Combinatorial Optimization

Lecture Notes, Summer Term 08

University of Freiburg

Alexander Souza

Contents

1 Introduction **4**
1.1 Examples 4
1.2 Combinatorial Optimization Problems 5
1.3 Algorithms and Approximation 5
1.4 Basic Graph Theory 6

I Optimization Algorithms **8**

2 Linear Programming **9**
2.1 Introduction 9
2.2 Polyhedra 11
2.3 Simplex Algorithm 14
2.4 Duality 19

3 Network Flows **23**
3.1 Maximum Flows and Minimum Cuts 23
3.2 Edmonds-Karp Algorithm 26
3.3 Minimum Cost Flows 27
3.4 Assignment Problem 29

II Approximation Algorithms **31**

4 Knapsack **32**
4.1 Fractional Knapsack and Greedy 33
4.2 Pseudo-Polynomial Time Algorithm 34
4.3 Fully Polynomial-Time Approximation Scheme 36

5 Set Cover **38**
5.1 Greedy Algorithm 39
5.2 Primal-Dual Algorithm 42
5.3 LP-Rounding Algorithms 44

6 Satisfiability **48**
6.1 Randomized Algorithm 49
6.2 Derandomization 51

7 Facility Location **53**
7.1 Complementary Slackness 54
7.2 Primal-Dual Algorithm 55

Chapter 1

Introduction

1.1 Examples

We start with two examples of combinatorial optimization problems.

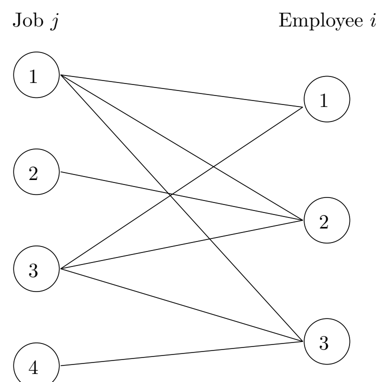
Example 1.1. We are given an amount of C Euro and wish to invest it among a set of n options. Each such option i has cost c_i and profit p_i. The goal is to maximize the total profit.

Consider $C = 100$ and the following cost-profit table:

Option	Cost	Profit
1	100	150
2	1	2
3	50	55
4	50	100

Our choice of purchased options must not exceed our capital C. Thus the feasible solutions are $\{1\}, \{2\}, \{3\}, \{4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}$. Which is the best solution? We evaluate all possibilities and find that $\{3, 4\}$ give 155 altogether which maximizes our profit.

Example 1.2. We have a set of n jobs that need to be done and each job j has a processing time p_j. Each job can be worked on by a subset of our m employees. For each job j we introduce a set S_j of the employees that are eligible to work on that particular job. The following diagramm visualizes the sets S_j: for each job j (on the left hand side) we see which employee (on the right hand side) is able to work on that job.



Several employees can contribute to the same job at the same time and each employee can contribute to several jobs (but not at the same time). The goal is to get all the jobs

done as early as possible. We can formulate our problem with the following mathematical program. We use the variables x_{ij} that tell us for how long employee i works on job j. We wish to minimize the longest duration some employee is busy.

$$
\begin{array}{llll}
\text{minimize} & \max_i \sum_j x_{ij} & & \text{“minimize longest duration”} \\
\text{subject to} & \sum_{i \in S_j} x_{ij} = p_j & j = 1, \dots, n & \text{“each job gets done”} \\
& x_{ij} \geq 0 & i = 1, \dots, m,\ j = 1, \dots, n & \text{“non-negative contributions”}
\end{array}
$$

We will learn soon how to solve such a mathematical program.

1.2 Combinatorial Optimization Problems

An instance of a *combinatorial optimization problem* (COP) can formally be defined as a tuple $I = (U, P, S, \mathrm{val}, \mathrm{extr})$ with the following meaning:

U	the *solution space* (on which val and S are defined),
P	the *feasibility predicate*,
S	the set of *feasible solutions*: $S = \{X \in U : X \text{ satisfies } P\}$,
val	the *objective function* $\mathrm{val} : U \to \mathbb{R}$,
extr	the *extremum* (usually max or min).

Our goal is to find a feasible solution where the desired extremum of val is attained. Any such solution is called an *optimum solution*, or simply an *optimum*. U and S are usually not given explicitly, but implicitly (S through the feasibility predicate P).

Let us formulate the problem in Example 1.1 in this manner.

$$
\begin{aligned}
U &= 2^{\{1,2,3,4\}}, \\
P &= \text{“total cost is at most } C\text{”, i.e., } X \in S \text{ if } \sum_{i \in X} c_i \leq C \\
S &= \{\{1\}, \{2\}, \{3\}, \{4\}, \{2,3\}, \{2,4\}, \{3,4\}\}, \\
\mathrm{val} &= \begin{cases} U \to \mathbb{R} \\ X \mapsto \sum_{i \in X} p_i, \end{cases} \\
\mathrm{extr} &= \max .
\end{aligned}
$$

The optimum solution here is $\{3, 4\}$ with value 155.

A central problem around combinatorial optimization is that it is often in principle possible to find an optimum solution by enumerating the set of feasible solutions, but this set mostly contains *too many* elements. This phenomenon is called *combinatorial explosion*.

1.3 Algorithms and Approximation

Many problems in combinatorial optimization can be solved by using an appropriate algorithm. Informally, an *algorithm* is given a (valid) input, i.e., a description of an instance of a problem and computes a solution after a finite number of “elementary steps”. The number of bits used to describe an input x is called the (binary) *length* or *size* of the input and denoted $\mathrm{size}(x)$.

Let $t : \mathbb{N} \to \mathbb{R}$ be a function. We say that an algorithm *runs* in time $O(t)$ if there is a constant α such that the algorithm uses at most $\alpha t(\text{size}(x))$ many elementary steps to compute a solution given any input x. An algorithm is called *polynomoial time* if $t : n \mapsto n^c$ for some constant c. This contrasts *exponential time* algorithms where $t : n \mapsto c^n$ for some constant $c > 1$.

Because the running times of exponential time algorithms grow rather rapidly as the input size grows, we are mostly interested in polynomial time algorithms. Of course, we desire to find an optimum solution for any given COP in polynomial time. Unfortunately this is not always possible as many COPs are NP-hard. (It is widely believed that no polynomial time algorithm exists that solves some NP-hard COP optimally on every instance.) Thus our goal is a find "good" solutions in polynomial time.

Let $\Pi = \{I_1, I_2, \dots\}$ be a set of instances of a COP, where each $I \in \Pi$ is of the form $I = (U, P, S, \text{val}, \text{extr})$. For any $I \in \Pi$, let $\text{OPT}(I) = \text{extr}_{X \in S(I)} \text{val}(X)$ denote the respective optimum value. An *approximation algorithm* ALG for Π is a polynomial time algorithm that computes some solution $X \in S(I)$ for every instance $I \in \Pi$. The respective value obtained is denoted $\text{ALG}(I) = \text{val}(X)$. The *approximation ratio* of ALG on an instance I is defined by

$$\rho_{\text{ALG}}(I) = \frac{\text{ALG}(I)}{\text{OPT}(I)}.$$

The algorithm ALG is a ρ-*approximation* algorithm if

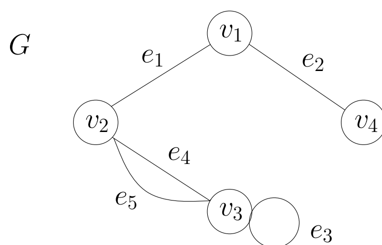
$$\begin{aligned} \rho_{\text{ALG}}(I) &\leq \rho \qquad \text{for all } I \in \Pi \text{ and extr} = \min, \\ \rho_{\text{ALG}}(I) &\geq \rho \qquad \text{for all } I \in \Pi \text{ and extr} = \max. \end{aligned}$$

1.4 Basic Graph Theory

Graphs often help us to model COPs in a simple and natural manner. For instance, the diagram in Example 1.2 is a graph.

An *undirected graph* $G = (V, E)$ consists of a *vertex set* $V = V(G)$, an *edge set* $E = E(G)$, and a relation Ψ, called *incidence*, that assigns to each edge $e \in E$ exactly two (not necessarily distinct) vertices $u, v \in V$. We write $e = uv$ for short. The vertices u and v are called the *endvertices* of $e = uv$. We say that e *joins* u and v, where v is the *head* and u the *tail* of the edge. We will also use the terms that u and v are *neighbors* and *adjacent*. A *loop* is an edge whose endvertices are equal. A *multiedge* is a set of edges that have the same respective endvertices. A graph without loops and multiedges is called *simple*.

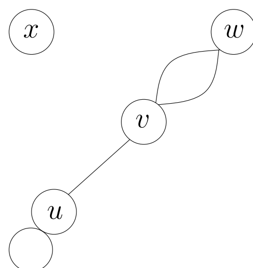
Example 1.3. The vertex set of the graph G below is $V = \{v_1, v_2, v_3, v_4\}$, the edge set is $E = \{e_1, e_2, e_3, e_4, e_5\}$, and the incidence relation is $\Psi = \{e_1 = v_1v_2, e_2 = v_1v_4, e_3 = v_3v_3, e_4 = v_2v_3, e_5 = v_2v_3\}$.



The graph G is not simple because it contains the loop e_3 and the multiedge $\{e_4, e_5\}$.

The number of edges incident with a vertex v is called its *degree* and denoted $\deg(v)$; loops count twice. A vertex with degree zero is *isolated*.

Example 1.4. In the graph G below vertex x is isolated, i.e., $\deg(x) = 0$. For the others we have $\deg(u) = 3$, $\deg(v) = 3$, and $\deg(w) = 2$.



We usually interpret Ψ so that the edges do not have directions, i.e., for an edge $e = uv$, we can reach vertex u from v and vice versa (hence the name undirected graph). Sometimes it is useful when edges have directions; then they are called *arcs*. For an arc $e = uv$, vertex v can be reached from u but not vice versa. Such graphs are called *directed graphs* (*digraphs*) and denoted $G = (V, A)$.

A series of vertices $v_1, v_2, \ldots, v_k$ is called a *trail* if $v_i v_{i+1} \in E$ for $i = 1, \ldots, k$. If $v_1 \neq v_k$ then the trail is *open*, or a *path*. Otherwise, i.e., $v_1 = v_k$, the trail is *closed*, or a *cycle*. If a trail $v_1, v_2, \ldots, v_k$ satisfies that $v_i \neq v_j$ for $i \neq j$, then the trail is called *simple*.

Two vertices u, v are *connected* if there exists a path between u and v. The equivalence classes of the relation "are connected" are called the *connected components* of G.

Part I

Optimization Algorithms

Chapter 2

Linear Programming

Linear programs (LP) play an important role in the theory and practice of optimization problems. Many COPs can directly be formulated as LPs. Furthermore, LPs are invaluable for the design and analysis of approximation algorithms. Generally speaking, LPs are COPs with linear objective function and linear constraints, where the variables are defined on a continous domain. We will be more specific below.

2.1 Introduction

We begin our treatment of linear programming with an example of a transportation problem to illustrate how LPs can be used to formulate optimization problems.

Example 2.1. There are two brickworks w_1, w_2 and three construction sites s_1, s_2, s_3. The works produce $b_1 = 60$ and $b_2 = 30$ tons of bricks per day. The sites require $c_1 = 30$, $c_2 = 20$ and $c_3 = 40$ tons of bricks per day. The transportation costs t_{ij} per ton from work w_i to site s_j are given in the following table:

t_{ij}	s_1	s_2	s_3
w_1	40	75	50
w_2	20	50	40

Which work delivers which site in order to minimize the total transportation cost? Let us write the problem as a mathematical program. We use variables x_{ij} that tell us how much we deliver from work w_i to site s_j.

$$
\begin{aligned}
\text{minimize} \quad & 40x_{11} + 75x_{12} + 50x_{13} + 20x_{21} + 50x_{22} + 40x_{23} \\
\text{subject to} \quad & x_{11} + x_{12} + x_{13} \leq 60 \\
& x_{21} + x_{22} + x_{23} \leq 30 \\
& x_{11} + x_{21} = 30 \\
& x_{12} + x_{21} = 20 \\
& x_{13} + x_{23} = 40 \\
& x_{ij} \geq 0 \quad i = 1, 2, \; j = 1, 2, 3.
\end{aligned}
$$

How do we find the best x_{ij}?

This question will be the subject of this chapter. The general linear programming problem reads as follows:

Problem 2.1 Linear Programming

Instance. Matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$.

Task. Solve the problem

$$\begin{aligned} \text{maximze} \quad & c^\top x, \\ \text{subject to} \quad & Ax \leq b, \\ & x \in \mathbb{R}^n. \end{aligned}$$

That means solve one of the following questions.

(1) Find a vector $x \in \mathbb{R}^n$ such that $Ax \leq b$ and $\text{val}(x) = c^\top x$ is maximum, or
(2) decide that the set $S = \{x \in \mathbb{R}^n : Ax \leq b\}$ is empty, or
(3) decide that for all $\alpha \in \mathbb{R}$ there is an $x \in \mathbb{R}^n$ with $Ax \leq b$ and $c^\top x > \alpha$.

As a shorthand we shall frequently write $\max\{c^\top x : Ax \leq b\}$. We can assume that we deal with a maximization problem without loss of generality because we can treat a minimization problem if we replace c with $-c$.

The function $\text{val}(x) = c^\top x$ is the *objective function.* A feasible x^* which maximizes val is an *optimum solution* and the value $z^* = \text{val}(x^*)$ is called *optimum value.* Any $x \in \mathbb{R}^n$ that satisfies $Ax \leq b$ is called *feasible.* The set $S = \{x \in \mathbb{R}^n : Ax \leq b\}$ is called the *feasible region*, i.e., the set of *feasible solutions.* If S is empty, then the problem is *infeasible.* If for every $\alpha \in \mathbb{R}$, there is a feasible x such that $c^\top x > \alpha$ then the problem is *unbounded.* This simply means that the maximum of the objective function does not exist.

Basic Ideas for Solving Linear Programming

Here we illustrate the main ideas for solving linear programming. A lot of the intuition for the general case can already be obtained when we restrict ourselves to LPs with only two variables. Our working example for this introduction will be the following:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 && (2.1) \\ \text{subject to} \quad & 4x_1 - x_2 \leq 8 && (2.2) \\ & 2x_1 + x_2 \leq 10 && (2.3) \\ & -5x_1 + 2x_2 \leq 2 && (2.4) \\ & x_1 \geq 0 && (2.5) \\ & x_2 \geq 0 && (2.6) \end{aligned}$$

Each setting of x_1, x_2 such that (2.2)–(2.6) are satisfied simultaneously is feasible. Their set is the feasible region S and is the intersection of linear half planes. In Figure 2.1, the gray shaded regions are excluded from the feasible region by the respective constraints; S is what "is left" and drawn in green.

