Lecture Notes of the Master Course:

Discrete Optimization

Utrecht University
Academic Year 2011/2012

Course website: http://www.cwi.nl/~schaefer/courses/lnmb-do11

Prof. dr. Guido Schäfer

Center for Mathematics and Computer Science (CWI)
Algorithms, Combinatorics and Optimization Group
Science Park 123, 1098 XG Amsterdam, The Netherlands

VU University Amsterdam
Department of Econometrics and Operations Research
De Boelelaan 1105, 1081 HV Amsterdam, The Netherlands

Website: http://www.cwi.nl/~schaefer
Email: g.schaefer@cwi.nl

Document last modified:
January 10, 2012

Disclaimer: These notes contain (most of) the material of the lectures of the course "Discrete Optimization", given at Utrecht University in the academic year 2011/2012. The course is organized by the Dutch Network on the Mathematics of Operations Research (LNMB) and is part of the Dutch Master's Degree Programme in Mathematics (Mastermath). Note that the lecture notes have undergone some rough proof-reading only. Please feel free to report any typos, mistakes, inconsistencies, etc. that you observe by sending me an email (g.schaefer@cwi.nl).

Note that these Lecture Notes have been last modified on January 10, 2012. I guess most of you will have printed the Lecture Notes of December 6, 2011 (see "document last modified" date on your printout). I therefore keep track of the changes that I made since then below. I also mark these changes as "[major]" or "[minor]", depending on their importance.

Changes made with respect to Lecture Notes of December 6, 2011:

- [**major**] Introducing the notion of a pseudoflow on page 42: corrected "it need to satisfy the flow balance constraints" to "it need *not* satisfy the flow balance constraints".

Changes made with respect to Lecture Notes of December 20, 2011: (Thanks to Rutger Kerkkamp for pointing these out.)

- [minor] Symbol for symmetric difference on page 4 is now the same as the one used in Chapter 7 ("$\triangle$").
- [minor] Strictly speaking we would have to add a constraint $x_j \leq 1$ for every $j \in \{1, \ldots, n\}$ to the LP relaxation (2) of the integer linear program (1) on page 5. However, these constraints are often (but not always) redundant because of the minimization objective. Note that the discussion that follows refers to the LP relaxation as stated in (2) (see also remark after statement of LP (2)).
- [minor] Corrected "multiplies" to "multipliers" in last paragraph on page 5.
- [minor] Lower part of Figure 2 on page 2: changed edge e from solid to dotted line.
- [minor] At various places in Chapter 3, Condition (M1) of the independent set system has not been mentioned explicitly (Example 3.1, Examples 3.3–3.5, Theorem 3.1). Mentioned now.
- [minor] Corrected "many application" to "many applications" in first paragraph on page 27.
- [minor] Corrected "$\delta(v)$" and "$\delta(u)$" to "$\delta(s,v)$" and "$\delta(s,u)$" in last paragraph of the proof of Lemma 5.5. on page 34 (3 occurrences).
- [minor] Algorithms 7 and 8 should mention that the capacity function is non-negative.
- [**major**] There was a mistake in the objective function of the dual linear program (6) on page 46: The sign in front of the second summation must be negative.
- [minor] Algorithms 9, 10 and 11 should mention that the capacity function is non-negative.
- [minor] Proof of Theorem 7.3 on page 56: corrected "$O(m\alpha(n,m))$" to "$O(m\alpha(n,\frac{m}{n}))$".
- [minor] Caption of the illustration on page 71 has been put on the same page.
- [minor] Typo in the second last sentence of the proof of Theorem 6.7 on page 47: "$c^{\pi}(u,v) < 0$" corrected to "$c^{\pi}(u,v) \leq 0$".
- [minor] Statement of Theorem 10.8 on page 90 has been corrected (FPTAS instead of 2-approximation).

Changes made with respect to Lecture Notes of January 5, 2012: (Thanks to Tara van Zalen and Arjan Dijkstra for pointing these out.)

- [minor] Example 3.2 on page 21 (uniform matroids): corrected "$\mathcal{I} = \{I \subseteq S \mid |S| \leq k\}$" to "$\mathcal{I} = \{I \subseteq S \mid |I| \leq k\}$"
- [minor] Statement of Lemma 5.4 on page 32: corrected "The flow of any flow" to "The flow value of any flow".
- [minor] There was a minus sign missing in the equation (7) on page 43.
- [minor] Second paragraph of Section 8.3 on page 60: Corrected "hyperplance" to "hyperplane".
- [minor] Removed sentence "We state the following proposition without proof" before statement of Lemma 8.4 on page 62.
- Corrected "Note hat b is integral" to "Note that b is integral" in the proof of Theorem 8.7 on page 63.

Please feel free to report any mistakes, inconsistencies, etc. that you encounter.

Guido

Contents

1. Preliminaries

1.1 Optimization Problems

We first formally define what we mean by an *optimization problem*. The definition below focusses on *minimization problems*. Note that it extends naturally to *maximization problems*.

Definition 1.1. A *minimization problem* Π is given by a set of instances $\mathcal{I}$. Each instance $I \in \mathcal{I}$ specifies

- a set $\mathcal{F}$ of feasible solutions for I;
- a cost function $c : \mathcal{F} \to \mathbb{R}$.

Given an instance $I = (\mathcal{F}, c) \in \mathcal{I}$, the goal is to find a feasible solution $S \in \mathcal{F}$ such that $c(S)$ is minimum. We call such a solution an *optimal solution* of I.

In *discrete (or combinatorial) optimization* we concentrate on optimization problems Π, where for every instance $I = (\mathcal{F}, c)$ the set $\mathcal{F}$ of feasible solutions is *discrete*, i.e., $\mathcal{F}$ is finite or countably infinite. We give some examples below.

Minimum Spanning Tree Problem (MST):

Given:	An undirected graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}$.
Goal:	Find a spanning tree of G of minimum total cost.

We have

$$\mathcal{F} = \{T \subseteq E \mid T \text{ is a spanning tree of } G\} \quad \text{and} \quad c(T) = \sum_{e \in T} c(e).$$

Traveling Salesman Problem (TSP):

Given:	An undirected graph $G = (V, E)$ with distances $d : E \to \mathbb{R}$.
Goal:	Find a tour of G of minimum total length.

Here we have

$$\mathcal{F} = \{T \subseteq E \mid T \text{ is a tour of } G\} \quad \text{and} \quad c(T) = \sum_{e \in T} d(e)$$

Linear Programming (LP):

Given: A set $\mathcal{F}$ of feasible solutions $x = (x_1, \ldots, x_n)$ defined by m linear constraints

$$\mathcal{F} = \left\{ (x_1, \ldots, x_n) \in \mathbb{R}^n_{\geq 0} \;\middle|\; \sum_{i=1}^n a_{ij} x_i \geq b_j \quad \forall j = 1, \ldots, m \right\}$$

and an objective function $c(x) = \sum_{i=1}^n c_i x_i$.

Goal: Find a feasible solution $x \in \mathcal{F}$ that minimizes $c(x)$.

Note that in this example the number of feasible solution in $\mathcal{F}$ is uncountable. So why does this problem qualify as a *discrete* optimization problem? The answer is that $\mathcal{F}$ defines a feasible set that corresponds to the convex hull of a finite number of vertices. It is not hard to see that if we optimize a linear function over a convex hull then there always exists an optimal solution that is a vertex. We can thus equivalently formulate the problem as finding a vertex x of the convex hull defined by $\mathcal{F}$ that minimizes $c(x)$.

1.2 Algorithms and Efficiency

Intuitively, an *algorithm* for an optimization problem Π is a sequence of instructions specifying a computational procedure that solves every given instance I of Π. Formally, the computational model underlying all our considerations is the one of a *Turing machine* (which we will not define formally here).

A main focus of this course is on *efficient* algorithms. Here, efficiency refers to the overall running time of the algorithm. We actually do not care about the actual running *time* (in terms of minutes, seconds, etc.), but rather about the number of basic operations. Certainly, there are different ways to represent the overall running time of an algorithm. The one that we will use here (and which is widely used in the algorithms community) is the so-called *worst-case* running time. Informally, the worst-case running time of an algorithm measures the running time of an algorithm on the worst possible input instance (of a given size).

There are at least two advantages in assessing the algorithm's performance by means of its worst-case running time. First, it is usually rather easy to estimate. Second, it provides a very strong performance guarantee: The algorithm is guaranteed to compute a solution to *every* instance (of a given size), using no more than the stated number of basic operations. On the downside, the worst-case running time of an algorithm might be an overly pessimistic estimation of its actual running time. In the latter case, assessing the performance of an algorithm by its *average case* running time or its *smoothed* running time might be suitable alternatives.

Usually, the running time of an algorithm is expressed as a function of the *size* of the input instance I. Note that a-priori it is not clear what is meant by the size of I because there are different ways to represent (or encode) an instance.

Example 1.1. Many optimization problems have a graph as input. Suppose we are given an undirected graph $G = (V, E)$ with n nodes and m edges. One way of representing G is by its $n \times n$ adjacency matrix $A = (a_{ij})$ with $a_{ij} = 1$ if $(i, j) \in E$ and $a_{ij} = 0$ otherwise. The size needed to represent G by its adjacency matrix is thus n^2. Another way to represent G is by its *adjacency lists*: For every node $i \in V$, we maintain the set $L_i \subseteq V$ of nodes that are adjacent to i in a list. Note that each edge occurs on two adjacency lists. The size to represent G by adjacency lists is $n + 2m$.

The above example illustrates that the size of an instance depends on the underlying *data structure* that is used to represent the instance. Depending on the kind of operations that an algorithm uses, one might be more efficient than the other. For example, checking whether a given edge (i, j) is part of G takes constant time if we use the adjacency matrix, while it takes time $|L_i|$ (or $|L_j|$) if we use the adjacency lists. On the other hand, listing all edges incident to i takes time n if we use the adjacency matrix, while it takes time $|L_i|$ if we use the adjacency lists.

Formally, we define the size of an instance I as the number of bits that are needed to store all data of I using encoding L on a digital computer and use $|L(I)|$ to refer to this number. Note that according to this definition we also would have to account for the number of bits needed to store the numbers associated with the instance (like nodes or edges). However, most computers nowadays treat all integers in their range, say from 0 to 2^{31}, the same and allocate a *word* to each such number. We therefore often take the freedom to rely on a more intuitive definition of size by counting the number of objects (like nodes or edges) of the instance rather than their total binary length.

Definition 1.2. Let Π be an optimization problem and let L be an encoding of the instances. Then algorithm ALG solves Π in (worst-case) running time f if ALG computes for every instance I of size $n_I = |L(I)|$ an optimal solution $S \in \mathcal{F}$ using at most $f(n_I)$ basic operations.

1.3 Growth of Functions

We are often interested in the *asymptotic running* time of the algorithm. The following definitions will be useful.

Definition 1.3. Let $g : \mathbb{N} \to \mathbb{R}^+$. We define

$$\begin{aligned}
O(g(n)) &= \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0,\ n_0 \in \mathbb{N} \text{ such that } f(n) \le c \cdot g(n)\ \forall n \ge n_0\} \\
\Omega(g(n)) &= \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0,\ n_0 \in \mathbb{N} \text{ such that } f(n) \ge c \cdot g(n)\ \forall n \ge n_0\} \\
\Theta(g(n)) &= \{f : \mathbb{N} \to \mathbb{R}^+ \mid f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))\}
\end{aligned}$$

We will often write, e.g., $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$, even though this is notationally somewhat imprecise.

We consider a few examples: We have $10n^2 = O(n^2)$, $\frac{1}{2}n^2 = \Omega(n^2)$, $10n \log n = \Omega(n)$, $10n \log n = O(n^2)$, $2^{n+1} = \Theta(2^n)$ and $O(\log m) = O(\log n)$[1] if $m \le n^c$ for some constant c.

1.4 Graphs

An undirected graph G consists of a finite set $V(G)$ of nodes (or vertices) and a finite set $E(G)$ of edges. For notational convenience, we will also write $G = (V, E)$ to refer to a graph with nodes set $V = V(G)$ and edge set $E = E(G)$. Each edge $e \in E$ is associated with an *unordered* pair $(u, v) \in V \times V$; u and v are called the *endpoints* of e. If two edges have the same endpoints, then they are called *parallel* edges. An edge whose endpoints

[1]Recall that $\log(n^c) = c \log(n)$.

are the same is called a *loop*. A graph that has neither parallel edges nor loops is said to be *simple*. Note that in a simple graph every edge $e = (u, v) \in E$ is uniquely identified by its endpoints u and v. Unless stated otherwise, we assume that undirected graphs are simple. We denote by n and m the number of nodes and edges of G, respectively. A *complete* graph is a graph that contains an edge for every (unordered) pair of nodes. That is, a complete graph has $m = n(n-1)/2$ edges.

A *subgraph* H of G is a graph such that $V(H) \subseteq V$ and $E(H) \subseteq E$ and each $e \in E(H)$ has the same endpoints in H as in G. Given a subset $V' \subseteq V$ of nodes and a subset $E' \subseteq E$ of edges of G, the subgraph H of G induced by V' and E' is defined as the (unique) subgraph H of G with $V(H) = V'$ and $E(H) = E'$. Given a subset $E' \subseteq E$, $G \setminus E'$ refers to the subgraph H of G that we obtain if we delete all edges in E' from G, i.e., $V(H) = V$ and $E(H) = E \setminus E'$. Similarly, given a subset $V' \subseteq V$, $G \setminus V'$ refers to the subgraph of G that we obtain if we delete all nodes in V' and its incident edges from G, i.e., $V(H) = V \setminus V'$ and $E(H) = E \setminus \{(u, v) \in E \mid u \in V'\}$. A subgraph H of G is said to be *spanning* if it contains all nodes of G, i.e., $V(H) = V$.

A *path* P in an undirected graph G is a sequence $P = \langle v_1, \dots, v_k \rangle$ of nodes such that $e_i = (v_i, v_{i+1})$ $(1 \le i < k)$ is an edge of G. We say that P is a path *from* v_1 *to* v_k, or a v_1, v_k*-path*. P is *simple* if all v_i $(1 \le i \le k)$ are distinct. Note that if there is a v_1, v_k-path in G, then there is a simple one. Unless stated otherwise, the *length* of P refers to the number of edges of P. A path $C = \langle v_1, \dots, v_k = v_1 \rangle$ that starts and ends in the same node is called a *cycle*. C is *simple* if all nodes $v_1, \dots, v_{k-1}$ are distinct. A graph is said to be *acyclic* if it does not contain a cycle.

A *connected component* $C \subseteq V$ of an undirected graph G is a maximal subset of nodes such that for every two nodes $u, v \in C$ there is a u, v-path in G. A graph G is said to be *connected* if for every two nodes $u, v \in V$ there is a u, v-path in G. A connected subgraph T of G that does not contain a cycle is called a *tree* of G. A *spanning* tree T of G is a tree of G that contains all nodes of G. A subgraph F of G is a *forest* if it consists of a (disjoint) union of trees.

A *directed* graph $G = (V, E)$ is defined analogously with the only difference that edges are directed. That is, every edge e is associated with an *ordered* pair $(u, v) \in V \times V$. Here u is called the *source* (or *tail*) of e and v is called the *target* (or *head*) of e. Note that, as opposed to the undirected case, edge (u, v) is different from edge (v, u) in the directed case. All concepts introduced above extend in the obvious way to directed graphs.

1.5 Sets, etc.

Let S be a set and $e \notin S$. We will write $S + e$ as a short for $S \cup \{e\}$. Similarly, for $e \in S$ we write $S - e$ as a short for $S \setminus \{e\}$.

The *symmetric difference* of two sets S and T is defined as $S \triangle T = (S \setminus T) \cup (T \setminus S)$.

We use $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ to refer to the set of natural, integer, rational and real numbers, respectively. We use $\mathbb{Q}^+$ and $\mathbb{R}^+$ to refer to the nonnegative rational and real numbers, respectively.

1.6 Basics of Linear Programming Theory

Many optimization problems can be formulated as an *integer linear program (ILP)*. Let Π be a minimization problem. Then Π can often be formulated as follows:

$$\begin{array}{lrcll} \text{minimize} & \sum_{j=1}^{n} c_j x_j & & & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j & \geq & b_i & \forall i \in \{1,\dots,m\} \\ & x_j & \in & \{0,1\} & \forall j \in \{1,\dots,n\} \end{array} \tag{1}$$

Here, x_j is a decision variable that is either set to 0 or 1. The above ILP is therefore also called a *0/1-ILP*. The coefficients a_{ij}, b_i and c_j are given rational numbers.

If we relax the integrality constraint on x_j, we obtain the following *LP-relaxation* of the above ILP (1):

$$\begin{array}{lrcll} \text{minimize} & \sum_{j=1}^{n} c_j x_j & & & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j & \geq & b_i & \forall i \in \{1,\dots,m\} \\ & x_j & \geq & 0 & \forall j \in \{1,\dots,n\} \end{array} \tag{2}$$

In general, we would have to enforce that $x_j \leq 1$ for every $j \in \{1,\dots,n\}$ additionally. However, these constraints are often redundant because of the minimization objective and this is what we assume subsequently. Let OPT and $\mathsf{OPT}_{\mathrm{LP}}$ refer to the objective function values of an optimal integer and fractional solution to the ILP (1) and LP (2), respectively. Because every integer solution to (1) is also a feasible solution for (2), we have $\mathsf{OPT}_{\mathrm{LP}} \leq \mathsf{OPT}$. That is, the optimal fractional solution provides a lower bound on the optimal integer solution. Recall that establishing a lower bound on the optimal cost is often the key to deriving good approximation algorithms for the optimization problem. The techniques that we will discuss subsequently exploit this observation in various ways.

Let (x_j) be an arbitrary feasible solution. Note that (x_j) has to satisfy each of the m constraints of (2). Suppose we multiply each constraint $i \in \{1,\dots,m\}$ with a non-negative value y_i and add up all these constraints. Then

$$\sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i.$$

Suppose further that the multipliers y_i are chosen such that $\sum_{i=1}^{m} a_{ij} y_i \leq c_j$. Then

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i \tag{3}$$

That is, every such choice of multipliers establishes a lower bound on the objective function value of (x_j). Because this holds for an arbitrary feasible solution (x_j) it also holds for the optimal solution. The problem of finding the best such multipliers (providing the largest lower bound on $\mathsf{OPT}_{\mathrm{LP}}$) corresponds to the so-called *dual program* of (2).

$$\begin{array}{lrcll} \text{maximize} & \sum_{i=1}^{m} b_i y_i & & & \\ \text{subject to} & \sum_{i=1}^{m} a_{ij} y_i & \leq & c_j & \forall j \in \{1,\dots,n\} \\ & y_i & \geq & 0 & \forall i \in \{1,\dots,m\} \end{array} \tag{4}$$

We use $\mathsf{OPT}_{\mathrm{DP}}$ to refer to the objective function value of an optimal solution to the dual linear program (4).

There is a strong relation between the primal LP (2) and its corresponding dual LP (4). Note that (3) shows that the objective function value of an arbitrary feasible dual solution (y_i) is less than or equal to the objective function value of an arbitrary feasible primal solution (x_j). In particular, this relation also holds for the optimal solutions and thus $\mathsf{OPT}_{\mathrm{DP}} \leq \mathsf{OPT}_{\mathrm{LP}}$. This is sometimes called *weak duality*. From linear programming theory, we know that even a stronger relation holds:

Theorem 1.1 (strong duality)**.** *Let $x = (x_j)$ and $y = (y_i)$ be feasible solutions to the LPs (2) and (4), respectively. Then x and y are optimal solutions if and only if*

$$\sum_{j=1}^{n} c_j x_j = \sum_{i=1}^{m} b_i y_i.$$

An alternative characterization is given by the *complementary slackness conditions*:

Theorem 1.2. *Let $x = (x_j)$ and $y = (y_i)$ be feasible solutions to the LPs (2) and (4), respectively. Then x and y are optimal solutions if and only if the following conditions hold:*

1. *Primal complementary slackness conditions: for every $j \in \{1,\dots,n\}$, either $x_j = 0$ or the corresponding dual constraint is tight, i.e.,*

$$\forall j \in \{1,\dots,n\}: \qquad x_j > 0 \quad \Rightarrow \quad \sum_{i=1}^{m} a_{ij} y_i = c_j.$$

2. *Dual complementary slackness conditions: for every $i \in \{1,\dots,m\}$, either $y_i = 0$ or the corresponding primal constraint is tight, i.e.,*

$$\forall i \in \{1,\dots,m\}: \qquad y_i > 0 \quad \Rightarrow \quad \sum_{j=1}^{n} a_{ij} x_j = b_i.$$

2. Minimum Spanning Trees

2.1 Introduction

We consider the *minimum spanning tree problem (MST)*, which is one of the simplest and most fundamental problems in network optimization:

Minimum Spanning Tree Problem (MST):

Given:	An undirected graph $G = (V, E)$ and edge costs $c : E \to \mathbb{R}$.
Goal:	Find a spanning tree T of G of minimum total cost.

Recall that T is a *spanning tree* of G if T is a spanning subgraph of G that is a tree. The cost $c(T)$ of a tree T is defined as $c(T) = \sum_{e \in T} c(e)$. Note that we can assume without loss of generality that G is connected because otherwise no spanning tree exists.

If all edges have non-negative costs, then the *MST* problem is equivalent to the *connected subgraph problem* which asks for the computation of a minimum cost subgraph H of G that connects all nodes of G.

2.2 Coloring Procedure

Most known algorithms for the *MST* problem belong to the class of *greedy algorithms*. From a high-level point of view, such algorithms iteratively extend a partial solution to the problem by always adding an element that causes the minimum cost increase in the objective function. While in general greedy choices may lead to suboptimal solutions, such choices lead to an optimal solution for the *MST* problem.

We will get to know different greedy algorithms for the *MST* problem. All these algorithms can be described by means of an *edge-coloring process*: Initially, all edges are uncolored. In each step, we then choose an uncolored edge and color it either *red* (meaning that the edge is rejected) or *blue* (meaning that the edge is accepted). The process ends if there are no uncolored edges. Throughout the process, we make sure that we maintain the following *color invariant*:

Invariant 2.1 (Color invariant)**.** There is a minimum spanning tree containing all the blue edges and none of the red edges.

The coloring process can be seen as maintaining a forest of *blue trees*. Initially, the forest consists of n isolated blue trees corresponding to the nodes in V. The edges are then iteratively colored red or blue. If an edge is colored blue, then the two blue trees containing the endpoints of this edge are combined into one new blue tree. If an edge is colored red, then this edge is excluded from the blue forest. The color invariant ensures that the forest of blue trees can always be extended to a minimum spanning tree (by using some of the uncolored edges and none of the red edges). Note that the color invariant ensures that the final set of blue edges constitutes a minimum spanning tree.

We next introduce two coloring rules on which our algorithms are based. We first need to introduce the notion of a *cut*. Let $G = (V,E)$ be an undirected graph. A *cut* of G is a partition of the node set V into two sets: X and $\bar{X} = V \setminus X$. An edge $e = (u,v)$ is said to *cross* a cut $(X,\bar{X})$ if its endpoints lie in different parts of the cut, i.e., $u \in X$ and $v \in \bar{X}$. Let $\delta(X)$ refer to the set of all edges that cross $(X,\bar{X})$, i.e.,

$$\delta(X) = \{(u,v) \in E \mid u \in X,\ v \in V \setminus X\}.$$

Note that $\delta(\cdot)$ is symmetric, i.e., $\delta(X) = \delta(\bar{X})$.

We can now formulate the two coloring rules:

Blue rule: Select a cut $(X,\bar{X})$ that is not crossed by any blue edge. Among the uncolored edges in $\delta(X)$, choose one of minimum cost and color it blue.

Red rule: Select a simple cycle C that does not contain any red edge. Among the uncolored edges in C, choose one of maximum cost and color it red.

Our greedy algorithm is free to apply any of the two coloring rules in an arbitrary order until all edges are colored either red or blue. The next theorem proves correctness of the algorithm.

Theorem 2.1. *The greedy algorithm maintains the color invariant in each step and eventually colors all edges.*

Proof. We show by induction on the number t of steps that the algorithm maintains the color invariant. Initially, no edges are colored and thus the color invariant holds true for $t = 0$ (recall that we assume that G is connected and thus a minimum spanning tree exists). Suppose the color invariant holds true after $t-1$ steps ($t \geq 1$). Let T be a minimum spanning tree satisfying the color invariant (after step $t-1$).

Assume that in step t we color an edge e using the blue rule. If $e \in T$, then T satisfies the color invariant after step t and we are done. Otherwise, $e \notin T$. Consider the cut $(X,\bar{X})$ to which the blue rule is applied to color $e = (u,v)$ (see Figure 1). Because T is a spanning tree, there is a path P_{uv} in T that connects the endpoints u and v of e. At least one edge, say e', of P_{uv} must cross $(X,\bar{X})$. Note that e' cannot be red because T satisfies the color invariant. Also e' cannot be blue because of the pre-conditions of applying the blue rule. Thus, e' is uncolored and by the choice of e, $c(e) \leq c(e')$. By removing e' from T and adding e, we obtain a new spanning tree $T' = (T - e') + e$ of cost $c(T') = c(T) - c(e') + c(e) \leq c(T)$. Thus, T' is a minimum spanning tree that satisfies the color invariant after step t.

Assume that in step t we color an edge e using the red rule. If $e \notin T$, the T satisfies the color invariant after step t and we are done. Otherwise, $e \in T$. Consider the cycle C to which the red rule is applied to color $e = (u,v)$ (see Figure 2). By removing e from T, we obtain two trees whose node sets induce a cut $(X,\bar{X})$. Note that e crosses $(X,\bar{X})$. Because C is a cycle, there must exist at least one other edge, say e', in C that crosses $(X,\bar{X})$. Note that e' cannot be blue because $e' \notin T$ and the color invariant. Moreover, e' cannot be red because of the pre-conditions of applying the red rule. Thus, e' is uncolored and by the choice of e, $c(e) \geq c(e')$. By removing e from T and adding e', we obtain a new spanning

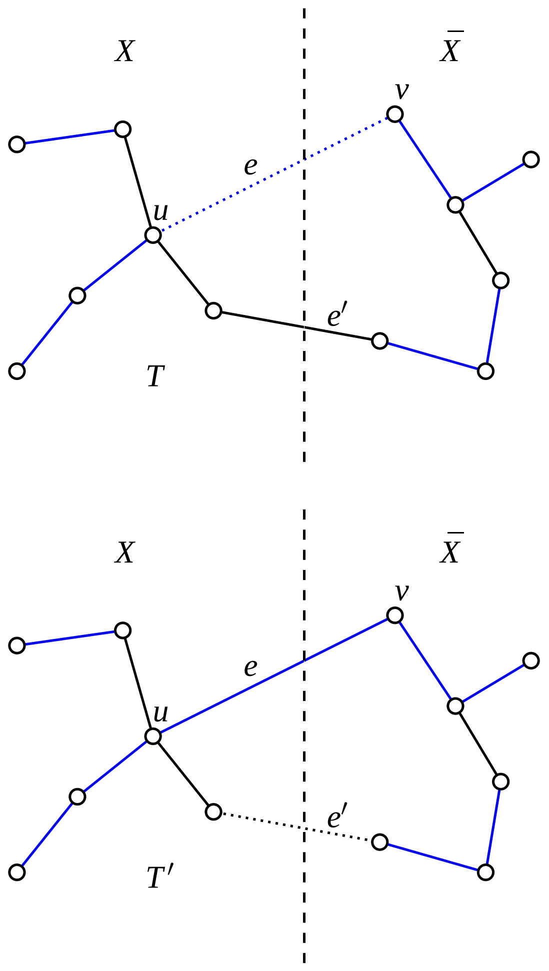


Figure 1: Illustration of the exchange argument used in the proof of Theorem 2.1 (blue rule).

tree $T' = (T - e) + e'$ of cost $c(T') = c(T) - c(e) + c(e') \leq c(T)$. Thus, T' is a minimum spanning tree that satisfies the color invariant after step t.

Finally, we show that eventually all edges are colored. Suppose the algorithm stops because neither the blue rule nor the red rule applies but there is still some uncolored edge $e = (u, v)$. By the color invariant, the blue edges constitute a forest of blue trees. If both endpoints u and v of e are part of the same blue tree T, then we can apply the red rule to the cycle induced by the unique path P_{uv} from u to v in T and e to color e red. If the endpoints u and v are contained in two different blue trees, say T_u and T_v, then the node set of one of these trees, say $X = V(T_u)$, induces a cut $(X, \bar{X})$ to which the blue rule can be applied to color an uncolored edge (which must exist because of the presence of e). Thus an uncolored edge guarantees that either the red rule or the blue rule can be applied. $\square$

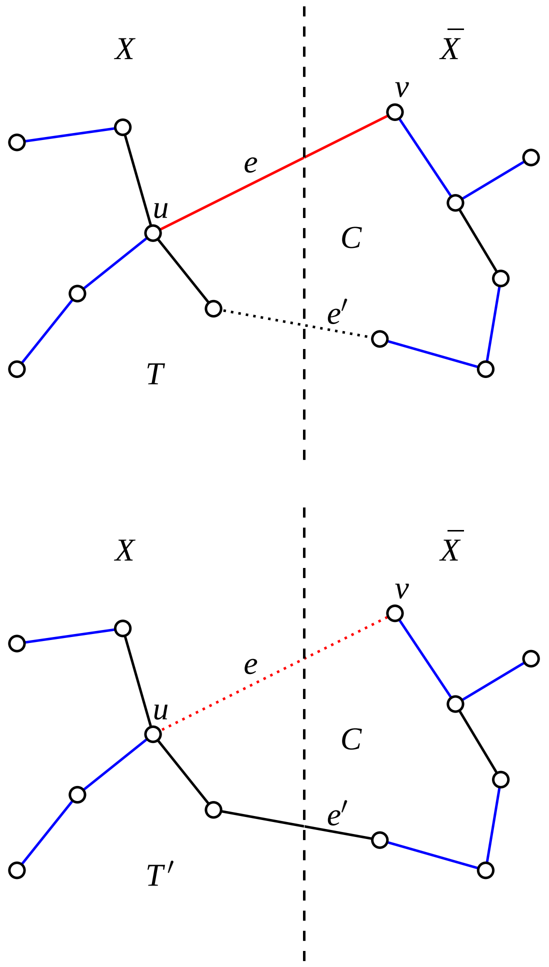


Figure 2: Illustration of the exchange argument used in the proof of Theorem 2.1 (red rule).

