# An Experimental Study of Minimum Cost Flow Algorithms

Zoltán Király[1], Péter Kovács[2]

[1] Dept. of Computer Science & CNL
Eötvös Loránd University, Budapest, Hungary
`kiraly@cs.elte.hu`

[2] Dept. of Algorithms and Their Applications & CNL
Eötvös Loránd University, Budapest, Hungary
`kpeter@inf.elte.hu`

ICAI 2010 – 8th International Conference on Applied Informatics

Eger, Hungary, January 27-30, 2010

# Overview

## The Minimum Cost Flow Problem

# 1. The Minimum Cost Flow Problem

# The Minimum Cost Flow Problem

The *minimum cost flow* problem is the following:

- Deliver specified amount of flow from a set of supply nodes to a set of demand nodes in a network.
- There are capacity constraints and costs on the arcs.
- The total cost of the transportation has to be minimized.

# The Minimum Cost Flow Problem

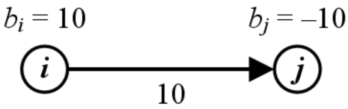
Formal definition:

- Let $G = (V, E)$ be a directed graph.
- We assign for each arc $(i, j) \in E$
  - a lower bound $l_{ij} \geq 0$,
  - an upper bound $u_{ij} \geq l_{ij}$ and
  - a cost $c_{ij}$ (per unit flow).

# The Minimum Cost Flow Problem

Formal definition:

- Let $G = (V, E)$ be a directed graph.
- We assign for each arc $(i, j) \in E$
  - a lower bound $l_{ij} \geq 0$,
  - an upper bound $u_{ij} \geq l_{ij}$ and
  - a cost $c_{ij}$ (per unit flow).
- For each node $i \in V$, we assign a signed supply/demand value $b_i$.



  - If $b_i > 0$, then $i$ is a supply node with $b_i$ supply.
  - If $b_j < 0$, then $j$ is a demand node with $-b_j$ demand.

# The Minimum Cost Flow Problem

Formal definition:

- Let $G = (V, E)$ be a directed graph.
- We assign for each arc $(i, j) \in E$
    - a lower bound $l_{ij} \geq 0$,
    - an upper bound $u_{ij} \geq l_{ij}$ and
    - a cost $c_{ij}$ (per unit flow).
- For each node $i \in V$, we assign a signed supply/demand value $b_i$.
- The goal is to find a feasible flow of minimum total cost.
- The objective function is linear.

R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

# The Minimum Cost Flow Problem

This model can be formulated as an LP problem.

## The Minimum Cost Flow Problem

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{1}$$

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b_i \qquad \forall i \in V \tag{2}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \forall (i,j) \in E \tag{3}$$

# The Minimum Cost Flow Problem

This model can be formulated as an LP problem.

## The Minimum Cost Flow Problem

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{1}$$

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b_i \qquad \forall i \in V \tag{2}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \forall (i,j) \in E \tag{3}$$

*Note.* $\sum_{i\in V} b_i = 0$ is necessary to have a feasible solution.

# The Minimum Cost Flow Problem

This model can be formulated as an LP problem.

### The Minimum Cost Flow Problem

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{1}$$

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b_i \qquad \forall i \in V \tag{2}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \forall (i,j) \in E \tag{3}$$

*Note.* $\sum_{i\in V} b_i = 0$ is necessary to have a feasible solution.

We usually assume that all input data are integer and we are looking for an integer-valued flow (ILP problem).

# The Minimum Cost Flow Problem

Applications:

- This model can be directly applied in various areas:
    - transportation,
    - logistics,
    - telecommunication,
    - network design,
    - resource planning,
    - scheduling
    - etc.
- It also arises as subproblems of more complex optimization models, such as multicommodity flows.

# The Minimum Cost Flow Problem

The main goals of our research:

- Implement several known algorithms as efficiently as possible.
- Study heuristics and implementation details.

# The Minimum Cost Flow Problem

The main goals of our research:

- Implement several known algorithms as efficiently as possible.
- Study heuristics and implementation details.
- Compare different implementations using the same benchmark framework on large problem instances.
- Compare our codes to widely known and used efficient solvers.

# The Minimum Cost Flow Problem

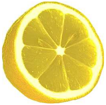
The main goals of our research:

- Implement several known algorithms as efficiently as possible.
- Study heuristics and implementation details.
- Compare different implementations using the same benchmark framework on large problem instances.
- Compare our codes to widely known and used efficient solvers.
- Provide open source implementations as part of the **LEMON** library.

# 2. Implementation and Testing

# LEMON

Our implementations are part of the **LEMON** combinatorial optimization library.



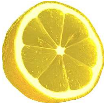
**LEMON** library:

- **L**ibrary for **E**fficient **M**odeling and **O**ptimization in **N**etworks
- It is an open source C++ graph library developed at Eötvös Loránd University, Budapest, Hungary.

http://lemon.cs.elte.hu

# LEMON

Our implementations are part of the **LEMON** combinatorial optimization library.



**LEMON** library:

- **L**ibrary for **E**fficient **M**odeling and **O**ptimization in **N**etworks
- It is an open source C++ graph library developed at Eötvös Loránd University, Budapest, Hungary.
- It contains highly efficient and well cooperating data structures and algorithms that help solving various optimization tasks related to graphs and networks.
- It is similar to BGL (Boost Graph Library) and LEDA.

`http://lemon.cs.elte.hu`

# Test Instances

Generated test instances:

- Several benchmark sets of random networks were generated using NETGEN, GRIDGEN and GOTO.
- The largest instances contain millions of nodes and arcs.

# Test Instances

Generated test instances:

- Several benchmark sets of random networks were generated using NETGEN, GRIDGEN and GOTO.
- The largest instances contain millions of nodes and arcs.
- Costs and capacities are in the range $[1..10000]$ and $[1..1000]$, respectively.
- In NETGEN and GRIDGEN instances, there are $\sqrt{n}$ supply and $\sqrt{n}$ demand nodes with total supply $1000\sqrt{n}$.
- The GOTO problems contain single source and single target nodes.

