Efficient Algorithms for Graph Optimization Problems

Ph.D. Thesis

Péter Kovács

Supervisor: Zoltán Király, Ph.D.

Eötvös Loránd University
Faculty of Informatics

Ph.D. School of Computer Science
Director: Erzsébet Csuhaj-Varjú, D.Sc.

Ph.D. Program of Foundations and Methodology of Informatics
Director: Zoltán Horváth, Ph.D.

Budapest, 2019

Acknowledgment

I am very grateful to my supervisor, Zoltán Király, for his deep knowledge, agility, and enthusiasm, which have greatly influenced me. Without his assistance and encouragement, this thesis could not have been written. Additionally, I would like to express my sincere appreciation to him for setting an excellent example of a lecturer and researcher.

I am also grateful to Alpár Jüttner for the opportunity to participate in the LEMON project as well as for his reliable guidance with regard to every detail. I have learned a lot from him over the years. Furthermore, I am thankful to Balázs Dezső for his help and many useful suggestions related to the development of the LEMON library.

I would like to express my gratitude and appreciation to András Frank for inviting me to the EGRES research group and for his outstanding knowledge and inspiring personality. In addition, I would like to thank all members of the EGRES group for working together.

I am also thankful to the professors and lecturers of the Faculty of Informatics and the Institute of Mathematics at ELTE. I am especially grateful to István Fekete for his valuable assistance and friendship.

The financial support of the Communication Networks Laboratory (CNL) is highly appreciated, as well as the opportunity to participate in its regular workshops.

Furthermore, I would like to express my thanks to Péter Englert for the effective and inspiring joint work at ChemAxon, which remains a memorable experience for me.

My special thanks are due to Miklós Vargyas for being a great mentor during my first years at ChemAxon. I was impressed and influenced by his generosity and exceptional ability to pay attention and listen to others. I also thank Ferenc Csizmadia and my co-workers at ChemAxon for the motivating and supporting atmosphere.

On a personal note, I would like to express my gratitude to my family. I cannot be thankful enough to my parents for their love, devoted support, and so many things I have learned from them. Finally, I am deeply grateful to my wife, Dia, and our children, Viktor, Dávid, and Ádám, for their unconditional love, understanding, and making my life complete. I dedicate this work to them.

Contents

Chapter 1

Introduction

Over the last several decades, graphs and related algorithms have been intensively studied and applied to solve countless real-world problems. Due to their expressiveness and strong theoretical background, graphs are essential modeling tools, which are utilized in diverse fields, such as describing transportation, communication, electrical, and social networks as well as modeling various real-life structures like molecules or abstract relations between objects of any kind. Therefore, significant efforts have been devoted to the development and analysis of efficient algorithms for solving combinatorial optimization problems related to graphs.

This thesis discusses fundamental graph optimization problems and efficient solution methods. The primary focus is on the *minimum-cost flow* and the *maximum common subgraph* problems, but we also consider the tasks of finding *minimum-mean cycles* and *maximum cliques*, which arise as subproblems in the algorithms. All of these are classic optimization problems having extensive literature and numerous applications.

The main contribution of this work is the highly efficient implementation of different algorithms for solving these problems, including the development of novel heuristic improvements and the thorough elaboration of important details. Extensive empirical studies were also carried out, which is essential because the practical performance of complex optimization algorithms is usually quite different from what is suggested by their theoretical running time. These experiments have verified that the most efficient algorithms of the author are usually faster or yield better results than other available implementations, including both open-source and commercial software. Some parts of this research are joint works with Zoltán Király, Péter Englert, Alpár Jüttner, and Balázs Dezső.

The achieved results were published in four journal articles [1, 2, 3, 4] and two conference papers [5, 6]. According to Google Scholar [a], these publications have more than 300 independent citations in total, among which more than 220 citations are also verified in the database of MTMT (Magyar Tudományos Művek Tára) [b].

The rest of the thesis is organized as follows. This chapter provides a brief introduction and motivation together with simple examples. Chapter 2 discusses the necessary definitions and preliminary results. In Chapter 3, the achievements related to the minimum-cost flow problem are presented. Chapter 4 discusses the contribution of this work related to the maximum common subgraph problem. Chapter 5 introduces the C++ optimization

[a] https://scholar.google.com/citations?user=7yee1R0AAAAJ

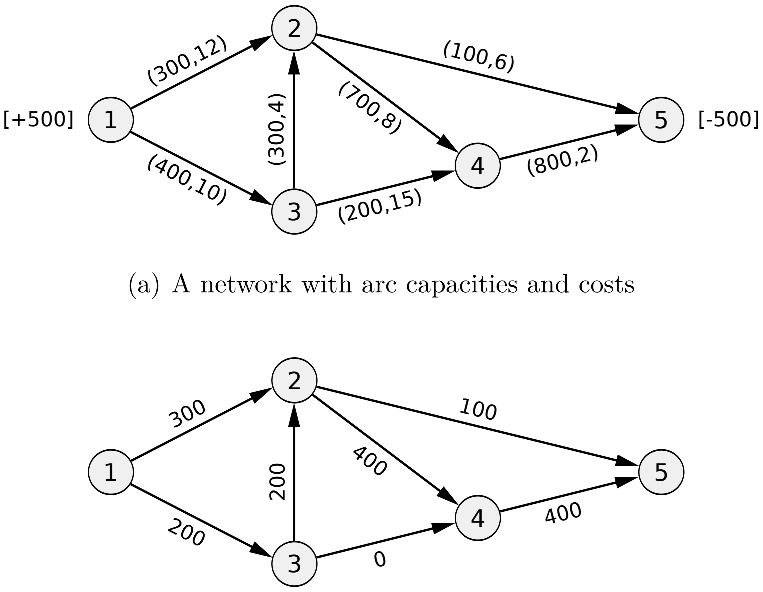
[b] https://m2.mtmt.hu/gui2/?type=authors&mode=browse&sel=10033546

library called LEMON, which involves the implementations of the presented minimum-cost flow algorithms. Finally, Chapter 6 summarizes the results.

1.1 Minimum-cost flows

Network flow theory comprises a wide variety of combinatorial optimization models, which have essential applications in numerous fields, including transportation, communication, engineering, network design, chemistry, and many other industries. In these applications, some entity (commodity, message, electricity, vehicle, person, etc.) is to be transported from one point to another in a network with respect to the given constraints.

In particular, the *minimum-cost flow* problem is one of the most widely used network flow models, in which a specified amount of entity is to be delivered with minimum total cost from supply nodes to demand nodes in a network with arc capacities and costs. Figure 1.1 depicts a simple instance of the problem and the corresponding optimal solution.



(a) A network with arc capacities and costs

(b) Optimal transportation of 500 units of flow from node 1 to node 5

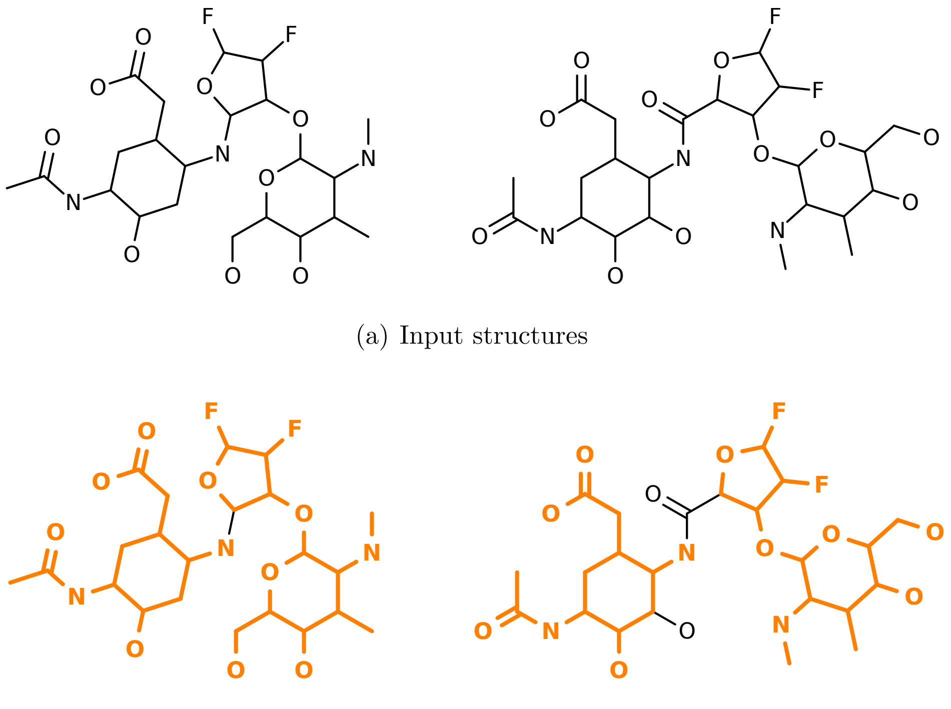
Figure 1.1: Simple instance of the minimum-cost flow problem. The first image displays the network. The label (u, c) of an arc denotes its capacity u and cost c, while the label $[b]$ of a node denotes its signed supply value (unless it is zero). The second image depicts the unique optimal solution in terms of flow values assigned to the arcs.

Chapter 3 considers this problem and presents the related work of the author. First, the deep mathematical background of the model and related results are outlined. Then, we discuss several algorithms that were implemented by the author, together with various improvements and details that turned out to be important for making the algorithms as efficient as possible in practice. In addition, a comprehensive experimental study is also presented, in which the implemented algorithms are compared with each other and with several other solvers, including the ones that have been widely used as benchmarks for a long time. Two algorithms of the author turned out to be highly efficient and robust; they usually outperform other implementations, often by an order of magnitude.

These algorithms have been applied by numerous subsequent studies to solve real-world problems related to, for example, transportation and logistics [178, 194, 237, 242], integrated circuit design [139, 199], compilers [172, 173], combinatorial auctions [181, 182], design of electric power systems [52], police patrol routing [77], microscope image processing [10], and genome sequencing [221].

1.2 Maximum common subgraphs

The other task of primary focus in this thesis is to find the maximum common subgraph of two undirected graphs having node and edge labels. This is a classic NP-hard optimization problem, which is applied in various fields, including computational chemistry and biology, pattern recognition, and computer vision. We study this task in the context of cheminformatics, where the structural similarity of molecules is to be analyzed in many applications. The typical model of a molecular structure is an undirected graph in which the nodes represent atoms and the edges represent chemical bonds. Given two molecular graphs, a practical approach to measure their similarity is by means of their largest common subgraph, because it is straightforward and has a clear visual justification. A simple example is shown in Figure 1.2.



(a) Input structures

(b) Maximum common subgraph

Figure 1.2: Example of maximum common subgraph search in the context of cheminformatics. The first image displays two molecular structures whose similarity is to be determined. The second image depicts the same structures with their maximum common subgraph highlighted.

Despite the computational complexity of this problem, its solution methods are usually required to be both fast and accurate (in terms of the approximation of the actual optimal result). To satisfy these requirements, Péter Englert and the author developed efficient heuristic algorithms incorporating novel ideas to improve upon their effectiveness. Moreover, special methods were also devised to handle additional requests that arise in some chemical applications. The provided implementations were thoroughly evaluated and compared with two other solution methods, to which they turned out to be greatly superior, in terms of both the running time and accuracy. These results are discussed in Chapter 4.

This research was conducted at ChemAxon (`https://chemaxon.com`), which is a leading software company developing applications and services for the pharmaceutical industry. The presented algorithms are utilized in multiple products of the company, especially for visualizing the structural similarity of molecular graphs.

1.3 The LEMON library

LEMON [164] is an open-source C++ template library with focus on the solution of graph optimization problems. Chapter 5 briefly introduces its features, which involve efficient implementations of algorithms, data structures, and other practical tools related to graphs and combinatorial optimization.

During his research, the author has made significant contribution to the development of LEMON, which is led by Alpár Jüttner. Among others, the author implemented several methods for finding minimum-cost flows, minimum-mean cycles, and maximum cliques. All algorithms discussed in Chapter 3 are included in LEMON with full source code, which makes it easier to evaluate them and combine them with other algorithms and tools to solve real-world problems.

Chapter 2

Preliminaries

This chapter is a brief summary of mathematical concepts and fundamental results that are important for the discussion of this thesis. For detailed introduction and proofs, the reader may refer to the excellent books of Schrijver [214], Korte and Vygen [159], Ahuja et al. [13], Cormen et al. [67], and András Frank [101].

2.1 Basic concepts and notations

We assume familiarity with the basic concepts and results of graph theory, and also with common data structures and algorithms related to graphs. Fundamental results and terminology of operations research and computational complexity theory are also used.

Throughout this work, we always study finite graphs. Parallel edges and loops are not considered for the sake of simplicity, but the presented results and algorithms can easily be adapted to allow such edges. We denote an *undirected graph* as $G = (V, E)$, where V is the set of *nodes* and E is the set of *edges*. An edge connecting nodes v and w is an unordered pair of nodes, denoted as vw (or wv). Similarly, a *directed graph* is denoted as $D = (V, A)$, where A is the set of directed edges, which we refer to as *arcs*. An arc from node i to node j is denoted as ij, which means an ordered pair of nodes in this case. Unless otherwise stated, n and m denote the number of nodes and edges (or arcs) in the graph, respectively.

The notions of (directed or undirected) *path* and *walk* are used in the usual manner. A *cycle* is always meant to be a simple cycle, that is, a closed walk containing distinct nodes, except for the ending node, which is the same as the starting node.

We often assign numbers to the nodes and arcs of a directed graph $D = (V, A)$, especially in forms of coefficients and variables of linear programming (LP) problems. In such cases, we use the notations b_i and c_{ij} instead of $b(i)$ and $c(ij)$ for functions $b : V \to \mathbb{R}$ and $c : A \to \mathbb{R}$, respectively.

2.2 Running time of algorithms

The running time (or time complexity) of an algorithm is measured as the number of elementary steps (including arithmetic operations) and expressed as a function of the input size. We use the standard $O(...)$ notation to describe the asymptotic behavior of running time.

In the case of graph algorithms, the input size is determined by the size of the graph (n and m) and the space (number of bits) required to store all numbers involved in the input data, for example, arc costs or capacities. A graph algorithm is *polynomial* (or *weakly polynomial*) if its running time is bounded by a polynomial in the input size. If the input numbers are nonnegative integers, each at most K, then this means a running time bounded by a polynomial in n, m, and $\log K$. In contrast, a graph algorithm is said to be *strongly polynomial* if its running time is bounded by a polynomial in only n and m, independently of the input numbers. In both cases, we also assume that the space required to store all numbers in intermediate computations is bounded by a polynomial in the size of the input numbers.

2.3 Shortest paths

The single-source shortest path problem is probably the most well-known graph optimization problem with countless applications. In a directed graph with arc costs (or weights), we are looking for directed paths of minimum total cost from a designated source node to all other nodes (or to a single target node).

In the case of nonnegative costs, shortest paths can be found with Dijkstra's famous algorithm. Using Fibonacci heaps [102], it runs in strongly polynomial time $O(m+n\log n)$. However, other data structures (for example, binary or d-ary heap) are typically more efficient in practice (see, e.g., [60]). If the arc costs are integers, better weakly polynomial bounds can also be achieved. The radix heap data structure of Ahuja et al. [14] yields $O(m + n\sqrt{\log C})$ time for nonnegative integer costs of at most C, while the best time bound $O(m + n\min\{\log\log n, \log\log C\})$ is achieved using the integer priority queues of Thorup [231, 232].

If negative costs are also allowed, then we usually require the cost function to be *conservative*, which means that each directed cycle has nonnegative total cost, because the shortest path problem is NP-complete without this restriction. The well-known Bellman-Ford algorithm solves the problem in strongly polynomial time $O(nm)$ or detects that a negative-cost cycle exists, so the cost function is not conservative. In the case of integer costs, improved running time bounds can also be achieved.

2.4 Potentials and reduced costs

Let $D = (V, A)$ be a directed graph with a cost function $c : A \to \mathbb{R}$. Given a node potential function $\pi : V \to \mathbb{R}$, the *reduced cost function* c^π is defined as

$$c^\pi_{ij} = c_{ij} + \pi_i - \pi_j \qquad \forall ij \in A. \tag{2.1}$$

Note that c^π_{ij} measures the relative cost of the arc ij with respect to the potentials of its end nodes.

The following proposition describes basic properties of reduced costs, which are simple corollary of the definition.

Proposition 2.1. *Let $\pi : V \to \mathbb{R}$ be an arbitrary potential function.*

- *For any directed path P from node s to node t, $\sum_{ij \in P} c^{\pi}_{ij} = \sum_{ij \in P} c_{ij} + \pi_s - \pi_t$.*
- *For any directed cycle W, $\sum_{ij \in W} c^{\pi}_{ij} = \sum_{ij \in W} c_{ij}$.*

Therefore, the minimum-cost paths and cycles are the same with respect to c and c^{π}.

A potential function π is called feasible if and only if

$$c^{\pi}_{ij} \geq 0 \qquad \forall ij \in A. \tag{2.2}$$

There is a fundamental connection between feasible potentials and conservative cost functions, which is stated as follows.

Theorem 2.2. *For any cost function $c : A \to \mathbb{R}$, a feasible potential $\pi : V \to \mathbb{R}$ exists if and only if c is conservative. Moreover, if c is integer-valued, then π can also be integer-valued.*

Proof. If a feasible potential π exists, then each cycle has nonnegative total cost with respect to c^{π}, and hence also with respect to c, according to Proposition 2.1. So c is conservative.

On the other hand, let c be a conservative cost function, for which a feasible potential is to be found. We add a new node s to the graph along with an arc si of zero cost for each original node i. Then we run the Bellman-Ford algorithm on this extended graph starting from node s to obtain the distance label d_i for each node i. Due to the optimality conditions of the shortest path problem, $d_j \leq d_i + c_{ij}$ holds for each arc $ij \in A$. That is, $c^{d}_{ij} \geq 0$ for each arc ij, so d is a feasible potential. Furthermore, if c is integer-valued, then the behavior of the algorithm ensures that d is also integer-valued. ■

This proof is presented here because of its importance. It reveals the connection between potentials and distance labels, and it provides an algorithm that runs in $O(nm)$ time and either finds a feasible potential or detects that the cost function is not conservative. Furthermore, note that a feasible potential can be used to transform a conservative cost function into a nonnegative cost function such that the shortest paths between any pair of nodes remain the same (although their costs may change).

2.5 Minimum-mean cycles

Suppose that a given cost function c is not conservative. We may attempt to make it conservative by increasing the cost of each arc uniformly by $\epsilon > 0$, which is denoted as $c+\epsilon$. By Theorem 2.2, $c+\epsilon$ is conservative if and only if there exists a potential function π so that $c^{\pi}_{ij} \geq -\epsilon$ holds for each arc ij.

The *mean cost* of a directed cycle is the average cost of its arcs. Note that increasing c uniformly by ϵ increases the mean cost of each cycle by ϵ, which directly implies the following proposition.

Proposition 2.3. *For a non-conservative cost function $c : A \to \mathbb{R}$, the smallest $\epsilon > 0$ for which $c + \epsilon$ is conservative equals to the negative of the minimum mean cost of a directed cycle.*

A directed cycle of minimum mean cost is called a *minimum-mean cycle*. Karp [148] proposed a strongly polynomial algorithm for finding minimum-mean cycles, which is based on the following theorem.

Theorem 2.4. *Given an arbitrary cost function, let $d_k(i)$ $(k = 0, 1, \ldots, n)$ denote the minimum total cost of a walk with exactly k arcs, ending at node i (or ∞ if no such walk exists). The minimum mean cost of a directed cycle (if one exists) is equal to*

$$\min_{i \in V} \max_{\substack{0 \le k \le n-1 \\ d_k(i) < \infty}} \frac{d_n(i) - d_k(i)}{n-k}. \tag{2.3}$$

For each node $i \in V$, $d_0(i) = 0$ and $d_{k+1}(i) = \min\{d_k(j) + c_{ji} : ji \in A\}$. This recursion and Theorem 2.4 suggest a dynamic programming algorithm that runs in $O(nm)$ time and either finds a minimum-mean cycle or detects that the graph is acyclic. Several other algorithms also exist, which can be more efficient in practice, see [74, 75, 114].

2.6 Maximum flows

Network flow problems form an important class of combinatorial optimization problems. They have been widely investigated in the last six decades, and they are essential in a huge number of different applications.

The simplest and most well-known network flow problem is the maximum flow problem. Let $u : A \to \mathbb{R}$ be a capacity function ($u_{ij} \ge 0$ for each arc $ij \in A$), and let $s, t \in V$ be two distinct nodes of the directed graph $D = (V, A)$. The maximum flow problem is to transport as many units as possible from node s to node t with respect to the arc capacities. Representing the solution as a function $x : A \to \mathbb{R}$, the problem can be stated as follows:

$$\max \sum_{j \,:\, sj \in A} x_{sj}, \tag{2.4}$$

subject to

$$\sum_{j \,:\, ij \in A} x_{ij} - \sum_{j \,:\, ji \in A} x_{ji} = 0 \qquad \forall i \in V \setminus \{s, t\}, \tag{2.5}$$

$$0 \le x_{ij} \le u_{ij} \qquad \forall ij \in A. \tag{2.6}$$

We refer to (2.5) as *flow conservation constraints* and (2.6) as *capacity constraints*. A function $x : A \to \mathbb{R}$ is a (feasible) *flow* if it satisfies all of these constraints; while it is a *pseudoflow* if it satisfies the capacity constraints (2.6), but might violate (2.5).

Many different algorithms have been devised for solving the maximum flow problem since the 1950s, but it was proved only in 2013 by Orlin [191] that any instance of the problem can be solved in $O(nm)$ time. A brief survey of the most important algorithms and running time bounds is provided in Table 2.1.

	$O(nmU)$	Ford and Fulkerson (1956) [94] *augmenting path*
	$O(nm^2)$	Dinic (1970) [81], Edmonds and Karp (1972) [88] *shortest augmenting path*
	$O(n^2m)$	Dinic (1970) [81] *shortest augmenting path, layered network*
	$O(m^2 \log U)$	Edmonds and Karp (1970) [87, 88] *capacity-scaling*
	$O(n^3)$	Karzanov (1974) [149] Malhotra, Kumar, Maheshwari (1978) [171] Goldberg and Tarjan (1986) [124, 126] *push-relabel*
	$O(nm \log n)$	Sleator (1980) [218], Sleator and Tarjan (1981) [219, 220] *dynamic trees*
	$O(nm \log(n^2/m))$	Goldberg and Tarjan (1986) [124, 126] *push-relabel with dynamic trees*
	$O(nm + n^2\sqrt{\log U})$	Ahuja, Orlin, and Tarjan (1989) [16] *push-relabel with scaling*
	$O(nm \log((n/m)\sqrt{logU} + 2))$	Ahuja, Orlin, and Tarjan (1989) [16] *push-relabel with scaling and dynamic trees*
$*$	$O(n^3/\log n)$	Cheriyan, Hagerupm and Mehlhorn (1990) [57, 58]
	$O(nm \log_{m/(n \log n)} n)$	King, Rao, and Tarjan (1994) [155]
$*$	$O(m \min\{n^{2/3}, m^{1/2}\} \log(n^2/m) \log U)$	Goldberg and Rao (1997) [121, 122]
$*$	$O(nm)$	Orlin (2013) [191] (together with the algorithm of King, Rao, and Tarjan)

Table 2.1: Survey of maximum flow algorithms and complexity bounds. In the case of time bounds involving U, we assume integer capacities, each at most U. $*$ indicates an asymptotically best complexity bound.

2.7 Residual network

Most network flow algorithms rely on the concept of *residual network*. It represents the idea to describe a flow or pseudoflow in terms of incremental changes with respect to a given pseudoflow (typically, the solution at an intermediate step of an algorithm).

Given a capacity function u and a pseudoflow x, the residual network corresponding to x is defined as follows. Let $D^x = (V, A^x)$ be a directed graph on the original node set V. With each original arc $ij \in A$, we associate a *forward arc* $ij \in A^x$ with residual capacity $r^x_{ij} = u_{ij} - x_{ij}$ if $r^x_{ij} > 0$, and a *backward arc* $ji \in A^x$ with residual capacity $r^x_{ji} = x_{ij}$ if $r^x_{ji} > 0$. These residual capacities represent the additional amount of flow that can be carried from node i to node j, or from node j to node i (by increasing or decreasing the flow value x_{ij} on the original arc, respectively).

Some flow problems also involve arc costs, for example, the minimum-cost flow problem (see Chapter 3). Given a cost function c, for each arc $ij \in A$, the corresponding forward arc has cost c_{ij}, while the cost of the backward arc is $-c_{ij}$ as it represents the reduction of the flow amount on the original arc.

We remark that the concept of residual network introduces notational difficulties. If the original graph contains both arcs ij and ji for any pair of nodes i and j, then a residual network may contain parallel arcs with different residual capacities (and costs). In practice, we assume an appropriate graph representation that can handle parallel arcs, but we keep using the simple notations ij and ji in the rest of this thesis for the sake of simplicity.

2.8 Maximum cliques

In an undirected graph, a *clique* is a subset of nodes, all adjacent to each other (also called a complete subgraph). A classic NP-hard graph optimization problem, called the *maximum clique problem*, is to find a clique with the largest possible number of nodes. When expressed as a decision problem, it considers whether a graph contains a clique of (at least) k nodes for a given integer $k > 0$, which is NP-complete [111]. A great variety of algorithms exists for finding maximum cliques (both exact and heuristic) [193, 248].

In contrast, a clique C is called a *maximal clique* if it cannot be extended, that is, if the graph does not have a larger clique containing all nodes of C. Considering the maximum clique problem, a maximal clique is a local optimum.

2.9 Graph and subgraph isomorphism

Two undirected graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if a bijective function $f : V \to V'$ exists such that any pair of nodes v and w are adjacent in G if and only if $f(v)$ and $f(w)$ are adjacent in G'. In this case, f is called an *isomorphism*.

The *graph isomorphism problem* is to determine whether two graphs are isomorphic, which is a remarkable problem because of its applications and theoretical significance. It is one of only a few standard computational problems that belong to the complexity class NP, but are not known to be either in P or NP-complete [111]. (These two classes are disjoint subsets of NP, provided that P≠NP.) However, polynomial-time graph isomorphism algorithms are known for various graph classes, such as trees and planar graphs, bounded-degree graphs, and permutation graphs [62, 137, 170].

A strongly related problem is the *subgraph isomorphism problem*: given two graphs G and G', the task is to determine whether G' contains a subgraph isomorphic to G. Since we usually do not distinguish between isomorphic graphs, we can also say that G' contains G. This problem is a generalization of both the maximum clique problem and the Hamiltonian cycle problem, so it is NP-complete. Garey et al. [112] showed that the Hamiltonian cycle problem remains NP-complete even for planar graphs with maximum node degree three, so the subgraph isomorphism problem also remains NP-complete. This problem is also referred to as *subgraph matching*, which emphasizes the task of finding an appropriate subgraph as opposed to the decision problem.

The *induced subgraph isomorphism problem* is to determine for two graphs G and G' whether G' contains an *induced subgraph* H that is isomorphic to G. This is a restricted version of the subgraph isomorphism problem as H is required to contain each edge of G' that connects two nodes of H. This problem is also NP-complete because the maximum clique problem is a special case of it (but the Hamiltonian cycle problem is not). Obviously, both the subgraph isomorphism and induced subgraph isomorphism problems are generalizations of the graph isomorphism problem.

Applications of (sub)graph isomorphism problems (or graph matching) range from bioinformatics and cheminformatics [42, 91, 246] to pattern recognition and computer vision [47, 64]. Over the last decades, several algorithms have been developed and compared in experimental studies (see, e.g., [54, 66, 146, 162, 236]).

2.10 Labeled graphs

In many applications of (sub)graph isomorphism problems, labeled graphs are considered. In cheminformatics, for example, molecules are typically represented as undirected labeled graphs, in which the nodes represent the atoms with atom type labels (C, N, O, etc.) and the edges represent the chemical bonds with bond type labels (single, double, aromatic, etc.). In such cases, (sub)graph isomorphism functions are required to be both edge-preserving and label-preserving. Here we provide formal definitions of labeled graphs and corresponding (sub)graph isomorphism.

Definition 2.5 (Labeled graph). *A labeled graph is defined as an undirected simple graph $G = (V, E, L_V, L_E, \ell_V, \ell_E)$, where V and E are the sets of nodes and edges, respectively, L_V and L_E are countable sets of node and edge labels, respectively, and $\ell_V : V \to L_V$ and $\ell_E : E \to L_E$ are the labeling functions.*

Let $G = (V, E, L_V, L_E, \ell_V, \ell_E)$ and $G' = (V', E', L_{V'}, L_{E'}, \ell_{V'}, \ell_{E'})$ be labeled graphs. Graph and (induced) subgraph isomorphism between G and G' are defined as follows.

Definition 2.6 (Subgraph isomorphism for labeled graphs). *G is isomorphic to a subgraph of G' (or simply, G is a subgraph of G') if an injective function $f : V \to V'$ exists such that:*

$$\forall v, w \in V : \textit{if } vw \in E, \textit{ then } f(v)f(w) \in E'; \tag{2.7}$$

$$\forall v \in V : \ell_V(v) = \ell_{V'}(f(v)); \tag{2.8}$$

$$\forall vw \in E : \ell_E(vw) = \ell_{E'}(f(v)f(w)). \tag{2.9}$$

Definition 2.7 (Induced subgraph isomorphism for labeled graphs). *G is isomorphic to an induced subgraph of G' (or simply, G is an induced subgraph of G') if an injective function $f : V \to V'$ exists such that:*

$$\forall v, w \in V : vw \in E \textit{ if and only if } f(v)f(w) \in E'; \tag{2.10}$$

$$\forall v \in V : \ell_V(v) = \ell_{V'}(f(v)); \tag{2.11}$$

$$\forall vw \in E : \ell_E(vw) = \ell_{E'}(f(v)f(w)). \tag{2.12}$$

(The latter two constraints are the same as in Definition 2.6.)

Definition 2.8 (Graph isomorphism for labeled graphs)**.** *G and G' are isomorphic if there exists a bijective function $f : V \to V'$ for which the constraints of Definition 2.7 hold. (That is, they are induced subgraphs of each other.)*

2.11 Line graphs

The *line graph* (or *edge graph*) of an undirected graph G is an undirected simple graph $L(G)$ that represents adjacencies between edges of G. The nodes of $L(G)$ correspond to the edges of G, and two nodes in $L(G)$ are adjacent if and only if their corresponding edges have a common endpoint in G.

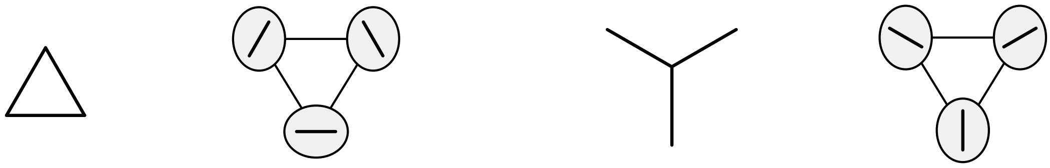
If G is a labeled graph, then we assign to a node in $L(G)$ the label of the corresponding edge along with the labels of the nodes it connects, while the label of an edge xy in $L(G)$ is set to the node label of the common endpoint of the edges corresponding to x and y (this node is uniquely defined if G is simple graph).

It is easy to see that if G is connected, then $L(G)$ is also connected, but a graph that contains isolated nodes, and is thereby disconnected, may also have a connected line graph. Another important result considers graph isomorphism and line graphs. If two graphs G and G' are isomorphic, then $L(G)$ and $L(G')$ are obviously isomorphic. For the other direction, Whitney [244] proved the following theorem.

Theorem 2.9 (Whitney)**.** *Two connected graphs are isomorphic if and only if their line graphs are isomorphic, with the single exception of the complete graph K_3 and the complete bipartite graph $K_{1,3}$, which are not isomorphic, but their line graphs are isomorphic.*

K_3 is also known as *triangle* or Δ *graph*, while $K_{1,3}$ is also known as *claw* or Y *graph*. They are obviously not isomorphic, but the line graph of both of them is K_3, as illustrated in Figure 2.1.

Furthermore, note that if G is a subgraph of G', then $L(G)$ is an induced subgraph of $L(G')$ because two edges in G are adjacent if and only if they are adjacent in G'.



(a) K_3 (Δ graph) and its line graph

(b) $K_{1,3}$ (Y graph) and its line graph

Figure 2.1: K_3 and $K_{1,3}$ have isomorphic line graphs.

Chapter 3

Minimum-cost flows

This chapter is about the *minimum-cost flow* (MCF) problem and its solution methods. It presents achievements related to the efficient implementation and improvements of various algorithms, along with a comprehensive experimental study. The author carried out this work with the guidance of his supervisor, Zoltán Király. The results were published in the articles [2, 3, 6], and preliminary work appeared in the MSc thesis of the author [9].

The chapter is organized as follows. Section 3.1 provides an overview of basic definitions, notations, and results related to the MCF problem. Section 3.2 is a detailed description of the algorithms and improvements implemented as part of this work. Finally, Section 3.3 presents an extensive computational study demonstrating the practical significance of the results, compared with other available implementations of MCF algorithms.

3.1 The minimum-cost flow problem

The MCF problem is a fundamental model in combinatorial optimization. It is to find a minimum-cost transportation of a specified amount of commodity from a set of supply nodes to a set of demand nodes in a directed network with capacity constraints and costs assigned to the arcs. This problem has a remarkably wide range of applications in diverse fields, for example, telecommunication, transportation, network design, resource planning, scheduling, engineering, manufacturing, and evacuation planning [13]. Furthermore, it often arises as a subtask of more complex optimization problems [99, 169].

The MCF problem has enormous literature, comprising several books and hundreds of journal articles. Over the last six decades, a great number of algorithms have been devised for solving this problem, and they have been thoroughly studied both from theoretical and from practical aspects. Comprehensive introduction of network flow theory and algorithms can be found, for example, in [13, 123, 159, 214]. More recent surveys of MCF algorithms and available implementations are presented in [2, 217].

3.1.1 Problem statement

The single-commodity, linear *minimum-cost flow* (MCF) problem is defined as follows. Let $D = (V, A)$ be a weakly connected directed graph consisting of $n = |V|$ nodes and $m = |A|$ arcs. We associate with each arc $ij \in A$ a *lower bound* $l_{ij} \in \mathbb{R}$, an *upper bound* (or *capacity*) $u_{ij} \in \mathbb{R} \cup \{\infty\}$, $u_{ij} \geq l_{ij}$, and a *cost* $c_{ij} \in \mathbb{R}$, which denotes the cost per unit flow on the arc. Each node $i \in V$ has a signed *supply* value b_i. If $b_i > 0$, then node i is called a *supply node* with a supply of b_i; if $b_i < 0$, then node i is called a *demand node* with a demand of $-b_i$; and if $b_i = 0$, then node i is referred to as a *transshipment node.* The solution of the problem is represented by flow values $x_{ij} \in \mathbb{R}$ assigned to the arcs. Therefore, the MCF problem can be stated as

$$\min \sum_{ij \in A} c_{ij} x_{ij}, \tag{3.1}$$

subject to

$$\sum_{j\,:\,ij \in A} x_{ij} - \sum_{j\,:\,ji \in A} x_{ji} = b_i \qquad \forall i \in V, \tag{3.2}$$

$$l_{ij} \le x_{ij} \le u_{ij} \qquad \forall ij \in A. \tag{3.3}$$

Similarly to the maximum flow problem (see Section 2.6), we refer to (3.2) as *flow conservation constraints* (or *mass balance constraints*) and (3.3) as *capacity constraints* (or *flow bound constraints*). A function $x : A \to \mathbb{R}$ is called a *feasible flow* (or *feasible solution*) if it satisfies all constraints defined in (3.2) and (3.3). A feasible flow is called *optimal* if it also minimizes the total flow cost over feasible solutions, see (3.1).

The MCF problem is a special type of *linear programming* (LP) problem, so it can be solved with general LP methods. However, the network model has various important advantages. First of all, graph optimization algorithms can solve the problem much more efficiently, especially in the case of large-scale instances that would require hundreds of thousands or millions of rows and columns when expressed as LP problems. Moreover, the graph model and related algorithms are more intuitive, and they yield integer-valued solutions if all input data are integers.

3.1.2 Special cases

Several well-known graph optimization problems can be formulated as special cases of the MCF problem, which also justifies the importance of this model and its solution methods.

Shortest path problem

The shortest path problem, which was discussed in Section 2.3, is an important special case of the MCF problem. Suppose that we have a conservative cost function c. For a given pair of nodes $s, t \in V$, if we set $b_s = 1$, $b_t = -1$ and $b_i = 0$ for each other node i, and we set $l_{ij} = 0$, $u_{ij} = 1$ for each arc ij, then an optimal solution of the MCF problem sends one unit of flow from node s to node t along a shortest path (assuming integer-valued solution). More precisely, the flow is sent along a shortest walk that may contain directed cycles of zero cost, but a shortest path can easily be derived from such a walk. If shortest paths are to be found from node s to every other node, then we can set $b_s = n - 1$ and $b_i = -1$ for each other node i, and let $l_{ij} = 0$, $u_{ij} = n - 1$ for each arc ij. An optimal solution in this network sends one unit of flow from s to each other node along a shortest path (or walk).

Maximum flow problem

Recall the maximum flow problem from Section 2.6. This model involves arc capacities, but without lower bounds and arc costs, so let $l_{ij} = 0$ and $c_{ij} = 0$ for each arc ij. Furthermore, we set $b_i = 0$ for each node i, and we add an additional arc from the target node t to the source node s, for which $l_{ts} = 0$, $u_{ts} = \infty$, and $c_{ts} = -1$. An optimal flow in this extended network maximizes the flow amount on the newly added arc ts (because it has negative cost), which also means that the same amount of flow is sent from s to t in the original network (because $b_i = 0$ for every node i). So an optimal MCF solution results in a maximum flow in the original graph.

Assignment problem

In the assignment problem, we have two sets V_1 and V_2 so that $|V_1| = |V_2|$, a set of possible assignments $A \subseteq V_1 \times V_2$, and a cost c_{ij} associated with each assignment $ij \in A$. The task is to assign exactly one element of V_2 to each element of V_1 and exactly one element of V_1 to each element of V_2 such that the total cost of the assignment is minimized. That is, we are looking for a *minimum-cost perfect matching* in the bipartite graph defined by the possible assignments. This problem can also be expressed as an MCF problem as follows. Let $D = (V_1 \cup V_2, A)$, $b_i = 1$ for each $i \in V_1$, $b_j = -1$ for each $j \in V_2$, and for each arc ij, let $l_{ij} = 0$ and $u_{ij} = 1$, and an integer-valued solution is required.

In 1955, Kuhn [161] developed a combinatorial optimization algorithm for solving the assignment problem, which he named the *Hungarian method* because it was largely based on previous works of two Hungarian mathematicians, Dénes Kőnig and Jenő Egerváry. Later, this algorithm became one of the fundamental methods of operations research, which influenced the development of various primal–dual algorithms related to network flows, matchings, and matroids [100].

Transportation problem

The transportation problem is a special case of MCF problem where the node set V is partitioned into two subsets V_1 and V_2 (with possibly different cardinality) so that each node $i \in V_1$ is a supply node and each node $j \in V_2$ is a demand node, and $A \subseteq V_1 \times V_2$. A typical example of this problem is the distribution of goods from warehouses to customers. Some applications do not require capacity constraints, that is, $l_{ij} = 0$, $u_{ij} = \infty$ for every arc ij.

Circulation problem

The circulation problem is an MCF problem with only transshipment nodes. That is, $b_i = 0$ for each $i \in V$, so all the flow circulates around the network. In such case, positive lower bounds or negative arc costs can incorporate the requirement for transporting flow instead of supplies and demands of the nodes. For example, see the expression of the maximum flow problem as an MCF problem.

3.1.3 Assumptions and transformations

In the followings, we make some assumptions that simplifies the MCF problem, and we describe the corresponding methods for checking and transforming instances of the problem. These methods are compatible with each other and can be applied in the order in which they are presented here.

Assumption 3.1. *All capacities are finite, that is, $u_{ij} < \infty$ for every arc ij.*

This assumption can be made without loss of generality, provided that the total cost of a feasible solution is bounded from below. The unbounded case can be detected according to the following theorem [123].

Theorem 3.2. *The MCF problem has an optimal solution if and only if it has a feasible solution and the network contains no negative-cost cycle consisting of arcs with infinite capacity.*

Considering the subgraph containing only the arcs of infinite capacity, the Bellman-Ford algorithm can be used to detect in $O(nm)$ time whether the network contains an uncapacitated negative-cost cycle. If such a cycle exists, then the problem is unbounded, otherwise the infinite capacities can be replaced with sufficiently large finite values.

Assumption 3.3. *All input data are integers (lower and upper bounds, costs, and supply values), and an integer-valued optimal solution is to be found.*

Practically, we can assume integer data without loss of generality. Since real numbers are typically represented with a certain precision in a computer, they can be replaced with integers if we scale them by multiplying with a sufficiently large constant.

Furthermore, we will show that if an MCF problem defined with integer data has an optimal solution, then it also has an integer-valued optimal solution (see Section 3.2.1). Therefore, we only consider integer-valued flows in the followings.

Assumption 3.4. *All lower bounds are zero, that is, $l_{ij} = 0$ for every arc ij.*

Every instance of the MCF problem can be transformed into an instance with zero lower bounds. Let ij denote an arc with $l_{ij} \neq 0$. We can eliminate this lower bound by sending l_{ij} units of flow on the arc ij, and modify the problem accordingly in order to find the additional amount of flow to be sent on the arc. That is, l_{ij} is set to zero, u_{ij} is set to $u_{ij} - l_{ij}$, and b_i is decreased by l_{ij}, while b_j is increased by l_{ij}. We can apply this transformation to each arc with nonzero lower bound, thereby transforming the problem into another one with zero lower bounds. It is easy to see that there is a bijective mapping between the feasible flows in the original network and in the transformed network such that the difference between their total cost always equals to the constant $\sum_{ij \in A} c_{ij} l_{ij}$. That is, the two problems are equivalent.

Assumption 3.5. *All arc costs are nonnegative, that is, $c_{ij} \geq 0$ for every arc ij.*

Similarly to the previous assumption, this one can also be ensured by an appropriate transformation of the network. We first saturate each arc of negative cost, and then we determine the amount of flow that should not actually be sent on the arc. Let ij denote an arc with $c_{ij} < 0$. Then we replace arc ij with the reverse arc ji with cost $c_{ji} = -c_{ij}$ and capacity $u_{ji} = u_{ij}$, and b_i is decreased by u_{ij}, while b_j is increased by u_{ij} (we assume that $l_{ij} = 0$). This modification is applied to every arc having negative cost. It can easily be seen that the feasible flows in the transformed network have a bijective mapping to the feasible flows of the original problem in a way that the difference in their total cost is constant, so the two problems are equivalent.

We remark that this transformation may introduce parallel arcs even if the original graph does not contain such arcs. This is merely a notational difficulty (similarly to the residual network, see Section 2.7) because in practice, we can always apply a graph representation that handles parallel arcs. In the following discussions, however, we keep the notations ij and ji for the sake of simplicity.

Assumption 3.6. *The sum of supply values is zero, that is,* $\sum_{i \in V} b_i = 0$.

The flow conservation constraints (3.2) imply that this assumption is required in order to have a feasible solution:

$$\sum_{i \in V} b_i = \sum_{i \in V} \left(\sum_{j\,:\,ij \in A} x_{ij} - \sum_{j\,:\,ji \in A} x_{ji} \right) = \sum_{ij \in A} x_{ij} - \sum_{ji \in A} x_{ji} = 0. \tag{3.4}$$

Taking all these assumptions into account, we formulate the MCF problem again. In the following sections, we consider this simplified version of the problem.

Definition 3.7 (MCF problem)**.** *Let* $D = (V, A)$ *be a weakly connected directed graph with* $b : V \to \mathbb{Z}$ *supply values,* $u : A \to \mathbb{Z}$ *capacities, and* $c : A \to \mathbb{Z}$ *costs, for which* $u_{ij} \geq 0$ *and* $c_{ij} \geq 0$ *for each* $ij \in A$*, and* $\sum_{i \in V} b_i = 0$*. Denoting the solution as* $x : A \to \mathbb{Z}$*, the MCF problem is stated as:*

$$\min \sum_{ij \in A} c_{ij} x_{ij}, \tag{3.5}$$

subject to

$$\sum_{j\,:\,ij \in A} x_{ij} - \sum_{j\,:\,ji \in A} x_{ji} = b_i \qquad \forall i \in V, \tag{3.6}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad \forall ij \in A. \tag{3.7}$$

3.1.4 Additional concepts and notations

We introduce a few additional concepts and notations that are required for further discussions. A *pseudoflow* is a function $x : A \to \mathbb{Z}$ that satisfies the nonnegativity and capacity constraints (3.7), but might violate the flow conservation constraints (3.6). That is, a node might have a certain amount of undelivered supply or unfulfilled demand, which is called the *excess* of the node. Formally, for a given pseudoflow x, the excess of node i is a signed value e_i^x defined as

$$e_i^x = b_i + \sum_{j\,:\,ji \in A} x_{ji} - \sum_{j\,:\,ij \in A} x_{ij}. \tag{3.8}$$

Node i is referred to as an *excess node* if $e_i^x > 0$, and as a *deficit node* if $e_i^x < 0$. Furthermore, note that $\sum_{i \in V} e_i^x = \sum_{i \in V} b_i = 0$. A pseudoflow x is a feasible flow if and only if $e_i^x = 0$ for every node i.