2.3 Kruskal's Algorithm

Kruskal's algorithm sorts the edges by non-decreasing cost and then considers the edges in this order. If the current edge $e_i = (u,v)$ has both its endpoints in the same blue tree, it is colored red; otherwise, it is colored blue. The algorithm is summarized in Algorithm 1.

It is easy to verify that in each case the pre-conditions of the respective rule are met: If the red rule applies, then the unique path P_{uv} in the blue tree containing both endpoints of e_i together with e_i forms a cycle C. The edges in $C \cap P_{uv}$ are blue and e_i is uncolored. We can thus apply the red rule to e_i. Otherwise, if the blue rule applies, then e_i connects two blue trees, say T_u and T_v, in the current blue forest. Consider the cut $(X, \bar{X})$ induced by the node set of T_u, i.e., $X = V(T_u)$. No blue edge crosses this cut. Moreover, e_i is an uncolored edge that crosses this cut. Also observe that every other uncolored edge $e \in \delta(X)$ has cost

```
Input: undirected graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}$
Output: minimum spanning tree $T$

1 Initialize: all edges are uncolored
        (Remark: we implicitly maintain a forest of blue trees below)
2 Let $\langle e_1, \dots, e_m \rangle$ be the list of edges of $G$, sorted by non-decreasing cost
3 for $i \leftarrow 1$ to $m$ do
4     if $e_i$ has both endpoints in the same blue tree then color $e_i$ red else color $e_i$
      blue
5 end
6 Output the resulting tree $T$ of blue edges
```

Algorithm 1: Kruskal's *MST* algorithm.

$c(e) \geq c(e_i)$ because we color the edges by non-decreasing cost. We can therefore apply the blue rule to e_i. An immediate consequence of Theorem 2.1 is that Kruskal's algorithm computes a minimum spanning tree.

We next analyze the time complexity of the algorithm: The algorithm needs to sort the edges of G by non-decreasing cost. There are different algorithms to do this with different running times. The most efficient algorithms sort a list of k elements in $O(k \log k)$ time. There is also a lower bound that shows that one cannot do better than that. That is, in our context we spend $\Theta(m \log m)$ time to sort the edges by non-decreasing cost.

We also need to maintain a data structure in order to determine whether an edge e_i has both its endpoints in the same blue tree or not. A trivial implementation stores for each node a unique identifier of the tree it is contained in. Checking whether the endpoints u and v of edge $e_i = (u, v)$ are part of the same blue tree can then be done in $O(1)$ time. Merging two blue trees needs time $O(n)$ in the worst case. Thus, the trivial implementation takes $O(m + n^2)$ time in total (excluding the time for sorting).

One can do much better by using a so-called *union-find* data structure. This data structure keeps track of the partition of the nodes into blue trees and allows only two types of operations: *union* and *find*. The *find* operation identifies the node set of the partition to which a given node belongs. It can be used to check whether the endpoints u and v of edge $e_i = (u, v)$ belong to the same tree or not. The *union* operation unites two node sets of the current partition into one. This operation is needed to update the partition whenever we color $e_i = (u, v)$ blue and have to join the respective blue trees T_u and T_v. Sophisticated union-find data structures support a series of n *union* and m *find* operations on a universe of n elements in time $O(n + m\alpha(n, \frac{m}{n}))$, where $\alpha(n, d)$ is the *inverse Ackerman function* (see [8, Chapter 2] and the references therein). $\alpha(n, d)$ is increasing in n but grows extremely slowly for every fixed d, e.g., $\alpha(2^{65536}, 0) = 4$; for most practical situations, it can be regarded as a constant.

The overall time complexity of Kruskal's algorithm is thus $O(m \log m + n + m\alpha(n, \frac{m}{n})) = O(m \log m) = O(m \log n)$ (think about it!).

Corollary 2.1. *Kruskal's algorithm solves the MST problem in time $O(m \log n)$.*

2.4 Prim's Algorithm

Prim's algorithm grows a single blue tree, starting at an arbitrary node $s \in V$. In every step, it chooses among all edges that are incident to the current blue tree T containing s an uncolored edge e_i of minimum cost and colors it blue. The algorithm stops if T contains all nodes. We implicitly assume that all edges that are not part of the final tree are colored red in a post-processing step. The algorithm is summarized in Algorithm 2.

```
Input: undirected graph $G = (V, E)$ with edge costs $c : E \to \mathbb{R}$
Output: minimum spanning tree $T$
1 Initialize: all edges are uncolored
      (Remark: we implicitly maintain a forest of blue trees below)
2 Choose an arbitrary node $s$
3 for $i \leftarrow 1$ to $n-1$ do
4     Let $T$ be the current blue tree containing $s$
5     Select a minimum cost edge $e_i \in \delta(V(T))$ incident to $T$ and color it blue
6 end
7 Implicitly: color all remaining edges red
8 Output the resulting tree $T$ of blue edges
```

Algorithm 2: Prim's *MST* algorithm.

Note that the pre-conditions are met whenever the algorithm applies one of the two coloring rules: If the blue rule applies, then the node set $V(T)$ of the current blue tree T containing s induces a cut $(X, \bar{X})$ with $X = V(T)$. No blue edge crosses $(X, \bar{X})$ by construction. Moreover, e_i is among all uncolored edges crossing the cut one of minimum cost and can thus be colored blue. If the red rule applies to edge $e = (u, v)$, both endpoints u and v are contained in the final tree T. The path P_{uv} in T together with e induce a cycle C. All edges in $C \cap P_{uv}$ are blue and we can thus color e red.

The time complexity of the algorithm depends on how efficiently we are able to identify a minimum cost edge e_i that is incident to T. To this aim, good implementations use a *priority queue* data structure. The idea is to keep track of the minimum cost connections between nodes that are outside of T to nodes in T. Suppose we maintain two data entries for every node $v \notin V(T)$: $\pi(v) = (u, v)$ refers to the edge that minimizes $c(u, v)$ among all $u \in V(T)$ and $d(v) = c(\pi(v))$ refers to the cost of this edge; we define $\pi(v) = \text{nil}$ and $d(v) = \infty$ if no such edge exists. Initially, we have for every node $v \in V \setminus \{s\}$:

$$\pi(v) = \begin{cases} (s,v) & \text{if } (s,v) \in E \\ \text{nil} & \text{otherwise.} \end{cases} \qquad \text{and} \qquad d(v) = \begin{cases} c(s,v) & \text{if } (s,v) \in E \\ \infty & \text{otherwise.} \end{cases}$$

The algorithm now repeatedly chooses a node $v \notin V(T)$ with $d(v)$ minimum, adds it to the tree and colors its connecting edge $\pi(v)$ blue. Because v is part of the new tree, we need to update the above data. This can be accomplished by iterating over all edges $(v, w) \in E$ incident to v and verifying for every adjacent node w with $w \notin V(T)$ whether the connection cost from w to T via edge (v, w) is less than the one stored in $d(w)$ (via $\pi(w)$). If so, we update the respective data entries accordingly. Note that if the value of

$d(w)$ changes, then it can only decrease.

There are several priority queue data structures that support all operations needed above: *insert*, *find-min*, *delete-min* and *decrease-priority*. In particular, using *Fibonacci heaps*, *m decrease-priority* and *n insert*/*find-min*/*delete-min* operations can be performed in time $O(m+n\log n)$.

Corollary 2.2. *Prim's algorithm solves the MST problem in time $O(m+n\log n)$.*

References

The presentation of the material in this section is based on [8, Chapter 6].

3. Matroids

3.1 Introduction

In the previous section, we have seen that the greedy algorithm can be used to solve the MST problem. An immediate question that comes to ones mind is which other problems can be solved by such an algorithm. In this section, we will see that the greedy algorithm applies to a much broader class of optimization problems.

We first define the notion of an *independent set system*.

Definition 3.1. Let S be a finite set and let $\mathcal{I}$ be a collection of subsets of S. $(S,\mathcal{I})$ is an *independent set system* if

(M1) $\emptyset \in \mathcal{I}$;
(M2) if $I \in \mathcal{I}$ and $J \subseteq I$, then $J \in \mathcal{I}$.

Each set $I \in \mathcal{I}$ is called an *independent set*; every other subset $I \subseteq S$ with $I \notin \mathcal{I}$ is called a *dependent set*. Further, suppose we are given a weight function $w : S \to \mathbb{R}$ on the elements in S.

Maximum Weight Independent Set Problem (MWIS):

Given:	An independent set system $(S,\mathcal{I})$ and a weight function $w : S \to \mathbb{R}$.
Goal:	Find an independent set $I \in \mathcal{I}$ of maximum weight $w(I) = \sum_{x \in I} w(x)$.

If $w(x) < 0$ for some $x \in S$, then x will not be included in any optimum solution because $\mathcal{I}$ is closed under taking subsets. We can thus safely exclude such elements from the ground set S. Subsequently, we assume without loss of generality that all weights are nonnegative.

As an example, consider the following independent set system: Suppose we are given an undirected graph $G = (V,E)$ with weight function $w : E \to \mathbb{R}^+$. Define $S = E$ and $\mathcal{I} = \{F \subseteq E \mid F \text{ induces a forest in } G\}$. Note that $\emptyset \in \mathcal{I}$ and $\mathcal{I}$ is closed under taking subsets because each subset J of a forest $I \in \mathcal{I}$ is a forest. Now, the problem of finding an independent set $I \in \mathcal{I}$ that maximizes $w(I)$ is equivalent to finding a spanning tree of G of maximum weight. (Note that the latter can also be done by one of the MST algorithms that we have considered in the previous section.)

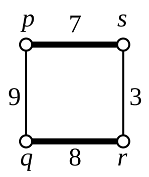
The greedy algorithm given in Algorithm 3 is a natural generalization of Kruskal's algorithm to independent set systems. It starts with the empty set $I = \emptyset$ and then iteratively extends I by always adding an element $x \in S \setminus I$ of maximum weight, ensuring that $I + x$ remains an independent set.

Unfortunately, the greedy algorithm does not work for general independent set systems as the following example shows:

```
Input: independent set system $(S,\mathcal{I})$ with weight function $w : S \to \mathbb{R}$
Output: independent set $I \in \mathcal{I}$ of maximum weight
1 Initialize: $I = \emptyset$.
2 while there is some $x \in S \setminus I$ with $I + x \in \mathcal{I}$ do
3     Choose such an $x$ with $w(x)$ maximum
4     $I \leftarrow I + x$
5 end
6 return $I$
```

Algorithm 3: Greedy algorithm for matroids.

Example 3.1.



Suppose that we are given an undirected graph $G = (V,E)$ with weight function $w : E \to \mathbb{R}$. Let $S = E$ and define $\mathcal{I} = \{M \subseteq E \mid M \text{ is a matching of } G\}$. (Recall that a subset $M \subseteq E$ of the edges of G is called a *matching* if no two edges of M share a common endpoint.) It is not hard to see that $\emptyset \in \mathcal{I}$ and $\mathcal{I}$ is closed under taking subsets. Thus Conditions (M1) and (M2) are satisfied and $(S,\mathcal{I})$ is an independent set system. Note that finding an independent set $I \in \mathcal{I}$ of maximum weight $w(I)$ is equivalent to finding a maximum weight matching in G. Suppose we run the above greedy algorithm on the independent set system induced by the matching instance depicted on the right. The algorithm returns the matching $\{(p,q),(r,s)\}$ of weight 12, which is not a maximum weight matching (indicated in bold).

3.2 Matroids

Even though the greedy algorithm described in Algorithm 3 does not work for general independent set systems, it does work for independent set systems that are *matroids*.

Definition 3.2 (Matroid)**.** An independent set system $M = (S,\mathcal{I})$ is a *matroid* if

(M3) if $I,J \in \mathcal{I}$ and $|I| < |J|$, then $I + x \in \mathcal{I}$ for some $x \in J \setminus I$.

Note that Condition (M3) essentially states that if I and J are two independent sets with $|I| < |J|$, then there must exist an element $x \in J \setminus I$ that can be added to I such that the resulting set $I + x$ is still an independent set.

Given a subset $U \subseteq S$, a subset $B \subseteq U$ is called a *basis* of U if B is an inclusionwise maximal independent subset of U, i.e., $B \in \mathcal{I}$ and there is no $I \in \mathcal{I}$ with $B \subset I \subseteq U$. It is not hard to show that Condition (M3) is equivalent to

(M4) for every subset $U \subseteq S$, any two bases of U have the same size.

The common size of the bases of $U \subseteq S$ is called the *rank* of U and denoted by $r(U)$. An independent set is simply called a *basis* if it is a basis of S. The common size of the bases of S is called the *rank* of the matroid M. Note that if all weights are nonnegative, the MWIS problem is equivalent to finding a maximum weight basis of M.

We give some examples of matroids.

Example 3.2 (Uniform matroid)**.** One of the simplest examples of a matroid is the so-called *uniform matroid*. Suppose we are given some set S and an integer k. Define the independent sets $\mathcal{I}$ as the set of all subsets of S of size at most k, i.e., $\mathcal{I} = \{I \subseteq S \mid |I| \leq k\}$. It is easy to verify that $M = (S, \mathcal{I})$ is a matroid. M is also called the *k-uniform matroid*.

Example 3.3 (Partition matroid)**.** Another simple example of a matroid is the *partition matroid*. Suppose S is partitioned into m sets $S_1, \dots, S_m$ and we are given m integers $k_1, \dots, k_m$. Define $\mathcal{I} = \{I \subseteq S \mid |I \cap S_i| \leq k_i \text{ for all } 1 \leq i \leq m\}$. Conditions (M1) and (M2) are trivially satisfied. To see that Condition (M3) is satisfied as well, note that if $I, J \in \mathcal{I}$ and $|I| < |J|$, then there is some i $(1 \leq i \leq m)$ such that $|J \cap S_i| > |I \cap S_i|$ and thus adding any element $x \in S_i \cap (J \setminus I)$ to I maintains independence. Thus, $M = (S, \mathcal{I})$ is a matroid.

Example 3.4 (Graphic matroid)**.** Suppose we are given an undirected graph $G = (V, E)$. Let $S = E$ and define $\mathcal{I} = \{F \subseteq E \mid F \text{ induces a forest in } G\}$. We already argued above that Conditions (M1) and (M2) are satisfied. We next show that Conditions (M4) is satisfied too. Let $U \subseteq E$. Consider the subgraph (V, U) of G induced by U and suppose that it consists of k components. By definition, each basis B of U is an inclusionwise maximal forest contained in U. Thus, B consists of k spanning trees, one for each component of the subgraph (V, U). We conclude that B contains $|V| - k$ elements. Because this holds for every basis of U, Condition (M4) is satisfied. We remark that any matroid $M = (S, \mathcal{I})$ obtained in this way is also called a *graphic matroid* (or *cycle matroid*).

Example 3.5 (Matching matroid)**.** The independent set system of Example 3.1 is *not* a matroid. However, there is another way of defining a matroid based on matchings. Let $G = (V, E)$ be an undirected graph. Given a matching $M \subseteq E$ of G, let $V(M)$ refer to the set of nodes that are incident to the edges of M. A node set $I \subseteq V$ is *covered* by M if $I \subseteq V(M)$. Define $S = V$ and $\mathcal{I} = \{I \subseteq V \mid I \text{ is covered by some matching } M\}$. Condition (M1) holds trivially. Condition (M2) is satisfied because if a node set $I \in \mathcal{I}$ is covered by a matching M, then each subset $J \subseteq I$ is also covered by M and thus $J \in \mathcal{I}$. It can also be shown that Condition (M3) is satisfied and thus $M = (S, \mathcal{I})$ is a matroid. M is also called a *matching matroid*.

3.3 Greedy Algorithm for Matroids

The next theorem shows that the greedy algorithm given in Algorithm 3 always computes a maximum weight independent set if the underlying independent set system is a matroid. The theorem actually shows something much stronger: Matroids are precisely the independent set systems for which the greedy algorithm computes an optimal solution.

Theorem 3.1. *Let $(S, \mathcal{I})$ be an independent set system. Further, let $w : S \to \mathbb{R}_+$ be a nonnegative weight function on S. The greedy algorithm (Algorithm 3) computes an independent set of maximum weight if and only if $M = (S, \mathcal{I})$ is a matroid.*

Proof. We first show that the greedy algorithm computes a maximum weight independent set if M is a matroid. Let X be the independent set returned by the greedy algorithm and let Y be a maximum weight independent set. Note that both X and Y are bases of M. Order the elements in $X = \{x_1, \dots, x_m\}$ such that x_i $(1 \leq i \leq m)$ is the i-th element chosen by the algorithm. Clearly, $w(x_1) \geq \cdots \geq w(x_m)$. Also order $Y = \{y_1, \dots, y_m\}$ such that $w(y_1) \geq \cdots \geq w(y_m)$. We will show that $w(x_i) \geq w(y_i)$ for every i. Let $k+1$ be the smallest integer such that $w(x_{k+1}) < w(y_{k+1})$. (The claim follows if no such choice exists.) Define $I = \{x_1, \dots, x_k\}$ and $J = \{y_1, \dots, y_{k+1}\}$. Because $I, J \in \mathcal{I}$ and $|I| < |J|$, Condition (M3) implies that there is some $y_i \in J \setminus I$ such that $I + y_i \in \mathcal{I}$. Note that $w(y_i) \geq w(y_{k+1}) > w(x_{k+1})$. That is, in iteration $k+1$, the greedy algorithm would prefer to add y_i instead of x_{k+1} to extend I, which is a contradiction. We conclude that $w(X) \geq w(Y)$ and thus X is a maximum weight independent set.

Next assume that the greedy algorithm always computes an independent set of maximum weight for every independent set system $(S, \mathcal{I})$ and weight function $w : S \to \mathbb{R}_+$. We show that $M = (S, \mathcal{I})$ is a matroid. Conditions (M1) and (M2) is satisfied by assumption. It remains to show that Condition (M3) holds. Let $I, J \in \mathcal{I}$ with $|I| < |J|$ and assume, for the sake of a contradiction, that $I + x \notin \mathcal{I}$ for every $x \in J \setminus I$. Let $k = |I|$ and consider the following weight function on S:

$$w(x) = \begin{cases} k+2 & \text{if } x \in I \\ k+1 & \text{if } x \in J \setminus I \\ 0 & \text{otherwise.} \end{cases}$$

Now, in the first k iterations, the greedy algorithms picks the elements in I. By assumption, the algorithm cannot add any other element from $J \setminus I$ and thus outputs a solution of weight $k(k+2)$. However, the independent set J has weight at least $|J|(k+1) \geq (k+1)(k+1) > k(k+2)$. That is, the greedy algorithm does not compute a maximum weight independent set, which is a contradiction. $\square$

References

The presentation of the material in this section is based on [2, Chapter 8] and [7, Chapters 39 & 40].

4. Shortest Paths

4.1 Introduction

We next consider shortest path problems. These problems are usually defined for *directed networks*. Let $G = (V, E)$ be a directed graph with cost function $c : E \to \mathbb{R}$. Consider a (directed) path $P = \langle v_1, \dots, v_k \rangle$ from $s = v_1$ to $t = v_k$. The *length* of path P is defined as $c(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1})$. We can then ask for the computation of an s,t-path whose length is shortest among all directed paths from s to t. There are different variants of shortest path problems:

1. *Single source single target shortest path problem*: Given two nodes s and t, determine a shortest path from s to t.
2. *Single source shortest path problem*: Given a node s, determine all shortest paths from s to every other node in V.
3. *All-pairs shortest path problem*: For every pair $(s,t) \in V \times V$ of nodes, compute a shortest path from s to t.

The first problem is a special case of the second one. However, every known algorithm for the first problem implicitly also solves the second one (at least partially). We therefore focus here on the *single source shortest path problem* and the *all-pairs shortest path problem*.

4.2 Single Source Shortest Path Problem

We consider the following problem:

Single Source Shortest Path Problem (SSSP):

Given:	A directed graph $G = (V, E)$ with cost function $c : E \to \mathbb{R}$ and a source node $s \in V$.
Goal:	Compute a shortest path from s to every other node $v \in V$.

Note that a shortest path from s to a node v might not necessarily exist because of the following two reasons: First, v might not be reachable from s because there is no directed path from s to v in G. Second, there might be arbitrarily short paths from s to v because of the existence of an s,v-path that contains a cycle of negative length (which can be traversed arbitrarily often). We call a cycle of negative total length also a *negative cycle*. The following lemma shows that these are the only two cases in which no shortest path exists.

Lemma 4.1. *Let v be a node that is reachable from s. Further assume that there is no path from s to v that contains a negative cycle. Then there exists a shortest path from s to v which is a simple path.*

Proof. Let P be a path from s to v. We can repeatedly remove cycles from P until we obtain a simple path P'. By assumption, all these cycles have non-negative lengths and thus $c(P') \leq c(P)$. It therefore suffices to show that there is a shortest path among all simple s,v-paths. But this is obvious because there are only finitely many simple paths from s to v in G. □

4.2.1 Basic properties of shortest paths

We define a *distance function* $\delta : V \to \mathbb{R}$ as follows: For every $v \in V$,

$$\delta(v) = \inf\{c(P) \mid P \text{ is a path from } s \text{ to } v\}.$$

With the above lemma, we have

$$\begin{aligned}
\delta(v) &= \infty && \text{if there is no path from } s \text{ to } v \\
\delta(v) &= -\infty && \text{if there is a path from } s \text{ to } v \text{ that contains a negative cycle} \\
\delta(v) &\in \mathbb{R} && \text{if there is a shortest (simple) path from } s \text{ to } v.
\end{aligned}$$

The next lemma establishes that δ satisfies the triangle inequality.

Lemma 4.2. *For every edge $e = (u,v) \in E$, we have $\delta(v) \leq \delta(u) + c(u,v)$.*

Proof. Clearly, the relation holds if $\delta(u) = \infty$. Suppose $\delta(u) = -\infty$. Then there is a path P from s to u that contains a negative cycle. By appending edge e to P, we obtain a path from s to v that contains a negative cycle and thus $\delta(v) = -\infty$. The relation again holds. Finally, assume $\delta(u) \in \mathbb{R}$. Then there is a path P from s to u of length $\delta(u)$. By appending edge e to P, we obtain a path from s to v of length $\delta(u) + c(u,v)$. A shortest path from s to v can only have shorter length and thus $\delta(v) \leq \delta(u) + c(u,v)$. □

The following lemma shows that subpaths of shortest paths are shortest paths.

Lemma 4.3. *Let $P = \langle v_1, \dots, v_k \rangle$ be a shortest path from v_1 to v_k. Then every subpath $P' = \langle v_i, \dots, v_j \rangle$ of P with $1 \leq i \leq j \leq k$ is a shortest path from v_i to v_j.*

Proof. Suppose there is a path $P'' = \langle v_i, u_1, \dots, u_l, v_j \rangle$ from v_i to v_j that is shorter than P'. Then the path $\langle v_1, \dots, v_i, u_1, \dots, u_l, v_j, \dots, v_k \rangle$ is a v_1, v_k-path that is shorter than P, which is a contradiction. □

Consider a shortest path $P = \langle s = v_1, \dots, v_k = v \rangle$ from s to v. The above lemma enables us to show that every edge $e = (v_i, v_{i+1})$ of P must be *tight* with respect to the distance function δ, i.e., $\delta(v) = \delta(u) + c(u,v)$.

Lemma 4.4. *Let $P = \langle s, \dots, u, v \rangle$ be a shortest s,v-path. Then $\delta(v) = \delta(u) + c(u,v)$.*

Proof. By Lemma 4.3, the subpath $P' = \langle s, \dots, u \rangle$ of P is a shortest s,u-path and thus $\delta(u) = c(P')$. Because P is a shortest s,v-path, we have $\delta(v) = c(P) = c(P') + c(u,v) = \delta(u) + c(u,v)$. □

Suppose now that we can compute $\delta(v)$ for every node $v \in V$. Using the above lemmas, it is not difficult to show that we can then also efficiently determine the shortest paths from s to every node $v \in V$ with $\delta(v) \in \mathbb{R}$: Let $V' = \{v \in V \mid \delta(v) \in \mathbb{R}\}$ be the set of nodes for which there exists a shortest path from s. Note that $\delta(s) = 0$ and thus $s \in V'$. Further, let E' be the set of edges that are tight with respect to δ, i.e.,

$$E' = \{(u,v) \in E \mid \delta(v) = \delta(u) + c(u,v)\}.$$

Let $G' = (V', E')$ be the subgraph of G induced by V' and E'. Observe that we can construct G' in time $O(n+m)$. By Lemma 4.4, every edge of a shortest path from s to some node $v \in V'$ is tight. Thus every node $v \in V'$ is reachable from s in G'. Consider a path $P = \langle s = v_1, \dots, v_k = v\rangle$ from s to v in G'. Then

$$c(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}) = \sum_{i=1}^{k-1} (\delta(v_{i+1}) - \delta(v_i)) = \delta(v) - \delta(s) = \delta(v).$$

That is, P is a shortest path from s to v in G. G' therefore represents all shortest paths from s to nodes $v \in V'$. We can now extract a spanning tree T from G' that is rooted at s, e.g., by performing a depth-first search from s. Such a tree can be computed in time $O(n+m)$. Observe that T contains for every node $v \in V'$ a unique s,v-path which is a shortest path in G. T is therefore also called a *shortest-path tree*. Note that T is a very compact way to store for every node $v \in V'$ a shortest path from s to v. This tree needs $O(n)$ space only, while listing all these paths explicitly may need $O(n^2)$ space.

In light of the above observations, we will subsequently concentrate on the problem of computing the distance function δ efficiently. To this aim, we introduce a function $d : V \to \mathbb{R}$ of *tentative* distances. The algorithm will use d to compute a more and more refined approximation of δ until eventually $d(v) = \delta(v)$ for every $v \in V$. We initialize $d(s) = 0$ and $d(v) = \infty$ for every $v \in V \setminus \{s\}$. The only operation that is used to modify d is to *relax* an edge $e = (u,v) \in E$:

```
RELAX$(u,v)$:
    if $d(v) > d(u) + c(u,v)$ then $d(v) = d(u) + c(u,v)$
```

It is obvious that the d-values can only decrease by edge relaxations.

We show that if we only relax edges then the tentative distances will never be less than the actual distances.

Lemma 4.5. *For every $v \in V$, $d(v) \geq \delta(v)$.*

Proof. The proof is by induction on the number of relaxations. The claim holds after the initialization because $d(v) = \infty \geq \delta(v)$ and $d(s) = 0 = \delta(s)$. For the induction step, suppose that the claim holds true before the relaxation of an edge $e = (u,v)$. We show that it remains valid after edge e has been relaxed. By relaxing (u,v), only $d(v)$ can be modified. If $d(v)$ is modified, then after the relaxation we have $d(v) = d(u) + c(u,v) \geq \delta(u) + c(u,v) \geq \delta(v)$, where the first inequality follows from the induction hypothesis and the latter inequality holds because of the triangle inequality (Lemma 4.2). □

That is, $d(v)$ decreases throughout the execution but will never be lower than the actual distance $\delta(v)$. In particular, $d(v) = \delta(v) = \infty$ for all nodes $v \in V$ that are not reachable from s. Our goal will be to use only few edge relaxations to ensure that $d(v) = \delta(v)$ for every $v \in V$ with $\delta(v) \in \mathbb{R}$.

Lemma 4.6. *Let $P = \langle s, \dots, u, v\rangle$ be a shortest s,v-path. If $d(u) = \delta(u)$ before the relaxation of edge $e = (u, v)$, then $d(v) = \delta(v)$ after the relaxation of edge e.*

Proof. Note that after the relaxation of edge e, we have $d(v) = d(u) + c(u, v) = \delta(u) + c(u, v) = \delta(v)$, where the last equality follows from Lemma 4.4. $\square$

4.2.2 Arbitrary cost functions

The above lemma makes it clear what our goal should be. Namely, ideally we should relax the edges of G in the order in which they appear on shortest paths. The dilemma, of course, is that we do not know these shortest paths. The following algorithm, also known as the *Bellman-Ford* algorithm, circumvents this problem by simply relaxing every edge exactly $n-1$ times, thereby also relaxing all edges along shortest path in the right order. An illustration is given in Figure 4.

```
Input: directed graph $G = (V, E)$, cost function $c : E \to \mathbb{R}$, source node $s \in V$
Output: shortest path distances $d : V \to \mathbb{R}$
1 Initialize: $d(s) = 0$ and $d(v) = \infty$ for every $v \in V \setminus \{s\}$
2 for $i \leftarrow 1$ to $n-1$ do
3     foreach $(u, v) \in E$ do RELAX$(u, v)$
4 end
5 return $d$
```

Algorithm 4: Bellman-Ford algorithm for the SSSP problem.

Lemma 4.7. *After the Bellman-Ford algorithm terminates, $d(v) = \delta(v)$ for all $v \in V$ with $\delta(v) > -\infty$.*

Proof. As argued above, after the initialization we have $d(v) = \delta(v)$ for all $v \in V$ with $\delta(v) = \infty$. Consider a node $v \in V$ with $\delta(v) \in \mathbb{R}$. Let $P = \langle s = v_1, \dots, v_k = v\rangle$ be a shortest s,v-path. Define a *phase* of the algorithm as the execution of the inner loop. That is, the algorithm consists of $n-1$ phases and in each phase every edge of G is relaxed exactly once. Note that $d(s) = \delta(s)$ after the initialization. Using induction on i and Lemma 4.6, we can show that $d(v_{i+1}) = \delta(v_{i+1})$ at the end of phase i. Thus, after at most $n-1$ phases $d(v) = \delta(v)$ for every $v \in V$ with $\delta(v) \in \mathbb{R}$. $\square$

Note that the algorithm does not identify nodes $v \in V$ with $\delta(v) = -\infty$. However, this can be accomplished in a post-processing step (see exercises). The time complexity of the algorithm is obviously $O(nm)$. Clearly, we might improve on this by stopping the algorithm as soon as all tentative distances remain unchanged in a phase. However, this does not improve on the worst case running time.

