\pi^*_{xy}$. Let $c^*_{xv}$ be the incremental cost of redirecting $f_x$ units of flow along the undirectional path $\pi^*_{xv}$ from $x$ to $v$, $c^*_{yv}$ be the incremental cost of redirecting $f_y$ units of flow along the undirectional path $\pi^*_{yv}$ from $y$ to $v$, and $c^*_{vu}$ be the incremental cost of redirecting $f_x + f_y$ units of flow along the undirectional path $\pi^*_{vu}$ from $v$ to $u$. The total incremental cost $c^*_u$ for $u$ is defined as $c^*_u = c^*_{xv} + c^*_{yv} + c^*_{vu}$. If $u$ is not in the subtree rooted at $c$, let $c^*_{xc}$ be the incremental cost of redirecting $f_x$ units of flow along the undirectional path $\pi^*_{xc}$ from $x$ to $c$, $c^*_{yc}$ be the incremental cost of redirecting $f_y$ units of flow along the undirectional path $\pi^*_{yc}$ from $y$ to $c$, and $c^*_{cu}$ be the incremental cost of redirecting $f_x + f_y$ units of flow along the undirectional path $\pi^*_{cu}$ from $c$ to $u$. The total incremental cost $c^*_u$ for $u$ is defined as $c^*_u = c^*_{xc} + c^*_{yc} + c^*_{cu}$.

Next, make two copies $G_x = (V, E_x)$ and $G_y = (V, E_y)$ of the network with existing flow $G = (V, E)$. In $G_x$, remove the following edges: all tree edges, all non-tree edges that incident into any tree vertex other than $x$, and all non-tree edges originated from any tree vertex in the subtrees rooted at $x$ or $y$, or from any tree vertex on the undirectional tree path $\pi^*_{xy}$. Compute the incremental cost $c_{uv}$ for an edge $(u, v)$ in $G_x$ as adding $f_x$ amount of flow on $(u, v)$. Similarly remove edges from $G_y$, and compute the incremental cost $c_{uv}$ for an edge $(u, v)$ in $G_y$ as adding $f_y$ amount of flow on $(u, v)$. Then, compute the shortest paths from all vertices in $G_x$ to $x$, and shortest paths from all vertices in $G_y$ to $y$. This can be achieved by running a single destination shortest path algorithm (or a simple modified single source shortest path algorithm) with $x$ or $y$ as the destination vertex. For each vertex $u$ in $G_x$ and $G_y$, record the shortest paths $\pi_{ux}$ in $G_x$ and $\pi_{uy}$ in $G_y$ as well as the shortest distance $c_{ux}$ and $c_{uy}$. Figure 7 shows the transformed graphs from the example networks in Figure 6. In particular, Figure 7(a) shows $G_x$ and Figure 7 (b) shows $G_y$ created from the existing flow.

At last, for any tree vertex $u$ other than $x$ and $y$, define the final total cost $C_u = c_{ux} + c_{uy} + c^*_u$. Find the vertex with the minimum final total cost. This step is illustrated in Figure 7(c). In particular, it is equivalent to first combined $G_x$ and $G_y$, then adding a pseudo source vertex $s'$, and connect $s'$ to any vertex $u$ in the combined graph where $c_{ux}$ and $c_{uy}$ are less than $\infty$. An additional edges from $s'$ to a vertex $u$ has the length equal to $C_u$. Thus, the shortest edge $(s', z)$ corresponds to the optimal split point $z$, and the paths $\pi_{zx}$ and $\pi_{zy}$ are already recorded in $G_x$ and $G_y$. $(s', z)$ and $\pi_{zx}$ and $\pi_{zy}$ are highlighted in Figure 7(c). After the optimal split point $z$ is identified, redirect $f_x$ amount of flow along $\pi_{zx}$ and $\pi^*_{xz}$, and $f_y$ amount of flow along $\pi_{zy}$ and $\pi^*_{yz}$. Compute the total cost of the updated flow. If the updated flow has a lower cost than the original flow, record the updated flow and repeat the above procedure. Otherwise, restore the original flow and stop.

The following pseudo code describes the bicycle reduction operations:

```
Repeat
    For any two leaf or non-branching tree vertices x and y with incoming flows of f_x and f_y, respectively.
        For another tree vertex u,
            If u is on the undirected tree path π*_xy, or u is in the subtree rooted at x or y,
                c*_u ← ∞.
            else
                Find the nearest common ancestor c of x and y.
                If u is in the subtree of c,
                    Let v be the nearest ancestor of u on π*_xy.
                    c*_xv ← incremental cost of redirecting f_x amount of flow along π*_xv.
                    c*_yv ← incremental cost of redirecting f_y amount of flow along π*_yv.
                    c*_vu ← incremental cost of redirecting f_x + f_y amount of flow along π*_vu.
                    c*_u ← c*_xv + c*_yv + c*_vu.
                else
                    c*_xc ← incremental cost of redirecting f_x amount of flow along π*_xc.
                    c*_yc ← incremental cost of redirecting f_y amount of flow along π*_yc.
                    c*_cu ← incremental cost of redirecting f_x + f_y amount of flow along π*_cu.
                    c*_u ← c*_xc + c*_yc + c*_cu.
        Create G_x = (V, E_x) and G_y = (V, E_y) from G.
        Find the shortest paths from all vertices to x in G_x and to y in G_y.
        For a tree vertex u,
            c_ux is the shortest distance from u to x in G_x,
            c_uy is the shortest distance from u to y in G_y.
            C_u ← c_ux + c_uy + c*_u.
        Find the optimal split point z that gives the minimum C_z.
        Redirect f_x units of flow along paths π*_xz and π_zx, and f_y units of flow along paths π*_yz and π_zy,
        If the modified flow x' has a higher cost than the original flow,
            Restore the original flow.
until no flow with lower cost can be obtained.
```