Our objective function is $\text{val}(x_1, x_2) = x_1 + x_2$. How do we find an optimum solution? We can not even evaluate this function for every feasible solution because these are infinitely many. However, in two dimensions we can optimize as follows. We search the largest value z (for example with binary search) such that the line $x_1 + x_2 = z$ has an

intersection with S. In the example this is $x_1 + x_2 = 8$ with unique feasible (optimum) solution $(x_1, x_2) = (2, 6)$. See Figure 2.2 for a visualization.

Unfortunately, this approach does not quite work when we have more than two variables. Here we explain the ideas for of an algorithm that works for general LPs.

Observe that the optimum solution $(x_1, x_2) = (2, 6)$ is one of the "corners" of the feasible region in Figure 2.1. We will see later that this also holds for general LPs: If the LP is bounded and feasible, there is always a "corner" with optimum objective value. This suggests the following algorithm, called SIMPLEX: Start with an arbitary "corner". If the current one is not optimal find a neighboring one so that the objective function improves. Progress in this manner from "corner" to "corner" until an optimum solution is found. The procedure can be seen in Figure 2.3.

2.2 Polyhedra

Recall that a point $x \in \mathbb{R}^n$ is feasible if it satisfies certain linear inequalities simultaneously. So the feasible region is the intersection of finitely many halfspaces. More precisely:

A *polyhedron* in $\mathbb{R}^n$ is a set $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ for some matrix $A \in \mathbb{R}^{m \times n}$ and some vector $b \in \mathbb{R}^m$. A bounded polyhedron is called *polytope*.

We denote by $\text{rank}(M)$ the rank of a matrix $M \in \mathbb{R}^{m \times n}$, i.e., the number of linear independent columns of M. The *dimension* of any non-empty set $X \subseteq \mathbb{R}^n$ is defined to be

$$\dim(X) = n - \max\{\text{rank}(M) : \ M \text{ is a } n \times n \text{ matrix with } Mx = My \text{ for all } x, y \in X\}.$$

A polyhedron $P \subseteq \mathbb{R}^n$ is *full-dimensional* if $\dim(P) = n$. A polyhedron is full-dimensional if and only if there is a point in its interior, i.e., if there are no implicit equalities in the system $Ax \leq b$.

Let $P = \{x : Ax \leq b\}$ be a non-empty polyhedron with dimension d. Let c be a vector for which $\delta := \max\{c^\top x : x \in P\} < \infty$, then $H_c = \{x : c^\top x = \delta\}$ is called *supporting hyperplane* of P. A *face* of P is the intersection of P with a supporting hyperplane of P. Three types of faces are particular important, see Figure 2.4:

(1) A *facet* is a face of dimension $d - 1$,

(2) a *vertex* is a face of dimension zero (a point), and

(3) an *edge* is a face of dimension one (a line segment).

The following lemma (whose proof is an exercise) essentially states that a set $F \subseteq P$ is a face of a polyhedron P if and only if some of the inequalities of $Ax \leq b$ are satisfied with equality for all elements of F.

Lemma 2.2. *Let $P = \{x : Ax \leq b\}$ be a polyhedron and $F \subseteq P$. Then the following statements are equivalent:*

(1) F is a face of P.

(2) There is a vector c with $\delta := \max\{c^\top x : x \in P\} < \infty$ and $F = \{x \in P : c^\top x = \delta\}$.

(3) $F = \{x \in P : A'x = b'\} \neq \emptyset$ for some subsystem $A'x \leq b'$ of $Ax \leq b$.

As important corollaries we have:

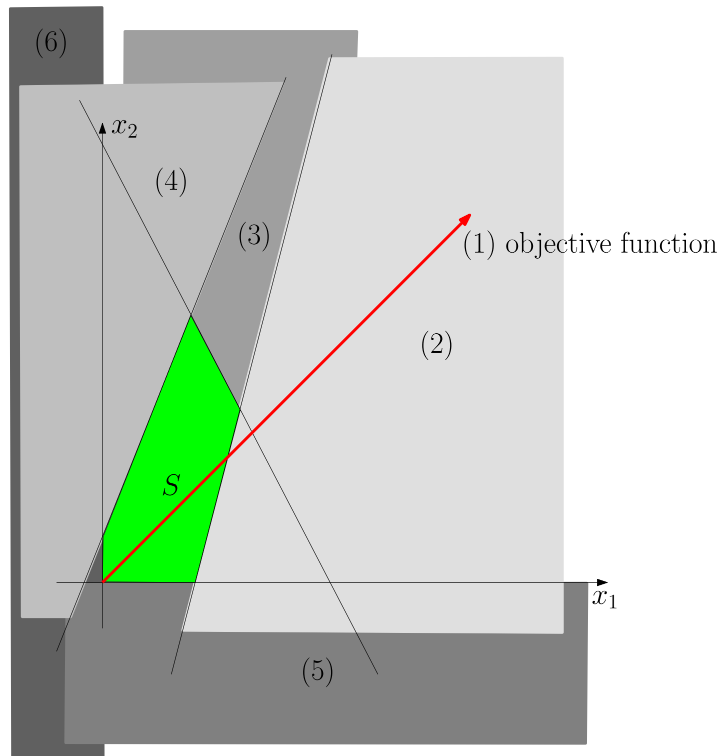


Figure 2.1: Visualization of the above LP.

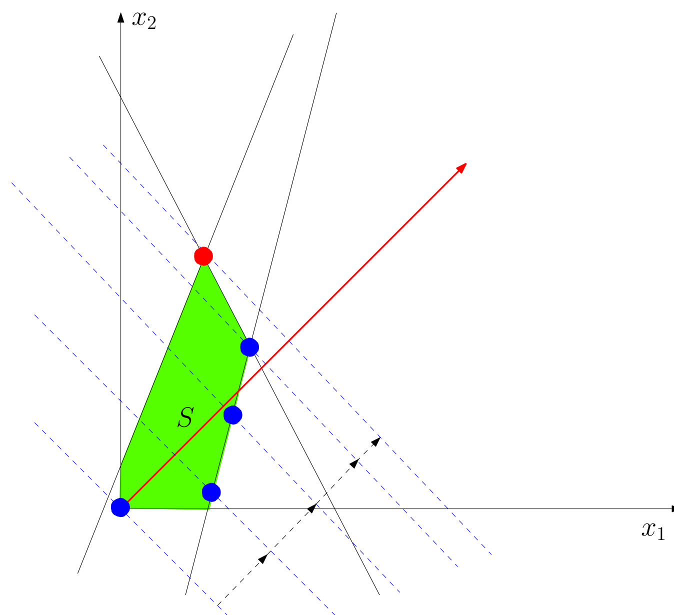


Figure 2.2: Searching the optimum: the direct way.

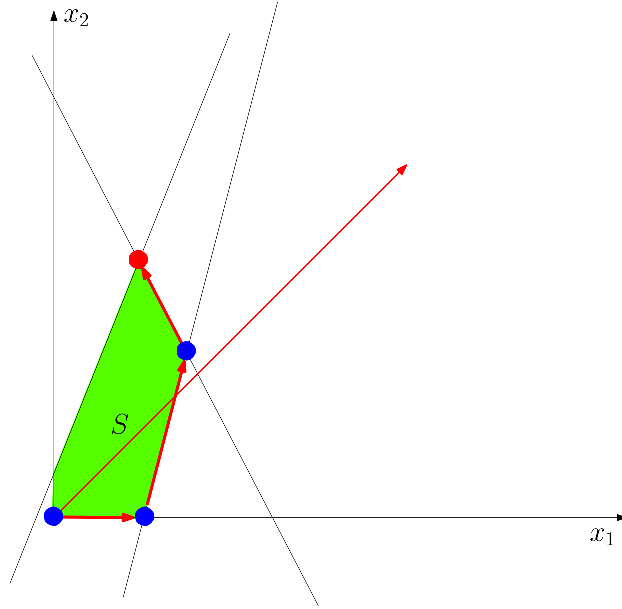


Figure 2.3: Searching the optimum: the SIMPLEX way.

Corollary 2.3. *If* $\max\{c^\top x : x \in P\} < \infty$ *for a non-empty polyhedron* P *and a vector* c*, then the set of points where the maximum is attained is a face of* P*.*

Corollary 2.4. *Let* P *be a polyhedron and* F *a face of* P*. Then* F *is again a polyhedron. Furthermore, a set* $F' \subseteq F$ *is a face of* P *if and only if it is a face of* F*.*

An inequality $c^\top x \leq \delta$ is *facet-defining* for P if $c^\top x \leq \delta$ for all $x \in P$ and $\{x \in P : c^\top x = \delta\}$ is a facet of P.

Lemma 2.5. *Let* $P \subseteq \{x \in \mathbb{R}^n : Ax = b\}$ *be a non-empty polyhedron of dimension* $n - \text{rank}(A)$*. Let* $A'x \leq b'$ *be a minimal inequality system such that* $P = \{x : Ax = b, A'x \leq b'\}$*. Then each inequality of* $A'x \leq b'$ *is facet-defining for* P*, and each facet of* P *is defined by an inequality of* $A'x \leq b'$*.*

Proof. If $P = \{x \in \mathbb{R}^n : Ax = b\}$, then there are no facets and the statement is trivial. Thus $P \subset \{x \in \mathbb{R}^n : Ax = b\}$. So let $A'x \leq b'$ be a minimal inequality system such that $P = \{x \in \mathbb{R}^n : Ax = b, A'x \leq b\}$, let $a^\top x \leq \beta$ be one of its inequalities, $A''x \leq b''$ be the rest of the system $A'x \leq b'$. Pick a vector y with $Ay = b$, $A''y \leq b$ and $a^\top y > \beta$ (which must exist because the inequality $a^\top x \leq \beta$ is not redundant) and pick $x \in P$ such that $a^\top x < \beta$ (which must exist because $\dim(P) = n - \text{rank}(A)$).

Consider $z = x + (\beta - a^\top x)/(a^\top y - a^\top x)(y - x)$ which satisfies $a^\top z = \beta$ and $z \in P$. Hence $F := \{x \in P : a^\top x = \beta\} \neq \emptyset$ and $F \neq P$. This yields that F is a face (by Lemma 2.2) and has dimension $n - \text{rank}(A) - 1$. □

An other important class of faces are *minimal faces*, i.e., faces that do not contain any other face. For these we have:

Lemma 2.6. *Let* $P = \{x : Ax \leq b\}$ *be a polyhedron. A non-empty set* $F \subseteq P$ *is a* minimal *face of* P *if and only if* $F = \{x \in \mathbb{R}^n : A'x = b'\}$ *for some subsystem of* $Ax \leq b$*.*

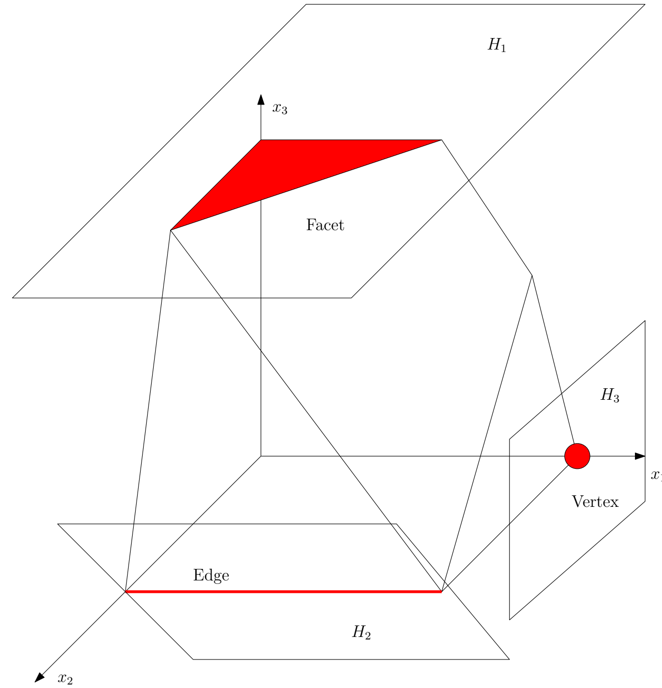


Figure 2.4: Facet, vertex, and edge are all faces.

Proof. If F is a minimal face of P, by Lemma 2.2 there is a subsystem $A'x \le b'$ of $Ax \le b$ such that $F = \{x \in P : A'x = b'\}$. We choose $A'x \le b'$ maximal. Let $A''x \le b''$ be a minimal subsystem of $Ax \le b$ such that $F = \{x : A'x = b', A''x \le b''\}$. We claim that $A''x \le b''$ does not contain any inequality.

Suppose, on the contrary, that $a''x \le \beta''$ is an inequality of $A''x \le b''$. Since it is not redundant for the description of F, Lemma 2.5 implies that $F' := \{x : A'x = b', A''x \le b'', a''x = \beta''\}$ is a facet of F. By Corollary 2.4 F' is also a face of P, contradicting the assumption that F is a minimal face of P.

Now let $\emptyset \ne F = \{x : A'x = b'\} \subseteq P$ for some subsystem $A'x \le b'$ of $Ax \le b$. Obviously F has no faces except itself. By Lemma 2.2, F is a face of P and by Corollary 2.4, F is a minimal face of P. □

Corollary 2.3 and Lemma 2.6 already imply that Linear Programming can be solved by solving the linear *equation* system $A'x = b'$ for each subsystem $A'x \le b'$. This approach obviously yields an exponential time algorithm.

2.3 Simplex Algorithm

An algorithm which is more practicable (although also exponential in the worst case) is the Simplex algorithm. The algorithm is based on the following important consequence of Lemma 2.6.

Corollary 2.7. *Let $P = \{x \in \mathbb{R}^n : Ax \le b\}$ be a polyhedron. Then all minimal faces of P have dimension $n - \text{rank}(A)$. The minimal faces of polytopes are vertices.*

This justifies that, if we are dealing with a linear program which is bounded with respect to $c^\top x$, then it suffices to search an optimum solution among the *vertices* of the

polyhedron induced by the feasible region S. This is precisely what the SIMPLEX algorithm is doing.

The SIMPLEX algorithm was introduced by Dantzig in the 1950s, but the underlying idea to traverse the vertices of the polyhedron dates back to Fourier in the 1820s. We first restrict our attention to the case when an initial vertex is given as an input. This certainly requires that the LP is feasible, but it could still be unbounded, i.e., we will solve the questions (1) and (3) in Problem 2.1. Later we discuss how to find an initial vertex if this is possible, i.e., we will also decide (2).