# Test Instances

Generated test instances:

- We usually obtained similar results for the NETGEN and GRIDGEN networks, thus GRIDGEN tests are omitted here.
- GOTO typically generates much harder problems than the other two generators.

# Test Instances

Generated test instances:

- We usually obtained similar results for the NETGEN and GRIDGEN networks, thus GRIDGEN tests are omitted here.
- GOTO typically generates much harder problems than the other two generators.
- The most important parameter is the density of the graph. The algorithms perform diversely on sparse and dense networks.
- For the sparse graphs, $m \approx 8n$ and for the dense networks, $m \approx \sqrt{n}\,n$.

# Test Instances

Real-world test instances:

- Some real-world problems were also tested.
- They are based on maximum flow instances that arose in medical image processing (`http://vision.csd.uwo.ca/`).
- Random costs are assigned to the arcs and we are looking for a maximum flow of minimum total cost.

# Benchmark System

Benchmark system:

- AMD Opteron Dual Core 2.2 GHz CPU (1 MB cache), 16 GB memory,
- openSUSE 10.1, GCC 4.1.0 compiler, –O3 option.

# 3. Solution Methods

# Algorithms

9 algorithms were implemented applying 4 different approaches.

# Algorithms

9 algorithms were implemented applying 4 different approaches.

1. Cycle Canceling – **primal** methods
    - **SCC**: *Simple Cycle Canceling*
    - **MMCC**: *Minimum Mean Cycle Canceling*
    - **CAT**: *Cancel and Tighten*

# Algorithms

9 algorithms were implemented applying 4 different approaches.

1. Cycle Canceling – **primal** methods
   - **SCC**: *Simple Cycle Canceling*
   - **MMCC**: *Minimum Mean Cycle Canceling*
   - **CAT**: *Cancel and Tighten*
2. Augmenting Path – **dual** methods
   - **SSP**: *Successive Shortest Path*
   - **CAS**: *Capacity Scaling*

# Algorithms

9 algorithms were implemented applying 4 different approaches.

1. Cycle Canceling – **primal** methods
   - **SCC**: *Simple Cycle Canceling*
   - **MMCC**: *Minimum Mean Cycle Canceling*
   - **CAT**: *Cancel and Tighten*
2. Augmenting Path – **dual** methods
   - **SSP**: *Successive Shortest Path*
   - **CAS**: *Capacity Scaling*
3. Cost Scaling – **primal–dual** methods
   - **COS-PR**: *Cost Scaling – Push-Relabel*
   - **COS-AR**: *Cost Scaling – Augment-Relabel*
   - **COS-PAR**: *Cost Scaling – Partial Augment-Relabel*

# Algorithms

9 algorithms were implemented applying 4 different approaches.

1. Cycle Canceling – **primal** methods
    - **SCC**: *Simple Cycle Canceling*
    - **MMCC**: *Minimum Mean Cycle Canceling*
    - **CAT**: *Cancel and Tighten*
2. Augmenting Path – **dual** methods
    - **SSP**: *Successive Shortest Path*
    - **CAS**: *Capacity Scaling*
3. Cost Scaling – **primal–dual** methods
    - **COS-PR**: *Cost Scaling – Push-Relabel*
    - **COS-AR**: *Cost Scaling – Augment-Relabel*
    - **COS-PAR**: *Cost Scaling – Partial Augment-Relabel*
4. Network Simplex method
    - **NS**: Primal Network Simplex

# 3. Solution Methods

## I. Cycle Canceling Algorithms

# Cycle Canceling Method

**Theorem 1.** Negative cycle optimality condition

A feasible solution $x$ of the minimum cost flow problem is optimal if and only if the residual network $G_x$ contains no directed cycle of negative total cost.

*Note.* The cost of a reversed arc is the opposite of the cost of the original arc: $c_{ji} = -c_{ij}$.

# Cycle Canceling Method

**Theorem 1.** Negative cycle optimality condition

A feasible solution $x$ of the minimum cost flow problem is optimal if and only if the residual network $G_x$ contains no directed cycle of negative total cost.

This theorem suggests a simple approach for solving the problem:

1. Find a feasible solution (by solving a maximum flow problem).

# Cycle Canceling Method

**Theorem 1.** Negative cycle optimality condition

A feasible solution $x$ of the minimum cost flow problem is optimal if and only if the residual network $G_x$ contains no directed cycle of negative total cost.

This theorem suggests a simple approach for solving the problem:

1. Find a feasible solution (by solving a maximum flow problem).
2. Iteratively find negative cycles in the residual network and cancel these cycles by augmenting flow along them. (An arc of each cycle is saturated.)

# Cycle Canceling Method

**Theorem 1.** Negative cycle optimality condition

A feasible solution $x$ of the minimum cost flow problem is optimal if and only if the residual network $G_x$ contains no directed cycle of negative total cost.

This theorem suggests a simple approach for solving the problem:

1. Find a feasible solution (by solving a maximum flow problem).
2. Iteratively find negative cycles in the residual network and cancel these cycles by augmenting flow along them. (An arc of each cycle is saturated.)
3. The algorithm terminates when there are no negative cost cycles.

# Cycle Canceling Method

**Theorem 1.** Negative cycle optimality condition

A feasible solution $x$ of the minimum cost flow problem is optimal if and only if the residual network $G_x$ contains no directed cycle of negative total cost.

This theorem suggests a simple approach for solving the problem:

1. Find a feasible solution (by solving a maximum flow problem).
2. Iteratively find negative cycles in the residual network and cancel these cycles by augmenting flow along them. (An arc of each cycle is saturated.)
3. The algorithm terminates when there are no negative cost cycles.

*Primal method*: it maintains a feasible solution and attempts to reduce the objective function value (the total cost of the flow) at every iteration.

# Cycle Canceling Algorithms

Implemented algorithms:

**SCC**: *Simple Cycle Canceling*

**MMCC**: *Minimum Mean Cycle Canceling*

**CAT**: *Cancel and Tighten*

# Cycle Canceling Algorithms

Implemented algorithms:

**SCC**: *Simple Cycle Canceling*

- The Bellman–Ford algorithm is used for finding negative cycles.
- Some practical heuristics were applied to reduce running time.