**Complexity Analysis** Let $n$ and $m$ be the numbers of vertices and edges in the network, and $k$ be the number of sink vertices. For each flow, it may need to check $O(k^2)$ vertices pairs before it can determine if a neighboring extreme flow with lower cost exist through flow redirection along a negative cost bicycle. It requires solving $k$ nearest common ancestor problems and computing $4k^2$ incremental path costs. However, the checking procedure is dominated by the shortest path tree computation. If we denote $S(n, m)$ as the time complexity of a single source shortest path algorithm in a graph with $n$ vertices and $m$ edges, then the time complexity of find a neighboring flow with lower cost in the bicycle reduction algorithm is $O(n^2 S(n, m))$, or $O(n^3(m + n^2 \log n))$ if the shortest path algorithm is implemented efficiently.

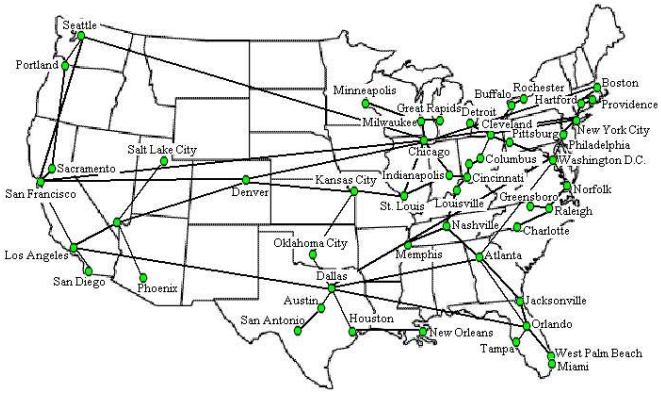



**Figure 8:** National network configuration.

# 6. Experimental Results and Analysis

In this section, we present the simulation studies of the bicycle reduction algorithm in networks with a simple concave edge cost function. We start with the description of the problem instances we generate in our simulations on different network topologies. Next, we explain the two estimated lower bounds we use for performance comparison. The results from our simulation studies are then presented with analysis.

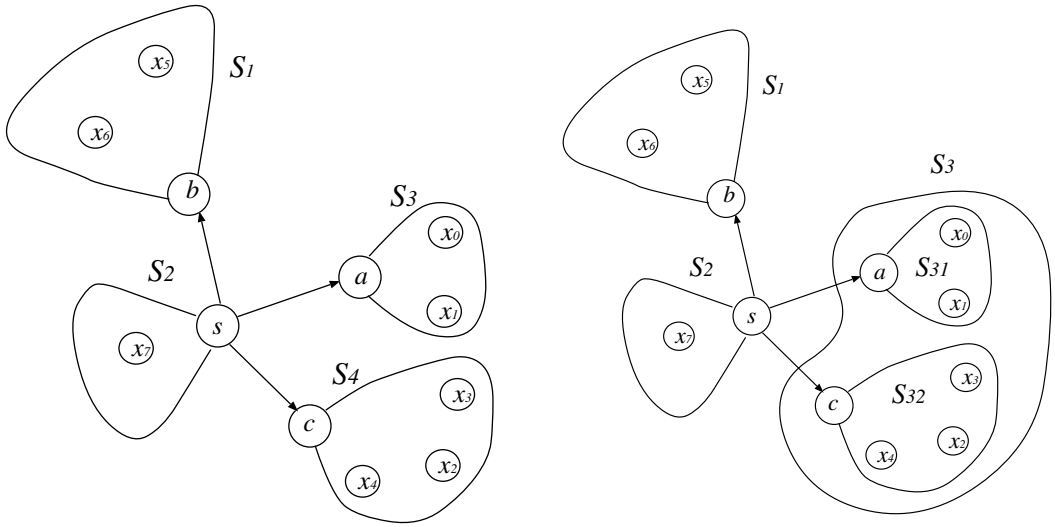
## 6.1. Simulation Setup

We evaluated the bicycle reduction algorithm on a number of network topologies. The first is a $15 \times 15$ torus (each node is connected to four neighbors forming a rectangular grid with "wrap-around edges" linking the top and bottom rows and the leftmost and rightmost columns). We used two types of networks with different link lengths: links in a *random torus* were uniformly distributed, with the longest links being ten times longer than the shortest, while links in a *uniform torus* have a fixed length. Links in random torus are restricted such that triangular inequality is observed. The demands for the sinks were uniformly distributed, all with the same mean demand.

The second network, shown in Figure 8, includes a node at each of the fifty largest metropolitan areas in the United States; the link lengths were chosen to be equal to the geographic distances between the locations, and the demands were chosen to be proportional to the populations of the metropolitan ares [10]. The locations of sources and sinks were selected randomly, with every node having the same probability of selection.

The vertices and edges of the third network are uniform randomly generated inside a unit square. The source node is located in the corner of the unit square as this choice creates extreme flows with deeper trees and greater effect of flow aggregation. The sinks are uniform randomly chosen in the unit square with uniform random sink demands drawn from the range of $[1, 10]$. Such a network would result in extreme flow solutions represented in deep trees, and thus would increase the running time of the local search algorithms. However, it should be noted that such a network has a higher degree of "incidental sharing", and is closer to the optimal solution already.

Besides these three network topologies, we also simulate the bicycle reduction algorithm in networks generated by a topology generator, Inet [11]. Recent studies showed that degree-based topology generators creates networks that have high resemblance of the Internet even though these generators do not consider the network structure specifically [12]. Original Inet is intended for large network with at least 3037 vertices. In our simulation studies, we used a modified version Inet generator so that smaller networks can be generated. The networks we generated for our simulations have 100 vertices, and the fraction of degree one vertices is 0.3. Because it uses a seed for random number




(a) An example partial solution. (b) Optimization with connected components.