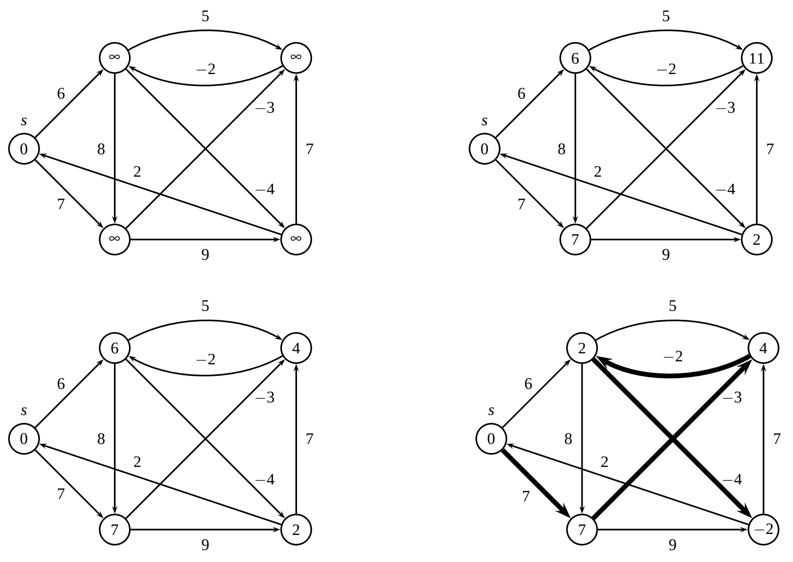


Figure 3: Illustration of the Bellman-Ford algorithm. The order in which the edges are relaxed in this example is as follows: We start with the upper right node and proceed in a clockwise order. For each node, edges are relaxed in clockwise order. Tight edges are indicated in bold. Only the first three phases are depicted (no change in the final phase).

Theorem 4.1. *The Bellman-Ford algorithm solves the SSSP problem without negative cycles in time* $\Theta(nm)$.

4.2.3 Nonnegative cost functions

The running time of $O(nm)$ of the Bellman-Ford algorithm is rather large. We can significantly improve upon this in certain special cases. The easiest such special case is if the graph G is acyclic.

Another example is if the edge costs are nonnegative. Subsequently, we assume that the cost function $c : E \to \mathbb{R}^+$ is nonnegative.

The best algorithm for the SSSP with nonnegative cost functions is known as *Dijkstra's algorithm*. As before, the algorithm starts with $d(s) = 0$ and $d(v) = \infty$ for every $v \in V \setminus \{s\}$. It also maintains a set V^* of nodes whose distances are tentative. Initially, $V^* = V$. The algorithm repeatedly chooses a node $u \in V^*$ with $d(u)$ minimum, removes it from V^* and relaxes all outgoing edges (u,v). The algorithm stops when $V^* = \emptyset$. A formal description is given in Algorithm 5.

Note that the algorithm relaxes every edge exactly once. Intuitively, the algorithm can be viewed as maintaining a "cloud" of nodes ($V \setminus V^*$) whose distance labels are exact. In each iteration, the algorithm chooses a node $u \in V^*$ that is closest to the cloud, declares its distance label as exact and relaxes all its outgoing edges. As a consequence, other nodes outside of the cloud might get closer to the cloud. An illustration of the execution of the algorithm is given in Figure 4.

```
Input: directed graph $G = (V,E)$, nonnegative cost function $c : E \to \mathbb{R}$, source node
       $s \in V$
Output: shortest path distances $d : V \to \mathbb{R}$
1 Initialize: $d(s) = 0$ and $d(v) = \infty$ for every $v \in V \setminus \{s\}$
2 $V^* = V$
3 while $V^* \neq \emptyset$ do
4     Choose a node $u \in V^*$ with $d(u)$ minimum.
5     Remove $u$ from $V^*$.
6     foreach $(u,v) \in E$ do RELAX$(u,v)$
7 end
8 return $d$
```

Algorithm 5: Dijkstra's algorithm for the SSSP problem.

The correctness of the algorithm follows from the following lemma.

Lemma 4.8. *Whenever a node u is removed from V^*, we have $d(u) = \delta(u)$.*

Proof. The proof is by contradiction. Consider the first iteration in which a node u is removed from V^* while $d(u) > \delta(u)$. Let $A \subseteq V$ be the set of nodes v with $d(v) = \delta(v)$. Note that u is reachable from s because $\delta(u) < \infty$. Let P be a shortest s,u-path. If we traverse P from s to u, then there must be an edge $(x,y) \in P$ with $x \in A$ and $y \notin A$ because $s \in A$ and $u \notin A$. Let (x,y) be the first such edge on P. We have $d(x) = \delta(x) \leq \delta(u) < d(u)$, where the first inequality holds because all edge costs are nonnegative. Consequently, x was removed from V^* before u. By the choice of u, $d(x) = \delta(x)$ when x was removed from V^*. But then, by Lemma 4.6, we must have $d(y) = \delta(y)$ after the relaxation of edge (x,y), which is a contradiction to the assumption that $y \notin A$. □

The running time of Dijkstra's algorithm crucially relies on the underlying data structure. An efficient way to keep track of the tentative distance labels and the set V^* is to use *priority queues*. We need at most n *insert* (initialization), n *delete-min* (removing nodes with minimum d-value) and m *decrease-priority* operations (updating distance labels after edge relaxations). *Fibonacci heaps* support these operations in time $O(m + n\log n)$.

Theorem 4.2. *Dijkstra's algorithm solves the SSSP problem with nonnegative edge costs in time $O(m + n\log n)$.*

4.3 All-pairs Shortest-path Problem

We next consider the following problem:

All-pairs Shortest Path Problem (APSP):

Given:	A directed graph $G = (V,E)$ with cost function $c : E \to \mathbb{R}$.
Goal:	Determine a shortest s,t-path for every pair $(s,t) \in V \times V$.

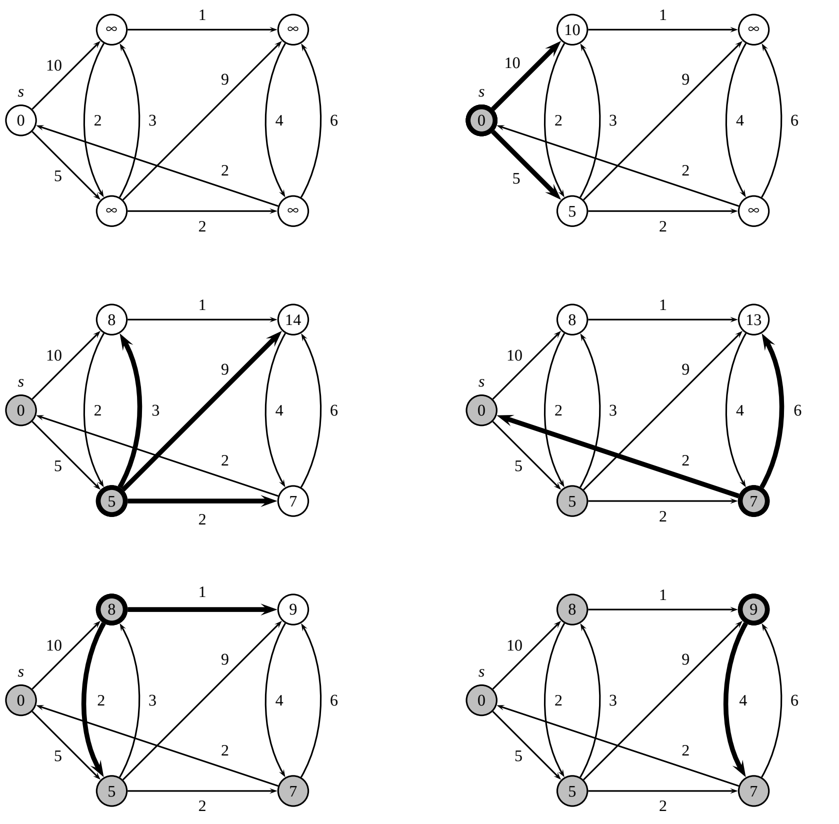


Figure 4: Illustration of Dijsktra's algorithm. The nodes in $V \setminus V^*$ are depicted in gray. The current node that is removed from V^* is drawn in bold. The respective edge relaxations are indicated in bold.

```
Input: directed graph $G = (V,E)$, nonnegative cost function $c : E \to \mathbb{R}$
Output: shortest path distances $d : V \times V \to \mathbb{R}$
1 Initialize: foreach $(u,v) \in V \times V$ do $d(u,v) = \begin{cases} 0 & \text{if } u = v \\ c(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise.} \end{cases}$
2 for $k \leftarrow 1$ to $n$ do
3     foreach $(u,v) \in V \times V$ do
4         if $d(u,v) > d(u,k) + d(k,v)$ then $d(u,v) = d(u,k) + d(k,v)$
5     end
6 end
7 return $d$
```

Algorithm 6: Floyd-Warshall algorithm for the APSP problem.

We assume that G contains no negative cycle.

Define a distance function $\delta : V \times V \to \mathbb{R}$ as

$$\delta(u,v) = \inf\{c(P) \mid P \text{ is a path from } u \text{ to } v\}.$$

Note that δ is not necessarily symmetric. As for the SSSP problem, we can concentrate on the computation of the distance function δ because the actual shortest paths can be extracted from these distances.

Clearly, one way to solve the APSP problem is to simply solve n SSSP problems: For every node $s \in V$, solve the SSSP problem with source node s to compute all distances $\delta(s,\cdot)$. Using the Bellman-Ford algorithm, the worst-case running time of this algorithm is $O(n^2m)$, which for dense graphs is $\Theta(n^4)$. We will see that we can do better.

The idea is based on a general technique to derive exact algorithms known as *dynamic programming*. Basically, the idea is to decompose the problem into smaller sub-problems which can be solved individually and to use these solutions to construct a solution for the whole problem in a bottom-up manner.

Suppose the nodes in V are identified with the set $\{1,\dots,n\}$. In order to define the dynamic program, we need some more notion. Consider a simple u,v-path $P = \langle u = v_1,\dots,v_l = v\rangle$. We call the nodes $v_2,\dots,v_{l-1}$ the *interior nodes* of P; P has no interior nodes if $l \leq 2$. A u,v-path P whose interior nodes are all contained in $\{1,\dots,k\}$ is called a *(u,v,k)-path*. Define

$$\delta_k(u,v) = \inf\{c(P) \mid P \text{ is a } (u,v,k)\text{-path}\}$$

as the shortest path distance of a (u,v,k)-path. Clearly, with this definition we have $\delta(u,v) = \delta_n(u,v)$. Our task is therefore to compute $\delta_n(u,v)$ for every $u,v \in V$.

Our dynamic program is based on the following observation. Suppose we are able to compute $\delta_{k-1}(u,v)$ for all $u,v \in V$. Consider a shortest (u,v,k)-path $P = \langle u = v_1,\dots,v_l = v\rangle$. Note that P is simple because we assume that G contains no negative cycles. By definition, the interior nodes of P belong to the set $\{1,\ldots,k\}$. There are two cases that can occur: Either node k is an interior node of P or not.

First, assume that k is not an interior node of P. Then all interior nodes of P must belong to the set $\{1,\ldots,k-1\}$. That is, P is a shortest $(u,v,k-1)$-path and thus $\delta_k(u,v) = \delta_{k-1}(u,v)$.

Next, suppose k is an interior node of P, i.e., $P = \langle u = v_1,\ldots,k,\ldots,v_l = v\rangle$. We can then break P into two paths $P_1 = \langle u,\ldots,k\rangle$ and $P_2 = \langle k,\ldots,v\rangle$. Note that the interior nodes of P_1 and P_2 are contained in $\{1,\ldots,k-1\}$ because P is simple. Moreover, because subpaths of shortest paths are shortest paths, we conclude that P_1 is a shortest $(u,k,k-1)$-path and P_2 is a shortest $(k,v,k-1)$-path. Therefore, $\delta_k(u,v) = \delta_{k-1}(u,k) + \delta_{k-1}(k,v)$.

The above observations lead to the following recursive definition of $\delta_k(u,v)$:

$$\delta_0(u,v) = \begin{cases} 0 & \text{if } u = v \\ c(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise.} \end{cases}$$

and

$$\delta_k(u,v) = \min\{\delta_{k-1}(u,v),\ \delta_{k-1}(u,k) + \delta_{k-1}(k,v)\} \qquad \text{if } k \geq 1$$

The Floyd-Warshall algorithm simply computes $\delta_k(u,v)$ in a bottom-up manner. The algorithm is given in Algorithm 6.

Theorem 4.3. *The Floyd-Warshall algorithm solves the APSP problem without negative cycles in time* $\Theta(n^3)$.

References

The presentation of the material in this section is based on [3, Chapters 25 and 26].

5. Maximum Flows

5.1 Introduction

The *maximum flow problem* is a fundamental problem in combinatorial optimization with many applications in practice. We are given a network consisting of a directed graph $G = (V, E)$ and nonnegative capacities $c : E \to \mathbb{R}^+$ on the edges and a source node $s \in V$ and a target node $t \in V$.

Intuitively, think of G being a water network and suppose that we want to send as much water as possible (say per second) from a source node s (producer) to a target node t (consumer). An edge of G corresponds to a pipeline of the water network. Every pipeline comes with a capacity which specifies the maximum amount of water that can be sent through it (per second). Basically, the *maximum flow problem* asks how the water should be routed through the network such that the total amount of water that can be sent from s to t (per second) is maximized.

We assume without loss of generality that G is complete. If G is not complete, then we simply add every missing edge $(u, v) \in V \times V \setminus E$ to G and define $c(u, v) = 0$. We also assume that every node $u \in V$ lies on a path from s to t (other nodes will be irrelevant).

Definition 5.1. A *flow* (or *s,t-flow*) in G is a function $f : V \times V \to \mathbb{R}$ that satisfies the following three properties:

1. **Capacity constraint**: For all $u, v \in V$, $f(u, v) \le c(u, v)$.
2. **Skew symmetry**: For all $u, v \in V$, $f(u, v) = -f(v, u)$.
3. **Flow conservation**: For every $u \in V \setminus \{s, t\}$, we have

$$\sum_{v \in V} f(u, v) = 0.$$

The quantity $f(u, v)$ can be interpreted as the *net flow* from u to v (which can be positive, zero or negative). The capacity constraint ensures that the flow value of an edge does not exceed the capacity of the edge. Note that skew symmetry expresses that the net flow $f(u, v)$ that is sent from u to v is equal to the net flow $f(v, u) = -f(u, v)$ that is sent from v to u. Also the total net flow from u to itself is zero because $f(u, u) = -f(u, u) = 0$. The flow conservation constraints make sure that the total flow out of a node $u \in V \setminus \{s, t\}$ is zero. Because of skew symmetry, this is equivalent to stating that the total flow into u is zero.

Another way of interpreting the flow conservation constraints is that the total positive net flow entering a node $u \in V \setminus \{s, t\}$ is equal to the total positive net flow leaving u, i.e.,

$$\sum_{v \in V : f(v,u) > 0} f(v, u) = \sum_{v \in V : f(u,v) > 0} f(u, v).$$

The *value* $|f|$ of a flow f refers to the total net flow out of s (which by the flow conserva-

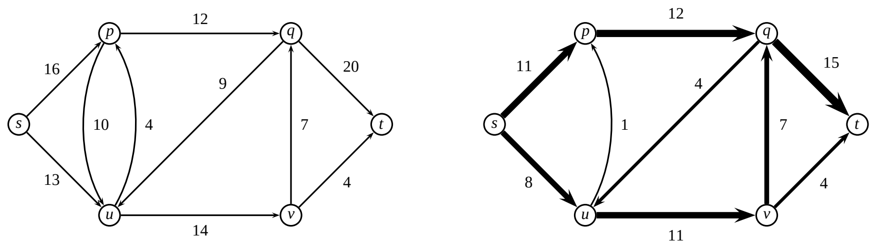


Figure 5: On the left: Input graph G with capacities $c : E \to \mathbb{R}^+$. Only the edges with positive capacity are shown. On the right: A flow f of G with flow value $|f| = 19$. Only the edges with positive net flow are shown.

tion constraints is the same as the total flow into t):

$$|f| = \sum_{v \in V} f(s,v).$$

An example of a network and a flow is given in Figure 5.

The maximum flow reads as follows:

Maximum Flow Problem:

Given:	A directed graph $G = (V,E)$ with capacities $c : E \to \mathbb{R}^+$, a source node $s \in V$ and a destination node $t \in V$.
Goal:	Compute an s,t-flow f of maximum value.

We introduce some more notation. Given two sets $X,Y \subseteq V$, define

$$f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y).$$

We state a few properties. (You should try to convince yourself that these properties hold true.)

Proposition 5.1. *Let f be a flow in G. Then the following holds true:*

1. *For every $X \subseteq V$, $f(X,X) = 0$.*
2. *For every $X,Y \subseteq V$, $f(X,Y) = -f(Y,X)$.*
3. *For every $X,Y,Z \subseteq V$ with $X \cap Y = \emptyset$,*

$$f(X \cup Y, Z) = f(X,Z) + f(Y,Z) \quad \textit{and} \quad f(Z, X \cup Y) = f(Z,X) + f(Z,Y).$$

5.2 Residual Graph and Augmenting Paths

Consider an s,t-flow f. Let the *residual capacity* of an edge $e = (u,v) \in E$ with respect to f be defined as

$$r_f(u,v) = c(u,v) - f(u,v).$$

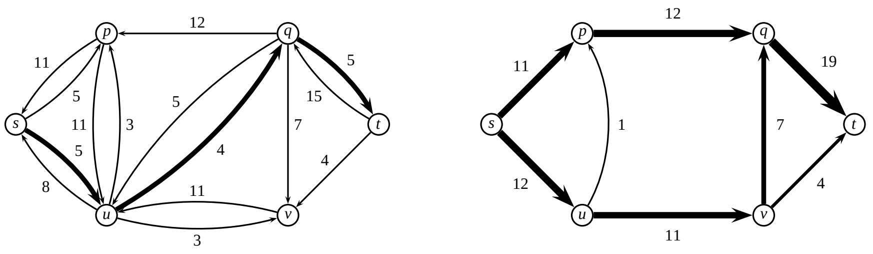


Figure 6: On the left: Residual graph G_f with respect to the flow f given in Figure 5. An augmenting path P with $r_f(P) = 4$ is indicated in bold. On the right: The flow f' obtained from f by augmenting $r_f(P)$ units of flow along P. Only the edges with positive flow are shown. The flow value of f' is $|f'| = 23$. (Note that f' is optimal because the cut $(X, \bar{X})$ of G with $\bar{X} = \{q, t\}$ has capacity $c(X, \bar{X}) = 23$.)

Intuitively, the residual capacity $r_f(u, v)$ is the amount of flow that can additionally be sent from u to v without exceeding the capacity $c(u, v)$. Call an edge $e \in E$ a *residual edge* if it has positive residual capacity and a *saturated* edge otherwise. The *residual graph* $G_f = (V, E_f)$ with respect to f is the subgraph of G whose edge set E_f consists of all residual edges, i.e.,

$$E_f = \{e \in E \mid r_f(e) > 0\}.$$

See Figure 6 (left) for an example.

Lemma 5.1. *Let f be a flow in G. Let g be a flow in the residual graph G_f respecting the residual capacities r_f. Then the combined flow $h = f + g$ is a flow in G with value $|h| = |f| + |g|$.*

Proof. We show that all properties of Definition 5.1 are satisfied.

First, h satisfies the skew symmetry property because for every $u, v \in V$

$$h(u, v) = f(u, v) + g(u, v) = -(f(v, u) + g(v, u)) = -h(v, u).$$

Second, observe that for every $u, v \in V$, $g(u, v) \le r_f(u, v)$ and thus

$$h(u, v) = f(u, v) + g(u, v) \le f(u, v) + r_f(u, v) = f(u, v) + (c(u, v) - f(u, v)) = c(u, v).$$

That is, the capacity constraints are satisfied.

Finally, we have for every $u \in V \setminus \{s, t\}$

$$\sum_{v \in V} h(u, v) = \sum_{v \in V} f(u, v) + \sum_{v \in V} g(u, v) = 0$$

and thus flow conservation is satisfied too.

Similarly, we can show that

$$|h| = \sum_{v\in V} h(s,v) = \sum_{v\in V} f(s,v) + \sum_{v\in V} g(s,v) = |f| + |g|.$$

□

An *augmenting path* is a simple s,t-path P in the residual graph G_f. Let P be an augmenting path in G_f. All edges of P are residual edges. Thus, there exists some $x > 0$ such that we can send x flow units additionally along P without exceeding the capacity of any edge. In fact, we can choose x to be as large as the *residual capacity* $r_f(P)$ of P which is defined as

$$r_f(P) = \min\{r_f(u,v) \mid e \in P\}.$$

Note that if we increase the flow of an edge $(u,v) \in P$ by $x = r_f(P)$, then we also have to decrease the flow value on (v,u) by x because of the skew symmetry property. We will also say that we *augment the flow f along path P*. See Figure 6 for an example.

Lemma 5.2. *Let f be a flow in G and let P be an augmenting path in G_f. Then $f' : V \times V \to \mathbb{R}$ with*

$$f'(u,v) = \begin{cases} f(u,v) + r_f(P) & \text{if } (u,v) \in P \\ f(u,v) - r_f(P) & \text{if } (v,u) \in P \\ f(u,v) & \text{otherwise} \end{cases}$$

is a flow in G of value $|f'| = |f| + r_f(P)$.

Proof. Observe that f' can be decomposed into the original flow f and a flow f_P that sends $r_f(P)$ units of flow along P and $-r_f(P)$ flow units along the *reversed* path of P, i.e., the path that we obtain from P if we reverse the direction of every edge $e \in P$. Clearly, f_P is a flow in G_f of value $r_f(P)$. By Lemma 5.1, the combined flow $f' = f + f_P$ is a flow in G of value $|f'| = |f| + r_f(P)$. □

5.3 Ford-Fulkerson Algorithm

The observations above already suggest a first algorithm for the max-flow problem: Initialize f to be the zero flow, i.e., $f(u,v) = 0$ for all $u,v \in V$. Let G_f be the residual graph with respect to f. If there exists an augmenting path P in the residual graph G_f, then augment f along P and repeat; otherwise terminate. This algorithm is also known as the *Ford-Fulkerson* algorithm and is summarized in Algorithm 7.

Note that it is not clear that the algorithm terminates nor that the computed flow is of maximum value. The correctness of the algorithm will follow from the *max-cut min-flow theorem* (Theorem 5.3) discussed in the next section.

The running time of the algorithm depends on the number of iterations that we need to perform. Every single iteration can be implemented to run in time $O(m)$. If all edge capacities are integral, then it is easy to see that after each iteration the flow value increases by at least one. The total number of iterations is therefore at most $|f^*|$, where f^* is a

```
Input: directed graph $G = (V, E)$, capacity function $c : E \to \mathbb{R}^+$, source and
       destination nodes $s, t \in V$
Output: maximum flow $f : V \times V \to \mathbb{R}$
1 Initialize: $f(u, v) = 0$ for every $u, v \in V$
2 while there exists an augmenting path $P$ in $G_f$ do
3     augment flow $f$ along $P$
4 end
5 return $f$
```

Algorithm 7: Ford-Fulkerson algorithm for the max-flow problem.

maximum flow. Note that we can also handle the case when each capacity is a rational number by scaling all capacities by a suitable integer D. However, note that the worst case running time of the Ford-Fulkerson algorithm can be prohibitively large. An instance on which the algorithm admits a bad running time is given in Figure 7.

Theorem 5.1. *The Ford-Fulkerson algorithm solves the max-flow problem with integer capacities in time $O(m|f^*|)$, where f^* is a maximum flow.*

Ford and Fulkerson gave an instance of the max-flow problem that shows that for irrational capacities the algorithm might fail to terminate.

Note that if all capacities are integers then the algorithm maintains an integral flow. That is, the Ford-Fulkerson also gives an algorithmic proof of the following integrality property.

Theorem 5.2 (Integrality property)**.** *If all capacities are integral, then there is an integer maximum flow.*

5.4 Max-Flow Min-Cut Theorem

A *cut* (or s,t-*cut*) of G is a partition of the node set V into two sets: X and $\bar{X} = V \setminus X$ such that $s \in X$ and $t \in \bar{X}$. Recall that G is directed. Thus, there are two types of edges crossing the cut $(X, \bar{X})$, namely the ones that leave X and the ones that enter X. As for flows, it will be convenient to define for $X, Y \subseteq V$,

$$c(X,Y) = \sum_{u \in X} \sum_{v \in Y} c(u,v).$$

The *capacity* of a cut $(X, \bar{X})$ is defined as the total capacity $c(X, \bar{X})$ of the edges leaving X. Fix an arbitrary flow f in G and consider an an arbitrary cut $(X, \bar{X})$ of G. The total net flow leaving X is $|f|$.

Lemma 5.3. *Let f be a flow and let $(X, \bar{X})$ be a cut of G. Then the net flow leaving X is $f(X, \bar{X}) = |f|$.*

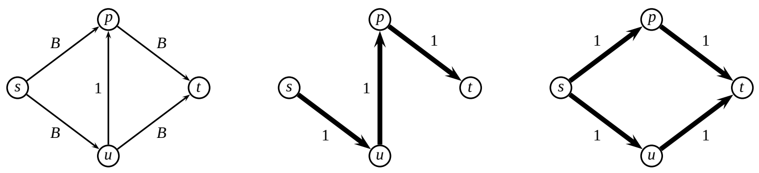


Figure 7: A bad instance for the Ford-Fulkerson algorithm (left). Suppose that B is a large integer. The algorithm alternately augments one unit of flow along the two paths $\langle s,u,p,t\rangle$ and $\langle s,p,u,t\rangle$. The flow after two augmentations is shown on the right. The algorithm needs $2B$ augmentations to find a maximum flow.

Proof.

$$f(X,V\setminus X) = f(X,V) - f(X,X) = f(X,V) = f(s,V) + f(X-s,V) = f(s,V) = |f|.$$

□

Intuitively, it is clear that if we consider an arbitrary cut $(X,\bar{X})$ of G, then the total flow $f(X,\bar{X})$ that leaves X is at most $c(X,\bar{X})$. The next lemma shows this formally.

Lemma 5.4. *The flow value of any flow f in G is at most the capacity of any cut $(X,\bar{X})$ of G, i.e., $f(X,\bar{X}) \le c(X,\bar{X})$.*

Proof. By Lemma 5.3, we have

$$|f| = f(X,\bar{X}) = \sum_{u\in X}\sum_{v\in V\setminus X} f(u,v) \le \sum_{u\in X}\sum_{v\in V\setminus X} c(u,v) = c(X,\bar{X}).$$

□

A fundamental result for flows is that the value of a maximum flow is equal to the minimum capacity of a cut.

Theorem 5.3 (Max-Flow Min-Cut Theorem)**.** *Let f be a flow in G. Then the following conditions are equivalent:*

1. *f is a maximum flow of G.*
2. *The residual graph G_f contains no augmenting path.*
3. *$|f| = c(X,\bar{X})$ for some cut $(X,\bar{X})$ of G.*

Proof. (1) ⇒ (2): Suppose for the sake of contradiction that f is a maximum flow of G and there is an augmenting path P in G_f. By Lemma 5.2, we can augment f along P and obtain a flow of value strictly larger than $|f|$, which is a contradiction.

(2) ⇒ (3): Suppose that G_f contains no augmenting path. Let X be the set of nodes that are reachable from s in G_f. Note that $t \notin X$ because there is no path from s to t in G_f. That is, $(X,\bar{X})$ is a cut of G. By Lemma 5.3, $|f| = f(X,\bar{X})$. Moreover, for every $u \in X$ and $v \in \bar{X}$, we must have $f(u,v) = c(u,v)$ because otherwise $(u,v) \in E_f$ and v would be part of X. We conclude $|f| = f(X,\bar{X}) = c(X,\bar{X})$.

(3) $\Rightarrow$ (1): By Lemma 5.4, the value of any flow is at most the capacity of any cut. The condition $|f| = c(X,\bar{X})$ thus implies that f is a maximum flow. (Also note that this implies that $(X,\bar{X})$ must be a cut of minimum capacity.) $\square$

5.5 Edmonds-Karp Algorithm

The Edmonds-Karp Algorithm works almost identical to the Ford-Fulkerson algorithm. The only difference is that it chooses in each iteration a *shortest* augmenting path in the residual graph G_f. An augmenting path is a *shortest* augmenting path if it has the minimum number of edges among all augmenting paths in G_f. The algorithm is given in Algorithm 8.

```
Input: directed graph $G = (V,E)$, capacity function $c : E \to \mathbb{R}^+$, source and
       destination nodes $s,t \in V$
Output: maximum flow $f : V \times V \to \mathbb{R}$
1 Initialize: $f(u,v) = 0$ for every $u,v \in V$
2 while there exists an augmenting path in $G_f$ do
3     determine a shortest augmenting path $P$ in $G_f$
4     augment $f$ along $P$
5 end
6 return $f$
```

Algorithm 8: Edmonds-Karp algorithm for the max-flow problem.

Note that each iteration can still be implemented to run in time $O(m)$. A shortest augmenting path in G_f can be found by using a breadth-first search algorithm. As we will show, this small change makes a big difference in terms of the running time of the algorithm.

Theorem 5.4. *The Edmonds-Karp algorithm solves the max-flow problem in time $O(nm^2)$.*

Note that the correctness of the Edmonds-Karp algorithm follows from the max-flow min-cut theorem: The algorithm halts when there is not augmenting path in G_f. By Theorem 5.3, the resulting flow is a maximum flow. It remains to show that the algorithm terminates after $O(nm)$ iterations. The crucial insight in order to prove this is that the shortest path distance of a node can only increase as the algorithm progresses.

Fix an arbitrary iteration of the algorithm. Let f be the flow at the beginning of the iteration and let f' be the flow at the end of the iteration. We obtain f' from f by augmenting f along an augmenting path P in G_f. Further, P must be a shortest augmenting path in G_f. Let $P = \langle s = v_0, \dots, v_k = t \rangle$. We define two distance functions (in terms of number of edges on a path): Let $\delta(u,v)$ be the number of edges on a shortest path from u to v in G_f. Similarly, let $\delta'(u,v)$ be the number of edges on a shortest path from u to v in $G_{f'}$.