The *residual network* associated with a pseudoflow is defined in the same way as for the maximum flow problem, see Section 2.7. For an original arc $ij \in A$, the cost of the corresponding forward arc ij in the residual network is defined as c_{ij}, while the cost of the backward arc ji is $-c_{ij}$.

Running time of MCF algorithms is measured in terms of the size of the graph and the magnitudes of the input numbers. Let U denote the largest supply value or capacity:

$$U = \max\{\max\{|b_i| : i \in V\}, \max\{u_{ij} : ij \in A\}\}, \tag{3.9}$$

and let C denote the largest arc cost:

$$C = \max\{c_{ij} : ij \in A\}. \tag{3.10}$$

Using these notations and the concepts discussed in Section 2.2, an algorithm is called *(weakly) polynomial* if its running time is bounded by a polynomial in n, m, $\log U$, and $\log C$; while a *strongly polynomial* bound is a polynomial function of n and m (regardless of U and C).

3.1.5 Feasible flows

A feasible solution to the MCF problem can be found by solving a maximum flow problem as follows. We extend the network with a source node s and a target node t so that for each supply node i, we add an arc si with capacity $b_i > 0$, and for each demand node j, we add an arc jt with capacity $-b_j > 0$. It can easily be shown that the original MCF problem has a feasible solution if and only if a maximum flow in the extended network saturates all outgoing arcs of node s and all incoming arcs of node t. Furthermore, restricting such a maximum flow to the original network results in a feasible flow of the MCF problem. If all capacities are integers, then an integer-valued feasible flow can be found this way.

3.1.6 Optimality conditions

In the followings, we formulate optimality conditions for the MCF problem in terms of both the residual network and the original network. These theorems are essential in multiple aspects. Not only do they provide simple methods for checking the optimality of a solution, but they also suggest algorithms for solving the problem. These results are discussed, for example, in [13, 98, 123, 159, 214].

Theorem 3.8 (Negative cycle optimality conditions)**.** *A feasible solution x of the MCF problem is optimal if and only if the residual network D^x contains no directed cycle of negative total cost (that is, the cost function is conservative in D^x).*

One direction of this theorem is obvious. If the residual network of a feasible flow contains a negative-cost cycle, then augmenting positive amount of flow along such a cycle results in another feasible flow with lower total cost. Therefore, no negative-cost cycle can exist in the residual network of an optimal solution. The converse direction can be shown based on the observation that any feasible flow can be decomposed into a finite set of augmenting paths and cycles.

An equivalent formulation of Theorem 3.8 can be stated using node potentials and reduced costs. Recall from Section 2.4 that the reduced cost of an arc ij is defined as $c^{\pi}_{ij} = c_{ij} + \pi_i - \pi_j$ with respect to a node potential function π. Actually, node potentials represent the LP dual solution of the MCF problem (see Section 3.1.7).

Theorem 3.9 (Reduced cost optimality conditions)**.** *A feasible solution x of the MCF problem is optimal if and only if for some potential function π, $c^{\pi}_{ij} \geq 0$ holds for each arc ij in the residual network D^x (that is, a feasible potential exists in D^x).*

The equivalence of these two theorems immediately follows from Theorem 2.2, which we proved in Section 2.4. That is, a cost function is conservative if and only if a feasible potential exists, and integer potentials can be found if the costs are integers.

We remark that the potential π_i can also be viewed as the price of a unit of some commodity at node i. In this sense, $c^{\pi}_{ij} = c_{ij} + \pi_i - \pi_j$ is the total cost of buying a unit of commodity at node i, delivering it to node j, and then selling it there. Theorem 3.9 expresses that it would not be beneficial to deliver more units of commodity on any arc of the residual network in the case of an optimal transportation.

Theorem 3.9 can be restated in terms of the original network as follows. This result is a special case of the general complementary slackness conditions of linear programming problems.

Theorem 3.10 (Complementary slackness optimality conditions)**.** *A feasible solution x of the MCF problem is optimal if and only if for some potential function π, the following complementary slackness conditions hold for each arc $ij \in A$:*

$$\text{if } c^{\pi}_{ij} > 0, \text{ then } x_{ij} = 0; \tag{3.11}$$

$$\text{if } 0 < x_{ij} < u_{ij}, \text{ then } c^{\pi}_{ij} = 0; \tag{3.12}$$

$$\text{if } c^{\pi}_{ij} < 0, \text{ then } x_{ij} = u_{ij}. \tag{3.13}$$

3.1.7 The dual problem

Since the MCF problem is an LP problem, the corresponding *dual problem* can also be formulated. We associate the variable γ_i with the flow conservation constraint of node $i \in V$ and the variable α_{ij} with the capacity constraint of arc $ij \in A$. The dual MCF problem can be stated in terms of these variables as

$$\max \sum_{i \in V} b_i \gamma_i - \sum_{ij \in A} u_{ij} \alpha_{ij} \tag{3.14}$$

subject to

$$\gamma_i - \gamma_j - \alpha_{ij} \leq c_{ij} \qquad \forall ij \in A, \tag{3.15}$$

$$\alpha_{ij} \geq 0 \qquad \forall ij \in A. \tag{3.16}$$

If we specify the potential of node i as $\pi_i = -\gamma_i$, then (3.15) can be rewritten as

$$\alpha_{ij} \geq -c^{\pi}_{ij} \qquad \forall ij \in A. \tag{3.17}$$

Let us eliminate the variables α_{ij} in the objective function (3.14). The coefficient of α_{ij} is $-u_{ij} \leq 0$, so the minimum feasible value of α_{ij} is to be used to maximize the objective function. Considering (3.16) and (3.17), we can set $\alpha_{ij} = \max\{0, -c^{\pi}_{ij}\}$. That is, the dual MCF problem can be restated as a mathematical programming problem without constraints:

$$\max \; -\sum_{i\in V} b_i\pi_i - \sum_{ij\in A} u_{ij}\max\{0, -c^{\pi}_{ij}\}. \tag{3.18}$$

Note that this transformation reduces the dual problem to finding optimal node potentials with respect to (3.18).

The well-known *weak* and *strong duality theorems* apply to the primal and dual MCF problems, as well as the following theorem that reveals the connection between duality and the previously studied optimality conditions.

Theorem 3.11. *A feasible solution x of the primal MCF problem and an arbitrary solution π of the dual MCF problem are optimal if and only if they together satisfy the complementary slackness optimality conditions.*

That is, the optimal potential functions in terms of Theorems 3.9 and 3.10 are the same as the optimal solutions of the dual problem (3.18). So any MCF algorithm that works with reduced costs can provide the dual solution (node potentials) along with the primal solution, which verify the optimality of each other.

3.1.8 Approximate optimality

In addition to the exact optimality conditions studied so far, the characterization of *approximate optimality* is also of particular importance. Several algorithms rely on the concept of ϵ-optimality, which was originally devised by Bertsekas [29, 30] and, independently, Tardos [228]. For a given $\epsilon \geq 0$, a pseudoflow x is called *ϵ-optimal* if for some potential function π, $c^{\pi}_{ij} \geq -\epsilon$ holds for each arc ij in the residual network D^x. This is a relaxation of the reduced cost optimality conditions defined in Theorem 3.9, and they are equivalent when $\epsilon = 0$.

The following proposition formulates two simple observations related to ϵ-optimality.

Proposition 3.12. *Any pseudoflow is ϵ-optimal if $\epsilon \geq C$. Furthermore, if the arc costs are integers and $\epsilon < 1/n$, then an ϵ-optimal feasible flow is optimal.*

Proof. Assigning $\pi_i = 0$ to each node i, the first statement is obvious. Furthermore, let x denote an ϵ-optimal feasible flow for $\epsilon < 1/n$, let π denote the corresponding potential function, and let W be a directed cycle in D^x. Proposition 2.1 implies that $\sum_{ij\in W} c_{ij} = \sum_{ij\in W} c^{\pi}_{ij}$, which is greater than -1 because $c^{\pi}_{ij} > -1/n$ for every arc ij in D^x. That is, the total cost of any cycle in D^x is nonnegative if the arc costs are integers, so x is optimal because of Theorem 3.8. ■

Note that an ϵ-optimal pseudoflow is also ϵ'-optimal for any $\epsilon' > \epsilon$, so the approximate optimality of a pseudoflow x can be described by the smallest value $\epsilon \geq 0$ for which x is ϵ-optimal, which we denote as $\epsilon(x)$. The following theorem is a direct consequence of the results discussed in Section 2.5.

Theorem 3.13. *For a non-optimal pseudoflow x, $\epsilon(x)$ equals to the negative of the minimum mean cost of a directed cycle in the residual network D^x. For an optimal pseudoflow x, $\epsilon(x) = 0$.*

3.1.9 Algorithms

The MCF problem and related algorithms have a rich history spanning over 60 years. Researchers first studied a classic special case, the transportation problem [136, 147], which we discussed in Section 3.1.2. Dantzig [72] was the first to completely solve the transportation problem by specializing his famous LP algorithm, the simplex method. Later, he also applied this approach directly to the MCF problem and developed a method that is known as the *network simplex* algorithm [73].

Ford and Fulkerson devised most of the fundamental concepts and results of network flow theory in the 1950s. They introduced the well-known augmenting path algorithm for solving the maximum flow problem [94], and they also developed the first combinatorial algorithms for the transportation problem [95, 96] by generalizing Kuhn's remarkable Hungarian Method [161]. Later, they also proposed a similar primal–dual algorithm for the MCF problem [97]. The results of Ford and Fulkerson were summarized in the report [98], which presents a detailed study of various network flow models, optimality conditions, and algorithms.

In the next few years, several other methods were also proposed, namely, the *successive shortest path* algorithm [51, 87, 141, 142, 233], the *out-of-kilter* algorithm [105, 179, 249], and the *cycle-canceling* algorithm [156]. These methods, however, do not ensure polynomial running time. Zadeh conducted worst-case studies [250, 251] and showed MCF problem instances for which all algorithms known by that time perform an exponential number of iterations. That is, both theoretical and practical expectations motivated further research on developing more efficient algorithms.

Edmonds and Karp introduced the scaling technique and developed the first weakly polynomial-time algorithm, the *capacity-scaling* method [88]. The problem of finding a strongly polynomial algorithm, however, remained a challenging open question of high interest for a long time. Finally, Éva Tardos was the first to develop such an algorithm [228], which was followed by various methods with improved running time bounds. The existence of strongly polynomial algorithms distinguishes the MCF problem from the general LP problem, for which only weakly polynomial algorithms are known.

Besides theoretical aspects, efficient implementation and experimental evaluation of MCF algorithms have also been objects of intensive research for decades. The *network simplex* algorithm became quite popular when spanning tree labeling techniques were developed to improve its practical performance [22, 115, 144, 223]. Efficient implementations of the network simplex algorithm [132, 154] served as benchmarks for several years. Later, different implementations of the *relaxation* algorithm [32, 33, 34] and the *cost-scaling* algorithm [50, 118, 120, 125, 129] also turned out to be highly efficient. Furthermore, polynomial-time versions of the primal and dual network simplex methods were also developed [18, 130, 187, 190, 192, 195, 229, 230]. Apart from these algorithms, other approaches were also applied to solve the MCF problem, for example, *interior-point* methods [24, 197, 198, 207, 208, 209]. Over the years, numerous computational studies have been conducted to compare the performance of various MCF algorithms, for example, [2, 3, 6, 35, 36, 37, 115, 118, 120].

Significant contributions to this area were achieved related to the First DIMACS Implementation Challenge [80, 143]. This workshop was devoted especially to experimental work in the field of network flows and matchings. The participants developed a standard for storing instances of the MCF problem, as well as different random network generators, which have been widely used in many studies since then.

Even in recent years, novel theoretical achievements were published. For instance, Brunsch et al. [44, 45] and Cornelissen and Manthey [68] applied the smoothed analysis model to different MCF algorithms to obtain better explanation of their practical performance, which often differs from what is suggested by worst-case time complexity. Furthermore, Becker et al. [23] developed a new dual-ascent method, which can be considered as a generalization of the successive shortest path algorithm.

Table 3.2 presents a survey of important MCF algorithms and complexity bounds using the notations defined in Table 3.1. Several MCF algorithms rely on finding shortest paths (with nonnegative arc costs) or maximum flows; SP(...) and MF(...) denote the running time bounds of corresponding algorithms, respectively. Detailed discussions, historical notes, and references related to MCF algorithms can be found, for example, in the books [13, 123, 159, 214] and in the articles [2, 3, 213, 217].

n	Number of nodes in the graph.
m	Number of arcs in the graph.
U	Maximum of supply values and arc capacities (assumed to be integers).
C	Maximum of arc costs (assumed to be integers).
$\mathrm{SP}(n, m, K)$	Running time of any algorithm solving the single-source shortest path problem in a directed graph with n nodes, m arcs, and nonnegative integer arc costs, each at most K. Dijkstra's algorithm with Fibonacci heaps [102] runs in $O(m + n \log n)$ time, but improved weakly polynomial bounds also exist (see Section 2.3).
$\mathrm{MF}(n, m, U)$	Running time of any algorithm solving the maximum flow problem in a directed graph with n nodes, m arcs, and integer capacities, each at most U. Orlin [191] proved that the bound $O(nm)$ holds, but other polynomial bounds can also be achieved (see Section 2.6).

Table 3.1: Notations used in running time bounds of MCF algorithms.

	$O(n^4CU)$	Ford and Fulkerson (1958) [97]
	$O(m^3U)$	Yakovleva (1959) [249], Minty (1960) [179], Fulkerson (1961) [105]
	$O(n^2mU)$	Jewell (1958) [142], Busacker and Gowen (1960) [51], Iri (1960) [141] *successive shortest path*
	$O(nm^2CU)$	Klein (1967) [156] *cycle-canceling*
$*$	$O(nU \cdot \mathrm{SP}(n, m, nC))$	Edmonds and Karp (1970) [87, 88], Tomizawa (1971) [233] *successive shortest path using potentials*
$*$	$O(m \log U \cdot \mathrm{SP}(n, m, nC))$	Edmonds and Karp (1972) [88] *capacity-scaling*
	$O(n \log C \cdot \mathrm{MF}(n, m, U))$	Röck (1980) [210], Bland and Jensen (1985) [36, 37]
	$O(m^2 \log n \cdot \mathrm{MF}(n, m, U))$	Tardos (1985) [228]
	$O(m^2 \log n(m + n \log n))$	Orlin (1984) [187], Fujishige (1986) [103]
	$O(n^2 \log n(m + n \log n))$	Galil and Tardos (1986) [108, 109]
	$O(n^2m \log(nC))$	Goldberg and Tarjan (1987) [125, 129] *cost-scaling, generic version*
	$O(n^3 \log(nC))$	Goldberg and Tarjan (1987) [125, 129], Bertsekas and Eckstein (1988) [31] *cost-scaling*
	$O(n^{5/3}m^{2/3} \log(nC))$	Goldberg and Tarjan (1987) [125, 129] *cost-scaling with blocking flows*
	$O(nm \log n \log(nC))$	Goldberg and Tarjan (1987) [125, 129] *cost-scaling with dynamic trees*
	$O(n^2m^2 \min\{\log(nC), m \log n\})$	Goldberg and Tarjan (1988) [127, 128] *minimum-mean cycle-canceling*
	$O(n^2m \min\{\log(nC), m \log n\})$	Goldberg and Tarjan (1988) [127, 128] *cancel-and-tighten*
		Orlin (1997) [190] *primal network simplex*
	$O(nm \log n \min\{\log(nC), m \log n\})$	Goldberg and Tarjan (1988) [127, 128] *cancel-and-tighten with dynamic trees*
		Tarjan (1997) [230] *primal network simplex with dynamic trees*
$*$	$O(m \log n \cdot \mathrm{SP}(n, m, nC))$	Orlin (1988) [188, 189] *enhanced capacity-scaling*
		Vygen (2000) [238, 239]
$*$	$O(nm \log \log U \log(nC))$	Ahuja, Goldberg, Orlin, and Tarjan (1988) [11, 12] *double scaling*
$*$	$O((m \min\{n, \sqrt{mU}\} + mU \log(mU)) \log(nC))$	Gabow and Tarjan (1989) [107]
	$O(n^3 \min\{\log(nC), m \log n\})$	Goldberg and Tarjan (1990) [129] *generalized cost-scaling*
$*$	$O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$	Goldberg and Tarjan (1990) [129] *generalized cost-scaling with dynamic trees*
	$O(m(m + n \log n) \min\{\log(nU), m \log n\})$	Orlin, Plotkin, and Tardos (1993) [192]
		Sokkalingam, Ahuja, and Orlin (2000) [222]
	$O(nm \log n(m + n \log n))$	Armstrong and Jin (1997) [18]

Table 3.2: Survey of MCF algorithms and complexity bounds. We use the notations defined in Table 3.1 and assume integer input data in the case of time bounds involving U or C. $*$ indicates an asymptotically best complexity bound.

3.2 Implemented algorithms

This section introduces seven different algorithms that the author implemented for solving the MCF problem. These implementations are based on well-known methods, which are thoroughly studied in the literature. Our main contribution is the efficient implementation of these algorithms, combining previously known heuristics with new ideas and improvements. We focused on practical efficiency and applicability, so we did not consider complex data structures that could improve the theoretical time bounds of the algorithms, but are reported to have poor performance in practice (for example, Fibonacci heaps and dynamic trees).

All implementations discussed in this section are part of the LEMON library. They are open-source and can be used under permissive license terms.

3.2.1 Simple cycle-canceling (SCC)

A basic method for solving the MCF problem is the cycle-canceling algorithm, which has several different variants. We first discuss the general method and then a simple implementation, which we call *simple cycle-canceling* (SCC). We also implemented two strongly polynomial cycle-canceling algorithms, which are discussed in the following subsections.

The cycle-canceling method was originally proposed by Klein [156]. It applies a general primal approach based on the negative cycle optimality conditions (see Theorem 3.8). A feasible solution x is first established, which can be carried out by solving a maximum flow problem (see Section 3.1.5). After that, the algorithm throughout maintains feasibility and gradually decreases the total flow cost. At each iteration, a directed cycle of negative cost is found in the current residual network D^x, and this cycle is canceled by augmenting the maximum possible amount of flow along its arcs. When D^x contains no negative-cost cycle, the algorithm terminates with an optimal solution found, according to Theorem 3.8. The general version of this algorithm is outlined in Figure 3.1.

```
procedure CYCLE-CANCELING
    establish a feasible flow $x$ (exit if not found)
    while $D^x$ contains a negative-cost cycle do
        find a negative-cost cycle $W$ in $D^x$
        $\delta \leftarrow \min\{r^x_{ij} : ij \in W\}$
        augment $\delta$ units of flow on the arcs of $W$
        update $x$ and $D^x$
    end while
end procedure
```

Figure 3.1: General cycle-canceling algorithm.

Being a primal method, this algorithm has an important property that it can be started with any feasible solution, and a better one is always available during the computations. This property enables early termination with an approximate result if necessary.

We assumed integer input data for the MCF problem. Under this assumption, if the problem has a feasible solution, then an integer-valued feasible solution can be found using various maximum flow algorithms [13, 67, 159, 214]. Starting with such a feasible solution, the cycle-canceling algorithm finds an integer-valued optimal solution because flow values are computed using only additions and subtractions [13, 214]. Similarly, all other algorithms discussed later also have this property, so we only consider integer-valued flows in the followings.

Researchers have developed numerous variants of the cycle-canceling algorithm by applying different methods for cycle selection [19, 127, 128, 222, 243]. These algorithms have quite different theoretical and practical behavior; some of them run in polynomial or even strongly polynomial time, although the general method is not polynomial. Moreover, the primal network simplex algorithm (see Section 3.2.7) can also be viewed as a particular variant of the cycle-canceling method.

The SCC algorithm is a simple cycle-canceling implementation that identifies negative cycles using the Bellman–Ford algorithm with successively increased limit on the number of iterations. It is well-known that the Bellman–Ford algorithm is capable of detecting a negative-cost directed cycle after performing n iterations or detecting that such a cycle does not exist [13, 67, 159, 214]. In most cases, however, it is not required to perform n iterations. If negative cycles exist in the graph, some of them typically appear in the subgraph identified by the predecessor pointers of the nodes after much less iterations. Unfortunately, we do not know the sufficient limit for the number of iterations in advance and searching for such negative cycles at an intermediate step of the algorithm is relatively slow. Therefore, our implementation performs these checks after a successively increased number of iterations of the Bellman–Ford algorithm. According to our tests, it is practical to search for negative cycles after executing $\lfloor 2 \cdot 1.5^k \rfloor$ iterations for each $k \geq 0$, until this limit reaches n. It also turned out to be beneficial to cancel all node-disjoint negative cycles that can be found at once when Bellman–Ford algorithm is stopped.

Since the total flow cost is decreased at each iteration, and mCU is a trivial upper bound on it, the SCC algorithm performs $O(mCU)$ iterations. Each iteration runs in $O(nm)$ time, so the worst-case time complexity of the method is $O(nm^2CU)$. For establishing an initial feasible flow, we use a push-relabel algorithm implemented in LEMON, which runs in $O(n^2\sqrt{m})$ time. Therefore, this step does not increase the total complexity of the SCC algorithm (we assumed the graph to be weakly connected, so $m = \Omega(n)$).

3.2.2 Minimum-mean cycle-canceling (MMCC)

Goldberg and Tarjan [127, 128] proposed a famous special variant of the cycle-canceling method, the *minimum-mean cycle-canceling* (MMCC) algorithm. It cancels a cycle of minimum mean cost in the residual network at each iteration. This approach yields the simplest strongly polynomial MCF algorithm; it performs $O(nm^2 \log n)$ iterations for arbitrary real-valued arc costs and $O(nm \log(nC))$ iterations for integer arc costs. Despite the algorithm's simplicity and elegance, the proof of these bounds is rather involved [13, 113, 123, 127, 128, 159, 214].

Recall from Section 3.1.8 that the approximate optimality of a feasible solution can be described by the minimum mean cost of a directed cycle in the residual network. This property justifies the selection rule of the MMCC algorithm. Furthermore, if we formulate the maximum flow problem as an MCF problem by adding an arc of cost -1 and sufficiently large capacity from the target node to the source node (see Section 3.1.2), then the MMCC algorithm reduces to the Edmonds–Karp algorithm [81, 88] (i.e., the shortest augmenting path algorithm).

Several algorithms exist for finding a minimum-mean directed cycle in a graph (see Section 2.5). The best strongly polynomial bound is $O(nm)$. Therefore, the overall complexity of the MMCC algorithm is $O(n^2m^2 \min\{\log(nC), m \log n\})$ for the MCF problem with integer data. Despite its theoretical significance, however, the MMCC algorithm is rather slow in practice, even if an efficient algorithm is used for finding minimum-mean cycles.

Finding minimum-mean cycles

We implemented three known algorithms for finding minimum-mean cycles: Karp's original algorithm [148]; an improved version of it that is due to Hartmann and Orlin [135]; and Howard's algorithm [74, 138]. The first two methods run in strongly polynomial time $O(nm)$. In contrast, Howard's algorithm is not known to be polynomial, but it is one of the fastest methods in practice [74, 75, 114].

Howard's algorithm gradually approximates the optimal solution by performing linear-time iterations. Relatively few iterations are typically sufficient to find a minimum-mean cycle, but no polynomial upper bound is known. Therefore, we devised a combined method in order to achieve good performance in practice while keeping the strongly polynomial time bound. Howard's algorithm is run with an explicit limit on the number of iterations. If this limit is reached without finding the optimal solution, we stop Howard's algorithm and execute the Hartmann–Orlin algorithm instead. This limit is set to n, so the total time complexity of this combined method is still $O(nm)$. In our experiments, however, the iteration limit was never reached (in the case of non-trivial graphs). That is, the combined method is practically identical to Howard's algorithm but with a worst-case running time $O(nm)$.

3.2.3 Cancel-and-tighten (CAT)

The *cancel-and-tighten* (CAT) algorithm is also devised by Goldberg and Tarjan [127, 128] as an improved version of the MMCC algorithm. The previous two cycle-canceling algorithms are pure primal methods in a sense that they do not consider the dual solution at all. In contrast, the CAT algorithm maintains node potentials to make the detection of negative residual cycles easier and faster. Since the reduced cost of a cycle is the same as its original cost (see Proposition 2.1), the algorithm can work with reduced costs with respect to the current node potentials. A residual arc is called *admissible* if its reduced cost is negative. The key idea of the algorithm is to cancel cycles that consist entirely of admissible arcs.

Unlike the MMCC algorithm, the CAT algorithm explicitly utilizes the concept of ϵ-optimality: it proceeds by ensuring ϵ-optimality of the current solution for successively smaller values of $\epsilon \geq 0$ until the solution becomes optimal. At every iteration, two main steps are performed. In the *cancel* step, directed cycles consisting of admissible arcs are canceled until no such cycle exists. In the *tighten* step, the node potentials are modified in order to introduce new admissible arcs and to decrease ϵ to at most $(1-1/n)$ times its former value.

The cancel step is the dominant part of the computation. We implemented it using a straightforward procedure based on depth-first search of the admissible arcs. This implementation has a worst-case complexity of $O(nm)$ as canceling a cycle takes $O(n)$ time, and at most $O(m)$ cycles can be successively canceled without modifying the node potentials. Using dynamic tree data structures [220], this step could be carried out in $O(m \log n)$ time [128], but we did not implement this variant.

The tighten step can be performed in $O(m)$ time based on a topological ordering of the nodes with respect to the admissible arcs [128]. However, this method does not ensure strongly polynomial bound on the number of iterations. Therefore, after every k iterations, this step is carried out in a stricter way: ϵ is set to its smallest possible value using a minimum-mean cycle computation (see Theorem 3.13). Node potentials are also recomputed to correspond to this new value of ϵ. Goldberg and Tarjan suggested to use $k = n$ to preserve the amortized running time $O(m)$ of the tighten step, but we use $k = \lfloor\sqrt{n}\rfloor$ since it turned out to be more efficient in practice. This way, the amortized running time of the tighten step becomes $O(m\sqrt{n})$, but it is still less than the $O(nm)$ time of our implementation of the cancel step. The minimum-mean cycle computations are performed using the same combined method that is applied in the MMCC algorithm.

Goldberg and Tarjan proved that $O(n \min\{\log(nC), m \log n\})$ iterations are performed by the CAT algorithm for the MCF problem with integer data [128], so our implementation runs in $O(n^2 m \min\{\log(nC), m \log n\})$ time. In practice, the CAT algorithm is usually much faster than both SCC and MMCC, but it is less efficient than other methods (e.g., cost-scaling and network simplex).

3.2.4 Successive shortest path (SSP)

The *successive shortest path* (SSP) method embodies another important approach for solving the MCF problem. It is a dual algorithm that maintains an optimal pseudoflow together with corresponding node potentials and attempts to achieve feasibility by successively augmenting flow along shortest paths in the residual network. In this sense, the SSP algorithm can be viewed as a generalization of the well-known augmenting path algorithms for the maximum flow problem [81, 88, 98].

The initial versions of the SSP method were devised independently by Jewell [142], Iri [141], and Busacker and Gowen [51]. Later, Edmonds and Karp [87, 88] and, independently, Tomizawa [233] suggested the usage of node potentials to ensure nonnegative arc costs throughout the algorithm, which greatly improves its performance both in theory and in practice.

The SSP algorithm begins with constant zero pseudoflow x and a constant potential function π, and it gradually converts x into a feasible flow while the reduced cost optimality conditions are throughout maintained (see Theorem 3.9). The purpose of using these conditions is twofold: not only do they verify the optimality of the primal and dual solutions, but they also ensure that the reduced cost of each arc is nonnegative in the residual network, so Dijkstra's algorithm can be used for finding shortest paths. At each iteration, we select an excess node s ($e_s^x > 0$) and augment flow from s to a deficit node t ($e_t^x < 0$) along a shortest path (see the corresponding definitions and notations in Section 3.1.4). After that, the node potentials are increased by the computed shortest path distances to preserve the optimality conditions. If the excess of each node becomes zero, a feasible flow is established, which is optimal at once. On the other hand, if we do not find a residual path from a given excess node to any deficit node, then the problem is infeasible.

In our implementation, we apply an important improvement as it is discussed in [13]: at every iteration, Dijkstra's algorithm is terminated once it permanently labels a deficit node t, and the potentials are modified accordingly. This simple improvement usually makes the algorithm much faster in practice. Figure 3.2 outlines the SSP algorithm including this improvement.

```
procedure Successive Shortest Path
    $x_{ij} \leftarrow 0 \quad \forall ij \in A$
    $\pi_i \leftarrow 0 \quad \forall i \in V$
    $S^x \leftarrow \{i \in V : e_i^x > 0\}$
    $T^x \leftarrow \{i \in V : e_i^x < 0\}$
    while $S^x \neq \emptyset$ do
        run Dijkstra's algorithm in $D^x$ from a selected node $s \in S^x$ w.r.t. the reduced costs $c^\pi$
        if no path is found from $s$ to any node $t \in T^x$ then
            exit, no feasible flow exists
        end if
        let $P$ be a shortest path from $s$ to $t \in T^x$, and let $d_i$ denote the distance label of node $i$
        for all $i \in V$ that was permanently labeled by Dijkstra's algorithm do
            $\pi_i \leftarrow \pi_i + d_i - d_t$
        end for
        $\delta \leftarrow \min\{e_s^x, -e_t^x, \min\{r_{ij}^x : ij \in P\}\}$
        augment $\delta$ units of flow on the arcs of $P$
        update $x$, $D^x$, $e^x$, $S^x$, $T^x$
    end while
end procedure
```

Figure 3.2: Successive shortest path algorithm.

The SSP algorithm performs $O(nU)$ path augmentations because it iteratively decreases the total excess of the nodes, which is at most $nU/2$ at the beginning. Our implementation uses the standard binary heap data structure for Dijkstra's algorithm, so an iteration is performed in $O(m \log n)$ time, and the overall worst-case complexity is $O(nmU \log n)$. We also experimented with other heap structures implemented in LEMON (d-ary, Fibonacci, pairing, radix, etc.) [165], but they did not turn out to be faster or more robust in practice.

The representation of the residual network is another important aspect of the implementation. We use an efficient storing scheme that allows fast traversal of the outgoing residual arcs of a node, which is crucial for the shortest path computations, and thereby for the entire algorithm. Instead of working with the original graph D or explicitly representing the residual network D^x (which changes frequently), we store a static auxiliary graph D' that contains all possible forward and backward arcs (i.e., $2m$ arcs), and their residual capacities are also maintained explicitly. Furthermore, we store for each arc an index to its reverse arc (often referred to as *sister arc*). The flow augmentations are carried out by decreasing the residual capacities of the arcs on the path and increasing the residual capacities of the corresponding reverse arcs. Note that using D', only the outgoing arcs of a node have to be traversed during the shortest path searches, and the ones with zero residual capacity are to be skipped. We can, therefore, use consecutive integers to represent the outgoing arcs of a node in D', which makes it possible to traverse them very quickly without iterating over the elements of an array or a linked list. Moreover, consecutive elements are usually accessed from the arrays storing the costs and residual capacities of the arcs, which helps effective caching.

3.2.5 Capacity-scaling (CAS)

The *capacity-scaling* (CAS) algorithm is an improved version of the SSP method. It was developed by Edmonds and Karp [88] as the first weakly polynomial-time MCF algorithm. This algorithm applies scaling, which is a popular technique used in most polynomial-time MCF algorithms and various other graph optimization methods. Scaling algorithms work by solving a sequence of subproblems that gradually closer approximate the parameters of the original problem, in a way that the solution of a subproblem helps to solve the next subproblems. In the case of the MCF problem, these subproblems can be obtained by successively increasing the precision of the capacities or the costs.

The CAS algorithm improves upon the SSP method by performing capacity-scaling phases to ensure that sufficiently large amounts of flow are augmented along the shortest paths, which often reduces the number of iterations. In a Δ-scaling phase, each path augmentation delivers at least Δ units of flow from a node s with $e_s^x \geq \Delta$ to a node t with $e_t^x \leq -\Delta$. When no such augmenting path is found, the value of Δ is halved, and the algorithm proceeds with the next phase. At the end of the phase with $\Delta = 1$, an optimal solution is found.

The shortest path searches are carried out in the so-called Δ*-residual network*, which consists of the arcs with residual capacity of at least Δ. The CAS algorithm maintains the reduced cost optimality conditions only in the Δ-residual network. Each Δ-scaling phase begins with saturating those newly introduced arcs of the current Δ-residual network that have negative reduced costs. This step increases the excess of some nodes, but these requirements are satisfied in the subsequent phases. At the end of the last phase with $\Delta = 1$, the solution becomes both feasible and optimal because the Δ-residual network then coincides with the residual network.

The CAS algorithm is proved to perform $O(m \log U)$ iterations under the additional assumption that a directed path of sufficiently large capacity exists between each pair of nodes. Although this condition can easily be achieved by a simple extension of the underlying network, we decided to avoid this transformation. As a result, more units of excess may remain at the end of a Δ-scaling phase, and the polynomial running time is thereby not guaranteed, but our experiments showed that this version does not perform more path augmentations and runs significantly faster in practice.

Our implementation of the CAS algorithm is based on a slightly modified variant that is due to Orlin [189] and also discussed in [13]. In addition, we uses a scaling factor α other than two, that is, Δ is initially set to $\alpha^{\lfloor \log_\alpha U \rfloor}$ and is divided by α at the end of each phase. We use $\alpha = 4$ by default because it turned out to provide the best overall performance. Furthermore, the practical improvements of the SSP implementation also apply to this algorithm. Our CAS code uses the same representation for the residual network and associated data, and we also use the improvement based on the early termination of Dijkstra's algorithm.

The experimental results presented in Section 3.3 show that the performance of the augmenting path algorithms SSP and CAS greatly depends on the characteristics of the input. On general problem instances, these algorithms are typically slower than the cost-scaling and network simplex codes, but in certain cases, when relatively few path augmentations are sufficient to solve the problem, they turned out to be quite efficient.

3.2.6 Cost-scaling (COS)

The cost-scaling technique for the MCF problem was first proposed independently by Röck [210] and Bland and Jensen [36, 37]. Their algorithms solved the problem by a sequence of $O(n \log C)$ maximum flow computations. Goldberg and Tarjan [125, 129] improved on these methods by utilizing the concept of ϵ-optimality. They developed an MCF algorithm that is a generalization of their famous push-relabel algorithm for the maximum flow problem [124, 126]. We refer to this method as the *cost-scaling* (COS) algorithm, which is actually one of the most efficient MCF algorithms, both in theory and in practice.

The COS algorithm is a primal–dual method that applies a successive approximation scheme by scaling upon the costs. It produces ϵ-optimal primal–dual solution pairs for gradually decreased values of $\epsilon \geq 0$. Initially, $\epsilon = C$, and each phase applies a *refine* procedure to transform the current ϵ-optimal solution into an (ϵ/α)-optimal solution for a given scaling factor $\alpha > 1$. When $\epsilon < 1/n$, the algorithm terminates and Proposition 3.12 implies that an optimal flow is found.

The refine procedure takes an ϵ-optimal primal–dual solution pair x and π as input and improves the approximation as follows. First, it converts x to an optimal pseudoflow by saturating each residual arc whose current reduced cost is negative. The value of ϵ is then divided by α, and the pseudoflow x is gradually transformed into a feasible flow again, preserving ϵ-optimality for the new value of ϵ. This is achieved by performing a sequence of local *push* and *relabel* operations (similarly to the push-relabel algorithm for the maximum flow problem).

Given a pseudoflow x and a potential function π, a residual arc ij is called *admissible* if $c^\pi_{ij} < 0$. Furthermore, we refer to the excess nodes as *active* nodes in the context of this algorithm. A basic operation of the COS algorithm selects an active node i ($e^x_i > 0$) and either pushes flow on an admissible residual arc ij or, if no such arc exists, relabels node i. Relabeling a node means that its potential is decreased by the largest possible amount that does not violate the ϵ-optimality conditions. This operation introduces new admissible outgoing arcs at node i and thereby allows subsequent push operations to carry the remaining excess of the node. The refine procedure terminates when no active node remains in the network, that is, an ϵ-optimal feasible solution is obtained (since the basic operations preserve ϵ-optimality). Figure 3.3 presents the pseudocode of the COS algorithm.

```
procedure COST-SCALING
    establish a feasible flow $x$ (exit if not found)
    $\pi_i \leftarrow 0 \quad \forall i \in V$
    $\epsilon \leftarrow C$
    while $\epsilon \geq 1/n$ do
        $\epsilon \leftarrow$ REFINE($\epsilon$)
    end while
end procedure

procedure REFINE($\epsilon$)
    for all admissible arcs $ij \in A^x$ do
        PUSH($ij, r^x_{ij}$)
    end for
    $\epsilon \leftarrow \epsilon/\alpha$
    while there exists an active node do
        select an active node $i \in V$
        if there exists an admissible arc $ij \in A^x$ then
            PUSH($ij, \min\{e^x_i, r^x_{ij}\}$)
        else
            RELABEL($i$)
        end if
    end while
    return $\epsilon$
end procedure

procedure PUSH($ij, \delta$)
    send $\delta$ units of flow on the arc $ij$
    update $x$, $D^x$, $e^x$
end procedure

procedure RELABEL($i$)
    $\pi_i \leftarrow \pi_i - \epsilon - \min\{c^\pi_{ij} : ij \in A^x\}$
end procedure
```

Figure 3.3: Cost-scaling algorithm.

The generic version of the refine procedure performs $O(n^2)$ relabel operations and $O(n^2m)$ push operations, so it runs in $O(n^2m)$ time [13, 129]. In addition, the number of ϵ-scaling phases is $O(\log(nC))$ because ϵ is initially set to C and divided by α in each phase until it decreases below $1/n$. Consequently, the generic COS algorithm runs in weakly polynomial time $O(n^2m\log(nC))$.

Goldberg and Tarjan [129] also devised special versions of improved running time bounds by applying particular selection rules of the basic operations and using complex data structures such as dynamic trees (see Table 3.2 in Section 3.1.9). They also developed a generalized framework to obtain a strongly polynomial bound on the number of ϵ-scaling phases by utilizing the same idea that is used in the CAT algorithm (see Section 3.2.3). The best complexity bound they achieved is $O(nm \log(n^2/m) \min\{\log(nC), m \log n\})$. Moreover, the COS algorithm turned out to be quite efficient in practice, and several complex heuristics were also developed to improve its performance [50, 118, 120].

We implemented three variants of the COS algorithm that perform the refine procedure rather differently. We first discuss the standard push-relabel implementation, and then describe the differences for the other two variants.

Push-relabel method

This implementation is based on the generic version of the COS algorithm, so it performs local push and relabel operations. However, we applied various effective heuristics and other improvements according to the ideas published in the articles [50, 118, 120, 129] In fact, most of these improvements are analogous to similar techniques devised for the push-relabel maximum flow algorithm (see, e.g., [59]).

The bottleneck operation in the COS algorithm is searching for admissible arcs for the basic operations. Therefore, we apply the same representation of the residual network as for the SSP and CAS algorithms (see Section 3.2.4). Furthermore, we maintain a current arc pointer for each node and continue the search for an admissible outgoing arc from this current arc every time. If an admissible arc is found, we perform a push operation, and when we reach the last outgoing arc of an active node without finding an admissible arc, the node is relabeled and its current arc is set to the first outgoing arc again. (Note that the definition of the basic operations imply that only the relabeling of node i can introduce a new admissible arc outgoing from node i.)

The optimal value of α depends on the heuristics applied in the algorithm and on the problem instance as well. Our experiments showed that its optimal value is usually between 8 and 24, and the differences are typically moderate, so we use $\alpha = 16$ by default.

A practical improvement of the COS algorithm targets the issue that internal computations are to be performed with non-integer values of ϵ and non-integer node potentials (even if all input data are integers). To overcome this drawback, we multiple arc costs by αn for a fixed integer scaling factor $\alpha \geq 2$, and ϵ is also scaled accordingly. Initially, ϵ is set to $\alpha^{\lceil \log_\alpha(\alpha n C) \rceil}$ and divided by α in each phase until $\epsilon = 1$. Thereby, it is ensured that all computations operate solely on integer numbers, which improves the performance of the operations and avoids potential numerical issues.

The strategy for selecting an active node for the next basic operation is also important. The number of active nodes is typically small, so it is beneficial to explicitly keep track of them. A particular variant of the COS algorithm, known as the "wave" implementation, selects the active nodes according to a topological ordering with respect to the admissible arcs. This choice is proved to yield an $O(n^3)$-time implementation of the refine procedure, instead of $O(n^2 m)$. However, our experiments showed that the simple FIFO selection rule using a queue data structure usually results in less basic operations and better performance in practice, which is in accordance with [50] and [118].

We also remark that dynamic trees [220] can be used in the COS algorithm to perform a number of push operations at once, which improves the theoretical running time [129]. However, they are not practical because of their computational overhead and because the *push-look-ahead* heuristic (see below) effectively decreases the number of push operations.

Heuristics

In addition to the improvements discussed above, we also applied three sophisticated heuristics out of the four proposed in [118, 120].

The *potential refinement* (or *price refinement*) heuristic is based on the observation that an ϵ-scaling phase often produces a solution that is not only ϵ-optimal, but also ϵ'-optimal for some $\epsilon' < \epsilon$ (or even optimal). Therefore, an additional step is introduced at the beginning of each phase to check if the current solution is already (ϵ/α)-optimal. This step attempts to adjust the potentials to satisfy the (ϵ/α)-optimality conditions, but without modifying the flow. If it succeeds, the refine procedure is skipped and the next phase begins. We implemented this potential refinement heuristic using an $O(nm)$-time scaling shortest path algorithm [117] as suggested in [118]. Our results also verified that this heuristic substantially improves the overall performance of the algorithm in most cases.

Another variant of this heuristic performs a minimum-mean cycle computation at the beginning of each phase to determine the smallest ϵ for which the current flow is ϵ-optimal and computes corresponding node potentials. This variant may skip more than one phase at once, and it also ensures a strongly polynomial bound on the number of scaling phases (similarly to the CAT algorithm, see Section 3.2.3). According to our tests, however, an efficient implementation of the former variant of the potential refinement heuristic performs significantly better in practice, so we use that one in our final implementation. This conclusion contradicts the experiments in [50].

The *global update* heuristic performs relabel operations on several nodes at once by iteratively applying the following *set-relabel* operation. Let $T \subset V$ denote a set of nodes such that all deficit nodes are in T, but at least one active node is in $V \setminus T$. If no admissible arc enters T, then the potential of the nodes in T can be increased uniformly by ϵ without violating the ϵ-optimality conditions. It is shown that the time complexity of the COS algorithm remains unchanged if the global update heuristic is applied only after every $\Omega(n)$ relabel operations. This heuristic turned out to make the entire algorithm much more efficient on some problem classes (up to 3-5 times faster), although it does not help too much or even slightly worsens the performance on other instances.

The *push-look-ahead* heuristic attempts to avoid pushing flow from node i to node j when a subsequent push operation is likely to send this amount of flow back to node i. To achieve this, only a limited amount of flow is allowed to be pushed into a node j (the sum of its deficit and the residual capacities of its admissible outgoing arcs). However, this idea requires the extension of the relabel operation to those nodes at which this limitation was applied regardless of their current excess values. This heuristic is rather effective in practice: it significantly decreases the number of push operations in most cases.

Another heuristic, the *speculative arc fixing*, is also proposed by Goldberg [118, 120]. Although this heuristic would most likely improve the performance of our implementation as well, we did not implement it because it is rather involved and seems to be sensitive to parameter settings.

Augment-relabel method

This is a special variant of the COS algorithm that performs path augmentations instead of local push operations, but heavily uses relabel operations to find augmenting paths. At each step of the refine procedure, this method selects an active node s and performs a depth-first search on the admissible arcs to find an augmenting path to a deficit node t. Whenever the search process steps back from a node i, then i is relabeled.

When such an admissible path is found, flow augmentation can be performed in multiple ways. We can either push the same amount of flow on each arc of the path or push the maximum possible amount of flow on each arc. According to our experiments, the latter variant performs slightly better, so it is applied in our implementation.

Note that this procedure of path search and flow augmentation corresponds to a particular sequence of local push and relabel operations. However, the actual push operations are carried out in a delayed and more guided manner, in aware of an entire admissible path to a deficit node. This concept helps to avoid such problems for which the push-look-ahead heuristic is devised (see above), but a lot of work may be required to find augmenting paths, especially if they are long.

Since this implementation can be viewed as a special version of the generic COS method, the same theoretical running time bound applies to it, as well as most of the practical improvements. We used the same data representation, improvements, and heuristics as for the push-relabel variant, except for the push-look-ahead heuristic.

Partial augment-relabel method

The third implementation of the COS algorithm is an intermediate approach between the other two variants. It is based on the partial augment-relabel algorithm proposed by Goldberg [119] as a new variant of the push-relabel algorithm for solving the maximum flow problem. As this method turned out to be highly efficient and robust in practice, Goldberg also suggested the utilization of the idea in the MCF context, but he did not investigate it. According to the author's knowledge, our implementation of the COS algorithm is the first to apply this technique to the MCF problem.

The partial augment-relabel implementation of the COS algorithm is quite similar to the augment-relabel variant, but it limits the length of the augmenting paths. The path search process is stopped either if a deficit node is reached or if the length of the path reaches a given parameter $k \geq 1$. Our code uses $k = 4$ by default, just like Goldberg's maximum flow algorithm, as it results in a quite robust algorithm in the MCF context as well. Note that the push-relabel and augment-relabel variants are special cases of this approach with $k = 1$ and $k = n - 1$, respectively.