**Figure 9:** Lower bound computation.

generator, the number of networks for a fixed number of vertices is at most 64. Thus, each data point is the average of results of 64 independent problem instances, instead of 100 instances as in simulations in other topologies.

We measure the relative costs of flows generated by different algorithms with the estimated bound. We first obtain an initial solution with the *largest demand first (LDF)* algorithm we developed in our previous study of the RDS configuration problem in [1], and apply the original cycle reduction algorithm as well as the bicycle reduction algorithm to find two local optimal solutions from the flow produced by LDF. The number of sinks is varied to show the performance of different algorithms in a variety of network conditions. We also measure the percentage improvements to the flows obtained by LDF after applying the cycle reduction algorithm and the bicycle reduction algorithm.

## 6.2. Lower Bounds Comparison

In our previous study of the RDS configuration problem, we used an easily computed estimated lower bound for the performance evaluation [1]. The idea is to assume all the sinks that are located in a certain geographical area share a single path to the source vertex, and thus achieve maximum possible path sharing among all the sink vertices in that area. In particular, estimated bound EB*(2) is computed by first dividing the sinks into two sets, those to the "left" of the source and those to the "right" of the source. Each of these subsets is then sorted by distance from the source and each node is assumed to share its path to the source with all nodes in the same subset that are at greater distance from the source. EB*(3) (and EB*(4)) is computed similarly, by first dividing the sinks into three (respectively four) sets of nodes defined by "pie-shaped" regions centered on the source, then sorting the subsets by distance from the source and assuming the maximum possible sharing of paths among nodes in the same set. For larger numbers of randomly distributed sinks, it's reasonable to expect EB*(2), EB*(3) and EB*(4) to be no larger than the cost of an optimal solution, although they do not constitute true lower bounds.

A tighter lower bound for a network with a small number of sink nodes can be computed as follows: define a *partial solution* as a subtree rooted at the source along with a partition of sinks among tree nodes. For instance, Figure 9(a) shows a partial solution in which the sink vertices are divided among sets $S_1$, $S_2$, $S_3$ and $S_4$. We can get a lower bound with partial solutions for a network $G = (V, E)$ with $s$ as the source vertex in the following way: let $T$ be a subtree

of $G$ rooted at $s$ with three edges. Partition the sink vertices such that there is a subset of sink vertices $S_i$ for each tree node $t_i$. Compute the total cost of $T$ assuming each tree node has the demand equal to the total demands of the sinks in the subset associated with that tree node. For each tree node $t_i$, compute the lower bound cost for supplying all sink vertices in $S_i$ from $t_i$, assuming all sink vertices in $S_i$ share a single path to $t_i$ and the distance from $t_i$ to a sink vertex $x$ is the same as the shortest path from $t_i$ to $x$. Add all these costs associated with tree nodes to the cost of $T$ to obtain an estimated total cost value. The pseudo code for this is shown below:

```
Create a subtree T rooted at s with three edges.
For each 4-partition (S_1, S_2, S_3, S_4) of sink vertices,
    Associate each tree node t_i with a subset of sink vertices S_i.
    Assign each tree node with a demand equal to the sum of all sink vertices in the associated subset.
    Compute the costs of all tree edges.
    For each sink vertices subset S_i,
        Compute lower bound cost for supplying from t_i, assuming shortest distance and maximum sharing.
    Add all contributions.
```

If we iterate over all such partial solutions with three edges, we get overall lower bound. The trouble with is this is that the best lower bound is likely to involve large set at root; however, if we iterate over all subsets of edges from the root, we don't need to leave any behind at the root.

We can speed up the computation by avoiding some sink vertices partitions that obviously can not produce good lower bounds. This can be done by considering assignment of subsets of sink vertices only to the tree nodes in the same connected component. In this case, we consider the subtree $T$ rooted at the source vertex with all the edges out of the source vertex. For each leaf node $u$ in $T$, if removing $(s, u)$ creates a connected component $C_u$ with some sink vertices, then we only assign the subset of sink vertices in $C_u$ to $u$. If a connected component $C$ can only be created after removing multiple edges in $T$, then we apply the original lower bound operation on the subgraph that includes $C$ and all edges that connect $s$ and $C$. Figure 9(b) shows an example of this optimization. In this example, we only assign subset of sink vertices $\{x_5, x_6\}$ to tree node $b$, and $\{x_7\}$ to $s$. In the subgraph that includes a connected component that is connected to $s$ through $(s, a)$ and $(s, c)$, we compute the lower bound by iterating the partitions of the sink vertices of $\{x_0, \cdots, x_4\}$, but this subgraph can be smaller than the original network, improving the computation time.

Similarly, if we can iterate over all depth two subtrees or "radius $d$" subtrees where tree nodes are within radius $d$ (for each sink must consider smallest radius for its incoming neighbors), we would get tighter lower bounds. We can also apply the similar optimization operations with connected components to improve the performance too.

Figure 10 shows the comparison of the two lower bounds on a randomly generated network with 32 vertices and a variant number of sink vertices. In this plot, the relative cost of a lower bound to the estimated lower bound when we assume all sink vertices share a single path ($EB(1)$). Each of the data point is the average of 100 problem instances. The plot indicates that the tighter lower bounds fall mostly between $EB(2)$ and $EB(3)$, and gradually approach $EB(3)$ as the number of sink vertices increases. This results suggests that $EB(3)$ and $EB(4)$ can serve as sufficient lower bounds for networks with larger sizes.