Given an Initial Vertex

For a set I of row indices we write A_I for the submatrix of A consisting of the rows in I only; similarly b_I for the respective components in the vector b. We abbreviate $a_i = A_{\{i\}}$ and $b_i = b_{\{i\}}$.

Now consider Algorithm 2.1. Before we prove the correctness, we explain the tasks of the individual steps. In Step 1 the algorithm finds a subsystem with n inequalities of $Ax \leq b$ that is satisfied with equality, i.e., the (given) solution x is an (initial) vertex of the polyhedron induced. This step relies on Lemma 2.6. Recall that the non-singular $n \times n$ matrix A_I is a basis of $\mathbb{R}^n$. The elements of the set I are thus called *basic indices.* In Step 2 the algorithm checks if the current vertex x is already optimal. This check uses the following important observation, called *weak duality.* Details follow below.

Lemma 2.8. *Let x and y be respective feasible solutions of the LPs*

$$\max\{c^\top x : Ax \leq b\}, \tag{2.7}$$

$$\min\{y^\top b : y^\top A = c^\top, y \geq 0\}. \tag{2.8}$$

Then $c^\top x \leq y^\top b$.

Proof. $c^\top x = (y^\top A)x = y^\top (Ax) \leq y^\top b$. □

The selection rules for i and j in Steps 3 and 4 (called *Bland's pivot rule*) avoid that the algorithm runs into cyclic repetitions. In Step 5 the algorithm changes the basic indices, "moves on" to a new vertex, and repeats.

We also show that the algorithm terminates after at most $\binom{m}{n}$ iterations (which is not polynomial). It was conjectured that SIMPLEX is polynomial until Klee and Minty gave an example where the algorithm (with Bland's pivot rule) uses 2^n iterations on an LP with n variables and $2n$ constraints. It is not known if there is a pivot rule that leads to polynomial running time. Nonetheless, SIMPLEX with Bland's pivot rule is frequently observed to terminate after few iterations when run on "practical instances".

Theorem 2.9. *The algorithm* SIMPLEX *terminates after at most $\binom{m}{n}$ iterations. If it returns x and y in Step 2, these vectors are optimum solutions for the LPs (2.7) and (2.8), respectively with*

$$c^\top x = y^\top b.$$

If the algorithm returns w in Step 3, then $c^\top w > 0$ and the LP (2.7) is unbounded.

Proof. We first show that the following conditions hold at any stage of the algorithm:

(a) $x \in P$,

(b) $A_I x = b_I$,

Algorithm 2.1 SIMPLEX

Input. A Matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and a vertex x of $P = \{x \in \mathbb{R}^n : Ax \leq b\}$.

Output. A vertex x of P attaining $\max\{c^\top x : x \in P\}$ or a vector $w \in \mathbb{R}^n$ with $Aw \leq 0$ and $c^\top w > 0$ (i.e., the LP is unbounded).

Step 1. Choose a set of n row indices I such that A_I is non-singular and $A_I x = b_I$.

Step 2. Compute $c^\top (A_I)^{-1}$ and add zeros in order to obtain a vector $y \in \mathbb{R}^m$ with $c^\top = y^\top A$ such that all entries outside I are zero. If $y \geq 0$ then return x (and y).

Step 3. Choose the minimum index i with $y_i < 0$. Let w be the column of $-(A_I)^{-1}$ with index i. Thus $A_{I-\{i\}} w = 0$ and $a_i w = -1$. If $Aw \leq 0$ then return w.

Step 4. Let

$$\lambda := \min\left\{ \frac{b_j - a_j x}{a_j w} : a_j w > 0 \; j = 1, \dots, m, \right\},$$

and let j be the smallest row index attaining this minimum.

Step 5. Set $I := (I - \{i\}) \cup \{j\}$ and $x := x + \lambda w$. Go to Step 2.

(c) A_I is non-singular,

(d) $c^\top w > 0$, and

(e) $\lambda \geq 0$.

(a) and (b) hold initially. Steps 2 and 3 guarantee $c^\top w = y^\top A w = -y_i > 0$, i.e., (d). By Step 4, $x \in P$ implies $\lambda \geq 0$, i.e., (e). Property (c) follows from $A_{I-\{i\}} w = 0$ and $a_j w > 0$. Thus we only have to show that Step 5 preserves (a) and (b).

We show that if $x \in P$, then also $x + \lambda w \in P$. For any row index k we have two cases: If $a_k w \leq 0$ then (using $\lambda \geq 0$) $a_k(x + \lambda w) \leq a_k x \leq b_k$. Otherwise $\lambda \leq (b_k - a_k x)/a_k w$ and hence $a_k(x + \lambda w) \leq a_k x + a_k w (b_k - a_k x)/a_k w = b_k$. (Indeed, λ is chosen in Step 4 to be the largest number such that $x + \lambda w \in P$). This implies (a).

To show (b), note that after Step 4 we have $A_{I-\{i\}} w = 0$ and $\lambda = (b_k - a_k x)/a_k w$, so $A_{I-\{i\}}(x + \lambda w) = A_{I-\{i\}} x = b_{I-\{i\}}$. Furthermore $a_j(x + \lambda w) = a_j x + a_j w (b_j - a_j x)/a_j w = b_j$. Thus, after Step 5 the condition $A_I x = b_I$ holds again.

Thus (a)-(e) hold at any stage. If the algorithm returns x and y in Step 2, then they are indeed feasible solutions for the LPs (2.7) and (2.8). Moreover,

$$c^\top x = y^\top A x = y^\top \begin{pmatrix} A_I \\ A_{\overline{I}} \end{pmatrix} x = y^\top \begin{pmatrix} b_I \\ b_{\overline{I}} \end{pmatrix} = y^\top b,$$

because the components of y outside I are zero. This proves the optimality by using Lemma 2.8. If the algorithm returns w in Step 3, then the LP (2.7) is unbounded because, for every $\mu \geq 0$ we have $A(x + \mu w) = Ax + \mu A w \leq b$, i.e., $x + \mu w \in P$, and $c^\top (x + \mu w) > c^\top x$.

We finally show termination after at most $\binom{m}{n}$ iterations. Let $I^{(k)}$ and $x^{(k)}$ be the set I and vertex x in iteration k. If the algorithm did not terminate after $\binom{m}{n}$ iterations, there are iterations $k < \ell$ with $I^{(k)} = I^{(\ell)}$. By (b) and (c) we must have $x^{(k)} = x^{(\ell)}$. By (d) and

(e), $c^\top x$ never decreases, and it strictly increases if $\lambda > 0$. Thus λ must be zero in the iterations $k, \dots, \ell - 1$, and $x^{(k)} = \cdots = x^{(\ell)}$.

Let h be the highest index leaving I in one of the iterations $k, \dots, \ell-1$, say in iteration p. Index h must also have been added to I in some iteration $q \in \{k, \dots, \ell-1\}$. Let y' be the vector y at iteration p and w' be the vector w at iteration q. We have $(y')^\top A w' = c^\top w' > 0$. So let r be an index for which $(y')^\top a_r w' > 0$. Since $y'_r \neq 0$, index r belongs to $I^{(p)}$. If $r > h$, index r would also belong to $I^{(q)}$ and $I^{(q+1)}$, implying $a_r w = 0$. So $r \leq h$. But by the choice of i in iteration p we have $y'_r < 0$ if and only if $r = h$, and by the choice of j in iteration q we have $a_r w' > 0$ if and only if $r = h$ (using $\lambda = 0$ and $a_r x^{(q)} = a_r x^{(p)} = b_r$ as $r \in I^{(p)}$). This is a contradiction. $\square$

The General Case

Now we show how to find an initial vertex, or to conclude that the respective LP $\max\{c^\top x : Ax \leq b\}$ is infeasible. First substitute $x = y - z$, where $y \geq 0$ and $z \geq 0$ are our new variables, and write the LP in equivalent form

$$\max\left\{\begin{pmatrix} c & -c \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} : \begin{pmatrix} A & -A \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} \leq b, y, z \geq 0\right\},$$

i.e., we have introduced the *sign-constrained* variables y and z. We may hence assume that the LP under consideration has the form

$$\max\{c^\top x : A'x \leq b', A''x \leq b'', x \geq 0\}, \tag{2.9}$$

where $b' \geq 0$ and $b'' < 0$. Now run Simplex on the instance

$$\min\{(1^\top A'')x + 1^\top y : A'x \leq b', A''x + y \geq b'', x, y \geq 0\}, \tag{2.10}$$

(1 denotes the vector whose entries are all one), where we can use the initial vertex $(x, y)^\top = (0, 0)^\top$. It is an exercise to show that the LP (2.9) is feasible if and only if the minimum value of (2.10) is exactly $1^\top b''$.

Example Execution

We consider our working example from Section 2.1 again, i.e., the algorithm is given

$$A = \begin{pmatrix} 4 & -1 \\ 2 & 2 \\ -5 & 2 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}, \quad b = \begin{pmatrix} 8 \\ 10 \\ 2 \\ 0 \\ 0 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1 \end{pmatrix},$$

and the initial vertex $x = (0, 0)^\top$.

Iteration 1

Step 1. $x = (0,0)^\top, \quad I = \{4,5\}$

Step 2. $A_I = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \quad A_I^{-1} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix},$

$c^\top A_i^{-1} = (-1,-1)^\top,$

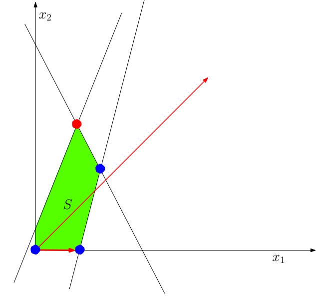
$y = (0,0,0,-1,-1)^\top \quad y \ngeq 0$

Step 3. $i = 4, \quad w = (1,0)^\top,$

$Aw = (4,2,-5,-1,0)^\top \nleq 0,$

Step 4. $\lambda = \min\left\{\frac{8-0}{4}, \frac{10-0}{2}\right\} = 2, \quad j = 1,$

Step 5. $I = \{1,5\}, \quad x = (0,0)^\top + 2(1,0)^\top = (2,0)^\top.$



Iteration 2

$x = (2,0)^\top, \quad I = \{1,5\}$

Step 2. $A_I = \begin{pmatrix} 4 & -1 \\ 0 & -1 \end{pmatrix}, \quad A_I^{-1} = \frac{1}{4}\begin{pmatrix} 1 & -1 \\ 0 & -4 \end{pmatrix},$

$c^\top A_i^{-1} = (1/4, -5/4)^\top,$

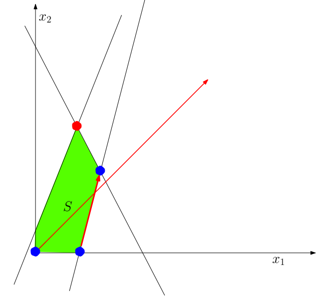
$y = (1/4,0,0,0,-5/4)^\top, \quad y \ngeq 0$

Step 3. $i = 5, \quad w = (1/4,1)^\top,$

$Aw = (0,3/2,-1/4,-1/4,-1)^\top \nleq 0,$

Step 4. $\lambda = \min\left\{\frac{10-4}{3/2}\right\} = 4, \quad j = 2,$

Step 5. $I = \{1,2\}, \quad x = (2,0)^\top + 4(1/4,1)^\top = (3,4)^\top.$



Iteration 3

$x = (3,4)^\top, \quad I = \{1,2\}$

Step 2. $A_I = \begin{pmatrix} 4 & -1 \\ 2 & 1 \end{pmatrix}, \quad A_I^{-1} = \frac{1}{6}\begin{pmatrix} 1 & 1 \\ -2 & -4 \end{pmatrix},$

$c^\top A_i^{-1} = (-1/6, 5/6)^\top,$

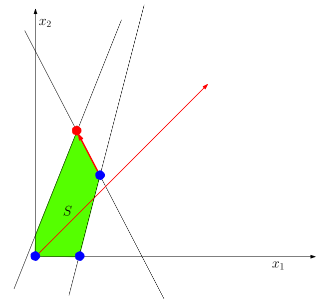
$y = (-1/6,5/6,0,0,0)^\top, \quad y \ngeq 0$

Step 3. $i = 1, \quad w = (-1/6,1/3)^\top,$

$Aw = (-1,0,9/6,1/6,-1/3)^\top \nleq 0,$

Step 4. $\lambda = \min\left\{\frac{2-(-7)}{9/6}, \frac{0-(-3)}{1/6}\right\} = 6, \quad j = 3,$

Step 5. $I = \{2,3\}, \quad x = (3,4)^\top + 6(-1/6,1/3)^\top = (2,6)^\top.$



Iteration 4

$$
\begin{aligned}
&x = (2,6)^\top, \quad I = \{2,3\} \\
\text{Step 2.}\quad & A_I = \begin{pmatrix} 2 & 1 \\ -5 & 2 \end{pmatrix}, \quad A_I^{-1} = \frac{1}{9}\begin{pmatrix} 2 & -1 \\ 5 & 2 \end{pmatrix}, \\
& c^\top A_i^{-1} = (7/9, 1/9)^\top, \\
& y = (0, 7/9, 1/9, 0, 0)^\top, \quad y \geq 0, \\
& \text{return } x = (2,6)^\top \text{ and } y = (0, 7/9, 1/9, 0, 0)^\top.
\end{aligned}
$$

2.4 Duality

Lemma 2.8 called *weak duality* was crucial for the optimality check of the SIMPLEX algorithm. In the lemma, the two considered LPs are related, but the choice of (2.8) seemed to “fall from the sky”. First we explain the rationale behind (2.7) and (2.8), then define *duality*, and finally state the *strong duality* theorem.

Ideas Behind Duality

Recall our working example from Section 2.1

$$
\begin{aligned}
\text{maximize}\quad & x_1 + x_2 && (2.11)\\
\text{subject to}\quad & 4x_1 - x_2 \leq 8 && (2.12)\\
& 2x_1 + x_2 \leq 10 && (2.13)\\
& -5x_1 + 2x_2 \leq 2 && (2.14)\\
& -x_1 \leq 0 && (2.15)\\
& -x_2 \leq 0 && (2.16)
\end{aligned}
$$

and notice that this LP is in the maximization form

$$\max\{c^\top x : Ax \leq b\}.$$

Because we are dealing with a maximization problem, every feasible solution x provides the *lower bound* $c^\top x$ on the optimum value $c^\top x^*$, i.e., we know $c^\top x \leq c^\top x^*$.