**MMCC**: *Minimum Mean Cycle Canceling*

**CAT**: *Cancel and Tighten*

# Cycle Canceling Algorithms

Implemented algorithms:

**SCC**: *Simple Cycle Canceling*

**MMCC**: *Minimum Mean Cycle Canceling*

- It cancels a *minimum mean cycle* at each iteration.
- A simple, well-known strongly polynomial algorithm.
- However, it is extremely slow in practice.

**CAT**: *Cancel and Tighten*

# Cycle Canceling Algorithms

Implemented algorithms:

**SCC**: *Simple Cycle Canceling*

**MMCC**: *Minimum Mean Cycle Canceling*

**CAT**: *Cancel and Tighten*

- It is an improved version of MMCC.
- Actually, it applies a *primal–dual* approach: storing node potentials (the dual solution), it finds negative cycles much faster on average.

# Cycle Canceling Algorithms

Implemented algorithms:

**SCC**: *Simple Cycle Canceling*

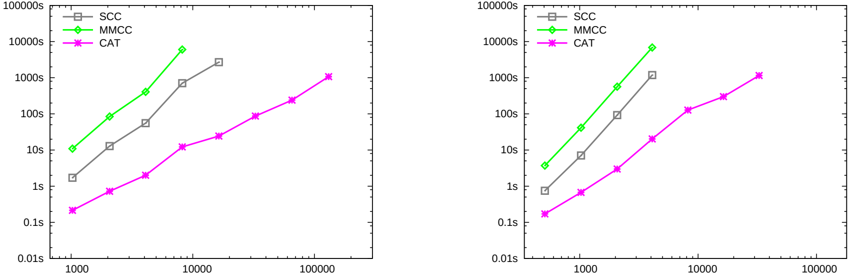
**MMCC**: *Minimum Mean Cycle Canceling*

**CAT**: *Cancel and Tighten*

- It is an improved version of MMCC.
- Actually, it applies a *primal–dual* approach: storing node potentials (the dual solution), it finds negative cycles much faster on average.
- It is also strongly polynomial, but it is much more efficient than the previous two algorithms (both in theory and in practice).

# Cycle Canceling Algorihms

In these charts, the cycle canceling algorithms are compared.
Running times are shown in seconds as a function of the number of nodes (logarithmic scale is used).



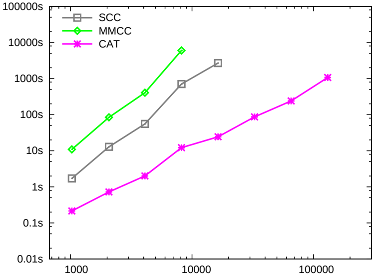
Sparse networks (NETGEN)

Dense networks (NETGEN)

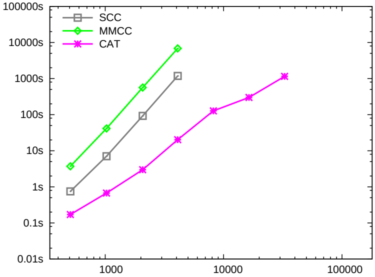
- **SCC**: *Simple Cycle Canceling*
- **MMCC**: *Minimum Mean Cycle Canceling*
- **CAT**: *Cancel and Tighten*

# Cycle Canceling Algorihms

In these charts, the cycle canceling algorithms are compared.
Running times are shown in seconds as a function of the number of nodes (logarithmic scale is used).



Sparse networks (NETGEN)



Dense networks (NETGEN)

- **SCC** is 6-8 times faster than **MMCC**.
- **CAT** is an order of magnitude faster than the others.

# Augmenting Path Method

## 3. Solution Methods

## II. Augmenting Path Algorithms

# Augmenting Path Method

Dual solution method:

- It maintains a dual feasible solution and attempts to reach primal feasibility.

# Augmenting Path Method

Dual solution method:

- It maintains a dual feasible solution and attempts to reach primal feasibility.
- At each iteration, a flow and node potentials are stored.
- The flow is *not necessarily feasible*. The capacity constraints are preserved, but the supply/demand constraints are not.

# Augmenting Path Method

Dual solution method:

- It maintains a dual feasible solution and attempts to reach primal feasibility.
- At each iteration, a flow and node potentials are stored.
- The flow is *not necessarily feasible*. The capacity constraints are preserved, but the supply/demand constraints are not.
- At each step, a certain amount of flow is sent from a node with excess to a node with deficit along a shortest path (with respect to the reduced costs).

# Augmenting Path Method

Dual solution method:

- It maintains a dual feasible solution and attempts to reach primal feasibility.
- At each iteration, a flow and node potentials are stored.
- The flow is *not necessarily feasible*. The capacity constraints are preserved, but the supply/demand constraints are not.
- At each step, a certain amount of flow is sent from a node with excess to a node with deficit along a shortest path (with respect to the reduced costs).
- If there are no nodes with excess, a primal feasible solution is reached.
- It is also optimal, since the dual feasibility is throughout preserved.

# Augmenting Path Method

The dual solution of the minimum cost flow problem is represented by node potentials.

Another optimality condition (equivalent to Theorem 1).

**Theorem 2.** Reduced cost optimality condition

A feasible solution $x$ of the minimum cost flow problem is optimal if and only if for some node potential function $\pi$, the *reduced cost* of each arc in the residual network $G_x$ is non-negative.

# Augmenting Path Method

The dual solution of the minimum cost flow problem is represented by node potentials.

Another optimality condition (equivalent to Theorem 1).

**Theorem 2.** Reduced cost optimality condition

A feasible solution $x$ of the minimum cost flow problem is optimal if and only if for some node potential function $\pi$, the *reduced cost* of each arc in the residual network $G_x$ is non-negative.

**Definition.** Reduced cost

For a given set of node potentials $\pi$, the reduced cost of an arc $(i,j)$ is defined as

$$c_{ij}^{\pi} = c_{ij} + \pi(i) - \pi(j).$$

# Augmenting Path Algorithms

Implemented algorithms:

**SSP**: *Successive Shortest Path*

**CAS**: *Capacity Scaling*

# Augmenting Path Algorithms

Implemented algorithms:

**SSP**: *Successive Shortest Path*

- Simple variant using Dijkstra's algorithm.