Note that $\delta(s,v_i) = i$. Also observe that if an edge (u,v) is part of $G_{f'}$ but not part of G_f, then $u = v_i$ and $v = v_{i-1}$ for some i. To see this observe that $f(u,v) = c(u,v)$ because $(u,v) \notin E_f$. On the other hand, $f'(u,v) < c(u,v)$ because $(u,v) \in E_{f'}$. That is, by augmenting f along P, the flow on edge (u,v) was decreased. That means that the flow on the reverse edge (v,u) was increased and thus (v,u) must be part of P.

The next lemma shows that for every node $v \in V$, the shortest path distance from s to v does not decrease.

Lemma 5.5. *For each $v \in V$, $\delta'(s,v) \geq \delta(s,v)$.*

Proof. Suppose there exists a node v with $\delta'(s,v) < \delta(s,v)$. Among all such nodes, let v be one with $\delta'(s,v)$ being smallest. Note that $v \neq s$ because $\delta(s,s) = \delta'(s,s) = 0$ by definition. Let P' be a shortest s,v-path in $G_{f'}$ of distance $\delta'(s,v)$ and let u be the second-last node of P'. Because P' is a shortest path and by the choice of v, we have

$$\delta(s,v) > \delta'(s,v) = \delta'(s,u) + 1 \geq \delta(s,u) + 1. \tag{5}$$

Note that the distance function δ satisfies the triangle inequality. Therefore, edge (u,v) cannot be part of G_f because otherwise we would have $\delta(s,v) \leq \delta(s,u)+1$. That is, (u,v) is contained in $G_{f'}$ but not contained in G_f. Using our observation above, we conclude that there is some i $(1 \leq i \leq k)$ such that $u = v_i$ and $v = v_{i-1}$. But then $\delta(s,v) = i - 1$ and $\delta(s,u) = i$ which is a contradiction to (5). ☐

Consider an augmentation of the current flow f along path P. We say that an edge $e = (u,v)$ is *critical* with respect to f and P if (u,v) is part of the augmenting path P and its residual capacity coincides with the amount of flow that is pushed along P, i.e., $r_f(u,v) = r_f(P)$. Note that after the augmentation of f along P, every critical edge on P will be saturated and thus vanishes from the residual network.

Lemma 5.6. *The number of times an edge $e = (u,v)$ is critical throughout the execution of the algorithm is bounded by $O(n)$.*

Proof. Suppose edge $e = (u,v)$ is critical with respect to flow f and path P. Let δ refer to the shortest path distances in G_f. We have

$$\delta(s,v) = \delta(s,u) + 1.$$

After the augmentation of f along P, edge e is saturated and thus disappears from the residual graph. It can only reappear in the residual graph when in a successive iteration some positive flow is pushed over the reverse edge (v,u). Suppose edge (v,u) is part of an augmenting path P' that is used to augment the current flow, say f'. Let δ' be the distance function with respect to $G_{f'}$. We have

$$\delta'(s,u) = \delta'(s,v) + 1.$$

By Lemma 5.5, $\delta(s,v) \leq \delta'(s,v)$ and thus

$$\delta'(s,u) = \delta'(s,v) + 1 \geq \delta(s,v) + 1 = \delta(s,u) + 2.$$

That is, between any two augmentations for which edge $e = (u,v)$ is critical, the distance of u from s must increases by at least 2.

Note that the distance of u from s is at least 0 initially and can never be more than $n-2$. The number of times edge $e = (u,v)$ is critical is thus bounded by $O(n)$. $\square$

The proof of Theorem 5.4 now follows trivially:

Proof of Theorem 5.4. As argued above, the Edmonds-Karp algorithm computes a maximum flow if it terminates. Note that in every iteration of the algorithm at least one edge is critical. By Lemma 5.6, every edge is at most $O(n)$ times critical. The number of iterations is thus bounded by $O(nm)$. $\square$

References

The presentation of the material in this section is based on [3, Chapter 27] and [2, Chapter 3]

6. Minimum Cost Flows

6.1 Introduction

We consider the *minimum cost flow problem.*

Minimum Cost Flow Problem:

Given:	A directed graph $G = (V,E)$ with capacities $w : E \to \mathbb{R}^+$ and costs $c : E \to \mathbb{R}^+$ and a balance function $b : V \to \mathbb{R}$.
Goal:	Compute a feasible flow f such that the overall cost $\sum_{e \in E} c(e) f(e)$ is minimized.

Here, a flow $f : E \to \mathbb{R}^+$ is said to be *feasible* if it respects the capacity constraints and the total flow at every node $u \in V$ is equal to the balance $b(u)$. More formally, f is feasible if the following two conditions are satisfied:

1. **Capacity constraint:** for every $(u,v) \in E$, $f(u,v) \le w(u,v)$.
2. **Flow balance constraints:** for every $u \in V$,

$$\sum_{(u,v)\in E} f(u,v) - \sum_{(v,u)\in E} f(v,u) = b(u).$$

Intuitively, a positive balance indicates that node u has a *supply* of $b(u)$ units of flow, while a negative balance indicates that node u has a *demand* of $-b(u)$ units of flow. A feasible flow f that satisfies the flow balance constraints with $b(u) = 0$ for every $u \in V$ is called a *circulation*.

The *minimum cost flow problem* can naturally be formulated as a linear program:

$$\begin{array}{lrcll} \text{minimize} & \sum_{e\in E} c(e) f(e) & & & \\ \text{subject to} & \sum_{(u,v)\in E} f(u,v) - \sum_{(v,u)\in E} f(v,u) & = & b(u) & \forall u \in V \\ & f(e) & \le & w(e) & \forall e \in E \\ & f(e) & \ge & 0 & \forall e \in E \end{array} \tag{6}$$

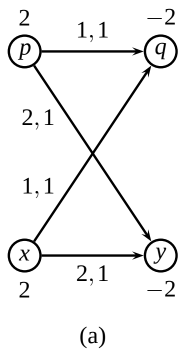
We use $c(f) = \sum_{e\in E} c(e) f(e)$ to refer to the total cost of a feasible flow f. We make a few assumptions throughout this section:

Assumption 6.1. *Capacities, costs and balances are integral.*

Note that we can enforce this assumption if all input numbers are rational numbers by multiplying by a suitably large constant.

Assumption 6.2. *The balance function satisfies* $\sum_{u\in V} b(u) = 0$ *and there is a feasible flow satisfying these balances.*

Note that we can test whether a feasible flow exists by a single max-flow computation as follows: Augment the network by adding a super-source s and a super-target t. Add



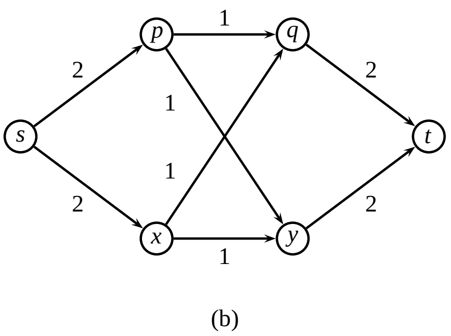


Figure 8: (a) Minimum cost flow instance. Every edge (u,v) is labeled with $c(u,v), w(u,v)$ and every node u is labeled with $b(u)$. (b) Augmented network to test feasibility. Every edge (u,v) is labeled with $w(u,v)$.

an edge (s,u) for every node $u \in V$ with $b(u) > 0$ of capacity $w(s,u) = b(u)$. Similarly, add an edge (u,t) for every node $u \in V$ with $b(u) < 0$ of capacity $w(u,t) = -b(u)$. Now, compute a maximum s,t-flow in the augmented network. It is not hard to see that there is a feasible flow for the original instance if and only if the maximum flow saturates all edges out of s (or, equivalently, into t).

Subsequently, let $W = \max_{e \in E} w(e)$ refer to the maximum capacity of an edge, $C = \max_{e \in E} c(e)$ to the maximum edge cost and $B = \max_{u \in V} b(u)$ to the maximum balance.

6.2 Flow Decomposition and Residual Graph

We establish a few basic properties of flows and circulations and introduce the important concept of residual graphs.

Lemma 6.1. *Let f be circulation of G. Then f can be decomposed into at most $m = |E|$ directed simple cycle flows.*

Proof. Let $G_f^+ = (V, E_f^+)$ be the *support subgraph* of G that contains all edges with positive flow value with respect to f, i.e.,

$$E_f^+ = \{e \in E \mid f(e) > 0\}.$$

Consider an arbitrary directed simple cycle C in G_f^+. Let x be the smallest flow value of an edge in C, i.e., $x = \min_{e \in C} f(e)$. We can decompose f such that $f = f' + f_C$, where $f_C(u,v) = x$ for every $(u,v) \in C$ and $f_C(u,v) = 0$ otherwise. Note that f' is a circulation and f_C is a cycle flow. We can now repeat this procedure with f' instead of f. Note that at least one edge e of C must satisfy $f'(e) = 0$ and thus vanishes from the support graph of f'. After at most m iterations, we therefore obtain a decomposition of f consisting of at most m directed simple cycle flows. □

As for the maximum flow problem, the concept of a *residual graph* will play a crucial role: Suppose f is a feasible flow of G. We introduce for each edge $e = (u,v) \in E$ the reverse edge (v,u) with cost $c(v,u) = -c(u,v)$. Subsequently, these edges will be called *backward edges*. In contrast, we refer to the original edges $(u,v) \in E$ as *forward edges*. The *residual capacity* of a forward edge (u,v) is defined as $r_f(u,v) = w(u,v) - f(u,v)$. The residual capacity of a backward edge (v,u) is $r_f(v,u) = f(u,v)$. The *residual graph* $G_f = (V,E_f)$ with respect to f is the graph that contains all edges with positive residual capacity.

Consider a directed simple cycle C in the residual graph G_f. Let the *residual capacity* of C be $r_f(C) = \min_{e\in C} r_f(e)$. We can then push $x = r_f(C)$ additional units of flow along C to obtain a feasible flow f'. Observe that an increase of x units on a backward edge (v,u) corresponds to a decrease of x units on the forward edge $(u,v) \in E$. More formally, the flow f' that we obtain from f by *augmenting x units of flow along C* is defined as follows: for every edge $e = (u,v) \in E$, we have

$$f'(u,v) = \begin{cases} f(u,v) + x & \text{if } (u,v) \in C \\ f(u,v) - x & \text{if } (v,u) \in C \\ f(u,v) & \text{otherwise.} \end{cases}$$

Let the total cost of a cycle C in G_f be $c(C) = \sum_{e\in C} c(e)$.

Lemma 6.2. *Let f be a feasible flow of G and let C be a directed simple cycle in G_f. Suppose f' is a flow that is obtained from f by augmenting $x = r_f(C)$ units of flow along C. Then f' is a feasible flow of G. Moreover, we have $c(f') = c(f) + x \cdot c(C)$.*

Proof. Observe that $x \le r_f(u,v)$ for every edge $(u,v) \in C$. If $(u,v) \in C$ is a forward edge, then $f'(u,v) = f(u,v) + x \le f(u,v) + w(u,v) - f(u,v) \le w(u,v)$. If $(v,u) \in C$ is a backward edge, then $f'(u,v) = f(u,v) - x \ge f(u,v) - f(u,v) = 0$. The new flow f' therefore respects the capacity and non-negativity constraints.

Note that by pushing x units of flow along C, the flow at a node u that is not part of C remains the same. Consider a node u that is part of C. Because C is simple there are exactly two edges of C incident to u, say e_1 and e_2. Note that the flow on all other edges incident to u remains the same. Also, the flow at u remains the same by pushing x units of flow along e_1 and e_2. (Note that in order to verify this we need to consider four different cases, depending on whether e_1 and e_2 are forward or backward edges.) We therefore have

$$\sum_{(u,v)\in E} f'(u,v) - \sum_{(v,u)\in E} f'(v,u) = \sum_{(u,v)\in E} f(u,v) - \sum_{(v,u)\in E} f(v,u) = b(u).$$

The flow balance constraints are therefore satisfied.

Finally, observe that by pushing x units of flow along C we effectively increase the cost of the flow by $x \cdot c(u,v)$ for every forward edge $(u,v) \in C \cap E$ and decrease the cost of the flow by $x \cdot c(u,v)$ for every backward edge $(v,u) \in C \setminus E$. The total cost of f' is

$$c(f') = \sum_{e \in E} c(e) f'(e) = \sum_{e \in E} c(e) f(e) + \sum_{(u,v) \in C \cap E} x \cdot c(u,v) - \sum_{(v,u) \in C \setminus E} x \cdot c(u,v)$$
$$= c(f) + x \sum_{e \in C} c(e) = c(f) + x \cdot c(C).$$

$\square$

We can generalize the above lemma as follows:

Lemma 6.3. *Let f be a feasible flow of G and let g be a circulation of G_f that respects the residual capacities r_f. Then f' can be obtained from f by augmenting along k simple cycles $C_1, \dots, C_k$ with $k \leq 2m$ such that $c(f') = c(f) + c(g)$, where $c(g) = \sum_{e \in E_f} c(e) g(e)$ is the total cost of g in the residual graph G_f.*

Proof. Using Lemma 6.1, we can decompose g into at most k directed simple cycle flows $f_{C_1}, \dots, f_{C_k}$ with $k \leq 2m$. (Recall that G_f has at most $2m$ edges.) Each cycle C_i corresponds to a directed cycle in G_f. By pushing f_{C_i} units of flow along every cycle C_i $(1 \leq i \leq k)$, we obtain a new flow f'.[2] From Lemma 6.2 it follows that f' is a feasible flow of G of total cost

$$c(f') = c(f) + \sum_{i=1}^{k} f_{C_i} \cdot c(C_i) = c(f) + \sum_{e \in E_f} c(e) g(e) = c(f) + c(g).$$

$\square$

Lemma 6.4. *Let f and f' be two feasible flows of G. Then f' can be obtained from f by augmenting flow along at most m cycles in G_f.*

Proof. Consider the difference $h = f' - f$. Let E^+ be the set of edges $(u,v) \in E$ with $h(u,v) > 0$. Similarly, let E^- be the set of edges $(u,v) \in E$ with $h(u,v) < 0$. Define a flow g as follows: $g(u,v) = h(u,v)$ for every edge E^+ and $g(v,u) = -h(u,v)$ for every edge $(u,v) \in E^-$. We claim that g is a circulation in G_f.

Note that for every edge $(u,v) \in E^+$ we have $0 < g(u,v) = h(u,v) = f'(u,v) - f(u,v) \leq w(u,v) - f(u,v) = r_f(u,v)$. Thus, $(u,v) \in E_f$ and $g(u,v) \leq r_f(u,v)$. Similarly, for every edge $(u,v) \in E^-$ we have $0 < g(v,u) = -h(u,v) = f(u,v) - f'(u,v) \leq f(u,v) = r_f(v,u)$. Thus, $(v,u) \in E_f$ and $g(v,u) \leq r_f(v,u)$. The flow g therefore respects the residual capacities of G_f.

It remains to be shown that g satisfies the flow balance constraints: Note that because both f' and f are feasible flows in G we have for every node $u \in V$

$$\sum_{(u,v) \in E} h(u,v) - \sum_{(v,u) \in E} h(v,u) = 0.$$

[2] We slightly abuse notation here and let f_{C_i} also refer to the flow value that is pushed along C_i.

Using this and the definition of h, we obtain

$$\begin{aligned}
0 &= \sum_{(u,v)\in E^+} g(u,v) - \sum_{(u,v)\in E^-} g(v,u) - \sum_{(v,u)\in E^+} g(v,u) + \sum_{(v,u)\in E^-} g(u,v)\\
&= \sum_{(u,v)\in E_f} g(u,v) - \sum_{(v,u)\in E_f} g(v,u) - \sum_{(v,u)\in E_f} g(v,u) + \sum_{(u,v)\in E_f} g(u,v)\\
&= 2\Big(\sum_{(u,v)\in E_f} g(u,v) - \sum_{(v,u)\in E_f} g(v,u)\Big).
\end{aligned}$$

Here the second equality follows from the observations above: for every $(u,v) \in E^+$ we have $(u,v) \in E_f$, for every $(u,v) \in E^-$ we have $(v,u) \in E_f$, and $g(u,v)$ is non-zero only on the edges in E^+ and E^-. We conclude that g is a circulation of G_f.

The proof now follows from Lemma 6.3. (Note that there are at most m edges with positive flow in g. Thus, g can be decomposed into at most m cycles flows.) □

6.3 Cycle Canceling Algorithm

Lemma 6.2 shows that if we are able to find a cycle C in G_f of negative cost $c(C) < 0$, then we can augment $r_f(C)$ units of flow along this cycle and obtain a flow f' of cost strictly smaller than $c(f)$. This observation gives rise to our first optimality condition:

Theorem 6.1 (Negative cycle optimality condition). *A feasible flow f of G is a minimum cost flow if and only if G_f does not contain a negative cost cycle.*

Proof. Suppose f is a minimum cost flow and G_f contains a negative cost cycle C. By Lemma 6.2, we can augment $r_f(C)$ units of flow along C and obtain a feasible flow f' with

$$c(f') = c(f) + r_f(C) \cdot c(C) < c(f),$$

which is a contradiction.

Next suppose that f is a feasible flow and G_f contains no negative cycle. Let f^* be a minimum cost flow and assume $f^* \neq f$. By Lemma 6.4, f^* can be obtained from f by augmenting along k cycles $C_1, \dots, C_k$ in G_f, where $k \leq m$. Let f_{C_i} be the flow that is pushed along C_i. By Lemma 6.3, the cost of f^* is equal to

$$c(f^*) = c(f) + \sum_{i=1}^{k} f_{C_i} \cdot c(C_i).$$

By assumption, each such cycle has nonnegative cost and thus $c(f^*) \geq c(f)$. We conclude that f is a minimum cost flow. □

This leads to our first algorithm:

Note that we can establish a feasible flow f by computing a maximum flow as explained above. This takes $O(nm^2)$ using the Edmonds-Karp algorithm. Also observe that in each

```
Input: directed graph $G = (V,E)$, capacity function $w : E \to \mathbb{R}^+$, cost function
       $c : E \to \mathbb{R}^+$ and balance function $b : V \to \mathbb{R}$.
Output: minimum cost flow $f : E \to \mathbb{R}^+$.
1 Initialize: compute a feasible flow $f$
2 while $G_f$ contains a negative cost cycle do
3     find a directed simple negative cycle $C$ of $G_f$
4     push $r_f(C)$ flow units along $C$ and let $f$ be the new flow
5 end
6 return $f$
```

Algorithm 9: Cycle canceling algorithm.

iteration we have to determine a cycle of negative cost in G_f. Using the Bellman-Ford algorithm, this can be done in time $O(nm)$.

We next bound the number of iterations that the algorithm needs to compute a minimum cost flow. Note that an arbitrary flow has cost at most mWC because every edge has flow value at most W and cost at most C. On the other hand, a trivial lower bound on the cost of a minimum cost flow is 0, because all edge costs are nonnegative. Every iteration of the above algorithm strictly decreases the cost of the current flow f. Since we assume that all input data is integral, the cost of f decreases by at least 1. The algorithm therefore terminates after at most $O(mWC)$ iterations.

Theorem 6.2. *The cycle canceling algorithm computes a minimum cost flow in time $O(nm^2WC)$.*

Note that the running time of the cycle canceling algorithm is not polynomial because W and C might be exponential in n and m. Algorithms whose running time is polynomial in the input size (here n and m) and the magnitude of the largest number in the instance (here W and C) are said to have *pseudo-polynomial running time*.

As a byproduct, the cycle canceling algorithm shows that there always exists a minimum cost flow that is integral if all capacities and balances are integral (see Assumption 6.1).

Theorem 6.3 (Integrality property)**.** *If all capacities and balances are integral, then there is an integer minimum cost flow.*

Proof. The proof is by induction on the number of iterations. We can assume without loss of generality that the flow f after the initialization is integral: Recall that f is obtained by computing a maximum flow in an augmented network as indicated above. Because all capacities and balances are integral, this augmented network has integral capacities. The resulting flow is therefore integral by Theorem 5.2. Suppose that the current flow f is integral after i iterations. The residual capacities in G_f are then also integral and a push along an augmenting cycle maintains the integrality of the resulting flow. □

We remark that the above algorithm can be turned into a polynomial-time algorithm if in each iterations one augments along a *minimum mean cost cycle*, i.e., a cycle that minimizes the ratio $c(C)/|C|$. A minimum mean cost cycle can be computed in time $O(nm)$. Using this idea, one can show that the resulting algorithm has an overall running time of $O(n^2m^3\log n)$.

6.4 Successive Shortest Path Algorithm

We derive an alternative optimality condition. Suppose we associate a *potential* $\pi(u)$ with every node $u \in V$. Define the *reduced cost* $c^\pi(u,v)$ of an edge (u,v) as

$$c^\pi(u,v) = c(u,v) - \pi(u) + \pi(v).$$

Note that this definition is applicable to both the original network and the residual graph.

Theorem 6.4 (Reduced cost optimality conditions)**.** *A feasible flow f of G is a minimum cost flow if and only if there exist some node potentials $\pi : V \to \mathbb{R}$ such that $c^\pi(u,v) \geq 0$ for every edge $(u,v) \in E_f$ of G_f.*

Proof. Suppose that there exist node potentials such that $c^\pi(u,v) \geq 0$ for every edge $(u,v) \in E_f$ of the residual graph G_f. Let C be an arbitrary simple directed cycle in G_f. Then

$$\sum_{e\in C} c(e) = \sum_{e\in C} c^\pi(e) \geq 0.$$

We conclude that G_f does not contain any negative cost cycle. By Theorem 6.1, f is a minimum cost flow.

Let f be a minimum cost flow. By Theorem 6.1, G_f contains no negative cycle. Let $\delta : V \to \mathbb{R}$ be the shortest path distances from an arbitrarily chosen source node $s \in V$ to every other node $u \in V$ (with respect to c). Note that δ is well-defined because G_f contains no negative cycles. The distance function δ must satisfy the triangle inequality (see Lemma 4.2), i.e., for every edge $(u,v) \in E_f$, $\delta(v) \leq \delta(u) + c(u,v)$. Define $\pi(u) = -\delta(u)$ for every node $u \in V$. With this definition, we have for every $(u,v) \in E_f$:

$$c^\pi(u,v) = c(u,v) - \pi(u) + \pi(v) = c(u,v) + \delta(u) - \delta(v) \geq 0,$$

which concludes the proof. □

We next introduce the notion of a *pseudoflow*. A pseudoflow x of G is a function $x : E \to \mathbb{R}^+$ that satisfies the nonnegativity and capacity constraints; it need *not* satisfy the flow balance constraints. Given a pseudoflow x, define the *excess* of a node $u \in V$ as

$$exs(u) = b(u) + \sum_{(v,u)\in E} x(v,u) - \sum_{(u,v)\in E} x(u,v).$$

Intuitively, $exs(u) > 0$ means that node u has an excess of $exs(u)$ units of flow; $exs(u) < 0$ means that node u has a deficit of $-exs(u)$ units of flow. We refer to such nodes as *excess* and *deficit* nodes, respectively. A node u with $exs(u) = 0$ is said to be *balanced*. Let

V_x^+ and V_x^-, respectively, be the sets of excess and deficit nodes with respect to x. (For notational convenience, we will omit the subscript x subsequently.) Observe that

$$\sum_{u \in V} exs(u) = \sum_{u \in V} b(u) = 0 \qquad \text{and thus} \qquad \sum_{u \in V^+} exs(u) = -\sum_{u \in V^-} exs(u). \tag{7}$$

That is, if the network contains an excess node then it must also contain a deficit node.

The residual graph G_x of a pseudoflow x is defined in the same way as we defined the residual graph of a flow.

Lemma 6.5. *Suppose that a pseudoflow x satisfies the reduced cost optimality conditions with respect to some node potentials π. Let $\delta : V \to \mathbb{R}$ be the shortest path distances from some node $s \in V$ to all other nodes in G_x with respect to c^π and define $\pi' = \pi - \delta$. The following holds:*

1. *The pseudoflow x also satisfies the reduced cost optimality conditions with respect to the node potentials π'.*
2. *The reduced cost $c^{\pi'}(u,v)$ is zero for every edge $(u,v) \in E_x$ that is part of a shortest path from s to some other node in G_x.*

Proof. Since x satisfies the reduced cost optimality conditions with respect to π, we have $c^\pi(u,v) \geq 0$ for every edge $(u,v) \in E_x$. Moreover, δ is a distance function and therefore satisfies the triangle inequality, i.e., $\delta(v) \leq \delta(u) + c^\pi(u,v)$ for every $(u,v) \in E_x$. Thus, for every edge $(u,v) \in E_x$

$$\begin{aligned} c^{\pi'}(u,v) &= c(u,v) - (\pi(u) - \delta(u)) + (\pi(v) - \delta(v)) \\ &= c(u,v) - \pi(u) + \pi(v) + \delta(u) - \delta(v) \\ &= c^\pi(u,v) + \delta(u) - \delta(v) \geq 0. \end{aligned}$$

This proves the first part of the lemma.

Consider a shortest path P from node s to some other node t in G_x. Every edge $(u,v) \in P$ must be tight, i.e., $\delta(v) = \delta(u) + c^\pi(u,v)$. Substituting $c^\pi(u,v) = c(u,v) - \pi(u) + \pi(v)$, we obtain $\delta(v) = \delta(u) + c(u,v) - \pi(u) + \pi(v)$. Thus,

$$c^{\pi'}(u,v) = c(u,v) - \pi(u) + \pi(v) + \delta(u) - \delta(v) = 0,$$

which proves the second part of the lemma. □

Corollary 6.1. *Suppose that a pseudoflow x satisfies the reduced cost optimality conditions and we obtain x' from x by sending flow along a shortest path P (with respect to c^π) from node s to some other node t in G_x. Then x' also satisfies the reduced cost optimality conditions.*

Proof. Define the potentials $\pi' = \pi - \delta$ as in the statement of Lemma 6.5. Then $c^{\pi'}(u,v) = 0$ for every edge $(u,v) \in P$. Sending flow along an edge $(u,v) \in P$ might add the reversed edge (v,u) to the residual graph. It is not hard to verify that $c^{\pi'}(v,u) = -c^{\pi'}(u,v)$ and thus the new edge (v,u) also satisfies the reduced cost optimality condition. The claim follows. □

This corollary leads to the following idea: Start with an arbitrary pseudoflow x and potentials π such that the reduced cost optimality conditions are satisfied. We then repeatedly compute a shortest path P from some excess node $s \in V^+$ to a deficit node $t \in V^-$ in G_x with respect to c^π and push the maximum possible amount of flow from s to t along P. The shortest path distances are used to update π. The algorithm stops if no further excess node exists. Note that by the above corollary the pseudoflow x satisfies the reduced cost optimality conditions at all times. Eventually, x becomes a feasible flow. By Theorem 6.4, x is then a minimum cost flow. The algorithm is summarized in Algorithm 10; see Figure 9 for an illustration.

```
Input: directed graph $G = (V,E)$, capacity function $w : E \to \mathbb{R}^+$, cost function
       $c : E \to \mathbb{R}^+$ and balance function $b : V \to \mathbb{R}$.
Output: minimum cost flow $x : E \to \mathbb{R}^+$.
1  Initialize: $x(u,v) = 0$ for every $(u,v) \in E$ and $\pi(u) = 0$ for every $u \in V$
2  $exs(u) = b(u)$ for every $u \in V$
3  let $V^+ = \{u \in V \mid exs(u) > 0\}$ and $V^- = \{u \in V \mid exs(u) < 0\}$
4  while $V^+ \neq \emptyset$ do
5      choose a source node $s \in V^+$
6      compute shortest path distances $\delta : V \to \mathbb{R}$ from $s$ to all other nodes $u \in V$ in
       $G_x$ with respect to $c^\pi$
7      let $P$ be a shortest path from $s$ to some node $t \in V^-$
8      update $\pi \leftarrow \pi - \delta$
9      augment $\Delta = \min\{exs(s),\ -exs(t),\ r_x(P)\}$ units of flow along $P$
10     update $x$, $G_x$, $exs(s)$, $exs(t)$, $V^+$, $V^-$ and $c^\pi$
11 end
12 return $x$
```

Algorithm 10: Successive shortest path algorithm.

Theorem 6.5. *The successive shortest path algorithm computes a minimum cost flow in time $O(nB(m + n\log n))$.*

Proof. We show by induction on the number of iterations that the pseudoflow x satisfies the reduced cost optimality conditions with respect to π. This is sufficient to establish the correctness of the algorithm because the algorithm terminates with $V^+ = V^- = \emptyset$ and the final pseudoflow x is thus a flow. It then follows from Theorem 6.4 that x is a minimum cost flow.