Can we also obtain *upper bounds* on $c^\top x^*$? For any feasible solution x, the constraints (2.12)–(2.16) are satisfied. Now compare the objective function (2.11) with the constraint (2.13) *coefficient-by-coefficient* (where we remember that the $x_i \geq 0$ in this example):

$$
\begin{array}{ccccc}
1\cdot x_1 & + & 1\cdot x_2 & & \\
\leq & & \leq & & \\
2\cdot x_1 & + & 1\cdot x_2 & \leq 10
\end{array}
$$

Thus for *every* feasible solution x we have the upper bound $x_1 + x_2 \leq 10$, i.e., the optimum value can be at most 10. Can we improve on this? We could try $\frac{7}{9}\cdot(2.13) + \frac{1}{9}\cdot(2.14)$:

$$
\begin{array}{ccccc}
1\cdot x_1 & + & 1\cdot x_2 & & \\
\leq & & \leq & & \\
(\frac{7}{9}\cdot 2 + \frac{1}{9}\cdot(-5))x_1 & + & (\frac{7}{9}\cdot 1 + \frac{1}{9}\cdot 2)x_2 & \leq \frac{7}{9}\cdot 10 + \frac{1}{9}\cdot 2 = \frac{72}{9} = 8
\end{array}
$$

Hence we have $x_1 + x_2 \leq 8$ for every feasible x and thus an upper bound of 8 on the optimum value. If we look closely, our choices 7/9 and 1/9 give $\frac{7}{9}\cdot 2 + \frac{1}{9}\cdot(-5) = 1$ and

$\frac{7}{9} \cdot 1 + \frac{1}{9} \cdot 2 = 1$, i.e., we have combined the coefficients of the objective function $c^\top x$ with *equality* (and need hence not argue that the x_i are non-negative).

This suggests the following approach. Combine the constraints with non-negative multipliers $y = (y_1, y_2, y_3)$ such that each coefficient in the result equals the corresponding coefficient in the objective function, i.e., we want $y^\top A = c^\top$. We associate y_1 with (2.12), y_2 with (2.13), y_3 with (2.14). Notice that the y_i are sign constrained because we are multiplying an *inequality* of the system $Ax \leq b$, i.e., if a multiplier y_i is negative we have to change the corresponding inequality from $\leq$ to $\geq$, but then we can not compare it to the other inequalities any more. Now

$$y_1(2.12) + y_2(2.13) + y_3(2.14)$$

evaluates to

$$\begin{aligned}
&y_1(4x_1 + (-1)x_2) + y_2(2x_1 + x_2) + y_3(-5x_1 + 2x_2) = \\
&(4y_1 + 2y_2 + (-5)y_3)x_1 + (-y_1 + y_2 + 2y_3)x_2 \leq 8y_1 + 10y_2 + 2y_3
\end{aligned}$$

and we want to find values for $y_1, y_2, y_3 \geq 0$ that satisfy:

$$\begin{array}{cccc}
1 \cdot x_1 & + & 1 \cdot x_2 & \\
= & & = & \\
(4y_1 + 2y_2 + (-5)y_3)x_1 & + & (-y_1 + y_2 + 2y_3)x_2 & \leq 8y_1 + 10y_2 + 2y_3.
\end{array}$$

Of course, we are interested in the best choice for $y = (y_1, y_2, y_3) \geq 0$ the approach can give. This means that we want to minimize the upper bound $8y_1 + 10y_2 + 2y_3$. We simply write down this task as a mathematical program.

$$\begin{aligned}
\text{minimize} \quad & 8y_1 + 10y_2 + 2y_3 && (2.17)\\
\text{subject to} \quad & 4y_1 + 2y_2 - 5y_3 = 1 && (2.18)\\
& -y_1 + y_2 + 2y_3 = 1 && (2.19)\\
& y_1, y_2, y_3 \geq 0 && (2.20)
\end{aligned}$$

Further note that the new objective function is the right hand side $(8, 10, 2)^\top$ of the original LP and that the new right hand side is the objective function $(1, 1)^\top$ of the original LP. Thus the above LP is of the form

$$\min\{y^\top b : y^\top A = c^\top, y \geq 0\}.$$

This should explain the special choice of the LPs (2.7) and (2.8). The rationale is that we can find upper bounds for the maximization LP by combining its constraints using multipliers.

Recall from Section 2.1 that there is a feasible solution $x = (2, 6)^\top$ that gives $c^\top x = 8$. And we have found multipliers $y = (0, 7/9, 1/9)^\top$ such that $y^\top b = 8$, i.e.,

$$c^\top x = y^\top b.$$

Hence we have a certificate that the solution $x = (2, 6)$ is indeed optimal (because we have a matching upper bound). Not surprisingly this is no exception but the principal statement of the *strong duality* theorem.

Strong Duality

Given an LP $P = \max\{c^\top x : Ax \leq b\}$ called *primal*, we define the *dual* to be $D = \min\{y^\top b : y^\top A = c^\top, y \geq 0\}$.

Lemma 2.10. *The dual of the dual of an LP is (equivalent to) the original LP.*

Now we can say that the LPs P and D are dual to each other or a *primal-dual pair*. The following *strong duality* theorem is the most important result in LP theory and the basis for a lot of algorithms for COPs.

Theorem 2.11 (Strong Duality)**.** *For any primal-dual pair $P = \max\{c^\top x : Ax \leq b\}$ and $D = \min\{y^\top b : y^\top A = c^\top, y \geq 0\}$ we have:*

(1) If P has an optimum solution x, say, then D also has an optimum solution y and

$$c^\top x = y^\top b.$$

(2) If P is unbounded, then D is infeasible.

(3) If P is infeasible, then D is infeasible or unbounded.

Proof. For (1) let us assume for the moment that P and D have both optimum solutions. We show that the optimal objective values of P and D are equal. By assumption, we have that D has a feasible vertex. Now run SIMPLEX on the LP D and this initial vertex. By Lemma 2.8, the (assumed) existence of some feasible x for P, shows that D is not unbounded. Thus, by Theorem 2.9, SIMPLEX returns optimum solutions y and z for D and its dual. However, this dual is equivalent to P and we have $y^\top b = c^\top x$ as claimed.

We still have to show that if P has an optimum solution then also D has an optimum solution. We use:

Lemma 2.12 (Farkas Lemma)**.** *There is a vector*

$$\begin{cases} x & \text{with } Ax \leq b \\ x \geq 0 & \text{with } Ax \leq b \\ x \geq 0 & \text{with } Ax = b \end{cases} \quad \text{if and only if } y^\top b \geq 0 \text{ for all} \quad \begin{cases} y \geq 0 & \text{with } y^\top A = 0 \\ y \geq 0 & y^\top A \geq 0 \\ y & \text{with } y^\top A \geq 0 \end{cases}.$$

Proof. For the first case, if there is a vector x with $Ax \leq b$, then $y^\top b \geq y^\top Ax = 0$ for each $y \geq 0$ with $y^\top A = 0$. Now consider the LP $\min\{1^\top w : Ax - w \leq b, w \geq 0\}$ which is equivalent to

$$\max \left\{ \begin{pmatrix} 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ w \end{pmatrix} : \begin{pmatrix} A & -I \\ 0 & -I \end{pmatrix} \begin{pmatrix} x \\ w \end{pmatrix} \leq \begin{pmatrix} b \\ 0 \end{pmatrix} \right\}.$$

The dual of this is $\min\{y^\top b : y^\top A = 0, 0 \leq y \leq 1\}$. Both LPs are feasible for example for $x = 0, w = |b|$, and $y = 0$ and we may apply what we have proved above and infer that $\min\{1^\top w : Ax - w \leq b, w \geq 0\} = \min\{y^\top b : y^\top A = 0, 0 \leq y \leq 1\}$. The system $Ax \leq b$ is feasible if and only if the optimum value of $\min\{1^\top w : Ax - w \leq b, w \geq 0\} = 0$. Thus the first case is proved.

For the second case apply the first to the system $(A, -I)^\top x \leq (b, 0)^\top$. For the third use the system $(A, -A)^\top x \leq (b, -b)^\top$. ☐

Now assume that the primal LP $P = \max\{c^\top x : Ax \leq b\}$ has an optimum solution x but the dual $D = \min\{y^\top b : y^\top A = c^\top, y^\top \geq 0\}$ is infeasible. (It can not be unbounded due to Lemma 2.8.) As D is infeasible there is no vector $y \geq 0$ with $y^\top A = c^\top$. But now we apply Lemma 2.12 and obtain that there must be a vector z with $Az \geq 0$ and $c^\top z < 0$. But now $x - z$ is feasible for P since $A(x-z) = Ax - Az \leq b$ and $c^\top(x-z) > c^\top x$ contradicts the optimality of x.

The claim (2) directly follows from Lemma 2.8. The claim (3) is just an alternative formulation for (1) and stated in the theorem for clarity. □

The theorem has a lot of implications but we only list two of them. The first (obvious) one is called *complementary slackness* (and gives another way of proving optimality).

Corollary 2.13. *Let* $\max\{c^\top x : Ax \leq b\}$ *and* $\min\{y^\top b : y^\top A = c, y \geq 0\}$ *be a primal-dual pair and let* x *and* y *be respective feasible solutions. Then the following statements are equivalent:*

(1) x *and* y *are both optimum solutions.*

(2) $c^\top x = y^\top b$.

(3) $y^\top(b - Ax) = 0$.

Furthermore, the fact that a system $Ax \leq b$ is infeasible can be proved by giving a vector $y \geq 0$ with $y^\top A = 0$ and $y^\top b < 0$.

Corollary 2.14. *There is a vector* x *with* $Ax \leq b$ *if and only if* $y^\top b \geq 0$ *for each vector* $y \geq 0$ *for which* $y^\top A = 0$.

Chapter 3

Network Flows

Flow problems are among the best-understood problems in combinatorial optimization. They are rather important because of their numerous applications.

3.1 Maximum Flows and Minimum Cuts

A *network* is a (simple) digraph $G = (V, A)$ where each edge has a *capacity* $c : A \to \mathbb{R}^+$ and we have two distinguished vertices, the *source* s and the *sink* t. Thus we often write $N = (G, c, s, t)$.

The maximum flow problem asks to transport as many units from the source to the sink without violating the edge capacities. More precisely, a *flow* f is a function $f : A \to \mathbb{R}^+$ which respects the edge capacities, i.e., $0 \leq f(e) \leq c(e)$ for all $e \in A$. We call f *flow conserving* at a vertex v if

$$\mathrm{ex}_f(v) := \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} = 0.$$

Here $\delta^-(v)$ is the set of *incoming edges* of v, i.e., $\delta^-(v) = \{uv \in A : u \in V\}$ and $\delta^+(v)$ the set of *outgoing* edges of v, i.e., $\delta^+(v) = \{vu \in A : u \in V\}$. The quantity $\mathrm{ex}_f(v)$ is the *excess* of v. An $s-t$*-flow* in a network N is a flow f with $\mathrm{ex}_f(s) \leq 0$ and $\mathrm{ex}_f(v) = 0$ for all $v \in V - \{s, t\}$. Its *value* is defined by $\mathrm{val}(f) = -\mathrm{ex}_f(s)$. See Figure 3.1

Problem 3.1 MAXIMUM FLOW

Instance. A network $N = (G, c, s, t)$.

Task. Find an $s-t$-flow of maximum value in N.

We can formulate the maximum flow problem as an LP in the variables f_e for $e \in A$.

$$\begin{aligned}
\text{maximize} \quad & \sum_{e \in \delta^+(s)} f_e - \sum_{e \in \delta^-(s)} f_e, \\
\text{subject to} \quad & \sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e = 0 \quad v \in V - \{s, t\}, \\
& f_e \leq c(e) \quad e \in A, \\
& f_e \geq 0.
\end{aligned}$$

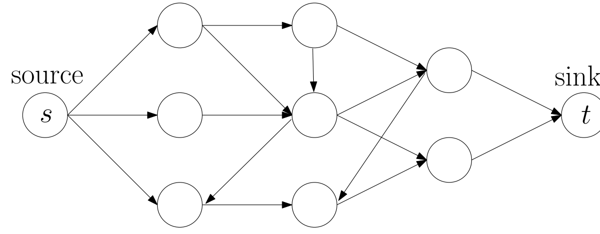


Figure 3.1: A network with source s and sink t.

Since the flow $f = 0$ is feasible for this LP, and the LP is obviously bounded (by $\sum_{e\in\delta^+(s)} c(e)$) we have that the MAXIMUM FLOW problem always has an optimum solution. Of course, we can solve the problem by using SIMPLEX but we are not satisfied with this – we want a combinatorial algorithm (not solving an LP) with guaranteed polynomial running time.

Let S be a subset of the vertices. The induced *cut* is the set of directed edges $\delta^+(S) = \{uv \in A : u \in S, v \in V - S\}$; denote $\delta^-(S) = \{vu \in A : u \in S, v \in V - S\}$. Its *capacity* $\text{cap}(S) = \sum_{e\in\delta^+(S)} c(e)$. An $s-t$-cut is a cut so that $s \in S$ and $t \in V - S$. A *minimum cut* refers to one with minimal capacity among all $s-t$-cuts.

The following result tells us that the value of a flow can be expressed through the incoming and outcoming flow of an arbitrary cut. Furthermore, the value of any flow (including the maximum one) is bounded from above by the capacity of any cut. Notice the similarity to weak duality. Not surprisingly, a strong duality result also holds for network flows. We will see below that the value of a maximum flow equals the capacity of a minimum cut.

Lemma 3.1. *For any $s-t$-cut S and any $s-t$-flow f we have that*

(1) $\text{val}(f) = \sum_{e\in\delta^+(S)} f(e) - \sum_{e\in\delta^-(S)} f(e)$,

(2) $\text{val}(f) \le \sum_{e\in\delta^+(S)} c(e) = \text{cap}(S)$.

Proof. We use the flow conservation property which holds for all $v \in S - \{s\}$ to find

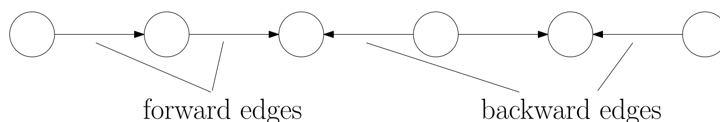
$$\begin{aligned}\text{val}(f) &= \sum_{e\in\delta^+(s)} f(e) - \sum_{e\in\delta^-(s)} f(e) = \sum_{v\in S}\left(\sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e)\right)\\ &= \sum_{e\in\delta^+(S)} f(e) - \sum_{e\in\delta^-(S)} f(e).\end{aligned}$$

Furthermore we have $\text{val}(f) \le \text{cap}(S)$ since $0 \le f(e) \le c(e)$. □

The following definition and structural result are the basis for an algorithm. A path P from s to a vertex v is called *f-augmenting* with respect to a flow f if

(1) $f(e) < c(e)$ for every *forward edge*, i.e., $e = uw \in A$ such that $uw \in P$,

(2) $f(e) > 0$ for every *backward edge*, i.e., $e = uw \in A$ such that $wu \in P$.