Apart from the length limitation for the augmenting paths, this variant is exactly the same as the augment-relabel method. However, the partial augment-relabel technique attains a good compromise between the former two approaches and turned out to be superior to them. Unless otherwise stated, we refer to this implementation as COS in the followings.

Section 3.3 provides experimental results for the COS algorithm and its different variants. The standard push-relabel implementation and especially the partial augment-relabel method using the described heuristics and improvements are highly efficient and robust. In contrast, the augment-relabel variant is often significantly slower.

3.2.7 Network simplex (NS)

The primal *network simplex* (NS) algorithm is one of the most popular methods for solving the MCF problem. It is a specialized version of the well-known LP simplex method that exploits the network structure of the MCF problem and performs the basic operations directly on the graph representation. The LP variables correspond to the arcs of the graph, and the LP bases correspond to spanning trees.

The NS algorithm was devised by Dantzig, the inventor of the LP simplex method. He first solved the uncapacitated transportation problem using this approach [72] and later generalized the bounded variable simplex method to directly solve the MCF problem [73]. Although the generic version of the algorithm does not run in polynomial time, it turned out to be rather effective in practice. Therefore, subsequent research has focused on efficient implementation of the NS algorithm [22, 41, 115, 132, 154, 168, 223]. Furthermore, researchers also developed particular variants of both the primal and dual network simplex methods that run in polynomial time [18, 130, 187, 190, 192, 195, 229, 230]. For a summary of the corresponding running time bounds, see Table 3.2 in Section 3.1.9. Detailed discussion of the NS method considering both theoretical and practical aspects can be found, for example, in [13] and [153].

Here we discuss the basic concepts and methods related to the primal NS algorithm as well as the important improvements and details of our implementation.

Spanning tree solutions

The NS algorithm is based on the concept of *spanning tree solutions*. Such a solution is a feasible solution of the MCF problem that can be represented by a partitioning of the arc set A into three subsets (T, L, U) such that the arcs in T form an undirected spanning tree of the network, and the flow value of each *non-tree arc* in L and U is restricted to either its lower bound (zero) or its upper bound (the capacity of the arc), respectively. The flow on the *tree arcs* also satisfy the nonnegativity and capacity constraints, but they are not restricted to any of the bounds.

It is proved that if an instance of the MCF problem has an optimal solution, then it also has an optimal spanning tree solution, which can be found by successively transforming a spanning tree solution to another one. These spanning tree solutions actually correspond to the LP basic feasible solutions of the problem. This observation allows us to implement the LP simplex method in a way that all operations are performed directly on the network, without maintaining the simplex tableau, which makes this approach very efficient.

Primal network simplex algorithm

The general LP simplex method maintains a basic feasible solution and gradually improves its objective function value by small transformations called *pivots*. Accordingly, the primal

NS algorithm throughout maintains a spanning tree solution of the MCF problem and successively decreases the total cost of the flow until it becomes optimal. Node potentials are also maintained such that the reduced cost of each arc in the spanning tree is zero. At each iteration, a non-tree arc violating its complementary slackness optimality condition (see Theorem 3.10) is added to the current spanning tree, which uniquely determines a residual cycle of negative cost. This cycle is canceled by augmenting flow along it, and a tree arc corresponding to a saturated residual arc is removed from the tree. The data structure representing the spanning tree is then updated, and the node potentials are also adjusted to preserve the property that the reduced cost of each tree arc is zero. If there are multiple saturated residual arcs, then an appropriate rule is applied to select the leaving arc (see later). This whole operation transforming a spanning tree solution to another one is called a pivot. If no suitable entering arc can be found, the current flow is optimal, and the algorithm terminates. The top-level description of the algorithm is shown in Figure 3.4.

```
procedure Network Simplex
    determine a feasible spanning tree solution (exit if not found)
    while a non-tree arc violates its optimality condition do
        select an entering arc $ij$ violating its optimality condition
        add $ij$ to the spanning tree and augment flow along the corresponding cycle
        determine the leaving arc
        update the spanning tree solution, flow values, and potentials
    end while
end procedure
```

Figure 3.4: Main steps of the primal network simplex algorithm.

In fact, the NS algorithm can also be viewed as a particular variant of the cycle-canceling algorithm (see Section 3.2.1). Due to the sophisticated method of maintaining spanning tree solutions, however, a negative cycle can be found and canceled much faster (in $O(m)$ time).

Strongly feasible spanning tree solutions

Similarly to the LP simplex method, *degeneracy* is a critical issue for the NS algorithm. If the spanning tree contains an arc whose flow value equals to zero or the capacity of the arc, then a pivot step may detect a cycle of zero residual capacity. Such *degenerate* pivots only modify the spanning tree, but the flow itself remains unchanged. Consequently, several consecutive pivots may not actually decrease the flow cost (called *stalling*), or, which is even worse, the same spanning tree solution may occur multiple times, and hence the algorithm may not terminate in a finite number of iterations (called *cycling*). Experiments with certain classes of large-scale MCF problems showed that more than 90% of the pivots may be degenerate.

A simple and popular technique to overcome these issues is based on the concept of *strongly feasible spanning tree solutions*, which was introduced by Cunningham [69] and, independently, Barr, Glover, and Klingman [21]. A spanning tree solution is called strongly feasible if a positive amount of flow can be sent from each node to a designated root node of the spanning tree along the tree path without violating the nonnegativity and capacity constraints. Using a simple rule for selecting the leaving arcs, the NS algorithm can throughout maintain a strongly feasible spanning tree. This technique is proved to ensure that the algorithm terminates in a finite number of iterations [13]. Furthermore, it substantially decreases the number of degenerate pivots in practice and thereby makes the algorithm faster. However, an exponential number of degenerate pivots may still occur, which can be avoided by pivot strategies that allow an arc to enter the basis only periodically [70, 131].

It can be shown, using a perturbation technique, that the NS algorithm maintaining a strongly feasible spanning tree solution performs $O(nmCU)$ pivots for the MCF problem with integer data [13, 15]. An entering arc can be found in $O(m)$ time, and the spanning tree structure can be updated in $O(n)$ time using an appropriate labeling technique. Therefore, a single pivot takes $O(m)$ time, and the total running time of the NS algorithm is $O(nm^2CU)$. The practical performance of the method, however, is much better than what is suggested by this worst-case bound. Recently, Cornelissen and Manthey applied the smoothed analysis technique to explain this difference [68].

Spanning tree data structures

The representation of spanning tree solutions is essential to implement the NS algorithm efficiently. Several storage schemes have been developed for this purpose along with efficient methods for updating them during the pivot operations [22, 115, 144, 154, 223].

We implemented two spanning tree storage schemes for the NS algorithm. The first one, which is usually referred to as the ATI (*Augmented Threaded Index*) method, represents a spanning tree as follows. The tree has a designated root node, and three numbers are stored for each node: the depth of the node in the tree, the index of its parent node, and a so-called thread index that is used to define a depth-first traversal of the spanning tree. It is a quite popular method, which is discussed in detail in [153] and [13].

The ATI technique has an improved version, which is due to Barr, Glover, and Klingman [22] and is referred to as the XTI (*eXtended Threaded Index*) method. The XTI scheme replaces the depth index by two values for each node: the number of successors of the node in the tree and the last successor of the node according to the traversal defined by the thread indexes. This modification allows faster update process since a tree alteration of a single pivot usually modifies the depth of several nodes in subtrees that are moved from a position to another one, while the set of successors is typically modified only for much fewer nodes.

We implemented the XTI scheme with an additional improvement that a reverse thread index is also stored for each node to represent the depth-first traversal as a doubly-linked list. This modification turned out to substantially improve the performance of the update process. In fact, the inventors of the XTI technique also discussed this extension [22], but they did not apply it in order to reduce the memory requirements of the representation.

Although the XTI labeling method is not as widely known and popular as the simpler ATI method, our experiments showed that it is much more efficient on all problem instances, so the final version of our code implements only the XTI scheme. Other storage techniques, for example, the so-called API and XPI methods, are also often applied, but we did not experiment with them.

Graph representation

Another interesting aspect of the NS algorithm is that we need not traverse the incident arcs of nodes throughout the algorithm, although such steps are typically used in most graph algorithms. Therefore, we used a quite simple and unusual graph representation to implement the NS algorithm. The nodes and arcs are represented by consecutive integers, and we store the source and target nodes for each arc (in arrays), but we do not keep track of the outgoing and incoming arcs of a node at all.

Initialization

The NS algorithm requires an initial spanning tree solution to start with. It is possible to transform any feasible solution x to a spanning tree solution x' such that the total cost of x' is less than or equal to the total cost of x. The required spanning tree indexes can also be computed by a depth-first traversal of the tree arcs. However, artificial initialization is more common in practice. It means that the underlying network is extended with an artificial root node and additional arcs connecting this node and the original nodes in a way that a strongly feasible spanning tree solution can easily be constructed.

Let s denote the artificial root node added to the graph. For each original node i, we add a new arc is if $b_i \geq 0$ and an arc si otherwise. We can set the capacity of each new arc to nU and its cost to nC. In this extended network, a strongly feasible spanning tree solution x can be constructed easily. For each original arc ij, let $x_{ij} = 0$, and for each original node i, let $x_{is} = b_i$ if $b_i \geq 0$ and let $x_{si} = -b_i$ otherwise. The initialization of the tree indexes and node potentials is also straightforward in this case. Furthermore, note that an optimal solution in the extended network does not send flow on artificial arcs because of their large costs unless the original problem is infeasible.

We experimented with both ways of initialization, and it turned out that the artificial method usually provides better overall performance mainly because of two reasons. First, the artificial spanning tree solution can be constructed easily and quickly. Furthermore, it allows efficient tree update for the first several pivots due to the rather small depth of the tree. Therefore, we decided to use only this variant in our final implementation.

Furthermore, we also developed an additional heuristic based on this artificial initialization procedure to make the first few pivots even faster. The initialization of node potentials implies that an arc ij is eligible for the first pivot if and only if $b_i \geq 0$ and $b_j < 0$. After such an arc enters the basis, new arcs incident to its source node may also become eligible. Therefore, we collect several arcs using a partial traversal of the graph starting from the demand nodes (using the reverse orientation of each arc). Then the first few pivots selects entering arcs from this list. Our computational results showed that this idea can make initial pivots substantially faster.

Pivot rules

One of the most important aspects of the NS algorithm is the selection of entering arcs for the pivot operations, which is usually referred to as *pivot rule* or *pricing strategy*. The applied method affects the "goodness" of the entering arcs and thereby the number of iterations as well as the time required for selecting an entering arc, which is a dominant part of the entire algorithm. Consequently, applying different strategies, we can obtain several variants of the NS algorithm with quite different theoretical and empirical behavior [41, 70, 131, 132, 153, 168].

Since a non-tree arc ij has a flow value fixed either at zero or at its capacity, it allows flow augmentation only in one direction. If the reduced cost of the residual arc associated with this direction is negative, the arc ij can be selected to enter the tree. In this case, a negative-cost residual cycle is formed by this arc and the unique tree path connecting nodes i and j because tree arcs have zero reduced costs. Formally, a non-tree arc ij is called *eligible* if it violates the optimality conditions, that is, either $x_{ij} = 0$ and $c^{\pi}_{ij} < 0$ or $x_{ij} = u_{ij}$ and $c^{\pi}_{ij} > 0$ with respect to the current potential function π. We refer to $|c^{\pi}_{ij}|$ as the *violation* of arc ij. When a pivot rule does not find an eligible arc, then the solution is optimal, and the algorithm terminates.

We implemented five pivot rules, which are briefly discussed in the followings. Four of them are widely known rules [13, 153], while the fifth one was developed by us as an improved version of the *candidate list* rule.

Best eligible arc. This is one of the simplest and earliest strategies, which was proposed by Dantzig and is also known as *Dantzig's pivot rule*. At each iteration, this method selects an eligible arc with the maximum violation to enter the tree. This means that a residual cycle having the most negative cost is selected to be canceled, which causes the maximum decrease of the objective function value per unit flow augmentation. Computational studies showed that this selection rule usually results in fewer iterations than other strategies. However, it has to check all non-tree arcs and recompute their reduced costs at each iteration, which is rather slow and yields a poor overall performance.

First eligible arc. Another straightforward idea is to select the first eligible arc at each iteration. The practical implementation of this rule examines the arcs cyclically by starting each search process at the position where the previous eligible arc is found. If we reach the end of the arc list, the examination is continued from the beginning of the list again. In contrast with the previous rule, this one rapidly finds an entering arc at each iteration, but these arcs typically have relatively small violation, and hence a lot of iterations are usually required.

Block search. Since the previous two rules do not perform well in practice, several other strategies have been devised to attain effective compromise between them. The *block search pivot rule* was proposed by Grigoriadis [132]. This method cyclically examines certain subsets of the arcs (called blocks) and selects the best eligible candidate among these arcs at each iteration. The search process starts from the position of the previous entering arc and checks a fixed number of arcs by recomputing their reduced costs. If this block contains eligible arcs, then the one with the maximum violation is selected to enter the basis. Otherwise, subsequent blocks are evaluated until an eligible arc is found.

The block size B is an important parameter of this method. In fact, the previous two rules are special cases of this one with $B = m$ and $B = 1$, respectively. Previous studies suggest to set B proportionally to the number of arcs, for example, between 1% and 10% [132, 153]. However, our experiments showed that a much more robust implementation can be achieved if we set $B = \lfloor \alpha\sqrt{m} \rfloor$ for a small value of α. In our implementation, $B = \lfloor \sqrt{m} \rfloor$ is used, which resulted in the best overall performance.

Similarly to the *first eligible rule*, this strategy also has the inherent advantage that an arc is allowed to enter the basis only periodically, which usually decreases the number of degenerate pivots in practice [70, 131].

Candidate list. This is another well-known pivot rule, which was proposed by Mulvey [183]. It occasionally builds a list of eligible arcs and selects the best arcs among these candidates at subsequent iterations. A so-called *major* iteration examines the arcs cyclically to build a list containing at most L eligible arcs. This list is then used by at most K subsequent iterations to select an arc of maximum violation among the candidates. If an arc becomes non-eligible, it is removed from the list. When K minor iterations are performed or the list becomes empty, another major iteration takes place.

This method is similar to the *block search rule*, but it considers the same subset of the arcs in several consecutive pivots, while the previous rule considers only the best arc of a block and then advances to the next block. We obtained the best average running time using $L = \lfloor \sqrt{m}/4 \rfloor$ and $K = \lfloor L/10 \rfloor$.

Altering candidate list. This strategy was developed by us and can be viewed as an improved version of the previous rule. It also maintains a candidate list, but it attempts to extend this list at each iteration, and only the several best candidates are kept for the next one. At least one arc block of size B is examined at every iteration to extend the list with new eligible arcs. After that, an arc of maximum violation is selected to enter the basis, while the list is partially sorted and truncated to contain at most H of the best candidate arcs. According to our measurements, this method is highly efficient using $B = \lfloor \sqrt{m} \rfloor$ and $H = \lfloor B/100 \rfloor$.

Our experimental results showed that the block search and the altering candidate list pivot rules are the most efficient on different classes of problem instances. Since the block search rule is simpler and turned out to be slightly more robust, it is the default pivot strategy, and we refer to this variant as NS in the followings.

Section 3.3 presents experimental results comparing the different pivot rules as well as comparing the NS algorithm with other methods. Our NS implementation turned out to be the most efficient method for the majority of test cases, although it is typically outperformed by the cost-scaling codes on very large and sparse networks.

3.3 Experimental results

The practical performance of complex optimization algorithms is usually quite different from what is suggested by their theoretical running time. Thorough empirical evaluation is, therefore, essential. This section provides a comprehensive experimental study of the presented MCF implementations and also compares them with other efficient solvers. This is a slightly reduced version of the analysis published in [2].

The contribution of this study is twofold. First, a wide variety of algorithms are compared with each other using the same benchmark suite, which provides insight about their relative performance on different classes of networks. Second, larger problem instances are also considered than in previous studies of MCF algorithms, which turned out to be important to draw appropriate conclusions related to the asymptotic behavior of these algorithms. The author is not aware of any previous work that provides an experimental analysis of comparable extent.

We first describe the problem families used for the experiments. Then we present a comparison of the algorithms implemented by the author in the LEMON library. After that, we focus on the most efficient ones among these codes and compare them with other publicly available implementations, including the most popular ones.

3.3.1 Test setup

Our test suite comprises numerous networks of various sizes and characteristics. Most of these networks were generated using standard random generators NETGEN, GRIDGEN, GOTO, and GRIDGRAPH, which are available as source codes on the website of the 1st DIMACS Implementation Challenge [80]. We used these generators with similar parameter settings as previous works (e.g., [34, 35, 50, 99, 118, 120, 168]), but we created larger networks as well. Other problem families were also generated based on either real-life road networks or maximum flow problems arising in computer vision applications. All instances involve solely integer data.

This collection was introduced in [2, 3] and can be accessed at [8]. Later, other independent studies also used some of these problem families, for example, [23, 200].

The presented experiments were conducted on a machine with AMD Opteron Dual Core 2.2 GHz CPU and 16 GB RAM (1 MB cache), running openSUSE 11.4 operating system. All codes were compiled with GCC 4.5.3 using -O3 optimization flag.

NETGEN networks

NETGEN is a classic generator developed by Klingman et al. [157]. It produces random instances of the MCF problem and other network optimization problems. In fact, NETGEN is known to create easy MCF instances.

We generated NETGEN networks as follows. Arc capacities and costs were selected uniformly at random from the ranges [1..1000] and [1..10000], respectively. The number of supply nodes and the number of demand nodes were both set to $\sqrt{n}$ (rounded to integer), and the average amount of supply per supply node was set to 1000. We defined two problem families that differ in the density of the networks since this parameter turned out to have the most significant impact on the performance of different algorithms.

- **NETGEN-8.** Sparse networks; $m = 8n$.
- **NETGEN-SR.** Dense networks; $m \approx n\sqrt{n}$.

GRIDGEN networks

GRIDGEN is a random generator that produces grid-like networks. It was written by Lee and Orlin [163]. We used the same parameters for GRIDGEN families as for the corresponding NETGEN families. Furthermore, we set the width of the grid to $\sqrt{n}$ (rounded to integer) for each instance. In fact, the shape of the grid did not turn out to be an important parameter, as GRIDGEN selects the supply and demand nodes uniformly at random.

- **GRIDGEN-8.** Sparse networks; $m = 8n$.
- **GRIDGEN-SR.** Dense networks; $m \approx n\sqrt{n}$.

GOTO networks

Another standard generator for the MCF problem is GOTO ("*Grid On Torus*"), which was developed by Goldberg [120]. It generates instances that are known to be rather hard, using a grid layout on the surface of a torus. Each GOTO instance has one supply node and one demand node, and the supply value is adjusted according to the arc capacities.

Similarly to the previous generators, we generated two GOTO families. The maximum arc capacity and cost were set to 1000 and 10000, respectively.

- **GOTO-8.** Sparse networks; $m = 8n$.
- **GOTO-SR.** Dense networks; $m \approx n\sqrt{n}$.

GRIDGRAPH networks

GRIDGRAPH generator was written by Resende and Veiga [208]. It also produces grid networks similarly to GRIDGEN but using a stricter scheme. A GRIDGRAPH network consists of transshipment nodes forming a grid of W rows and L columns, together with a single source node s and a single sink node t. Arcs go from s to the nodes of the first column; from the nodes of the last column to t; and from each transshipment node (w, l) to nodes $(w + 1, l)$ and $(w, l + 1)$, except for the last row and column, respectively. The arc capacities and costs are set uniformly at random within a specified range.

The shape of the grid is the most important property in the case of a GRIDGRAPH instance, so we specified two families based on grids of different shapes. The maximum arc capacity and cost were set to 1000 and 10000, respectively.

- **GRID-WIDE.** Wide grids; $L = 16$ and W increases.
- **GRID-LONG.** Long grids; $W = 16$ and L increases.

ROAD networks

Special MCF instances based on real-world road networks were also included in our experiments. To generate such instances, we used the TIGER/Line road network files of several states of the USA. These data files are available on the website of the 9th DIMACS Implementation Challenge [79].

We selected seven states having road networks of increasing size (namely, DC, DE, NH, NV, WI, FL, and TX) and generated MCF problem instances as follows. The original undirected graphs were converted to directed graphs by replacing each edge with two oppositely directed arcs. The cost of an arc was set to the travel time on the corresponding road section. We selected K supply nodes and K demand nodes randomly, where $K = \lfloor \sqrt{n}/10 \rfloor$. Then the supply-demand values were determined by a maximum flow computation to maximize the total supply with respect to the fixed arc capacities and fixed set of supply and demand nodes.

We generated two problem families with different arc capacity settings.

- **ROAD-PATHS.** The capacity of each arc is 1. That is, a specified number of arc-disjoint directed paths are to be found from supply nodes to demand nodes with minimum total cost.
- **ROAD-FLOW.** The capacity of an arc is set to 40, 60, 80, or 100 according to the category of the corresponding road section.

VISION networks

Our test suite also contains MCF instances based on large-scale maximum flow problems arising in computer vision applications. The corresponding data files were made available by the Computer Vision Research Group at the University of Western Ontario [63] for benchmarking maximum flow algorithms (see, e.g., [119]).

We used some of the segmentation instances related to medical image analysis, which are defined on three-dimensional grid networks. Those variants were selected in which the underlying graphs are 6-connected and the maximum arc capacity is 100 (the `bone_sub*_n6c100` files). We converted these networks to minimum-cost maximum flow instances using different arc cost functions. The original networks also contain arcs of zero capacity, but we skipped these arcs during the transformation and thereby did not preserve the exact 6-connectivity.

- **VISION-RND.** The arc costs are selected uniformly at random from the range [1..100].
- **VISION-PROP.** The cost of an arc is approximately proportional to its capacity: $c_{ij} = \lfloor \alpha_{ij} u_{ij} \rfloor$, where α_{ij} is a random factor selected uniformly from the range $[0.9, 1.1)$.
- **VISION-INV.** The cost of an arc is approximately inversely proportional to its capacity: $c_{ij} = \lfloor \alpha_{ij} K / u_{ij} \rfloor$, where $K = 1000$ and α_{ij} is a random factor selected uniformly from the range $[0.9, 1.1)$.

For all problem families introduced above, we generated five instances of each network size using different random seeds. Throughout this section, we always report the average running time over such five instances.

3.3.2 Comparison of the implemented algorithms

Here we present benchmark results for the algorithms discussed in Section 3.2. These experiments were conducted using version 1.3 of LEMON.

Figure 3.5 and Table 3.3 show the results on NETGEN problems, while Figure 3.6 and Table 3.4 show the results on GOTO instances. The first part of each table presents running times in seconds, while the second part reports normalized times. The best running time is highlighted for each problem size, and a "–" sign denotes the cases when the explicit time limit of one hour was exceeded. The charts display running time in seconds as a function of the number of nodes in the network. Logarithmic scale is used for both axes. Each time result is the average running time on five different problem instances, which were generated with exactly the same settings but with different random seeds.

In accordance with previous studies, we found that GOTO instances are substantially harder than NETGEN instances, and the relative performance of the algorithms also turned out to be rather different on them. On the other hand, the results on GRIDGEN networks are quite similar to those obtained on the corresponding NETGEN families, so we omit GRIDGEN results here.

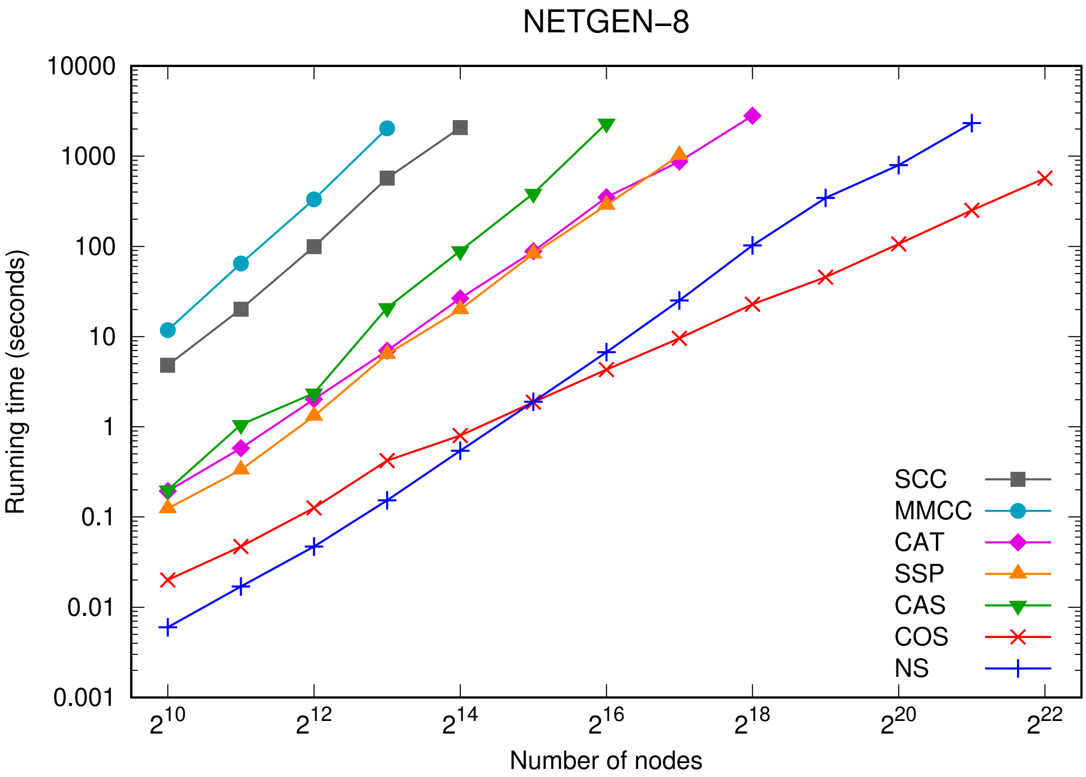
Running time (seconds)

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
NETGEN-8	2^{10}	8	4.82	11.78	0.19	0.12	0.20	0.02	**0.01**
	2^{13}	8	571.14	2032.79	6.99	6.38	20.82	0.42	**0.15**
	2^{16}	8	–	–	349.54	286.58	2316.07	**4.30**	6.71
	2^{19}	8	–	–	–	–	–	**45.57**	345.16
	2^{22}	8	–	–	–	–	–	**569.68**	–
NETGEN-SR	2^{10}	32	33.18	82.47	0.71	0.31	1.51	0.06	**0.02**
	2^{12}	64	1442.21	–	15.85	6.75	76.22	0.80	**0.21**
	2^{14}	128	–	–	320.55	132.74	–	8.00	**3.47**
	2^{16}	256	–	–	–	2895.84	–	103.72	**63.22**

Normalized time

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
NETGEN-8	2^{10}	8	482.00	1178.00	19.00	12.00	20.00	2.00	**1.00**
	2^{13}	8	3807.60	13551.93	46.60	42.53	138.80	2.80	**1.00**
	2^{16}	8	–	–	81.29	66.65	538.62	**1.00**	1.56
	2^{19}	8	–	–	–	–	–	**1.00**	7.57
	2^{22}	8	–	–	–	–	–	**1.00**	–
NETGEN-SR	2^{10}	32	1659.00	4123.50	35.50	15.50	75.50	3.00	**1.00**
	2^{12}	64	6867.67	–	75.48	32.14	362.95	3.81	**1.00**
	2^{14}	128	–	–	92.38	38.25	–	2.31	**1.00**
	2^{16}	256	–	–	–	45.81	–	1.64	**1.00**

Table 3.3: Comparison of the implemented algorithms on NETGEN networks.



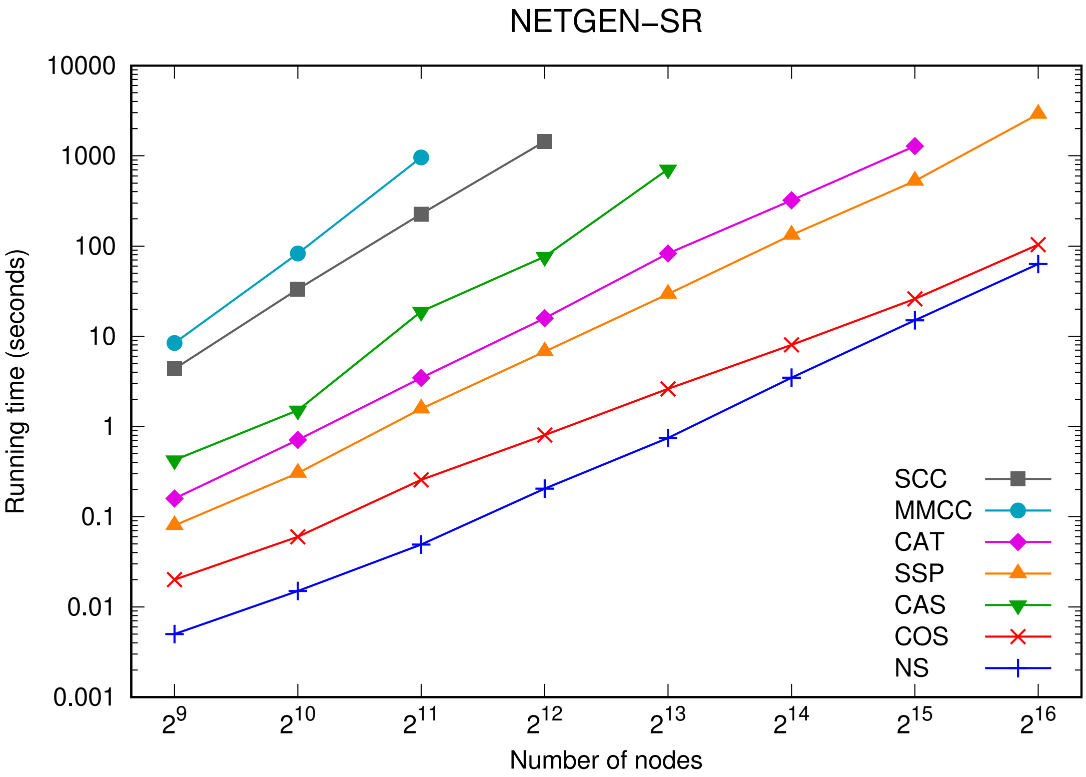
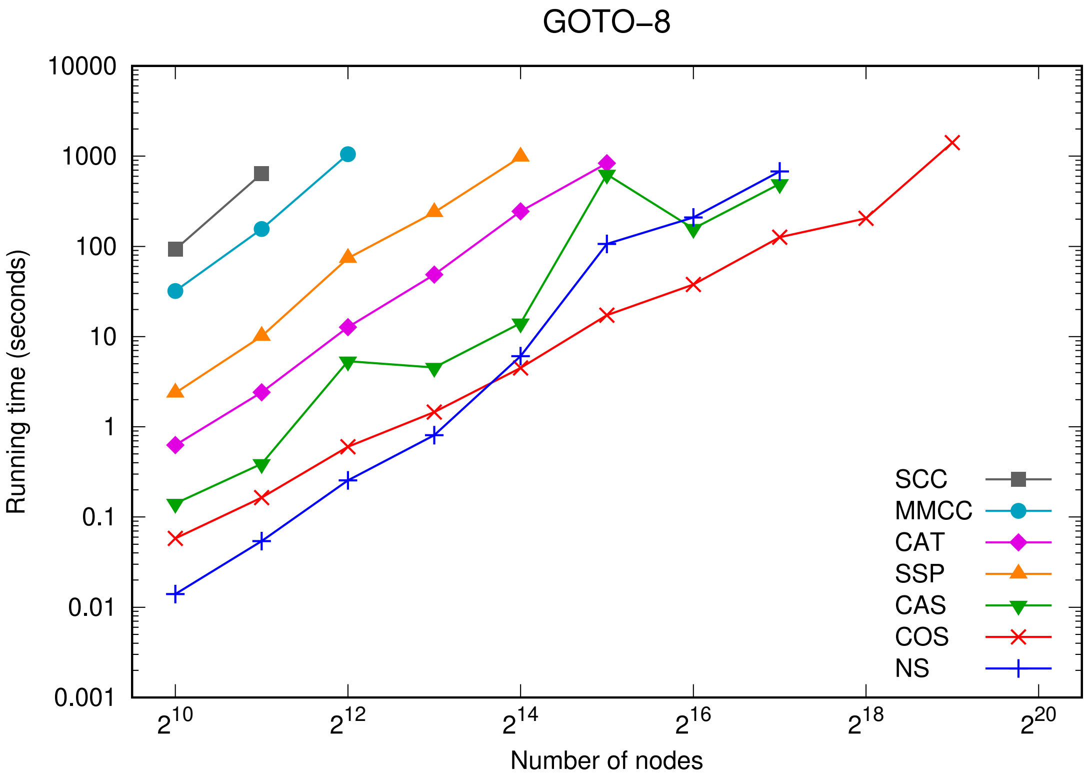


Figure 3.5: Comparison of the implemented algorithms on NETGEN networks.



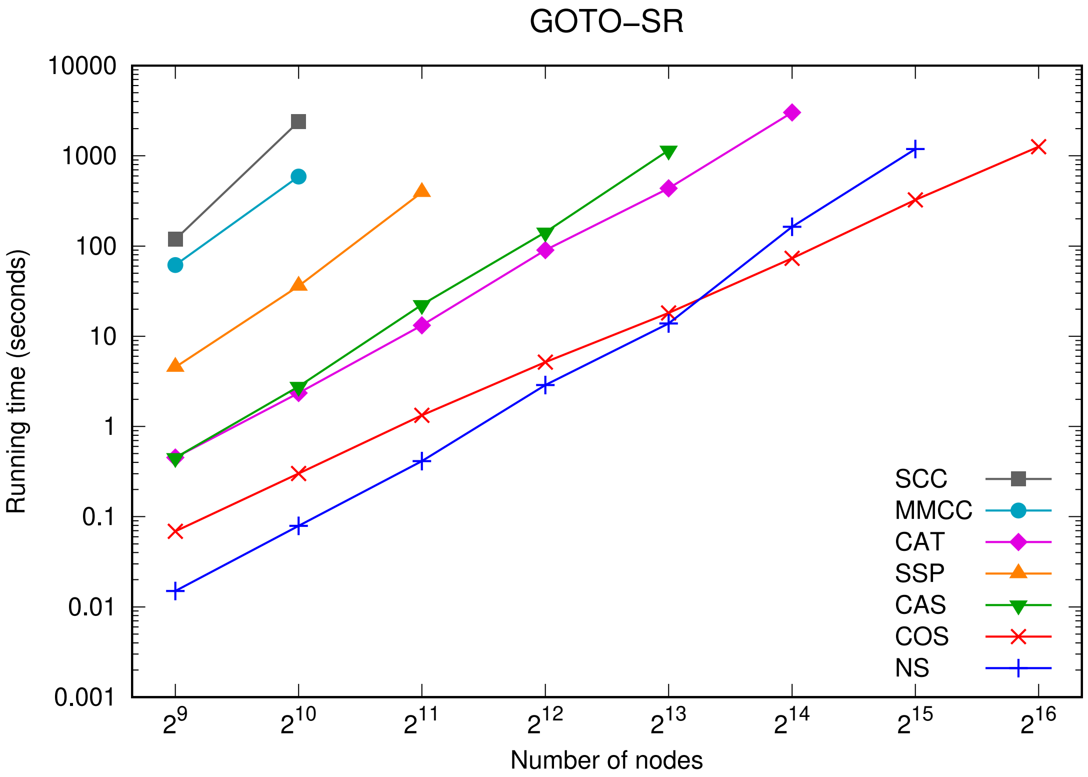


Figure 3.6: Comparison of the implemented algorithms on GOTO networks.

Running time (seconds)

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
GOTO-8	2^{10}	8	92.57	32.03	0.63	2.37	0.14	0.06	**0.01**
	2^{13}	8	–	–	48.65	237.60	4.53	1.45	**0.81**
	2^{16}	8	–	–	–	–	157.15	**37.87**	208.61
	2^{19}	8	–	–	–	–	–	**1413.75**	–
GOTO-SR	2^{10}	32	2389.67	588.14	2.35	35.92	2.77	0.30	**0.08**
	2^{12}	64	–	–	89.97	–	142.50	5.17	**2.88**
	2^{14}	128	–	–	3024.20	–	–	**72.98**	163.26
	2^{16}	256	–	–	–	–	–	**1260.33**	–

Normalized time

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
GOTO-8	2^{10}	8	9257.00	3203.00	63.00	237.00	14.00	6.00	**1.00**
	2^{13}	8	–	–	60.06	293.33	5.59	1.79	**1.00**
	2^{16}	8	–	–	–	–	4.15	**1.00**	5.51
	2^{19}	8	–	–	–	–	–	**1.00**	–
GOTO-SR	2^{10}	32	29870.88	7351.75	29.38	449.00	34.63	3.75	**1.00**
	2^{12}	64	–	–	31.24	–	49.48	1.80	**1.00**
	2^{14}	128	–	–	41.44	–	–	**1.00**	2.24
	2^{16}	256	–	–	–	–	–	**1.00**	–

Table 3.4: Comparison of the implemented algorithms on GOTO networks.

According to these experiments, the basic cycle-canceling methods, SCC and MMCC, are orders of magnitude slower than all other algorithms. CAT, which is an advanced cycle-canceling algorithm, is much faster and usually performs similarly to the dual algorithms SSP and CAS. The relative performance of these three methods greatly depends on the characteristics of the problem instance. The COS and NS algorithms are generally the most efficient. COS shows better asymptotic behavior than NS, and hence it is typically faster on the largest networks, while NS outperforms COS on the smaller ones.

Table 3.5 compares the efficiency of LEMON implementations on GRIDGRAPH families. The shape of the underlying grid is an important parameter for these networks as it determines the length of augmenting paths or cycles an algorithm should find. This phenomenon has consistently been observed before, for example, see [34]. On GRID-WIDE networks, NS is by far the fastest and COS is the second, while on GRID-LONG instances, the augmenting path algorithms, SSP and CAS, greatly outperform all other methods.

The benchmark results on ROAD networks are presented in Table 3.6. As one would expect, the SSP algorithm is the fastest on these special instances. On the ROAD-PATHS family, the CAS algorithm works exactly the same as SSP since arc capacities are uniformly set to one, but it is slower than SSP on ROAD-FLOW instances. The COS and NS algorithms perform significantly worse than SSP and CAS, especially in the case of the ROAD-PATHS family, while all the three cycle-canceling algorithms are extremely slow on these networks.

Finally, Figure 3.7 and Table 3.7 summarize the benchmark results on VISION networks. These problem instances turned out to be rather hard. The slowest algorithms,

Running time (seconds)

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
GRID-WIDE	2^{10}	2.04	0.84	1.35	0.11	0.05	**0.01**	0.02	**0.01**
	2^{13}	2.06	72.62	150.15	2.44	4.19	0.97	0.32	**0.03**
	2^{16}	2.06	–	–	54.27	392.94	100.52	9.96	**0.66**
	2^{19}	2.06	–	–	1027.71	–	–	463.73	**12.83**
GRID-LONG	2^{10}	1.95	0.59	0.93	0.13	0.02	**0.01**	**0.01**	**0.01**
	2^{13}	1.94	66.32	41.77	7.36	0.11	**0.05**	0.21	0.08
	2^{16}	1.94	2857.30	1520.42	818.85	0.46	**0.36**	2.38	5.53
	2^{19}	1.94	–	–	–	**2.93**	3.05	22.39	504.82

Normalized time

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
GRID-WIDE	2^{10}	2.04	84.00	135.00	11.00	5.00	**1.00**	2.00	**1.00**
	2^{13}	2.06	2420.67	5005.00	81.33	139.67	32.33	10.67	**1.00**
	2^{16}	2.06	–	–	82.23	595.36	152.30	15.09	**1.00**
	2^{19}	2.06	–	–	80.10	–	–	36.14	**1.00**
GRID-LONG	2^{10}	1.95	59.00	93.00	13.00	2.00	**1.00**	**1.00**	**1.00**
	2^{13}	1.94	1326.40	835.40	147.20	2.20	**1.00**	4.20	1.60
	2^{16}	1.94	7936.94	4223.39	2274.58	1.28	**1.00**	6.61	15.36
	2^{19}	1.94	–	–	–	**1.00**	1.04	7.64	172.29

Table 3.5: Comparison of the implemented algorithms on GRIDGRAPH networks.

Running time (seconds)

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
ROAD-PATHS	9,559	3.11	2.62	396.79	2.77	**0.01**	**0.01**	0.16	0.06
	116,920	2.27	203.66	–	247.86	**0.30**	**0.30**	4.69	3.15
	519,157	2.44	–	–	3318.63	**3.46**	**3.46**	36.13	42.58
	2,073,870	2.49	–	–	–	**19.06**	**19.06**	248.32	506.40
ROAD-FLOW	9,559	3.11	7.71	649.40	3.35	0.04	**0.03**	0.21	0.06
	116,920	2.27	742.47	–	399.39	**1.10**	1.50	8.43	4.65
	519,157	2.44	–	–	–	**11.06**	19.05	54.38	47.70
	2,073,870	2.49	–	–	–	**88.74**	336.75	471.64	1106.19

Normalized time

Problem family	n	m/n	SCC	MMCC	CAT	SSP	CAS	COS	NS
ROAD-PATHS	9,559	3.11	262.00	39679.00	277.00	**1.00**	**1.00**	16.00	6.00
	116,920	2.27	678.87	–	826.20	**1.00**	**1.00**	15.63	10.50
	519,157	2.44	–	–	959.14	**1.00**	**1.00**	10.44	12.31
	2,073,870	2.49	–	–	–	**1.00**	**1.00**	13.03	26.57
ROAD-FLOW	9,559	3.11	257.00	21646.67	111.67	1.33	**1.00**	7.00	2.00
	116,920	2.27	674.97	–	363.08	**1.00**	1.36	7.66	4.23
	519,157	2.44	–	–	–	**1.00**	1.72	4.92	4.31
	2,073,870	2.49	–	–	–	**1.00**	3.79	5.31	12.47

Table 3.6: Comparison of the implemented algorithms on ROAD networks.

SCC and MMCC, were unable to solve any of them within one hour, so they are excluded from these results. Clearly, COS is the most robust algorithm on VISION families, the asymptotic trend of its running time is much better than that of NS. However, CAT, SSP, and CAS are even much slower than NS.

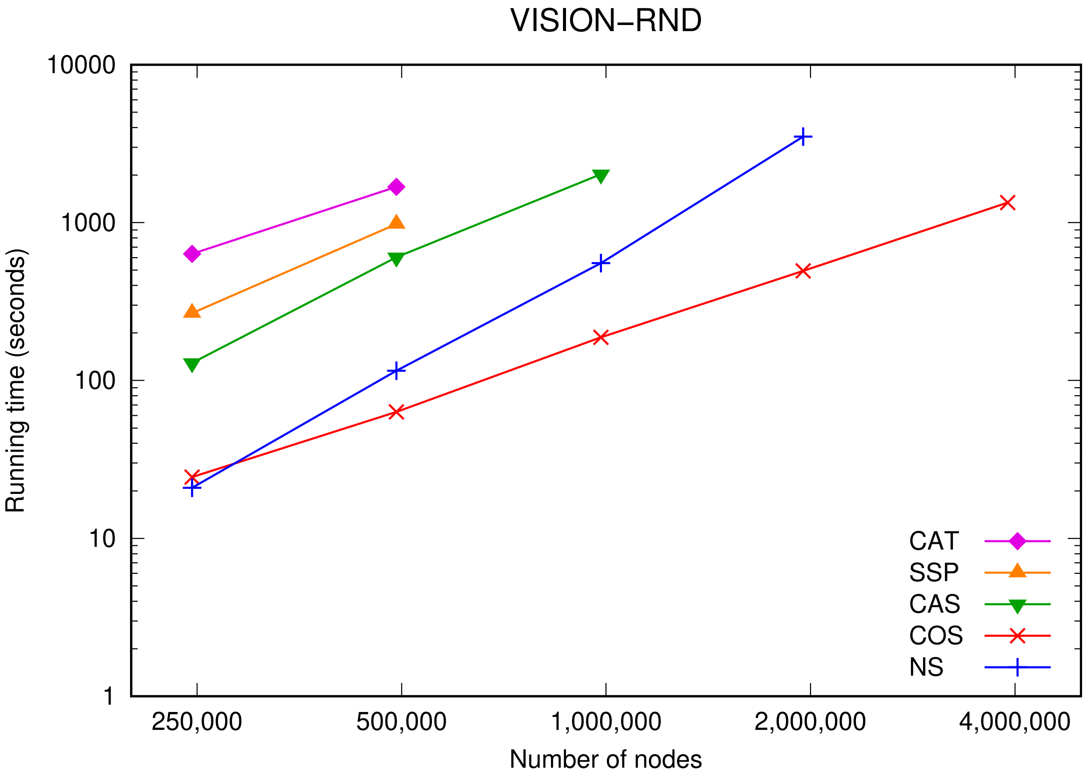


Figure 3.7: Comparison of the implemented algorithms on VISION-RND networks.

According to these results and many additional experiments, we can conclude that COS and NS are generally the most efficient and robust algorithms among the ones that were implemented as part of this research. On particular problem instances, however, the dual-ascent augmenting path algorithms, SSP and CAS, could be significantly faster (GRID-LONG and ROAD networks).