The performance of the above lower bound algorithms are largely determined by the partition enumeration method employed. Although these algorithms avoid a total enumeration of possible partitions, they are exponential in the worst case. Therefore, we compare the estimated lower bounds and the tighter lower bounds described above only for networks with small number of vertices (less than 30) to show how approximate the estimated lower bounds are. We use the Bit-Vector Representation (BVR) as used in [13] to efficiently enumerate through the partitions for the connected components. In particular, a subset of vertices are represented as a number bits in a word, in which the $i$th bit is set to 1 if vertex $i$ is in the subset and 0 otherwise. Thus, a $k$-partition is represented as $k$ integers that sum to $2^d-$ where $d$ is the number of vertices in the connected component. For more general networks with larger numbers of vertices, we only compare the total cost relative to the estimated lower bound with the assumption that the tighter lower bounds maintain the similar ratio to the estimated lower bounds.

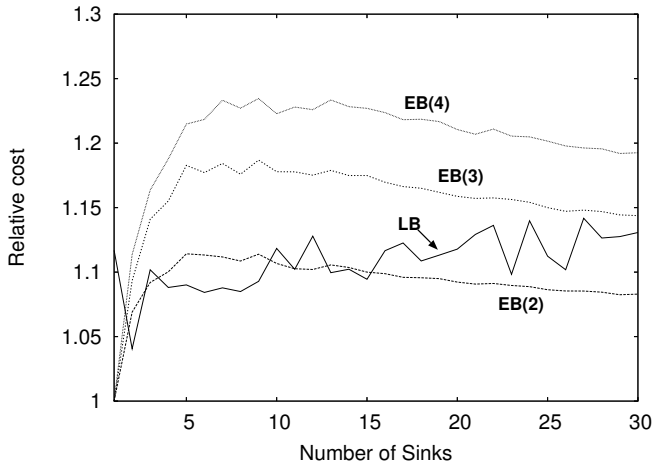
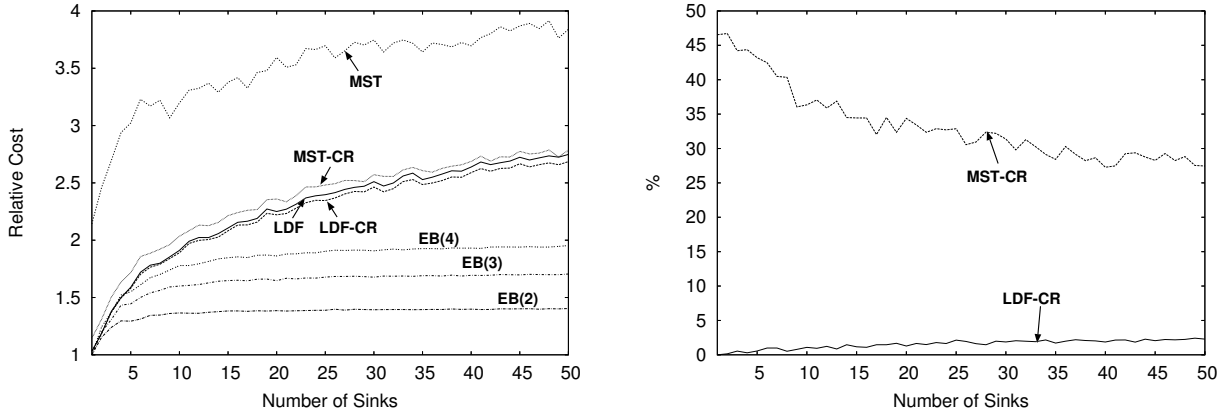



**Figure 10:** Comparison of estimated bounds and lower bounds.



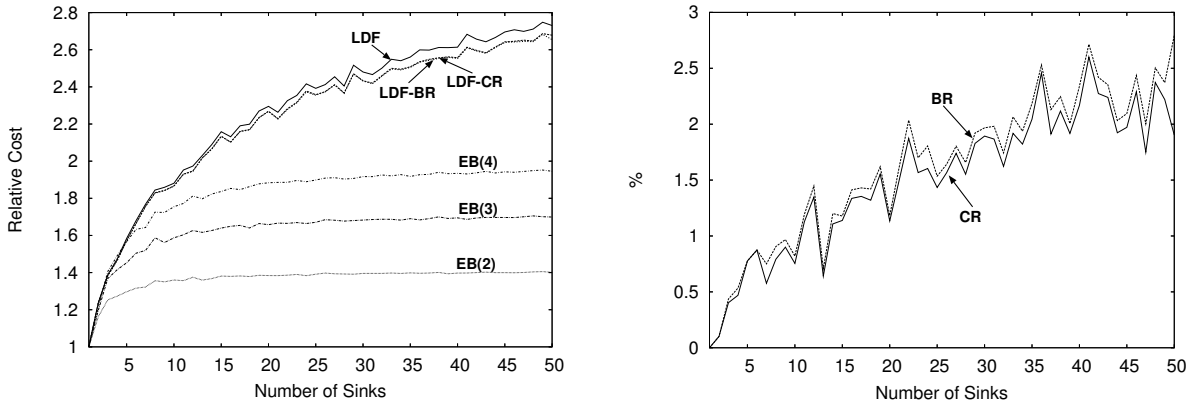

(a) Cost comparison with varied numbers of sinks.

(b) Percentage cost improvement.

**Figure 11:** Cost comparison of cycle reduction algorithm with initial LDF solutions and MST solutions in torus networks.