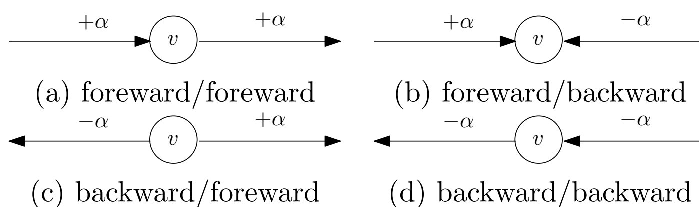
By how much can we increase the current flow value using a particular augmenting path P? Define the quantity

$$\alpha = \min\{c(e) - f(e) : e \text{ foreward edge in } P\} \cup \{f(e) : e \text{ backward edge in } P\}.$$

The following construction of a new flow f' is called *augmenting* f by α along P. Set $f'(e) = f(e) + \alpha$ if e is foreward edge in P, $f'(e) = f(e) - \alpha$ if e is backward edge in P, and $f'(e) = f(e)$ otherwise.

Observation 3.2. *The function f' defines a flow.*

Proof. By definition of the quantity α we have that $0 \leq f'(e) \leq c(e)$ for all $e \in A$. It remains to show that f' is flow conserving. It is clear that $\text{ex}_{f'}(s) \leq \text{ex}_f(s) \leq 0$. For any vertex $v \in V - \{s, t\}$ we distinguish four cases:



This yields the claim. □

Algorithm 3.1 FORD-FULKERSON

Input. Network $N = (G, c, s, t)$ with $c : A \to \mathbb{N}$.

Output. $s - t$-flow f of maximum value.

Step 1. Set $f(e) = 0$ for all $e \in A$.

Step 2. Find an f-augmenting path P. If none exists then return f.

Step 3. Compute

$$\alpha = \min\{c(e) - f(e) : e \text{ foreward edge in } P\} \cup \{f(e) : e \text{ backward edge in } P\}.$$

and augment f by α along P. Go to Step 2.

Theorem 3.3. *In a network N, the maximum value of an $s - t$-flow equals the minimum capacity of an $s - t$-cut.*

Proof. We show that an $s - t$-flow f has maximum value if and only if there is no f-augmenting path from s to t. In that case we will be able to find a minimum cut R with equal capacity.

Let there be an f-augementing path P from s to t, let α be as above and obtain f' by augmenting f by α along P. Observe that $\text{val}(f') > \text{val}(f)$, i.e., that f is not maximal.

Now let there be no f-augmenting path from s to t. Consider the set S of vertices with augmenting paths from s, i.e., $S = \{v \in V : \text{there is an } f\text{-augmenting path from } s \text{ to } v\}$, i.e., $t \notin S$. By definition of augmenting paths, we must have $f(e) = c(e)$ for all $e \in \delta^+(S)$ and $f(e) = 0$ for all $e \in \delta^-(S)$. Hence, using Lemma 3.1 (1), we have $\text{val}(f) = \sum_{e \in \delta^+(S)} c(e) = \text{cap}(S)$. By Lemma 3.1 (2) f must be a maximum flow and S be a minimum cut. □

If all capacities are integers then α is an integer and the algorithm terminates after a finite number of iterations. Thus we obtain the following important consequence:

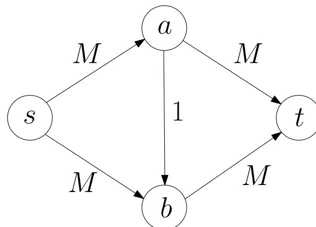
Corollary 3.4. *If the capacities of a network N are integers, then there is an integral maximum flow.*

It is an exercise to show the following *flow decomposition* result.

Theorem 3.5. *Given a network $N = (G, c, s, t)$ and an $s-t$-flow f then there is a familiy $\mathcal{P}$ of simple paths, a famlily $\mathcal{C}$ of simple cycles and positive numbers $h : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}^+$ such that* $\operatorname{val}(f) = \sum_{T \in \mathcal{P} \cup \mathcal{C}} h(T)$ *and* $|\mathcal{P}| + |\mathcal{C}| \leq |A|$.

We have not yet specified how we actually choose the augmenting paths mentioned in Step 2 of the algorithm Ford-Fulkerson. This must be done carefully in order to obtain a polynomial time algorithm as the following instance illustrates.

Example 3.6. To show that Ford-Fulkerson is not a polynomial time algorithm consider the following network. Here M is a large number.



Alternatingly augmenting one unit of flow along the paths s-a-b-t and s-b-a-t requires $2M$ augmentations. This is already exponential because the (binary) input size of the graph is $O(\log M)$. In contrast the augmenting paths s-a-t and s-b-t already give a maximum flow after two augmentations.

3.2 Edmonds-Karp Algorithm

Example 3.6 suggests that it may be a good idea to always choose shortest augmenting paths, i.e., with minimum number edges. Indeed, the algorithm Edmonds-Karp below uses this strategy and yields polynomial running time.

Algorithm 3.2 Edmonds-Karp

Input. Network $N = (G, c, s, t)$ with $c : A \to \mathbb{N}$.

Output. $s - t$-flow f of maximum value.

Step 1. Set $f(e) = 0$ for all $e \in A$.

Step 2. Find a shortest f-augmenting path P w.r.t. the number of edges. If none exists then return f.

Step 3. Compute α as above and augment f by α along P. Go to Step 2.

Theorem 3.7. *The algorithm* Edmonds-Karp *computes a maximum $s-t$-flow f in any network N with n vertices and m edges in time $O(nm^2)$.*

The following lemma is crucial for the proof of the worst-case running time. Let $f_0, f_1, f_2, \ldots$ be the flows constructed by the algorithm. Denote the shortest length of an augmenting path from s to a vertex v with respect to f_k by $x_v(k)$ and respectively from v to t by $y_v(k)$.

Lemma 3.8. *We have that*

(1) $x_v(k+1) \geq x_v(k)$ *for all* k *and* v*,*

(2) $y_v(k+1) \geq y_v(k)$ *for all* k *and* v*.*

Proof. Suppose for the sake of contradiction that (1) is violated for some pair (v, k). We may assume that $x_v(k+1)$ is minimal among the $x_w(k+1)$ for which (1) does not hold.

Let e be the last edge in a shortest augmenting path from s to v with respect to f_{k+1}. Suppose $e = uv$ is a forward edge. Hence $f_{k+1}(e) < c(e)$, $x_v(k+1) = x_u(k+1) + 1$, and $x_u(k+1) \geq x_u(k)$ by our choice of $x_v(k+1)$. Thus $x_v(k+1) \geq x_u(k) + 1$. Suppose that $f_k(e) < c(e)$ which yields $x_v(k) \leq x_u(k) + 1$ and thus $x_v(k+1) \geq x_v(k)$, a contradiction.

Hence we must have $f_k(e) = c(e)$ which implies that e was a backward edge when f_k was changed to f_{k+1}. As we used an augmenting path of shortest length we have $x_u(k) = x_v(k) + 1$ and thus $x_v(k+1) - 1 = x_u(k+1) \geq x_u(k) \geq x_v(k) + 1$. Hence $x_v(k+1) \geq x_v(k) + 2$ yields a contradiction.

Similarly when e is a backward edge. The proof of (2) is analogous to (1). □

Proof of Theorem 3.7. When we increase the flow, the augmenting path always contains a *critical* edge, i.e., an edge where the flow is either increased to meet the capacity or reduced to zero.

Let $e = uv$ be critical in the augmenting path w.r.t. f_k. This path has $x_v(k) + y_v(k) = x_u(k) + y_u(k)$ edges. If e is used the next time in an augmenting path w.r.t. f_h, say, then it must be used in the opposite direction as w.r.t. f_k.

Suppose that $e = uv$ was a forward edge w.r.t. f_k. Then $x_v(k) = x_u(k) + 1$ and $x_u(h) = x_v(h) + 1$. By Lemma 3.8 $x_v(h) \geq x_v(k)$ and $y_u(h) \geq y_u(k)$. Hence $x_u(h) + y_u(h) = x_v(h) + 1 + y_u(h) \geq x_v(k) + 1 + y_u(k) \geq x_u(k) + y_u(k) + 2$. Thus the augmenting path w.r.t. f_h is at least two edges longer than the augmenting path w.r.t. f_k. Similarly if e is a backward edge.

No shortest augmenting path can contain more than $n - 1$ edges and hence each edge can be critical at most $(n-1)/2$ times. As each augmenting path contains at least one critical edge, there can be at most $O(nm)$ augmentations and each one takes time $O(m)$. This yields the running time of $O(nm^2)$. □

There are further algorithms that solve the Maximum Flow problem in less time. For example the Goldberg-Tarjan algorithm runs in time $O\left(n^2\sqrt{m}\right)$; with sophisticated implementations $O\left(nm\log(n^2/m)\right)$ and $O\left(\min\{m^{1/2}, n^{2/3}\}m\log(n^2/m)\log c_{\max}\right)$ can be reached.

3.3 Minimum Cost Flows

In this section we treat a more general problem than the Maximum Flow problem, namely the Minimum Cost Flow problem. We are again given a digraph $G = (V, A)$ with edge capacities $c : A \to \mathbb{R}^+$ and in addition to that a weight function $w : A \to \mathbb{R}^+$ indicating the *cost* of an edge.

Now we define a modified notion of a flow. Let $b : V \to \mathbb{R}$ be given such that $\sum_{v \in V} b(v) = 0$. The value $b(v)$ is called the *balance* of a vertex v; if $b(v) < 0$ then v is called a *source*, if $b(v) > 0$ a *sink*. A *b-flow* in N is a function $f : A \to \mathbb{R}$ such that $0 \leq f(e) \leq c(e)$ for all $e \in A$ and $\text{ex}_f(v) = \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) = b(v)$. A 0-flow is called a *circulation*.

The *cost* of any flow f is

$$\text{val}(f) = \sum_{e \in A} f(e)w(e).$$

Now the problem is to find a b-flow with minimum cost.

Problem 3.2 MINIMUM COST FLOW

Instance. A network $N = (G, c, w, b)$.

Task. Find an b-flow of minimum cost in N or decide that none exists.

The second part of our task is easy. Given a network $N = (G, c, w, b)$ with balance vector b, we can decide if a b-flow exists by solving a MAXIMUM FLOW problem: Add two vertices s and t and edges sv, vt with capacities $c(sv) = \max\{0, b(v)\}$ and $c(vt) = \max\{0, -b(v)\}$ for all $v \in V$ to N. Then any $s-t$-flow with value $\sum_{v \in V} c(sv)$ in the resulting network corresponds to a b-flow in the original network N.

For the remainder of the section we give an optimality criterion which leads directly to an algorithm similar to the FORD-FULKERSON method. But here we augment along cycles instead of paths. Again, the choice of the augmenting cycles must be done carefully. But we omit this here and state the following theorem which refers to ORLIN's algorithm without proof.

Theorem 3.9. *There is an algorithm which solves the* MINIMUM COST FLOW *problem on any network with n vertices and m edges in time $O(m \log m(m + n \log n))$.*

We begin our discussion of an optimality criterion with a definition. Given a digraph $G = (V, A)$ with capacities c, weights w, and a flow f in G, construct the graph $R = (V, A + A_R)$ with $A_R = \{wv : vw \in A\}$, where $r \in A_R$ is called a *reverse* edge. (The notation "+" here means that we actually allow parallel edges in R). The *residual capacities* $c_R : A + A_R \to \mathbb{R}^+$ are $c_R(vw) = c(vw) - f(vw)$ for $vw \in A$ and $c_R(wv) = f(vw)$ for $wv \in A_R$. The *residual weight* $w_R : A \to \mathbb{R}$ is $w_R(vw) = w(vw)$ for $vw \in A$ and $w_R(wv) = -w(vw)$ for $wv \in A_R$. Finally define the *residual graph* $G_f = (V, A_f)$ with $A_f = \{e \in A + A_R : c_R(e) > 0\}$.

Now, given a digraph G with capacities c and a b-flow f, an *f-augmenting cycle* is a simple cycle in G_f. The following theorem is an optimality criterion for the MINIMUM COST FLOW problem.

Theorem 3.10. *Let $N = (G, c, w, b)$ be an instance of the* MINIMUM COST FLOW *problem. A b-flow f is of minimum cost if and only if there is no f-augmenting cycle with negative total cost.*

We prove the theorem in two steps. First we show that the difference between any two b-flows gives rise to a circulation and second that this circulation can be decomposed into circulations on simple cycles.

Lemma 3.11. *Let G be a digraph with capacities c and let f and f' be b-flows in (G, c). Construct R and G_f as above and define $g : A + A_R \to \mathbb{R}^+$ by $g(e) = \max\{0, f'(e) - f(e)\}$*

for $e \in A$ *and* $g(e) = \max\{0, f(e) - f'(e)\}$ *for all* $e \in A_R$*. Then* g *is a circulation in* R*,* $g(e) = 0$ *for all* $e \notin A_f$ *and* $\mathrm{val}(g) = \mathrm{val}(f') - \mathrm{val}(f)$*.*

Proof. At each vertex $v \in R$ we have

$$\sum_{e \in \delta_R^+(v)} g(e) - \sum_{e \in \delta_R^-(v)} g(e) = \sum_{e \in \delta_G^+(v)} (f'(e) - f(e)) - \sum_{e \in \delta_G^-(v)} (f'(e) - f(e))$$
$$= b(v) - b(v) = 0.$$

so g is a circulation in R.

For $e \notin A_f$ consider two cases: If $e \in A$ then $f(e) = c(e)$ and hence $f'(e) \leq f(e)$ which gives $g(e) = 0$. If $e = wv \in A_R$ then $e' = vw \in A$ and $f(e') = 0$ which yields $f(e) = 0$.