After the initialization, x is a pseudoflow and $G_x = G$. Since $\pi(u) = 0$ for every $u \in V$, $c^\pi(u,v) = c(u,v)$ for every $(u,v) \in E_x$. Since all edge costs are assumed to be nonnegative, x satisfies the reduced cost optimality conditions with respect to π. Let x be the pseudoflow at the beginning of iteration i and assume that it satisfies the reduced cost optimality conditions with respect to π. The shortest path distances δ are well-defined because G_x does not contain a negative cycle with respect to c^π. By (7), V^+ is nonempty iff V^- is nonempty. The algorithm therefore succeeds in finding a shortest path from s to some node $t \in V^-$ because otherwise the problem would be infeasible. (Recall that we assume that there is a feasible solution; see Assumption 6.2.) By Corollary 6.1, the

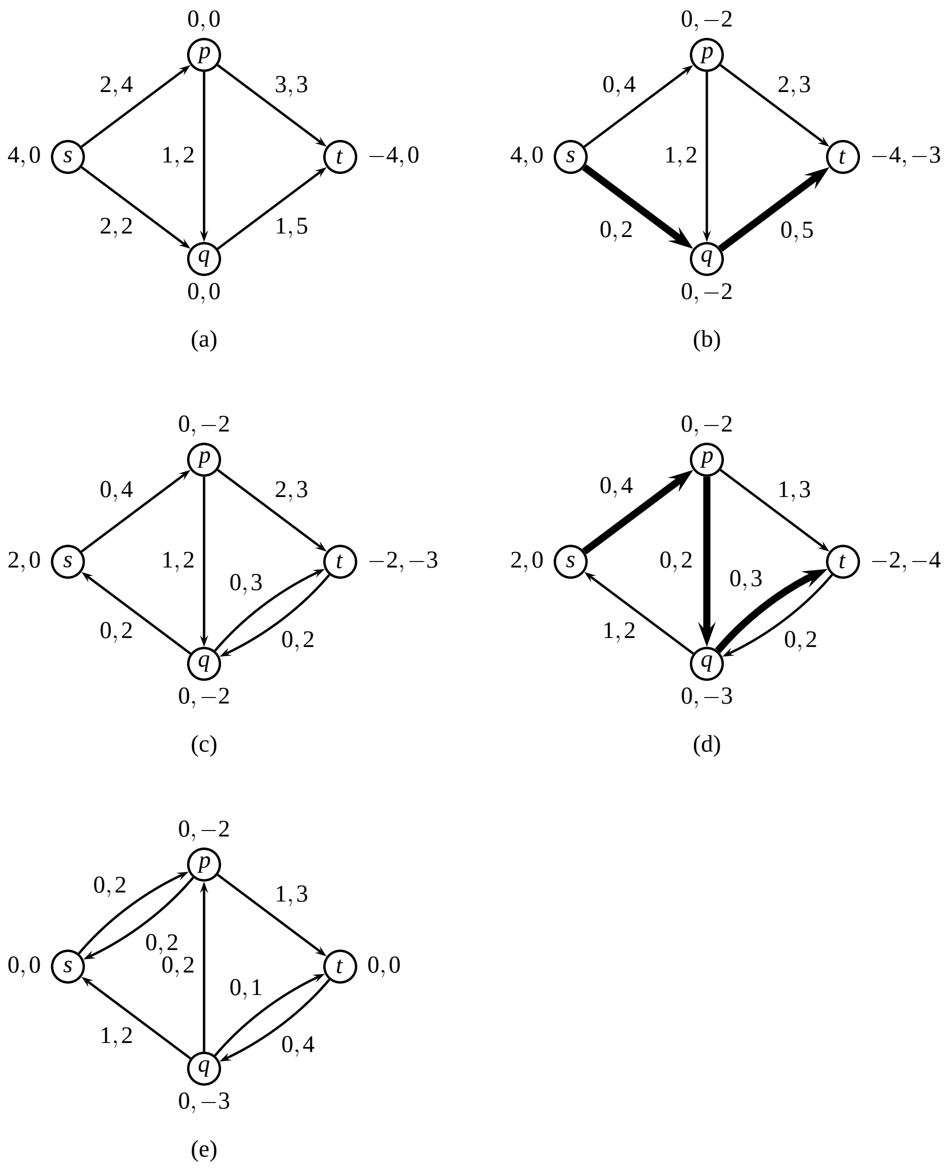


Figure 9: Illustration of the successive shortest path algorithm. The residual graph G_x with respect to the current pseudoflow x is depicted. Every edge (u,v) is labeled with $c^{\pi}(u,v), r_x(u,v)$ and every node u is labeled with $exs(u), \pi(u)$. (a) G_x with respect to $x = 0$ and $\pi = 0$. (b) G_x after potential update: two units of flow are sent along the bold path. (c) G_x after flow augmentation. (d) G_x after potential update: two units of flow are sent along the bold path. (e) G_x after flow augmentation: no further excess/deficit nodes exist and the resulting flow is optimal.

pseudoflow that we obtain from x by sending Δ units along P satisfies the reduced cost optimality conditions with respect to $\pi - \delta$.

It remains to be shown that the algorithm terminates. In each iteration, Δ is chosen such that either s or t become balanced or one of the edge on P becomes saturated. Each iteration therefore strictly reduces the excess of the chosen source node s. Since we assume that all input data is integral, the excess of s is reduced by at least 1. The algorithm therefore terminates after at most nB iterations. Each iteration requires to solve a single source shortest path problem with respect to c^π. Because the reduced costs c^π are nonnegative, we can use Dijkstra's algorithm which requires $O(m + n\log n)$ time. The overall running time of the successive shortest path algorithm is thus $O(nB(m + n\log n))$. $\square$

6.5 Primal-Dual Algorithm

We use linear programming duality to derive our third algorithm for the minimum cost flow problem. We associate a dual variable $\pi(u)$ with every node $u \in V$ and $\alpha(e)$ with every edge $e \in E$. The dual of the linear program (6) is as follows:

$$\begin{array}{lrcll} \text{maximize} & \sum_{u\in V} b(u)\pi(u) - \sum_{e\in E} w(e)\alpha(e) & & & \\ \text{subject to} & \pi(u) - \pi(v) - \alpha(u,v) & \leq & c(u,v) & \forall (u,v) \in E \\ & \alpha(e) & \geq & 0 & \forall e \in E \end{array} \tag{8}$$

As in the previous section, let the reduced cost of an edge $(u,v) \in E$ be defined as $c^\pi(u,v) = c(u,v) - \pi(u) + \pi(v)$. The above constraints then require that $-\alpha(u,v) \leq c^\pi(u,v)$ and $\alpha(u,v) \geq 0$ for every edge $(u,v) \in E$. Since the dual has a maximization objective and because capacities are nonnegative, an optimal solution to (8) satisfies $\alpha(u,v) = \max\{0, -c^\pi(u,v)\}$. In a sense, the dual variable $\alpha(u,v)$ are therefore redundant: Given optimal dual values $\pi(u)$ for every $u \in V$, we can extend this solution to a feasible dual solution (π, α) of (8) using the above relation.

We next derive the *complementary slackness conditions* of the primal linear program (6) and the dual linear program (8):

1. **Primal complementary slackness condition:** for every edge $e \in E$:

$$f(e) > 0 \quad \Rightarrow \quad \alpha(e) = -c^\pi(e),$$

   which is equivalent to

$$f(e) > 0 \quad \Rightarrow \quad c^\pi(e) \leq 0.$$

2. **Dual complementary slackness condition:** for every edge $e \in E$:

$$\alpha(e) > 0 \quad \Rightarrow \quad f(e) = w(e),$$

which is equivalent to

$$c^\pi(e) < 0 \quad \Rightarrow \quad f(e) = w(e).$$

Theorem 6.6 (Complementary slackness optimality conditions)**.** *A feasible flow f of G is a minimum cost flow if and only if there exist dual values $\pi(u)$ for every $u \in V$ satisfying that for every edge $e \in E$:*

1. *If $c^\pi(e) > 0$ then $f(e) = 0$.*
2. *If $c^\pi(e) < 0$ then $f(e) = w(e)$.*

Proof. The proof follows directly from the complementary slackness conditions. □

The complementary slackness optimality conditions can actually be seen to be equivalent to the reduced cost optimality conditions that we introduced earlier:

Theorem 6.7. *A feasible flow f satisfies the reduced cost optimality conditions with respect to node potentials $\pi : V \to \mathbb{R}$ if and only if f satisfies the complementary slackness optimality conditions with respect to π.*

Proof. Suppose $c^\pi(u,v) \geq 0$ for every $(u,v) \in E_f$. Let $(u,v) \in E$ and suppose $c^\pi(u,v) < 0$. Then $(u,v) \notin E_f$ and thus $f(u,v) = w(u,v)$. Next suppose $c^\pi(u,v) > 0$. Because $c^\pi(v,u) = -c^\pi(u,v) < 0$, the backward edge (v,u) is not part of G_f and thus $f(u,v) = 0$.

Assume that the complementary slackness conditions are satisfied for every edge $(u,v) \in E$. Consider a forward edge $(u,v) \in E_f$. Then $f(u,v) < w(u,v)$ and thus $c^\pi(u,v) \geq 0$. Next consider a backward edge $(v,u) \in E_f$. Then $f(u,v) > 0$ and thus $c^\pi(u,v) \leq 0$. Since $c^\pi(v,u) = -c^\pi(u,v)$, we conclude that $c^\pi(v,u) \geq 0$. □

The primal-dual algorithm for the minimum cost flow problem follows a general *primal-dual paradigm*: We start with an infeasible primal solution x and a feasible dual solution π. We ensure that the algorithm satisfies the complementary slackness conditions with respect to x and π throughout the entire execution of the algorithm. The algorithm successively reduces the degree of infeasibility of the primal solution x with respect to the current dual solution π. If no further improvement is possible, then π will be updated so as to ensure that the infeasibility of x can be further reduced. The dual solution π remains feasible throughout the entire process. Eventually, x is a feasible primal solution and thus a minimum cost flow.

The algorithm works with a transformed instance of the problem having exactly one excess and one deficit node: Augment the original graph by adding a super-source s and a super-targe t. Add an edge (s,u) for every node $u \in V$ with $b(u) > 0$ of capacity $w(s,u) = b(u)$ and cost $c(s,u) = 0$. Similarly, add an edge (u,t) for every node $u \in V$ with $b(u) < 0$ of capacity $w(u,t) = -b(u)$ and cost $c(u,t) = 0$. Let $b(s) = \sum_{u \in V : b(u) > 0} b(u)$ and $b(t) = \sum_{u \in V : b(u) < 0} b(u)$. All other balances are set to zero. Clearly, every minimum cost flow in the augmented network corresponds to a minimum cost flow in the original network and vice versa. Subsequently, we will use the augmented network.

The algorithm starts with the pseudoflow $x(e) = 0$ for every $e \in E$ and dual $\pi(u) = 0$ for every $u \in V$. Note that x is an infeasible primal solution and π is a feasible dual solution. Also x and π satisfy the complementary slackness conditions because for every edge $e \in E$, $x(e) = 0$ and $c^{\pi}(e) = c(e) \geq 0$.

In order to reduce the infeasibility of x, the algorithm basically pushes as much flow as possible from s to t along shortest s,t-paths in G_x. Let $\delta : V \to \mathbb{R}$ be the shortest path distances from s to all other nodes $u \in V$ in G_x with respect to c^{π}. Define $\pi' = \pi - \delta$. Then every shortest s,t-path in G_x with respect to c^{π} is a zero cost path with respect to $c^{\pi'}$ and vice versa. Let the *admissible graph* G_x^0 be the subgraph of $G_x = (V, E_x)$ that consists of all edges $e \in E_x$ with $c^{\pi'}(e) = 0$. The algorithm computes a maximum flow g^0 in G_x^0, where the capacities of the edges are their respective residual capacities. We can then augment x by g^0 in the obvious way: Increase the flow value $x(u,v)$ of every forward edge $(u,v) \in E$ by $g^0(u,v)$ and decrease the flow value $x(u,v)$ of every backward edge (v,u) by $g^0(v,u)$. As a result, the excess at s is reduced by the flow value $|g^0|$. It is not hard to see that the resulting flow x' is a pseudoflow. Moreover, in light of Theorem 6.7 and Corollary 6.1, x' satisfies the complementary slackness conditions with respect to the new feasible dual π'.

The algorithm continues in this manner until eventually the total excess of s is exhausted and the pseudoflow x becomes a flow. Since the algorithm maintains the invariant that x and π satisfy the complementary slackness conditions and π is a feasible dual solution, x (and also π) are eventually optimal solutions to the respective linear programs in (6) (and (8)).

The algorithm is summarized in Algorithm 11; see Figure 10 for an illustration.

```
Input: directed graph $G = (V,E)$, capacity function $w : E \to \mathbb{R}^+$, cost function
       $c : E \to \mathbb{R}^+$ and balance function $b : V \to \mathbb{R}$.
Output: minimum cost flow $x : E \to \mathbb{R}^+$.

1  Initialize: $x(u,v) = 0$ for every $(u,v) \in E$ and $\pi(u) = 0$ for every $u \in V$
2  $exs(s) = b(s)$
3  while $exs(s) > 0$ do
4      compute shortest path distances $\delta : V \to \mathbb{R}$ from $s$ to all other nodes $u \in V$ in
       $G_x$ with respect to $c^{\pi}$
5      update $\pi \leftarrow \pi - \delta$
6      construct the admissible network $G_x^0$
7      compute a maximum flow $g^0$ from $s$ to $t$ in $G_x^0$
8      augment $x$ by $g^0$
9      update $x$, $exs(s)$, $G_x$ and $c^{\pi}$
10 end
11 return $x$
```

Algorithm 11: Primal-dual algorithm.

Theorem 6.8. *The primal-dual algorithm computes a minimum cost flow in time* $O(\min\{nC, nB\} \cdot nm^2)$.

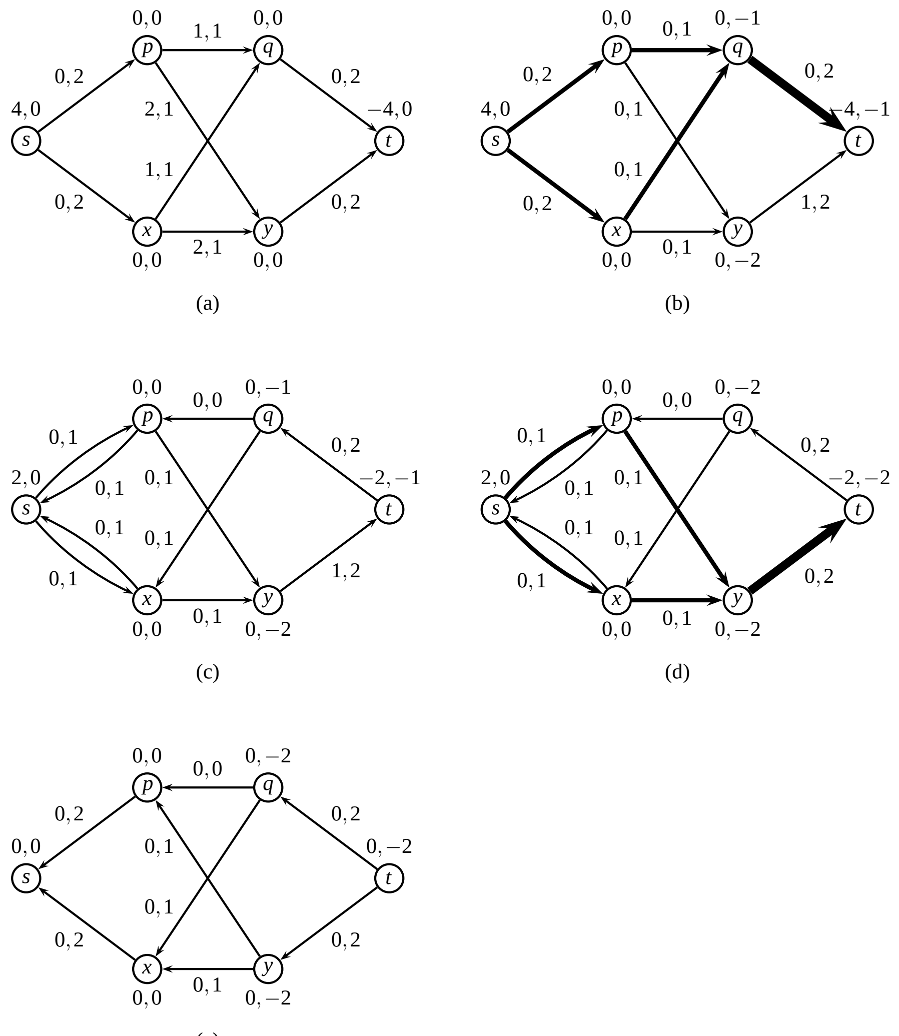


Figure 10: Illustration of the primal-dual algorithm. The residual graph G_x with respect to the current pseudoflow x is depicted. Every edge (u,v) is labeled with $c^\pi(u,v), r_x(u,v)$ and every node u is labeled with $exs(u), \pi(u)$. (a) G_x with respect to $x = 0$ and $\pi = 0$ of the transformed network of the example instance depicted in Figure 8(a). (b) G_x after potential update: max flow in G_x^0 has value 2. (c) G_x after flow augmentation. (d) G_x after potential update: max flow in G_x^0 has value 2. (e) G_x after flow augmentation: excess of source node s is zero and the resulting flow is optimal.

Proof. The correctness of the algorithm follows from the discussion above.

Observe that in each iteration, the excess of s is reduced by at least 1 (assuming integer capacities and balances). The maximum number of iterations is thus at most nB, which is the maximum excess of s at the beginning of the algorithm.

We can establish a second bound on the number of iterations: In each iteration, g^0 is a maximum flow in G_x^0. By the max-flow min-cut theorem, there is a cut $(X, \overline{X})$ in G_x^0 such that for every edge (u,v) with $u \in X$ and $v \in \overline{X}$, $g^0(u,v) = r_x(u,v)$. As a consequence, after the augmentation, all these edges vanish from the residual graph $G_{x'}$ of the new flow x'. Thus, every s,t-path in $G_{x'}$ has length at least 1 with respect to $c^{\pi'}(u,v)$ (because edge costs are integral). The potential of t therefore reduces by at least 1 in the next iteration. Note that no node potential $\pi(u)$ for $u \neq s$ can ever be less than $-nC$ (think about it!). The total number of iterations is therefore bounded by nC.

The running time of each iteration is dominated by the shortest path and max flow computations. The total running time is thus at most $O(\min\{nC, nB\} \cdot nm^2)$. □

References

The presentation of the material in this section is based on [1, Chapter 9].

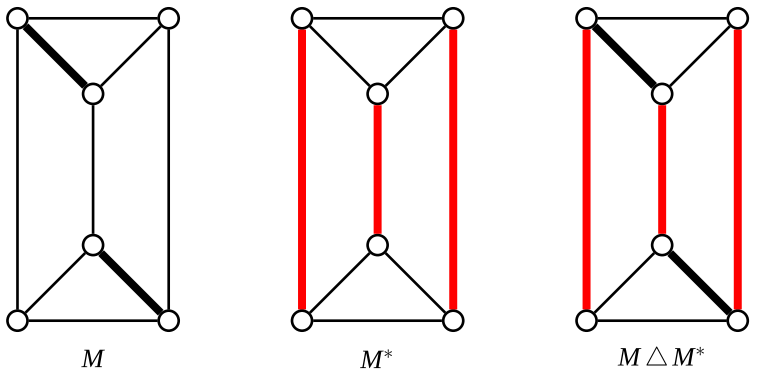


Figure 11: Illustration of the existence of an M-augmenting path.

7. Matchings

7.1 Introduction

Recall that a *matching* M in an undirected graph $G = (V,E)$ is a subset of edges satisfying that no two edges share a common endpoint. More formally, $M \subseteq E$ is a matching if for every two distinct edges $(u,v),(x,y) \in M$ we have $\{u,v\} \cap \{x,y\} = \emptyset$. Every node $u \in V$ that is incident to a matching edge is said to be *matched*; all other nodes are said to be *free*. A matching M is *perfect* if every node $u \in V$ is matched by M.

We consider the following optimization problem:

Maximum Matching Problem:

Given:	An undirected graph $G = (V,E)$.
Goal:	Compute a matching $M \subseteq E$ of G of maximum size.

Note that if the underlying graph is bipartite, then we can solve the maximum matching problem by a maximum flow computation.

Given two sets $S,T \subseteq E$, let $S \triangle T$ denote the *symmetric difference* of S and T, i.e., $S \triangle T = (S \setminus T) \cup (T \setminus S)$.

7.2 Augmenting Paths

Given a matching M, a path P is called *M-alternating* (or simply *alternating*) if the edges of P are alternately in M and not in M. If the first and last node of an M-alternating path P are free, then P is called an *M-augmenting* (or *augmenting*) path. Note that an augmenting path must have an odd number of edges. An M-augmenting path P can be used to increase the size of M: Simply make every non-matching edge on P a matching edge and vice versa. We also say that we *augment M along P*.

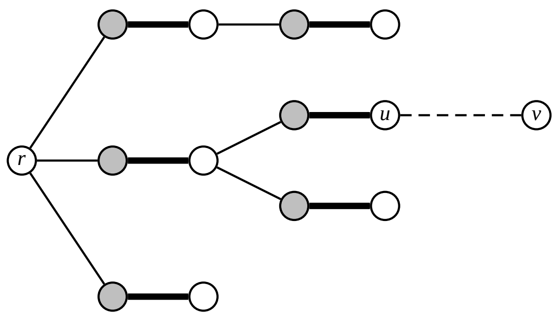


Figure 12: Illustration of an alternating tree. The nodes in X and Y are indicated in white and gray, respectively. Note that there is an augmenting path from r to v.

Theorem 7.1. *A matching M in a graph $G = (V,E)$ is maximum if and only if there is no M-augmenting path.*

Proof. Suppose M is maximum and there is an M-augmenting path P. Then augmenting M along P gives a new matching $M' = M \triangle P$ of size $|M|+1$, which is a contradiction.

Suppose that M is not maximum. Let M^* be a maximum matching. Consider the symmetric difference $M \triangle M^*$. Because M and M^* are matchings, the subgraph $G' = (V, M \triangle M^*)$ consists of isolated nodes and node-disjoint paths and cycles. The edges of every such path or cycle belong alternately to M and M^*. Each cycle therefore has an even number of edges. Because $|M^*| > |M|$ there must exist one path P that has more edges of M^* than of M. P is an M-augmenting path; see Figure 11 for an illustration. □

7.3 Bipartite Graphs

The above theorem gives an idea how to compute a maximum matching: Start with the empty matching $M = \emptyset$. Find an M-augmenting path P and augment M along P. Repeat this procedure until no M-augmenting path exists and M is maximum.

A natural approach to search for augmenting paths is to iteratively build an *alternating tree*. Suppose M is a matching and r is a free node. We inductively construct a tree T rooted at r as follows. We partition the node set of T into two sets X and Y: For every node $u \in X$, there is an even-length alternating path from r to u in T; for every node $u \in Y$, there is an odd-length alternating path from r to u in T. We start with $X = \{r\}$ and $Y = \emptyset$ and then iteratively extend T using the following operation:

```
EXTEND TREE USING $(u,v)$:
    (Precondition: $(u,v) \in E$, $u \in X$, $v \notin X \cup Y$ and $(v,w) \in M$)
    Add edge $(u,v)$ to $T$, $v$ to $Y$, edge $(v,w)$ to $T$ and $w$ to $X$
```

This way we obtain a layered tree rooted at r (starting with layer 0); see Figure 12 for an illustration. All nodes in X are on even layers and all nodes in Y are on odd layers. Moreover, every node in layer $2i-1$ ($i \geq 1$) is matched to a node in layer $2i$. In particular, $|X| = |Y| + 1$.

```
Input: undirected bipartite graph $G = (V, E)$.
Output: maximum matching $M$.
Initialize: $M = \emptyset$
foreach $r \in V$ do
    if $r$ is matched then continue
    else
        $X = \{r\}, Y = \emptyset, T = \emptyset$
        while there exists an edge $(u, v) \in E$ with $u \in X$ and $v \notin X \cup Y$ do
            if $v$ is free then AUGMENT MATCHING USING $(u, v)$
            else EXTEND TREE USING $(u, v)$
        end
    end
end
return $M$
```

Algorithm 12: Augmenting path algorithm.

Suppose that during the extension of the alternating tree T we encounter an edge $(u, v) \in E$ with $u \in X$ and $v \notin X \cup Y$ being a free node. We have then found an augmenting path from r to v; see Figure 12.

AUGMENT MATCHING USING (u, v)
(Precondition: $(u, v) \in E$, $u \in X$, $v \notin X \cup Y$ free)
Augment M along the concatenation of the r, u-path in T with edge (u, v)

These two operations form the basis of the augmeting path algorithm given in Algorithm 12.

The correctness of the algorithm depends on whether alternating trees truly capture all augmenting paths. Clearly, whenever the algorithm finds an augmenting path starting at r, this is an augmenting path. But can we conclude that there is no augmenting path if the algorithm does not find one? As it turns out, the algorithm works correctly if the underlying graph satisfies the *unique label property*: A graph satisfies the *unique label property* with respect to a given matching M and a root node r if the above tree building procedure uniquely assigns every node $u \in V(T)$ to one of the sets X and Y, irrespective of the order in which the nodes are examined.

Lemma 7.1. *Suppose a graph satisfies the unique label property. If there exists an M-augmenting path, then the augmenting path algorithm finds it.*

Proof. Let $P = \langle r, \ldots, u, v \rangle$ be an augmenting path with respect to M. Because of the unique label property, the algorithm always ends up with adding node u to X and thus discovers an augmenting path via edge (u, v). □

Using the above characterization, we can show that the augmenting path algorithm given in Algorithm 12 is correct for bipartite graphs: Recall that in a bipartite graph, the node set V is partitioned into two sets V_0 and V_1. Every node that is part of $V(T)$ and belongs

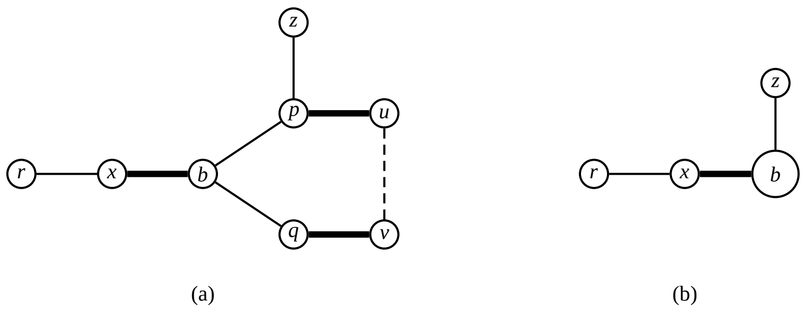


Figure 13: Illustration of a blossom shrinking. (a) The odd cycle $B = \langle b,p,u,v,q,b\rangle$ constitutes a blossom with base b and stem $\langle r,x,b\rangle$. Note that there is an augmenting path from z to r via edge (u,v). (b) The resulting graph after shrinking blossom B into a super-node b.

to the set V_i with $r \in V_i$ is added to X; those that belong to V_{1-i} are added to Y. Thus bipartite graphs satisfy the unique label property.

Theorem 7.2. *The augmenting path algorithm computes a maximum matching in bipartite graphs in time $O(nm)$.*

Proof. The correctness of the algorithm follows from the discussion above. Note that each iteration can be implemented to run in time $O(n+m)$ and there are at most n iterations. □

7.4 General Graphs

It is not hard to see that graphs do in general not satisfy the unique label property. Consider an odd cycle consisting of three edges $(r,u),(u,v),(v,r)$ and suppose that $(u,v) \in M$ and r is free. Then the algorithm adds u to Y if it considers edge (r,u) first, while it adds u to X if it considers edge (r,v) first. Odd cycles are precisely the objects that cause this dilemma (and which are not present in bipartite graphs).

A deep insight that was first gained by Edmonds in 1965 is that one can “shrink” such odd cycles. Suppose during the construction of the alternating tree, the algorithm encounters an edge (u,v) with $u,v \in X$; see Figure 13(a). Let b be the lowest common ancestor of u and v in T. Note that $b \in X$. Consider the cycle B that follows the unique b,u-path in T, then edge (u,v) and then the unique v,b-path in T. B is an odd length cycle, which is also called a *blossom*. The node b is called the *base* of B. The even length path from b to the root node r is called the *stem* of B; if $r = b$ then we say that the stem of B is empty. Suppose we shrink the cycle B to a super-node, which we identify with b; see Figure 13(b). Note that the super-node b belongs to X after shrinking.

```
SHRINK BLOSSOM USING $(u,v)$:
    (Precondition: $(u,v) \in E$ and $u,v \in X$)
    Let $b$ be the lowest common ancestor of $u$ and $v$ in $T$.
    Shrink the blossom $B = \langle b,\dots,u,v,\dots,b\rangle$ to a super-node $b$.
```

Let G' be the resulting graph and let M' be the restriction of M to the edges of G'. The next two lemmas show that by shrinking blossoms, we do not add or lose any augmenting paths.

Lemma 7.2. *Suppose there is an M'-augmenting path P' from r to v (or the respective super-node) in G'. Then there is an M-augmenting path from r to v in G.*

Proof. If P' does not involve the super-node b, then P' is also an augmenting path in G. Suppose P' contains the super-node b. There are two cases we need to consider:

Case 1: $r \neq b$. Let $P' = \langle r, \dots, x, b, z, \dots, v \rangle$ be the augmenting path in G'. Let P'_{rx} and P'_{zv} refer to the subpaths $\langle r, \dots, x \rangle$ and $\langle z, \dots, v \rangle$ of P', respectively. Note that $(x,b) \in M'$ and $(b,z) \notin M'$. If we expand the blossom B corresponding to super-node b, then b is the base of B with incident matching edge (x,b). Let p be the node of B such that (p,z) is part of G. Then there is an even length M-alternating path $P_{bp} = \langle b, \dots, p \rangle$ from b to p in B. The path $P = \langle P'_{rx}, (x,b), P_{bp}, (p,z), P'_{zv} \rangle$ is an M-augmenting path in G.

Case 2: $r = b$. Let $P' = \langle b, z, \dots, v \rangle$ be the augmenting path in G'. Let P'_{zv} refer to the subpath $\langle z, \dots, v \rangle$ of P'. If we expand the blossom B corresponding to super-node b, then b is the base of B which is free. Let p be the node of B such that (p,z) is part of G. Then there is an even length M-alternating path $P_{bp} = \langle b, \dots, p \rangle$ from b to p in B. The path $P = \langle b, P_{bp}, (p,z), P'_{zv} \rangle$ is an M-augmenting path in G. ☐

Lemma 7.3. *Suppose there is an M-augmenting path P from r to v in G. Then there is an M'-augmenting path from r to v (or the respective super-nodes) in G'.*

Proof. We assume without loss of generality that r and v are the only free nodes with respect to M. (Otherwise, we can remove all other free nodes from G without affecting P.) If P has no nodes in common with the nodes of the blossom B, then P is an M'-augmenting path in G' and we are done. Suppose $P = \langle r, \dots, v \rangle$ contains some nodes of B. We consider two cases:

Case 1: The stem of B is empty. The base b of B is then a free node and therefore coincides with one of the endpoints of P. Assume that $r = b$; the other case follows similarly. Let p be the last node of P that is part of B and let $P_{pv} = \langle p, z, \dots, v \rangle$ be the subpath of P starting at p. Note that $(p,z) \notin M$. The path $P' = \langle b, z, \dots, v \rangle$ is then an M'-augmenting path in G'.