## 6.3. Simulation Results and Analysis

Figure 11 shows how the initial solution may make a difference. We apply the original cycle reduction algorithm to initial solutions obtained with the largest demand first (LDF) described in [1] and a minimum spanning tree (MST) algorithm in random torus networks, and compare the results. Note that although we use MSTs in our experiments, other simple initial solutions such as random spanning trees have similar results. Figure 11(a) shows the ratio of the cost of the solution produced by different algorithms to the estimated lower bound, as the number of sink vertices increases from 1 to 50. Each data point represents the average of results from 100 independent problem instances. As shown in the charts, for large numbers of sink vertices, the improved solutions obtained from the cycle reduction algorithm are similar: improved solutions from MST solutions are on average 2.75 times of the estimated lower bound, and the improved solutions from the initial LDF solutions are around 2.7 times the estimated lower bound. However, they are both closer to the initial LDF solutions obtained as they are no more than 2.8 times of the estimated lower bound, while the MST initial solutions are up to 3.85 times of the estimated lower bound. Figure 11(b) shows the percentage improvements of the cycle reduction algorithm from the MST and LDF solutions with varied numbers of




(a) Relative cost comparison.

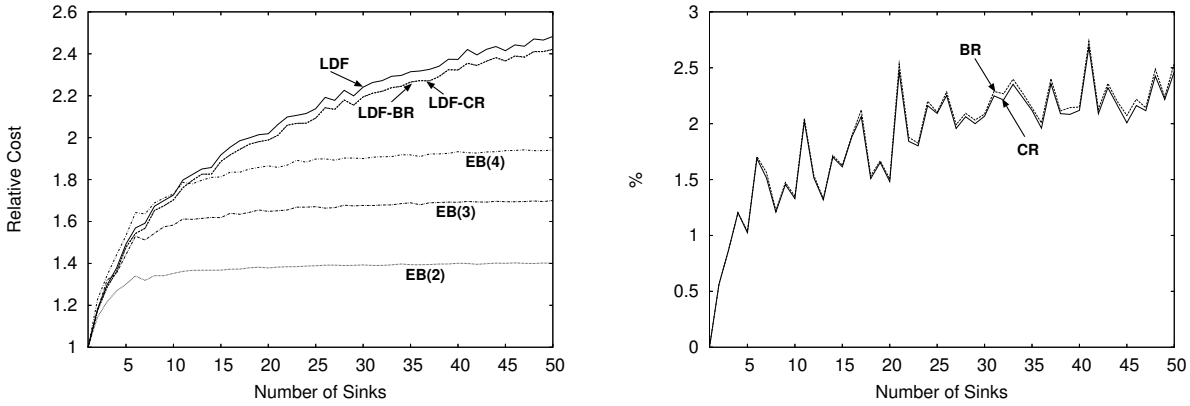
(b) Percentage cost improvement.

**Figure 12:** Cost comparison on torus networks.

sinks. It shows that the space for improvements from a MST solution is much larger than a LDF solution, as the cycle reduction algorithm improves the MST solutions by no less than $26\%$ and even up to $46\%$ in Figure 11(b), while the improvements from LDF solutions are all less than $3\%$. Because it is closer to the local optimal solutions, the initial solutions obtained by LDF has less negative cost cycles than those obtained by MST. Thus, it takes shorter time to reach improved solutions when we use an initial solution obtained from LDF. When we start with an arbitrary tree solution, the results are similar to the MST case. This result indicates that LDF algorithm provides solutions that are reasonablely close to optimal.

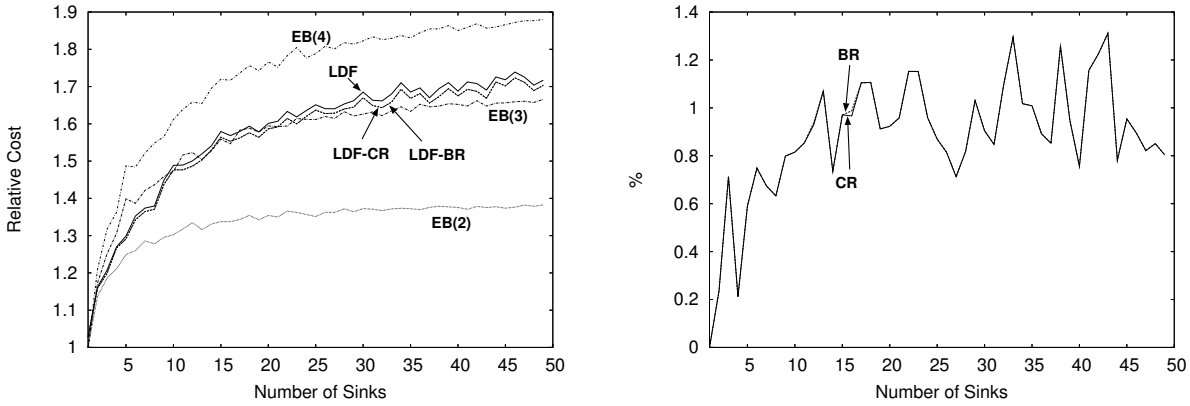
Figure 12 shows the simulation results on random torus networks. The curve labeled with LDF is the relative cost of the initial LDF solution, and the curves labeled LDF-CR and LDF-BR are the results obtained by applying the bicycle reduction algorithm and the original cycle reduction algorithm to the initial solution, respectively. Figure 12(a) shows the ratio of the cost of the solutions produced by different algorithms to the estimated lower bound, as the number of sink vertices increases from 1 to 50. For large numbers of sink vertices, the LDF algorithm produces solutions costing no more than about 2.8 times the estimated lower bound. The cycle reduction algorithm (LDF-CR) improves the solutions from LDF to no more than 2.7 times the estimated lower bound. The bicycle reduction algorithm makes some further improvements to the cycle reduction algorithm, but it is relatively small. This is more clear in the percentage improvements results of both the bicycle reduction and the original cycle reduction algorithms as the number of sink vertices increase in Figure 12(b). It shows that the original cycle reduction algorithm improvement of the LDF solutions grows as the number of sink vertices increase. When there are a large number of sink vertices, the improvement is about $2.5\%$. The bicycle reduction algorithm improves the cycle reduction results by up to $0.5\%$ in some cases, but the average improvement is about $0.1\%$. A similar set of results obtained in uniform torus networks are shown in Figure 13. In these charts, when there are many sink vertices, the cycle reduction and bicycle reduction algorithms improved the average total cost from about 2.5 times of the estimated lower bound to about 2.4 times (Figure 13(a)), or about $2.25\%$ average improvement (Figure 13(b)). The improvement grows as the number of sinks increases from 0 to $2.5\%$. However, the improvement contributed from the bicycle reduction algorithm is noticeablely smaller than in the random torus networks, ranging from 0 to $0.05\%$ with an average improvement of $0.02\%$. This result indicates that negative cost bicycles (or other negative cost multi-cycles) are less likely to exist in torus networks, especially in uniform torus networks. The local optimal flows obtained by the cycle reduction and bicycle reduction algorithm have marginal difference, and they offer only very small improvements to the LDF solutions, while the two local search algorithms have much higher computation complexity. Thus, the LDF algorithm offers solutions very close to local optimal, while more time consuming local search algorithms with cycle and bicycle reductions only provide marginal improvements.