3.3.3 Comparison of algorithm variants

Recall from Sections 3.2.6 and 3.2.7 that we considered different variants of the two most efficient algorithms, COS and NS. These variants were also compared systematically to determine the default options. Here we only present a brief selection of these results.

Table 3.8 compares the variants of the COS algorithm on NETGEN-8 and GOTO-8 networks. The partial augment-relabel (COS-PAR) technique was clearly faster than the other two approaches on all kinds of problem instances. The augment-relabel implementation (COS-AR) performed similarly to the standard push-relabel method (COS-PR) on easy problem instances, such as the NETGEN networks, but it was an order of magnitude slower on harder instances, such as the GOTO networks. These results show that COS-AR

Running time (seconds)

Problem family	n	m/n	CAT	SSP	CAS	COS	NS
VISION-RND	245,762	5.82	633.99	267.18	129.16	24.43	**20.92**
	491,522	5.85	1684.60	979.09	603.44	**63.27**	115.48
	983,042	5.88	–	–	2027.55	**187.69**	554.55
	1,949,698	5.91	–	–	–	**494.24**	3510.13
	3,899,394	5.92	–	–	–	**1341.04**	–
VISION-PROP	245,762	5.82	611.30	500.43	225.14	49.75	**21.17**
	491,522	5.85	1644.80	2049.59	800.09	**111.03**	112.52
	983,042	5.88	–	–	3086.13	**406.98**	458.90
	1,949,698	5.91	–	–	–	**842.25**	2830.07
	3,899,394	5.92	–	–	–	**2426.39**	–
VISION-INV	245,762	5.82	406.30	116.26	41.53	41.29	**14.26**
	491,522	5.85	1137.09	457.24	181.83	97.07	**81.93**
	983,042	5.88	–	2922.38	745.12	**298.76**	304.45
	1,949,698	5.91	–	–	2683.08	**820.11**	2168.45
	3,899,394	5.92	–	–	–	**2201.84**	–

Normalized time

Problem family	n	m/n	CAT	SSP	CAS	COS	NS
VISION-RND	245,762	5.82	30.31	12.77	6.17	1.17	**1.00**
	491,522	5.85	26.63	15.47	9.54	**1.00**	1.83
	983,042	5.88	–	–	10.80	**1.00**	2.95
	1,949,698	5.91	–	–	–	**1.00**	7.10
	3,899,394	5.92	–	–	–	**1.00**	–
VISION-PROP	245,762	5.82	28.88	23.64	10.63	2.35	**1.00**
	491,522	5.85	14.81	18.46	7.21	**1.00**	1.01
	983,042	5.88	–	–	7.58	**1.00**	1.13
	1,949,698	5.91	–	–	–	**1.00**	3.36
	3,899,394	5.92	–	–	–	**1.00**	–
VISION-INV	245,762	5.82	28.49	8.15	2.91	2.90	**1.00**
	491,522	5.85	13.88	5.58	2.22	1.18	**1.00**
	983,042	5.88	–	9.78	2.49	**1.00**	1.02
	1,949,698	5.91	–	–	3.27	**1.00**	2.64
	3,899,394	5.92	–	–	–	**1.00**	–

Table 3.7: Comparison of the implemented algorithms on VISION networks.

is not so robust than the other two methods, which is in accordance with Goldberg's experiments in the maximum flow context [119].

For the NS algorithm, we implemented five pivot rules, which significantly affect the efficiency of the algorithm. Table 3.9 compares the overall performance of these strategies on NETGEN-8 and GOTO-8 families.

These results and our other experiments verified that the block search (NS-BS) and the altering candidate list (NS-ACL) rules are generally the most efficient. On GOTO instances, other rules also performed similarly to these methods, but they were much slower in other cases, for example, on NETGEN instances. Since the NS-BS implementation turned out to be slightly more robust than NS-ACL on harder problem instances, it was selected to be the default pivot strategy.

The best eligible (NS-BE) rule is known to yield the least number of iterations, but its search process is really slow because it checks all non-tree arcs and recomputes their reduced costs at each iteration. As a result, its overall performance is by far the worst among all rules we considered.

Running time (seconds)					
Problem family	n	m/n	COS-PR	COS-AR	COS-PAR
NETGEN-8	2^{10}	8	0.03	**0.02**	**0.02**
	2^{12}	8	0.17	0.14	**0.13**
	2^{14}	8	0.94	1.11	**0.78**
	2^{16}	8	6.37	5.72	**4.24**
	2^{18}	8	35.17	28.00	**22.40**
	2^{20}	8	176.87	179.13	**103.83**
	2^{22}	8	1064.62	901.07	**615.42**
GOTO-8	2^{10}	8	0.10	0.16	**0.05**
	2^{12}	8	0.89	2.48	**0.60**
	2^{14}	8	7.60	33.34	**4.43**
	2^{16}	8	78.55	911.05	**41.27**
	2^{18}	8	342.75	–	**195.36**

Table 3.8: Comparison of different variants of the COS algorithm on NETGEN-8 and GOTO-8 networks. COS-PR: push-relabel method; COS-AR: augment-relabel method; COS-PAR: partial augment-relabel method (default).

Running time (seconds)							
Problem family	n	m/n	NS-BE	NS-FE	NS-BS	NS-CL	NS-ACL
NETGEN-8	2^{10}	8	0.20	**0.01**	**0.01**	**0.01**	**0.01**
	2^{12}	8	3.40	0.14	0.05	0.06	**0.04**
	2^{14}	8	60.47	3.64	0.54	1.02	**0.47**
	2^{16}	8	1285.91	117.99	6.88	27.68	**6.41**
	2^{18}	8	–	–	104.69	808.10	**98.97**
	2^{20}	8	–	–	**799.26**	–	800.36
GOTO-8	2^{10}	8	0.50	**0.01**	**0.01**	**0.01**	0.02
	2^{12}	8	9.59	0.43	**0.25**	**0.25**	0.28
	2^{14}	8	151.92	6.84	6.11	**5.90**	6.16
	2^{16}	8	3024.80	251.48	**202.47**	216.21	220.16

Table 3.9: Comparison of NS pivot rules on NETGEN-8 and GOTO-8 networks. NS-BE, NS-FE, NS-BS, NS-CL, and NS-ACL denote the NS algorithm using best eligible, first eligible, block search (default), candidate list, and altering candidate list pivot rules, respectively.

3.3.4 Comparison with other solvers

The algorithms we implemented in LEMON were also compared with other publicly available MCF solvers, which are briefly introduced here. Some of these solvers, namely CS2, MCFZIB, CPLEX, and RelaxIV, are widely known and have served as benchmarks for a long time (see, e.g., [34, 99, 118, 168]), while the others have rarely been included in previous experimental studies.

The MCFClass project [27] was of great help for us in these experiments. This project features a common and flexible C++ interface for several MCF solvers, which are all considered in this paper. However, MCFClass did not support the latest versions of CS2 and MCFZIB, so we used these two solvers directly.

MCF solvers

CS2. This is an authoritative implementation of the cost-scaling push-relabel algorithm, which is known to be highly efficient and robust. It was written in C language by Goldberg and Cherkassky applying the improvements and heuristics described in [118, 120]. CS2 has been widely used as a benchmark in several studies [99, 118, 168]. It is available for academic research and evaluation purposes free of charge, but commercial use requires a license. We used the latest version, CS2 4.6 [116].

LEDA. This is a comprehensive commercial C++ library providing efficient implementations of various data types and algorithms [17, 177]. It is widely used in many application areas, such as telecommunication, scheduling, traffic planning, integrated circuit design, and computational biology. The `MIN_COST_FLOW()` procedure of the LEDA library implements the cost-scaling push-relabel algorithm similarly to CS2. We used LEDA 5.0 in our experiments.

MCFZIB. Löbel [168] implemented a network simplex code in C language at the Zuse Institute Berlin (ZIB). Its original name is MCF, but we denote this implementation as MCFZIB in order to differentiate it from the problem itself. This solver features both a primal and a dual network simplex implementation, from which the former one is used by default as it tends to be more efficient. It applies a usual data structure for representing the spanning tree solutions along with an improved version of the candidate list pivot rule, which is called *multiple partial pricing*. This implementation was shown to be rather efficient [99, 168]. We used the latest version, MCF 1.3, which is available for academic use free of charge [167].

CPLEX. IBM ILOG CPLEX Optimization Studio [140], usually referred to simply as CPLEX, is a well-known and powerful software suite aimed at large-scale optimization problems. It provides efficient methods for solving different kinds of mathematical programming problems, including linear, mixed integer, and convex quadratic programming. The NETOPT module of CPLEX implements the primal network simplex algorithm for solving the MCF problem. We used this component through a "wrapper class" called MCFCplex, which is provided by the MCFClass project [27]. MCFCplex implements the common interface defined in this project using the CPLEX Callable Library. Although CPLEX is a commercial software, it is available free of charge for academic use and non-commercial research. We used version 12.4, the latest release at the time of writing.

MCFSimplex. This is a recent, open-source software written by Bertolini and Frangioni in C++ language. It is available as part of the MCFClass project [27] and can be used under the flexible LGPL license. This code implements both the primal and dual network simplex algorithms, from which the former one is the default because it is usually faster and more robust. MCFSimplex can also solve separable quadratic MCF problems as well. We abbreviate the primal and dual versions of MCFSimplex as MSim and MSim-D, respectively. According to our experiments, MSim-D turned out to be superior to the dual version of MCFZIB on the majority of test instances, so only the former one is considered in the rest of this paper.

RelaxIV. This is an efficient, authoritative implementation of the relaxation algorithm [32]. The original FORTRAN code was written by Bertsekas and Tseng [33, 34] and is available at [28]. We used a C++ translation of this code, which was made by Frangioni and Gentile and is available as part of the MCFClass project [27]. Similarly to CS2, the RelaxIV solver and its previous versions have been used in experimental studies for a long time [33, 34, 99, 118, 168].

PDNET. This code implements the truncated primal-infeasible dual-feasible interior-point algorithm for solving linear network flow problems. It was written in Fortran and C language by Portugal, Resende, Veiga, Patrício, and Júdice [197, 198]. This solver can be used free of charge, its source code is available at [196]. For more information about interior-point network flow algorithms, see, for example, [24, 207, 209].

All of these codes were compiled with the same compiler and optimization settings as we used for the LEMON implementations (GCC 4.5.3 with -O3 optimization), and they were also executed applying a time limit of one hour. We used all solvers with their default options as we were interested in evaluating their robustness without exploiting the flexibility they provide in parameter settings.

CS2 and LEDA operate on integer numbers by default, just like the LEMON implementations. MCFZIB, MCFSimplex, and RelaxIV support both integer and floating-point input, and the number types they use can be customized. In order to ensure fair comparison, these codes were also compiled using integer types. In contrast, PDNET inherently uses floating-point numbers. Finally, CPLEX most likely operate on floating-point numbers as well, but it does not provide options to change this.

Results

In the following comparisons, we only consider our most efficient implementations, mainly COS and NS, and compare them with the other solvers introduced above. As previously, we always report average running time over five different problem instances of the same size. On the charts, our implementations are indicated with solid lines, and the other solves are indicated with dashed lines. Furthermore, in order to avoid overwhelmed charts, the results of LEDA and MSim are not depicted on them. LEDA usually performs worse than or similarly to the other two cost-scaling methods, COS and CS2; while MSim has much in common with MCFZIB, and thereby their performance is also similar. However, the tables report the results for these two solvers as well.

Figure 3.8 and Table 3.10 compare the different methods on NETGEN instances, and Table 3.11 compares them on GRIDGEN instances. Our NS implementation is the most efficient on most of these networks, but it is consistently outperformed by the cost-scaling algorithms and RelaxIV on the largest sparse graphs. The other primal network simplex codes, MCFZIB, CPLEX, and MSim, are significantly slower than NS. The dual network simplex implementation, MSim-D is competitive with the primal versions on these families, especially in the case of large networks.

Running time (seconds)

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
NETGEN-8	2^{10}	8	0.02	**0.01**	0.02	0.03	0.02	0.02	**0.01**	0.04	**0.01**	0.74
	2^{13}	8	0.42	**0.15**	0.31	0.47	0.57	1.07	0.43	1.83	0.26	13.51
	2^{16}	8	4.30	6.71	4.28	8.34	18.29	116.11	23.04	19.45	**3.54**	336.62
	2^{19}	8	45.57	345.16	48.08	*error*	740.96	–	1040.51	304.64	**41.85**	–
	2^{22}	8	569.68	–	547.95	*error*	–	–	–	–	**431.21**	–
NETGEN-SR	2^{10}	32	0.06	**0.02**	0.05	0.08	0.05	0.09	0.05	0.27	0.03	3.82
	2^{12}	64	0.80	**0.21**	0.58	1.42	0.68	1.46	0.70	3.71	1.09	62.73
	2^{14}	128	8.00	**3.47**	8.01	19.84	21.11	30.45	15.50	32.25	4.54	1250.90
	2^{16}	256	103.72	63.22	99.44	266.39	637.93	967.10	815.55	261.94	**41.40**	–

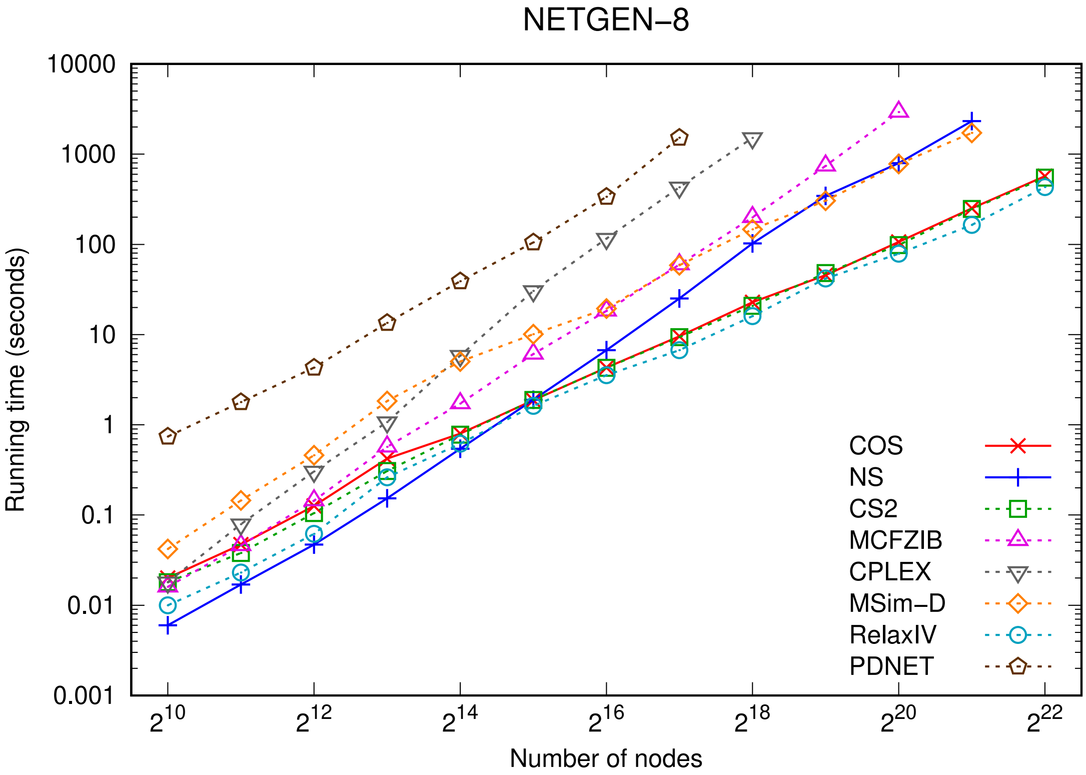
Normalized time

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
NETGEN-8	2^{10}	8	2.00	**1.00**	2.00	3.00	2.00	2.00	**1.00**	4.00	**1.00**	74.00
	2^{13}	8	2.80	**1.00**	2.07	3.13	3.80	7.13	2.87	12.20	1.73	90.07
	2^{16}	8	1.21	1.90	1.21	2.36	5.17	32.80	6.51	5.49	**1.00**	95.09
	2^{19}	8	1.09	8.25	1.15	*error*	17.71	–	24.86	7.28	**1.00**	–
	2^{22}	8	1.32	–	1.27	*error*	–	–	–	–	**1.00**	–
NETGEN-SR	2^{10}	32	3.00	**1.00**	2.50	4.00	2.50	4.50	2.50	13.50	1.50	191.00
	2^{12}	64	3.81	**1.00**	2.76	6.76	3.24	6.95	3.33	17.67	5.19	298.71
	2^{14}	128	2.31	**1.00**	2.31	5.72	6.08	8.78	4.47	9.29	1.31	360.49
	2^{16}	256	2.51	1.53	2.40	6.43	15.41	23.36	19.70	6.33	**1.00**	–

Table 3.10: Comparison of different MCF solvers on NETGEN networks.

Our COS code performs similarly to or slightly slower than CS2 on these instances. However, the third cost-scaling implementation, LEDA, is at least 1.5-2 times slower than them. Furthermore, it failed to solve the largest instances due to number overflow errors, which is denoted as “*error*” in the tables. Since the LEDA library has closed source, we could not change the internally used number types to eliminate this issue. RelaxIV turned out to be highly efficient on these families, it is competitive with or even faster than the other codes. Finally, PDNET runs much slower than all other algorithms, even on small networks.

The performance results for the GOTO families are presented in Figure 3.9 and Table 3.12. In these tests, COS and CS2 also perform similarly, and they are the most efficient on the largest instances of both families. The performance of LEDA is also close to COS and CS2 on GOTO-8 networks, but it is 2-3 times slower on the GOTO-SR networks. Similarly to the previous results, NS turned out to be an order of magnitude faster than the other primal network simplex codes, MCFZIB, CPLEX, and MSim. Furthermore, NS is the most efficient on relatively small networks, but it is substantially slower



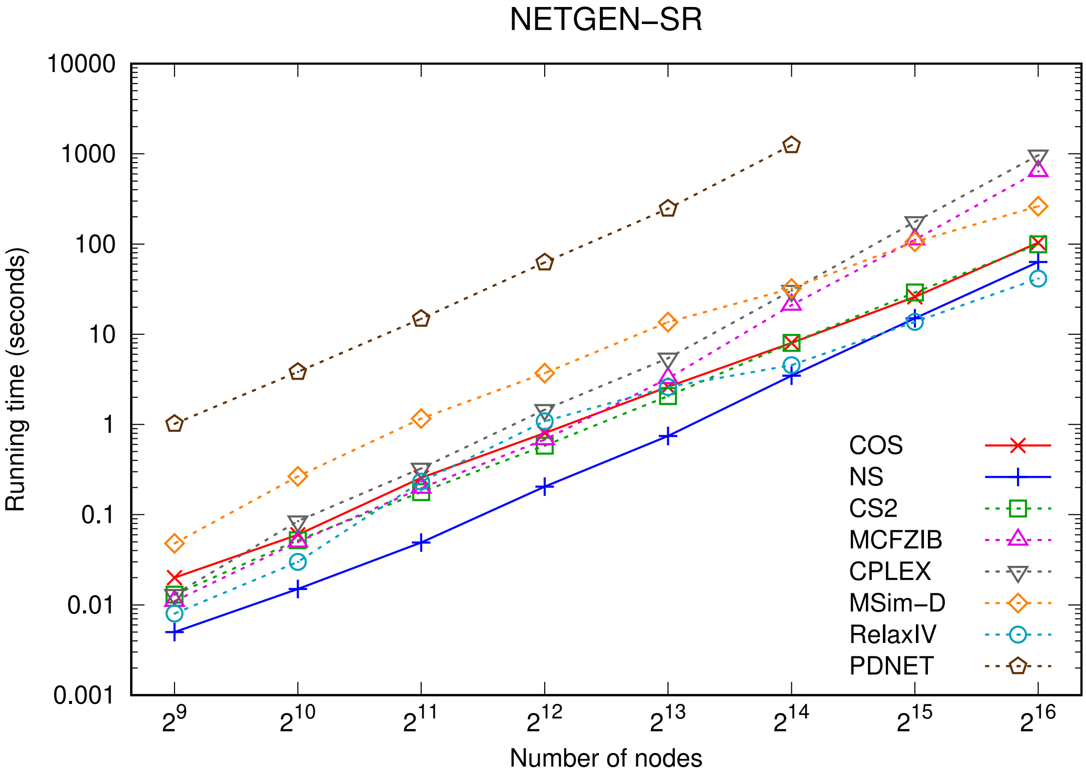


Figure 3.8: Comparison of different MCF solvers on NETGEN networks.

Running time (seconds)

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
GRIDGEN-8	2^{10}	8	0.02	**0.01**	0.02	0.03	0.02	0.02	**0.01**	0.05	**0.01**	0.81
	2^{13}	8	0.40	**0.14**	0.34	0.52	0.53	1.39	0.43	1.33	0.34	13.07
	2^{16}	8	4.24	4.27	4.31	8.29	11.59	109.57	12.68	37.11	**3.77**	361.02
	2^{19}	8	**48.51**	174.48	56.15	*error*	553.96	–	569.76	437.70	63.75	–
	2^{22}	8	532.30	–	**465.14**	*error*	–	–	–	–	470.27	–
GRIDGEN-SR	2^{10}	32	0.08	**0.02**	0.06	0.11	0.05	0.13	0.06	0.55	0.07	3.37
	2^{12}	64	0.95	**0.19**	0.70	1.69	0.59	1.94	0.59	10.17	1.15	40.27
	2^{14}	128	9.72	**2.94**	9.02	21.33	15.13	64.37	16.49	70.48	11.71	846.06
	2^{16}	256	153.09	**73.43**	135.30	368.18	376.05	2537.32	783.91	591.63	113.79	–

Normalized time

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
GRIDGEN-8	2^{10}	8	2.00	**1.00**	2.00	3.00	2.00	2.00	**1.00**	5.00	**1.00**	81.00
	2^{13}	8	2.86	**1.00**	2.43	3.71	3.79	9.93	3.07	9.50	2.43	93.36
	2^{16}	8	1.12	1.13	1.14	2.20	3.07	29.06	3.36	9.84	**1.00**	95.76
	2^{19}	8	**1.00**	3.60	1.16	*error*	11.42	–	11.75	9.02	1.31	–
	2^{22}	8	1.14	–	**1.00**	*error*	–	–	–	–	1.01	–
GRIDGEN-SR	2^{10}	32	4.00	**1.00**	3.00	5.50	2.50	6.50	3.00	27.50	3.50	168.50
	2^{12}	64	5.00	**1.00**	3.68	8.89	3.11	10.21	3.11	53.53	6.05	211.95
	2^{14}	128	3.31	**1.00**	3.07	7.26	5.15	21.89	5.61	23.97	3.98	287.78
	2^{16}	256	2.08	**1.00**	1.84	5.01	5.12	34.55	10.68	8.06	1.55	–

Table 3.11: Comparison of different MCF solvers on GRIDGEN networks.

than the cost-scaling codes on the large sparse graphs. On these hard problem instances, RelaxIV and MSim-D turned out to be very slow, despite their good performance on the easier NETGEN and GRIDGEN problems. In contrast, the relative performance of PDNET compared with the other solvers is much better in the case of GOTO networks than on easier instances. It outperforms the network simplex codes on the large GOTO-8 instances, with the only exception of our implementation.

Figure 3.10 and Table 3.13 show the results for the GRIDGRAPH families, while Figure 3.11 and Table 3.14 present the results for the ROAD families. PDNET failed on most of these special MCF instances with an error message: "STOP disconnected network," which may be due to inappropriate setting of tolerance limits for floating-point computations. Therefore, its running time is not reported for these problem families. However, the CAS algorithm of LEMON is included instead because the augmenting path methods turned out to be interesting in the case of these particular networks.

On the GRID-WIDE instances, the primal network simplex algorithms are the fastest, probably because they are capable of canceling a lot of short negative cycles efficiently due to the limited depth of the underlying spanning tree data structure. NS turned out to be the most efficient implementation again. In the case of GRID-LONG networks, however, the augmenting paths or cycles are much longer, and CAS turned out to be by far the fastest algorithm. The cost-scaling codes are significantly slower, but they outperform the network simplex codes and RelaxIV.

On the ROAD-PATHS family, CAS is much faster than the primal network simplex and cost-scaling methods, while MSim-D and RelaxIV are even orders of magnitude slower.

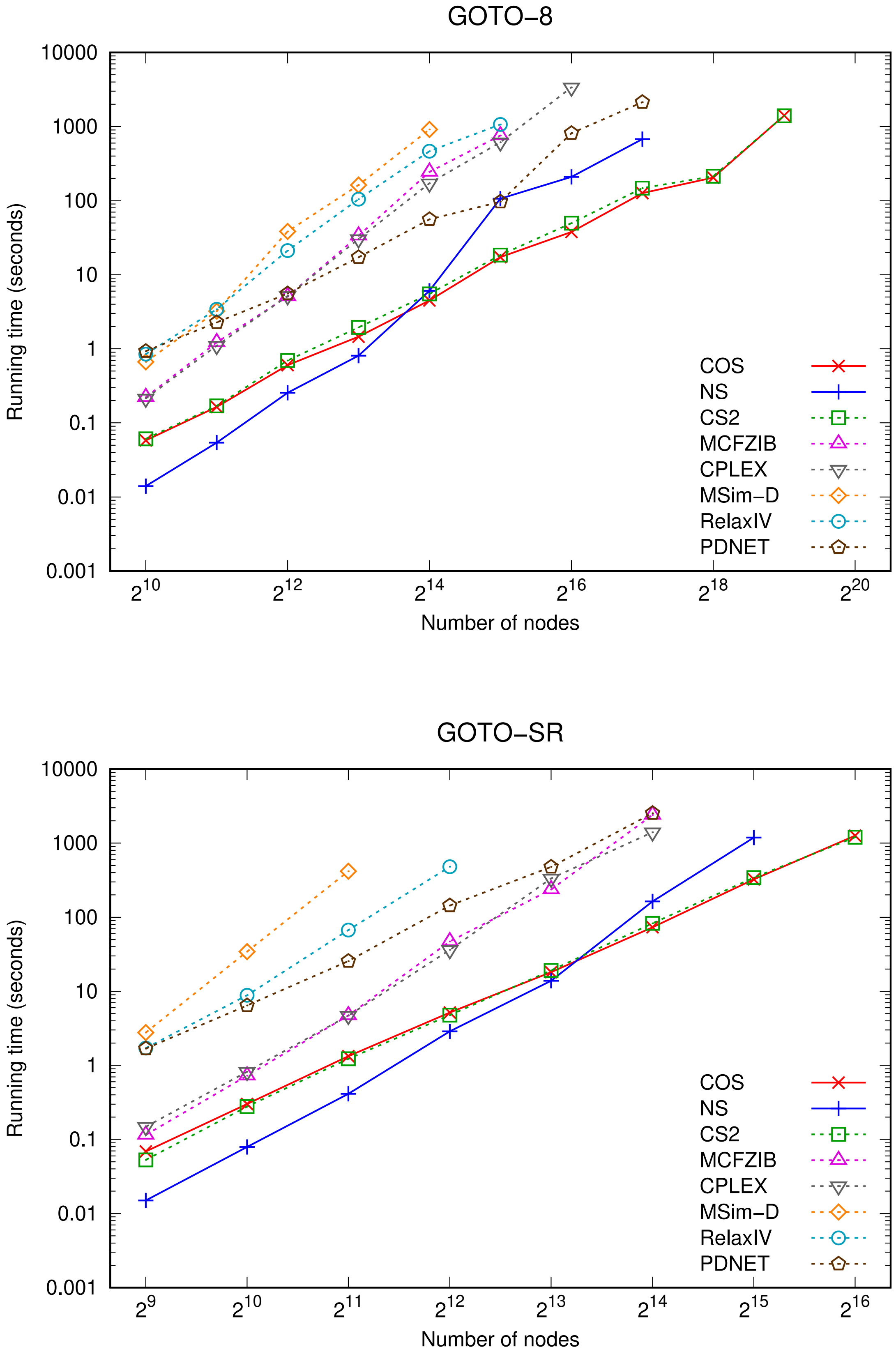


Figure 3.9: Comparison of different MCF solvers on GOTO networks.

Running time (seconds)

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
GOTO-8	2^{10}	8	0.06	**0.01**	0.06	0.08	0.22	0.21	0.17	0.67	0.85	0.93
	2^{13}	8	1.45	**0.81**	1.94	1.68	33.60	30.41	29.20	162.14	104.71	17.22
	2^{16}	8	**37.87**	208.61	49.67	53.46	–	3379.22	–	–	–	811.74
	2^{19}	8	1413.75	–	**1398.30**	1659.08	–	–	–	–	–	–
GOTO-SR	2^{10}	32	0.30	**0.08**	0.28	0.37	0.72	0.82	0.58	34.34	8.84	6.45
	2^{12}	64	5.17	**2.88**	4.77	9.62	47.02	36.13	42.12	–	479.58	143.41
	2^{14}	128	**72.98**	163.26	82.72	190.09	2395.41	1407.59	1601.92	–	–	2534.18
	2^{16}	256	1260.33	–	**1203.92**	3095.03	–	–	–	–	–	–

Normalized time

			LEMON		Other solvers							
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
GOTO-8	2^{10}	8	6.00	**1.00**	6.00	8.00	22.00	21.00	17.00	67.00	85.00	93.00
	2^{13}	8	1.79	**1.00**	2.40	2.07	41.48	37.54	36.05	200.17	129.27	21.26
	2^{16}	8	**1.00**	5.51	1.31	1.41	–	89.23	–	–	–	21.43
	2^{19}	8	1.01	–	**1.00**	1.19	–	–	–	–	–	–
GOTO-SR	2^{10}	32	3.75	**1.00**	3.50	4.63	9.00	10.25	7.25	429.25	110.50	80.63
	2^{12}	64	1.80	**1.00**	1.66	3.34	16.33	12.55	14.63	–	166.52	49.80
	2^{14}	128	**1.00**	2.24	1.13	2.60	32.82	19.29	21.95	–	–	34.72
	2^{16}	256	1.05	–	**1.00**	2.57	–	–	–	–	–	–

Table 3.12: Comparison of different MCF solvers on GOTO networks.

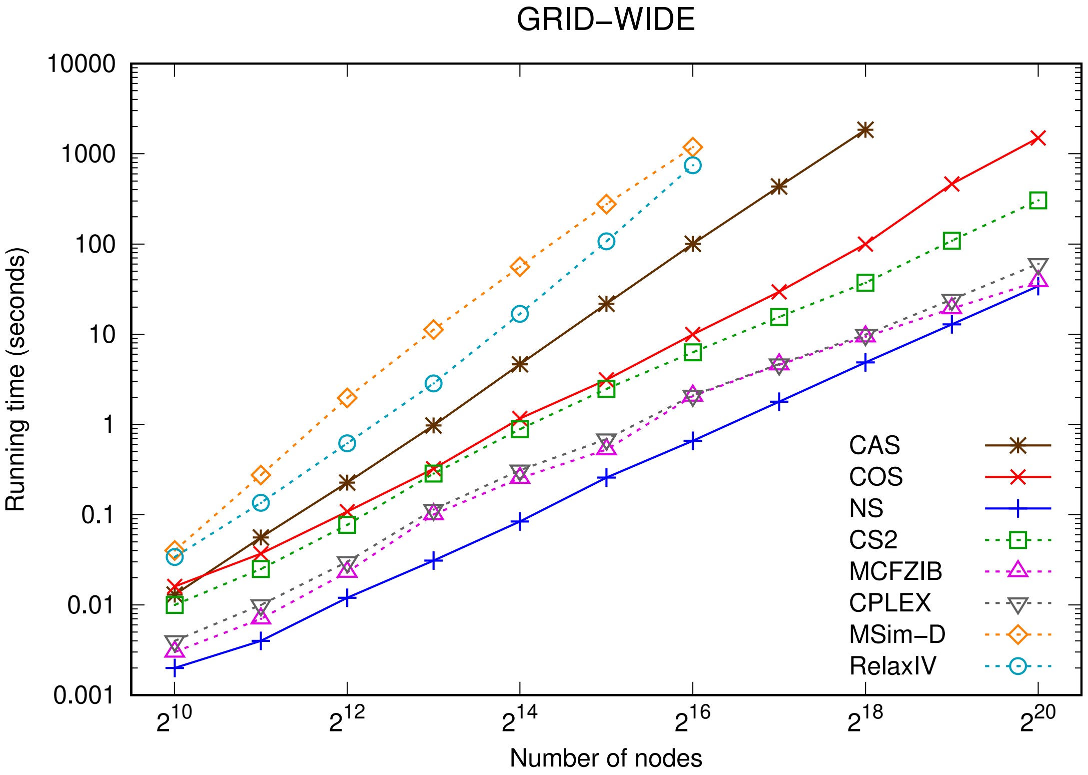
Running time (seconds)

			LEMON			Other solvers						
Problem family	n	m/n	CAS	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV
GRID-WIDE	2^{10}	2.04	**0.01**	0.02	**0.01**	**0.01**	0.02	**0.01**	**0.01**	**0.01**	0.04	0.03
	2^{13}	2.06	0.97	0.32	**0.03**	0.29	0.54	0.10	0.11	0.07	11.23	2.84
	2^{16}	2.06	100.52	9.96	**0.66**	6.30	49.45	2.07	2.10	1.77	1184.36	743.83
	2^{19}	2.06	–	463.73	**12.83**	108.82	*error*	19.19	24.21	16.31	–	–
GRID-LONG	2^{10}	1.95	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	0.02	0.02
	2^{13}	1.94	**0.05**	0.21	0.08	0.13	0.16	0.43	0.73	0.44	2.85	0.81
	2^{16}	1.94	**0.36**	2.38	5.53	2.82	2.20	47.80	51.06	37.97	264.50	73.75
	2^{19}	1.94	**3.05**	22.39	504.82	24.85	*error*	–	3483.86	3319.92	–	–

Normalized time

			LEMON			Other solvers						
Problem family	n	m/n	CAS	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV
GRID-WIDE	2^{10}	2.04	**1.00**	2.00	**1.00**	**1.00**	2.00	**1.00**	**1.00**	**1.00**	4.00	3.00
	2^{13}	2.06	32.33	10.67	**1.00**	9.67	18.00	3.33	3.67	2.33	374.33	94.67
	2^{16}	2.06	152.30	15.09	**1.00**	9.55	74.92	3.14	3.18	2.68	1794.48	1127.02
	2^{19}	2.06	–	36.14	**1.00**	8.48	*error*	1.50	1.89	1.27	–	–
GRID-LONG	2^{10}	1.95	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	2.00	2.00
	2^{13}	1.94	**1.00**	4.20	1.60	2.60	3.20	8.60	14.60	8.80	57.00	16.20
	2^{16}	1.94	**1.00**	6.61	15.36	7.83	6.11	132.78	141.83	105.47	734.72	204.86
	2^{19}	1.94	**1.00**	7.34	165.51	8.15	*error*	–	1142.25	1088.50	–	–

Table 3.13: Comparison of different MCF solvers on GRIDGRAPH networks. PDNET failed on most of these instances, so it is excluded.



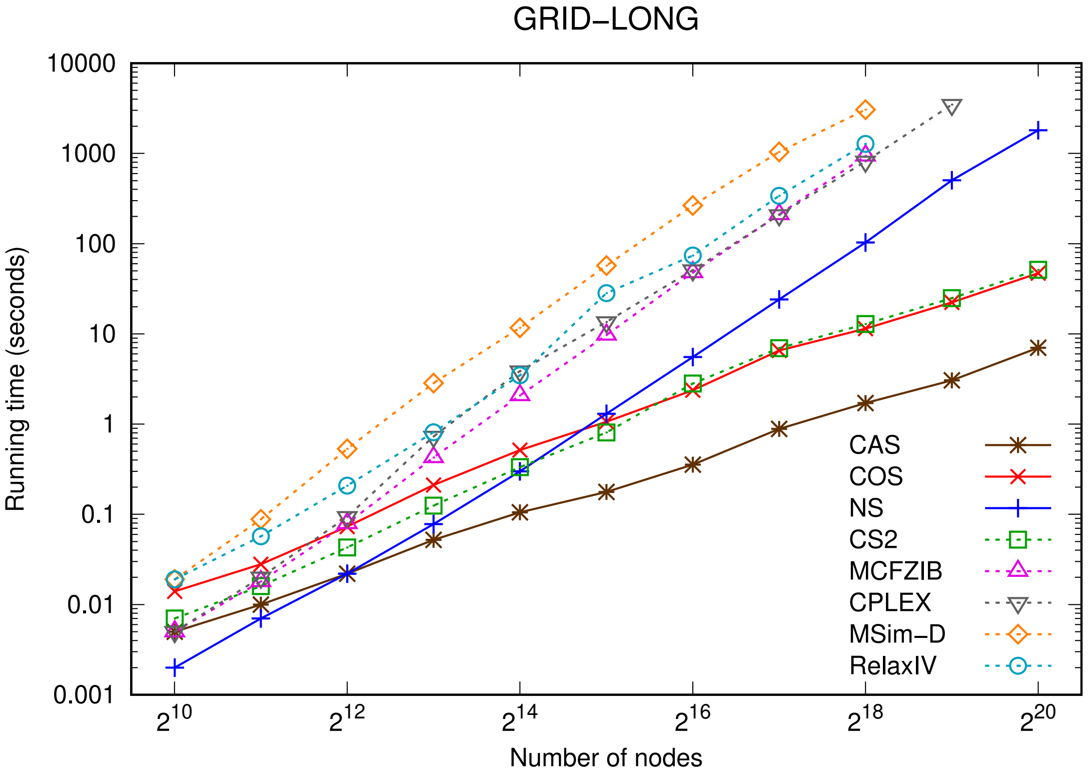


Figure 3.10: Comparison of different MCF solvers on GRIDGRAPH networks. PDNET failed on most of these instances, so it is excluded.

The results are similar on the ROAD-FLOW networks, with the exception that CAS has smaller advantage. CS2 outperforms CAS on the largest ROAD-FLOW instances, but note that the SSP algorithm of LEMON is even faster (cf. Tables 3.6 and 3.14).

Running time (seconds)

			LEMON			Other solvers						
Problem family	n	m/n	CAS	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV
ROAD-PATHS	9,559	3.11	**0.01**	0.16	0.06	0.13	0.15	0.15	0.22	0.12	1.35	0.67
	116,920	2.27	**0.30**	4.69	3.15	2.65	3.30	6.22	6.52	5.70	188.71	219.60
	519,157	2.44	**3.46**	36.13	42.58	19.38	*error*	83.00	80.64	75.95	–	–
	2,073,870	2.49	**19.06**	248.32	506.40	101.02	*error*	948.34	968.58	831.71	–	–
ROAD-FLOW	9,559	3.11	**0.03**	0.21	0.06	0.14	0.19	0.15	0.18	0.12	1.48	0.72
	116,920	2.27	**1.50**	8.43	4.65	4.37	5.51	11.21	11.13	10.09	310.20	376.14
	519,157	2.44	**19.05**	54.38	47.70	23.59	*error*	105.36	91.22	97.09	–	–
	2,073,870	2.49	336.75	471.64	1106.19	**157.24**	*error*	1288.25	2200.65	1352.18	–	–

Normalized time

			LEMON			Other solvers						
Problem family	n	m/n	CAS	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV
ROAD-PATHS	9,559	3.11	**1.00**	16.00	6.00	13.00	15.00	15.00	22.00	12.00	135.00	67.00
	116,920	2.27	**1.00**	15.63	10.50	8.83	11.00	20.73	21.73	19.00	629.03	732.00
	519,157	2.44	**1.00**	10.44	12.31	5.60	*error*	23.99	23.31	21.95	–	–
	2,073,870	2.49	**1.00**	13.03	26.57	5.30	*error*	49.76	50.82	43.64	–	–
ROAD-FLOW	9,559	3.11	**1.00**	7.00	2.00	4.67	6.33	5.00	6.00	4.00	49.33	24.00
	116,920	2.27	**1.00**	5.62	3.10	2.91	3.67	7.47	7.42	6.73	206.80	250.76
	519,157	2.44	**1.00**	2.85	2.50	1.24	*error*	5.53	4.79	5.10	–	–
	2,073,870	2.49	2.14	3.00	7.04	**1.00**	*error*	8.19	14.00	8.60	–	–

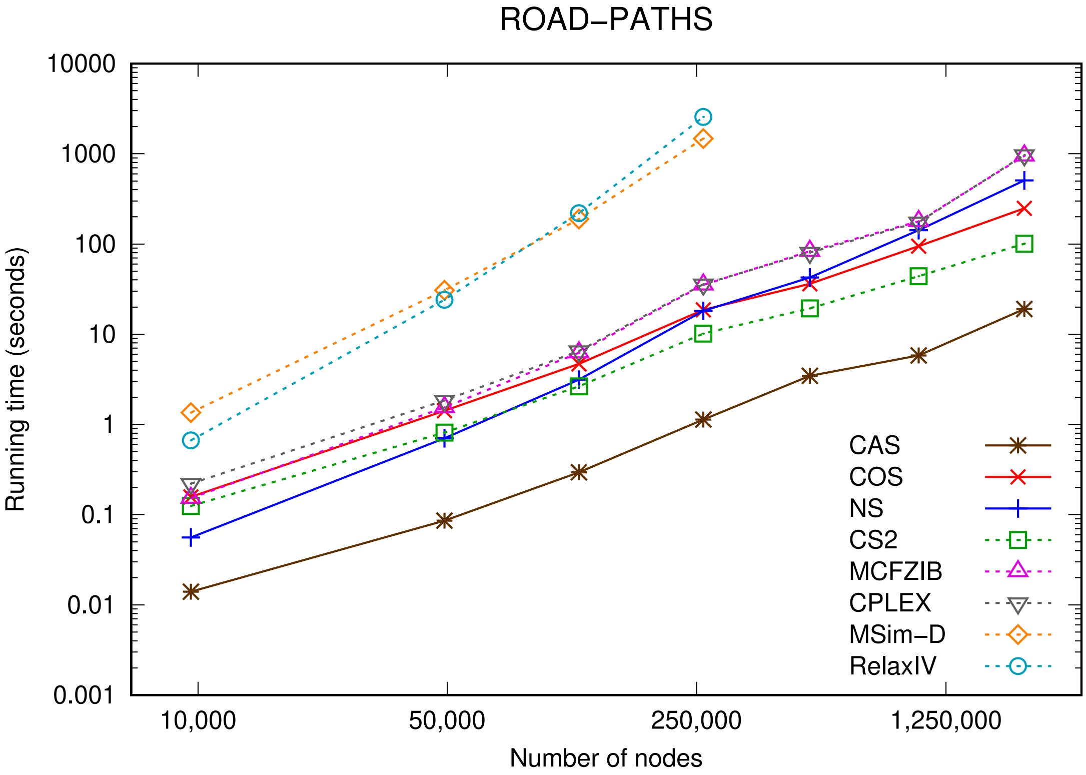
Table 3.14: Comparison of different MCF solvers on ROAD networks. PDNET failed on most of these instances, so it is excluded.

Figure 3.12 and Table 3.15 present the benchmark results for the VISION families. CS2 is the fastest to solve these hard instances, while COS is about 1.5-2 times slower. All other codes perform much worse, including the third cost-scaling implementation, LEDA. It even failed to solve all VISION-PROP instances due to number overflow errors. NS turned out to be much faster than the other three primal network simplex codes again. RelaxIV is very slow on these instances, but MSim-D and PDNET perform even worse, they could not solve any VISION instance within the one-hour time limit. Therefore, the latter two solvers are not included in Figure 3.12 and Table 3.15.

The choice of the cost function has only modest impact on the performance of the algorithms on VISION networks. Surprisingly, the cost-scaling codes tend to be slightly slower on both VISION-PROP and VISION-INV instances than in the case of random costs, while other methods are faster on these networks, especially on VISION-INV instances.

Finally, in order to provide a compact summary of our experiments, Tables 3.16 and 3.17 compare the implementations on networks of various problem families having approximately the same number of arcs (2^{18} and 2^{21}). Apart from the average running time, these tables also report the standard deviation for each algorithm, measured on five instances that were generated using the same parameters but different random seeds.

These tables also demonstrate that the best overall performance is achieved by our COS and NS codes and the CS2 solver. On relatively small networks, NS is usually



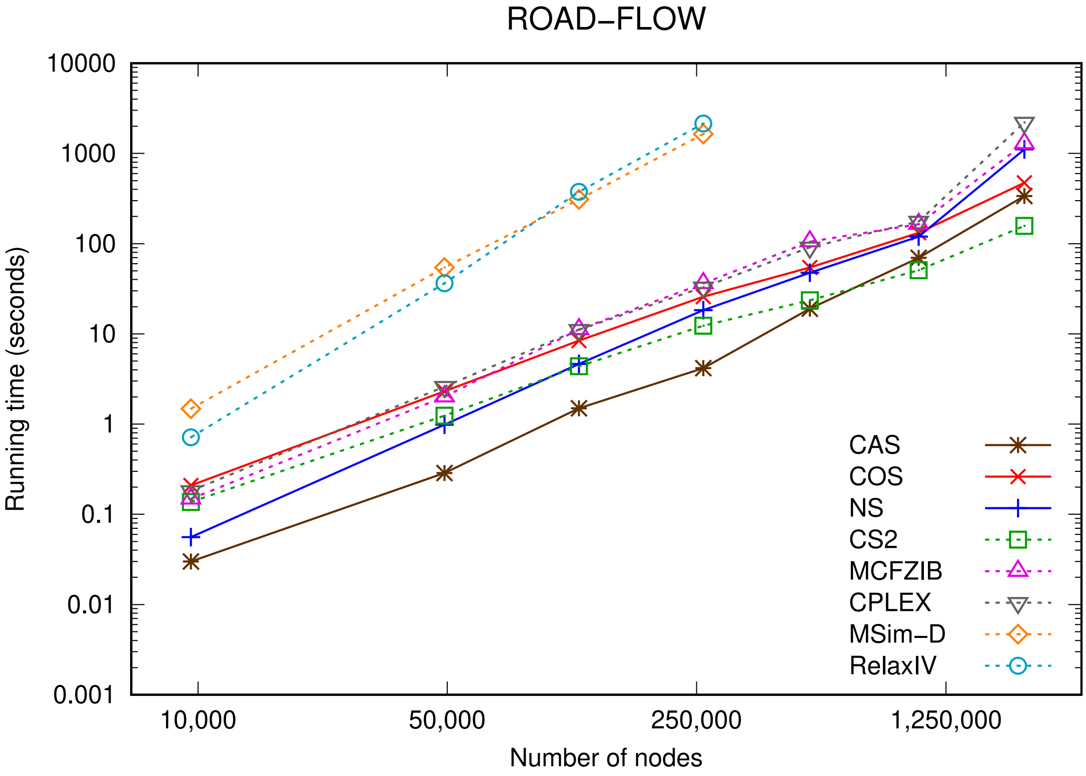


Figure 3.11: Comparison of different MCF solvers on ROAD networks. PDNET failed on most of these instances, so it is excluded.