Case 2: The stem of B is non-empty. Let $P_{rb} = \langle r, \dots, b \rangle$ be the stem of B. Consider the matching $\hat{M} = M \triangle P_{rb}$. Then r is matched in $\hat{M}$ and thus b and v are the only free nodes with respect to $\hat{M}$. Further, $|\hat{M}| = |M|$. Note that M is not a maximum matching (because there is an M-augmenting path in G) and thus also $\hat{M}$ is not a maximum matching. Thus, there is an $\hat{M}$-augmenting path $\hat{P}$ in G that starts at b and ends at v. Note that the stem of B with respect to $\hat{P}$ is empty and we can thus use the proof of Case 1 to show that the contracted graph G' contains an $\hat{M}'$-augmenting path from b to v. Note that $\hat{M}'$ is different from M'. However, because $|M'| = |\hat{M}'|$ we conclude that G' must contain an augmenting path with respect to M' as well. ☐

The matching algorithm for general graphs is also known as the *blossom-shrinking* algorithm. The algorithm maintains a graph G' of super-nodes and a respective matching M' on the super-nodes throughout each iteration. At the end of each iteration, all super-nodes of G' are expanded and the matching M on the original graph is obtained from M' as described in the proof of Lemma 7.2.

```
Input: undirected graph $G = (V, E)$.
Output: maximum matching $M$.
Initialize: $M = \emptyset$
foreach $r \in V$ do
    if $r$ is matched then continue
    else
        $G' \leftarrow G$ and $M' \leftarrow M$.
        $X \leftarrow \{r\}, Y \leftarrow \emptyset, T \leftarrow \emptyset$
        while there exists an edge $(u,v) \in E'$ with $u \in X$ and $v \notin Y$ do
            if $v$ is free, $v \neq r$ then AUGMENT MATCHING USING $(u,v)$
            else if $v \notin X \cup Y$, $(v,w) \in M'$ then EXTEND TREE USING $(u,v)$
            else SHRINK BLOSSOM USING $(u,v)$
        end
        Extend $M'$ to a matching $M$ of $G$ by expanding all super-nodes of $G'$.
    end
end
return $M$
```

Algorithm 13: Blossom shrinking algorithm.

Theorem 7.3. *The blossom-shrinking algorithm computes a maximum matching in general graphs in time $O(nm\alpha(n, \frac{m}{n}))$.*

Proof. The correctness of the algorithm follows from Lemmas 7.2 and 7.3 and Theorem 7.1. It remains to show that the algorithm can be implemented to run in time $O(m\alpha(n, \frac{m}{n}))$ per iteration. The key here is to maintain an implicit representation of the graph G' of super-nodes: We keep track of the partition of the original nodes into super-nodes by means of a *union-find* data structure. Considering an edge $(u,v) \in E$ during an iteration, we need to check whether edge (u,v) is part of G'. This can be done by verifying whether u and v belong to the same set of the partition. Shrinking a blossom is tantamount to uniting the node sets of the respective super-nodes. We have at most $2m$ find and n union operations per iterations and these operations take time $O(n + m\alpha(n, \frac{m}{n}))$. All remaining operations (extending the tree, augmenting the matching, extracting the matching on G) can be done in time $O(n+m)$ per iteration. The time bound follows. □

There are algorithms with better running times for the matching problem. For the bipartite case, Hopcroft and Karp showed that the running time of the augmenting path algorithm can be reduced to $O(\sqrt{n}m)$. The basic idea is to augment the current matching in each iteration by a maximal set of node-disjoint shortest paths (in terms of number of edges). Using this idea, one can show that the shortest path lengths increase with each iteration.

Now, fix an arbitrary matching M and suppose $|M| \leq k - \sqrt{k}$, where $k = |M^*|$ is the cardinality of a maximum matching. It is not hard to see that then there is an M-augmenting path of length at most $2\sqrt{k}+1$. That is, after at most $2\sqrt{k}+1$ iterations, the algorithm has found a matching M of size at least $k - \sqrt{k}$. After at most $\sqrt{k}$ additional iterations, the algorithm terminates with a maximum matching. Each iteration can be implemented to run in time $O(n+m)$, which gives a total running time $O(\sqrt{k}m) = O(\sqrt{n}m)$. A similar idea can be used in the general case to obtain an algorithm that computes a maximum matching in time $O(\sqrt{n}m)$.

References

The presentation of the material in this section is based on [1, Chapter 12] and [2, Chapter 5].

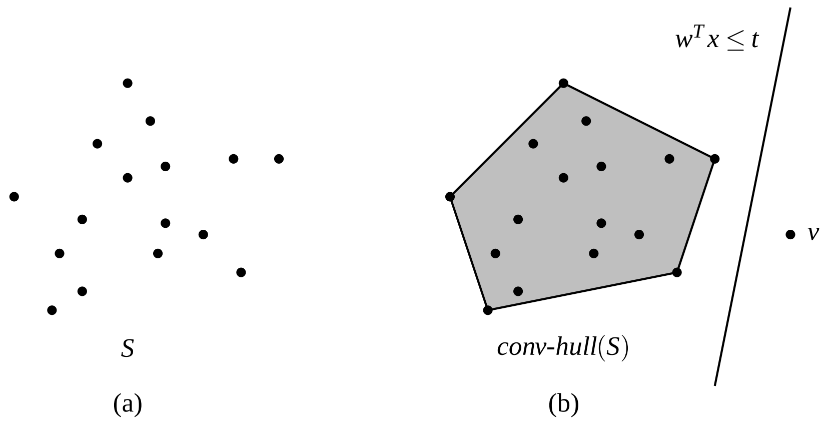


Figure 14: (a) Finite point set S. (b) Convex hull *conv-hull*(S) and a separating inequality for $v \notin$ *conv-hull*(S).

8. Integrality of Polyhedra

8.1 Introduction

Many algorithms for combinatorial optimization problems crucially exploit min-max relations in order to prove optimality of the computed solution. We have seen examples of such algorithms for the maximum flow problem, minimum cost flow problem and the matching problem. A question that arises is whether there is a general approach to derive such min-max relations. As we will see in this section, such relations can often be derived via *polyhedral methods*.

8.2 Convex Hulls

Suppose we are given a finite set $S = \{s_1, \dots, s_k\} \subseteq \mathbb{R}^n$ of n-dimensional vectors. A vector $x \in \mathbb{R}^n$ is a *convex combination* of the vectors in S if there exist non-negative scalars $\lambda_1, \dots, \lambda_k$ with $\sum_{i=1}^k \lambda_i = 1$ such that $x = \sum_{i=1}^k \lambda_i s_i$. The *convex hull* of S is defined as the set of all convex combinations of vectors in S. Subsequently, we use *conv-hull*(S) to refer to the convex hull of S.

Suppose we want to solve the following mathematical programming problem: Given some $w \in \mathbb{R}^n$, $\max\{w^T x \mid x \in S\}$. Intuitively, it is clear that this is the same as maximizing $w^T x$ over the convex hull of S.

Theorem 8.1. *Let $S \subseteq \mathbb{R}^n$ be a finite set and let $w \in \mathbb{R}^n$. Then*

$$\max\{w^T x \mid x \in S\} = \max\{w^T x \mid x \in \textit{conv-hull}(S)\}.$$

Proof. Let $x \in \textit{conv-hull}(S)$. Then

$$w^T x = \lambda_1 w^T s_1 + \cdots + \lambda_k w^T s_k \le \max\{w^T x \mid x \in S\}.$$

Thus $\max\{w^T x \mid x \in \textit{conv-hull}(S)\} \le \max\{w^T x \mid x \in S\}$. Equality now follows because $S \subseteq \textit{conv-hull}(S)$. □

The next proposition states that if $v \in \mathbb{R}^n \setminus \textit{conv-hull}(S)$ then there must exist a *separating inequality* $w^T x \le t$ that separates v from $\textit{conv-hull}(S)$, i.e., $w^T x \le t$ for all $x \in \textit{conv-hull}(S)$ but $w^T v > t$.

Theorem 8.2. *Let $S \subseteq \mathbb{R}^n$ be a finite set and let $v \in \mathbb{R}^n \setminus \textit{conv-hull}(S)$. Then there is a separating inequality $w^T x \le t$ that separates v from $\textit{conv-hull}(S)$.*

Proof. Note that verifying whether $v \in \textit{conv-hull}(S)$ is equivalent to checking whether there is a solution $(\lambda_1, \ldots, \lambda_k)$ to the following linear system:

$$\sum_{i=1}^{k} \lambda_i s_i = v \tag{9}$$

$$\sum_{i=1}^{k} \lambda_i = 1 \tag{10}$$

$$\lambda_i \ge 0 \qquad \forall i \in \{1, \ldots, k\} \tag{11}$$

Conversely, $v \in \mathbb{R}^n \setminus \textit{conv-hull}(S)$ iff the above linear system has no solution. Using Farkas Lemma (see below) with

$$A = \begin{pmatrix} s_{1,1} & \cdots & s_{k,1} \\ \vdots & \cdots & \vdots \\ s_{1,n} & \cdots & s_{k,n} \\ 1 & \cdots & 1 \end{pmatrix}, \qquad x = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_k \end{pmatrix} \qquad \text{and} \qquad b = \begin{pmatrix} v_1 \\ \vdots \\ v_n \\ 1 \end{pmatrix}$$

we obtain that $v \in \mathbb{R}^n \setminus \textit{conv-hull}(S)$ iff there exists a $y \in \mathbb{R}^n$ and $z \in \mathbb{R}$ such that

$$(y^T\ z)A \ge 0 \qquad \text{and} \qquad (y^T\ z)b < 0,$$

or, equivalently,

$$\begin{aligned} y^T s_i &\ge -z \qquad \forall i \in \{1, \ldots, k\} \\ y^T v &< -z. \end{aligned}$$

By setting $w = -y$ and $t = z$, we obtain that $w^T s_i \le t$ for every $i \in \{1, \ldots, k\}$. Theorem 8.1 implies that $w^T x \le t$ for every $x \in \textit{conv-hull}(S)$. Moreover, $w^T v > t$, which concludes the proof. □

We state the following proposition without proof.

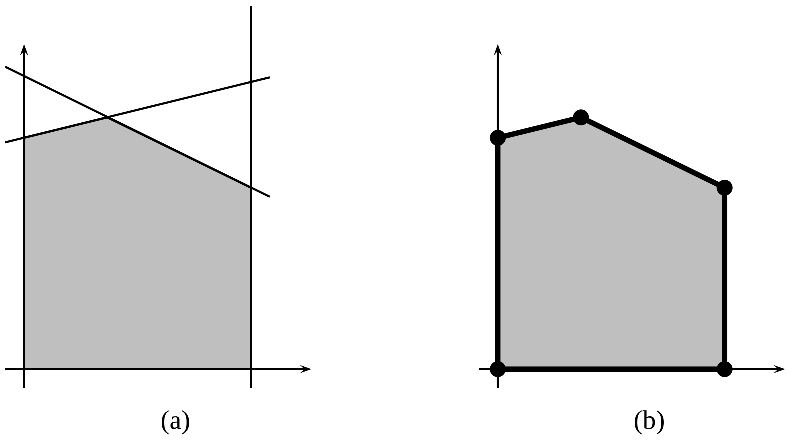


Figure 15: (a) Polytope described by five linear inequalities. (b) Faces of the polytope (indicated in bold).

Proposition 8.1 (Farkas Lemma)**.** *The system $Ax = b$ has a non-negative solution $x \geq 0$ if and only if there is no vector y such that $y^T A \geq 0$ and $y^T b < 0$.*

8.3 Polytopes

A *polyhedron* $P \subseteq \mathbb{R}^n$ is described by a system of linear inequalities, i.e., $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$. A polyhedron P is a *polytope* if P is bounded, i.e., there exist $l, u \in \mathbb{R}^n$ such that $l \leq x \leq u$ for every $x \in P$.

An inequality $w^T x \leq t$ is called *valid* for a polyhedron P if $P \subseteq \{x \in \mathbb{R}^n \mid w^T x \leq t\}$. A *hyperplane* is given by $\{x \in \mathbb{R}^n \mid w^T x = t\}$. It is called a *supporting hyperplane* if $w^T x \leq t$ is valid for P and $P \cap \{x \mid w^T x = t\} \neq \emptyset$. The intersection of a supporting hyperplane with P is called a *face*. In the plane, the faces of a polyhedron are the edges and corner points of P.

Lemma 8.1. *A non-empty set $F \subseteq P = \{x \mid Ax \leq b\}$ is a face of P if and only if for some subsystem $A^\circ x \leq b^\circ$ of $Ax \leq b$, we have $F = \{x \in P \mid A^\circ x = b^\circ\}$. Moreover, if F is an (inclusionwise) minimal face of P, then the rank of A° is equal to the rank of A.*

A vector $v \in P$ is a *vertex* of P if $\{v\}$ is a face of P. A polyhedron P is *pointed* if it has at least one vertex.

Lemma 8.2. *If a polyhedron P is pointed then every minimal non-empty face of P is a vertex.*

Lemma 8.3. *Let $P = \{x \mid Ax \leq b\}$ and $v \in P$. Then v is a vertex of P if and only if v cannot be written as a convex combination of vectors in $P \setminus \{v\}$.*

Proof. Suppose v is a vertex of P and let $A^\circ x \leq b^\circ$ be a subsystem of $Ax \leq b$ such that $\{v\} = \{x \in P \mid A^\circ x = b^\circ\}$. Suppose v can be written as a convex combination $\lambda_1 x_1 +$

$\cdots + \lambda_k x_k$ of vectors $x_1, \ldots, x_k \in P$. Then $A^\circ x_i = b^\circ$ for every $i \in \{1, \ldots, k\}$. But this is a contradiction to the assumption that v is the unique solution to the system $A^\circ x = b^\circ$.

Conversely, suppose v cannot be written as a convex combination of vectors in $P \setminus \{v\}$. Let $A^\circ x \leq b^\circ$ consist of the inequalities of $Ax \leq b$ which v satisfies with equality. Let $F = \{x \mid A^\circ x = b^\circ\}$. It suffices to show that $F = \{v\}$. Suppose that this is not true. Let $u \in F \setminus \{v\}$ and consider the line $L = \{v + \lambda(u - v) \mid \lambda \in \mathbb{R}\}$ through u and v. Clearly, $L \subseteq F$. For every inequality $a_i x \leq b_i$ of $Ax \leq b$ which is not part of $A^\circ x \leq b^\circ$, we have $a_i x < b_i$. We can therefore determine a sufficiently small $\varepsilon > 0$ such that $v^+ = v + \varepsilon(u - v) \in P$ and $v^- = v - \varepsilon(u - v) \in P$. But $v = \frac{1}{2}(v^+ + v^-)$, which is a contradiction. ☐

Theorem 8.3. *A polytope is equal to the convex hull of its vertices.*

Proof. Let P be a polytope. Since P is bounded, P must be pointed. Let $V = \{v_1, \ldots, v_k\}$ be the vertices of P. Clearly, $\mathit{conv\text{-}hull}(V) \subseteq P$. It remains to be shown that $P \subseteq \mathit{conv\text{-}hull}(V)$. Suppose there exists some $u \in P \setminus \mathit{conv\text{-}hull}(V)$. Then by Theorem 8.2, there exists an inequality $w^T x \leq t$ that separates u from $\mathit{conv\text{-}hull}(V)$, i.e., $w^T x \leq t$ for every $x \in \mathit{conv\text{-}hull}(V)$ and $w^T u > t$. Let $t^* = \max\{w^T x \mid x \in P\}$ and consider the face $F = \{x \in P \mid w^T x = t^*\}$. Because $u \in P$, we have $t^* \geq w^T u > t$. That is, F contains no vertex of P, which is a contradiction. ☐

Theorem 8.4. *A set P is a polytope if and only if there exists a finite set V such that P is the convex hull of V.*

The above theorem suggests the following approach to obtain a min-max relation for a combinatorial optimization problem.

1. Formulate the combinatorial problem Π as an optimization problem over a finite set S of feasible solutions (e.g., by considering all characteristic vectors).
2. Determine a linear description of $\mathit{conv\text{-}hull}(S)$.
3. Use duality of linear programming theory to obtain a min-max relation.

Note that by Theorem 8.1, solving the problem Π over S is equivalent to solving the problem over $\mathit{conv\text{-}hull}(S)$. By Theorem 8.4, there must exist a polyhedral description of $\mathit{conv\text{-}hull}(S)$. Thus, Π can be described as a linear program. Dualizing and using strong duality, we can deduce a min-max relation for the problem.

We remark that the results given above show that the above approach as such is applicable. However, there are (at least) two difficulties here: (i) It is not clear how to derive a linear description of $\mathit{conv\text{-}hull}(S)$ above. (ii) Even though such a description is guaranteed to exist, the number of linear inequalities might be by far larger than the size of the original problem. That is, even if we are able to come up with such a description, this might not lead to a polynomial-time algorithm.

We exemplify the above approach for the *perfect matching problem* in bipartite graphs. Let $G = (V, E)$ be a bipartite graph. Recall that a matching is *perfect* if every node of the graph is matched. Define $PM(G) \subseteq \mathbb{R}^E$ as the set of characteristic vectors of the perfect matchings of G.

Theorem 8.5 (Birkhoff's Theorem)**.** *Let $G = (V, E)$ be a bipartite graph. The convex hull* *of $PM(G)$ is defined as*

$$\begin{aligned} \sum_{e=(u,v)\in E} x_e &= 1 \quad \forall u \in V \\ x_e &\geq 0 \quad \forall e \in E \end{aligned} \tag{12}$$

Proof. Let P be the polytope defined by (12). Clearly, each perfect matching $x \in PM(G)$ is contained in P. It suffices to show that all vertices of P are integral. Suppose for the sake of contradiction that x is a vertex of P that is not integral. Let $\bar{E} = \{e \in E \mid 0 < x_e < 1\}$ be the fractional edges of x. Because $\sum_{e=(u,v)\in E} x_e = 1$ for every node $u \in V$, each node incident to an edge in $\bar{E}$ is incident to at least two edges in $\bar{E}$. Thus, there exists a cycle C in $\bar{E}$. Also, C must be even because G is bipartite. Let $d \in \mathbb{R}^E$ be a vector that is 0 for all edges not in C and alternately 1 and -1 for the edges along C. Because all edges of C are contained in $\bar{E}$, there is an $\varepsilon > 0$ such that $x^+ = x + \varepsilon d$ and $x^- = x - \varepsilon d$ are in P. Note that $x = \frac{1}{2}(x^+ + x^-)$. But this is a contradiction to the assumption that x is a vertex of P. $\square$

8.4 Integral Polytopes

Many combinatorial optimization problems can naturally be formulated as an integer linear program. Such programs are in general hard to solve. However, sometimes we are able to derive a polyhedral description of the problem: Suppose that by relaxing the integrality constraints of the IP formulation of the optimization we obtain a linear program whose feasible region is an integral polyhedron. We can then solve the optimization problem in polynomial time simply by computing an optimal solution to the LP, e.g., by using Khachiyan's ellipsoid method. An important question in this context is therefore whether a resulting polyhedron is integral. Proving integrality of polyhedra is often a difficult task. We next consider a technique that facilitates showing that a polyhedron is integral.

Subsequently, we concentrate on rational polyhedra, i.e., polyhedra that are defined by rational linear inequalities. A rational polyhedron P is called *integral* if every non-empty face F of P contains an integral vector. Clearly, it suffices to show that every minimal face of P is integral because every face contains a minimal face. Note that if P is pointed then this is equivalent to showing that every vertex of P is integral.

Lemma 8.4. *Let $B \in \mathbb{Z}^{m\times m}$ be an invertible matrix. Then $B^{-1}b$ is integral for every integral vector b if and only if* $\det(B) = \pm 1$.

Proof. Suppose $\det(B) = \pm 1$. By Cramer's Rule, B^{-1} is integral, which implies that $B^{-1}b$ is integral for every integral b. Conversely, suppose $B^{-1}b$ is integral for every integral vector b. Then also $B^{-1}e_i$ is integral for all $i \in \{1,\dots,m\}$, where e_i is the ith unit vector. As a consequence, B^{-1} is integral. Thus, $\det(B)$ and $\det(B^{-1})$ are both integers. This in combination with $\det(B)\det(B^{-1}) = 1$ implies that $\det(B) = \pm 1$. $\square$

A matrix A is *totally unimodular* if every square submatrix of A has determinant 0, 1 or -1. Clearly, every entry in a totally unimodular matrix is 0, 1 or -1.

Theorem 8.6. *Let $A \in \mathbb{Z}^{m \times n}$ be a totally unimodular matrix and let $b \in \mathbb{Z}^m$. Then the polyhedron $P = \{x \mid Ax \leq b\}$ is integral.*

Proof. Let F be a minimal face of P. Then $F = \{x \mid A^\circ x = b^\circ\}$ for some subsystem $A^\circ x \leq b^\circ$ of $Ax \leq b$ and A° has full row rank. By reordering the columns of A° we may write A° as $(B\ N)$, where B is a basis of A°. Because A is totally unimodular and B is a basis, $\det(B) = \pm 1$. By Lemma 8.4, it follows that $x = \left(\begin{smallmatrix} B^{-1}b^\circ \\ 0 \end{smallmatrix}\right)$ is an integral vector in F. □

Let $A \in \mathbb{R}^{m \times n}$ be a matrix of full row rank. A *basis* B of A is a non-singular submatrix of A of order m. A matrix A of full row rank is *unimodular* if A is integral and each basis B of A has $\det(B) = \pm 1$.

Theorem 8.7. *Let $A \in \mathbb{Z}^{m \times n}$ be a matrix of full row rank. Then the polyhedron $P = \{x \mid Ax = b,\ x \geq 0\}$ is integral for every vector $b \in \mathbb{Z}^m$ if and only if A is unimodular.*

Proof. Suppose A is unimodular. Let $b \in \mathbb{Z}^m$ and let x be a vertex of P. (Note that the non-negativity constraint ensures that P has vertices.) Then there are n linearly independent constraints satisfied by x with equality. The columns of A corresponding to non-zero entries of x are linearly independent. We can extend these columns to a basis B of A. Note that $\det(B) = \pm 1$ because A is unimodular. Then x restricted to the coordinates corresponding to B is $B^{-1}b$, which is integral by Lemma 8.4. The remaining entries of x are zero. Thus, x is integral.

Assume that P is integral for every integer vector b. Let B be a basis of A. We need to show that $\det(B) = \pm 1$. By Lemma 8.4, it suffices to show that $B^{-1}v$ is integral for every integral vector v. Let v be an integral vector. Let y be an integral vector such that $z = y + B^{-1}v \geq 0$ and let $b = Bz = B(y + B^{-1}v) = By + v$. Note that b is integral. By adding zero components to z, we obtain a vector $z' \in \mathbb{Z}^n$ such that $Az' = Bz = b$. Then z' is a vertex of $\{x \mid Ax = b,\ x \geq 0\}$, because z' is in the polyhedron and satisfies n linearly independent constraints with equality: the m equations $Ax = b$ and the $n - m$ equations $x_i = 0$ for the columns i outside of B. So z' is integral and thus $B^{-1}v = z - y$ is integral. □

Theorem 8.8. *Let $A \in \mathbb{Z}^{m \times n}$. The polyhedron $P = \{x \mid Ax \leq b,\ x \geq 0\}$ is integral for every vector $b \in \mathbb{Z}^m$ if and only if A is totally unimodular.*

Proof. It is not hard to show that A is totally unimodular if and only if $(A\ I)$ is unimodular, where I is the $m \times m$ identity matrix. By Theorem 8.7, $(A\ I)$ is unimodular if and only if $P' = \{z \mid (A\ I)z = b,\ z \geq 0\}$ is integral for every $b \in \mathbb{Z}^m$. The latter is equivalent to $P = \{x \mid Ax \leq b,\ x \geq 0\}$ being integral for every $b \in \mathbb{Z}^m$. □

8.5 Example Applications

Theorem 8.9. *A matrix A is totally unimodular if*

1. *each entry is 0, 1 or -1;*

2. *each column contains at most two non-zeros;*
3. *the set N of row indices of A can be partitioned into $N_1 \cup N_2$ so that in each column j with two non-zeros we have $\sum_{i \in N_1} a_{i,j} = \sum_{i \in N_2} a_{i,j}$.*

Proof. Suppose that A is not totally unimodular. Let t be the smallest integer such that B is a $t \times t$ square submatrix of A with $\det(B) \notin \{-1,0,1\}$. Suppose B has a column with a single non-zero entry, say $b_{k,j}$. By expanding the determinant along row j (Laplace expansion), we obtain

$$\det(B) = \sum_{i=1}^{t} (-1)^{i+j} b_{i,j} M_{i,j} = (-1)^{k+j} b_{k,j} M_{k,j}$$

where $M_{i,j}$ is the *minor* defined as the determinant of the submatrix obtained by removing row i and column j from B. By (1), $b_{k,j} \in \{-1,0,1\}$ and because $\det(B) \notin \{-1,0,1\}$, $M_{k,j} \notin \{-1,0,1\}$, which is a contradiction to the choice of B. By (2), every column of B must therefore contain exactly two non-zero entries. By (3), adding up the rows of B (N_1 with positive sign, N_2 with negative sign) yields the zero vector. The row vectors are therefore linearly dependent and thus $\det(B) = 0$, which is a contradiction. ☐

The *incidence matrix* $A = (a_{u,e})$ of an undirected graph $G = (V,E)$ is an $n \times m$ matrix ($n = |V|$ and $m = |E|$) such that for every $u \in V$ and $e \in E$:

$$a_{u,e} = \begin{cases} 1 & \text{if } e = (u,v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

The *incidence matrix* $A = (a_{u,e})$ of a directed graph $G = (V,E)$ is an $n \times m$ matrix such that for every $u \in V$ and $e \in E$:

$$a_{u,e} = \begin{cases} 1 & \text{if } e = (u,v) \in E \\ -1 & \text{if } e = (v,u) \in E \\ 0 & \text{otherwise.} \end{cases}$$

The following corollary follows immediately from Theorem 8.9.

Corollary 8.1. *If A is an incidence matrix of an undirected bipartite graph or an incidence matrix of a directed graph, then A is totally unimodular.*

Proof. The proof follows from Theorem 8.9 by choosing $N_1 = V_0$ and $N_2 = V_1$ in the bipartite case (where $V = V_0 \cup V_1$) and $N_1 = V$ and $N_2 = \emptyset$ in the directed case. ☐

Recall that a *node cover* of an undirected graph $G = (V,E)$ is a subset $C \subseteq V$ such that for every edge $e = (u,v)$ at least one of the endpoints is in C, i.e., $\{u,v\} \cap C \neq \emptyset$. Let $\nu(G)$ be the size of a maximum matching of G and let $\tau(G)$ be the size of a minimum node cover of G.

The size of a maximum matching can be formulated as an integer program:

$$\begin{array}{rlrcll} \nu(G) = & \text{maximize} & \sum_{e\in E} x_e & & & \\ & \text{subject to} & \sum_{e=(u,v)\in E} x_e & \le & 1 & \forall u \in V \\ & & x_e & \in & \{0,1\} & \forall e \in E \end{array}$$

Equivalently, we can write this IP in a more compact way:

$$\nu(G) = \{\mathbf{1}^T x \mid Ax \le 1,\ x \ge 0,\ x \in \mathbb{Z}^m\}, \tag{13}$$

where $A \in \mathbb{Z}^{n\times m}$ is the incidence matrix of G with $n = |V|$ and $m = |E|$.

Similarly, the size of a minimum node cover can be expressed as

$$\begin{array}{rlrcll} \tau(G) = & \text{minimize} & \sum_{u\in V} y_u & & & \\ & \text{subject to} & y_u + y_v & \ge & 1 & \forall (u,v) \in E \\ & & y_u & \in & \{0,1\} & \forall u \in V \end{array}$$

or, equivalently,

$$\tau(G) = \{y^T\mathbf{1} \mid A^T y \ge 1,\ y \ge 0,\ y \in \mathbb{Z}^n\} \tag{14}$$

Theorem 8.10. *Let $G = (V,E)$ be a bipartite graph. The size of a maximum matching of G is equal to the size of a minimum node cover of G, i.e., $\nu(G) = \tau(G)$.*

Proof. Let $A \in \mathbb{Z}^{n\times m}$ be the incidence matrix of G with $n = |V|$ and $m = |E|$. As observed above, we can express $\nu(G)$ and $\tau(G)$ by the two integer linear programs (13) and (14). Consider the respective LP relaxations of (13) and (14):

$$\nu'(G) = \{\mathbf{1}^T x \mid Ax \le 1,\ x \ge 0\} \tag{15}$$

$$\tau'(G) = \{y^T\mathbf{1} \mid A^T y \ge 1,\ y \ge 0\} \tag{16}$$

Note that both LPs are feasible. Because A is totally unimodular, both LPs have integral optimal solutions and thus $\nu(G) = \nu'(G)$ and $\tau(G) = \tau'(G)$. Finally, observe that (16) is the dual of (15). By strong duality, $\nu'(G) = \tau'(G)$, which proves the claim. $\square$

A matrix A is called an *interval matrix* if every entry of A is either 0 or 1 and the the 1's of each row appear consecutively (without interfering zeros).

Theorem 8.11. *Each interval matrix A is totally unimodular.*

Proof. Let B be a $t \times t$ submatrix of A. Define a $t \times t$ matrix N as follows:

$$N = \begin{pmatrix} 1 & -1 & 0 & \dots & 0 & 0 \\ 0 & 1 & -1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -1 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

Note that $\det(N) = 1$. Consider NB^T. Then NB^T is a submatrix of an incident matrix of some directed graph. (Think about it!) Therefore, NB^T is totally unimodular. We conclude

$$\det(B) = \det(N)\det(B^T) = \det(NB^T) \in \{-1, 0, 1\}.$$

□

References

The presentation of the material in this section is based on [2, Chapter 6].

9. Complexity Theory

9.1 Introduction

The problems that we have considered in this course so far are all solvable *efficiently*. This means that we were always able to design an algorithm for the respective optimization problem that solves every instance in time that is polynomially bounded in the size of the instance. For example, we have seen that every instance of the *minimum spanning tree problem* with n vertices and m edges can be solved in time $O(m + n \log n)$. Unfortunately, for many natural and fundamental optimization problems efficient algorithms are not known to exist. A well-known example of such a problem is the *traveling salesman problem*.

Traveling Salesman Problem (TSP):

Given:	An undirected graph $G = (V,E)$ and non-negative distances $d : E \to \mathbb{Z}^+$ on the edges.
Goal:	Find a tour that visits every vertex of G exactly once (starting and ending in the same vertex) and has minimum total length.

Despite 50 years of intensive research, no efficient algorithm has been found for the TSP problem. On the other hand, researchers have also not been able to disprove the existence of such algorithms. Roughly speaking, complexity theory aims to answer the question if the research community has been too stupid or unlucky to find efficient algorithms for optimization problems such as the TSP problem, or that these problem are in fact intrinsically more difficult than other problems. It provides a mathematical framework to separate problems that are computationally hard to solve from the ones that are efficiently solvable.