Figure 14 shows the simulation results on the national network topology. The curves in the charts are similarly



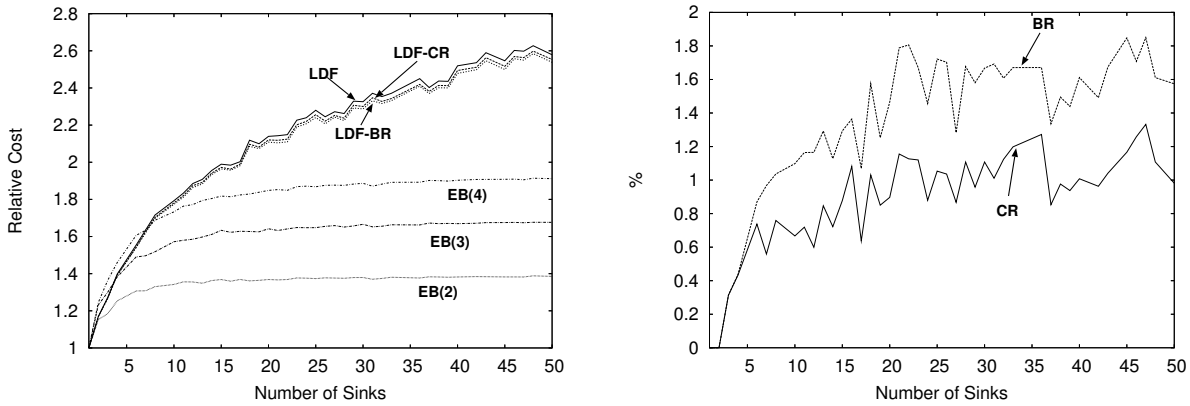

(a) Relative cost comparison. (b) Percentage cost improvement.

**Figure 13:** Cost comparison on uniform torus networks.




(a) Relative cost comparison. (b) Percentage cost improvement.

**Figure 14:** Cost comparison on the national network.




(a) Relative cost comparison.

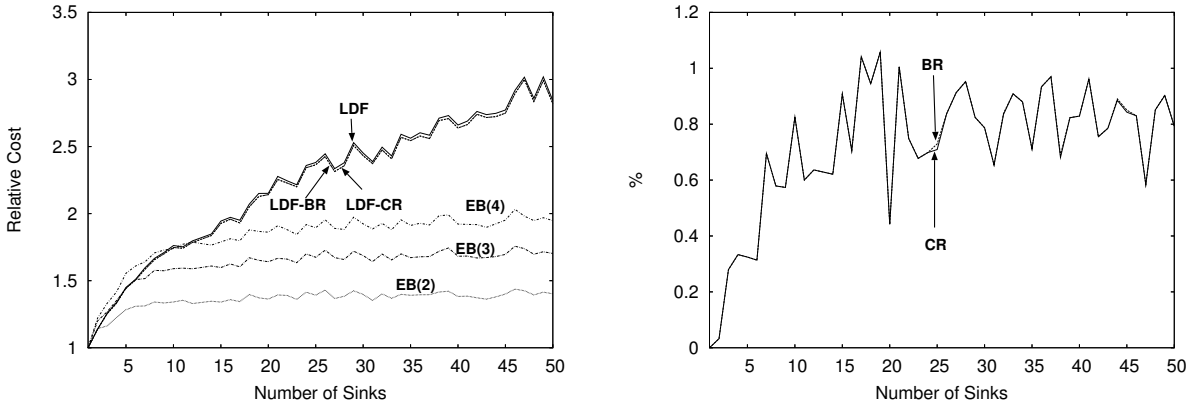
(b) Percentage cost improvement.

**Figure 15:** Cost comparison on random networks.

labeled as the previous charts for torus networks. Figure 14(a) shows the ratio of the cost of the solution produced by different algorithms to the estimated lower bound, as the number of sink vertices increases from 1 to 50. For large numbers of sink vertices, the LDF algorithm produces solutions costing no more than about 1.75 times the estimated lower bound. Both cycle reduction algorithm (LDF-CR) and bicycle reduction algorithm (LDF-BR) improve the LDF solutions, but the bicycle reduction algorithm offers very marginal improvements beyond the cycle reduction solutions, as this is more clearly showed in Figure 14(b), the percentage improvements of the bicycle reduction algorithm over the original algorithm when the number of sink varies. First, it shows that the improvements by the cycle and bicycle reduction algorithms in the national network ($\leq 1.4\%$)are less than in the torus networks ($\leq 2.7\%$). It also shows that the cycle reduction algorithm improves the LDF solutions by an average of $1\%$, and the bicycle reduction algorithm does not offer further improvement in most cases, and very small improvement in small number of cases. These results are largely because the national network is very parse, creating a greater degree of incidental path sharing. In such a sparse network, the LDF algorithm usually is sufficient to create solutions close to optimal, as the more complex local search algorithms can not offer much improvements.