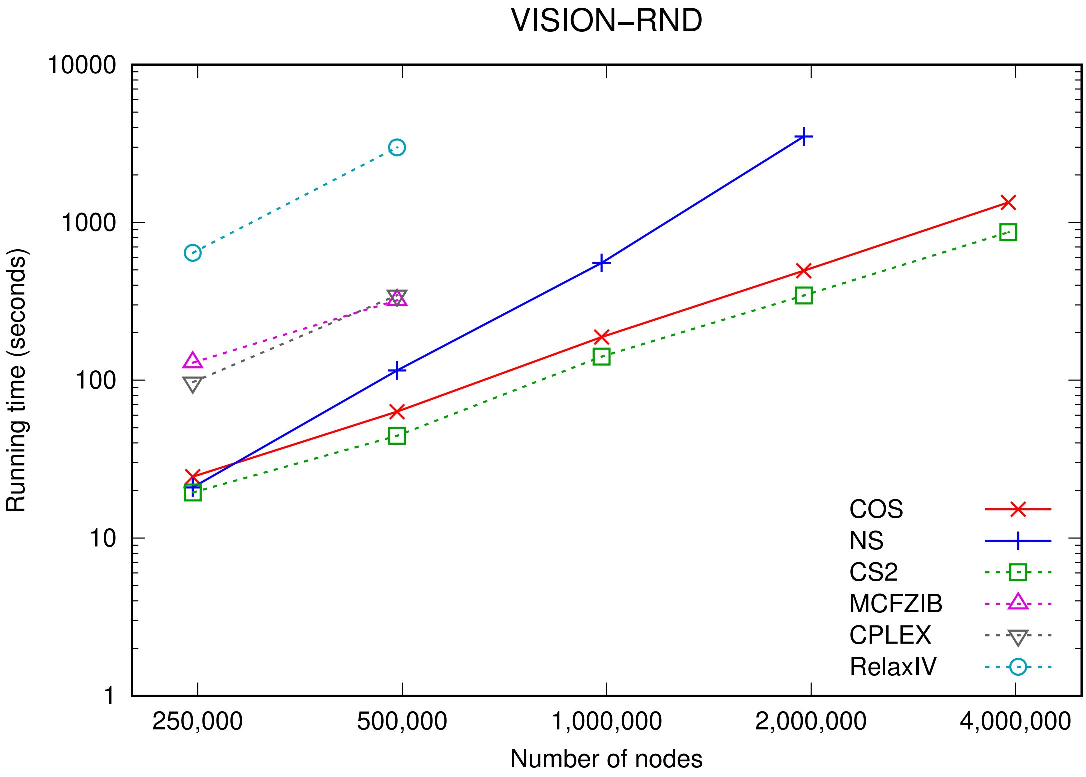


Figure 3.12: Comparison of different MCF solvers on VISION-RND networks. MSim-D and PDNET could not solve any of these instances in one hour, so they are excluded.

superior to the other implementations, but on large sparse networks, COS and CS2 are the fastest and most robust. The other implementations of the cost-scaling and primal network simplex methods (i.e., LEDA, MCFZIB, CPLEX, and MSim) turned out to be less efficient in general, and LEDA also has numerical issues in the case of large problem instances. MSim-D, RelaxIV, and PDNET are not robust at all, although they perform reasonably well on some problem instances. The deviation of running time turned out to be moderate in most cases, but MSim-D and RelaxIV seem to be less stable in this aspect as well.

Apart from the presented results, many other experiments were also conducted varying several parameters of the problem instances. A notable observation is that the magnitudes of the supply, capacity, and cost values hardly affect the running time of the algorithms, but the relation between supply and capacity values is more important, which was also observed before [35]. If the arc capacities are large compared to the supply values and, thereby, incorporate only "loose" (or infinite) bounds for the feasible solutions, then the problem instances are generally easier to solve, especially for network simplex methods. In the case of tight capacities, however, the cost-scaling algorithms are more robust according to our experiments. For more details, see [2, 3].

We also compared the number of basic operations performed by the cost-scaling and primal network simplex implementations whose source codes we had access to. We found that our COS code tends to perform significantly less push operations than CS2 due to the partial augment method (see Section 3.2.6), but it performs approximately the

Running time (seconds)

			LEMON		Other solvers					
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	RelaxIV
VISION-RND	245,762	5.82	24.43	20.92	**19.46**	57.48	129.25	97.11	163.30	641.78
	491,522	5.85	63.27	115.48	**44.50**	218.61	320.91	345.68	409.43	2987.68
	983,042	5.88	187.69	554.55	**141.07**	1132.98	–	–	–	–
	1,949,698	5.91	494.24	3510.13	**344.35**	–	–	–	–	–
	3,899,394	5.92	1341.04	–	**865.86**	–	–	–	–	–
VISION-PROP	245,762	5.82	49.75	**21.17**	25.33	*error*	130.91	100.84	140.90	730.04
	491,522	5.85	111.03	112.52	**61.70**	*error*	228.54	333.83	243.70	2112.62
	983,042	5.88	406.98	458.90	**202.76**	*error*	3356.82	2387.97	–	–
	1,949,698	5.91	842.25	2830.07	**459.16**	*error*	–	–	–	–
	3,899,394	5.92	2426.39	–	**1127.42**	*error*	–	–	–	–
VISION-INV	245,762	5.82	41.29	**14.26**	25.11	50.25	27.95	60.27	31.36	367.71
	491,522	5.85	97.07	81.93	58.10	205.03	**57.15**	280.85	63.09	1451.90
	983,042	5.88	298.76	304.45	**177.39**	599.32	2040.11	2632.67	2213.28	–
	1,949,698	5.91	820.11	2168.45	**434.57**	2314.01	–	–	–	–
	3,899,394	5.92	2201.84	–	**1040.63**	*error*	–	–	–	–

Normalized time

			LEMON		Other solvers					
Problem family	n	m/n	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	RelaxIV
VISION-RND	245,762	5.82	1.26	1.08	**1.00**	2.95	6.64	4.99	8.39	32.98
	491,522	5.85	1.42	2.60	**1.00**	4.91	7.21	7.77	9.20	67.14
	983,042	5.88	1.33	3.93	**1.00**	8.03	–	–	–	–
	1,949,698	5.91	1.44	10.19	**1.00**	–	–	–	–	–
	3,899,394	5.92	1.55	–	**1.00**	–	–	–	–	–
VISION-PROP	245,762	5.82	2.35	**1.00**	1.20	*error*	6.18	4.76	6.66	34.48
	491,522	5.85	1.80	1.82	**1.00**	*error*	3.70	5.41	3.95	34.24
	983,042	5.88	2.01	2.26	**1.00**	*error*	16.56	11.78	–	–
	1,949,698	5.91	1.83	6.16	**1.00**	*error*	–	–	–	–
	3,899,394	5.92	2.15	–	**1.00**	*error*	–	–	–	–
VISION-INV	245,762	5.82	2.90	**1.00**	1.76	3.52	1.96	4.23	2.20	25.79
	491,522	5.85	1.70	1.43	1.02	3.59	**1.00**	4.91	1.10	25.41
	983,042	5.88	1.68	1.72	**1.00**	3.38	11.50	14.84	12.48	–
	1,949,698	5.91	1.89	4.99	**1.00**	5.32	–	–	–	–
	3,899,394	5.92	2.12	–	**1.00**	*error*	–	–	–	–

Table 3.15: Comparison of different MCF solvers on VISION networks. MSim-D and PDNET could not solve any of these instances in one hour, so they are excluded.

same number of relabel operations in most cases. On the other hand, CS2 may find admissible arcs faster due to its speculative arc fixing heuristic (see [118, 120]). The COS implementation could be further improved by applying this technique.

As for the network simplex algorithms, MCFZIB and MSim perform exactly the same number of iterations as they apply the same pivot strategy. Our NS code, however, tends to perform substantially less iterations than them, although its selection rule checks either less or more arcs depending on the characteristics of the problem instance.

Average running time (seconds) and standard deviation

		LEMON		Other solvers							
Problem family	m	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
NETGEN-8	262,144	1.88	1.90	1.88	3.66	6.10	30.44	6.58	10.11	**1.63**	105.22
		0.09	0.09	0.05	0.12	0.40	2.04	1.21	3.98	0.39	6.77
NETGEN-SR	262,144	0.80	**0.21**	0.58	1.42	0.68	1.46	0.70	3.71	1.09	62.73
		0.06	0.01	0.01	0.05	0.02	0.05	0.02	1.12	0.63	1.76
GRIDGEN-8	262,096	1.96	**1.67**	1.88	3.30	3.96	26.68	3.49	8.45	1.89	127.28
		0.06	0.12	0.11	0.08	0.49	2.20	0.46	1.29	0.29	28.39
GRIDGEN-SR	262,208	0.95	**0.19**	0.70	1.69	0.59	1.94	0.59	10.17	1.15	40.27
		0.05	0.01	0.02	0.05	0.05	0.02	0.04	3.59	0.33	2.02
GOTO-8	262,144	**17.30**	106.37	18.45	17.94	756.04	613.39	763.73	–	1070.03	96.37
		0.91	4.74	1.49	1.28	88.83	81.26	147.10		76.45	4.35
GOTO-SR	262,144	5.17	**2.88**	4.77	9.62	47.02	36.13	42.12	–	479.58	143.41
		0.22	0.06	0.07	0.97	5.89	2.93	2.08		22.53	6.54
GRID-WIDE	270,320	29.56	**1.79**	15.55	347.94	4.60	4.67	4.00	–	–	*error*
		5.07	0.04	0.37	36.00	0.33	0.24	0.27			
GRID-LONG	253,968	**6.55**	24.02	6.93	7.50	209.99	208.11	169.43	1041.44	338.70	*error*
		1.32	0.84	1.82	1.62	5.95	3.09	6.35	67.53	44.75	
ROAD-PATHS	265,402	4.69	3.15	**2.65**	3.30	6.22	6.52	5.70	188.71	219.60	*error*
		0.31	0.39	0.13	0.10	0.75	1.08	0.66	59.62	81.73	
ROAD-FLOW	265,402	8.43	4.65	**4.37**	5.51	11.21	11.13	10.09	310.20	376.14	*error*
		2.14	1.50	1.35	1.05	4.50	4.22	3.93	164.46	215.84	

Table 3.16: Comparison of different MCF solvers on networks with approx. 2^{18} arcs.

Average running time (seconds) and standard deviation

		LEMON		Other solvers							
Problem family	m	COS	NS	CS2	LEDA	MCFZIB	CPLEX	MSim	MSim-D	RelaxIV	PDNET
NETGEN-8	2,097,152	22.82	102.60	20.87	*error*	198.84	1511.92	286.87	146.76	**16.01**	–
		2.03	3.91	1.09		14.79	46.15	22.09	50.81	5.59	
NETGEN-SR	2,097,152	8.00	**3.47**	8.01	19.84	21.11	30.45	15.50	32.25	4.54	1250.90
		0.33	0.13	0.27	0.40	0.73	0.35	0.48	4.00	0.26	59.38
GRIDGEN-8	2,097,160	**22.92**	53.08	26.03	*error*	150.53	1381.98	162.02	130.17	24.76	–
		4.45	3.23	6.41		50.91	85.60	21.54	54.41	9.61	
GRIDGEN-SR	2,097,280	9.72	**2.94**	9.02	21.33	15.13	64.37	16.49	70.48	11.71	846.06
		0.56	0.27	0.41	0.20	0.75	7.52	2.37	21.29	5.57	30.78
GOTO-8	2,097,152	**204.43**	–	214.20	236.63	–	–	–	–	–	–
		14.18		12.09	22.74						
GOTO-SR	2,097,152	**72.98**	163.26	82.72	190.09	2395.41	1407.59	1601.92	–	–	2534.18
		4.75	0.64	12.80	7.38	94.82	178.40	475.33			85.64
GRID-WIDE	2,162,672	1499.82	34.21	305.21	*error*	38.33	60.21	**31.92**	–	–	*error*
		196.15	1.05	73.16		1.18	1.46	1.12			
GRID-LONG	2,031,632	**47.13**	1804.92	51.38	*error*	–	–	–	–	–	*error*
		5.38	101.48	16.34							
ROAD-PATHS	2,653,624	94.37	141.91	**43.99**	*error*	178.05	174.28	164.20	–	–	*error*
		9.80	24.90	0.72		40.38	36.33	34.93			
ROAD-FLOW	2,653,624	131.93	119.44	**50.74**	*error*	164.37	176.40	153.19	–	–	*error*
		8.45	7.88	1.25		21.05	14.57	18.59			
VISION-RND	2,877,382	63.27	115.48	**44.50**	218.61	320.91	345.68	409.43	–	2987.68	–
		5.51	8.39	1.10	33.35	24.24	104.82	137.19		1070.66	

Table 3.17: Comparison of different MCF solvers on networks with approx. 2^{21} arcs.

Chapter 4

Maximum common subgraphs

This chapter presents algorithms and heuristics for finding maximum common subgraphs, together with an experimental study that demonstrates their efficiency in practice. This is based on a joint work with Péter Englert, which was carried out at ChemAxon and published in the article [1].

The chapter is organized as follows. Section 4.1 discusses the problem and its applications in the field of cheminformatics. Section 4.2 describes the algorithms we considered. Section 4.3 presents the heuristic improvements and extensions implemented as part of this work. Finally, Section 4.4 presents computational results.

4.1 The maximum common subgraph problem

Finding the largest common subgraph of two graphs is a computationally hard optimization problem with important applications in diverse fields, such as computational chemistry [85, 90, 206], pattern recognition and image processing [47, 48, 64], video indexing [215], and classification of text documents [212]. Over the last decades, numerous algorithms have been developed for solving this problem [65, 84, 85, 90, 151, 204, 206].

In this work, we study this problem in the context of cheminformatics, where it is commonly applied to describe the structural similarity of molecular graphs and to match them to each other. The presented ideas and methods are, however, also applicable in other fields where undirected labeled graphs are to be compared.

4.1.1 Problem statement

The maximum common subgraph problem has two different definitions, both of which are widely used in the literature. It can be formulated either as the *maximum common induced subgraph* (MCIS) problem or as the *maximum common edge subgraph* (MCES) problem.

Let G_1 and G_2 be two simple, undirected, labeled graphs (see corresponding definitions in Section 2.10). G is a *common induced subgraph* of G_1 and G_2 if it is an induced subgraph of both G_1 and G_2, and G is a *common subgraph* if it is a subgraph of both G_1 and G_2. The MCIS problem is to find a common induced subgraph of two labeled graphs containing as many *nodes* as possible. In contrast, the MCES problem is to find a common subgraph (not necessarily induced) containing as many *edges* as possible.

Both the MCIS and MCES problems are inherently complex. They are generalizations of the induced subgraph isomorphism problem and the subgraph isomorphism problem, respectively, which are discussed in Section 2.9. That is, they are NP-hard (or, if stated as decision problems, NP-complete).

Further distinction can be made between *connected* and *disconnected* MCIS and MCES problems. In the connected case, a common subgraph must be composed of a single component, while in the disconnected case, it can consist of an arbitrary number of connected components.

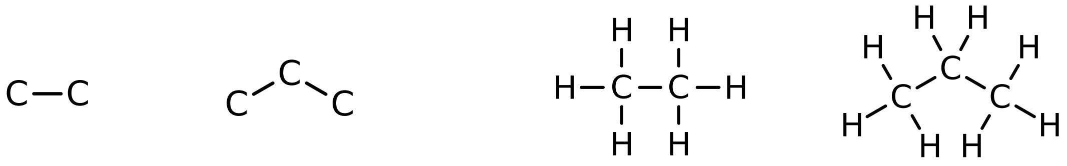
In this thesis, we mainly study the MCES problem as it is more relevant in cheminformatics. Furthermore, unless otherwise stated, we allow common subgraphs to be disconnected. However, the presented algorithms are capable of solving the connected MCES problem as well.

4.1.2 Molecular graphs

This research was conducted at ChemAxon, a company developing software solutions and services for computational chemistry and biology. Therefore, the MCES problem was studied in this domain, applied to molecular graphs. Here we discuss the basic concepts and notations related to the representation and analysis of molecular graphs, as well as the applications of the MCES problem in this context.

In cheminformatics, the typical representation of molecules is by means of *molecular graphs* [235, 246]. A molecular graph is defined to be a simple, undirected, labeled graph in which the nodes represent the atoms, the edges represent the chemical bonds, and the labels of the nodes and edges represent the atom types (C, N, O, etc.) and bond orders (single, double, triple, aromatic), respectively. Therefore, we usually refer to the nodes, edges, and (short) cycles of molecular graphs as *atoms*, *bonds*, and *rings*, respectively.

With respect to structural comparison and graph matching, hydrogen atoms are typically not included in molecular graphs. Only the other atoms and the bonds between them are represented explicitly, while hydrogens are assumed to fill the unused valences of the other atoms, so they are determined implicitly. This representation is required for the practical usage of the subgraph concept, not only for the sake of simplicity, as it is demonstrated in Figure 4.1. In the followings, we always assume that hydrogen atoms are excluded from molecular graphs.



(a) Molecular graphs without hydrogen atoms

(b) Molecular graphs with hydrogen atoms

Figure 4.1: Examples of different representations of molecules. The molecular graph of ethane (a carbon chain with two carbon atoms) is a subgraph of the molecular graph of propane (a carbon chain with three carbon atoms) only if the hydrogen atoms are excluded.

Figure 4.2 shows a typical example of depicting molecular graphs. According to the conventions of chemical drawing, labels of carbon atoms (C) are not displayed in the followings. Single, double, and triple chemical bonds are depicted as single, double, and triple lines, respectively. Aromatic rings are indicated with a circle inside the ring (which usually contains 5 or 6 bonds and is depicted as a pentagon or hexagon, respectively). For our study, nevertheless, these bonds only mean edges with a label that is distinguished from other bond types (single, double, and triple).

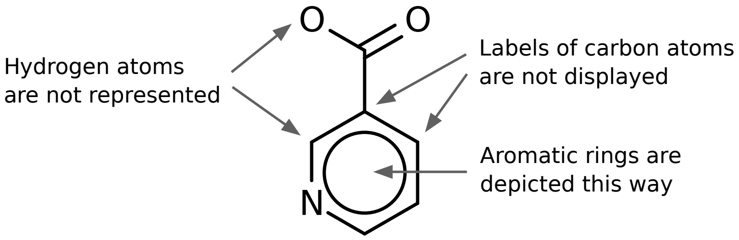

Hydrogen atoms are not represented
O
O
Labels of carbon atoms are not displayed
N
Aromatic rings are depicted this way

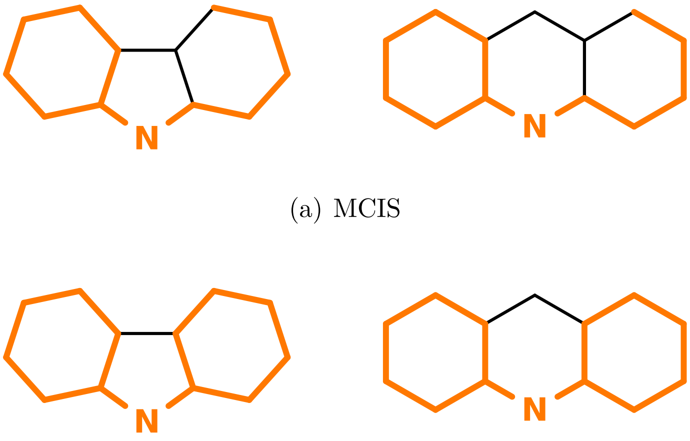
Figure 4.2: Example of displaying molecular graphs.

Finding maximum common subgraphs, by means of either the MCIS or the MCES problem, plays an important role in a wide range of applications in the field of computational chemistry and, increasingly, biology [90]. A typical application is similarity search, which is essential in the early stages of the drug discovery process [76, 245, 246, 247]. Intuitive measures of the similarity of two molecular graphs can be defined on the basis of the size of their maximum common subgraph [49, 145, 204, 205]. Other applications include clustering [38, 110, 224, 225], NMR spectral studies [56], design of chemical space networks [252], molecular alignment [152, 174], and reaction mapping [93, 176, 234].

In the cheminformatics literature, this problem is usually referred to as *maximum common substructure* (MCS) search or the *maximum overlapping set* (MOS) problem, and it is defined to be either the MCIS or the MCES problem. Although several algorithms solve the MCIS problem (e.g., [53, 151]), the MCES concept is considered to be more relevant in the case of molecular graphs since it is closer to the intuitive notion of chemical similarity [83, 176, 206]. Figure 4.3 illustrates the difference.

In some applications, connected common subgraphs are to be found. Figure 4.4 illustrates the difference between the connected and disconnected MCES of two molecular graphs. As it is shown in this example, the connected MCES can be larger than the largest connected component of the disconnected MCES. Intermediate approaches have also been developed applying constraints on the number of components or on their topological relationships [151]. Another interesting approach is to find connected common subgraphs tolerating a few mismatches in atom and bond types [241].

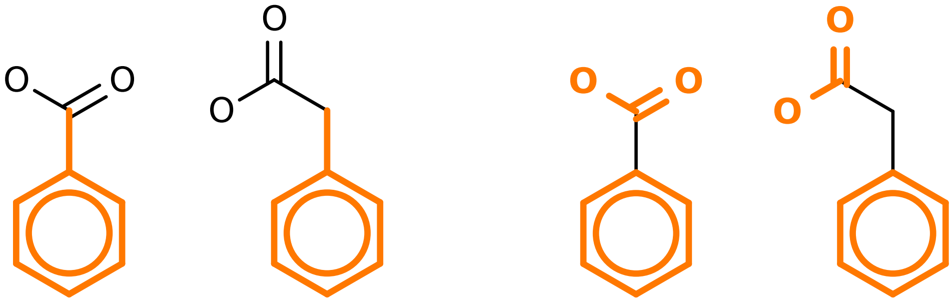
A common subgraph of the input graphs G_1 and G_2 is usually represented as a partial *mapping* from the bonds of G_1 to the bonds of G_2. That is, a bond of G_2 is assigned to each bond of a subgraph of G_1 so that associated bonds represent the same bond in the common subgraph. Not only is this mapping practical for the algorithms, but it is also required in numerous applications, as it defines a correspondence between equivalent (isomorphic) parts of the two molecular graphs. (In the case of the MCIS problem, the



(a) MCIS

(b) MCES

Figure 4.3: Comparison of the MCIS and MCES of two molecular graphs.



(a) Connected MCES

(b) Disconnected MCES

Figure 4.4: Comparison of the connected and the disconnected MCES of two molecular graphs.

mapping is composed of atom pairs instead of bond pairs.) Note that multiple mappings may define isomorphic common subgraphs. Later, we elaborate upon the importance of the mapping considering other features apart from the common subgraph it defines.

Finally, we remark that finding the maximum common subgraph of more than two graphs, often called the *multi-MCS problem*, is also required in certain applications. Although it can be approximated using consecutive MCES or MCIS computations between pairs of graphs, more efficient methods are often desirable [71, 134], but this is not in the scope of this work.

4.2 Algorithms

There are many published algorithms for finding maximum common subgraphs. They can be categorized according to multiple aspects, for example, whether the algorithm is exact or heuristic, whether it solves the MCIS or the MCES variant of the problem, or whether it finds connected or disconnected common subgraphs. The applied approaches also show a wide variety, including backtracking [53, 56, 160, 175], reducing the problem to finding a maximum clique of a derived graph [86, 158, 203, 204], genetic algorithms [43, 240], and greedy heuristics [151]. Some methods also utilize the concept of reduced graphs [20, 110, 202, 227]. Remarkable surveys and experimental studies of various algorithms can be found in [65, 84, 85, 90, 206].

Applications of MCIS/MCES search impose strict and often conflicting requirements on the solution methods. Although the problem is NP-hard, algorithms are required to be fast, which is why heuristic methods are often applied. On the other hand, the results should be equal to, or at least very close to the actual maximum. Moreover, in cheminformatics applications, features that make sense to a chemist but are hard to grasp formally are sometimes desired, such as taking topological features into account when parts of the input molecules are mapped to each other.

In order to satisfy these requirements, we developed two efficient heuristic algorithms for MCES search: one based on the maximum clique approach and another one based on a greedy approach called the build-up method [151]. We also devised several novel heuristics to improve both their running time and the approximation of the optimal results.

In this section, we briefly introduce the two algorithms we implemented according to the available literature, while the next section presents the improvements and extensions, which are the main contribution of this work.

4.2.1 Clique-based algorithm

One of the most popular algorithmic approaches is the reduction of the MCIS/MCES problem to the maximum clique problem. Several state-of-the-art algorithms, both exact and heuristic, use this approach [86, 158, 203, 204] because of the extensive literature and efficient algorithms available for clique search [193, 248].

The original variant of this method solves the MCIS problem. It involves constructing a *modular product graph* from the two molecular graphs, which represents the compatibility of their atom pairs (hence it is also known as the *compatibility graph*). Let $G_1 \times G_2$ denote the modular product graph of two molecular graphs G_1 and G_2. A node in $G_1 \times G_2$ corresponds to each pair of atoms (u_1, u_2), where u_1 is an atom in G_1, u_2 is an atom in G_2, and they are of the same label. Two distinct nodes (u_1, u_2) and (v_1, v_2) of $G_1 \times G_2$ are connected by an edge if and only if they are compatible, that is, a mapping of a common induced subgraph can contain both atom pairs at the same time. This means that $u_1 \neq v_1$, $u_2 \neq v_2$, and either u_1 is not adjacent to v_1 and u_2 is not adjacent to v_2, or both are adjacent, and the connecting bonds are of the same type. Consequently, a one-to-one correspondence exists between the cliques of $G_1 \times G_2$ and the atom mappings that define common induced subgraphs of G_1 and G_2. The size of a clique is obviously equal to the atom count of the corresponding common induced subgraph. Therefore, finding an (approximate) MCIS of two graphs G_1 and G_2 can be reduced to finding an (approximate) maximum clique in $G_1 \times G_2$ (see, e.g., [158, 166, 204]).

As we are interested in the MCES of molecular graphs instead of the MCIS, the above approach is applied to the line graphs of the original graphs. (Recall the concept and properties of line graphs from Section 2.11.) For two molecular graphs G_1 and G_2, every common subgraph (not necessarily induced) that does not have isolated atoms corresponds to a common induced subgraph of the line graphs $L(G_1)$ and $L(G_2)$. Conversely, it can be shown based on Theorem 2.9 that a common induced subgraph of $L(G_1)$ and $L(G_2)$ has a corresponding common subgraph of G_1 and G_2 (without isolated atoms) unless a ΔY *exchange* occurs [186] (see details below). Obviously, the edge count of the common subgraph of G_1 and G_2 is equal to the node count of the corresponding common induced subgraph of $L(G_1)$ and $L(G_2)$. Furthermore, we can completely ignore common subgraphs having isolated atoms since the bond count is to be maximized in the case of MCES search. Consequently, finding an (approximate) MCES of two graphs G_1 and G_2 can be reduced to finding an (approximate) MCIS of $L(G_1)$ and $L(G_2)$, that is, finding an (approximate) maximum clique in $L(G_1) \times L(G_2)$, provided that ΔY exchange does not occur (see, e.g., [86, 158, 203, 204]).

Figure 4.5 demonstrates how the modular product graph is built from two molecular graphs when solving the MCES problem (the example is taken from [203, 204]). G_1 and G_2 are first transformed into their line graphs $L(G_1)$ and $L(G_2)$. In the line graphs, the label of a node is the type of the corresponding bond along with the types of the atoms it connects. Furthermore, the label of an edge in the line graphs is the type of the common atom of the two adjacent bonds. The product graph contains a node for each pair (e_1, e_2), where e_1 is a node in $L(G_1)$, e_2 is a node in $L(G_2)$, and their labels are the same.

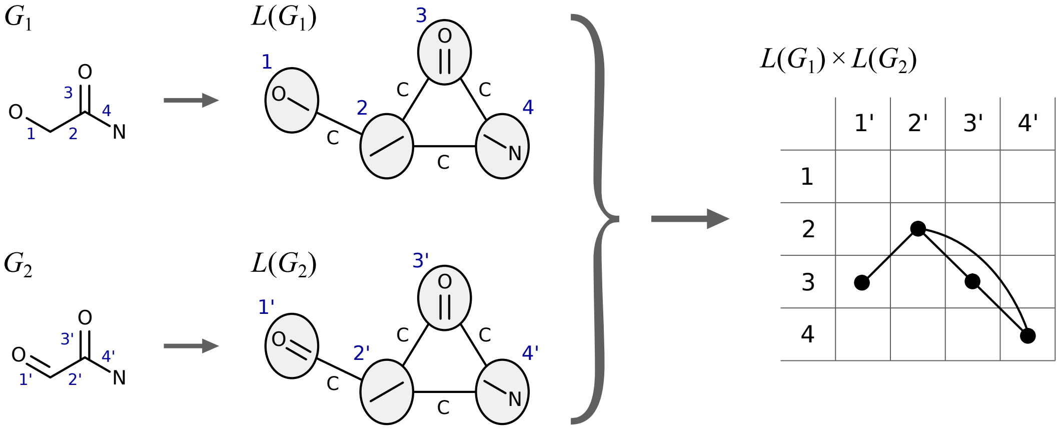


Figure 4.5: Example of building the modular product graph for clique-based MCES algorithms. The product graph contains two maximal cliques: the first one is composed of the nodes $(3, 1')$ and $(2, 2')$, and the second one is composed of the nodes $(2, 2')$, $(3, 3')$, and $(4, 4')$. That is, the maximum clique is the second one. This clique corresponds to the common subgraph made up of edges 2, 3, and 4 in G_1 and edges $2'$, $3'$, and $4'$ in G_2.

ΔY exchange

As discussed in Section 2.11, K_3 (or Δ graph) and $K_{1,3}$ (or Y graph) have isomorphic line graphs but are not themselves isomorphic. Therefore, an algorithm that operates on line graphs may erroneously match these different subgraphs, which is called a ΔY exchange. This problem becomes apparent when projecting the mapping of the line graphs onto the original graphs as it is impossible to create a consistent node mapping from the edge mapping if a ΔY exchange occurred. Luckily, ΔY exchange is very rare when molecular graphs are concerned since they rarely contain a Δ subgraph. For reference, in the PubChem Compound database [39, 185], only about 4% of the molecular graphs contain a Δ subgraph. And even in these cases, ΔY exchange can only occur if the result contains a Δ subgraph as a separate connected component (due to Theorem 2.9).

The usual way of handling this phenomenon is to continuously check whether a ΔY exchange occurs during search [86, 204]. If an exchange is discovered, the algorithm can eliminate it, for example, by selecting a different bond pair instead of the one that caused the ΔY exchange to occur. This approach is easier to implement in some algorithms and harder in others, but the performance usually suffers (e.g., the atom–atom mapping have to be maintained in addition to the bond–bond mapping during search).

Our implementations apply another approach. ΔY exchanges are ignored during search, but afterwards, when the atom–atom mapping is calculated, a bond is removed from the common subgraph for each ΔY exchange that occurred to ensure a consistent mapping. This is a simple and efficient solution for the problem, but it can yield suboptimal results even for small inputs, so it cannot be used in an exact MCES algorithm. An example is shown in Figure 4.6. An algorithm using this approach would find no difference in size between the results shown in Figure 4.6(a) and Figure 4.6(b), so it may return the former one, which contains a ΔY exchange. In this case, a bond, for example the one indicated by a dotted red line, must be removed afterwards, and the final result is not optimal.

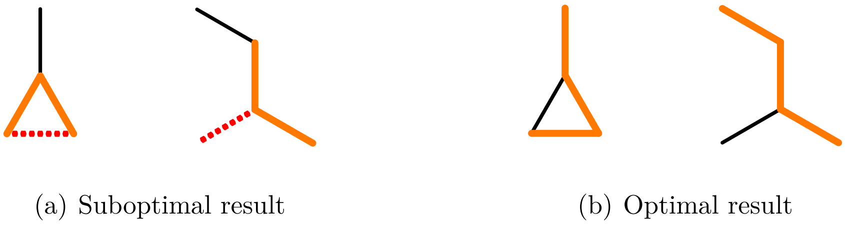

(a) Suboptimal result (b) Optimal result

Figure 4.6: Example showing the disadvantage of detecting ΔY exchange after MCES search. The removed bond is indicated by a dotted red line.

Maximum clique algorithm

Our implementation of the clique-based MCES algorithm uses the heuristic method of Grosso, Locatelli, and Pullan [133] for finding a maximum clique in the modular product graph. This is an iterated local search algorithm comprising basic movement operations and additional heuristics that drive the search process.

The movement operations can be either *add* moves or *swap* moves. An add move augments the current clique with an additional node that is adjacent to all nodes in the clique, so it results in a larger clique. Add moves are applied whenever possible, in order to always obtain maximal cliques. In contrast, a swap move adds an additional node to the current clique that is not adjacent to exactly one node in the clique, and then removes this one node. That is, the clique is transformed to another one, which has the same size but with a partially different location within the input graph.

These movement operations are applied iteratively in order to perform local neighborhood search among cliques. In addition, effective heuristics are applied to bias the search space and to restart the local search from another location of the input graph. A penalty value is assigned to each node in the graph, which is increased whenever the node is involved in a maximal clique found by the algorithm. For each add and swap move, a candidate node with minimum penalty is selected (ties are broken randomly). That is, the node penalties are used for diversification of the search process. Finally, a restart rule is applied to regularly perform a larger modification on the current clique, for example, by adding a random node and removing all other nodes that are not adjacent to this new one. A global parameter of the algorithm determines the total number of iterations (restarts) to be performed. That is, it specifies a trade-off between running time and accuracy (the expected approximation ratio of the results). By default, this parameter is set to 1000.

Despite its simplicity, this algorithm is proved to be highly efficient and robust in practice [133]. Moreover, it allows us to inject further heuristics in order to prefer certain nodes or edges of the input graph, which we exploited in our improvements (see later).

4.2.2 Build-up algorithm

Kawabata [151] developed a greedy heuristic algorithm for MCIS search, which is called the build-up method. The notion of topologically constrained common subgraphs and the topological heuristic scores described along with the algorithm are of special interest since they have the potential to yield results that are chemically more relevant.

The original version of the build-up algorithm solves the MCIS problem. It maintains an ordered list of common subgraphs, represented by atom mappings, which are gradually augmented. The algorithm begins with trivial mappings having only one atom pair. These mappings are sorted by a scoring function, and only the first K mappings are kept. In the next step, each mapping is extended with each feasible atom pair, and the K best extended mappings are selected again according to the scoring function. This method is iterated until none of the maintained mappings can be extended.

The scoring function consists of multiple components. The neighbor score measures, for each pair of atoms in the mapping, the dissimilarity of their immediate neighbor atoms in the two molecules. Another score, originally introduced by Morgan [180], is defined as the difference in the extended connectivity values of the matched atoms. Furthermore, when a topologically constrained MCIS is to be found, Kawabata also suggests an additional score comparing the topological distances of the matched atoms in the two chemical graphs. In this study, however, we do not consider topologically constraints, so this latter score is discarded.

The build-up algorithm evaluates each atom mapping with the sum of these scores. A mapping is considered to be redundant if the current list already contains another mapping that includes the same number of atoms for each atom type and has the same score.

As discussed previously, an MCIS algorithm can be adapted to the MCES problem by considering the line graphs instead of the original input graphs. We implemented such a modified version of the build-up algorithm, using $K = 5$. While it initially performed significantly worse than the clique-based method, after being augmented with heuristics described later, this algorithm turned out to be a viable alternative, in terms of both running time and accuracy. These results are presented in Section 4.4.

4.2.3 Upper bound calculation

An important aspect of MCES search is the existence of various upper bound calculation methods [204, 206]. These methods are generally much faster than MCES algorithms, even heuristic ones. Therefore, they can effectively be used for screening pairs of graphs and filtering out those that are guaranteed to have a small MCES. This is useful, for example, to accelerate the search for molecules with large common subgraphs during similarity search or clustering. Furthermore, upper bounds can also be used to evaluate the accuracy of a common subgraph found by an algorithm (see later).

We implemented both upper bound calculation methods described by Raymond et al. [204]. The first one groups the bonds in both molecular graphs by their type along with the type of their atoms, and the minimum count is summed for each group. The more accurate method finds a maximum-weight matching between the atoms of G_1 and G_2, where the weight of assigning two atoms equals to the maximum number of pairs of their incident bonds that can be matched in a common subgraph (with respect to atom and bond types). The maximum weight is divided by two in order to obtain an upper bound on the bond count of the MCES.

4.3 Improvements and heuristics

This section details the methods that we applied to improve upon the original algorithms in regard to accuracy, running time, memory usage, and the chemical relevance of the results [1].

Most of the available literature concentrates on pruning heuristics or reducing the search space [20, 110, 203, 204, 206, 227]. However, it is reasonable to assume that heuristic algorithms benefit less from such techniques. Therefore, we concentrate more on making the algorithms faster and better at guessing which part of the search space to explore.

4.3.1 Representation of the modular product graph

One of the applied improvements concerns the representation of the modular product graph in the clique-based algorithm. The size and density of this graph are among the main disadvantages of the clique-based approach as they impact both memory usage and running time. However, we argue that this graph is so dense that the complement graph is to be stored instead, as an adjacency list. The same idea is used in [134] to improve the efficiency of the Bron–Kerbosch algorithm, which finds all maximal cliques.

We exploit the properties of molecular graphs, namely, that they are very sparse and each atom has a strictly limited degree. In a random selection of 10^6 molecules from the PubChem Compound Database [39, 185], the maximum degree of an atom was found to be 6, the average degree was 2.15, and the average of the maximum degree for each molecule was 3.31 (hydrogen atoms are ignored).

Let d denote the maximum degree of an atom in the input molecular graphs G_1 and G_2. In their line graphs, the maximum degree of a node corresponding to a bond uv is $2(d-1)$ since both u and v may have $d-1$ other bonds connected to them. Let the number of bonds in G_1 be m_1 and the number of bonds in G_2 be m_2. The modular product graph $L(G_1) \times L(G_2)$ has at most $m_1 m_2$ nodes.

For a node e of a line graph, let $N(e)$ denote the union of $\{e\}$ and the set of nodes adjacent to e. If two distinct nodes (e_1, e_2) and (f_1, f_2) of the product graph $L(G_1) \times L(G_2)$ are not adjacent (incompatible), then at least one of $f_1 \in N(e_1)$ and $f_2 \in N(e_2)$ holds. That is, two bonds must be equal or adjacent to each other to cause incompatibility. Therefore, the number of nodes that are not adjacent to (e_1, e_2) in the modular product graph is at most

$$\begin{aligned} |N(e_1)|m_2 + |N(e_2)|m_1 &\leq (2(d-1)+1)m_2 + (2(d-1)+1)m_1 \\ &= O(d(m_1+m_2)). \end{aligned} \tag{4.1}$$

Since d is a small constant for molecular graphs (usually at most 4), the total number of edges in the complement of the modular product graph is $O(m_1 m_2(m_1+m_2))$. In contrast, the number of edges in the original product graph is $O((m_1 m_2)^2)$. So this is a reduction in memory usage from $O(m^4)$ to $O(m^3)$, where $m = \max\{m_1, m_2\}$.

This change also significantly improves the running time because it allows the modular product graph to be built in $O(m^3)$ time instead of $O(m^4)$ time. This is rather important as building the product graph is a dominant phase of the algorithm, which often accounts for the majority of the total running time, especially in the case of large input graphs.

Furthermore, the new representation also improves the performance of the clique search algorithm because it accesses the underlying graph in only one way. When a node is added to or removed from the current clique, the nodes that are not adjacent to it are traversed, and the internal data structures are updated. This operation is made much faster with the new representation since the nodes to be traversed are explicitly stored in a list, so it is not necessary to iterate over an entire row in an adjacency matrix.

Our benchmark tests verified the effectiveness of this improvement in terms of both memory usage and running time. In accordance with our calculations, experimental results showed an order of magnitude difference in memory requirements.

4.3.2 Early termination

Early termination is a common heuristic in optimization methods: the algorithm is terminated when it is guaranteed that an optimal solution has already been found. In our case, such a guarantee can be provided by calculating an upper bound on the size of the MCES. We implemented this heuristic in our clique-based algorithm using the more accurate upper bound calculation method described in Section 4.2.3. Experimental results showed that this improvement makes the algorithm significantly faster in many cases (see Section 4.4), although both the common subgraph and the calculated upper bound must be optimal to allow early termination. In the build-up algorithm, however, we do not apply this heuristic because the amount of computation we could save in that method is negligible.

4.3.3 Connectivity heuristic

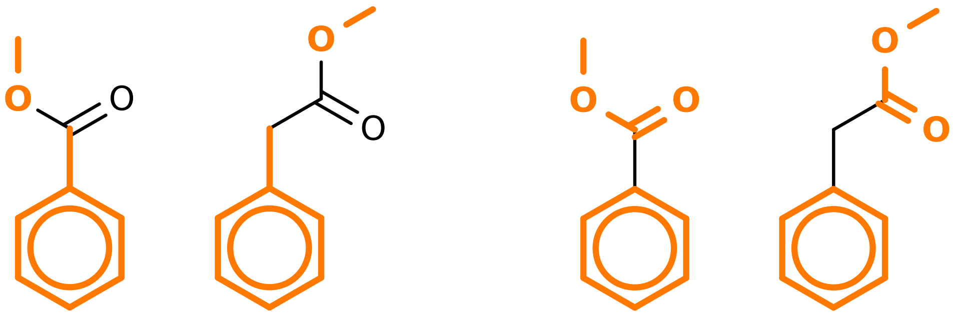
An important heuristic of MCES algorithms is to prefer atoms and bonds that are connected to the current common subgraph when it is extended. This idea is applied in, for example, the backtracking algorithm of Cao et al. [53], and it turned out to substantially improve our algorithms as well. This heuristic is natural when the connected MCES problem is to be solved, but it greatly improves the results even in the disconnected case. Moreover, it can also decrease the number of connected components in the common subgraphs found by the algorithms, which is usually beneficial.

We implemented this heuristic as follows. When the current mapping is to be extended, a bond pair (e_1, e_2) is preferred if there is at least one bond pair (f_1, f_2) in the mapping such that e_1 and f_1 are adjacent (which also implies the adjacency of e_2 and f_2 due to the compatibility requirement). This is straightforward to be implemented in the build-up method as it operates directly on the input graphs. In the case of the clique-based algorithm, however, we must distinguish *C-edges* and *D-edges* in the modular product graph (cf. [151, 158]) to implement the heuristic. C-edges connect nodes for which the corresponding bond pairs are adjacent in both molecular graphs (connected), while D-edges connect nodes for which the corresponding bond pairs are not adjacent (disconnected). As the number of C-edges is strictly limited, we can store for each node the list of adjacent nodes connected to it with C-edges. Using the notations introduced previously, a node can have at most $(2(d-1))^2 = O(d^2)$ such neighbors, that is, each adjacent bond in G_1 paired with each adjacent bond in G_2. (Recall that d is a small constant, typically at most 4.) Considering the improved representation discussed in Section 4.3.1, we store for each node of the product graph, both the non-adjacent nodes and the nodes that are connected to it with C-edges (while D-edges are not stored explicitly).

An additional question is how to consider the number of C-edges connecting a candidate node to the current clique, which we call the *connectivity score*. Our experiments show that it is generally beneficial to select the next candidate node having the highest connectivity score, so our implementation works this way.

In particular cases, however, the greedy use of this heuristic leads to suboptimal results, as illustrated in Figure 4.7. Nevertheless, we found that the local search process of the clique detection algorithm discussed in Section 4.2.1 can effectively correct such suboptimal results by swapping nodes of the product graph in the current clique (i.e., swapping bond pairs in the mapping). In contrast, the build-up algorithm cannot make such corrections: after being added to a mapping, bond pairs are never removed.

When the connected MCES problem is to be solved, this heuristic is applied in the same way, but we keep only the largest connected component of each common subgraph, and their sizes are compared accordingly. For example, in the case of the molecular graphs shown in Figure 4.7, the optimal connected MCES is derived from the suboptimal solution of the disconnected MCES problem because its largest component contains an additional bond compared to the largest component of the optimal disconnected MCES.