In complexity theory one usually considers decision problems instead of optimization problems.

Definition 9.1. A *decision problem* Π is given by a set of instances $\mathcal{I}$. Each instance $I \in \mathcal{I}$ specifies

- a set $\mathcal{F}$ of feasible solutions for I;
- a cost function $c : \mathcal{F} \to \mathbb{Z}$;
- an integer K.

Given an instance $I = (\mathcal{F}, c, K) \in \mathcal{I}$, the goal is to decide whether there exists a feasible solution $S \in \mathcal{F}$ whose cost $c(S)$ is at most K. If there is such a solution, we say that I is a "yes-instance"; otherwise, I is a "no-instance".

Example 9.1. The decision problem of the TSP problem is to determine whether for a given instance $I = (G, d, K) \in \mathcal{I}$ there exists a tour in G of total length at most K.

Many decision problems can naturally be described without the need of introducing a cost function c and a parameter K. Some examples are the following ones.

Prime:

Given:	A natural number n.
Goal:	Determine whether n is a prime.

Graph Connectedness:

Given:	An undirected graph $G = (V, E)$.
Goal:	Determine whether G is connected.

Hamiltonian Cycle:

Given:	An undirected graph $G = (V, E)$.
Goal:	Determine whether G has a Hamiltonian cycle.

Subsequently, we will mostly focus on decision problems. For notational convenience we will use the same naming as for the respective optimization counterparts (e.g., *TSP* will refer to the decision problem of TSP); no confusion should arise from this.

Recall that an algorithm ALG for a problem Π is said to be *efficient* if it solves every instance $I \in \mathcal{I}$ of Π in time that is bounded by a polynomial function of the size of I. It is not hard to see that the decision version of an optimization problem is easier than the optimization problem itself. But in most cases, an efficient algorithm for solving the decision version can also be turned into an efficient algorithm for the optimization problem (e.g., by using binary search on the possible optimal value).

9.2 Complexity Classes *P* and *NP*

Intuitively, the complexity classes *P* and *NP* refer to decision problems that can be *solved* efficiently and those for which yes-instances can be *verified* efficiently, respectively. If we insisted on formal correctness here, we would define these classes in terms of a specific computer model called *Turing machines*. However, this is beyond the scope of this course and we therefore take the freedom to introduce these classes using a more high-level (but essentially equivalent) point of view.

We define the complexity class *P* (which stands for *polynomial-time*).

Definition 9.2. A decision problem Π belongs to the complexity class P if there exists an algorithm that for every instance $I \in \mathcal{I}$ determines in polynomial time whether I is a yes-instance or a no-instance.

All problems that we have treated so far in this course belong to this class. But also the *linear programming problem (LP)* belongs to this class, even though the simplex algorithm is not a polynomial-time algorithm for LP (the interested reader is referred to Section 8.6 in [6]). The simplex algorithm works almost always very fast in practice for any LP of whatever size, but as mentioned before the running time of an algorithm is determined by its worst-case running time. For most pivoting rules devised for the simplex algorithm, there have been constructed instances on which the algorithm has to visit an exponential number of basic feasible solutions in order to arrive at an optimal one. A polynomial-time algorithm for LP is the *ellipsoid method* (the interested reader is referred to Section 8.7 in

[6]). This algorithm is an example where the time bound is polynomial in the logarithm of the largest coefficient in the instance next to the number of variables and number of restrictions. One of the most interesting open research questions in Operations Research is whether there exists an algorithm for LP whose running time is a polynomial in the number of variables and the number of restrictions only.

Next we define the complexity class *NP*. *NP* does not stand for "non-polynomial-time" as one might guess, but for "non-deterministic polynomial-time" because this class is formally defined in terms of non-deterministic Turing machines.

Given a yes-instance $I \in \mathcal{I}$ of a decision problem Π, we say that S is a *certificate* for I if $S \in \mathcal{F}$ and $c(S) \leq K$. Note that every yes-instance I must have a certificate. The specialty of a problem in *NP* is that yes-instances admit certificates that can be verified in polynomial time.

Definition 9.3. A decision problem Π belongs to the complexity class *NP* if every yes-instance $I \in \mathcal{I}$ admits a certificate whose validity can be verified in polynomial time.

Note that the polynomial-time verifiability of S implies that the size of S must be polynomially bounded in $|I|$ (because we need to look at S to verify its validity). That is, the definition above also states that yes-instances of problems in *NP* have *short*, i.e., polynomially bounded, certificates.

We consider some examples:

Example 9.2. The Hamiltonian cycle problem is in *NP*: A certificate for a yes-instance corresponds to a set of edges $S \subseteq E$. One can verify in $O(n)$ time whether S constitutes a cycle in G that visits all vertices exactly once.

Example 9.3. Consider the decision variant of the linear programming problem:

Linear Programming Problem (LP):

Given: A set $\mathcal{F}$ of feasible solutions $x = (x_1, \ldots, x_n)$ defined by m linear constraints

$$\mathcal{F} = \left\{ (x_1, \ldots, x_n) \in \mathbb{R}^n_{\geq 0} \ : \ \sum_{i=1}^{n} a_{ij} x_i \geq b_j \text{ for every } j = 1, \ldots, m \right\}$$

together with an objective function $c(x) = \sum_{i=1}^{n} c_i x_i$ and a parameter K.

Goal: Determine whether there exists a feasible solution $x \in \mathcal{F}$ that satisfies $c(x) \leq K$.

LP is in *NP*: A certificate for a yes-instance corresponds to a solution $x = (x_1, \ldots, x_n)$. We need $O(n)$ time to verify each of the m constraints and $O(n)$ time to compute the objective function value $c(x)$. The total time needed to check whether $x \in \mathcal{F}$ and $c(x) \leq K$ is thus $O(nm)$.

9.3 Polynomial-time Reductions and *NP*-completeness

After thinking for a little while, we conclude that $P \subseteq NP$. Several decades of intensive research seem to suggest that there are problems in *NP* that are intrinsically more difficult than the ones in *P* and thus $P \neq NP$: Despite the many research efforts, no polynomial-time algorithms have been found for problems in *NP* such as *TSP*, *Hamiltonian cycle*, *Steiner tree*, etc. On the other hand, all attempts to show that these problems are in fact harder than the ones in *P* failed as well. The question whether $P \neq NP$ is one of the biggest mysteries in mathematics to date and constitutes one of the seven millennium-prize problems; see http://www.claymath.org/millennium for more information.

Complexity theory attempts to give theoretical evidence to the conjecture that $P \neq NP$. It defines within the complexity class *NP* a subclass of most difficult problems, the so-called *NP-complete* problems. This subclass is defined in such a way that if for *any* of the *NP*-complete problems there will ever be found a polynomial-time algorithm then this implies that for *every* problem in *NP* there exists a polynomial-time algorithm, and thus $P = NP$. The definition of this class crucially relies on the notion of *polynomial-time reductions*:

Definition 9.4. A *polynomial-time reduction* from a decision problem Π_1 to a decision problem Π_2 is a function $\varphi : \mathcal{I}_1 \to \mathcal{I}_2$ that maps every instance $I_1 \in \mathcal{I}_1$ of Π_1 to an instance $I_2 = \varphi(I_1) \in \mathcal{I}_2$ of Π_2 such that:

1. the mapping can be done in time that is polynomially bounded in the size of I_1;
2. I_1 is a yes-instance of Π_1 if and only if I_2 is a yes-instance of Π_2.

If there exist such a polynomial-time reduction from Π_1 to Π_2 then we say that Π_1 can be *reduced to* Π_2, and we will write $\Pi_1 \preceq \Pi_2$.

Lets think about some consequences of the above definition in terms of polynomial-time computability. Suppose $\Pi_1 \preceq \Pi_2$. Then Π_2 is more difficult to solve than Π_1 (which also justifies the use of the symbol $\preceq$). To see this, note that every polynomial-time algorithm ALG_2 for Π_2 can be used to derive a polynomial-time algorithm ALG_1 for Π_1 as follows:

1. Transform the instance I_1 of Π_1 to a corresponding instance $I_2 = \varphi(I_1)$ of Π_2.
2. Run ALG_2 on I_2 and report that I_1 is a yes-instance if and only if ALG_2 concluded that I_2 is a yes-instance.

By the first property of Definition 9.4, the transformation in Step 1 above takes time polynomial in the size $n_1 = |I_1|$ of I_1. As a consequence, the size $n_2 = |I_2|$ of I_2 is polynomially bounded in n_1. (Think about it!) In Step 2, ALG_2 solves I_2 in time polynomial in the size n_2 of I_2, which is polynomial in the size n_1.[3] The overall time needed by ALG_1 to output a solution for I_1 is thus bounded by a polynomial in n_1. Note that the second property of Definition 9.4 ensures that ALG_1 correctly identifies whether I_1 is a yes-instance or not.

Observe the existence of a polynomial-time algorithm for Π_1 has in general no implications for the existence of a polynomial-time algorithm for Π_2, even if we assume that we can compute the inverse of φ efficiently. The reason for that is that φ is not necessarily a one-to-one mapping and may thus map the instances of Π_1 to a subset of the instances of

[3]Observe that we exploit here that if p_1, p_2 are polynomial functions in n then $p_2(p_1(n))$ is a polynomial function in n.

Π_2 which correspond to easy instances of Π_2. Thus, being able to efficiently solve every instance of Π_1 reveals nothing about the problem of solving Π_2.

It is not hard to show that polynomial-time reductions are transitive:

Lemma 9.1. *If* $\Pi_1 \preceq \Pi_2$ *and* $\Pi_2 \preceq \Pi_3$ *then* $\Pi_1 \preceq \Pi_3$.

We can now define the class of *NP-complete* problems.

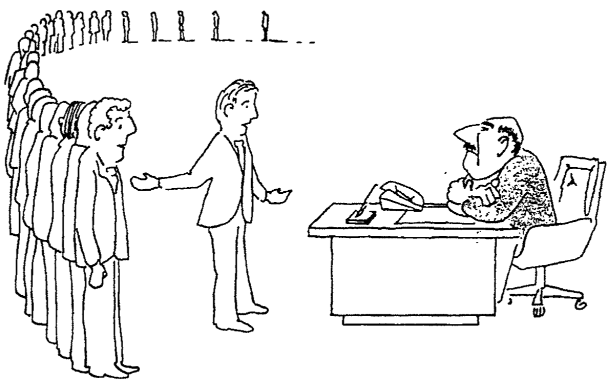
Definition 9.5. A decision problem Π is *NP-complete* if

1. Π belongs to *NP*;
2. every problem in *NP* is polynomial-time reducible to Π.

Intuitively, the above definition states that an *NP*-complete problem is as difficult as any other problem in *NP*. The above definition may not seem very helpful at first sight: How do we prove that *every* problem in *NP* is polynomial-time reducible to the problem Π we are interested in? Lets assume for the time being that there are some problems that are known to be *NP*-complete. In order to prove *NP*-completeness of Π it is then sufficient to show that Π is in *NP* and that *some NP*-complete problem is polynomial-time reducible to Π. (Think about it!) That is, showing *NP*-completeness of a problem becomes much easier now because we "just" need to find an appropriate *NP*-complete problem that can be reduced to it. Nevertheless, we remark that the reductions of many *NP*-completeness proofs are highly non-trivial and often require a deep understanding of the structural properties of the problem.

The class *NP* has a very precise definition in terms of executions of non-deterministic Turning machines (which we skipped and persist in skipping), which enabled Steven Cook in 1974 to prove that any such execution can be reduced to an instance of a famous problem in Boolean logic called the *satisfiability problem (SAT)* (stated below). Thus, Cook provided us with a problem that is *NP*-complete. Starting from this, many other problems were proven to be *NP*-complete.

In a way, proving that a problem is *NP*-complete is a beautiful way of stating:[4]



"I can't find an efficient algorithm, but neither can all these famous people."

[4]The illustration is taken from the book [4], which is an excellent book on the complexity of algorithms containing many fundamental *NP*-completeness proofs.

$$\begin{tikzcd}
 & \text{SAT} \arrow[d] & \\
 & \text{3-SAT} \arrow[dl] \arrow[dr, "(\star)"] & \\
\textit{Hamiltonian cycle} \arrow[d, "(\star)"'] & & \textit{vertex cover} \arrow[d, "(\star)"] \\
\textit{TSP} & & \textit{clique} \arrow[d, "(\star)"] \\
 & & \textit{independent set}
\end{tikzcd}$$

Figure 16: Reductions to proof *NP*-completeness of the example problems considered here; proofs are given for the ones marked with $(\star)$.

9.4 Examples of *NP*-completeness Proofs

We introduce some more problems and show that they are *NP*-complete. However, most of the reductions are technically involved and will be omitted here because the intention is to gain some basic understanding of the proof methodology rather than diving into technical details.

We first introduce the *satisfiability problem* for which Cook established *NP*-completeness. The basic ingredients are *variables*. A variable reflects an expression which can be TRUE or FALSE. For example,

$$x_1 = \textit{Koen is taller than Michael} \quad \text{and} \quad x_2 = \textit{Soup is always eaten with a fork}.$$

A variable can also occur negated. For example, we write $\neg x_1$ to express that *Koen is not taller than Michael*. A *literal* refers to a negated or unnegated variable. We compose more complicated expressions, called *clauses*, from literals. An example of a clause is

$$C_1 = (x_1 \vee \neg x_2 \vee x_3 \vee x_4).$$

The interpretation is that clause C_1 is TRUE if and only if x_1 is TRUE or (indicated by $\vee$) not-x_2 is TRUE or x_3 is TRUE or x_4 is TRUE. That is, a clause is TRUE if at least one of its literals is TRUE. An instance of the SAT problem is a *Boolean formula F* in so-called *conjunctive normal form (CNF)*:

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where each C_i is a clause. F is TRUE if C_1 is TRUE and (indicated by $\wedge$) C_2 is TRUE and ... and C_m is TRUE, i.e., if *all* its clauses are TRUE.

Satisfiability Problem (SAT):

Given:	A Boolean formula F in CNF.
Goal:	Determine whether there is a TRUE/FALSE-assignment to the variables such that F is TRUE.

Theorem 9.1. *SAT is NP-complete.*

The proof is involved and skipped here (the interested reader is referred to Section 15.5 in [6].)

The following restriction of *satisfiability* is also *NP*-complete.

3-Satisfiability Problem (3-SAT):

Given:	A Boolean formula F in CNF with each clause consisting of 3 literals.
Goal:	Determine whether there is a TRUE/FALSE-assignment to the variables such that F is TRUE.

Theorem 9.2. *3-SAT is NP-complete.*

The proof reduces *SAT* to *3-SAT*. We refer the reader to [6, Theorem 15.2].

We introduce some more problems and give some examples of *NP*-completeness proofs.

Let $G = (V,E)$ be an undirected graph. We need the following definitions: A *clique* of G is a subset V' of the vertices that induces a complete subgraph, i.e., for every two vertices $u, v \in V'$, $(u,v) \in E$. An *independent set* of G is a subset V' of vertices such that no two of them are incident to the same edge, i.e., for every two vertices $u, v \in V'$, $(u,v) \notin E$. A *vertex cover* of G is a subset V' of vertices such that every edge has at least one of its two incident vertices in V', i.e., for every edge $(u,v) \in E$, $\{u,v\} \cap V' \neq \emptyset$.

Clique:

Given:	An undirected graph $G = (V,E)$ and an integer K.
Goal:	Determine whether G contains a clique of size at least K.

Independent Set:

Given:	An undirected graph $G = (V,E)$ and an integer K.
Goal:	Determine whether G contains an independent set of size at least K.

Vertex Cover:

Given:	An undirected graph $G = (V,E)$ and an integer K.
Goal:	Determine whether G contains a vertex cover of size at most K.

Theorem 9.3. *Vertex cover is NP-complete.*

Proof. We first argue that *vertex cover* is in *NP*. A certificate of a yes-instance is a subset $V' \subseteq V$ of vertices with $|V'| \leq K$ that forms a vertex cover of $G = (V,E)$. This can be verified in time at most $O(n+m)$ by checking whether each edge $(u,v) \in E$ has at least one of its incident vertices in V'.

In order to prove that *vertex cover* is *NP*-complete, we will show that *3-SAT* $\preceq$ *vertex cover*. Note that this is sufficient because *3-SAT* is *NP*-complete.

We transform an instance of *3-SAT* to an instance of *vertex cover* as follows: Consider a Boolean formula F in CNF with each clause having three literals. Let n and m denote the

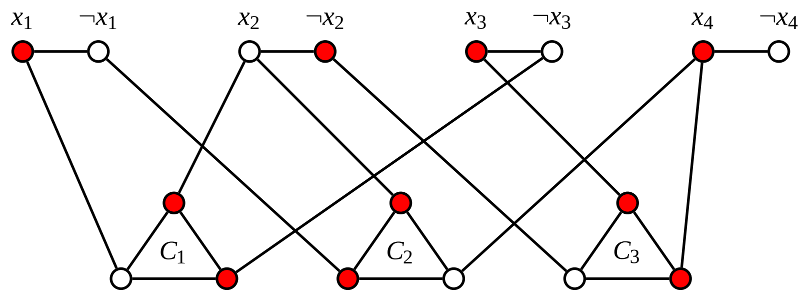


Figure 17: Illustration of the construction in the proof of Theorem 9.3 for the formula $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_4)$. The red vertices constitute a vertex cover of size $K = n + 2m = 10$.

number of variables and clauses of F, respectively. We create a *variable-gadget* for each variable x consisting of two vertices x and $\neg x$ that are connected by an edge. Moreover, we create a *clause-gadget* for each clause $C = (l_1 \vee l_2 \vee l_3)$ consisting of three vertices l_1, l_2, l_3 that are connected by a triangle. Finally, we connect each vertex representing a literal in a clause-gadget to the corresponding vertex representing the same literal in the variable-gadget. Let $G = (V,E)$ be the resulting graph; see Figure 17 for an example. Note that this transformation can be done in polynomial time.

We show that F is satisfiable if and only if G has a vertex cover of size at most $K = n+2m$. First note that every assignment satisfying F can be turned into a vertex cover of size K: For each variable-gadget we pick the vertex that corresponds to the literal which is TRUE. This covers all edges in the variable-gadgets and their respective connections to the clause-gadgets. For each clause-gadget we choose two additional vertices so as to ensure that all remaining edges are covered. The resulting vertex cover has size $K = n + 2m$ as claimed. Next suppose that we are given a vertex cover V' of G of size at most K. Note that every vertex cover has to pick at least one vertex for every variable-gadget and two vertices for each clause-gadget just to cover all edges inside these gadgets. Thus, V' contains exactly K vertices. The vertices in V' now naturally induce an assignment as described above that satisfies F. We conclude that yes-instances correspond under the above reduction, which completes the proof. □

Theorem 9.4. *Clique is NP-complete.*

Proof. We first argue that *clique* is in *NP*. A certificate for a yes-instance is a subset V' of vertices that forms a clique. To verify this, we just need to check that there is an edge between every pair of vertices in V'. This can be done in $O(n+m)$ time.

We prove that *vertex cover* $\preceq$ *clique* in order to establish *NP*-completeness of *clique*. We need the notion of a *complement graph* for this reduction. Given a graph $G = (V,E)$, the complement graph of G is defined as the graph $\overline{G} = (V, \overline{E})$ with $(u,v) \in \overline{E}$ if and only if $(u,v) \notin E$.

Given an instance $G = (V,E)$ with parameter K of *vertex cover*, we create the complement graph of G and let $\overline{G}$ with parameter $n - K$ be the respective instance of *clique*. Note that this mapping can be done in polynomial time by adding an edge (u,v) to $\overline{E}$ for every pair of vertices $u,v \in V$ with $(u,v) \notin E$. This takes at most $O(n^2)$ time.

It remains to show that yes-instances correspond. We claim that V' is a vertex cover in G if and only if $V \setminus V'$ is a clique in $\overline{G}$. V' is a vertex cover in G if and only if every edge $(u,v) \in E$ has not both its endpoints in $V \setminus V'$, or, equivalently, every edge $(u,v) \notin \overline{E}$ has not both its endpoints in $V \setminus V'$. The latter statement holds if and only if for every pair of vertices $u,v \in V \setminus V'$ there exists an edge $(u,v) \in \overline{E}$ in $\overline{G}$, which is equivalent to $V \setminus V'$ being a clique of $\overline{G}$. This proves the claim. We conclude that V' is a vertex cover of size K in G if and only if $V \setminus V'$ is a clique of size $n-K$ in $\overline{G}$. □

Theorem 9.5. *Independent set is NP-complete.*

Proof. We first argue that *independent set* is in *NP*. A certificate for a yes-instance is a subset V' of vertices that forms an independent set. To verify this, we just need to check that there is no edge between every pair of vertices in V'. This can be done in $O(n+m)$ time.

We prove that *clique* $\preceq$ *independent set* in order to establish *NP*-completeness of *independent set*. We need the notion of a *complement graph* for this reduction. Given a graph $G=(V,E)$, the complement graph of G is defined as the graph $\overline{G}=(V,\overline{E})$ with $(u,v) \in \overline{E}$ if and only if $(u,v) \notin E$.

Given an instance $G=(V,E)$ with parameter K of *clique*, we create the complement graph of G and let $\overline{G}$ with parameter K be the respective instance of *independent set*. Note that this mapping can be done in polynomial time by adding an edge (u,v) to $\overline{E}$ for every pair of vertices $u,v \in V$ with $(u,v) \notin E$. This takes at most $O(n^2)$ time.

It remains to show that yes-instances correspond. We claim that V' is a clique of G if and only if V' is an independent set of $\overline{G}$. Note that V' is a clique of G if and only if for each pair of vertices in V' there is an edge in E. The latter is true if and only if for each pair of vertices in V' there is no edge in $\overline{E}$, which is equivalent to V' being an independent set of $\overline{G}$. This proves the claim. We conclude that V' is a clique of size K in G if and only if V' is an independent set of size K in $\overline{G}$. □

Theorem 9.6. *Hamiltonian cycle is NP-complete.*

The proof follows by reducing *3-SAT* to *Hamiltonian cycle*. The reader is referred to [6, Theorem 15.6].

Theorem 9.7. *TSP is NP-complete.*

Proof. We argued before that *TSP* is in *NP*. The proof now follows trivially because *Hamiltonian cycle* is a special case of TSP: Given an instance $G=(V,E)$ of *Hamiltonian cycle* we construct an instance of *TSP* as follows: Let $G'=(V,E')$ be the complete graph on V and define $d_e = 1$ if $e \in E$ and $d_e = 2$ otherwise. Now a tour in G' of length at most $K=n$ relates to a Hamiltonian cycle in G and vice versa. □

The above proof actually shows that the restriction of *TSP* in which all distances are either 1 or 2 is *NP*-complete. Because *TSP* is in *NP* and it is a generalization of this problem, *NP*-completeness of *TSP* follows immediately. The same holds true for *satisfiability*: If we would not know that it is *NP*-complete but we would know that *3-SAT* is *NP*-complete, then *NP*-completeness of *SAT* followed automatically using the fact that *3-SAT* is a special case of *SAT*. While restrictions can create an easier subclass of problem instances, generalizations always create more difficult problems. This gives sometimes easy ways to show *NP*-completeness of problems.

We list some more *NP*-complete problems (without proof) that are often used in reductions.

2-Partition:

Given:	Integers $s_1, \dots, s_n$.
Goal:	Decide whether there is a set $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} s_i = \frac{1}{2} \sum_{i=1}^{n} s_i$.

3-Partition:

Given:	Rational numbers $s_1, \dots, s_{3n}$ with $\frac{1}{4} < s_i < \frac{1}{2}$ for every $i = 1, \dots, 3n$.
Goal:	Determine whether the set $\{1, \dots, 3n\}$ can be partitioned into n triplets $S_1, \dots, S_n$ such that $\sum_{i \in S_k} a_i = 1$ for every $k = 1, \dots, n$.

Set Cover.:

Given:	A universe $U = \{1, \dots, n\}$ of n elements, a family of m subsets $S_1, \dots, S_m \subseteq U$ and an integer K.
Goal:	Determine whether there is a selection of at most K subsets such that their union is U.

9.5 More on Complexity Theory

9.5.1 *NP*-hard Problems

Sometimes we may be unable to prove that a problem Π is in *NP* but nevertheless can show that all problems in *NP* are reducible to Π. According to Definition 9.5, Π does not qualify to be an *NP*-complete problem because it is not in *NP*. Yet, Π is as hard as any other problem in *NP* and thus probably a difficult problem. We call such problem *NP-hard*. An example of such a problem is the *Lth heaviest subset problem*:

Lth Heaviest Subset Problem:

Given:	Integers $w_1, \dots, w_n, L$ and a parameter K.
Goal:	Determine whether the weight of the Lth heaviest subset of $\{1, \dots, n\}$ is at least K. (Formally, determine whether there are L distinct subsets $S_1, \dots, S_L \subseteq \{1, \dots, n\}$ such that $w(S_i) = \sum_{j \in S_i} w_j \geq K$ for every $i = 1, \dots, L$.)

It can be proven that all problems in *NP* are polynomial-time reducible to the *Lth Heaviest Subset* problem (see [6, Theorem 16.8]). However, a proof that short certificates exist for yes-instance is non-existent. How else could we provide a certificate for a yes-instance other than explicitly listing L subsets that are heavier than K? (Note that this is not a short certificate because L can be exponential in n.)

9.5.2 Complexity Class co-*NP*

Another complexity class that is related to *NP* is the class *co-NP* (which stands for *complement of NP*). Here one considers complements of decision problems. As an example, the complement of *Hamiltonian cycle* reads as follows:

Hamilton Cycle Complement:

Given:	An undirected graph $G = (V,E)$.
Goal:	Determine whether G does *not* contain a Hamiltonian cycle.

There are no *short* certificates known for yes-instances of this problem.

Definition 9.6. A decision problem Π belongs to the class co-*NP* if and only if its complement belongs to the class *NP*. Said differently, a decision problem belongs to co-*NP* if every no-instance $I \in \mathcal{I}$ admits a certificate whose validity can be verified in polynomial time.

It is not hard to see that every problem in P also belongs to co-*NP*. Thus, $P \subseteq NP \cap \text{co-}NP$. Similar to the $P \neq NP$ conjecture, it is widely believed that $NP \neq \text{co-}NP$.

Theorem 9.8. *If the complement of an NP-complete problem is in NP, then NP = co-NP.*

Proof. Assume that the complement $\bar{\Pi}_2$ of an *NP*-complete problem Π_2 is is in *NP*. We will show that the complement $\bar{\Pi}_1$ of an *arbitrary* problem $\Pi_1 \in NP$ is also in *NP* thus showing that $NP = \text{co-}NP$.

Because Π_2 is *NP*-complete, we know that Π_1 is polynomial-time reducible to Π_2. Note that the reduction φ from Π_1 to Π_2 is also a polynomial-time reduction from $\bar{\Pi}_1$ to $\bar{\Pi}_2$. We can therefore exhibit a short certificate for every yes-instance $\bar{I}_1$ of $\bar{\Pi}_1$ as follows: We first transform $\bar{I}_1$ to $\bar{I}_2 = \varphi(\bar{I}_1)$ and then use the short certificate for the yes-instance $\bar{I}_2$ (which must exist because $\bar{\Pi}_2 \in NP$). We conclude that $\bar{\Pi}_1$ is in *NP* which finishes the proof. $\square$

Note that the above theorem also implies that if the complement of a problem in *NP* is also in *NP* then (unless $NP = \text{co-}NP$) this problem is not *NP*-complete. Said differently, a problem that belongs to $NP \cap \text{co-}NP$ is unlikely to be *NP*-complete. As an example, consider the linear programming problem *LP*. Using duality theory, it is not hard to see that $LP \in NP \cap \text{co-}NP$. Before *LP* was known to be polynomial-time solvable, it was in fact the above observation that gave strong evidence to the conjecture that $LP \in P$.

Exercise 9.1. *Show that* $LP \in NP \cap \text{co-}NP$.

9.5.3 Pseudo-polynomiality and Strong *NP*-completeness

Sometimes the running time of an algorithm is polynomial in the size of the instance *and* the largest number in the input. As an example, consider the *integer knapsack problem*.

Integer Knapsack Problem:

Given:	Integers $c_1, \dots, c_n$ and a parameter K.
Goal:	Determine whether there exist integers $x_1, \dots, x_n$ such that $\sum_{k=1}^{n} c_k x_k = K$.

This problem can be solved as follows: Create a directed graph $G = (V, A)$ with $K+1$ vertices $V = \{0, 1, \dots, K\}$ and $O(nK)$ arcs:

$$A = \{(i, j) \mid 0 \le i < j \le K \text{ and } j = i + c_k \text{ for some } k\}.$$

It is not hard to prove that an instance I of the *integer knapsack* problem is a yes-instance if and only if there exists a path from 0 to K in G. The latter problem can be solved in time $O(n+m) = O(nK)$. This running time is *not* polynomial in the size of I. To see this, recall that we defined the size $|I|$ of I to be the number of bits that are needed to represent I in binary. Making the (reasonable) assumption that $c_1, \dots, c_n < K$, the size of I is therefore at most $O(n \log K)$. The running time of the algorithm is therefore not polynomially bounded in general. However, if K is bounded by a polynomial function of n then the algorithm would be a polynomial-time algorithm. That is, depending on the application the above algorithm might indeed be considered to be reasonably efficient, despite the fact that the problem is *NP*-complete (which it is).

The above observation gives rise to the following definition. Given an instance I, let $\text{num}(I)$ refer to the largest integer appearing in I.

Definition 9.7. An algorithm ALG for a problem Π is *pseudo-polynomial* if it solves every instance $I \in \mathcal{I}$ of Π in time bounded by a polynomial function in $|I|$ and $\text{num}(I)$.

Problems that remain *NP*-complete even if the largest integer appearing in its description is bounded polynomially in the size of the instance is called *strongly NP-complete*.

Definition 9.8. A problem Π is *strongly NP*-complete if the restriction of Π to instances $I \in \mathcal{I}$ satisfying that $\text{num}(I)$ is polynomially bounded in $|I|$ is *NP*-complete.