**CAS**: *Capacity Scaling*

# Augmenting Path Algorithms

Implemented algorithms:

**SSP**: *Successive Shortest Path*

**CAS**: *Capacity Scaling*

- A faster (polynomial) version of SSP algorithm.
- At each step, we are looking for a shortest path on which at least $\Delta$ amount of flow can be sent.

# Augmenting Path Algorithms

Implemented algorithms:

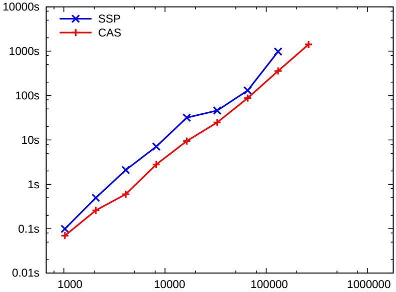
**SSP**: *Successive Shortest Path*

**CAS**: *Capacity Scaling*

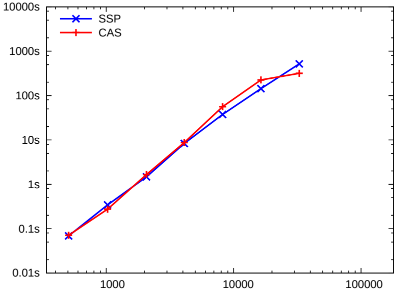
- A faster (polynomial) version of SSP algorithm.
- At each step, we are looking for a shortest path on which at least $\Delta$ amount of flow can be sent.
- If such a path is not found, the value of $\Delta$ is halved and another phase is performed.
- The last phase ($\Delta = 1$) results in a feasible and optimal flow.

# Augmenting Path Algorithms

The augmenting path algorithms are compared in these charts.



Sparse networks (NETGEN)



Dense networks (NETGEN)

- **CAS** usually performs better than **SSP**.
- However, if the capacities or the supply/demand values are rather small, then **SSP** is clearly the fastest solution method. (Only a few calls of Dijkstra's algorithm are required.)

# Cost Scaling Method

## 3. Solution Methods

## III. Cost Scaling Algorithms

# Cost Scaling Method

Cost scaling method:

- It applies a *primal–dual* approach.
- It can be viewed as a generalization of the *preflow push-relabel* algorithm for the maximum flow problem.

# Cost Scaling Method

Cost scaling method:

- It applies a *primal–dual* approach.
- It can be viewed as a generalization of the *preflow push-relabel* algorithm for the maximum flow problem.
- At each phase, an $\epsilon$-optimal primal-dual solution pair is computed.
- It means that for each arc $(i, j)$ in the residual network, $c_{ij}^{\pi} \geq -\epsilon$ holds.

# Cost Scaling Method

Cost scaling method:

- It applies a *primal–dual* approach.
- It can be viewed as a generalization of the *preflow push-relabel* algorithm for the maximum flow problem.
- At each phase, an $\epsilon$-optimal primal-dual solution pair is computed.
- It means that for each arc $(i, j)$ in the residual network, $c_{ij}^{\pi} \geq -\epsilon$ holds.
- After that, $\epsilon$ is halved and another phase is performed.
- If $\epsilon < 1/n$, then optimal primal–dual solutions are found.

# Cost Scaling Method

Cost scaling method:

- It applies a *primal–dual* approach.
- It can be viewed as a generalization of the *preflow push-relabel* algorithm for the maximum flow problem.
- At each phase, an $\epsilon$-optimal primal-dual solution pair is computed.
- It means that for each arc $(i, j)$ in the residual network, $c_{ij}^{\pi} \geq -\epsilon$ holds.
- After that, $\epsilon$ is halved and another phase is performed.
- If $\epsilon < 1/n$, then optimal primal–dual solutions are found.
- In the scaling phases, *push* and *relabel* operations are used.

# Cost Scaling Method

Implemented algorithms:

**COS-PR**: *Push-Relabel*

**COS-AR**: *Augment-Relabel*

**COS-PAR**: *Partial Augment-Relabel*

# Cost Scaling Method

Implemented algorithms:

**COS-PR**: *Push-Relabel*

- The original variant using local push and relabel operations.

**COS-AR**: *Augment-Relabel*

**COS-PAR**: *Partial Augment-Relabel*

# Cost Scaling Method

Implemented algorithms:

**COS-PR**: *Push-Relabel*

**COS-AR**: *Augment-Relabel*

- Instead of the push operations, augmenting paths are found from excess nodes to deficit nodes.
- A path augmentation is equal to several consecutive push operations.

**COS-PAR**: *Partial Augment-Relabel*

# Cost Scaling Method

Implemented algorithms:

**COS-PR**: *Push-Relabel*

**COS-AR**: *Augment-Relabel*

**COS-PAR**: *Partial Augment-Relabel*

- Goldberg's new idea is applied to this problem: the length of an augmenting path is limited.

Andrew V. Goldberg. *The partial augment-relabel algorithm for the maximum flow problem.* ESA 2008, 466–477, 2008.

# Cost Scaling Method

Implemented algorithms:

**COS-PR**: *Push-Relabel*

**COS-AR**: *Augment-Relabel*

**COS-PAR**: *Partial Augment-Relabel*

- Goldberg's new idea is applied to this problem: the length of an augmenting path is limited.
- At once, flow is sent on a path consisting of at most $k = 4$ arcs.
- It proved to be a good compromise between the above two methods. It is significantly faster in practice.

Andrew V. Goldberg. *The partial augment-relabel algorithm for the maximum flow problem.* ESA 2008, 466–477, 2008.

# Cost Scaling Heuristics

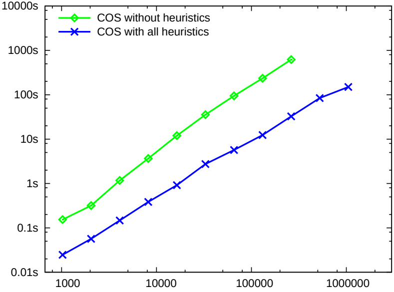
The performance of the Cost Scaling algorithm highly depends on the applied heuristics.