(a) Suboptimal result

(b) Optimal result

Figure 4.7: The greedy use of the connectivity heuristic can lead to suboptimal results in the case of disconnected MCES search.

4.3.4 ECFP-based heuristic

A novel heuristic we developed to improve MCES algorithms is based on the generation method of *extended connectivity fingerprints* (ECFPs) [211], which are widely used for similarity search of molecular graphs. According to our experiments, the ECFP-based heuristic substantially enhances the accuracy of both the clique-based and build-up algorithms, especially when it is combined with the connectivity heuristic.

An ECFP hash code of an atom in a molecular graph is an integer hash code representing the environment of the atom with a given radius. The environment of radius r for a given atom is taken to mean the subgraph induced by the atoms at a distance of at most r from that atom. The ECFP hash of radius 0 represents the type of the atom. Then the ECFP hash of radius $r+1$ for a given atom is computed in the following way. For each adjacent atom, its ECFP hash of radius r is combined with the type of the bond leading to it. These combined values are then hashed in an order-independent way to produce the ECFP hash code of radius $r+1$ of the considered atom. As a result, given two atoms, their environments of radius r can be isomorphic subgraphs only if their ECFP hash codes of radius r are the same. This generation procedure is based on the classic method devised by Morgan [180], but ECFP hash codes also incorporates the types of atoms and bonds, not only the degrees.

The goal of using ECFP hash codes in MCES algorithms is to prefer matching atoms and bonds that have large isomorphic environments in the two input graphs. Figure 4.8 illustrates this concept. Similar molecules usually contain relatively large subgraphs that are isomorphic to each other, and this heuristic is very effective in finding and matching such subgraphs. We define the ECFP score of a pair of atoms (u_1, u_2), denoted as $s_{(u_1,u_2)}$, to be the largest r such that for each $r' \leq r$, the ECFP hash codes of radius r' are the same for u_1 and u_2. MCES algorithms, however, operate on bonds instead of atoms, so we need to combine the scores of atom pairs to obtain suitable scores for bond pairs. For a bond pair (e_1, e_2), where $e_1 = u_1v_1$ is a bond in G_1 and $e_2 = u_2v_2$ is a bond in G_2, we define the ECFP score as

$$s_{(e_1,e_2)} = \max\{\min\{s_{(u_1,u_2)}, s_{(v_1,v_2)}\}, \min\{s_{(u_1,v_2)}, s_{(v_1,u_2)}\}\}. \tag{4.2}$$

There are two ways to match the atoms of a bond pair to each other, and we use the score of the better one.

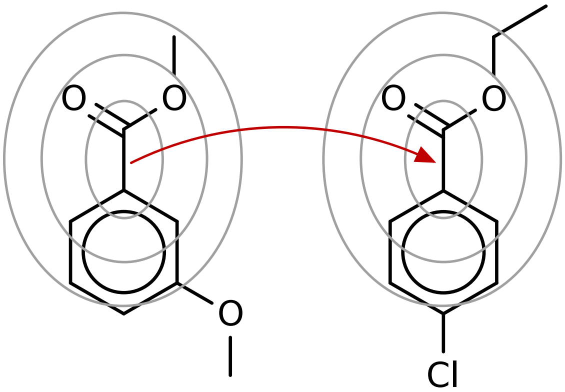


Figure 4.8: The concept of the ECFP-based heuristic.

In both MCES algorithms, we prefer the selection of bond pairs with higher ECFP scores. This heuristic turned out to be really effective, but it sometimes drives the search process towards a local maximum. In our clique-based algorithm, we attempt to avoid getting stuck in such a local maximum by gradually decreasing the preference of the bond pairs that were selected in previous iterations. Recall that the implemented clique search algorithm uses node penalties to drive the search process (see Section 4.2.1). We used these penalties to inject the ECFP scores into the algorithm. Instead of initializing node penalties uniformly to zero, we use the negative of the corresponding ECFP score as initial penalty for each node. That is, ECFP-based node selection is strongly preferred at the beginning, but as the clique search algorithm advances, the penalties of the selected nodes are increased at each iteration, so the selection strategy is gradually diversified.

4.3.5 Mapping optimization

Standard MCIS/MCES algorithms do not consider anything but the size of the common subgraphs as the measure of their quality. However, the topological features determined by the mapping, such as the orientation of the common subgraph in the input graphs, are essential in certain applications, for example, reaction mapping and molecule alignment. These aspects are rarely studied in the available literature. Notable examples are the work of Kawabata [151], which introduces the notion of topologically constrained MCIS; the method of Stahl et al. [224], where similarity scores are adjusted with respect to the arrangement of the connected components of the common subgraph; and the *Small Molecule Subgraph Detector* (SMSD) toolkit [201], which ranks common subgraphs according to multiple chemically relevant criteria. In this work, we present a general method that can effectively handle typical cases.

A straightforward method for finding an appropriate mapping would be to enumerate all possible mappings of the common subgraph and to evaluate each of them. This can be achieved by running a subgraph matching algorithm to find all possible mappings between the common subgraph G_c and G_1 as well as between G_c and G_2. Then mappings between G_1 and G_2 can be constructed by composing each pair of $G_c \to G_1$ and $G_c \to G_2$ mappings. However, this method is not feasible in many cases because of the large number of mappings that would be produced (due to symmetry).

A more effective approach is based on the observation that in all examples we studied, if a mapping was found to be inadequate, the problem was with atoms at the edge of the common subgraph, that is, matched atoms that are adjacent to unmatched atoms in G_1 or G_2. The essence of our idea is to consider only those mappings where these atoms are matched differently. First, an initial mapping for the common subgraph is to be found. With respect to this mapping, we call an atom in G_1 or G_2 *interesting* if it is part of the common subgraph but has an adjacent atom that is not part of the subgraph. Figure 4.9 shows an example. If possible, it is usually preferred to match interesting atoms of G_1 to interesting atoms of G_2.

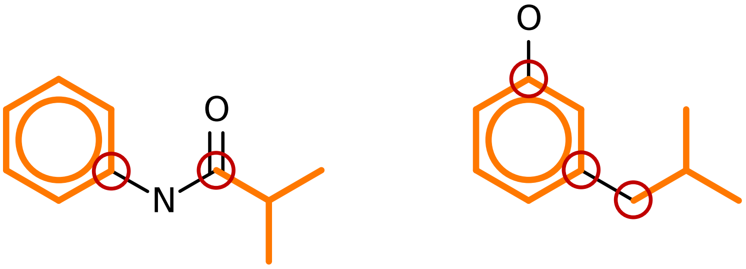


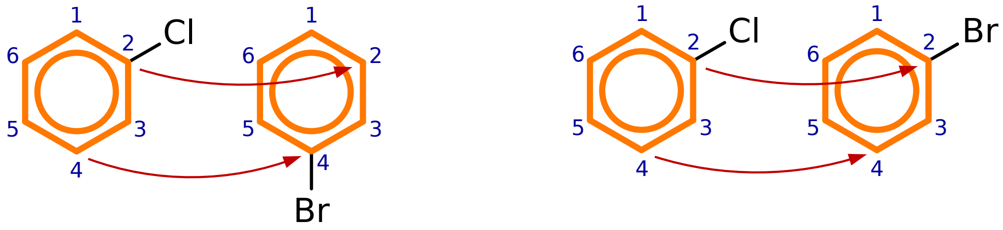
Figure 4.9: Interesting atoms for mapping optimization. The MCES is highlighted in orange, and the interesting atoms are circled in red in both molecular graphs. The two components of the common subgraph are symmetric, so multiple mappings exist. It is preferred to match interesting atoms to interesting atoms (if possible).

Focusing on these interesting atoms, we can usually reduce the number of mappings to be considered. In the clique-based algorithm, we implemented a mapping optimization heuristic using this idea. First, an (approximate) MCES and an initial mapping are found as usual. A bond in G_1 or G_2 is called interesting if it is part of the common subgraph and one of its atoms is interesting. Then, we add a pair of interesting bonds to the mapping, remove their connected components (so, for example, a whole chain can be reversed), and run a local search to increase the size of the common subgraph again. If a larger subgraph is found, or if the size is the same as before, but the new mapping is to be preferred according to a general scoring function, then it is kept instead of the previous mapping. Each pair of interesting bonds is added this way to find the best mapping.

We experimented with different functions for evaluating mappings and found the following one to be effective and robust. Using the notations $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a mapping defining a common subgraph is a bijective function between a subset of E_1 and a subset of E_2. Let the functions $deg_1 : E_1 \to \mathbb{N}$ and $deg_2 : E_2 \to \mathbb{N}$ assign the number of adjacent bonds to each bond in G_1 and G_2, respectively. Furthermore, let $dist_1 : E_1 \times E_1 \to \mathbb{N}$ and $dist_2 : E_2 \times E_2 \to \mathbb{N}$ assign the distance to each bond pair in G_1 and G_2, respectively. If two bonds are not connected with a path, their distance is defined to be a large finite constant, for example, $\max\{|E_1|, |E_2|\}$. The score of a mapping M is then defined as

$$s_M = \sum_{(e_1,e_2)\in M} |deg_1(e_1) - deg_2(e_2)| \\ + \sum_{(e_1,e_2)\in M} \sum_{(f_1,f_2)\in M} (dist_1(e_1, f_1) - dist_2(e_2, f_2))^2. \tag{4.3}$$

The lower the score of a mapping is, the better we consider it among mappings of the same size. The first part of the expression sums the differences in degree in G_1 and G_2 for each bond pair in the mapping. This allows us to match the bonds of the common subgraph in a way that adjacent parts outside the subgraph are also aligned. A simple example is to match two rings with different substituents connected to them in the input graphs, as shown in Figure 4.10. The second part of the score is the sum of squares of the differences in distance between pairs of matched bonds. This allows us to optimize the alignment of the connected components of the common subgraph with respect to each other in G_1 and G_2. Figure 4.11 illustrates the importance of this.



(a) Suboptimal mapping

(b) Preferred mapping

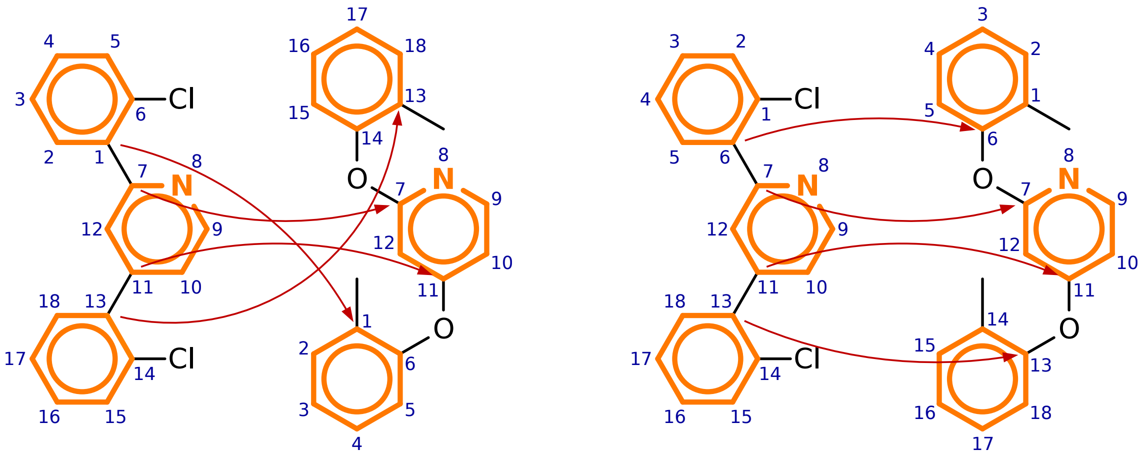
Figure 4.10: Example of mapping optimization. It is preferred to match the carbon atoms that are adjacent to the halogen atoms (Cl and Br).

An important additional benefit of this mapping optimization procedure is that it can also yield a larger common subgraph and thereby improves the accuracy of the algorithm. A simple example is depicted in Figure 4.12. Because of its inherent inexactness, the first phase of our algorithm often finds a solution that is suboptimal in size. However, when we search for better mappings of this common subgraph, we can discover a mapping that can be extended to a larger one by local search.

4.3.6 Preserving rings

In applications such as clustering and alignment, a frequent request is to consider only those common subgraphs in which rings (cycles of the graphs) are not broken [134]. That is, if a matched bond is part of a ring in G_1 or G_2, then it is required to be part of a ring in the common subgraph as well. This restriction is straightforward to implement in MCES algorithms using a subgraph enumeration approach (e.g., [71, 158]), but it is more involved in the case of, for example, the maximum clique approach.

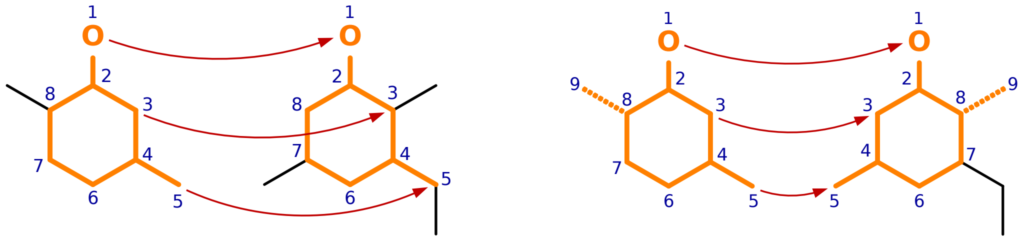
A bond in G_1 or G_2 is called a *ring bond* if it is part of at least one ring. Our algorithms provide three options for handling ring bonds (the same as in [134]), which are illustrated



(a) Suboptimal mapping

(b) Preferred mapping

Figure 4.11: Example of mapping optimization. It is preferred to match atoms that are close to each other in one molecular graph to atoms that are close to each other in the other molecular graph.



(a) Suboptimal result

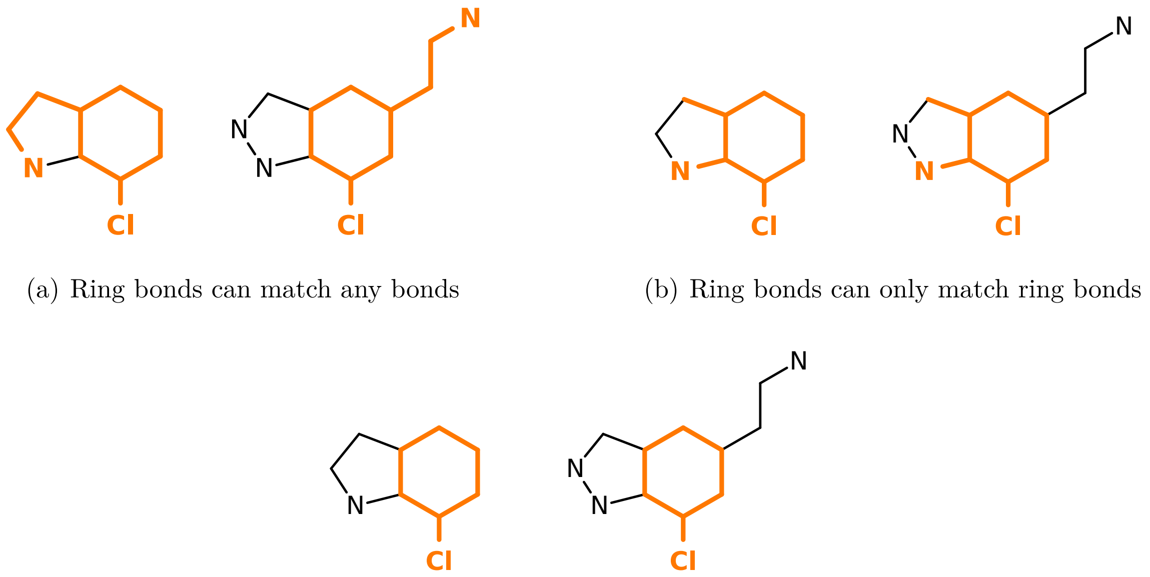
(b) Optimal result

Figure 4.12: The additional benefit of mapping optimization. Finding an appropriate mapping for a suboptimal solution can allow us to extend the common subgraph. The newly added bond is denoted by a dotted line in both molecular graphs.

in Figure 4.13. The first option is to ignore ring membership, so a ring bond is allowed to match any bond with the same label; the second one is to allow ring bonds to match ring bonds only, but rings do not need to completely match; while the third one is preserving rings, that is, not allowing rings to match partially.

The first option is the default behavior of the algorithms, and the second one is also trivial to implement by incorporating topology information into the bond labels. The third option is, however, more complicated. We implemented it by limiting the search space according to the second option and then removing broken rings that appear in the common subgraphs.

The rationale for this approach is that optimizing for the size of the common subgraph already favors matching full rings of the same size, and by disallowing matching of ring bonds and non-ring bonds, we usually end up with a good approximation of the desired result. As shown in Figure 4.14, this approach can yield suboptimal results, but the

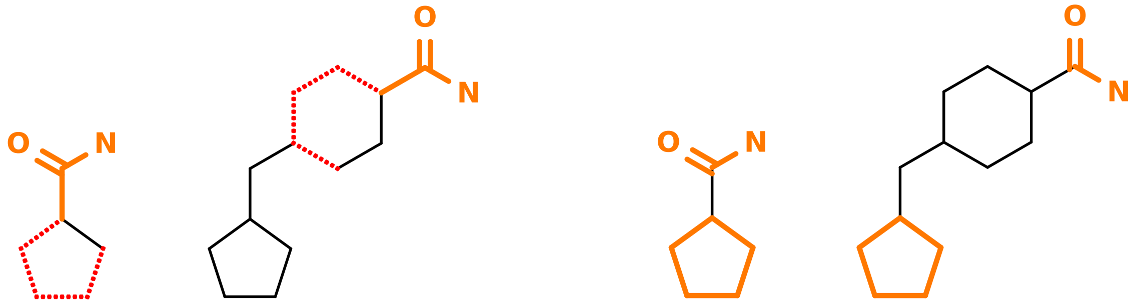


(a) Ring bonds can match any bonds

(b) Ring bonds can only match ring bonds

(c) Partial ring matching is not allowed

Figure 4.13: Examples showing the MCES of the same pair of molecular graphs with different ways of handling ring bonds.



(a) Suboptimal result

(b) Optimal result

Figure 4.14: Example where removing bonds of partially matched rings results in a suboptimal solution. On the left side, the colored bonds represent the common subgraph found by the algorithm according to the second option, and the dotted red bonds are the ones removed afterwards. The optimal result is shown on the right side.

iterated local search method of the clique-based algorithm often finds an optimal solution even in such cases. To further increase the accuracy of the algorithms, the bond topology information is also incorporated into the ECFP hash codes (see Section 4.3.4) in the case of the second and third options of ring handling.

We also remark that the request to avoid breaking rings usually does not extend to large rings (called macrocycles). Therefore, our implementations consider only rings within a size limit, which is set to 8 by default.

4.4 Experimental results

Extensive benchmark tests were performed to evaluate the presented algorithms and heuristics [1]. This section contains a selection of measurements and results that demonstrate the effects of the developed improvements as well as the overall performance of the algorithms. They were also compared with the corresponding method of the Indigo toolkit [92] and the KCOMBU code of Kawabata [150, 151].

4.4.1 Test setup

The presented algorithms were implemented in Java programming language as part of the JChem software suite of ChemAxon [55]. The experiments were conducted using JChem 14.10.6 on a PC with an Intel Core i7-4800MQ 2.7 GHz CPU and 16 GB of RAM, running a 64-bit Microsoft Windows 7 Professional SP1 operating system. The codes were compiled and run using Java 1.8.0_20.

Unlike several other graph optimization problems, the MCIS and MCES problems do not have standard sets of benchmark graphs. Therefore, we collected a test suite of nine data sets containing various molecular graphs, which are available at [7]. Table 4.1 summarizes the main properties of these data sets.

	K	N	n_{min}	n_{max}	n_{avg}	m_{min}	m_{max}	m_{avg}	s_{min}	s_{max}	s_{avg}
NCI1	32	496	24	24	24.0	26	30	27.6	0.43	1.00	0.69
NCI2	16	120	37	46	42.5	41	51	46.8	0.36	1.00	0.59
NCI3	16	120	71	78	74.8	71	84	77.9	0.18	1.00	0.52
NCI4	12	66	106	123	115.6	112	125	117.9	0.44	1.00	0.70
NCI-S	39	741	20	45	30.4	22	50	33.4	0.35	1.00	0.64
CAH2	116	6670	8	35	20.0	8	37	21.0	0.18	1.00	0.48
CDK2	154	11781	9	36	24.7	10	43	27.2	0.16	1.00	0.55
NEU	15	105	20	28	21.7	20	29	21.8	0.42	1.00	0.69
NCI-I	40	20	8	241	102.8	8	264	112.1	0.70	0.87	0.83

Table 4.1: Properties of the benchmark data sets. K denotes the number of input graphs. N denotes the number of MCES problem instances: $K/2$ for NCI-I and $K(K-1)/2$ for the other data sets. $n_{min/max/avg}$ and $m_{min/max/avg}$ denote the min./max./average number of non-hydrogen atoms and bonds between them, respectively, in the K molecular graphs. $s_{min/max/avg}$ denote the min./max./average similarity score among the N molecule pairs.

Some of these sets were composed for this study by selecting molecules from a public database provided by the National Cancer Institute (NCI) [184]. The sets NCI1, NCI2, NCI3, and NCI4 are intended to represent different ranges of molecule size as the efficiency and accuracy of MCES algorithms strongly depend on that. However, within each of these sets, the molecular graphs are similar to each other in both size and characteristics. Another data set, called NCI-S, consists of those molecular graphs from the NCI database that contain both *benzenesulfonamide* and *indole* as a subgraph. This set is more diverse in size than the previous ones, but it also contains similar graphs since they are ensured to have a relatively large common subgraph (both benzenesulfonamide and indole contain

10 bonds between non-hydrogen atoms). In the case of these data sets, MCES search was performed for each pair of molecules, so $K(K-1)/2$ problem instances were derived from a set of K molecules.

In addition, we also used three of the five data sets that were introduced in the experimental study of the KCOMBU code [151]. They are collections of protein ligands that can be downloaded from the Worldwide Protein Data Bank (wwPDB) [25, 26] according to the list files that are available on the web site of KCOMBU [150]. Our test suite includes the data sets CAH2, CDK2, and NEU, which were processed by pairwise comparison like the previous ones. However, in accordance with Kawabata's study [151], we ignored bond types in the case of these molecular graphs.

Finally, we constructed a special data set, denoted as NCI-I, that contains molecule pairs of gradually increasing size. First, two large and very similar molecular graphs were selected from the NCI database (both containing about 240 non-hydrogen atoms). Then we successively removed small parts of both molecules. The pairs of similar graphs obtained thereby can be used, in the reverse order of their construction, as MCES problem instances of steadily increasing size. Despite its artificial construction, this data set also consists of reasonable chemical graphs, and it allows easier evaluation and clear visualization of the experimental behavior of the algorithms as a function of the input size.

Table 4.1 presents basic statistical data for these test sets. The similarity of two molecular graphs G_1 and G_2 is measured using the following Tanimoto (Jaccard) score:

$$s_{(G_1,G_2)} = \frac{m_c}{m_1 + m_2 - m_c} \, , \tag{4.4}$$

where m_1 and m_2 denote the number of bonds in G_1 and G_2, respectively, and m_c denotes an upper bound on the number of bonds in their MCES, which was computed using the more accurate upper bound calculation method (see Section 4.2.3).

4.4.2 Evaluation of heuristics

Experimental results are presented here to demonstrate the effects of the different improvements on the accuracy and running time of the implemented algorithms. Unless otherwise stated, we consider disconnected MCES search as it is the primary focus of this study. However, we also experimented with the connected case, and the developed improvements turned out to be similarly effective.

To measure the accuracy of the algorithms, we use the notion of *result size ratio*. It is defined as the ratio of the number of bonds in a common subgraph found by an algorithm to the upper bound on this number calculated using the more accurate method. (We do not consider trivial cases where the upper bound is zero, and thus the ratio is undefined.) This measure expresses a relative accuracy and makes it easier to aggregate and compare results obtained for problem instances of different sizes.

The result size ratio is always between 0 and 1, but reaching 1 is typically not possible because the upper bound is also an estimate and the exact MCES is usually not known. In fact, the ratio gives a bound on the approximation of both the MCES algorithm and the upper bound calculation.

Clique-based algorithm

We evaluated five variants of the clique-based algorithm that differed in whether the different heuristics were applied or not, but all of them used the improved representation (see Section 4.3.1). Table 4.2 reports the average result size ratio and average running time (in milliseconds) for these variants measured on eight data sets. The result size ratios are also compared in Figure 4.15. These experiments show how each heuristic improves the accuracy and that their combination works quite well: using all of the heuristics together yields significantly better results than using any one of them alone. As discussed in Section 4.3.5, the mapping optimization heuristic, beyond its primary goal, can also help to find larger common subgraphs (see Figure 4.12). Indeed, the results in Table 4.2 verify that it often improves the accuracy.

	Average result size ratio					Average running time (ms)				
	None	CONN	ECFP	MAP	All	None	CONN	ECFP	MAP	All
NCI1	0.934	0.938	0.941	0.936	0.943	14.4	13.8	14.2	15.7	11.0
NCI2	0.876	0.892	0.888	0.882	0.895	23.4	20.0	22.0	25.0	19.4
NCI3	0.875	0.893	0.894	0.884	0.907	74.1	57.3	65.0	79.3	59.2
NCI4	0.827	0.863	0.859	0.855	0.900	250.3	198.9	291.2	345.4	237.9
NCI-S	0.959	0.969	0.971	0.964	0.972	12.0	9.8	10.8	12.7	7.1
CAH2	0.942	0.943	0.942	0.942	0.943	5.0	4.9	4.9	5.3	3.6
CDK2	0.884	0.888	0.888	0.886	0.890	11.3	10.6	10.8	11.9	10.2
NEU	0.934	0.939	0.939	0.936	0.939	8.3	8.2	7.8	8.0	6.0

Table 4.2: Effects of the heuristics on the accuracy and running time of the clique-based algorithm. The columns correspond to different variants of the algorithm. None: the basic algorithm without heuristics; CONN: only the connectivity heuristic is applied; ECFP: only the ECFP heuristic is applied; MAP: only the mapping optimization heuristic is applied; All: all improvements (including the early termination heuristic) are applied. Each variant uses the improved representation of the modular product graph.

These experiments also show that the larger the input graphs are, the more significant the heuristic improvements are. Figure 4.16 also illustrates this by depicting the result size as a function of its upper bound for the problem instances defined by the NCI-I set. In the case of the largest instances, the heuristics had a great impact on the accuracy: the common subgraphs found by the improved algorithm were 15-30% larger than the ones found by the original method. Furthermore, the improvements also turned out to make the algorithm more robust.

As for the running time, Table 4.2 shows that the three main heuristics do not make the algorithm much slower. On the contrary, combined with the early termination technique, they actually make the algorithm faster in many cases. Using all heuristics together reduces the average running time on each data set. Figure 4.17 visualizes the effect of the early termination heuristic on the NCI-S set. The running time is depicted for each problem instance as a function of the input size, which is measured as the product of the bond counts of the input graphs. The cases when early termination was achieved can easily be separated from the others. Recall that the solution is also guaranteed to be optimal in

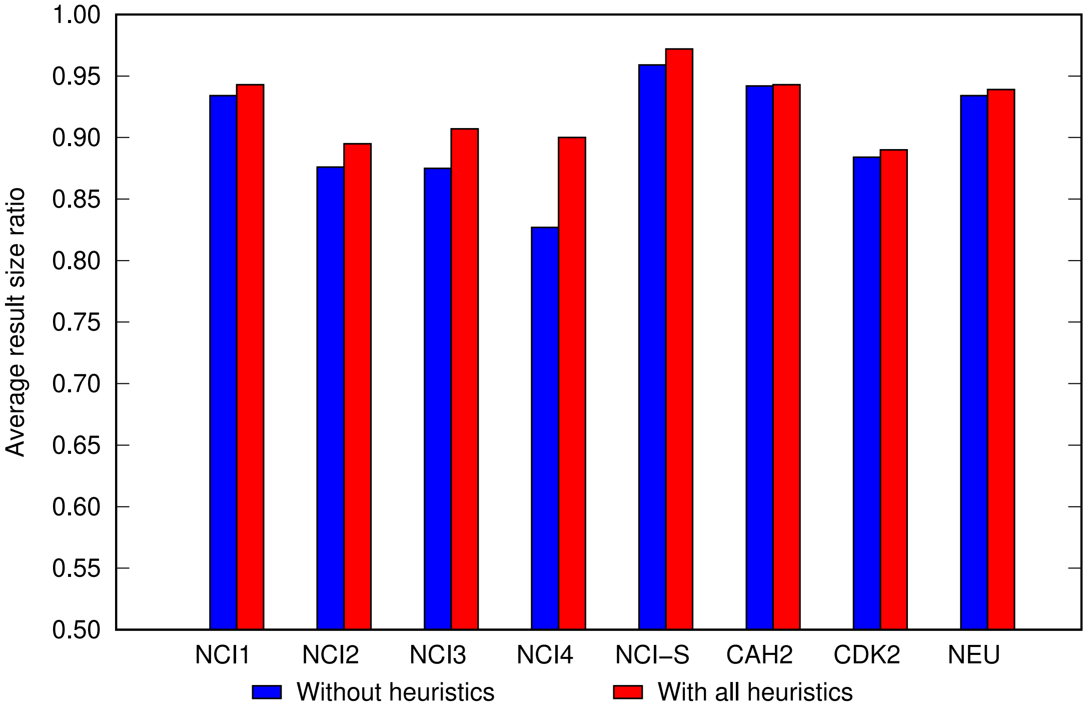


Figure 4.15: Effects of the heuristics on the accuracy of the clique-based algorithm.

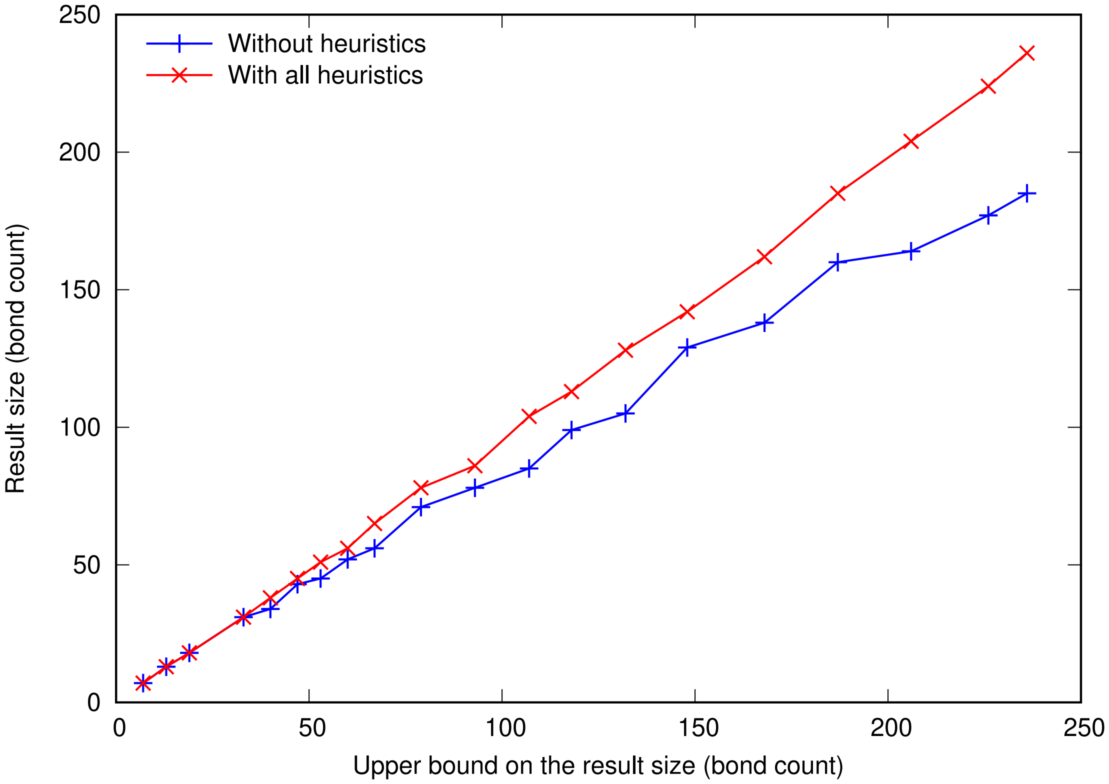


Figure 4.16: Effects of the heuristics on the accuracy of the clique-based algorithm, measured on the NCI-I data set.

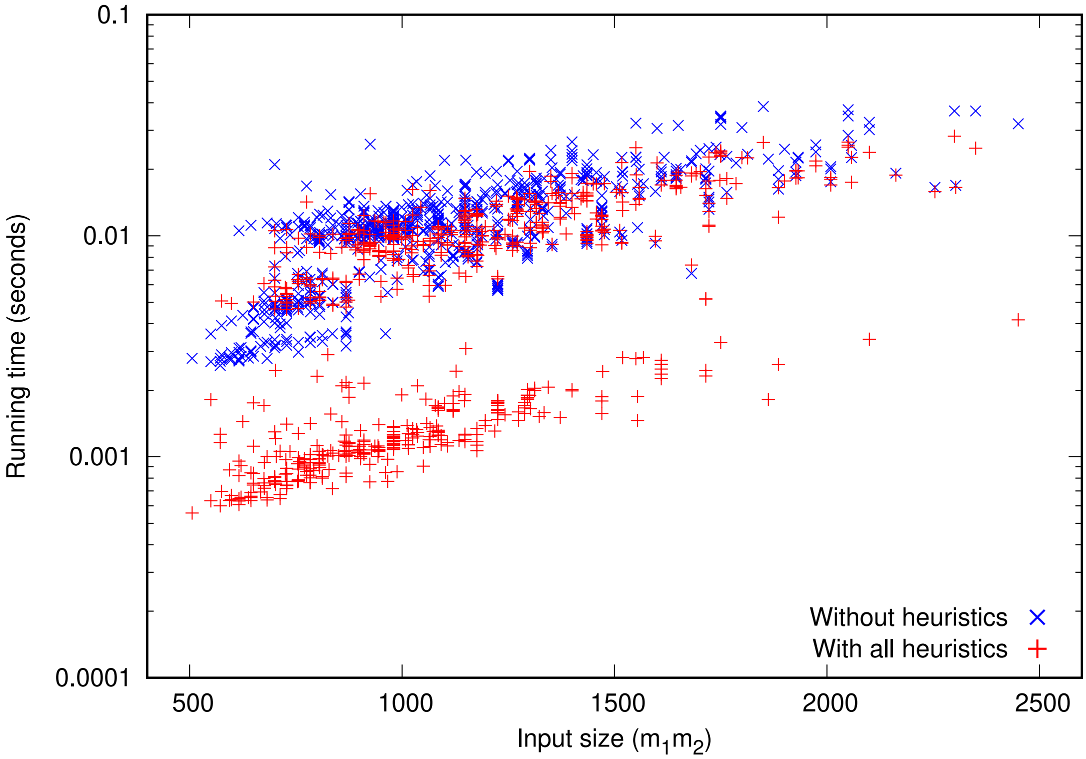


Figure 4.17: Effects of the heuristics on the running time of the clique-based algorithm, measured on the NCI-S data set.

these cases, but it may be optimal in other cases as well (provided that the calculated upper bound is suboptimal).

Build-up algorithm

Some of the presented improvements are also applied to the build-up algorithm. The connectivity and ECFP heuristics are used similarly as in the clique-based method, but we did not implement a mapping optimization heuristic for the build-up algorithm. Table 4.3 and Figure 4.18 report the benchmark results for different variants of the build-up algorithm.

An interesting observation is that the ECFP heuristic turned out to have a variable impact on this algorithm. On some data sets, it yields a minor improvement (e.g., NCI2 and CAH2) or even slightly decreases the accuracy (NCI3 and NCI4), while it results in a great improvement on other sets (e.g., NCI1, NCI-S, and NEU). These differences are probably due to the less robust behavior of the build-up approach compared with the clique-based method. The connectivity heuristic, however, consistently makes the build-up algorithm much more accurate (see Table 4.3). Furthermore, combined with this improvement, the ECFP heuristic also becomes more robust.

4.4.3 Comparison of the implemented algorithms

Besides evaluating the effects of the developed improvements, we also compared the two different algorithmic approaches. Table 4.4 and Figure 4.19 present benchmark results for

	Average result size ratio				Average running time (ms)			
	None	CONN	ECFP	All	None	CONN	ECFP	All
NCI1	0.756	0.890	0.792	0.909	4.5	3.3	5.3	3.5
NCI2	0.729	0.842	0.739	0.838	8.8	7.1	10.6	6.8
NCI3	0.761	0.859	0.755	0.880	46.9	29.9	45.9	29.8
NCI4	0.739	0.850	0.737	0.885	335.2	206.1	387.7	213.5
NCI-S	0.864	0.956	0.913	0.959	5.4	3.3	5.4	3.5
CAH2	0.833	0.907	0.842	0.912	1.1	0.7	1.2	0.7
CDK2	0.702	0.822	0.721	0.831	2.5	1.7	2.8	1.7
NEU	0.770	0.914	0.868	0.923	2.4	1.7	3.4	1.6

Table 4.3: Effects of the heuristics on the accuracy and running time of the build-up algorithm. The columns correspond to different variants of the algorithm. None: the basic algorithm without heuristics; CONN: only the connectivity heuristic is applied; ECFP: only the ECFP heuristic is applied; All: both heuristics are applied.

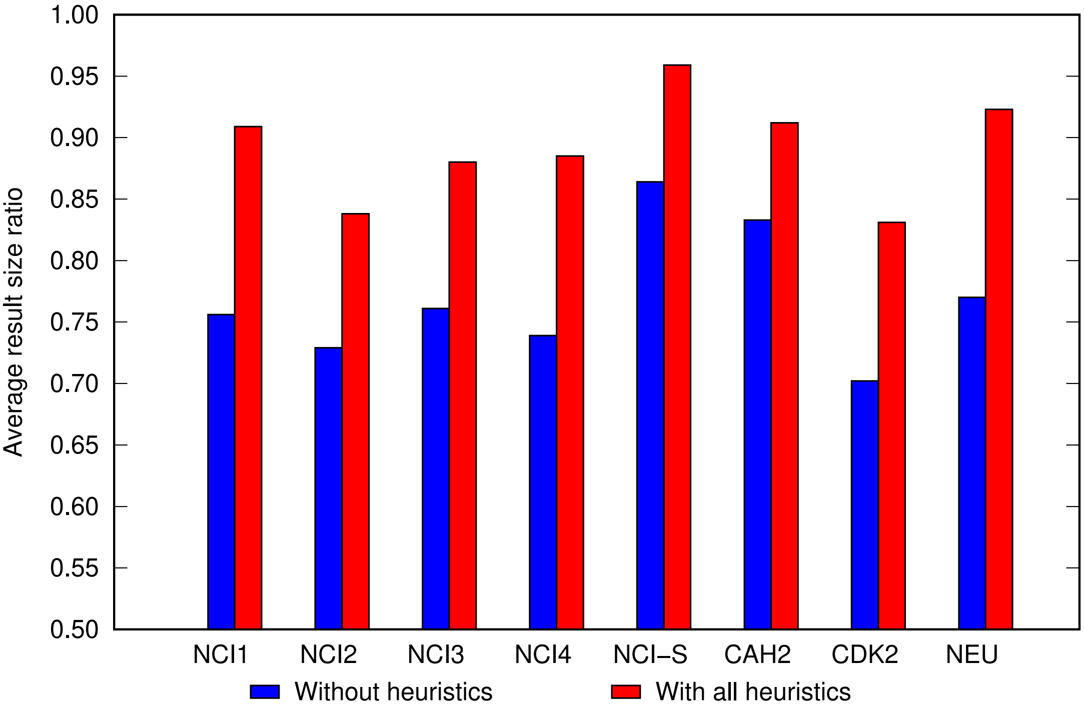


Figure 4.18: Effects of the heuristics on the accuracy of the build-up algorithm.

three implementations: CB denotes the clique-based algorithm using all of the improvements and the default options; CB-F denotes a fast variant of the CB algorithm, in which much fewer iterations are performed by the clique search procedure (20 instead of 1000), but all heuristics are applied in the same way; and BU denotes the build-up algorithm with all heuristics. These results verify that the CB algorithm substantially outperforms the BU method in terms of accuracy. However, it is usually slower than BU, which is why the CB-F variant was also considered in these experiments. CB-F turned out to be consistently faster than BU while still producing better results (see Figure 4.19).

	Average result size ratio			Average running time (ms)		
	CB	CB-F	BU	CB	CB-F	BU
NCI1	0.943	0.939	0.909	11.0	3.1	3.5
NCI2	0.895	0.878	0.838	19.4	3.5	6.8
NCI3	0.907	0.899	0.880	59.2	12.8	29.8
NCI4	0.900	0.896	0.885	237.9	69.6	213.5
NCI-S	0.972	0.971	0.959	7.1	1.6	3.5
CAH2	0.943	0.939	0.912	3.6	0.6	0.7
CDK2	0.890	0.879	0.831	10.2	1.5	1.7
NEU	0.939	0.937	0.923	6.0	1.4	1.6

Table 4.4: Comparison of the accuracy and running times of different algorithms.

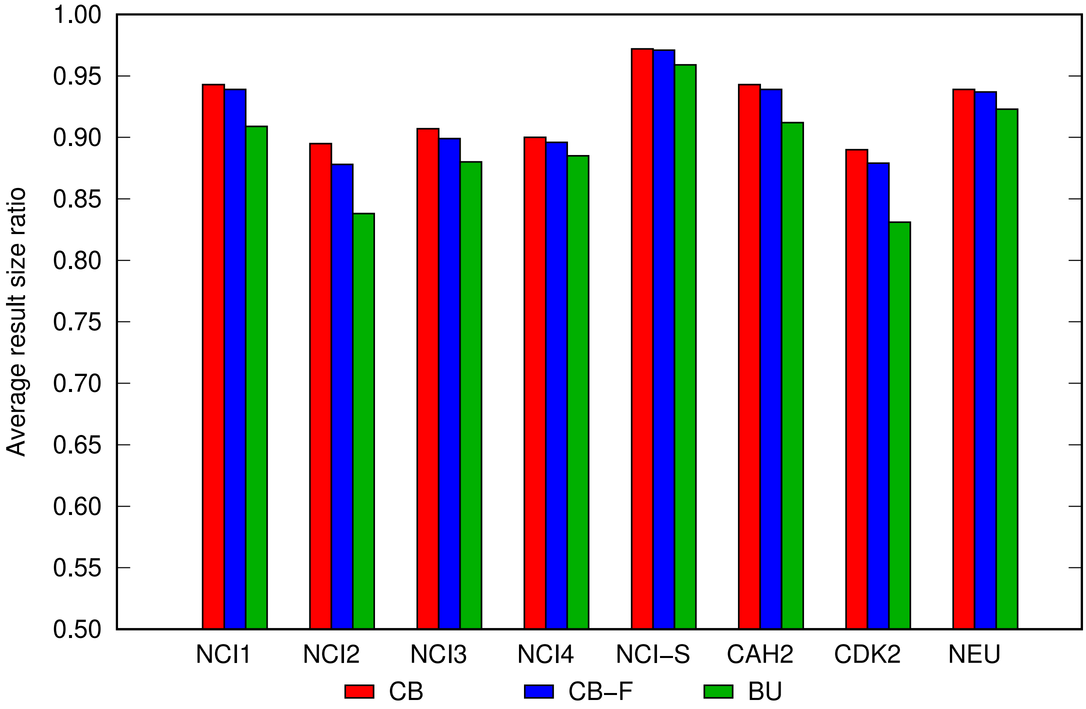


Figure 4.19: Comparison of the accuracy of different algorithms.

Figures 4.20 and 4.21 provide more insight into the performance trends of these three algorithms. Figure 4.20 shows that CB-F is the fastest implementation, while CB is either slower or faster than BU due to the early termination heuristic. On the other hand, Figure 4.21 demonstrates that this relation is rather different in the case of large input graphs. The running time of BU shows worse trend as the input size increases, and it is significantly slower than CB and CB-F for the largest molecular graphs in the NCI-I set (having 150-200 or more non-hydrogen atoms).

These three implementations were also evaluated in the case of connected MCES search, and the relations between their performance and accuracy turned out to be similar. The results are presented in Section 4.4.4, compared with the corresponding algorithm of the Indigo toolkit.