We verify the last statement

$$\mathrm{val}(g) = \sum_{e \in A + A_R} w(e) g(e) = \sum_{e \in A} w(e) f'(e) - \sum_{e \in A} w(e) f(e) = \mathrm{val}(f') - \mathrm{val}(f)$$

and the proof is complete. □

Lemma 3.12. *For any circulation* f *in a digraph* $G = (V, A)$ *there is a familiy* $\mathcal{C}$ *of at most* $|A|$ *simple cycles in* G *and for each* $C \in \mathcal{C}$ *a positive number* $h(C)$ *such that* $f(e) = \sum_{C \in \mathcal{C} : e \in C} h(C)$*.*

Proof. Follows from Theorem 3.5. □

Proof of Theorem 3.10. If there is an f-augmenting cycle C with weight $\gamma < 0$, we can augment f along C by some $\alpha > 0$ and get a b-flow f' with cost decreased by $-\gamma\alpha$. So f is not a mimimum cost flow.

If f is not a minimum cost b-flow, there is another b-flow f' with smaller cost. Consider g as defined in Lemma 3.11 and observe that g is a circulation with $\mathrm{val}(g) < 0$. By Lemma 3.12, g can be decomposed into flows on simple cycles. Since $g(e) = 0$ for all $e \notin A_f$, all these cycles are f-augmenting and one of them must have negative total cost. □

3.4 Assignment Problem

A graph $G = (V, E)$ with vertex set $V = L \cup R$ ("left" and "right") is called *bipartite* if the edge set satisfies $E \subseteq \{\ell r : \ell \in L, r \in R\}$. An *assignment* (also called a *matching*) is a subset $M \subseteq E$ such that for every $v \in V$ in the graph $H = (V, M)$ we have $\deg_H(v) \leq 1$. A matching is called *perfect* if $\deg_H(v) = 1$ for every $v \in V$. (Of course, a necessary condition for the existence of a perfect matching in a bipartite graph is $|L| = |R|$.)

The Assignment Problem has numerous applications and refers to the following. We are given a bipartite graph $G = (L \cup R, E)$ and a weight function $w : E \to \mathbb{R}$. We are asked to find a subset $M \subseteq E$ with minimum total weight, i.e.,

$$\mathrm{val}(M) = \sum_{e \in M} w(e),$$

such that M is a perfect matching or to conclude that no such matching exists.

Theorem 3.13. *The* Assignment *problem is a* Minimum Cost Flow *problem.*

Problem 3.3 ASSIGNMENT

Instance. Bipartite graph $G = (L \cup R, E)$ and a weight function $w : E \to \mathbb{R}$.

Task. Find perfect matching M with minimum weight $\mathrm{val}(M) = \sum_{e \in M} w(e)$ or conclude that no such matching exists.

Proof. Let $G = (V, E)$ be a bipartite graph with $V = L \cup R$ and $|L| = |R| = n$. Now we construct a network N for the MINIMUM COST FLOW problem. We start with the vertices V, add a vertex s and connect it with every vertex $\ell \in L$ with directed edges $s\ell$. Further add a vertex t and introduce the directed edges rt for every $r \in R$. Further add directed versions of all edges $e \in E$, i.e., a directed edge ℓr is added for every undirected edge ℓr. The capacities of all these edges is one. The weights of the $s\ell$ edges and the rt edges are zero – the weights of the ℓr edges are equal to their weights in G. Finally add a directed edge ts with infinite capacity and zero weight.

Now every integral 0-flow f in N with $f(ts) = n$ corresponds to a perfect matching in G with the same weight, and vice versa. □

Below we give several applications of the ASSIGNMENT problem. In most applications the requirement $|L| = |R|$ is disturbing, but can usually be handeled by adding artificial vertices and edges.

Bipartite Cardinality Matching

In the BIPARTITE CARDINALITY MATCHING problem we are given a bipartite graph $G = (V, E)$ with $V = L \cup R$, where $|L| \leq |R|$. Our task is to find a matching with maximum number of edges. We construct a network similarly as before: we add vertices s and t and the directed edges $s\ell$ and rt for all $\ell \in L$ and $r \in R$. All these edges have capacity equal to one. Any integral $s-t$-flow of value k corresponds to a matching with k edges. Thus we have to solve a MAXIMUM FLOW problem.

Moving Costs

A company has opened up n new factories and wants to transfer n managers to these in a way that minimizes the total moving cost. A manager i incurs cost w_{ij} when moving to factory j. An optimal solution for this problem is obviously a perfect matching with minimum cost, i.e., we have a direct application of the ASSIGNMENT problem.

Scheduling on Parallel Machines

In the SCHEDULING ON PARALLEL MACHINES problem we are given m machines and n jobs, where job j takes time p_{ij} if assigned to machine i. The jobs assigned to any machine are scheduled in a certain order. The completion time of job j is denoted c_j and refers to the following: If job j is assigned to machine i then the times p_{ik} of the jobs k also assigned to machine i but scheduled before job j contribute to the completion time of j, i.e., $c_j = \sum_{k \text{ on } i,\, k \leq j} p_{ik}$ (where "$k \leq j$" means that job k is scheduled before job j). The objective is to minimize the total completion time $\sum_j c_j$. It is an exercise to show that this problem can be formulated as an ASSIGNMENT problem.

Part II

Approximation Algorithms

Chapter 4

Knapsack

This chapter is concerned with the KNAPSACK problem. This problem is of interest in its own right because it formalizes the natural problem of selecting items so that a given budget is not exceeded but profit is as large as possible. Questions like that often also arise as subproblems of other problems. Typical applications include: option-selection in finance, cutting, and packing problems.

In the KNAPSACK problem we are given a budget W and n items. Each item i comes along with a profit c_i and a weight w_i. We are asked to choose a subset of the items as to maximize total profit but the total weight not exceeding W.

Example 4.1. We are given an amount of W and we wish to buy a subset of n items and sell those later on. Each such item j has cost w_j but yields profit c_j. The goal is to maximize the total profit. Consider $W = 100$ and the following profit-cost table:

j	c_j	w_j
1	150	100
2	2	1
3	55	50
4	100	50

Our choice of purchased items must not exceed our capital W. Thus the feasible solutions are $\{1\}, \{2\}, \{3\}, \{4\}, \{2,3\}, \{2,4\}, \{3,4\}$. Which is the best solution? Evaluating all possibilities yields that $\{3,4\}$ gives 155 altogether which maximizes our profit.

Problem 4.1 KNAPSACK

Instance. Non-negative integral vectors $c \in \mathbb{N}^n, w \in \mathbb{N}^n$, and an integer W.

Task. Solve the problem

$$\begin{aligned} \text{maximize} \quad & \text{val}(x) = \sum_{j=1}^{n} c_j x_j, \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq W, \\ & x_j \in \{0,1\} \quad j = 1, \dots n. \end{aligned}$$

For an *item* j the quantity c_j is called its *profit*. The profit of a vector $x \in \{0,1\}^n$ is $\mathrm{val}(x) = \sum_{j=1}^n c_j x_j$.

The number w_j is called the *weight* of item j. The *weight* of a vector $x \in \{0,1\}^n$ is given by $\mathrm{weight}(x) = \sum_{j=1}^n w_j x_j$. In order to obtain a non-trivial problem we assume $w_j \leq W$ for all $j = 1, \ldots, n$ and $\sum_{j=1}^n w_j > W$ throughout.

KNAPSACK is NP-hard which means that "most probably", there is no polynomial time optimization algorithm for it. However, in Section 4.1 we derive a simple 1/2-approximation algorithm. In Section 4.3 we can even improve on this by giving a polynomial-time $1-\varepsilon$-approximation algorithm (for every fixed $\varepsilon > 0$).

4.1 Fractional Knapsack and Greedy

A direct relaxation of KNAPSACK as an LP is often referred to as the FRACTIONAL KNAPSACK problem:

$$\begin{aligned} \text{maximize} \quad & \mathrm{val}(x) = \sum_{j=1}^n c_j x_j, \\ \text{subject to} \quad & \sum_{j=1}^n w_j x_j \leq W, \\ & 0 \leq x_j \leq 1 \quad j = 1, \ldots, n. \end{aligned}$$

This problem is solvable in polynomial time quite easily. The proof of the observation below is left as an exercise.

Observation 4.2. *Let $c, w, \in \mathbb{N}^n$ be non-negative integral vectors with*

$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n}$$

and let

$$k = \min\left\{ j \in \{1, \ldots, n\} : \sum_{i=1}^j w_i > W \right\}.$$

Then an optimum solution for the FRACTIONAL KNAPSACK *problem is given by*

$$\begin{aligned} x_j &= 1 \quad \text{for } j = 1, \ldots, k-1, \\ x_j &= \frac{W - \sum_{i=1}^{k-1} w_i}{w_k} \quad \text{for } j = k, \text{ and} \\ x_j &= 0 \quad \text{for } j = k+1, \ldots, n. \end{aligned}$$

The ratio c_j/w_j is called the *efficiency* of item j. The item number k, as defined above, is called the *break item*.

Now we turn our attention back to the original KNAPSACK problem. We may assume that the items are given in non-increasing order of efficiency. Observation 4.2 suggests the following simple algorithm: $x_j = 1$ for $j = 1, \ldots, k-1$, $x_j = 0$ for $j = k, \ldots, n$.

Unfortunately, the approximation ratio of this algorithm can be arbitrarily bad as the example below shows. The problem is that more efficient items can "block" more profitable ones.

Example 4.3. Consider the following instance, where W is a sufficiently large integer.

j	c_j	w_j	c_j/w_j
1	1	1	1
2	$W-1$	W	$1-1/W$

The algorithm chooses item 1, i.e., the solution $x = (1, 0)$ and hence $\text{val}(x) = 1$. The optimum solution is $x^* = (0, 1)$ and thus $\text{val}(x^*) = W - 1$. The approximation ratio of the algorithm is $1/(W - 1)$, i.e., arbitrarily bad. However, this natural algorithm can be turned into a 1/2-approximation.

Algorithm 4.1 GREEDY

Input. Integer W, vectors $c, w \in \mathbb{N}^n$ with $w_j \le W$, $\sum_j w_j > W$, and $c_1/w_1 \ge \cdots \ge c_n/w_n$.

Output. Vector $x \in \{0, 1\}^n$ such that $\text{weight}(x) \le W$.

Step 1. Define $k = \min\{j \in \{1, \ldots, n\} : \sum_{i=1}^{j} w_i > W\}$.

Step 2. Let x and y be the following two vectors: $x_j = 1$ for $j = 1, \ldots, k - 1$, $x_j = 0$ for $j = k, \ldots, n$, and $y_j = 1$ for $j = k$, $y_j = 0$ for $j \ne k$.

Step 3. Return x if $\text{val}(x) \ge \text{val}(y)$; otherwise y.

Theorem 4.4. *The algorithm* GREEDY *is a* $1/2$*-approximation for* KNAPSACK.

Proof. The value obtained by the GREEDY algorithm is equal to $\max\{\text{val}(x), \text{val}(y)\}$.

Let x^* be an optimum solution for the KNAPSACK instance. Since every solution that is feasible for the KNAPSACK instance is also feasible for the respective FRACTIONAL KNAPSACK instance we have that

$$\text{val}(x^*) \le \text{val}(z^*),$$

where z^* is the respective optimum solution for FRACTIONAL KNAPSACK. Observe that it has the structure $z^* = (1, \ldots, 1, \alpha, 0, \ldots, 0)$, where $\alpha \in [0, 1)$ is at the break item k. The solutions x and y are $x = (1, \ldots, 1, 0, 0, \ldots, 0)$ and $y = (0, \ldots, 0, 1, 0, \ldots, 0)$.

In total we have

$$\text{val}(x^*) \le \text{val}(z^*) = \text{val}(x) + \alpha c_k \le \text{val}(x) + \text{val}(y) \le 2 \max\{\text{val}(x), \text{val}(y)\}$$

which implies the approximation ratio of 1/2. □

4.2 Pseudo-Polynomial Time Algorithm

Here we give a pseudo-polynomial time algorithm that solves KNAPSACK correctly by using dynamic programming.

The idea is the following: Suppose you restrict yourself to choose only among the first j items, for some integer $j \in \{0, \ldots, n\}$. So all the solutions x you consider have the form $x_i \in \{0, 1\}$ for $i = 1, \ldots, j$ and $x_i = 0$ for $i = j + 1, \ldots, n$. With abuse of notation write $x \in \{0, 1\}^j 0^{n-j}$. Now the variable $m_{j,k}$ equals the minimum total weight

of such a solution x with $\mathrm{weight}(x) \le W$ and $\mathrm{val}(x) = k$. That is, after defining the set $W_{j,k} = \{\mathrm{weight}(x) : \mathrm{weight}(x) \le W, \mathrm{val}(x) = k, x \in \{0,1\}^j 0^{n-j}\}$ we require

$$m_{j,k} = \inf W_{j,k}.$$

(Recall that for any finite set S of integers $\inf S = \min S$ if $S \neq \emptyset$ and $\inf S = \infty$, otherwise.)

Let C be any upper bound on the optimum profit, for example $C = \sum_i c_i$. Clearly, the value of an optimum solution for KNAPSACK is the largest value $k \in \{0, \ldots, C\}$ such that $m_{n,k} < \infty$. The algorithm DYNAMIC PROGRAMMING KNAPSACK recursively computes the values for $m_{j,k}$ and then returns the optimum value for the given KNAPSACK instance. In the algorithm below, the variables $x(j,k)$ are n-dimensional vectors that store the solutions corresponding to $m_{j,k}$, i.e., with weight equal to $m_{j,k}$ and value k.

Algorithm 4.2 DYNAMIC PROGRAMMING KNAPSACK

Input. Integers W, C, vectors $w, c \in \mathbb{N}^n$.

Output. Vector $x \in \{0,1\}^n$ such that $\mathrm{weight}(x) \le W$.

Step 1. Set $m_{0,0} = 0$, $m_{0,k} = \infty$ for $k = 1, \ldots, C$, and $x(0,0) = 0$.

Step 2. For $j = 1, \ldots, n$ and $k = 0, \ldots, C$ do

$$m_{j,k} = \begin{cases} m_{j-1,k-c_j} + w_j & \text{if } c_j \le k \text{ and } m_{j-1,k-c_j} + w_j \le \min\{W, m_{j-1,k}\}, \\ m_{j-1,k} & \text{otherwise.} \end{cases}$$

If the first case applied set $x(j,k)_i = x(j-1, k-c_j)_i$ for $i \neq j$ and $x(j,k)_j = 1$. Otherwise set $x(j,k) = x(j-1,k)$.

Step 3. Determine the largest $k \in \{0, \ldots, C\}$ such that $m_{n,k} < \infty$. Return $x(n,k)$.