Figure 15 shows the simulation results on randomly generated networks. The relative cost results (Figure 15(a)) show that LDF produces results up to 2.6 times the estimated lower bound when there are 50 sink vertices, while the cycle reduction and bicycle reduction algorithm both improve the quality of the LDF results, and the bicycle reduction algorithm provides more consistent improvement to the solutions produced by the cycle reduction algorithm. As Figure 15(b) shows more clearly that the cycle reduction algorithm improves up to $1.2\%$ over the LDF solutions, and the bicycle reduction algorithm can improve up to $1.85\%$ over the LDF solutions. In addition, the bicycle reduction algorithm offers up to twice the improvement than the cycle reduction algorithm in most cases, the biggest improvement by bicycle reduction algorithm among the topologies simulated. This indicates that there are more negative cost bicycles in the randomly generated and relatively dense networks than the more regular torus networks and sparse national network.

Figure 16 shows the cost comparison on the networks generated by the Inet topology generator [11]. It shows similar improvements of the cycle and bicycle reduction algorithms as the number of sink vertices increases in Figure 16(a). Figure 16(b) shows that both the cycle and bicycle reduction algorithms offer small improvements ($\leq 1.2\%$) over the LDF results, and the bicycle reduction algorithm only improves over the cycle reduction results very marginally in a small number of cases. This is an indication that the small topologies generated by Inet with default parameters are relatively sparse, and thus contain less negative cost cycles and even less bicycles.




(a) Relative cost comparison. (b) Percentage cost improvement.

**Figure 16:** Cost comparison on networks generated by inet topology generator.

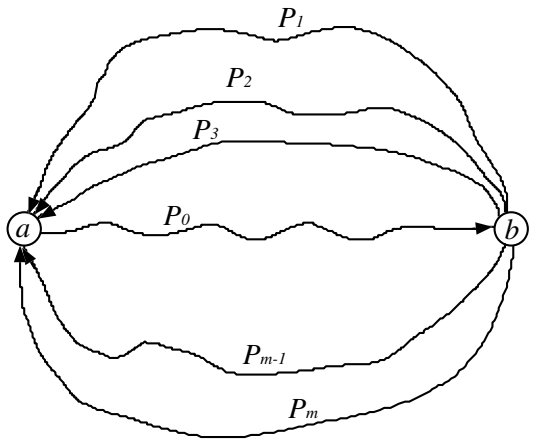



**Figure 17:** Negative cost multi-cycles.

# 7. Negative Cost Multi-cycles

## 7.1. Negative Cost Multi-cycles

Besides negative cost cycles and bicycles, in a network with concave edge costs, there could exist negative multi-cycles that could transform an existing flow to flows with lower costs. We define a *negative cost multi-cycle* as a group of $m$ directed cycles that share a common segment, with the remainder of the cycles edge disjoint. When we add flow along the $m$ cycles, the total cost of the resulting flow is lower than the original flow. We refer to such a multi-cycle as negative cost $m$-cycle. For some special cases, when $m = 2$, it is a negative cost bicycle; when $m = 3$, it is a negative cost tricycle. A general negative cost multi-cycle is illustrated in Fig. 17 with a pair of vertices $a$ and $b$. There is a common path $P_0$ from $a$ to $b$, and $m$ paths from $b$ to $a$. The sum of the cost of all these path is negative. Let $d_i$ be the length of path $P_i$. Then, the incremental cost of adding a unit flow along the multi-cycle could be expressed as $(1+\epsilon)d_0 + \sum_{i=1}^{m} d_i$, where $0 \leq \epsilon \leq m-1$. If $\epsilon = m-1$, the cost of the multi-cycle is equal to the sum of the cost of all $m$ cycles with the usual definition of flow costs. If $\epsilon = 0$, it is only charged once for the shared segment. Any other $0 < \epsilon < 1$ would result in a cost falls in between, showing the benefits of path sharing.

### 7.2. General Multi-cycle Reduction Algorithm

The bicycle reduction algorithm can be further extended to handle negative cost multi-cycles. In particular, we can find the adjacent extreme flows of an existing extreme flow by redirecting flow along a negative cost multi-cycle with $k$ cycles and a common path segment, or negative cost $k$-cycle.

We pick $k$ tree vertices $(v_1, v_2, \cdots, v_k)$, and search for the optimal split points inside the existing flow to redirect flow through $k$ paths out of the existing flow. In particular, for each of the $k$ vertices, we first determine the undirectional paths to to all potential split points in the tree. These potential split points are similarly defined as in the bicycle reduction algorithm, namely, all tree vertices that are neither in the subtree of any one of the $k$ vertices nor on the undirectional paths between any pair of the $k$ vertices.

For each of the potential split point, construct a subtree rooted at that split point connecting all $k$ vertices, and sum up the total incremental cost of redirecting flows originally into the root of the subtree in a similar way to the computation in the bicycle reduction algorithm. Note that in this case, some of the $k$ vertices share some edges on their paths to the root. The flow increment on these edges is the sum of the flow into these subset of vertices in the existing flow. As a result, each potential split point has an associated total incremental cost from the $k$ vertices.

Next, for rach of the $k$ vertices, construct a subgraph from the subgraph induced by the non-tree edges in the existing flow, in which edges into the tree vertices except the ones into the chosen vertex are removed as well as the edges leaving the tree vertices either in the subtrees of the $k$ vertices or on the paths between any pair of vertices in the $k$ vertices. In the constructed subgraph, assign the incremental cost of adding the flow into the chosen vertex as the edge length on an edge. Apply the single destination shortest path algorithm in the subgraph to the chosen vertex, and record the shortest distances and paths from all the potential split points.