Note that many problems that we showed to be *NP*-complete do not involve any numerical data that is larger than the input size itself. For example, all graph problems such as *Hamiltonian cycle*, *clique*, *independent set*, *vertex cover*, etc. satisfy $\text{num}(I) = O(n)$ and are therefore even strongly *NP*-complete by definition. In fact, also *TSP* is strongly *NP*-complete because we established *NP*-completeness even for instances with distances 1 or 2.

As the theorem below shows, we cannot expect to find a pseudo-polynomial algorithm for a strongly *NP*-complete problem (unless $P = NP$).

Theorem 9.9. *There does not exist a pseudo-polynomial algorithm for a strongly NP-*

complete problem, unless $P = NP$.

Proof. Let Π be a strongly *NP*-complete problem and suppose that ALG is a pseudo-polynomial algorithm for Π. Consider the restriction $\overline{\Pi}$ of Π to instances $I \in \mathcal{I}$ that satisfy that $\text{num}(I)$ is polynomially bounded in $|I|$. By Definition 9.8, $\overline{\Pi}$ is *NP*-complete. But ALG can solve every instance $\overline{I}$ of $\overline{\Pi}$ in time polynomial in $|\overline{I}|$ and $\text{num}(\overline{I})$, which is polynomial in $|\overline{I}|$. This is impossible unless $P = NP$. $\square$

9.5.4 Complexity Class *PSPACE*

The common criterion that we used to define the complexity classes *P* and *NP* was time: *P* refers to the set of problems that are solvable in polynomial time; *NP* contains all problems for which yes-instances can be verified in polynomial time. There are other complexity classes that focus on the criterion *space* instead: The complexity class *PSPACE* refers to the set of problems for which algorithms exist that only require a polynomial amount of space (in the size of the input).

Definition 9.9. A decision problem Π belongs to the complexity class *PSPACE* if there exists an algorithm that for every instance $I \in \mathcal{I}$ determines whether I is a yes-instance or a no-instance using space that is polynomially bounded in the size of I.

Clearly, every polynomial-time algorithm cannot consume more than polynomial space and thus $P \subseteq PSPACE$. However, even exponential-time algorithms are feasible as long as they only require polynomial space. We can use this observation to see that $NP \subseteq PSPACE$: Consider an arbitrary problem Π in *NP*. We know that every yes-instances of Π admits a short certificate. We can therefore generate *all* potential short certificates one after another and verify the validity of each one. If we encounter a valid certificate throughout this procedure then we report that the instance is a yes-instance; otherwise, we report that it is a no-instance. The algorithm may take exponential time because the number of certificates to be checked might be exponential. However, it can be implemented to use only polynomial space by deleting the previously generated certificate each time.

As a final remark: We actually just got a tiny glimpse of the many existing complexity classes. There is a whole “zoo” of complexity classes; see, for example, the wiki page http://qwiki.stanford.edu/index.php/Complexity_Zoo if you want to learn more about many other complexity classes and their relations.

References

The presentation of the material in this section is based on [6, Chapters 8, 15 & 16].

10. Approximation Algorithms

10.1 Introduction

As we have seen, many combinatorial optimization problems are *NP*-hard and thus there is very little hope that we will be able to develop efficient algorithms for these problems. Nevertheless, many of these problems are fundamental and solving them is of great importance. There are various ways to cope with these hardness results:

1. *Exponential Algorithms*: Certainly, using an algorithm whose running time is exponential in the worst case might not be too bad after all if we only insist on solving instances of small to moderate size.
2. *Approximation Algorithms*: Approximation algorithms are efficient algorithms that compute suboptimal solutions with a provable approximation guarantee. That is, here we insist on polynomial-time computation but relax the condition that the algorithm has to find an optimal solution by requiring that it computes a feasible solution that is "close" to optimal.
3. *Heuristics*: Any approach that solves the problem without a formal guarantee on the quality of the solution can be considered as a heuristic for the problem. Some heuristics provide very good solutions in practice. An example of such an approach is *local search*: Start with an arbitrary solution and perform local improvement steps until no further improvement is possible. Moreover, heuristics are often practically appealing because they are simple and thus easy to implement.

We give some more remarks:

Some algorithms might perform very well in practice even though their worst-case running time is exponential. The simplex algorithm solving the *linear programming* problem is an example of such an algorithm. Most real-world instances do not correspond to worst-case instances and thus "typically" the algorithms' performance in practice is rather good. In a way, the worst-case running time viewpoint is overly pessimistic in this situation.

A very successful approach to attack optimization problems originating from practical applications is to formulate the problem as an *integer linear programming (ILP)* problem and to solve the program by *ILP*-solvers such as CPLEX. Such solvers are nowadays very efficient and are capable to solve large instances. Constructing the right *ILP*-method for solving a given problem is a matter of smart engineering. Some *ILP*-problems can be solved by just running an ILP-solver; others can only be solved with the help of more sophisticated methods such as branch-and-bound, cutting-plane, column generation, etc. Especially rostering problems, like classes of universities or schedules of personnel in hospitals, are notorious for being extremely hard to solve, already for small sizes. Solving ILP-problems is an art that can be learned only in practice.

Here we will focus on approximation algorithms in order to cope with *NP*-hardness of problems. We give a formal definition of these algorithms first.

Definition 10.1. An algorithm ALG for a minimization problem Π is an *α-approximation algorithm* with $\alpha \geq 1$ if it computes for every instance $I \in \mathcal{I}$ in polynomial time a feasible solution $S \in \mathcal{F}$ whose cost $c(S)$ is at most α times the cost $\mathsf{OPT}(I)$ of an optimal solution for I, i.e., $c(S) \leq \alpha \cdot \mathsf{OPT}(I)$.

The definition is similar for maximization problems. Here it is more natural to assume that we want to maximize a weight (or profit) function $w : \mathcal{F} \to \mathbb{R}$ that maps every feasible solution $S \in \mathcal{F}$ of an instance $I \in \mathcal{I}$ to some real value.

Definition 10.2. An algorithm ALG for a maximization problem Π is an *α-approximation algorithm* with $\alpha \geq 1$ if it computes for every instance $I \in \mathcal{I}$ in polynomial time a feasible solution $S \in \mathcal{F}$ whose weight (or profit) $w(S)$ is at least $\frac{1}{\alpha}$ times the weight $\mathsf{OPT}(I)$ of an optimal (i.e., maximum weight) solution for I, i.e., $w(S) \geq \frac{1}{\alpha} \cdot \mathsf{OPT}(I)$.

Note that we would like to design approximation algorithms with the approximation ratio α being as small as possible. A lot of research in theoretical computer science and discrete mathematics is dedicated to the finding of "good" approximation algorithms for combinatorial optimization problems.

10.2 Approximation Algorithm for *Vertex Cover*

We start with an easy approximation algorithm for the *vertex cover* problem, which has been introduced before: Given a graph $G = (V, E)$, find a vertex cover $V' \subseteq V$ of smallest cardinality. Recall that we showed that the decision variant of *vertex cover* is *NP*-complete.

One of the major difficulties in the design of approximation algorithms is to come up with a good estimate for the optimal solution cost $\mathsf{OPT}(I)$. (We will omit I subsequently.) Recall that a matching M is a subset of the edges having the property that no two edges share a common endpoint. We call a matching M *maximum* if the cardinality of M is maximum; we call it *maximal* if it is *inclusion-wise* maximal, i.e., we cannot add another edge to M without rendering it infeasible. Note that a maximum matching is a maximal one but not vice versa.

Lemma 10.1. *Let $G = (V, E)$ be an undirected graph. If M is a matching of G then $OPT \geq |M|$.*

Proof. Consider an arbitrary vertex cover V' of G. Every matching edge $(u, v) \in M$ must be covered by at least one vertex in V', i.e., $\{u, v\} \cap V' \neq \emptyset$. Because the edges in M do not share any endpoints, we have $|V'| \geq |M|$. $\square$

We conclude that we can derive an easy 2-approximation algorithm for *vertex cover* as follows:

Theorem 10.1. *Algorithm 14 is a 2-approximation algorithm for vertex cover.*

Proof. Clearly, the running time of Algorithm 14 is polynomial because we can find a maximal matching in time at most $O(n + m)$. The algorithm outputs a feasible vertex

```
Input: Undirected graph $G = (V,E)$.
Output: Vertex cover $V' \subseteq V$.
1 Find a maximal matching $M$ of $G$.
2 Output the set $V'$ of matched vertices.
```

Algorithm 14: Approximation algorithm for *vertex cover*.

cover because of the maximality of M. To see this, suppose that the resulting set V' is not a vertex cover. Then there is an edge (u,v) with $u,v \notin V'$ and thus both u and v are unmatched in M. We can then add the edge (u,v) to M and obtain a feasible matching, which contradicts the maximality of M. Finally, observe that $|V'| = 2|M| \leq 2\mathsf{OPT}$ by Lemma 10.1. □

Note that it suffices to compute a maximal (not necessarily maximum) matching in Algorithm 14, which can be done in linear time $O(m)$.

An immediate question that comes to ones mind is whether the approximation ratio is best possible. This indeed involves two kinds of questions in general:

1. Is the approximation ratio α of the algorithm tight?
2. Is the approximation ratio α of the algorithm best possible for *vertex cover*?

The first question essentially asks whether the analysis of the approximation ratio is tight. This is usually answered by exhibiting an example instance for which the algorithm computes a solution whose cost is α times the optimal one. The second one asks for much more: Can one show that there is no approximation algorithm with approximation ratio $\alpha - \varepsilon$ for every $\varepsilon > 0$? Such an *inapproximability result* usually relies on some conjecture such as that $P \neq NP$.

Lets first argue that the approximation ratio of Algorithm 14 is indeed tight.

Example 10.1. Consider a complete bipartite graph with n vertices on each side. The above algorithm will pick all $2n$ vertices, while picking one side of the bipartition constitutes an optimal solution of cardinality n. The approximation ratio of 2 is therefore tight.

The answer to the second question is not clear, despite intensive research. The currently best known lower bound on the inapproximability of *vertex cover* is as follows (stated without proof).

Theorem 10.2. *Vertex cover cannot be approximated within a factor of* 1.3606*, unless* $P = NP$*.*

10.3 Approximation Algorithms for *TSP*

As introduced before, the *traveling salesman problem* asks for the computation of a shortest tour in a given graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$.

We first show the following inapproximability result.

Theorem 10.3. *For any polynomial-time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$.*

Proof. Suppose we have an algorithm ALG that approximates *TSP* within a factor $\alpha(n)$. We show that we can use ALG to decide in polynomial time whether a given graph has a Hamiltonian cycle or not, which is impossible unless $P = NP$.

Let $G = (V, E)$ be a given graph on n vertices. We extend G to a complete graph and assign each original edge a cost of 1 and every other edge a cost of $n\alpha(n)$. Run the $\alpha(n)$-approximation algorithm ALG on the resulting instance. We claim that G contains a Hamiltonian cycle if and only if the TSP tour computed by ALG has cost less than or equal to $n\alpha(n)$.

Suppose G has a Hamiltonian cycle. Then the optimal TSP tour in the extended graph has cost n. The approximate TSP tour computed by ALG must therefore have cost less than or equal to $n\alpha(n)$. Suppose G does not contain a Hamiltonian cycle. Then every feasible TSP tour in the extended graph must use at least one edge of cost $n\alpha(n)$, i.e., the cost of the tour is greater than $n\alpha(n)$ (assuming that G has at least $n \geq 2$ vertices). Thus, the cost of the approximate TSP tour computed by ALG is greater than $n\alpha(n)$. The claim follows. ☐

The above inapproximability result is extremely bad news. The situation changes if we consider the *metric TSP* problem.

Metric Traveling Salesman Problem (Metric TSP):

Given:	An undirected complete graph $G = (V, E)$ with non-negative costs $c : E \to \mathbb{R}^+$ satisfying the *triangle inequality*, i.e., for every $u, v, w \in V$, $c_{uw} \leq c_{uv} + c_{vw}$.
Goal:	Compute a tour in G that minimizes the total cost.

The *metric TSP* problem remains *NP*-complete: Recall that we showed that the *TSP* problem is *NP*-complete by reducing *Hamiltonian Cycle* to this problem. The reduction only used edge costs 1 and 2. Note that such edge costs always constitute a metric. Thus, the same proof shows that *metric TSP* is *NP*-complete.

We next derive two constant factor approximation algorithms for this problem.

Given a subset $Q \subseteq E$ of the edges, we define $c(Q)$ as the total cost of all edges in Q, i.e., $c(Q) = \sum_{e \in Q} c_e$.

The following lemma establishes a lower bound on the optimal cost:

Lemma 10.2. *Let T be a minimum spanning tree of G. Then $\mathsf{OPT} \geq c(T)$.*

Proof. Consider an optimal *TSP* tour and remove an arbitrary edge from this tour. We obtain a spanning tree of G whose cost is at most OPT. The cost of T is thus at most OPT. ☐

This lemma leads to the following idea:[5]

```
Input: Complete graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$
       satisfying the triangle inequality.
Output: TSP tour of $G$.
1 Compute a minimum spanning tree $T$ of $G$.
2 Double all edges of $T$ to obtain a Eulerian graph $G'$.
3 Extract a Eulerian tour $C'$ from $G'$.
4 Traverse $C'$ and short-cut previously visited vertices.
5 Output the resulting tour $C$.
```

Algorithm 15: Approximation algorithm for *metric TSP*.

Theorem 10.4. *Algorithm 15 is a 2-approximation algorithm for metric TSP.*

Proof. Note that the algorithm has polynomial running time. Also, the returned tour is a TSP tour by construction. Because edge costs satisfy the triangle inequality, the tour C resulting from short-cutting the Eulerian tour C' in Algorithm 15 has cost at most $2c(T)$, where T is the minimum spanning tree computed in Step 1. By Lemma 10.2, the cost of C is thus at most $2\mathsf{OPT}$. $\square$

We can actually derive a better approximation algorithm by refining the idea of Algorithm 15. Note that the reason for doubling the edges of a minimum spanning tree T was that we would like to obtain a Eulerian graph from which we can then extract a Eulerian tour. Are there better ways to construct a Eulerian graph starting with a minimum spanning tree T? Certainly, we only have to take care of the odd degree vertices, say V', of T. Note that in a tree there must be an even number of odd degree vertices.

So one way of making these odd degree vertices become even degree vertices is to add the edges of a perfect matching on V' to T. Intuitively, we would like to keep the total cost of the augmented tree small and thus compute a minimum cost perfect matching. As the following lemma shows, the cost of this matching can be related to the optimal cost.

Lemma 10.3. *Let $V' \subseteq V$ be a subset containing an even number of vertices. Let M be a minimum cost perfect matching on V'. Then $OPT \geq 2c(M)$.*

Proof. Consider an optimal *TSP* tour C of length OPT. Traverse this tour and short-cut all vertices in $V \setminus V'$. Because of the triangle inequality, the resulting tour C' on V' has length at most OPT. C' can be seen as the union of two perfect matchings on V'. The cheaper matching of these two must have cost at most $\frac{1}{2}\mathsf{OPT}$. We conclude that a minimum cost perfect matching M on V' has cost at most $\frac{1}{2}\mathsf{OPT}$. $\square$

We combine the above observations in the following algorithm, which is also known as *Christofides' algorithm*.

[5] Recall that a *Eulerian graph* is a connected graph that has no vertices of odd degree. A *Eulerian tour* is a cycle that visits every edge of the graph exactly once. Given a Eulerian graph, we can always find a Eulerian tour.

```
Input: Complete graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$
       satisfying the triangle inequality.
Output: TSP tour of $G$.
1 Compute a minimum spanning tree $T$ of $G$.
2 Compute a perfect matching $M$ on the odd degree vertices $V'$ of $T$.
3 Combine $T$ and $M$ to obtain a Eulerian graph $G'$.
4 Extract a Eulerian tour $C'$ from $G'$.
5 Traverse $C'$ and short-cut previously visited vertices.
6 Output the resulting tour $C$.
```

Algorithm 16: Approximation algorithm for *metric TSP*.

Theorem 10.5. *Algorithm 16 is a $\frac{3}{2}$-approximation algorithm for metric TSP.*

Proof. Note that the algorithm can be implemented to run in polynomial time (computing a perfect matching in an undirected graph can be done in polynomial time). The proof follows because the Eulerian graph G' has total cost $c(T)+c(M)$. Because of the triangle-inequality, short-cutting the Eulerian tour C' does not increase the cost. The resulting tour C has thus cost at most $c(T)+c(M)$, which by Lemmas 10.2 and 10.3 is at most $\frac{3}{2}\mathsf{OPT}$. □

The algorithm is tight (example omitted). Despite intensive research efforts, this is still the best known approximation algorithm for the *metric TSP* problem.

10.4 Approximation Algorithm for *Steiner Tree*

We next consider a fundamental network design problem, namely the *Steiner tree problem*. It naturally generalizes the *minimum spanning tree problem*:

Steiner Tree Problem:

Given:	An undirected graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$ and a set of terminal nodes $R \subseteq V$.
Goal:	Compute a minimum cost tree T in G that connects all terminals in R.

The nodes in R are usually called *terminals*; those in $V \setminus R$ are called *Steiner* nodes. The *Steiner tree* problem thus asks for the computation of a minimum cost tree, also called *Steiner tree*, that spans all terminals in R and possibly some Steiner nodes. The decision variant of the problem is *NP*-complete. Note that if we knew the set $S \subseteq V \setminus R$ of Steiner nodes that are included in an optimal solution, then we could simply compute an optimal Steiner tree by computing a minimum spanning tree on the vertex set $R \cup S$ in G. Thus, the difficulty of the problems is that we do not know which Steiner nodes to include.

We first show that we can restrict our attention without loss of generality to the so-called *metric Steiner tree problem*. In the metric version of the problem, we are given a *complete* graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$ that satisfy the *triangle* *inequality*, i.e., for every $u, v, w \in V$, $c_{uw} \leq c_{uv} + c_{vw}$.

Given a subset $Q \subseteq E$ of the edges, we define $c(Q)$ as the total cost of all edges in Q, i.e., $c(Q) = \sum_{e \in Q} c_e$.

Theorem 10.6. *There is an approximation preserving polynomial-time reduction from Steiner tree to metric Steiner tree.*

Proof. Consider an instance $I = (G, c, R)$ of the *Steiner tree problem* consisting of a graph $G = (V, E)$ and edge costs $c : E \to \mathbb{R}^+$. We construct a corresponding instance $I' = (G', c', R)$ of *metric Steiner tree* as follows. Let $G' = (V, E')$ be the complete undirected graph on vertex set V and let E' be its set of edges. Define the cost c'_e of edge $e = (u, v) \in E'$ as the cost of a shortest path between u and v in G. (G', c') is called the *metric closure* of (G, c). The set of terminals in I and I' is identical.

Suppose we are given a Steiner tree T in G. Then the cost of the Steiner tree T in (G', c') can only be smaller. Next suppose we are given a Steiner tree T' of (G', c'). Each edge $e = (u, v) \in T'$ corresponds to a shortest u, v-path in G. The subgraph of G induced by all edges in T' connects all terminals in R and has cost at most $c'(T')$ but may contain cycles in general. If so, remove edges to obtain a tree T in G. Clearly, $c(T) \leq c'(T')$. $\square$

In light of Theorem 10.6, we concentrate on the *metric Steiner tree problem* subsequently.

As mentioned before, the key to derive good approximation algorithms for a problem is to develop good lower bounds on the optimal cost OPT. One such lower bound is the following:

Lemma 10.4. *Let T be a minimum spanning tree on the terminal set R of G. Then $OPT \geq \frac{1}{2}c(T)$.*

Proof. Consider an optimal Steiner tree of cost OPT. By doubling the edges of this tree, we obtain a Eulerian graph of cost $2\mathsf{OPT}$ that connects all terminals in R and a (possibly empty) subset of Steiner vertices. Find a Eulerian tour C' in this graph (e.g., by traversing vertices in their depth-first search order). We obtain a Hamiltonian cycle C on R by traversing C' and short-cutting Steiner vertices and previously visited terminals. Because of the triangle inequality, this short-cutting will not increase the cost and the cost of C is thus at most $c(C') = 2\mathsf{OPT}$. Delete an arbitrary edge of C to obtain a spanning tree on R of cost at most $2\mathsf{OPT}$. The cost of a minimum spanning tree T on R is less than or equal to the cost of this spanning tree, which is at most $2\mathsf{OPT}$. $\square$

Lemma 10.4 gives rise to the following approximation algorithm.

Theorem 10.7. *Algorithm 17 is a* 2*-approximation algorithm for metric Steiner tree.*

Proof. Certainly, the algorithm has polynomial running time and outputs a feasible solution. The approximation ratio of 2 follows directly from Lemma 10.4. $\square$

```
Input: Complete graph $G = (V,E)$ with non-negative edge costs $c : E \to \mathbb{R}^+$
       satisfying the triangle inequality and a set of terminal vertices $R \subseteq V$.
Output: Steiner tree $T$ on $R$.
1 Compute a minimum spanning tree $T$ on terminal set $R$.
2 Output $T$.
```

Algorithm 17: Approximation algorithm for *metric Steiner tree*.

The analysis of Algorithm 10.4 is tight as the following example shows:

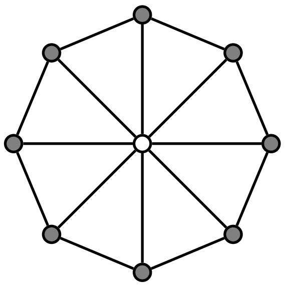


Figure 18: Example graph

Example 10.2. Consider a complete graph that consists of k outer terminal vertices $R = \{t_1, \dots, t_k\}$ that are connected to one inner Steiner vertex; see Figure 18 for an example with $k = 8$. The edges to the Steiner vertex have cost 1; all remaining ones have cost 2. Note that these edge costs satisfy the triangle inequality. A minimum spanning tree on the gray vertices has total cost $2(k-1)$ while the minimum Steiner tree has cost k. That is, the approximation ratio of Algorithm 10.4 on this instance is $2 - 2/k$. As k goes to infinity, this ratio approaches 2.

There are much better approximation algorithms for this problem. The current best approximation ratio is 1.386. Inapproximability results show that the problem cannot be approximated arbitrarily well. In particular, there is no 96/95-approximation algorithm for the *metric Steiner tree* problem.

10.5 Approximation Scheme for *Knapsack*

We next consider the *knapsack problem*:

Knapsack Problem:

Given:	A set $N = \{1, \dots, n\}$ of n items with each item $i \in N$ having a profit $p_i \in \mathbb{Z}^+$ and a weight $w_i \in \mathbb{Z}^+$, and a knapsack whose (weight) capacity is $B \in \mathbb{Z}^+$.
Goal:	Find a subset $X \subseteq N$ of items whose total weight $w(X) = \sum_{i \in X} w_i$ is at most B such that the total profit $p(X) = \sum_{i \in X} p_i$ is maximum.

We will assume without loss of generality that $w_i \le B$ for every $i \in N$ and that $p_i > 0$ for every $i \in N$; items not satisfying one of these conditions can safely be ignored.

The *knapsack problem* is *NP*-hard and we therefore seek a good approximation algorithm for the problem. As we will see, we can even derive an *approximation scheme* for this problem:

Definition 10.3. An algorithm ALG is an *approximation scheme* for a maximization problem Π if for every given error parameter $\varepsilon > 0$ and every instance $I \in \mathcal{I}$, it computes a feasible solution $S \in \mathcal{F}$ of profit $p(S) \geq (1-\varepsilon)\mathsf{OPT}(I)$. An approximation scheme ALG is a

- *polynomial time approximation scheme (PTAS)* if for every fixed $\varepsilon > 0$ its running time is polynomial in the size of the instance I;
- *fully polynomial time approximation scheme (FPTAS)* if its running time is polynomial in the size of the instance I and $\frac{1}{\varepsilon}$.

Note that the running time of a PTAS might grow exponentially in $\frac{1}{\varepsilon}$, e.g., like $O(2^{1/\varepsilon}n^2)$, while this is not feasible for a FPTAS.

10.5.1 Dynamic Programming Approach

The algorithm is based on the following *dynamic programming* approach. Let the maximum profit of an item in a given instance I be denoted by $P(I)$ (we will omit I subsequently). A trivial upper bound on the total profit that any solution for I can achieve is nP. Define for every $i \in N$ and every $p \in \{0,\dots,nP\}$:

$$A(i,p) = \text{minimum weight of a subset } S \subseteq \{1,\dots,i\} \text{ whose profit } p(S) \text{ is exactly } p.$$

Let $A(i,p) = \infty$ if no such set exists. Suppose we were able to compute $A(i,p)$. We can then easily determine the total profit of an optimal solution:

$$\mathsf{OPT} = \max\{p \in \{0,\dots,nP\} \mid A(n,p) \leq B\}.$$

Clearly, $A(1,p_1) = w_1$ and $A(1,p) = \infty$ for every $p > 0$ with $p \neq p_1$. We further set $A(i,0) = 0$ for every $i \in N$ and implicitly assume that $A(i,p) = \infty$ for every $p < 0$. Lets see how to compute $A(i+1,p)$ for $i \geq 1$ and $p > 0$. There are two options: either we include item $i+1$ into the knapsack or not. If we include item $i+1$, then it contributes a profit of p_{i+1} and thus the minimum weight of a subset of $\{1,\dots,i+1\}$ with profit p is equal to the minimum weight $A(i,p-p_{i+1})$ of the first i items yielding profit $p-p_{i+1}$ plus the weight w_{i+1} of item $i+1$. If we do not include item $i+1$, then $A(i+1,p)$ is equal to $A(i,p)$. Thus

$$A(i+1,p) = \min\{w_{i+1} + A(i,p-p_{i+1}),\ A(i,p)\}. \tag{17}$$

We can therefore compute the table of entries $A(i,p)$ with $i \in N$ and $p \in \{0,\dots,nP\}$ in time $O(n^2P)$.

Note that the dynamic program has *pseudo-polynomial* running time (cf. Definition 9.7), i.e., the running time of the algorithm is polynomial in the size of the instance (here n) and the largest integer appearing in the instance (here P). However, we can use this pseudo-polynomial algorithm in combination with the following rounding idea to obtain a fully polynomial-time approximation scheme for this problem.

```
Input: A set $N = \{1,\dots,n\}$ of items with a profit $p_i \in \mathbb{Z}^+$ and a weight $w_i \in \mathbb{Z}^+$ for
       every item $i \in N$ and a knapsack capacity $B \in \mathbb{Z}^+$.
Output: Subset $X' \subseteq N$ of items.
1 Set $t = \lfloor \log_{10}(\frac{\varepsilon P}{n}) \rfloor$.
2 Define truncated profits $\bar{p_i} = \lfloor p_i/10^t \rfloor$ for every $i \in N$.
3 Use the dynamic program (17) to compute an optimal solution $X'$ for the
  knapsack instance $(N, (\bar{p_i}), (w_i), B)$.
4 Output $X'$.
```

Algorithm 18: Approximation scheme for *knapsack*.

10.5.2 Deriving a FPTAS for *Knapsack*

Note that the above algorithm runs in polynomial time if all profits of the items are small numbers, e.g., if they are polynomially bounded in n. The key idea behind deriving a FPTAS is to ignore a certain number (depending on the error parameter ε) of least significant bits of the items' profits. The modified profits can be viewed as numbers that are polynomially bounded in $|I|$ and $\frac{1}{\varepsilon}$. As a consequence, we can compute an optimal solution for the modified profits in time polynomial in $|I|$ and $\frac{1}{\varepsilon}$ using the above dynamic program. Because we only ignore the least significant bits, this solution will be a $(1-\varepsilon)$-approximate solution with respect to the original profits. Subsequently, we elaborate on this idea in more detail.

Suppose we truncate the last t digits of each item's profit. That is, define the *truncated profit* $\bar{p_i}$ of item i as $\bar{p_i} = \lfloor p_i/10^t \rfloor$. Now use the dynamic program above to compute an optimal solution X' for the instance with truncated profits. This takes time at most $O(n^2 P/10^t)$.

Certainly, X' may be sub-optimal for the original problem, but its total profit relates to the one of an optimal solution X for the original problem as follows:

$$\sum_{i \in X'} p_i \geq \sum_{i \in X'} 10^t \bar{p_i} \geq \sum_{i \in X} 10^t \bar{p_i} \geq \sum_{i \in X} (p_i - 10^t) \geq \sum_{i \in X} p_i - n10^t.$$

Here, the first and third inequalities hold because of the definition of truncated profits. The second inequality follows from the optimality of X'. Thus, the total profit of X' satisfies

$$p(X') \geq p(X) - n10^t = \mathsf{OPT}\left(1 - \frac{n10^t}{\mathsf{OPT}}\right) \geq \mathsf{OPT}\left(1 - \frac{n10^t}{P}\right).$$

Note that the last inequality holds because $\mathsf{OPT} \geq P$. Suppose we wish to obtain an approximation ratio of $1-\varepsilon$. We can accomplish this by letting t be the smallest integer such that $n10^t/P \leq \varepsilon$, or, equivalently,

$$t = \left\lfloor \log_{10}\left(\frac{\varepsilon P}{n}\right) \right\rfloor.$$

With this choice, the running time of the dynamic program is $O(n^2 P/10^t) = O(n^3/\varepsilon)$.

That is, for any $\varepsilon > 0$ we have an $(1-\varepsilon)$-approximation algorithm whose running time is polynomial in the size of the instance and $\frac{1}{\varepsilon}$.

We summarize the result in the following theorem.

Theorem 10.8. *Algorithm 18 is a fully polynomial time approximation scheme for the knapsack problem.*

References

The presentation of the material in this section is based on [9, Chapters 1 & 3].

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: Theory, algorithms, and applications*, Prentice Hall, New Jersey, 1993.

[2] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and Schrijver A., *Combinatorial optimization*, Wiley, 1998.

[3] T. H. Cormen, C. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, 2001.

[4] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[5] B. Korte and J. Vygen, *Combinatorial optimization: Theory and algorithms*, Springer, 2008.

[6] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: Algorithms and complexity*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[7] A. Schrijver, *Combinatorial optimization: Polyhedra and efficiency*, Springer, Berlin, Heidelberg, New York, 2003.

[8] R. E. Tarjan, *Data structures and network algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[9] Vijay V. Vazirani, *Approximation algorithms*, Springer, Berlin, Heidelberg, New-York, 2001.