In our implementations, the following heuristics are used:

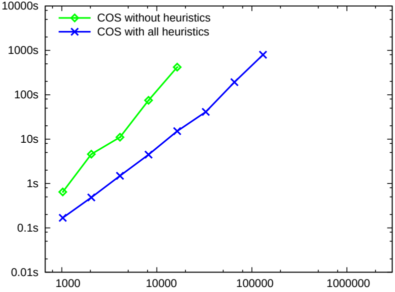
- price refinement,
- early termination,
- global update,
- push-look-ahead (only in the push-relabel version).

# Cost Scaling Heuristics

The performance of the Cost Scaling algorithm highly depends on the applied heuristics.



Sparse networks (NETGEN)

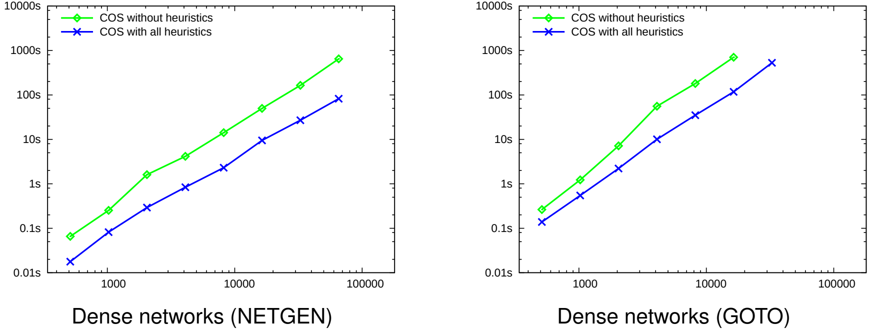


Sparse networks (GOTO)

**COS** with all heuristics was faster than **COS** without heuristics by a factor between 6 and 30 on these problem instances.

# Cost Scaling Heuristics

The performance of the Cost Scaling algorithm highly depends on the applied heuristics.



Dense networks (NETGEN)

Dense networks (GOTO)

**COS** with all heuristics was faster than **COS** without heuristics by a factor between 6 and 30 on these problem instances.

# 3. Solution Methods

## IV. Network Simplex Algorithm

# Network Simplex Method

Primal network simplex algorithm:

- It is the specialized version of the LP simplex method directly for the minimum cost flow problem.

# Network Simplex Method

Primal network simplex algorithm:

- It is the specialized version of the LP simplex method directly for the minimum cost flow problem.
- The LP variables correspond to the arcs of the graph.
- The LP bases are represented by *spanning tree solutions*.
- Such a solution is given by a spanning tree of the network for which the flow values are fixed on all arcs outside the tree (i.e. they have a flow value either at the lower bound or at the upper bound).

# Network Simplex Method

Primal network simplex algorithm:

- It is the specialized version of the LP simplex method directly for the minimum cost flow problem.
- The LP variables correspond to the arcs of the graph.
- The LP bases are represented by *spanning tree solutions*.
- Such a solution is given by a spanning tree of the network for which the flow values are fixed on all arcs outside the tree (i.e. they have a flow value either at the lower bound or at the upper bound).
- The algorithm maintains a spanning tree with flow values (primal solution) and node potentials (dual solutions).
- At each iteration, we attempt to reduce the objective function value (the total cost of the flow) by moving from one spanning tree solution to another.

# Network Simplex Method

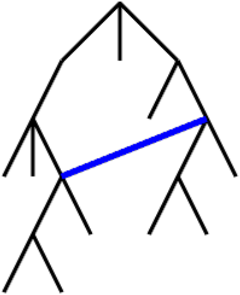
Primal network simplex algorithm:

- At each step, a non-tree arc violating the optimality condition is selected.

# Network Simplex Method

Primal network simplex algorithm:

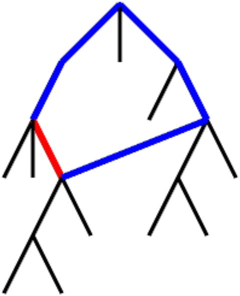
- At each step, a non-tree arc violating the optimality condition is selected.
- This arc is added to the spanning tree (a variable is added to the base). By this operation, a cycle of negative total cost is determined.



# Network Simplex Method

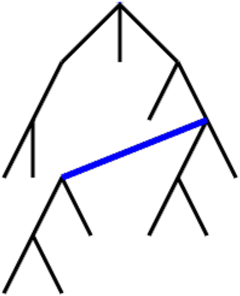
Primal network simplex algorithm:

- At each step, a non-tree arc violating the optimality condition is selected.
- This arc is added to the spanning tree (a variable is added to the base). By this operation, a cycle of negative total cost is determined.
- This cycle is canceled by augmenting flow along it and one of the saturated (fixed) arcs is removed from the tree. This operation is called *pivot*.



# Network Simplex Method

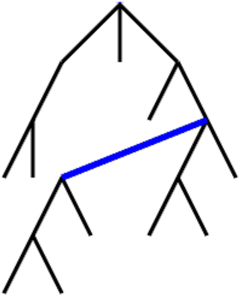
Primal network simplex algorithm:



- At each step, a non-tree arc violating the optimality condition is selected.
- This arc is added to the spanning tree (a variable is added to the base). By this operation, a cycle of negative total cost is determined.
- This cycle is canceled by augmenting flow along it and one of the saturated (fixed) arcs is removed from the tree. This operation is called *pivot*.
- If no suitable incomming arc can be selected, then the flow is optimal.

# Network Simplex Method

Primal network simplex algorithm:



- At each step, a non-tree arc violating the optimality condition is selected.
- This arc is added to the spanning tree (a variable is added to the base). By this operation, a cycle of negative total cost is determined.
- This cycle is canceled by augmenting flow along it and one of the saturated (fixed) arcs is removed from the tree. This operation is called *pivot*.
- If no suitable incomming arc can be selected, then the flow is optimal.

Actually, this algorithm is a particular variant of the basic primal approach (cycle canceling). Due to the sophisticated method of maintaining spanning tree solutions, a negative cycle can be found much faster (in $O(m)$ time).