After all $k$ vertices are processed, each potential split point has $k$ shortest paths to the $k$ vertices. Pick the potential split point $u$ with the least total distance as the split point. Once the split point is chosen, redirect the flow from the $k$ vertices to $u$ along the paths in the original tree, and also redirect flows to the $k$ vertices along the recorded paths recorded in the constructed subgraphs. If the modified flow has a lower cost than the original flow, repeat the above procedure; otherwise, restore the original flow, and stops with the local optimal solution.

This generalized multi-cycle reduction algorithm describes the cycle reduction algorithm when $k = 1$ and the bicycle reduction algorithm when $k = 2$. Clearly, it has much greater complexity when $k$ grows larger for general multi-cycle reduction. However, as our simulation results indicate, the degree of quality improvements is limited even for bicycle reduction algorithm. As $k$ increases, we would expect more diminishing returns for increased complexity. Therefore, we think bicycle reduction would be sufficient for most practical problems, while the general multi-cycle reduction algorithm has theoretical values but may not be necessary for practical problems.

## 8. Related Work

Gallo and Sodini [5] first proposed two elegant polynomial local search algorithms with cycle reduction of an extreme flow for both uncapacitated and capacitated MCCNFP. They started with an existing extreme flow, and used a simple enumeration operation that is based on the idea of negative cost cycle reduction to examine the adjacent extreme flows. The adjacent extreme flow with the least cost is chosen as the approximation solution. Fontes, Hadjiconstantinou, and Christofides used this cycle reduction method to find an upper bound [14] and develop a branch-and-bound algorithm [13] for MCCNFP.

Another algorithm exclusively for capacitated networks that finds all the cycles that consists of a undirectional path from a vertex $u$ in the existing extreme flow to another vertex $v$ also in the existing extreme flow, and a directed path from $v$ to $u$ outside the existing extreme flow, using a modified depth first search method.

# 9. Conclusions

In this paper, we present a local search algorithm using a multi-cycle reduction heuristic for the general minimum concave cost network flow problem (MCCNFP). The original cycle reduction algorithm proposed by Gallo and Sodini [5] only searches for adjacent extreme flows reachable from an existing flow by redirecting flow along negative cost cycles. We implement a path compression technique to improve the original cycle reduction algorithm such that it only has to compute at most $2m$ shortest path trees, where $m$ is the number of sink nodes, compared with $n$ ($n$ is the total number of vertices) shortest path tree computations in the original algorithm.

In addition, it is shown in this paper that there exists *negative cost multi-cycles* in a concave cost network with an existing flow. By redirecting flow along these multi-cycles, more local optimal extreme flows can be reached. We present a multi-cycle reduction algorithm by identifying the negative cost multi-cycles and redirecting flow along these multi-cycles to get a local optimal extreme flow. Although we focus on the identification of negative cost bicycles, we describe how it can be extended to negative cost multi-cycles. We study the performance of the bicycle reduction algorithm with simulations on different topologies. The experimental results as well as our analysis show that the bicycle reduction can improve the quality of results, but it would reach a point of diminishing return as the quality improvement is limited but the computational complexity grows when we attempt to identify more general negative cost multi-cycles.

# References


[1] R. Qiu and J. S. Turner, "Configuration of reserved delivery subnetworks," in *Proceedings of IEEE Globecom*, (Taipei, Taiwan), Novmber 2002.

[2] G. M. Guisewite and P. M. Pardalos, "Minimum concave-cost network flow problems: Applications, complexity, and algorithms," *Annals of Operations Research*, vol. 25, pp. 75–99, 1990.

[3] G. M. Guisewite and P. M. Pardalos, "Algorithms for the single-source uncapacitated minimum concave-cost network flow problem," *Journal of Global Optimization*, vol. 1, pp. 245–265, 1991.

[4] G. M. Guisewite and P. M. Pardalos, "Global search algorithms for minimum concave-cost network flow problems," *Journal of Global Optimization*, vol. 1, pp. 309–330, 1991.

[5] G. Gallo and C. Sodini, "Adjacent extreme flows and application to min concave cost flow problems," *Networks*, vol. 9, pp. 95–121, 1979.

[6] E. Aarts and J. K. Lentra, *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[7] R. K. Ahuja, T. Magnanti, and J. Orlin, *Network Flows*. Prentice Hall, 1993.

[8] A. V. Goldberg and R. E. Tarjan, "Finding minimum-cost circulations by canceling negative cycles," *Journal of ACM*, vol. 36, pp. 873–886, 1989.

[9] F. Barahona and E. Tardos, "Note on Weintraub's minimum-cost circulation algorithm," *SIAM Journal of Computing*, vol. 18, pp. 579–583, June 1989.

[10] U.S. Census Bereau, "Census 2000." URL http://www.census.gov/population/www/ cen2000/.

[11] J. Winick and S. Jamin, "Inet-3.0: Internet topology generator," Tech. Rep. UM-CSE-TR-456-02, Department of Computer Science and Engineering, University of Michigan, 2002.

[12] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, "Network topology generators: Degree-based vs structural," in *Proceedings of ACM SIGCOMM 2002*, August 2002.

[13] D. B. M. M. Fontes, E. Hadjiconstantinou, and N. Christofides, "A new branch-and-bound algorithm for network design using concave cost flows," tech. rep., Imperial College, London, UK, 2002.

[14] D. B. M. M. Fontes, E. Hadjiconstantinou, and N. Christofides, "Upper bounds for single-source uncapacitated concave minimum-cost network flow problems," *Networks*, vol. 41, pp. 221–228, July 2003.