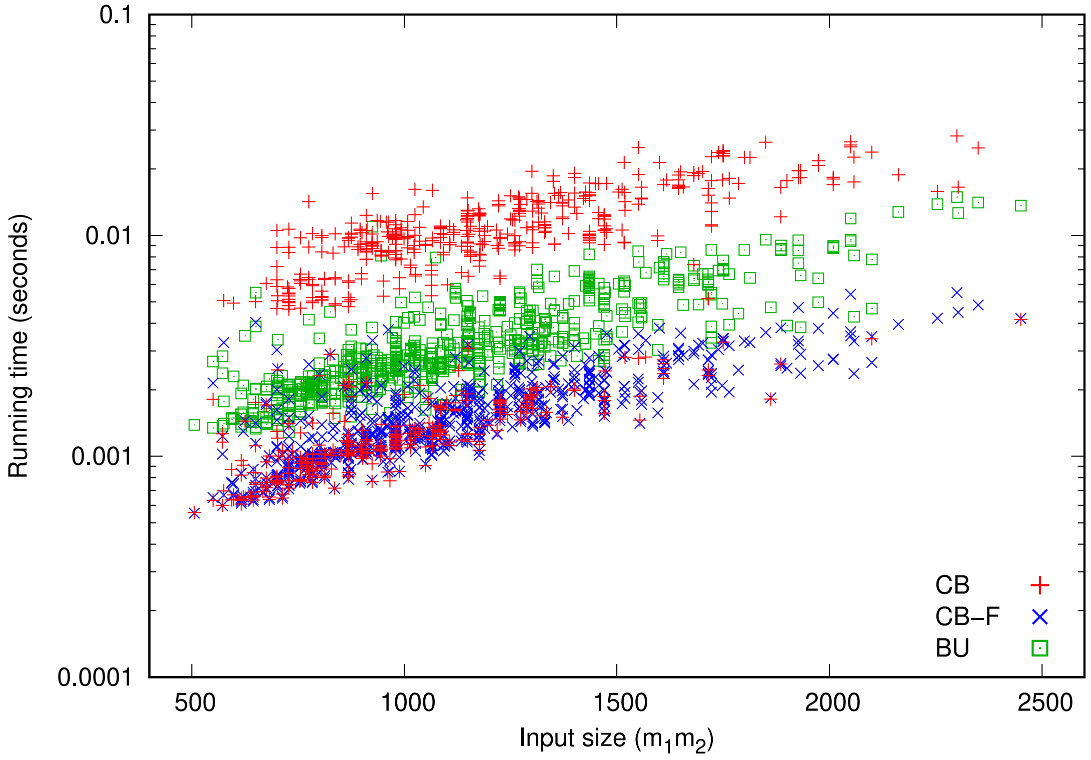


Figure 4.20: Comparison of the running times of different algorithms on the NCI-S set.

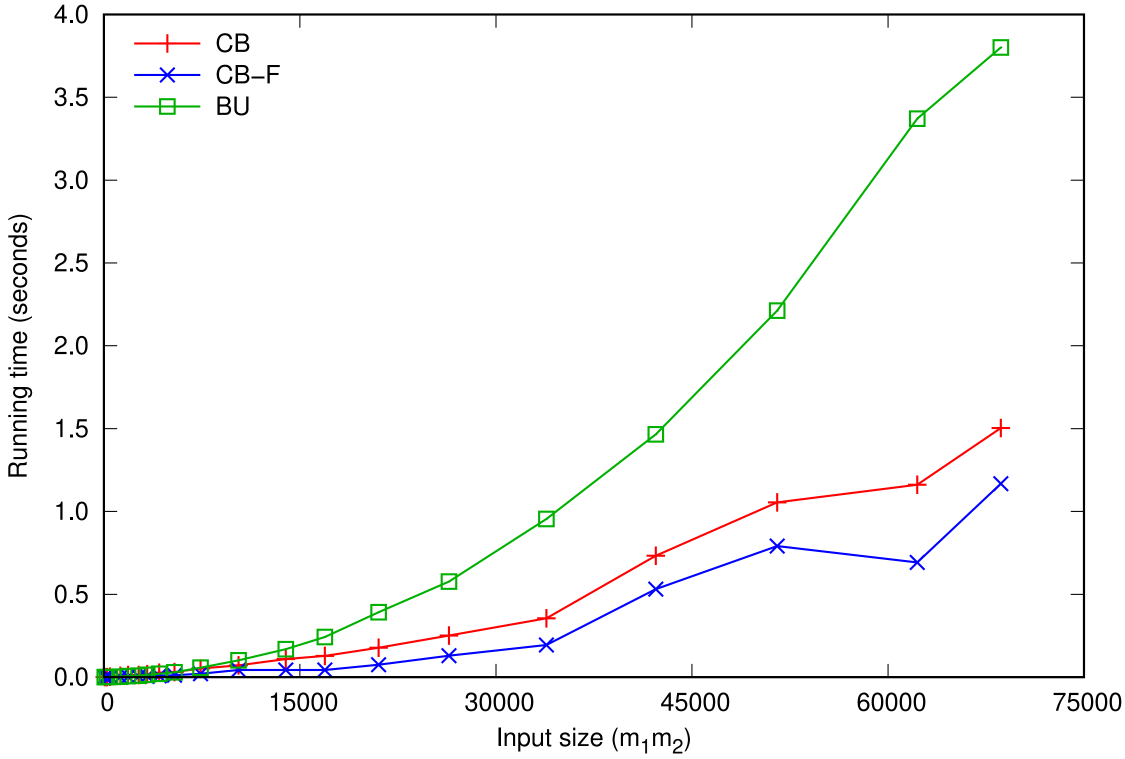


Figure 4.21: Comparison of the running times of different algorithms on the NCI-I set.

According to the presented experiments, we can conclude that the clique-based algorithms are superior to the build-up method, although the developed heuristics effectively decrease the difference. We remark, however, that the accuracy and efficiency of the clique-based algorithms strongly depend on the applied clique search method. We used the algorithm of Grosso et al. [133] for this purpose because it is both simple and very efficient. In contrast, an important advantage of the build-up algorithm is that it is easier to implement, so it can be a reasonable alternative, especially being enhanced with the presented heuristics.

4.4.4 Comparison with other solvers

We also intended to compare our algorithms with other available methods for finding maximum common subgraphs. Such comparison is, however, complicated because the different algorithms solve different variants of the problem (e.g., MCIS or MCES), and the majority of them are restricted to the detection of connected common subgraphs [84, 85]. For our experiments, we selected two solvers: an algorithm implemented in Indigo and KCOMBU, and we applied some restrictions to allow reasonable comparison.

Comparison with Indigo

Indigo is a popular general-purpose open-source cheminformatics toolkit [92]. Its core library is written in C++ and involves several efficient algorithms. For MCES search, Indigo provides both an exact and a heuristic algorithm. The latter is an implementation of the two-stage optimization method (called 2DOM) of Funabiki and Kitamichi [106] and solves the connected MCES problem. The Indigo library (version 1.1.12) and the simple benchmark code we used were compiled with GCC 4.8.1 using -O2 optimization flag.

Table 4.5 and Figure 4.22 summarize the experimental results for the heuristic method of Indigo and our algorithms in the case of connected MCES search. Note that the connected MCES is often much smaller than the disconnected MCES, but the upper bound calculation method does not consider connectivity. Therefore, the result size ratios tend to be significantly smaller than in the disconnected case, but they are still useful for comparing the accuracy of different algorithms.

The relative performance of our implementations turned out to be similar to the disconnected case. In regard to accuracy, BU performs substantially worse than CB and CB-F. However, all of them greatly outperform the Indigo algorithm. While their running times are similar to or better than that of the Indigo code, our algorithms are much more accurate, especially on large graphs. For example, they found 1.5-2 times larger common subgraphs on average for the problem instances in the NCI3 data set and about 5 times larger results on average for the NCI4 set.

Comparison with KCOMBU

Our algorithms were also compared with the KCOMBU code of Kawabata [150, 151], which is the authoritative implementation of the build-up algorithm. KCOMBU turned

	Average result size ratio				Average running time (ms)			
	CB	CB-F	BU	Indigo	CB	CB-F	BU	Indigo
NCI1	0.505	0.497	0.451	0.388	16.6	2.2	1.3	10.8
NCI2	0.548	0.513	0.449	0.400	18.6	2.9	2.7	30.7
NCI3	0.434	0.409	0.344	0.215	38.3	7.8	12.2	99.6
NCI4	0.615	0.549	0.500	0.116	125.5	33.1	107.0	190.6
NCI-S	0.633	0.623	0.600	0.582	9.2	1.6	1.5	12.9
CAH2	0.743	0.734	0.680	0.653	4.4	0.5	0.4	7.3
CDK2	0.662	0.637	0.528	0.464	10.6	1.4	0.8	10.6
NEU	0.870	0.864	0.858	0.814	6.5	1.1	1.1	9.0

Table 4.5: Comparison of the accuracy and running times of the implemented algorithms and Indigo in the case of connected MCES search.

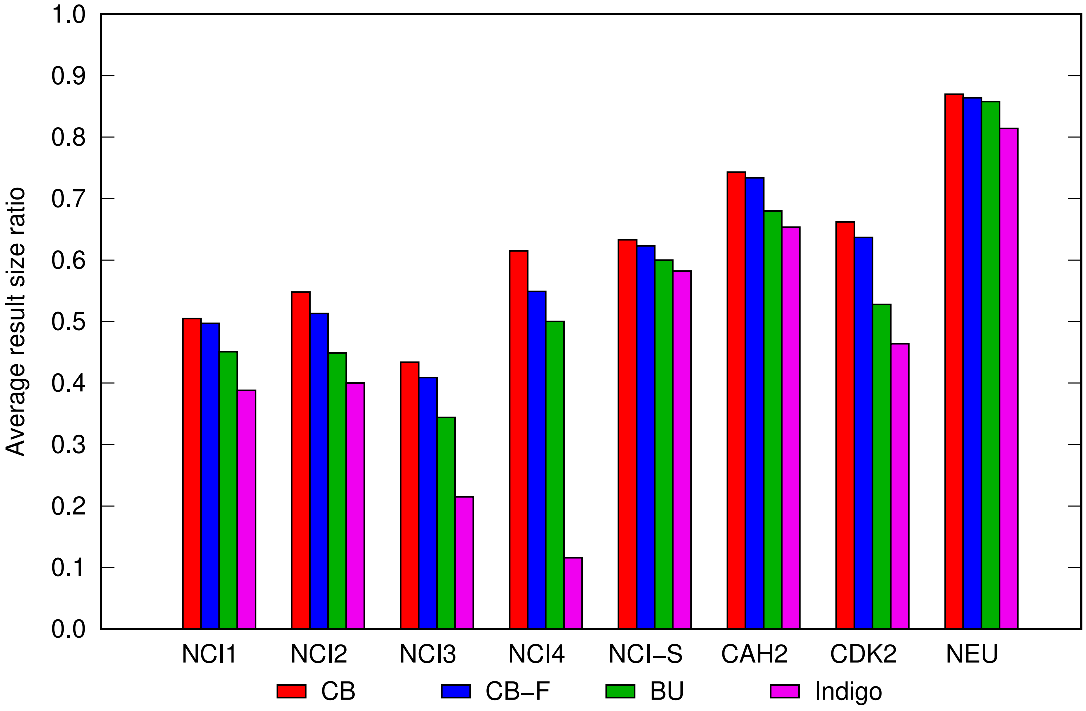


Figure 4.22: Comparison of the accuracy of the implemented algorithms and Indigo in the case of connected MCES search.

out to be effective in different applications [46, 104, 152], and hence it serves as a benchmark. However, it solves the MCIS variant of the problem, which makes it difficult to perform the comparison in a reasonable and fair way. Therefore, we applied a simple and practical approach: our experiments were restricted to those problem instances for which each MCES algorithm actually found a common induced subgraph. That is, their results are also feasible for an MCIS algorithm, so MCES algorithms do not have inherent advantage in these cases. Moreover, we measured the result size in terms of the node count (instead of the edge count) of the common subgraph. This is beneficial for the MCIS approach, especially if the result contains isolated nodes, because they are infeasible for MCES algorithms as they operate on edges.

In this comparison, only three input sets were included: NCI-S, CAH2, and CDK2, which represent a sufficiently large number of problem instances. We considered only disconnected MCIS/MCES search in this case, for which 215 out of the 741 instances from the NCI-S set, 1316 out of 6670 from the CAH2 set, and 372 out of 11781 from the CDK2 set fulfill the restriction described above.

The source code of KCOMBU was obtained from its web site [150], and it was extended with a short code snippet enabling us to measure its running time without the file input–output. The code was compiled with GCC 4.8.1 using -O2 optimization flag, and it was executed using the following command line options: "-alg B" (the build-up algorithm is selected), "-con D" (disconnected MCIS is to be found), and "-at E" (atoms are classified by their element types). Furthermore, the "-bo C" option was applied for the NCI-S problem instances to consider the bond types, while "-bo X" was used for the CAH2 and CDK2 instances to ignore the bond types (in accordance with previous experiments).

Table 4.6 presents the results of these experiments. Although the method of comparison was advantageous for KCOMBU, our algorithms performed much better. They turned out to be an order of magnitude faster, while providing more accurate results. In the case of the NCI-S and CDK2 sets, CB and BU found common subgraphs with 8-10% more nodes on average than the results of KCOMBU. These measurements also verify the effectiveness of the improvements developed in this work.

	Average result size			Average running time (ms)		
	CB	BU	KCOMBU	CB	BU	KCOMBU
NCI-S	22.66	22.66	20.51	3.5	2.5	59.2
CAH2	10.97	10.93	10.76	1.0	0.5	6.5
CDK2	16.59	16.57	15.37	3.1	1.5	22.7

Table 4.6: Comparison of the implemented algorithms and KCOMBU. The results are restricted to the problem instances for which each algorithm found a common induced subgraph. The result size is measured in terms of the number of nodes in the subgraph.

Chapter 5

The LEMON library

In this chapter, we briefly introduce LEMON, the C++ optimization library that involves the minimum-cost flow algorithms presented in Chapter 3. This discussion is based on a joint work with Alpár Jüttner and Balázs Dezső. The results were published in more detail in the paper [4] and the Ph.D. thesis of Balázs Dezső [78].

The chapter is organized as follows. Section 5.1 provides a short overview of LEMON. Sections 5.2, 5.3, 5.4 introduce the main features of the library: data structures, algorithms, and other tools, respectively. Finally, Section 5.5 compares LEMON with two other libraries in terms of performance.

5.1 Overview

LEMON [164] is an abbreviation of *Library for Efficient Modeling and Optimization in Networks*. It is an open-source C++ template library providing efficient and easy-to-use implementations of data structures, algorithms, and other tools for solving combinatorial optimization problems, especially in conjunction with graphs and networks. LEMON is maintained by the MTA-ELTE Egerváry Research Group on Combinatorial Optimization (EGRES) [89], and it is also part of COIN-OR [61], a collection of open-source projects related to operations research. The permissive license of the library enables its use for commercial and non-commercial software development as well as research. The source code and documentation of LEMON can be accessed at `https://lemon.cs.elte.hu`.

The primary goal of the library is to provide highly efficient and flexible components that can easily be combined to solve complex real-world optimization problems. These components include data structures and algorithms related to graphs (such as graph traversal, shortest path, spanning tree, matching, and network flow algorithms) as well as various auxiliary tools. Furthermore, the library features a common high-level interface for several linear programming (LP) and mixed integer programming (MIP) solvers. LEMON is designed to be a cross-platform library that supports a wide range of operating systems, compilers (e.g., GCC, Visual C++, Intel C++), and IDEs (e.g., Eclipse and Visual Studio).

Compared with other alternatives on the market, such as the Boost Graph Library [40, 216] and LEDA [17, 177], LEMON is intended to provide simpler concepts and interface together with a wider variety of complex algorithms, while achieving the highest possible performance. Over more than a decade, LEMON has been extensively used for research and development projects in various fields including network design, logistics, integrated circuit design, image processing, and bioinformatics. The large number of related citations has made it evident that the minimum-cost flow algorithms presented in this thesis play an important role in the popularity of LEMON.

Figure 5.1 shows an introductory code example to demonstrate the basic features of LEMON. It constructs a directed graph, assigns lengths to its arcs, executes Dijkstra’s algorithm, and prints the result (the distance of the target node). For more details, the readers are referred to the documentation of LEMON [165].

```
// Create a directed graph with an arc length map
ListDigraph g;
ListDigraph::ArcMap<int> length(g);

// Add nodes
ListDigraph::Node s = g.addNode();
ListDigraph::Node t = g.addNode();
// ... add more nodes

// Add arcs
ListDigraph::Arc a = g.addArc(s, t);
length[a] = 42;
// ... add more arcs

// Run Dijkstra's algorithm
ListDigraph::NodeMap<int> dist(g);
dijkstra(g, length).distMap(dist).run(s);

// Print the result
std::cout << "dist[t] = " << dist[t] << std::endl;
```

Figure 5.1: Code example demonstrating the usage of LEMON.

5.2 Concepts and data structures

In LEMON, the interface and functionality of generic data types, such as graphs and property maps, are described by concepts, similarly to the Standard Template Library (STL) [226].

LEMON defines two graph concepts: `Digraph` for directed graphs and `Graph` for undirected graphs. The `Graph` concept is designed to also satisfy the requirements of the `Digraph` concept in a way that each edge of an undirected graph can also be viewed as a pair of oppositely directed arcs. The `Arc` type of an undirected graph is convertible to the `Edge` type, so the corresponding edge of an arc can always be obtained conveniently, without calling any functions. Consequently, each undirected graph, without any transformation, can also be considered as a directed graph. The main benefit of this design is that all algorithms that operate on directed graphs are automatically applicable to undirected graphs as well.

5.2.1 Graph structures

The example in Figure 5.1 uses ListDigraph, which is a general directed graph representation based on doubly-linked adjacency lists. Another graph structure is SmartDigraph, which uses vectors for storing the nodes and arcs, and simply-linked lists for keeping track of the incident arcs of each node. Therefore, it requires less memory and can be considerably faster than ListDigraph, but it does not support node and arc removal. Similarly, the classes ListGraph and SmartGraph are provided for representing undirected graphs. Furthermore, LEMON also contains data structures for special classes of graphs, such as grid graphs and full graphs.

Although LEMON is a generic library, its main graph types are not template classes due to an important design decision: all data assigned to nodes and arcs are stored separately (see Section 5.2.3).

5.2.2 Iterators

LEMON defines special iterator concepts for traversing through the nodes, arcs, and edges of graphs. An iterator is initialized to the first element in the traversed range by its constructor, and its validity is checked by comparing to a global constant called INVALID. Continuing the code example shown in Figure 5.1, the nodes of the graph can be iterated and their distances can be printed as follows.

```
for (ListDigraph::NodeIt v(g); v != INVALID; ++v) {
    std::cout << g.id(v) << ": " << dist[v] << std::endl;
}
```

In contrast with the standard C++ iterators in STL, each iterator class in LEMON is convertible to the corresponding graph element type, without having to use operator*(). In the first line of the example above, all occurrences of v refer to the iterator, while its occurrences inside the loop refer to the corresponding node object.

This feature makes the usage of LEMON iterators simpler, although it could not be applied to general iterators. STL defines iterators for containers of arbitrary objects, so the iterator type and the element type may have conflicting functionality. Therefore, an iterator and the referred object must be distinguished. In the case of an iterator called it, for example, ++it affects the iterator itself, while ++(*it) affects the referred object *it. The node and arc types in LEMON, however, have a strictly limited set of features, which does not conflict with the interface of iterators. That is, the program context always indicates whether we refer to an iterator or to a graph element.

5.2.3 Maps

In most applications involving graphs, we need additional data assigned to the nodes and arcs. For this purpose, LEMON provides special data structures called maps. In contrast with other graph libraries, only external property maps are supported by LEMON. That is, the maps are stored separately from the related graph object, but they are updated automatically on the changes of the graph. The main advantage of this design is its great flexibility: maps can be constructed and destructed any time. For example, node and arc maps can be constructed for a graph object `g` as follows.

```
ListDigraph::NodeMap<std::string> label(g);
ListDigraph::ArcMap<int> length(g);
```

Accessing and modifying data assigned to nodes and arcs are among the most frequently used operations in graph algorithms, so they should be highly efficient and convenient. Therefore, the standard map implementations in LEMON use vectors and support constant-time access of their elements. (So they are not to be confused with the `map` container of STL, which provides logarithmic-time access to its elements.) The data assigned to nodes and arcs can be retrieved and overwritten using `operator[]()` of maps as follows.

```
label[v] = "source";
length[e] = 2 * length[f];
```

5.3 Algorithms

LEMON features highly efficient and easy-to-use implementations of a wide range of graph algorithms. Besides the fundamental methods, such as graph traversal and shortest path algorithms, more complex optimization methods are also provided for finding maximum matchings and flows, maximum cliques, minimum-mean cycles, minimum-cost flows, and planar embedding of a graph.

In accordance with the generic programming paradigm, these algorithms are implemented separately from the data structures. For instance, Dijkstra's algorithm is implemented in the class template called `Dijkstra`, which can be used as follows.

```
Dijkstra<ListDigraph> alg(g, length);
alg.distMap(dist);
alg.run(s);
```

For the sake of convenience, however, function-type interfaces are also available for some common algorithms. They are significantly simpler, but the so-called named parameter technique makes them suitable for most use cases. That is, various parameters can be provided by calling appropriate functions, which can be chained together in a single statement. As shown in Figure 5.1, the function interface of Dijkstra's algorithm can be used as follows.

```
dijkstra(g, length).distMap(dist).run(s);
```

This line has the same effect but with a more compact syntax than using the class interface as shown in the previous example. On the other hand, class interfaces provide more options and flexibility in controlling an algorithm (e.g., step-by-step execution).

LEMON also allows the internal data structures of algorithms to be replaced. For example, the implementation of Dijkstra's algorithm uses a writable node map of `bool` value type to record whether a node has already been processed or not. Once the actual distance of a node has been found, the associated value is set to true. Using a special map of type `LoggerBoolMap` instead of a standard node map, we can easily store the processing order of the nodes in a container, as shown in the following code example.

```
    std::vector<ListDigraph::Node> vector;
    dijkstra(g, length)
        .processedMap(loggerBoolMap(std::back_inserter(vector)))
        .run(s);
```

This example also demonstrates another important advantage of the function interface that temporary objects can be passed as reference parameters. Both the insert iterator object and the map object are created only temporarily.

5.4 Other features

5.4.1 Adaptors

LEMON also provides various practical tools for working with graphs effectively. The so-called graph adaptors are particular class templates that wrap other graph data structures to provide modified views of them. In many algorithms and applications, certain alterations of graphs are often required, for instance, some nodes or arcs are to be removed, or the reverse oriented graph is to be used. The actual modification or copying the storage, however, can be rather expensive (in time or in memory usage) compared to the operations to be performed on the altered graph. The graph adaptors can be used in such cases to provide an appropriate view of the underlying graph, using its original operations. These adaptor classes also conform to the graph concepts, so they can be used the same way as standalone graph representations: any generic algorithm works with them, and additional adaptors can also be applied to them.

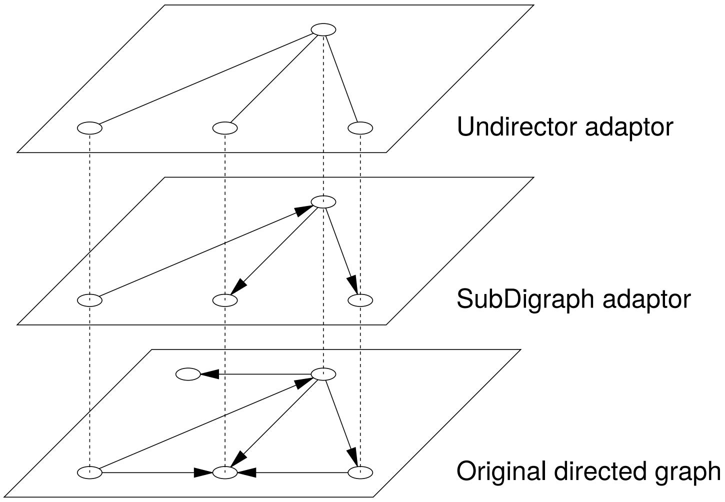


Figure 5.2: Illustration of graph adaptors in LEMON. An adaptor is applied to the original directed graph to hide some nodes and arcs, and another adaptor is used to provide an undirected view of the subgraph.

Figure 5.2 demonstrates the concept and usage of graph adaptors. LEMON provides adaptors for typical use cases such as obtaining a subgraph or the reverse oriented graph of a directed graph, as well as some more complex adaptors, for instance, for representing the residual network related to network flow problems.

Similarly to graph adaptors, LEMON also offers adaptor classes for maps, which apply various numerical and boolean operations on the properties of nodes and arcs without modifying or copying the original data.

5.4.2 LP interface

One of the most important features of LEMON is its convenient high-level LP interface that supports several external LP libraries, including open-source and commercial software. Similarly to the CPLEX Concert Technology [140], this interface applies an object-oriented design, which is more convenient and flexible than the native interfaces of some LP solvers. Furthermore, it also makes it easy to replace the underlying LP library at any stage of the development process. An example of the usage of this interface is shown in Figure 5.3.

```
Lp lp;
Lp::Col x1 = lp.addCol();
Lp::Col x2 = lp.addCol();

lp.max();
lp.obj(10 * x1 + 6 * x2);

lp.addRow(0 <= x1 + x2 <= 100);
lp.addRow(2 * x1 <= x2 + 32);

lp.colLowerBound(x1, 0);
lp.colUpperBound(x2, 10);

lp.solve();
std::cout << "Solution: " << lp.primal() << std::endl;
std::cout << "x1 = " << lp.primal(x1) << std::endl;
std::cout << "x2 = " << lp.primal(x2) << std::endl;
```

$$\boxed{\begin{array}{rl} \text{max} & 10x_1 + 6x_2 \\ \text{subject to} & 0 \le x_1 + x_2 \le 100 \\ & 2x_1 \le x_2 + 32 \\ & x_1 \ge 0 \\ & x_2 \le 10 \end{array}}$$

Figure 5.3: Code example demonstrating the usage of the LP interface of LEMON. The problem to be solved is displayed in the bordered box, next to the corresponding lines of the code.

5.5 Performance

Efficiency is definitely one of the most important properties of a graph optimization library. According to extensive benchmark tests, the fundamental algorithms and data structures of LEMON are significantly more efficient than the corresponding tools of other C++ graph libraries, for instance, the Boost Graph Library (BGL) [40, 216] and LEDA [17, 177].

The article [4] and the thesis [78] introduce experimental studies comparing the performance of LEMON to that of BGL and LEDA in terms of various algorithms for finding shortest paths, maximum matchings, maximum flows, and minimum-cost flows as well as testing the planarity of graphs. Among these results, only a few selected experiments are presented here. In addition, the minimum-cost flow algorithms of LEMON and LEDA are compared in Section 3.3.4, while BGL does not provide any algorithm for solving this problem.

Figures 5.4 and 5.5 show the running times of the implementations of Dijsktra's algorithm (using the standard binary heap data structure) and the push-relabel algorithm for finding maximum flows, respectively. These measurements were carried out using LEMON 1.2, Boost 1.43.0, and LEDA 5.0. The test instances were generated using NETGEN with similar parameters to that of the minimum-cost flow instances described in Section 3.3.1, with the only exception that m was set to be about $n \log_2 n$. For more details, the readers are referred to [4].

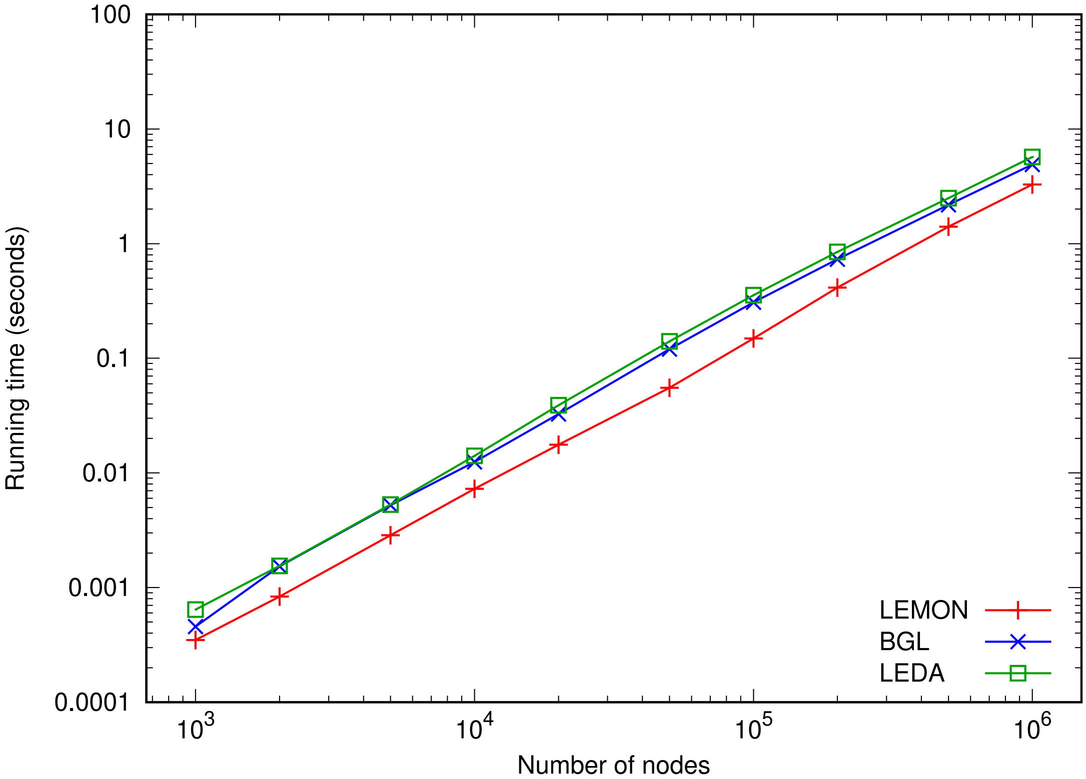


Figure 5.4: Comparison of the performance of Dijkstra's algorithm in different libraries.

Since Dijkstra's algorithm is rather simple, the differences shown in Figure 5.4 are obviously due to the efficiency of the applied graph, map, and heap data structures. In the case of the maximum flow problem, however, the differences between the performance of the libraries turned out to be more significant, as shown in Figure 5.5, which is probably due to the different heuristics applied by the implementations. In this case, LEDA clearly outperforms BGL, but LEMON is even much faster than LEDA.

These results and several other experiments have verified that the algorithms and data structures offered by LEMON are usually more efficient than the corresponding implementations of the other two libraries. This is one of the most important achievements of LEMON, which can be a major reason for using this library.

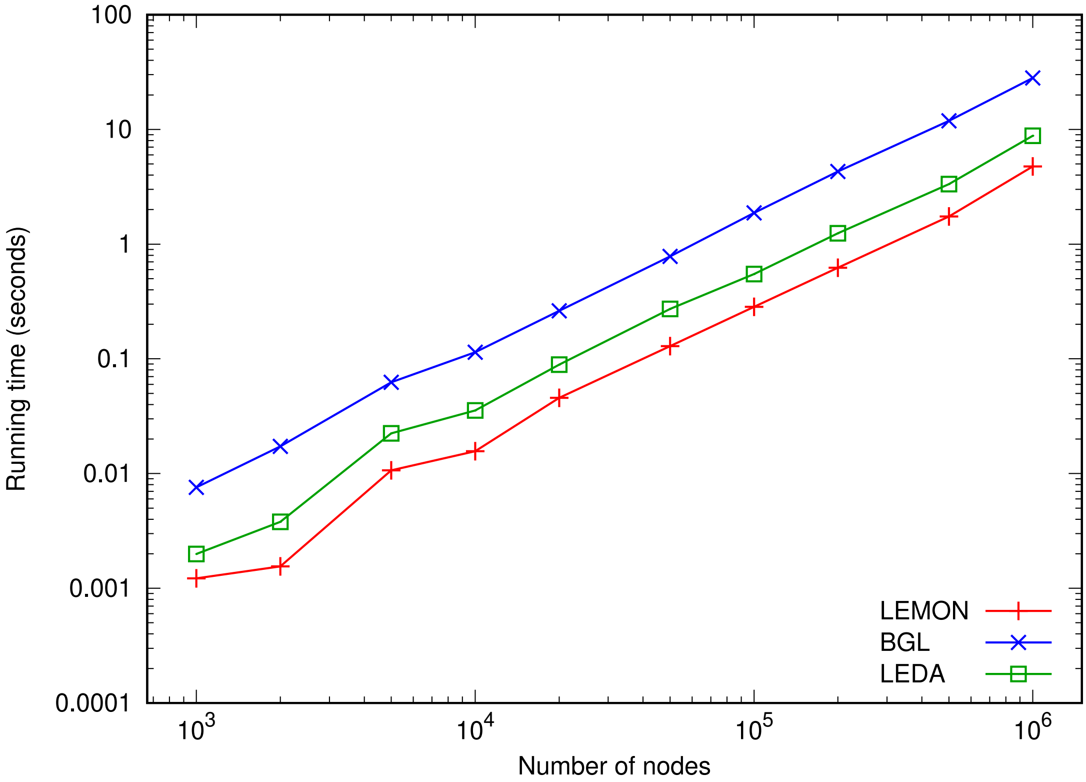


Figure 5.5: Comparison of the performance of the push-relabel maximum flow algorithm in different libraries.

Chapter 6

Summary

This thesis presents efficient algorithms for two fundamental graph optimization problems. Our experiments confirmed that the published algorithms in their basic forms are typically insufficient for effectively solving these complex optimization problems. Therefore, we developed and thoroughly analyzed various improvements, such as special representations of graph and tree structures as well as practical heuristics for guiding basic steps of the algorithms. These improvements make the algorithms much more efficient and more applicable in practice.

This work also involves extensive experimental studies that compare the implemented algorithms with each other and with other publicly available and widely used solvers. The presented results have verified that the most efficient algorithms of the author are usually faster or yield better results than the other methods. These achievements have also been confirmed by independent studies, for example, [82, 85, 181, 182].

The results and discussions in Chapter 3 are based on the articles [2] and [3], Chapter 4 is based on [1], and Chapter 5 is based on [4]. According to Google Scholar [a], these publications have more than 300 independent citations in total.

6.1 Minimum-cost flows

Chapter 3 discusses the contribution of this work related to the minimum-cost flow (MCF) problem, which is one of the most fundamental and most widely studied graph optimization problems. The author implemented several algorithms for this problem, together with various practical improvements and heuristics. An interesting novel result is the application of Goldberg's partial augment-relabel idea [119] in the cost-scaling algorithm, which turned out to be a significant improvement. Another popular solution method is the network simplex algorithm, which was implemented using a quite efficient data structure and various pivot strategies, with great care for all important details. Moreover, three cycle-canceling algorithms and two augmenting path algorithms were also implemented.

An extensive computational analysis was carried out to evaluate the implemented algorithms. The problem instances were generated either using standard generators or based on networks arising in real-life problems. Apart from our implementations, eight other solvers were involved in our study: CS2, LEDA, MCFZIB, CPLEX, the primal and dual versions of MCFSimplex, RelaxIV, and PDNET. Although empirical analysis of

[a] https://scholar.google.com/citations?user=7yee1R0AAAAJ

MCF algorithms has always been of high interest, our work is more comprehensive than previous studies as it involves more algorithms and a greater variety of problem instances, including networks with millions of nodes and arcs.

According to the presented results, we can conclude that implementations of the cost-scaling and primal network simplex algorithms are usually the most efficient and most robust. However, in certain cases, if optimal flows need not be split into many paths, the augmenting path algorithms outperform these methods.

The primal network simplex (NS) code of the author turned out to be significantly faster, often by an order of magnitude, than the other implementations of this method: MCFZIB, MCFSimplex, and CPLEX. Furthermore, NS is usually the most efficient among all algorithms on relatively small networks (up to tens of thousands of nodes). The cost-scaling algorithms, however, tend to be superior to simplex-based methods on very large sparse networks due to their better asymptotic behavior in terms of the number of nodes. The cost-scaling (COS) code of the author performs similarly to or slightly slower than CS2, which is a highly efficient authoritative implementation of this algorithm. The LEDA library also implements this method, but it is slower and numerically less stable than COS and CS2. The other three solvers we considered (the dual version of MCFSimplex, RelaxIV, and PDNET) are less robust and typically perform worse than primal simplex and cost-scaling codes. In many cases, they are slower by orders of magnitude, although RelaxIV is very efficient on certain networks.

The presented implementations are included in LEMON, which makes it easy to combine them with other algorithms and tools to solve complex real-world optimization problems.

6.2 Maximum common subgraphs

Chapter 4 considers the problem of finding maximum common subgraphs, which has important applications in various fields. The author and Péter Englert developed efficient algorithms enhanced with various heuristics for solving this problem. Since this research was conducted at ChemAxon, a cheminformatics software company, the presented algorithms and heuristics were designed for application to molecular graphs. These methods, however, are conceptually general and applicable in other fields where maximum common subgraphs of undirected labeled graphs are to be found.

The maximum common subgraph problem has two different formulations, called MCIS and MCES. We considered the MCES definition, in which the common subgraphs are not required to be induced, as it is more relevant in cheminformatics. Several heuristics were developed for improving the accuracy and efficiency of two MCES algorithms: the clique-based method and the build-up algorithm. Some of the presented ideas were mentioned in the literature before (the improved representation for the clique-based method, the early termination technique, and the connectivity heuristic), but they were applied to different algorithms or in a different way. The other improvements are, to the author's knowledge, novel methods. The heuristic based on extended connectivity fingerprints (ECFPs) considerably improves the accuracy of both algorithms, and the mapping optimization procedure yields chemically more relevant mapping of the atoms and bonds of the two molecular graphs that are compared.

Both algorithms are greatly improved by the developed heuristics, but the clique-based method turned out to be superior. An experimental study was also carried out, including two other solvers as well: KCOMBU and the corresponding algorithm of the open-source Indigo toolkit, both of which were significantly outperformed by our implementations.

The presented algorithms have been integrated into multiple software products of ChemAxon, which are used by leading international companies in the pharmaceutical industry. For academic research and evaluation purposes, the algorithms are available free of charge at https://chemaxon.com.

6.3 The LEMON library

Chapter 5 is a brief introduction to LEMON, which is an open-source C++ library providing efficient implementations of various data structures and algorithms for solving graph optimization problems. The author has been one of the main contributors of the library, together with Alpár Jüttner, the leader of the LEMON project, and Balázs Dezső. Besides the implementation of several algorithms for finding minimum-cost flows, minimum-mean cycles, and maximum cliques, his contribution also involves design, development, review, and documentation tasks related to the core features of the library.

The design principles and concepts of LEMON enable the provided graph representations and algorithms to be both easy-to-use and very efficient. Extensive benchmark tests have verified that the algorithms and data structures of LEMON are typically more efficient than the corresponding tools of other widely used graph libraries, namely the Boost Graph Library (BGL) and LEDA. The large variety of optimization algorithms provided by LEMON, and particularly the minimum-cost flow algorithms, turned out to be essential in making the library widely applied in research and development projects.

Publications of the author

Journal articles

[1] P. Englert and P. Kovács. Efficient heuristics for maximum common substructure search. *J. Chem. Inf. Model.*, 55:941–955, 2015. DOI: 10.1021/acs.jcim.5b00036. IF: **3.657**. Supporting information appeared in [7]. (Cited on pages 1, 65, 73, 82, 101)

[2] P. Kovács. Minimum-cost flow algorithms: an experimental evaluation. *Optim. Method Softw.*, 30:94–127, 2015. DOI: 10.1080/10556788.2014.895828. IF: **0.841**. (Cited on pages 1, 13, 22, 41, 62, 101)

[3] Z. Király and P. Kovács. Efficient implementations of minimum-cost flow algorithms. *Acta Univ. Sapientiae, Inform.*, 4:67–118, 2012. (Cited on pages 1, 13, 22, 41, 62, 101)

[4] B. Dezső, A. Jüttner, and P. Kovács. LEMON – an open source C++ graph template library. *Electron. Notes Theor. Comput. Sci.*, 264:23–45, 2011. DOI: 10.1016/j.entcs.2011.06.003. A preliminary version appeared in [5]. (Cited on pages 1, 93, 98, 99, 101)

Conference papers

[5] B. Dezső, A. Jüttner, and P. Kovács. LEMON – an open source C++ graph template library. In *Proc. 2nd Workshop on Generative Technologies*, WGT 2010, pages 3–13, Paphos, Cyprus, 2010. (Cited on pages 1, 105)

[6] Z. Király and P. Kovács. An experimental study of minimum cost flow algorithms. In *Proc. 8th International Conference on Applied Informatics*, ICAI 2010, Vol. 2. Pages 227–235, Eger, Hungary, 2010. (Cited on pages 1, 13, 22)

Other publications

[7] P. Englert and P. Kovács. Supporting information for *Efficient heuristics for maximum common substructure search*. https://pubs.acs.org/doi/suppl/10.1021/acs.jcim.5b00036 (accessed Mar. 2019). (Cited on pages 82, 105)

[8] P. Kovács. Benchmark data for the minimum-cost flow problem. https://lemon.cs.elte.hu/trac/lemon/wiki/MinCostFlowData (accessed Mar. 2019). (Cited on page 41)

[9] P. Kovács. *Hálózati folyamok és alkalmazásaik – Hatékony algoritmusok a minimális költségű folyam feladatra*. Master's thesis, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary, 2007. (Cited on page 13)

Bibliography

[10] S. J. Aerni, X. Liu, C. B. Do, S. S. Gross, A. Nguyen, S. D. Guo, F. Long, H. Peng, S. S. Kim, and S. Batzoglou. Automated cellular annotation for high-resolution images of adult Caenorhabditis elegans. *Bioinformatics*, 29:i18–i26, 2013. (Cited on page 3)

[11] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding minimum-cost flows by double scaling. Technical report Sloan W. P. No. 2047-88, Stanford University, Stanford, CA, USA, 1988. (Cited on pages 23, 107)

[12] R. K. Ahuja, A. V. Goldberg, J. B. Orlin, and R. E. Tarjan. Finding minimum-cost flows by double scaling. *Math. Program.*, 53:243–266, 1992. A preliminary version appeared in [11]. (Cited on page 23)

[13] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1993. ISBN: 978-0136175490. (Cited on pages 5, 13, 18, 22, 25, 28, 30, 31, 35, 37, 39)

[14] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37:213–223, 1990. (Cited on page 6)

[15] R. K. Ahuja and J. B. Orlin. The scaling network simplex algorithm. *Oper. Res.*, 40:S5–S13, 1992. (Cited on page 37)

[16] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM J. Comput.*, 18:939–954, 1989. (Cited on page 9)

[17] Algorithmic Solutions Software GmbH. The LEDA library, version 5.0. `http://www.algorithmic-solutions.com/index.php/products/leda-for-c` (accessed Mar. 2019). (Cited on pages 52, 93, 98)

[18] R. D. Armstrong and Z. Jin. A new strongly polynomial dual network simplex algorithm. *Math. Program.*, 78:131–148, 1997. (Cited on pages 22, 23, 35)

[19] F. Barahona and É. Tardos. Note on Weintraub's minimum-cost circulation algorithm. *SIAM J. Comput.*, 18:579–583, 1989. (Cited on page 25)

[20] E. J. Barker, D. Buttar, D. A. Cosgrove, E. J. Gardiner, P. Kitts, P. Willett, and V. J. Gillet. Scaffold hopping using clique detection applied to reduced graphs. *J. Chem. Inf. Model.*, 46:503–511, 2006. (Cited on pages 69, 73)

[21] R. S. Barr, F. Glover, and D. Klingman. The alternating basis algorithm for assignment problems. *Math. Program.*, 13:1–13, 1977. (Cited on page 36)

[22] R. S. Barr, F. Glover, and D. Klingman. Enhancements to spanning tree labelling procedures for network optimization. *INFOR*, 17:16–34, 1979. (Cited on pages 21, 35, 37)

[23] R. Becker, M. Fickert, and A. Karrenbauer. A novel dual ascent algorithm for solving the min-cost flow problem. In M. Goodrich and M. Mitzenmacher, editors, *Proc. 18th Workshop on Algorithm Engineering and Experiments*, ALENEX 2016, pages 151–159, 2016. (Cited on pages 22, 41)

[24] R. Becker and A. Karrenbauer. A simple efficient interior point method for min-cost flow. In H.-K. Ahn and C.-S. Shin, editors, *Proc. 25th International Symposium on Algorithms and Computation*, ISAAC 2014, pages 753–765, 2014. (Cited on pages 22, 53)

[25] H. M. Berman, K. Henrick, H. Nakamura, and J. L. Markley. The Worldwide Protein Data Bank (wwPDB): Ensuring a single, uniform archive of PDB data. *Nucleic Acids Res.*, 35:D301–D303, 2007. (Cited on page 83)

[26] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Res.*, 28:235–242, 2000. (Cited on page 83)

[27] A. Bertolini, A. Frangioni, and C. Gentile. The MCFClass Project. `http://www.di.unipi.it/optimize/Software/MCF.html` (accessed Mar. 2019). (Cited on pages 52, 53)

[28] D. P. Bertsekas. Network Optimization Codes. `http://web.mit.edu/dimitrib/www/noc.htm` (accessed Mar. 2019). (Cited on page 53)

[29] D. P. Bertsekas. A distributed algorithm for the assignment problem. Working Paper, Laboratory for Information and Decision Systems, MIT, Cambridge, MA, 1979. (Cited on page 20)

[30] D. P. Bertsekas. Distributed asynchronous relaxation methods for linear network flow problems. In *Proc. 25th IEEE Conference on Decision and Control*, pages 2101–2106, Athens, Greece, 1986. (Cited on page 20)

[31] D. P. Bertsekas and J. Eckstein. Dual coordinate step methods for linear network flow problems. *Math. Program.*, 42:203–243, 1988. (Cited on page 23)

[32] D. P. Bertsekas and P. Tseng. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Oper. Res.*, 36:93–114, 1988. (Cited on pages 22, 53)

[33] D. P. Bertsekas and P. Tseng. The relax codes for linear minimum cost network flow problems. *Ann. Oper. Res.*, 13:125–190, 1988. (Cited on pages 22, 53)

[34] D. P. Bertsekas and P. Tseng. RELAX-IV: A faster version of the RELAX code for solving minimum cost flow problems. Technical report LIDS-P-2276, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA, USA, 1994. (Cited on pages 22, 41, 47, 51, 53)