# Network Simplex Method

Implementation:

- A complex data structure is required to store and update spanning trees efficiently.
- Several different methods are known for this, e. g. ATI, API, XTI, XPI. One of the most efficient schemes, the XTI method was implemented.

# Network Simplex Method

Implementation:

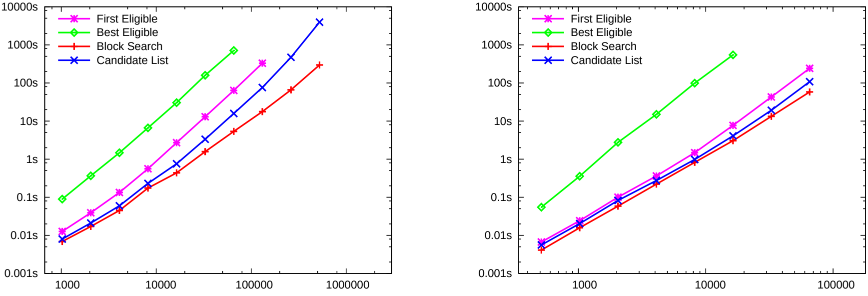
- A complex data structure is required to store and update spanning trees efficiently.
- Several different methods are known for this, e. g. ATI, API, XTI, XPI. One of the most efficient schemes, the XTI method was implemented.
- Apart from maintaining the spanning tree solutions, the most critical operation is the selection of the entering arc (the pivot rule).
- It should be fast, but it should find an arc having reduced cost as small as possible. These requirements are clearly contrary.

# Network Simplex Method

Implementation:

- A complex data structure is required to store and update spanning trees efficiently.
- Several different methods are known for this, e. g. ATI, API, XTI, XPI. One of the most efficient schemes, the XTI method was implemented.
- Apart from maintaining the spanning tree solutions, the most critical operation is the selection of the entering arc (the pivot rule).
- It should be fast, but it should find an arc having reduced cost as small as possible. These requirements are clearly contrary.
- Various pivot rules were implemented applying different approaches. They highly affect the overall running time of the algorithm.

# Network Simplex Pivot Rules



Sparse networks (NETGEN)

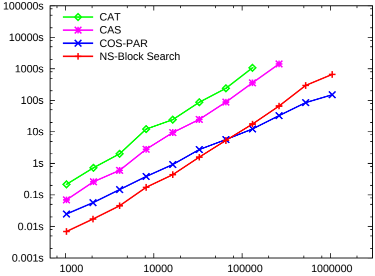
Dense networks (NETGEN)

- **First Eligible** is relatively efficient although it is very simple.
- **Best Eligible** method is by far the slowest one.
- **Block Search** proved to be the most efficient and most robust.
- **Candidate List** is also very efficient.

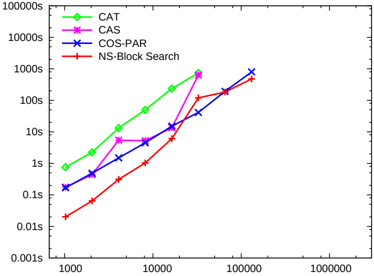
# 4. Experimental Results

# Comparison

These charts compare our fastest implementations of the four approaches.



Sparse networks (NETGEN)

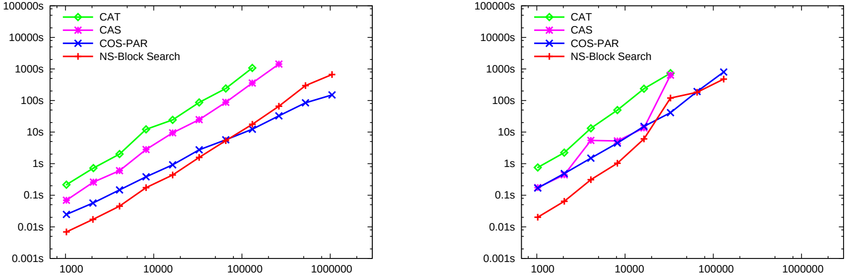


Sparse networks (GOTO)

- Cancel and Tighten (**CAT**) is the slowest among these four implentations.
- Capacity Scaling (**CAS**) is significantly faster.
- The most efficient methods are clearly the Cost Scaling (**COS**) and Network Simplex (**NS**) algorithms.

# Comparison

These charts compare our fastest implementations of the four approaches.

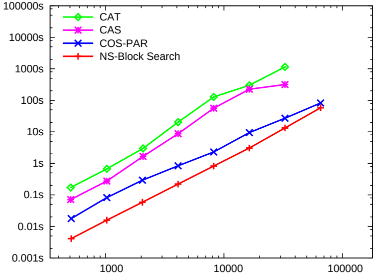


Sparse networks (NETGEN)　　Sparse networks (GOTO)

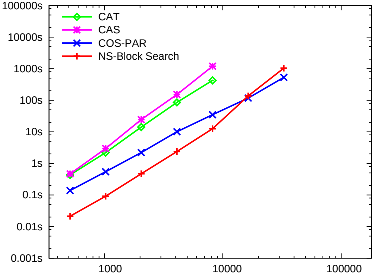
- **COS** proved to be *asymptotically* faster than all other methods both on sparse and dense networks.
- Therefore, **COS** is the absolute winner on huge networks, especially when they are relatively sparse.
- However, on small and medium sized graphs, **NS** is typically much faster.

# Comparison

These charts compare our fastest implementations of the four approaches.



Dense networks (NETGEN)



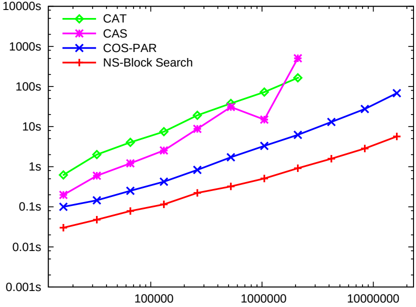
Dense networks (GOTO)

- **COS** proved to be *asymptotically* faster than all other methods both on sparse and dense networks.
- Therefore, **COS** is the absolute winner on huge networks, especially when they are relatively sparse.
- However, on small and medium sized graphs, **NS** is typically much faster.