Theorem 4.5. *The* DYNAMIC PROGRAMMING KNAPSACK *algorithm computes the optimum value of the* KNAPSACK *instance W, $w, c \in \mathbb{N}^n$ in time $O(nC)$, where C is an arbitrary upper bound on this optimum value.*

Proof. The running time is obvious. For the correctness we prove that the values $m_{j,k}$ computed by the algorithm satisfy

$$m_{j,k} = \inf W_{j,k}$$

by induction on j. Here $W_{j,k} = \{\mathrm{weight}(x) : \mathrm{weight}(x) \le W, \mathrm{val}(x) = k, x \in \{0,1\}^j 0^{n-j}\}$ by definition.

The base case $j = 0$ is clear. For the inductive case first consider a situation when the algorithm sets

$$m_{j,k} = m_{j-1,k-c_j} + w_j,$$

i.e. we "take" the j-th item. Let $y = x(j-1, k-c_j)$ be the solution that corresponds to $m_{j-1,k-c_j}$. The solution $x = x(j,k)$ that corresponds to $m_{j,k}$ is obtained from y by setting $x_i = y_i$ for $i \neq j$ and $x_j = 1$. The value of x is $\mathrm{val}(x) = k$. By definition of the algorithm we have $\mathrm{weight}(x) = \mathrm{weight}(y) + w_j = m_{j-1,k-c_j} + w_j \le W$ and thus $x \in W_{j,k}$.

By construction of the algorithm and induction hypothesis we have $\text{weight}(x) \leq \inf W_{j-1,k}$ and $\text{weight}(x) = w_j + \inf W_{j-1,k-c_j}$. That is, the weight of x is at most the weight of any solution without the j-th item and at most the weight of any solution including the j-th item. Hence $m_{j,k} = \inf W_{j,k}$.

In the other situation, when the algorithm sets

$$m_{j,k} = m_{j-1,k},$$

then either $c_j > k$ and hence no solution with value equal to k can contain the j-th item, or $m_{j-1,k} + w_j > W$, i.e., adding the j-th item is infeasible, or $m_{j-1,k} + w_j > \inf W_{j-1,k}$, i.e., there is a solution with less weight and still value equal to k. □

4.3 Fully Polynomial-Time Approximation Scheme

Here we give a *fully polynomial time approximation scheme* (FPTAS), i.e., we show that for every fixed $\varepsilon > 0$ there is an $1 - \varepsilon$-approximation algorithm that runs in time polynomial in the input size and $1/\varepsilon$. From a complexity-theoretic point of view this is the best that can be hoped for: Assuming $\mathsf{P} \neq \mathsf{NP}$ there is no polynomial time algorithm that solves Knapsack optimally on every instance, but the FPTAS delivers solutions with arbitrarily good approximation guarantees in polynomial time. (Unfortunately not many problems admit an FPTAS.)

A common theme in constructing FPTASs is the following: First find an algorithm that solves the problem exactly (mostly using the dynamic programming paradigm). This algorithm usually has pseudo-polynomial or even exponential running time. Second construct an algorithm for "rounding" input-instances, i.e., reducing the input-size. This modification reduces the running time but may lead to inaccurate solutions.

The running time of Dynamic Programming Knapsack is proportional to nC. If we divide we profit c_j of each item by a number t and round the result down, then this improves the running time of Dynamic Programming Knapsack by a factor of t but may yield suboptimal solutions.

Algorithm 4.3 Knapsack FPTAS

Input. Integer W, vectors $w, c \in \mathbb{N}^n$, a number $\varepsilon > 0$.

Output. Vector $x \in \{0,1\}^n$ such that $\text{weight}(x) \leq W$.

Step 1. Run Greedy on the instance W, w, c and let x be the solution. If $\text{val}(x) = 0$ then return x.

Step 2. Set $t = \max\{1, \varepsilon \text{val}(x)/n\}$ and set

$$c'_j = \left\lfloor \frac{c_j}{t} \right\rfloor \quad \text{for } j = 1, \dots, n.$$

Step 3. Set $C = 2\text{val}(x)/t$ and apply the Dynamic Programming Knapsack algorithm on the instance W, C, w, c' and let y be the solution obtained.

Step 4. If $\text{val}(x) \geq \text{val}(y)$ return x otherwise y.

Theorem 4.6. *For every fixed $\varepsilon > 0$, the* Knapsack FPTAS *algorithm is a $1-\varepsilon$-approximation algorithm with running time $O\left(n^2/\varepsilon\right)$.*

Proof. The value of the solution returned by the algorithm is equal to $\max\{\text{val}(x), \text{val}(y)\}$. Let x^* be an optimum solution for the instance W, w, c. By Theorem 4.4 we have $2\text{val}(x) \geq \text{val}(x^*)$ and hence the choice $C = 2\text{val}(x)/t$ is a legal upper bound for the optimum value of the rounded instance W, w, c'. By Theorem 4.5 y is an optimum solution for this instance and we have

$$\begin{aligned}\text{val}(y) &= \sum_j c_j y_j \geq \sum_j t c'_j y_j = t \sum_j c'_j y_j \\ &\geq t \sum_j c'_j x^*_j = \sum_j t c'_j x^*_j > \sum_j (c_j - t) x^*_j \geq \text{val}(x^*) - nt.\end{aligned}$$

If $t = 1$ then y is optimal by Theorem 4.5. Otherwise the above inequality and the choice of t yields $\text{val}(y) \geq \text{val}(x^*) - \varepsilon\text{val}(x)$ and hence

$$\text{val}(x^*) \leq \text{val}(y) + \varepsilon\text{val}(x) \leq (1+\varepsilon)\max\{\text{val}(x), \text{val}(y)\}$$

which yields the approximation guarantee $1 - \varepsilon/(1+\varepsilon)$.

The running time of Dynamic Programming Knapsack on the rounded instance is

$$O\left(nC\right) = O\left(\frac{n\text{val}(x)}{t}\right) = O\left(\frac{n^2}{\varepsilon}\right)$$

which dominates the time needed for the other steps. □

Chapter 5

Set Cover

The SET COVER problem this chapter deals with is again a very simple – yet quite general – NP-hard combinatorial problem. It is widely applicable in sometimes unexpected ways. The problem is the following: We are given a set U (called *universe*) of n elements, a collection of sets $\mathcal{S} = \{S_1, \ldots, S_k\}$ where $S_i \subseteq U$, and a cost function $c : \mathcal{S} \to \mathbb{R}^+$. The task is to find a minimum cost subcollection $\mathcal{S}' \subseteq \mathcal{S}$ that *covers* U, i.e., such that $\cup_{S \in \mathcal{S}'} S = U$.

Example 5.1. Consider this instance: $U = \{1, 2, 3\}$, $\mathcal{S} = \{S_1, S_2, S_3\}$ with $S_1 = \{1, 2\}$, $S_2 = \{2, 3\}$, $S_3 = \{1, 2, 3\}$ and cost $c(S_1) = 10$, $c(S_2) = 50$, and $c(S_3) = 100$. These collections cover U: $\{S_1, S_2\}$, $\{S_3\}$, $\{S_1, S_3\}$, $\{S_2, S_3\}$, $\{S_1, S_2, S_3\}$. The cheapest one is $\{S_1, S_2\}$ with cost equal to 60.

For each set S, we associate a variable $x_S \in \{0, 1\}$ that indicates of we want to choose S or not. We may thus write solutions for SET COVER as a vector $x \in \{0, 1\}^k$. With this, we write SET COVER as a mathematical program.

Problem 5.1 SET COVER

Instance. Universe U with n elements, collection $\mathcal{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq U$, a cost function $c : \mathcal{S} \to \mathbb{R}$.

Task. Solve the problem

$$\begin{aligned} \text{minimize} \quad & \text{val}(x) = \sum_{S \in \mathcal{S}} c(S) x_S, \\ \text{subject to} \quad & \sum_{S : e \in S} x_S \geq 1 \quad e \in U, \\ & x_S \in \{0, 1\} \quad S \in \mathcal{S}. \end{aligned}$$

Define the *frequency* of an element to be the number of sets it is contained in. Let f denote the frequency of the most frequent element. In this chapter we present several algorithms that either achieve approximation ratio $O(\log n)$ or f. Why are we interested in a variety algorithms? Is one algorithm not sufficient? Yes, but here the focus is on the *techniques* that yield these algorithms.

5.1 Greedy Algorithm

The GREEDY algorithm follows the natural approach of iteratively choosing the most cost-effective set and remove all the covered elements until all elements are covered. Let C be the set of elements already covered at the beginning of an iteration. During this iteration define the *cost-effectiveness* of a set S as $c(S)/|S - C|$, i.e., the average cost at which it covers new elements. For later reference, the algorithm sets the *price* at which it covered an element equal to the cost-effectiveness of the covering set. Further recall that $H_n = \sum_{i=1}^{n} 1/i$ is called the *n-th Harmonic number* and that $\log n \leq H_n \leq \log n + 1$.

Algorithm 5.1 GREEDY

Input. Universe U with n elements, collection $\mathcal{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq U$, a cost function $c : \mathcal{S} \to \mathbb{R}$.

Output. Vector $x \in \{0, 1\}^k$

Step 1. $C = \emptyset$, $x = 0$.

Step 2. While $C \neq U$ do the following:

(a) Find the most cost-effective set in the current iteration, say S.

(b) Let $\alpha = c(S)/|S - C|$.

(c) Set $x_S = 1$ and for each $e \in S - C$ set $\text{price}(e) = \alpha$.

(d) $C = C \cup S$.

Step 3. Return x.

Theorem 5.2. *The* GREEDY *algorithm is an H_n-approximation algorithm for the* SET COVER *problem.*

It is an exercise to show that this bound is tight.

Direct Analysis

The following lemma is crucial for the proof of the approximation-guarantee. Number the elements of U in the order in which they were covered by the algorithm, say $e_1, \ldots, e_n$. Let x^* be an optimum solution.

Lemma 5.3. *For each $k \in \{1, \ldots, n\}$,* $\text{price}(e_k) \leq \text{val}(x^*)/(n - k + 1)$.

Proof. In any iteration, the leftover sets of the optimal solution x^* can cover the remaining elements at a cost of at most $\text{val}(x^*)$. Therefore, among these, there must be one set having cost-effectiveness of at most $\text{val}(x^*)/|\overline{C}|$. In the iteration in which element e_k was covered, $\overline{C}$ contained at least $n - k + 1$ elements. Since e_k was covered by the most cost-effective set in this iteration, we have that

$$\text{price}(e_k) \leq \frac{\text{val}(x^*)}{|\overline{C}|} \leq \frac{\text{val}(x^*)}{n - k + 1}$$

which was claimed. $\square$

Proof of Theorem 5.2. Since the cost of each set is distributed evenly among the new elements covered, the total cost of the set cover picked is

$$\mathrm{val}(x) = \sum_{i=k}^{n} \mathrm{price}(e_k) \le \mathrm{val}(x^*) H_n,$$

where we have used Lemma 5.3. □

Dual-Fitting Analysis

Here we will give an alternative analysis of the Greedy algorithm for Set Cover. We will use the *dual fitting* method, which is quite general and helps to analyze a broad variety of combinatorial algorithms.

For sake of exposition we consider a minimization problem, but the technique works similarly for maximization. Consider an algorithm Alg which does the following:

(1) Let (P) be an integer programming formulation of the problem of interest. We are interested in its optimal solution x^*, respectively its objective value $\mathrm{val}(x^*)$. Let (D) be the dual of an linear programming relaxation of (P).

(2) The algorithm Alg computes a feasible solution x for (P) and a "solution" y for (D), where we allow that y is *infeasible* for (D). But the algorithm has to ensure that

$$\mathrm{val}(x) \le \overline{\mathrm{val}}(y),$$

where val is the objective function of (P) and $\overline{\mathrm{val}}$ is the objective function of (D).

(3) Now divide the entries of y by a certain quantity α until $y' = y/\alpha$ *is* feasible for (D). (The method of dual fitting is applicable only if this property can be ensured.) Then $\overline{\mathrm{val}}(y')$ is a lower bound for $\mathrm{val}(x^*)$ by weak duality, i.e.,

$$\overline{\mathrm{val}}(y') \le \mathrm{val}(x^*)$$

by Lemma 2.8.

(4) Putting these things together, we obtain the approximation guarantee of α by

$$\mathrm{val}(x) \le \overline{\mathrm{val}}(y) = \overline{\mathrm{val}}(\alpha y') = \alpha\overline{\mathrm{val}}(y') \le \alpha \mathrm{val}(x^*).$$

Now we apply this recipe to Set Cover and consider the Greedy algorithm. For property (1) we use our usual formulation

$$\begin{aligned} \text{minimize} \quad & \sum_{S \in \mathcal{S}} c(S) x_S, \\ \text{subject to} \quad & \sum_{S : e \in S} x_S \ge 1 \quad e \in U, \\ & x_S \in \{0,1\} \quad S \in \mathcal{S}. \end{aligned} \tag{P}$$

When we relax the constraints $x_S \in \{0,1\}$ to $0 \le x_S \le 1$ and dualize the corresponding linear program we find

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in U} y_e, \\ \text{subject to} \quad & \sum_{e \in S} y_e \le c(S) \quad S \in \mathcal{S}, \\ & y_e \ge 0. \end{aligned} \tag{D}$$

This dual can be derived purely mechanically (by applying the primal-dual-definition and rewriting constraints if needed) this program has an intuitive interpretation. The constraints of (D) state that we want to "pack stuff" into each set S such that the cost $c(S)$ of each set is not exceeded, i.e., the sets are not overpacked. We seek to maximize the total amount packed.

How about property (2)? The algorithm GREEDY computes a certain feasible solution x for (P), i.e., a solution $x_S = 1$ if the algorithm picks set S and $x_S = 0$ otherwise. What about the vector y? Define the following vector: For each $e \in U$ set $y_e = \text{price}(e)$, where $\text{price}(e)$ is the value computed during the execution of the algorithm.

By construction of the algorithm we have

$$\text{val}(x) = \sum_{S \in \mathcal{S}} c(S) x_S = \sum_{e \in U} \text{price}(e) = \sum_{e \in U} y_e = \overline{\text{val}}(y),$$

i.e., GREEDY satisfies property (2) of the dual fitting method (even with equality).

For property (3) the following result is useful.