[35] R. G. Bland, J. Cheriyan, D. Jensen, and L. Ladányi. An empirical study of min cost flow algorithms. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, pages 119–156, 1993. (Cited on pages 22, 41, 62)

[36] R. G. Bland and D. L. Jensen. On the computational behavior of a polynomial-time network flow algorithm. Technical report 661, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, USA, 1985. (Cited on pages 22, 23, 30, 109)

[37] R. G. Bland and D. L. Jensen. On the computational behavior of a polynomial-time network flow algorithm. *Math. Program.*, 54:1–39, 1992. A preliminary version appeared in [36]. (Cited on pages 22, 23, 30)

[38] A. Böcker. Toward an improved clustering of large data sets using maximum common substructures and topological fingerprints. *J. Chem. Inf. Model.*, 48:2097–2107, 2008. (Cited on page 67)

[39] E. E. Bolton, Y. Wang, P. A. Thiessen, and S. H. Bryant. PubChem: Integrated platform of small molecules and biological activities. In *Annual Reports in Computational Chemistry*. Volume 4, chapter 12. American Chemical Society, Washington, DC, USA, 1990. (Cited on pages 71, 73)

[40] Boost C++ Libraries. `https://www.boost.org` (accessed Mar. 2019). (Cited on pages 93, 98)

[41] G. Bradley, G. Brown, and G. Graves. Design and implementation of large scale primal transshipment algorithms. *Manage. Sci.*, 24:1–34, 1977. (Cited on pages 35, 39)

[42] N. Brown. Chemoinformatics – An introduction for computer scientists. *ACM Comput. Surv.*, 41:8–38, 2009. (Cited on page 11)

[43] R. D. Brown, G. Jones, P. Willett, and R. C. Glen. Matching two-dimensional chemical graphs using genetic algorithms. *J. Chem. Inf. Comput. Sci.*, 34:63–70, 1994. (Cited on page 69)

[44] T. Brunsch, K. Cornelissen, B. Manthey, and H. Röglin. Smoothed analysis of the successive shortest path algorithm. In *Proc. 24th ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1180–1189, 2013. (Cited on pages 22, 109)

[45] T. Brunsch, K. Cornelissen, B. Manthey, H. Röglin, and C. Rösner. Smoothed analysis of the successive shortest path algorithm. *SIAM J. Comput.*, 44:1798–1819, 2015. A preliminary version appeared in [44]. (Cited on page 22)

[46] M. Brylinski. Nonlinear scoring functions for similarity-based ligand docking and binding affinity prediction. *J. Chem. Inf. Model.*, 53:3097–3112, 2013. (Cited on page 91)

[47] H. Bunke. Graph matching: Theoretical foundations, algorithms, and applications. In *Proc. Vision Interface 2000*, VI 2000, pages 82–88, Montreal, Canada, 2000. (Cited on pages 11, 65)

[48] H. Bunke, C. Irniger, and M. Neuhaus. Graph matching – challenges and potential solutions. In *Image Analysis and Processing*, ICIAP 2005, pages 1–10, 2005. (Cited on page 65)

[49] H. Bunke and K. Shearer. A graph distance metric based on the maximal common subgraph. *Pattern Recognit. Lett.*, 19:255–259, 1998. (Cited on page 67)

[50] U. Bünnagel, B. Korte, and J. Vygen. Efficient implementation of the Goldberg-Tarjan minimum-cost flow algorithm. *Optim. Method Softw.*, 10:157–174, 1998. (Cited on pages 22, 32, 33, 41)

[51] R. G. Busacker and P. J. Gowen. A procedure for determining a family of minimum-cost network flow patterns. Technical report ORO-TP-15, Operations Research Office, The Johns Hopkins University, Bethesda, MD, USA, 1960. (Cited on pages 21, 23, 27)

[52] C. Bussar, M. Moos, R. Alvarez, P. Wolf, T. Thien, H. Chen, Z. Cai, M. Leuthold, D. U. Sauer, and A. Moser. Optimal allocation and capacity of energy storage systems in a future European power system with 100% renewable energy generation. *Energy Procedia*, 46:40–47, 2014. (Cited on page 3)

[53] Y. Cao, T. Jiang, and T. Girkez. A maximum common substructure-based algorithm for searching and predicting drug-like compounds. *Bioinformatics*, 24:366–374, 2008. (Cited on pages 67, 68, 75)

[54] V. Carletti, P. Foggia, and M. Vento. VF2 Plus: An improved version of VF2 for biological graphs. In C.-L. Liu, B. Luo, W. G. Kropatsch, and J. Cheng, editors, *Graph-Based Representations in Pattern Recognition*, pages 168–177, 2015. (Cited on page 11)

[55] ChemAxon. JChem Suite. `https://chemaxon.com/products/jchem-engines` (accessed Mar. 2019). (Cited on page 82)

[56] L. Chen and W. Robien. MCSS: A new algorithm for perception of maximal common substructures and its application to NMR spectral studies. 1. The algorithm. *J. Chem. Inf. Comput. Sci.*, 32:501–506, 1992. (Cited on pages 67, 68)

[57] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a maximum flow be computed in $o(nm)$ time? In *Automata, Languages and Programming*, pages 235–248, 1990. (Cited on page 9)

[58] J. Cheriyan, T. Hagerup, and K. Mehlhorn. An $o(n^3)$-time maximum-flow algorithm. *SIAM J. Comput.*, 25:1144–1170, 1996. (Cited on page 9)

[59] B. V. Cherkassky and A. V. Goldberg. On implementing the push–relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997. (Cited on page 32)

[60] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Math. Program.*, 73:129–174, 1996. (Cited on page 6)

[61] COIN-OR – Computational Infrastructure for Operations Research. https://www.coin-or.org (accessed Mar. 2019). (Cited on page 93)

[62] C. J. Colbourn. On testing isomorphism of permutation graphs. *Networks*, 11:13–21, 1981. (Cited on page 10)

[63] Computer Vision Research Group, University of Western Ontario. Max-flow problem instances in vision. https://vision.cs.uwaterloo.ca/data/maxflow (accessed Mar. 2019). (Cited on page 43)

[64] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *Int. J. Pattern Recognit. Artif. Intell.*, 18:265–298, 2004. (Cited on pages 11, 65)

[65] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *J. Graph Alg. Appl.*, 11:99–143, 2007. (Cited on pages 65, 69)

[66] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26:1367–1372, 2004. (Cited on page 11)

[67] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 3rd edition, 2009. ISBN: 978-0262033848. (Cited on pages 5, 25)

[68] K. Cornelissen and B. Manthey. Smoothed analysis of the minimum-mean cycle canceling algorithm and the network simplex algorithm. In D. Xu, D. Du, and D. Du, editors, *Proc. 21st International Computing and Combinatorics Conference*, COCOON 2015, pages 701–712, 2015. (Cited on pages 22, 37)

[69] W. H. Cunningham. A network simplex method. *Math. Program.*, 11:105–116, 1976. (Cited on page 36)

[70] W. H. Cunningham. Theoretical properties of the network simplex method. *Math. Oper. Res.*, 4:196–208, 1979. (Cited on pages 37, 39, 40)

[71] A. Dalke and J. Hastings. FMCS: A novel algorithm for the multiple MCS problem. *J. Cheminf.*, 5:O6, 2013. (Cited on pages 68, 79)

[72] G. B. Dantzig. Application of the simplex method to a transportation problem. In T. C. Koopmans, editor, *Activity analysis of production and allocation*, pages 359–373. John Wiley & Sons, Inc., New York, NY, USA, 1951. (Cited on pages 21, 35)

[73] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, USA, 1963. (Cited on pages 21, 35)

[74] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. Des. Automat. Electron. Syst.*, 9:385–418, 2004. (Cited on pages 8, 26)

[75] A. Dasdan and R. K. Gupta. Faster maximum and minimum mean cycle algorithms for system performance analysis. *IEEE Trans. Comput.-Aided Des.*, 17:889–899, 1998. (Cited on pages 8, 26)

[76] P. M. Dean. *Molecular Similarity in Drug Design*. Blackie Academic & Professional, London, UK, 1995. (Cited on page 67)

[77] R. Dewil, P. Vansteenwegen, D. Cattrysse, and D. V. Oudheusden. A minimum cost network flow model for the maximum covering and patrol routing problem. *Eur. J. Oper. Res.*, 247:27–36, 2015. (Cited on page 3)

[78] B. Dezső. *Optimization methods in remote sensing and geoinformatics*. PhD thesis, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary, 2012. (Cited on pages 93, 98)

[79] DIMACS. 9th DIMACS Implementation Challenge: Shortest Paths. http://diag.uniroma1.it/challenge9/ (accessed Mar. 2019), 2005-2006. (Cited on page 43)

[80] DIMACS. 1st DIMACS Implementation Challenge: Network Flows and Matching. http://dimacs.rutgers.edu/archive/pub/netflow/ (accessed Mar. 2019), 1990-1991. (Cited on pages 22, 41)

[81] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280, 1970. (Cited on pages 9, 26, 27)

[82] E. Duesbury. *Applications and variations of the maximum common subgraph for the determination of chemical similarity*. PhD thesis, Information School, University of Sheffield, Sheffield, UK, 2015. (Cited on page 101)

[83] E. Duesbury, J. D. Holliday, and P. Willett. Maximum common substructure-based data fusion in similarity searching. *J. Chem. Inf. Model.*, 55:222–230, 2015. (Cited on page 67)

[84] E. Duesbury, J. D. Holliday, and P. Willett. Maximum common subgraph isomorphism algorithms. *MATCH Commun. Math. Comput. Chem.*, 77:213–232, 2017. (Cited on pages 65, 69, 90)

[85] E. Duesbury, J. D. Holliday, and P. Willett. Comparison of maximum common subgraph isomorphism algorithms for the alignment of 2D chemical structures. *ChemMedChem*, 13:588–598, 2018. (Cited on pages 65, 69, 90, 101)

[86] P. J. Durand, R. Pasari, J. W. Baker, and C.-C. Tsai. An efficient algorithm for similarity analysis of molecules. *Internet J. Chem.*, 2:1–16, 1999. (Cited on pages 69–71)

[87] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. In *Combinatorial Structures and Their Applications*, pages 93–96, 1970. (Cited on pages 9, 21, 23, 27, 112)

[88] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972. A preliminary version appeared in [87]. (Cited on pages 9, 21, 23, 26, 27, 29)

[89] EGRES – MTA-ELTE Egerváry Research Group on Combinatorial Optimization. http://www.cs.elte.hu/egres/ (accessed Mar. 2019). (Cited on page 93)

[90] H.-C. Ehrlich and M. Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: A review. *WIREs Comput. Mol. Sci.*, 1:68–79, 2011. (Cited on pages 65, 67, 69)

[91] T. Engel. Basic overview of chemoinformatics. *J. Chem. Inf. Model.*, 46:2267–2277, 2006. (Cited on page 11)

[92] EPAM. Indigo Toolkit. http://lifescience.opensource.epam.com/indigo/ (accessed Mar. 2019). (Cited on pages 82, 90)

[93] D. Fooshee, A. Andronico, and P. Baldi. ReactionMap: An efficient atom-mapping algorithm for chemical reactions. *J. Chem. Inf. Model.*, 53:2812–2819, 2013. (Cited on page 67)

[94] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Can. J. Math.*, 8:399–404, 1956. (Cited on pages 9, 21)

[95] L. R. Ford and D. R. Fulkerson. Solving the transportation problem. *Manage. Sci.*, 3:24–32, 1956. (Cited on page 21)

[96] L. R. Ford and D. R. Fulkerson. A primal-dual algorithm for the capacitated Hitchcock problem. *Nav. Res. Logist. Quart.*, 4:47–54, 1957. (Cited on page 21)

[97] L. R. Ford and D. R. Fulkerson. Constructing maximal dynamic flows from static flows. *Oper. Res.*, 6:419–433, 1958. (Cited on pages 21, 23)

[98] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 1962. (Cited on pages 18, 21, 27)

[99] A. Frangioni and A. Manca. A computational study of cost reoptimization for min-cost flow problems. *INFORMS J. Comput.*, 18:61–70, 2006. (Cited on pages 13, 41, 51–53)

[100] A. Frank. On Kuhn's Hungarian Method – A tribute from Hungary. *Nav. Res. Logist.*, 52:2–5, 2005. (Cited on page 15)

[101] A. Frank. *Connections in Combinatorial Optimization*. Oxford Lecture Series in Mathematics and Its Applications, Vol. 38. Oxford University Press, Oxford, UK, 2011. ISBN: 978-0199205271. (Cited on page 5)

[102] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987. (Cited on pages 6, 22)

[103] S. Fujishige. A capacity-rounding algorithm for the minimum-cost circulation problem: A dual framework of the Tardos algorithm. *Math. Program.*, 35:298–308, 1986. (Cited on page 23)

[104] Y. Fukunishi and H. Nakamura. Integration of ligand-based drug screening with structure-based drug screening by combining maximum volume overlapping score with ligand docking. *Pharmaceuticals*, 5:1332–1345, 2012. (Cited on page 91)

[105] D. R. Fulkerson. An out-of-kilter method for minimal-cost flow problems. *J. Soc. Indust. Appl. Math.*, 9:18–27, 1961. (Cited on pages 21, 23)

[106] N. Funabiki and J. Kitamichi. A two-stage discrete optimization method for largest common subgraph problems. *IEICE Trans. Inf. Syst.*, E82–D:1145–1153, 1999. (Cited on page 90)

[107] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18:1013–1036, 1989. (Cited on page 23)

[108] Z. Galil and É. Tardos. An $O(n^2(m + n \log n) \log n)$ min-cost flow algorithm. In *Proc. 27th Symposium on Foundations of Computer Science*, FOCS '86, pages 1–9, 1986. (Cited on pages 23, 114)

[109] Z. Galil and É. Tardos. An $O(n^2(m + n \log n) \log n)$ min-cost flow algorithm. *J. ACM*, 35:374–386, 1988. A preliminary version appeared in [108]. (Cited on page 23)

[110] E. J. Gardiner, V. J. Gillet, P. Willett, and D. A. Cosgrove. Representing clusters using a maximum common edge substructure algorithm applied to reduced graphs and molecular graphs. *J. Chem. Inf. Model.*, 47:354–366, 2007. (Cited on pages 67, 69, 73)

[111] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, CA, USA, 1979. ISBN: 978-0716710455. (Cited on page 10)

[112] M. R. Garey, D. S. Johnson, and R. E. Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM J. Comput.*, 5:704–714, 1976. (Cited on page 10)

[113] J. B. Gauthier, J. Desrosiers, and M. E. Lübbecke. About the minimum mean cycle-canceling algorithm. *Discrete Appl. Math.*, 196:115–134, 2015. (Cited on page 25)

[114] L. Georgiadis, A. V. Goldberg, R. E. Tarjan, and R. F. Werneck. An experimental study of minimum mean cycle algorithms. In I. Finocchi and J. Hershberger, editors, *Proc. 11th Workshop on Algorithm Engineering and Experiments*, ALENEX 2009, pages 1–13, 2009. (Cited on pages 8, 26)

[115] F. Glover, D. Karney, D. Klingman, and A. Napier. A computation study on start procedures, basis change criteria, and solution algorithms for transportation problems. *Manage. Sci.*, 20:793–813, 1974. (Cited on pages 21, 22, 35, 37)

[116] A. V. Goldberg. CS2 Software, version 4.6. `http://www.avglab.com/andrew/soft.html` (accessed Mar. 2019). (Cited on page 52)

[117] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24:494–504, 1995. (Cited on page 33)

[118] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22:1–29, 1997. (Cited on pages 22, 32, 33, 41, 51–53, 63)

[119] A. V. Goldberg. The partial augment-relabel algorithm for the maximum flow problem. In *Proc. 16th Annual European Symposium on Algorithms*, ESA '08, pages 466–477, 2008. (Cited on pages 34, 43, 50, 101)

[120] A. V. Goldberg and M. Kharitonov. On implementing scaling push-relabel algorithms for the minimum-cost flow problem. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, pages 157–198, 1993. (Cited on pages 22, 32, 33, 41, 42, 52, 63)

[121] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. In *Proc. 38th Symposium on Foundations of Computer Science*, FOCS '97, pages 2–11, 1997. (Cited on pages 9, 115)

[122] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45:783–797, 1998. A preliminary version appeared in [121]. (Cited on page 9)

[123] A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network flow algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Paths, Flows and VLSI-Layout*, pages 101–164. Springer, Berlin, Germany, 1990. (Cited on pages 13, 16, 18, 22, 25)

[124] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. In *Proc. 18th ACM Symposium on Theory of Computing*, STOC '86, pages 136–146, 1986. (Cited on pages 9, 30, 115)

[125] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proc. 19th ACM Symposium on Theory of Computing*, STOC '87, pages 7–18, 1987. (Cited on pages 22, 23, 30, 115)

[126] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35:921–940, 1988. A preliminary version appeared in [124]. (Cited on pages 9, 30)

[127] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. In *Proc. 20th ACM Symposium on Theory of Computing*, STOC '88, pages 388–397, 1988. (Cited on pages 23, 25, 26, 115)

[128] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36:873–886, 1989. A preliminary version appeared in [127]. (Cited on pages 23, 25–27)

[129] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15:430–466, 1990. A preliminary version appeared in [125]. (Cited on pages 22, 23, 30–32)

[130] D. Goldfarb and J. Hao. Polynomial-time primal simplex algorithms for the minimum cost network flow problem. *Algorithmica*, 8:145–160, 1992. (Cited on pages 22, 35)

[131] D. Goldfarb, J. Hao, and S.-R. Kai. Anti-stalling pivot rules for the network simplex algorithm. *Networks*, 20:79–91, 1990. (Cited on pages 37, 39, 40)

[132] M. D. Grigoriadis. An efficient implementation of the network simplex method. *Math. Program. Stud.*, 26:83–111, 1986. (Cited on pages 21, 35, 39, 40)

[133] A. Grosso, M. Locatelli, and W. Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *J. Heuristics*, 14:587–612, 2008. (Cited on pages 71, 72, 90)

[134] R. Hariharan, A. Janakiraman, R. Nilakantan, B. Singh, S. Varghese, G. Landrum, and A. Schuffenhauer. MultiMCS: A fast algorithm for the maximum common substructure problem on multiple molecules. *J. Chem. Inf. Model.*, 51:788–806, 2011. (Cited on pages 68, 73, 79)

[135] M. Hartmann and J. B. Orlin. Finding minimum cost to time ratio cycles with small integral transit times. *Networks*, 23:567–574, 1993. (Cited on page 26)

[136] F. L. Hitchcock. The distribution of a product from several sources to numerous localities. *J. Math. Phys.*, 20:224–230, 1941. (Cited on page 21)

[137] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. 6th ACM Symposium on Theory of Computing*, STOC '74, pages 172–184, 1974. (Cited on page 10)

[138] R. A. Howard. *Dynamic Programming and Markov Processes.* The MIT Press, Cambridge, MA, USA, 1960. (Cited on page 26)

[139] E. Hung, T. Todman, and W. Luk. Transparent insertion of latency-oblivious logic onto FPGAs. In *24th International Conference on Field Programmable Logic and Applications*, FPL 2014, pages 1–8, 2014. (Cited on page 3)

[140] IBM ILOG CPLEX Optimization Studio, v12.4. `https://www.ibm.com/hu-en/marketplace/ibm-ilog-cplex` (accessed Mar. 2019). (Cited on pages 52, 98)

[141] M. Iri. A new method for solving transportation-network problems. *J. Oper. Res. Soc. Japan*, 3:27–87, 1960. (Cited on pages 21, 23, 27)

[142] W. S. Jewell. Optimal flow through networks. Technical report No. 8, Operations Research Center, MIT, Cambridge, MA, USA, 1958. (Cited on pages 21, 23, 27)

[143] D. S. Johnson and C. C. McGeoch, editors, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12. Providence, RI, USA, 1993. American Mathematical Society. (Cited on page 22)

[144] E. L. Johnson. Networks and basic solutions. *Oper. Res.*, 14:619–623, 1966. (Cited on pages 21, 37)

[145] M. A. Johnson and G. M. Maggiora. *Concepts and Applications of Molecular Similarity.* Wiley-Interscience Publication. John Wiley & Sons, Inc., New York, NY, USA, 1990. ISBN: 9780471621751. (Cited on page 67)

[146] A. Jüttner and P. Madarasi. VF2++ – An improved subgraph isomorphism algorithm. *Discrete Appl. Math.*, 242:69–81, 2018. (Cited on page 11)

[147] L. V. Kantorovich. Mathematical methods of organizing and planning production, 1939. Translated in *Manage. Sci.*, 6:366–422, 1960. (Cited on page 21)

[148] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23:309–311, 1978. (Cited on pages 8, 26)

[149] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Doklady*, 15:434–437, 1974. (Cited on page 9)

[150] T. Kawabata. KCOMBU: for matching chemical structure by the build-up algorithm. http://strcomp.protein.osaka-u.ac.jp/kcombu/ (accessed Mar. 2019). (Cited on pages 82, 83, 90, 92)

[151] T. Kawabata. Build-up algorithm for atomic correspondence between chemical structures. *J. Chem. Inf. Model.*, 51:1775–1787, 2011. (Cited on pages 65, 67, 69, 72, 75, 77, 82, 83, 90)

[152] T. Kawabata and H. Nakamura. 3D flexible alignment using 2D maximum common substructure: dependence of prediction accuracy on target-reference chemical similarity. *J. Chem. Inf. Model.*, 54:1850–1863, 2014. (Cited on pages 67, 91)

[153] D. J. Kelly and G. M. O'Neill. *The minimum cost flow problem and the network simplex solution method*. Master's thesis, University College, Dublin, Ireland, 1991. (Cited on pages 35, 37, 39, 40)

[154] J. L. Kennington and R. V. Helgason. *Algorithms for Network Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1980. (Cited on pages 21, 35, 37)

[155] V. King, S. Rao, and R. E. Tarjan. A faster deterministic maximum flow algorithm. *J. Algorithms*, 17:447–474, 1994. (Cited on page 9)

[156] M. Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Manage. Sci.*, 14:205–220, 1967. (Cited on pages 21, 23, 24)

[157] D. Klingman, A. Napier, and J. Stutz. NETGEN: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Manage. Sci.*, 20:814–821, 1974. (Cited on page 41)

[158] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250:1–30, 2001. (Cited on pages 69, 70, 75, 79)

[159] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Algorithms and Combinatorics, Vol. 21. Springer, Berlin, Germany, 5th edition, 2012. (Cited on pages 5, 13, 18, 22, 25)

[160] E. B. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Pract. Exper.*, 34:591–607, 2004. (Cited on page 68)

[161] H. W. Kuhn. The Hungarian Method for the assignment problem. *Nav. Res. Logist. Quart.*, 2:83–97, 1955. (Cited on pages 15, 21)

[162] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endowment*, 6:133–144, 2012. (Cited on page 11)

[163] Y. Lee. *Computational analysis of network optimization algorithms*. PhD thesis, Dept. of Civil and Environmental Engineering, MIT, Cambridge, MA, USA, 1993. (Cited on page 42)

[164] LEMON – Library for Efficient Modeling and Optimization in Networks. https://lemon.cs.elte.hu (accessed Mar. 2019). (Cited on pages 4, 93)

[165] LEMON Documentation, version 1.3.1. https://lemon.cs.elte.hu/pub/doc/1.3.1/ (accessed Mar. 2019). (Cited on pages 28, 94)

[166] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9:341–352, 1973. (Cited on page 69)

[167] A. Löbel. MCF version 1.3 – A network simplex implementation. http://www.zib.de/opt-long_projects/Software/Mcf/ (accessed Mar. 2019). (Cited on page 52)

[168] A. Löbel. Solving large-scale real-world minimum-cost flow problems by a network simplex method. Technical report SC 96-7, Zuse Institute Berlin (ZIB), Berlin, Germany, 1996. (Cited on pages 35, 39, 41, 51–53)

[169] A. Löbel. Vehicle scheduling in public transit and Lagrangean pricing. *Manage. Sci.*, 44:1637–1649, 1999. (Cited on page 13)

[170] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25:42–65, 1982. (Cited on page 10)

[171] V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inf. Process. Lett.*, 7:277–278, 1978. (Cited on page 9)

[172] S. Mallach. *Exact Integer Programming Approaches to Sequential Instruction Scheduling and Offset Assignment*. PhD thesis, Universität zu Köln, Köln, Germany, 2015. (Cited on page 3)

[173] S. Mallach. More general optimal offset assignment. *Leibniz Trans. Embed. Syst.*, 2:02:1–02:18, 2015. (Cited on page 3)

[174] J. Marialke, R. Körner, S. Tietze, and J. Apostolakis. Graph-based molecular alignment (GMA). *J. Chem. Inf. Model.*, 47:591–601, 2007. (Cited on page 67)

[175] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Pract. Exper.*, 12:23–34, 1982. (Cited on page 68)

[176] J. J. McGregor and P. Willett. Use of a maximum common subgraph algorithm in the automatic identification of ostensible bond changes occurring in chemical reactions. *J. Chem. Inf. Comput. Sci.*, 21:137–140, 1981. (Cited on page 67)

[177] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, New York, NY, USA, 1999. ISBN: 0-521-56329-1. (Cited on pages 52, 93, 98)

[178] R. Mesa-Arango and S. V. Ukkusuri. Minimum cost flow problem formulation for the static vehicle allocation problem with stochastic lane demand in truckload strategic planning. *Transp. A: Transp. Sci.*, 13:893–914, 2017. (Cited on page 3)

[179] G. J. Minty. Monotone networks. *Proc. R. Soc. Lond., Series A*, 257:194–212, 1960. (Cited on pages 21, 23)

[180] H. L. Morgan. The generation of a unique machine description for chemical structures – A technique developed at Chemical Abstracts Service. *J. Chem. Doc.*, 5:107–113, 1965. (Cited on pages 72, 76)

[181] J. C. Müller. *Auctions in Exchange Trading Systems: Modeling Techniques and Algorithms*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany, 2014. (Cited on pages 3, 101)

[182] J. C. Müller, S. Pokutta, A. Martin, S. Pape, A. Peter, and T. Winter. Pricing and clearing combinatorial markets with singleton and swap orders. *Math. Meth. Oper. Res.*, 85:155–177, 2017. (Cited on pages 3, 101)

[183] J. M. Mulvey. Pivot strategies for primal-simplex network codes. *J. ACM*, 25:266–270, 1978. (Cited on page 40)

[184] National Cancer Institute. Downloadable Structure Files of NCI Open Database Compounds. https://cactus.nci.nih.gov/download/nci/ (accessed Mar. 2019). (Cited on page 82)

[185] National Center for Biotechnology Information. The PubChem Project. https://pubchem.ncbi.nlm.nih.gov (accessed Mar. 2019). (Cited on pages 71, 73)

[186] V. Nicholson, C.-C. Tsai, M. Johnson, and M. Naim. A subgraph isomorphism theorem for molecular graphs. In R. B. King and D. H. Rouvray, editors, *Graph Theory and Topology in Chemistry*, pages 226–230, 1987. (Cited on page 70)

[187] J. B. Orlin. Genuinely polynomial simplex and non-simplex algorithms for the minimum cost flow problem. Technical report 1615-84, Sloan School of Management, MIT, Cambridge, MA, USA, 1984. (Cited on pages 22, 23, 35)

[188] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proc. 20th ACM Symposium on Theory of Computing*, STOC '88, pages 377–387, 1988. (Cited on pages 23, 119)

[189] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Oper. Res.*, 41:338–350, 1993. A preliminary version appeared in [188]. (Cited on pages 23, 30)

[190] J. B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Math. Program.*, 78:109–129, 1997. (Cited on pages 22, 23, 35)

[191] J. B. Orlin. Max flows in $O(nm)$ time, or better. In *Proc. 45th ACM Symposium on Theory of Computing*, STOC '13, pages 765–774, 2013. (Cited on pages 8, 9, 22)

[192] J. B. Orlin, S. A. Plotkin, and É. Tardos. Polynomial dual network simplex algorithms. *Math. Program.*, 60:255–276, 1993. (Cited on pages 22, 23, 35)

[193] M. Pelillo. Heuristics for maximum clique and independent set. In C. A. Floudas and P. M. Pardalos, editors, *Encyclopedia of Optimization*, pages 1508–1520. Springer, Boston, MA, USA, 2009. (Cited on pages 10, 69)

[194] I. T. Peres, H. M. Repolho, R. Martinelli, and N. J. Monteiro. Optimization in inventory-routing problem with planned transshipment: A case study in the retail industry. *Int. J. Prod. Econ.*, 193:748–756, 2017. (Cited on page 3)

[195] S. A. Plotkin and É. Tardos. Improved dual network simplex. In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 367–376, 1990. (Cited on pages 22, 35)

[196] L. F. Portugal, M. G. C. Resende, G. Veiga, J. Patrício, and J. J. Júdice. PDNET – Software for network flow optimization using an interior point method. http://mauricio.resende.info/pdnet/ (accessed Mar. 2019). (Cited on page 53)

[197] L. F. Portugal, M. G. C. Resende, G. Veiga, J. Patrício, and J. J. Júdice. Fortran subroutines for network flow optimization using an interior point algorithm. Technical report TD-5X2SLN, AT&T Labs Research, 2004. (Cited on pages 22, 53, 120)

[198] L. F. Portugal, M. G. C. Resende, G. Veiga, J. Patrício, and J. J. Júdice. Fortran subroutines for network flow optimization using an interior point algorithm. *Pesquisa Operacional*, 28:243–261, 2008. A preliminary version appeared in [197]. (Cited on pages 22, 53)

[199] J. Qian, Z. Zhou, T. Gu, L. Zhao, and L. Chang. Optimal reconfiguration of high-performance VLSI subarrays with network flow. *IEEE Trans. Parallel Distrib. Syst.*, 27:3575–3587, 2016. (Cited on page 3)

[200] A. Quaglino and R. Krause. Towards a multigrid method for the minimum-cost flow problem, 2016. arXiv preprint, arXiv:1612.00201. (Cited on page 41)

[201] S. Rahman, M. Bashton, G. Holliday, R. Schrader, and J. Thornton. Small Molecule Subgraph Detector (SMSD) toolkit. *J. Cheminf.*, 1:12, 2009. (Cited on page 77)

[202] M. Rarey and J. S. Dixon. Feature trees: A new molecular similarity measure based on tree matching. *J. Comput.-Aided Mol. Des.*, 12:471–490, 1998. (Cited on page 69)

[203] J. W. Raymond, E. J. Gardiner, and P. Willett. Heuristics for similarity searching of chemical graphs using a maximum common edge subgraph algorithm. *J. Chem. Inf. Comput. Sci.*, 42:305–316, 2002. (Cited on pages 69, 70, 73)

[204] J. W. Raymond, E. J. Gardiner, and P. Willett. RASCAL: Calculation of graph similarity using maximum common edge subgraphs. *Comput. J.*, 45:631–644, 2002. (Cited on pages 65, 67, 69–71, 73)

[205] J. W. Raymond and P. Willett. Effectiveness of graph-based and fingerprint-based similarity measures for virtual screening of 2D chemical structure databases. *J. Comput.-Aided Mol. Des.*, 16:59–71, 2002. (Cited on page 67)

[206] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *J. Comput.-Aided Mol. Des.*, 16:521–533, 2002. (Cited on pages 65, 67, 69, 73)

[207] M. G. C. Resende and P. M. Pardalos. Interior point algorithms for network flow problems. In J. E. Beasley, editor, *Advances in linear and integer programming*, pages 147–187, 1996. (Cited on pages 22, 53)

[208] M. G. C. Resende and G. Veiga. An efficient implementation of a network interior point method. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, pages 299–348, 1993. (Cited on pages 22, 42)

[209] M. G. C. Resende and G. Veiga. An annotated bibliography of network interior point methods. *Networks*, 42:114–121, 2003. (Cited on pages 22, 53)

[210] H. Röck. Scaling techniques for minimal cost network flows. In U. Pape, editor, *Discrete Structures and Algorithms*, pages 181–191, 1980. (Cited on pages 23, 30)

[211] D. Rogers and M. Hahn. Extended-connectivity fingerprints. *J. Chem. Inf. Model.*, 50:742–754, 2010. (Cited on page 76)

[212] A. Schenker, M. Last, H. Bunke, and A. Kandel. Classification of web documents using graph matching. *Int. J. Pattern Recognit. Artif. Intell.*, 18:475–496, 2004. (Cited on page 65)

[213] A. Schrijver. Paths and flows – a historical survey. *CWI Quarterly*, 6:169–183, 1993. (Cited on page 22)

[214] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Algorithms and Combinatorics, Vol. 24. Springer, Berlin, Germany, 2003. ISBN: 978-3540443896. (Cited on pages 5, 13, 18, 22, 25)

[215] K. Shearer, H. Bunke, and S. Venkatesh. Video indexing and similarity retrieval by largest common subgraph detection using decision trees. *Pattern Recognit.*, 34:1075–1091, 2001. (Cited on page 65)

[216] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Reading, MA, USA, 2002. (Cited on pages 93, 98)

[217] A. Sifaleras. Minimum cost network flows: Problems, algorithms, and software. *Yugosl. J. Oper. Res.*, 23:3–17, 2013. (Cited on pages 13, 22)

[218] D. D. Sleator. An $O(nm \log n)$ algorithm for maximum network flow. Technical report STAN-CS-80-831, Dept. of Computer Science, Stanford University, Stanford, CA, USA, 1980. (Cited on page 9)

[219] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In *Proc. 13th ACM Symposium on Theory of Computing*, STOC '81, pages 114–122, 1981. (Cited on pages 9, 121)

[220] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26:362–391, 1983. A preliminary version appeared in [219]. (Cited on pages 9, 27, 32)

[221] A. Sobih, A. I. Tomescu, and V. Mäkinen. MetaFlow: Metagenomic profiling based on whole-genome coverage analysis with min-cost flows. In M. Singh, editor, *Research in Computational Molecular Biology*, 20th Annual Conference, RECOMB 2016, pages 111–121, 2016. (Cited on page 3)

[222] P. T. Sokkalingam, R. K. Ahuja, and J. B. Orlin. New polynomial-time cycle-canceling algorithms for minimum-cost flows. *Networks*, 36:53–63, 2000. (Cited on pages 23, 25)

[223] V. Srinivasan and G. L. Thompson. Benefit-cost analysis of coding techniques for the primal transportation algorithm. *J. ACM*, 20:194–213, 1973. (Cited on pages 21, 35, 37)

[224] M. Stahl and H. Mauser. Database clustering with a combination of fingerprint and maximum common substructure methods. *J. Chem. Inf. Model.*, 45:542–548, 2005. (Cited on pages 67, 77)

[225] M. Stahl, H. Mauser, M. Tsui, and N. R. Taylor. A robust clustering method for chemical structures. *J. Med. Chem.*, 48:4358–4366, 2005. (Cited on page 67)

[226] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 4th edition, 2013. ISBN: 978-0321563842. (Cited on page 94)

[227] Y. Takahashi, M. Sukekawa, and S. Sasaki. Automatic identification of molecular similarity using reduced-graph representation of chemical structure. *J. Chem. Inf. Comput. Sci.*, 32:639–643, 1992. (Cited on pages 69, 73)

[228] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5:247–255, 1985. (Cited on pages 20, 21, 23)

[229] R. E. Tarjan. Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem. *Math. Oper. Res.*, 16:272–291, 1991. (Cited on pages 22, 35)

[230] R. E. Tarjan. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. *Math. Program.*, 78:169–177, 1997. (Cited on pages 22, 23, 35)

[231] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proc. 35th ACM Symposium on Theory of Computing*, STOC '03, pages 149–158, 2003. (Cited on pages 6, 122)

[232] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Syst. Sci.*, 69:330–353, 2004. A preliminary version appeared in [231]. (Cited on page 6)

[233] N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1:173–194, 1971. (Cited on pages 21, 23, 27)

[234] C. Tonnelier, P. Jauffret, T. Hanser, and G. Kaufmann. Machine learning of generic reactions: 3. An efficient algorithm for maximal common substructure determination. *Tetrahedron Comput. Method.*, 3:351–358, 1990. (Cited on page 67)

[235] N. Trinajstic. *Chemical Graph Theory*. CRC Press, London, UK, 1992. ISBN: 0-8493-4256-2. (Cited on page 66)

[236] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23:31–42, 1976. (Cited on page 11)

[237] C. L. D. S. Vieira, M. M. M. Luna, and J. M. Azevedo. Minimum-cost flow algorithms: A performance evaluation using the Brazilian road network. *World Rev. Int. Transport. Res.*, 8:3–21, 2019. (Cited on page 3)

[238] J. Vygen. On dual minimum cost flow algorithms. In *Proc. 32nd ACM Symposium on Theory of Computing*, STOC '00, pages 117–125, 2000. (Cited on pages 23, 123)

[239] J. Vygen. On dual minimum cost flow algorithms. *Math. Meth. Oper. Res.*, 56:101–126, 2002. A preliminary version appeared in [238]. (Cited on page 23)

[240] T. Wang and J. Zhou. EMCSS: A new method for maximal common substructure search. *J. Chem. Inf. Comput. Sci.*, 37:828–834, 1997. (Cited on page 69)

[241] Y. Wang, T. W. H. Backman, K. Horan, and T. Girke. fmcsR: Mismatch tolerant maximum common substructure searching in R. *Bioinformatics*, 29:2792–2794, 2013. (Cited on page 67)

[242] Y. Wang, D. Zhang, Q. Liu, F. Shen, and L. H. Lee. Towards enhancing the last-mile delivery: An effective crowd-tasking model with scalable solutions. *Transport. Res. Part E: Logist. Transport. Rev.*, 93:279–293, 2016. (Cited on page 3)

[243] A. Weintraub. A primal algorithm to solve network flow problems with convex costs. *Manage. Sci.*, 21:87–97, 1974. (Cited on page 25)

[244] H. Whitney. Congruent graphs and the connectivity of graphs. *Am. J. Math.*, 54:150–168, 1932. (Cited on page 12)

[245] P. Willett. Similarity-based approaches to virtual screening. *Biochem. Soc. Trans.*, 31:603–606, 2003. (Cited on page 67)

[246] P. Willett. Searching techniques for databases of two- and three-dimensional chemical structures. *J. Med. Chem.*, 48:4183–4199, 2005. (Cited on pages 11, 66, 67)

[247] P. Willett, J. M. Barnard, and G. M. Downs. Chemical similarity searching. *J. Chem. Inf. Comput. Sci.*, 38:983–996, 1998. (Cited on page 67)

[248] Q. Wu and J.-K. Hao. A review on algorithms for maximum clique problems. *Eur. J. Oper. Res.*, 242:693–709, 2015. (Cited on pages 10, 69)

[249] M. A. Yakovleva. A problem on minimum transportation cost. In V. S. Nemchinov, editor, *Applications of Mathematics in Economic Research*, pages 390–399, Moscow, Russia, 1959. (Cited on pages 21, 23)

[250] N. Zadeh. A bad network problem for the simplex method and other minimum cost flow algorithms. *Math. Program.*, 5:255–266, 1973. (Cited on page 21)

[251] N. Zadeh. More pathological examples for network flow problems. *Math. Program.*, 5:217–224, 1973. (Cited on page 21)

[252] B. Zhang, M. Vogt, G. M. Maggiora, and J. Bajorath. Design of chemical space networks using a Tanimoto similarity variant based upon maximum common substructures. *J. Comput.-Aided Mol. Des.*, 29:937–950, 2015. (Cited on page 67)

Abstract

This thesis presents efficient algorithms for solving complex combinatorial optimization problems related to graphs. The main contribution of this work is the development of various improvements for different solution methods, including novel heuristics and special representations of graph and tree structures. Extensive experimental studies have verified that the most efficient algorithms of the author are usually faster or yield better results than other available implementations.

Chapter 1 is a brief introduction, while Chapter 2 discusses the necessary mathematical concepts and results.

Chapter 3 presents seven different algorithms and several practical improvements for solving the minimum-cost flow problem, which is a fundamental graph optimization task with diverse applications in various fields. A comprehensive computational study was carried out to compare these algorithms with eight other solvers, including the most widely used and acknowledged ones. The network simplex code of the author turned out to be significantly more efficient and robust than the other implementations of this method. Furthermore, it is the fastest algorithm on the majority of the problem instances. The presented cost-scaling implementation is also highly efficient; it outperforms the network simplex methods on large sparse networks.

Chapter 4 discusses the maximum common subgraph problem in the context of cheminformatics. We developed several heuristics for improving the accuracy and efficiency of two solution methods, as well as for achieving chemically more relevant mapping of the atoms and bonds of the input molecular graphs. The presented algorithms were compared with two other solvers, to which they turned out to be superior. The author conducted this part of his research at ChemAxon, together with Péter Englert. The developed methods have been integrated into multiple software solutions of the company, which are used by leading international companies in the pharmaceutical industry.

Chapter 5 briefly introduces the open-source C++ graph optimization library called LEMON, which includes the algorithms presented in Chapter 3. Over the years, the author has made significant contribution to the development of LEMON, in cooperation with Alpár Jüttner, the leader of the project, and Balázs Dezső. In particular, the minimum-cost flow algorithms have played an important role in the increasing popularity of the library.

Finally, Chapter 6 summarizes the contribution of this work. The presented results and discussions are based on four journal articles, which have more than 300 independent citations in total according to the database of Google Scholar.

Összefoglalás

A doktori értekezés hatékony algoritmusokat mutat be gráfokon értelmezett nehéz kombinatorikus optimalizálási feladatok megoldására. A kutatás legfontosabb eredményét különböző megoldási módszerekhez kidolgozott javítások jelentik, amelyek magukban foglalnak új heurisztikákat, valamint gráfok és fák speciális reprezentációit is. Az elvégzett elemzések igazolták, hogy a szerző által adott leghatékonyabb algoritmusok az esetek többségében gyorsabbak, illetve jobb eredményeket adnak, mint más elérhető implementációk.

Az 1. fejezet egy rövid bevezetés, a 2. fejezet pedig a szükséges matematikai fogalmakat és eredményeket tárgyalja.

A 3. fejezet hét különböző algoritmust és számos hasznos javítást mutat be a minimális költségű folyam feladatra, amely a legtöbbet vizsgált és alkalmazott gráfoptimalizálási problémák egyike. Az implementációinkat egy átfogó tapasztalati elemzés keretében összehasonlítottuk nyolc másik megoldóprogrammal, köztük a leggyakrabban használt és legelismertebb implementációkkal. A hálózati szimplex algoritmusunk lényegesen hatékonyabbnak és robusztusabbnak bizonyult, mint a módszer más implementációi, továbbá a legtöbb tesztadaton ez az algoritmus a leggyorsabb. A bemutatott költségskálázó algoritmus szintén rendkívül hatékonynak bizonyult; nagy méretű ritka gráfokon felülmúlja a hálózati szimplex implementációkat.

A 4. fejezet a legnagyobb közös részgráf problémát vizsgálja kémiai alkalmazások szempontjából. Hatékony heurisztikákat dolgoztunk ki, amelyek jelentősen javítják két megoldási módszer pontosságát és sebességét, valamint kémiailag relevánsabb módon rendelik egymáshoz molekulagráfok atomjait és kötéseit. Az algoritmusainkat összehasonlítottuk két ismert megoldóprogrammal, amelyeknél lényegesen jobb eredményeket sikerült elérnünk. Ezt a kutatómunkát a szerző Englert Péterrel közösen, a ChemAxon Kft. alkalmazottjaként végezte. A kifejlesztett implementációk bekerültek a cég több szoftvertermékébe, amelyek vezető nemzetközi gyógyszercégek használatában állnak.

Az 5. fejezet röviden bemutatja a LEMON nevű nyílt forrású C++ gráfoptimalizációs programkönyvtárat, amely magában foglalja a 3. fejezetben tárgyalt algoritmusokat is. A szerző kutatómunkája során jelentős szerepet vállalt a LEMON fejlesztésében, a projektet vezető Jüttner Alpárral, valamint Dezső Balázzsal együttműködve. A minimális költségű folyam feladatra adott algoritmusok nagy mértékben hozzájárultak a programcsomag népszerűségének növekedéséhez.

Végül a 6. fejezet összefoglalja az értekezést. A bemutatott munka eredményei négy folyóiratcikken jelentek meg, amelyekre összesen több mint 300 független hivatkozás történt a Google Scholar adatbázisa szerint.

[1]ADATLAP
a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai
A szerző neve: **Kovács Péter**
MTMT-azonosító: **10033546**
A doktori értekezés címe és alcíme: **Efficient Algorithms for Graph Optimization Problems**
DOI-azonosító[2]: **10.15476/ELTE.2019.029**
A doktori iskola neve: **ELTE Informatika Doktori Iskola**
A doktori iskolán belüli doktori program neve: **Az Informatika Alapjai és Módszertana**
A témavezető neve és tudományos fokozata: **Király Zoltán, PhD (egyetemi docens)**
A témavezető munkahelye: **ELTE TTK Számítógéptudományi Tanszék**

II. Nyilatkozatok

1. A doktori értekezés szerzőjeként[3]

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét, Kulcsár Adinát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[4]

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (*dátum*)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[5]

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.[6]

2. A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: **Budapest, 2019. február 8.**

Kovács Péter
a doktori értekezés szerzőjének aláírása

[1] Beiktatta az Egyetemi Doktori Szabályzat módosításáról szóló CXXXIX/2014. (VI. 30.) Szen. sz. határozat. Hatályos: 2014. VII.1. napjától.

[2] A kari hivatal ügyintézője tölti ki.

[3] A megfelelő szöveg aláhúzandó.

[4] A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.

[5] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.

[6] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.