# Comparison

In this chart, the number of nodes is fixed (to 4096) and the running times are shown as a function of the number of arcs.

The largest instance is the full graph containing 16 million arcs.

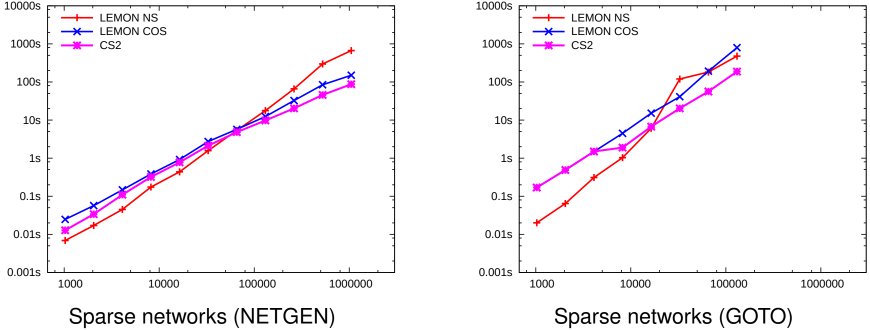


Network Simplex (**NS**) is by far the most efficient in such tests.

# Comparison with Other Solvers

On the following slides, our two fastest implementations, the cost scaling (**COS**) and the network simplex (**NS**) algorithms are compared to widely known efficient solvers.

# Comparison with Other Solvers

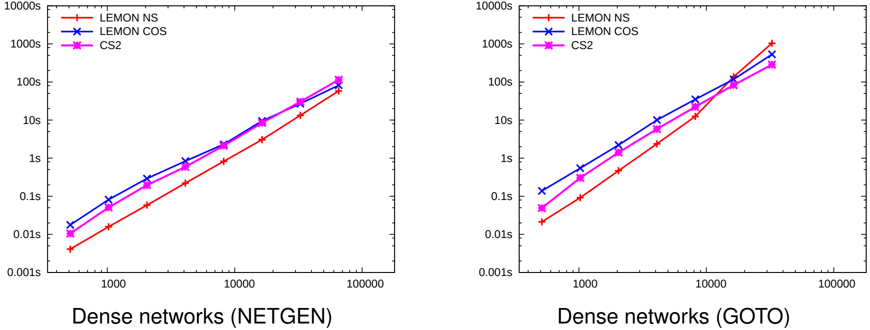


Sparse networks (NETGEN)

Sparse networks (GOTO)

- **CS2**: CS2 4.6 (latest version) by A. V. Goldberg (IG Systems).
- It is an efficient implementation of the cost scaling method.
- It proved to be slightly faster than our cost scaling implementation (**COS**).

# Comparison with Other Solvers

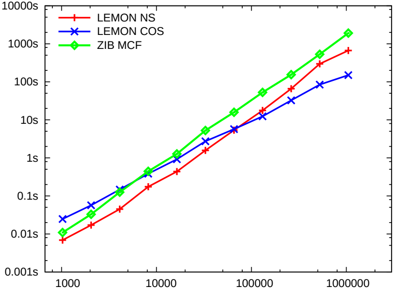


Dense networks (NETGEN)

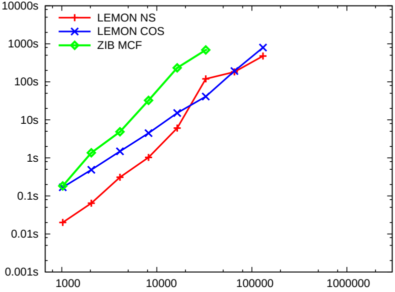
Dense networks (GOTO)

- **CS2**: CS2 4.6 (latest version) by A. V. Goldberg (IG Systems).
- It is an efficient implementation of the cost scaling method.
- It proved to be slightly faster than our cost scaling implementation (**COS**).

# Comparison with Other Solvers



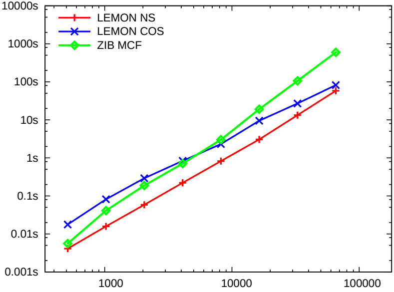
Sparse networks (NETGEN)



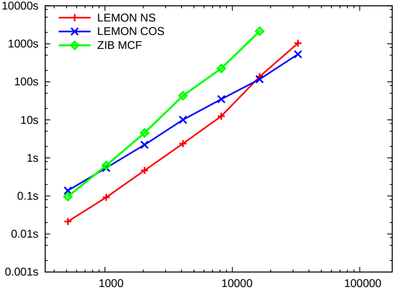
Sparse networks (GOTO)

- **ZIB MCF**: MCF 1.3 (latest version) by A. Lbel (Zuse Institute Berlin).
- It is a network simplex implementation.
- Our **NS** code was much faster on all problem sets.

# Comparison with Other Solvers



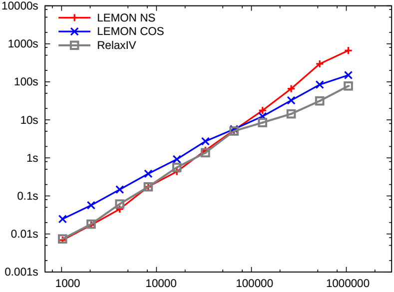
Dense networks (NETGEN)



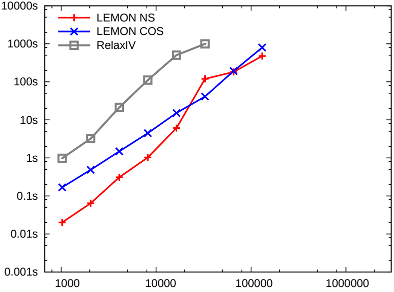
Dense networks (GOTO)

- **ZIB MCF**: MCF 1.3 (latest version) by A. Lbel (Zuse Institute Berlin).
- It is a network simplex implementation.
- Our **NS** code was much faster on all problem sets.

# Comparison with Other Solvers



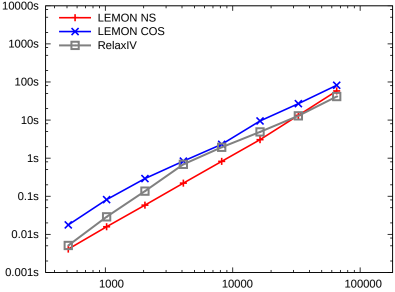
Sparse networks (NETGEN)



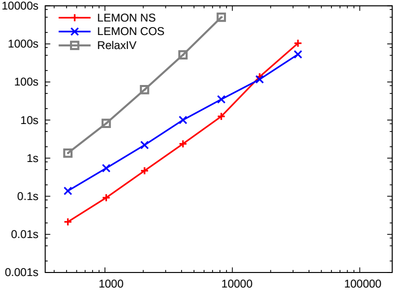
Sparse networks (GOTO)

- **RelaxIV**: an efficient implementation of the relaxation algorithm by D. P. Bertsekas and P. Tseng.
- It proved to be remarkably efficient on NETGEN problems, but it performed extremely poorly on GOTO instances.

# Comparison with Other Solvers



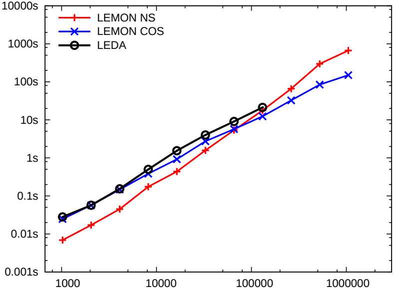
Dense networks (NETGEN)



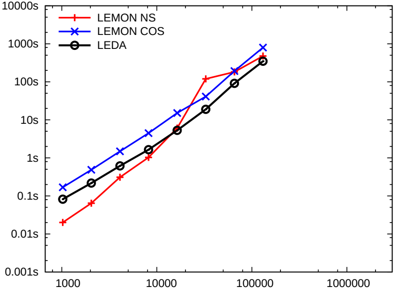
Dense networks (GOTO)

- **RelaxIV**: an efficient implementation of the relaxation algorithm by D. P. Bertsekas and P. Tseng.
- It proved to be remarkably efficient on NETGEN problems, but it performed extremely poorly on GOTO instances.

# Comparison with Other Solvers



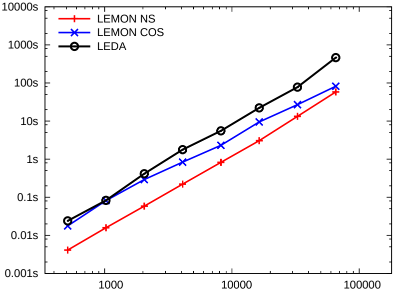
Sparse networks (NETGEN)



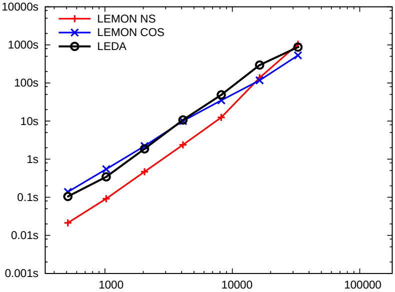
Sparse networks (GOTO)

- **LEDA**: The minimum cost flow method of the LEDA 5.0 C++ optimization library (which is a commercial software).
- LEDA provides an efficient cost scaling implementation.
- Our **COS** code often outperformed it, especially on NETGEN networks.
- Moreover, LEDA failed on the largest instances with *cost overflow* error.

# Comparison with Other Solvers



Dense networks (NETGEN)

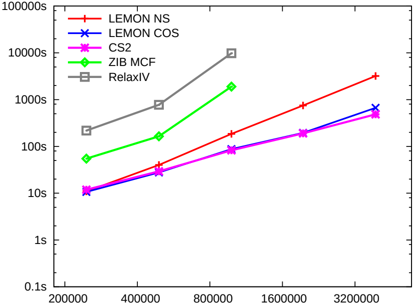


Dense networks (GOTO)

- **LEDA**: The minimum cost flow method of the LEDA 5.0 C++ optimization library (which is a commercial software).
- LEDA provides an efficient cost scaling implementation.
- Our **COS** code often outperformed it, especially on NETGEN networks.
- Moreover, LEDA failed on the largest instances with *cost overflow* error.

# Comparison on Real-World Networks

This chart show the running times on the real-world networks that arose in segmentation problems in medical image processing.



- Our implementations (**NS** and **COS**) solved these problems efficiently.
- **CS2** was even slightly faster than our **COS** algorithm.
- **ZIB MCF** and **RelaxIV** were not competitive in these tests.

# 5. Summary

# Summary

- 9 algorithms were implemented with various heuristics.
- They were compared systematically on large scale generated and real-world problem instances.

# Summary

- 9 algorithms were implemented with various heuristics.
- They were compared systematically on large scale generated and real-world problem instances.
- Our implementations proved to be rather efficient and competitive or superior to highly regarded public solvers.
- This is a remarkable achievement considering that this problem has been a subject of high theoretical and practical interest for decades.

# Summary

- 9 algorithms were implemented with various heuristics.
- They were compared systematically on large scale generated and real-world problem instances.
- Our implementations proved to be rather efficient and competitive or superior to highly regarded public solvers.
- This is a remarkable achievement considering that this problem has been a subject of high theoretical and practical interest for decades.
- Cost scaling algorithms proved to be more efficient than network simplex and relaxation methods on large and relatively sparse networks.
- On small and medium sized graphs and on rather dense graphs, network simplex methods are typically faster.

# Summary

- 9 algorithms were implemented with various heuristics.
- They were compared systematically on large scale generated and real-world problem instances.
- Our implementations proved to be rather efficient and competitive or superior to highly regarded public solvers.
- This is a remarkable achievement considering that this problem has been a subject of high theoretical and practical interest for decades.
- Cost scaling algorithms proved to be more efficient than network simplex and relaxation methods on large and relatively sparse networks.
- On small and medium sized graphs and on rather dense graphs, network simplex methods are typically faster.
- Our implementations are available as part of the LEMON open source C++ optimization library.

`http://lemon.cs.elte.hu`