Lemma 5.4. *For every $S \in \mathcal{S}$ we have that*

$$\sum_{e \in S} y_e \leq H_n c(S).$$

Proof. Let $S \in \mathcal{S}$ with, say, k elements. Consider these in the ordering the algorithm covered them, say, $e_1, \ldots, e_k$. At the iteration when e_i gets covered S contains $k - i + 1$ uncovered elements. Since GREEDY chooses the most cost-effective set we have that

$$\text{price}(e_i) \leq \frac{c(S)}{k - i + 1},$$

i.e., the cost-effectiveness of the set the algorithm chooses can only be smaller than the cost-effectiveness of S. (Be aware that "smaller" is "better" here.)

Summing over all elements gives

$$\sum_{i=1}^{k} y_e \leq c(S) \sum_{i=1}^{k} \frac{1}{k - i + 1} = c(S) H_k \leq c(S) H_n$$

as claimed. □

Now we are in position to finalize the dual-fitting analysis using property (4).

Proof of Theorem 5.2. Define the vector $y' = y / H_n$, where y is defined above. Observe that for each set $S \in \mathcal{S}$ we have

$$\sum_{e \in S} y'_e = \sum_{e \in S} \frac{y_e}{H_n} = \frac{1}{H_n} \sum_{e \in S} y_e \leq c(S)$$

using Lemma 5.4. That means y' is feasible for (D). Using the property (4) of the dual fitting method proves the approximation guarantee of at most H_n. □

5.2 Primal-Dual Algorithm

The primal-dual schema introduced here is the method of choice for designing approximation algorithms because it often gives algorithms with good approximation guarantees and good running times. After introducing the ideas behind the method, we will use it to design a simple factor f algorithm, where f is the frequency of the most frequent element.

The general idea is to work with an LP-relaxation of an NP-hard problem and its dual. Then the algorithm iteratively changes a primal and a dual solution until the relaxed primal-dual complementary slackness conditions are satisfied.

Primal-Dual Schema

Consider the following primal program:

$$\begin{aligned} \text{minimize} \quad & \mathrm{val}(x) = \sum_{j=1}^{n} c_j x_j, \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad i = 1, \ldots, m, \\ & x_j \geq 0 \quad j = 1, \ldots, n. \end{aligned}$$

The dual program is:

$$\begin{aligned} \text{maximize} \quad & \overline{\mathrm{val}}(y) = \sum_{i=1}^{m} b_i y_i, \\ \text{subject to} \quad & \sum_{i=1}^{m} a_{ij} y_i \leq c_j \quad j = 1, \ldots, n, \\ & y_i \geq 0 \quad i = 1, \ldots, m. \end{aligned}$$

Most known approximation algorithms using the primal-dual schema run by ensuring one set of conditions and suitably relaxing the other. We will capture both situations by relaxing both conditions. If primal conditions are to be ensured, we set $\alpha = 1$ below, and if dual conditions are to be ensured, we set $\beta = 1$.

Primal Complementary Slackness Conditions. Let $\alpha \geq 1$. For each $1 \leq j \leq n$:

$$\text{either} \quad x_j = 0 \quad \text{or} \quad c_j/\alpha \leq \sum_{i=1}^{m} a_{ij} y_i \leq c_j.$$

Dual Complementary Slackness Conditions. Let $\beta \geq 1$. For each $1 \leq i \leq m$:

$$\text{either} \quad y_i = 0 \quad \text{or} \quad b_i \leq \sum_{i=1}^{n} a_{ij} x_j \leq \beta b_i.$$

Lemma 5.5. *If x and y are primal and dual feasible solutions respectively satisfying the complementary slackness conditions stated above, then*

$$\mathrm{val}(x) \leq \alpha\beta\overline{\mathrm{val}}(y).$$

Proof. We calculate directly using the slackness conditions and obtain

$$\begin{aligned}\mathrm{val}(x) = \sum_{j=1}^{n} c_j x_j &\le \alpha \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j \\ &= \alpha \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j \right) y_i \le \alpha\beta \sum_{i=1}^{m} b_i y_i = \overline{\mathrm{val}}(y)\end{aligned}$$

which was claimed. □

The algorithm starts with a primal infeasible solution and a dual feasible solution; usually these are $x = 0$ and $y = 0$ initially. It iteratively improves the feasibility of the primal solution and the optimality of the dual solution ensuring that in the end a primal feasible solution is obtained and all conditions stated above, with a suitable choice for α and β, are satisfied. The primal solution is always extended integrally, thus ensuring that the final solution is integral. The improvements to the primal and the dual go hand-in-hand: the current primal solution is used to determine the improvement to the dual, and vice versa. Finally, the cost of the dual solution is used as a lower bound on the optimum value, and by Lemma 5.5, the approximation guarantee of the algorithm is $\alpha\beta$.

Primal-Dual Algorithm

Here we derive a factor f approximation algorithm for Set Cover using the primal-dual schema. For this algorithm we will choose $\alpha = 1$ and $\beta = f$. We will work with the following primal LP for Set Cover

$$\begin{aligned} \text{minimize} \quad & \mathrm{val}(x) = \sum_{S \in \mathcal{S}} c(S) x_S, \\ \text{subject to} \quad & \sum_{S : e \in S} x_S \ge 1 \quad e \in U, \\ & x_S \ge 0 \quad S \in \mathcal{S}. \end{aligned}$$

and its dual

$$\begin{aligned} \text{maximize} \quad & \overline{\mathrm{val}}(y) = \sum_{e \in U} y_e, \\ \text{subject to} \quad & \sum_{e \in S} y_e \le c(S) \quad S \in \mathcal{S}, \\ & y_e \ge 0 \quad e \in U. \end{aligned}$$

For these LPs the primal and dual complementary slackness conditions are:

Primal Complementary Slackness Conditions. For each $S \in \mathcal{S}$:

$$\text{either} \quad x_S = 0 \quad \text{or} \quad \sum_{e \in S} y_e = c(S).$$

A set S will be said to be *tight* if $\sum_{e \in S} y_e = c(S)$. So, this condition states that: "Pick only tight sets into the cover."

Dual Complementary Slackness Conditions. For each $e \in U$:

$$\text{either} \quad y_e = 0 \quad \text{or} \quad \sum_{S: e \in S} x_S \leq f.$$

Since we will find a 0/1 solution for x, these conditions are equivalent to: "Each element having non-zero dual value can be covered at most f times." Since each element is in at most f sets, this condition is trivially satisfied for all elements.

These conditions suggest the following algorithm:

Algorithm 5.2 PRIMAL-DUAL SET COVER

Input. Universe U with n elements, collection $\mathcal{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq U$, a cost function $c : \mathcal{S} \to \mathbb{R}$.

Output. Vector $x \in \{0,1\}^k$

Step 1. $x = 0$, $y = 0$. Declare all elements uncovered.

Step 2. Unless all elements are covered, do:

(a) Pick an uncovered element, say e, and raise y_e until some set goes tight.

(b) Pick all tight sets S in the cover, i.e., set $x_S = 1$.

(c) Declare all the elements occuring in these sets as covered.

Step 3. Return x.

Theorem 5.6. *The algorithm* PRIMAL-DUAL SET COVER *is a f-approximation algorithm for* SET COVER.

Proof. At the end of the algorithm, there will be no uncovered elements. Further no dual constraint is violated since we pick only tight sets S into the cover and no element $e \in S$ will later on be a candidate for increasing y_e. Thus, the primal and dual solutions will both be feasible. Since they satisfy the primal and dual complementary slackness conditions with $\alpha = 1$ and $\beta = f$, by Lemma 5.5, the approximation guarantee is f. □

Example 5.7. A tight example for this algorithm is provided by the following set system. The universe is $U = \{e_1, \ldots, e_{n+1}\}$ and $\mathcal{S}$ consists of $n-1$ sets $\{e_1, e_n\}, \ldots, \{e_{n-1}, e_n\}$ of cost 1 and one set $\{e_1, \ldots, e_{n+1}\}$ of cost $1 + \varepsilon$ for some small $\varepsilon > 0$. Since e_n appears in all n sets, this system has $f = n$.

Suppose the algorithm raises y_{e_n} in the first iteration. When y_{e_n} is raised to 1, all sets $\{e_i, e_n\}$, $i = 1, \ldots, n-1$ go tight. They are all picked in the cover, thus covering the elements $e_1, \ldots, e_n$. In the second iteration $y_{e_{n+1}}$ is raised to ε and the set $\{e_1, \ldots, e_{n+1}\}$ goes tight. The resulting set cover has cost $n + \varepsilon$, whereas the optimum cover has cost $1 + \varepsilon$.

5.3 LP-Rounding Algorithms

The central idea behind algorithms that make use of the *LP-rounding* technique is as follows: Suppose you have an LP-relaxation of a certain NP-hard problem. Then you can solve this optimally and try to "round" the optimal fractional solution to an integral one.

Here we derive a factor f approximation algorithm for SET COVER but this time by rounding the fractional solution of an LP to an integral solution (instead of the primal-dual schema). We consider our usual LP relaxation for SET COVER

$$\begin{aligned} \text{minimize} \quad & \text{val}(x) = \sum_{s \in \mathcal{S}} c(S) x_S, \\ \text{subject to} \quad & \sum_{S: e \in S} x_S \geq 1 \quad e \in U, \\ & x_S \geq 0 \quad S \in \mathcal{S}. \end{aligned}$$

Simple Rounding Algorithm

The idea of the algorithm below is to include those sets S into the cover for which the corresponding value z_S in the optimal solution z of the LP is "large enough".

Algorithm 5.3 SIMPLE ROUNDING SET COVER

Input. Universe U with n elements, collection $\mathcal{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq U$, a cost function $c : \mathcal{S} \to \mathbb{R}$.

Output. Vector $x \in \{0, 1\}^k$

Step 1. Set $x = 0$, solve the LP relaxation below, and call the optimal solution z.

$$\begin{aligned} \text{minimize} \quad & \text{val}(x) = \sum_{S \in \mathcal{S}} c(S) x_S, \\ \text{subject to} \quad & \sum_{S: e \in S} x_S \geq 1 \quad e \in U, \\ & x_S \geq 0 \quad S \in \mathcal{S}. \end{aligned}$$

Step 2. For each set S set $x_S = 1$ if $z_S \geq 1/f$.

Step 3. Return x.

Theorem 5.8. *The algorithm* SIMPLE ROUNDING SET COVER *is an f-approximation algorithm for* SET COVER.

Proof. Let x be the solution returned by the algorithm and z be the optimal solution of the LP. Consider an arbitrary element $e \in U$. Since e is in at most f sets, one of these sets must be picked to the extent of at least $1/f$ in the fractional solution z. Thus e is covered due to the definition of the algorithm and x is hence a feasible cover. We further have $x_S \leq f z_S$ and thus

$$\text{val}(x) \leq f \text{val}(z) \leq f \text{val}(x^*)$$

where x^* is an optimal solution for the SET COVER problem. □

Randomized Rounding

Another natural idea for rounding fractional solutions is to use randomization: For example, for the above relaxation, observe that the values z_S are between zero and one. We may thus interpret these values as probabilities for choosing a certain set S.

Here is the idea of the following algorithm: Solve the LP-relaxation optimally and call the solution z. With probability z_S include the set S into the cover.

This basic procedure yields a vector x with expected value equal to the optimal fractional solution value but might not cover all the elements. We thus repeat the procedure "sufficiently many" times and include a set into our cover if it was included in any of the iterations. We will show that $O(\log n)$ many iterations suffice yielding an $O(\log n)$-approximation algorithm.

Algorithm 5.4 RANDOMIZED ROUNDING SET COVER

Input. Universe U with n elements, collection $\mathcal{S} = \{S_1, \ldots, S_k\}$, $S_i \subseteq U$, a cost function $c : \mathcal{S} \to \mathbb{R}$.

Output. Vector $x \in \{0,1\}^k$

Step 1. Set $x = 0$, solve the LP relaxation below, and call the optimal solution z.

$$\begin{aligned} \text{minimize} \quad & \text{val}(x) = \sum_{S \in \mathcal{S}} c(S) x_S, \\ \text{subject to} \quad & \sum_{S : e \in S} x_S \geq 1 \quad e \in U, \\ & x_S \geq 0 \quad S \in \mathcal{S}. \end{aligned}$$

Step 2. Repeat $3 \log n$ times: For each set S set $x_S = 1$ with probability z_S.

Step 3. Return x.

Theorem 5.9. *In expectation, the algorithm* RANDOMIZED ROUNDING SET COVER *is an* $3 \log n$*-approximation algorithm for* SET COVER.

Proof. Let z be an optimal solution for the LP. For each set S set $x_S = 1$ with probability z_S. Then we have

$$\mathbb{E}[\text{val}(x)] = \sum_{S \in \mathcal{S}} \mathbb{E}[c(S) x_S] = \sum_{S \in \mathcal{S}} c(S) \Pr[x_S = 1] = \sum_{S \in \mathcal{S}} c(S) z_S = \text{val}(z).$$

We estimate the probability that an element $u \in U$ is covered in one iteration. Let u be contained in k sets and let $z_1, \ldots, z_k$ be the probabilities given in the solution z. Since u is fractionally covered we have $z_1 + \cdots + z_k \geq 1$. With easy but tedious calculus we see that – under this condition – the probability for u being covered is minimized when the z_i are all equal, i.e., $z_1 = \cdots = z_k = 1/k$:

$$\Pr[x_S = 1] = 1 - (1 - z_1) \cdot \cdots \cdot (1 - z_k) \geq 1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{e}.$$

Summing up this step: In each of the iterations the expected value of the solution x constructed increases by at most $\text{val}(z)$ in expectation. Each element is covered with probability at least $1 - 1/e$. But maybe we have not covered all elements after $3 \log n$ iterations. Here we show that we will have with high probability.

The probability that element u is not covered at the end of the algorithm, i.e., after $3 \log n$ iterations is

$$\Pr[u \text{ is not covered}] \leq \left(\frac{1}{e}\right)^{3 \log n} \leq \frac{1}{n^3}.$$

Thus the probability that there is an uncovered element is at most $1/n^2$, i.e., rather small.

Clearly,

$$\mathbb{E}[\text{val}(x)] \leq 3 \log n \mathbb{E}[\text{val}(z)] \leq 3 \log n \text{val}(x^*),$$

where x^* is an optimal solution for Set Cover. So, the algorithm returns a feasible solution, with high probability, whose expected value is at most $3 \log n$. □

The proof above shows that the algorithm is a $3 \log n$-approximation in expectation. But we can actually strengthen the statement by showing that the approximation ratio is $12 \log n$ with probability around $1/4$. Use Markov's inequality to show

$$\Pr[\text{val}(x) \geq 12 \log n \text{val}(z)] \leq \frac{\mathbb{E}[\text{val}(x)]}{12 \log n \text{val}(z)} \leq \frac{1}{4}$$

The probability that either not all elements are covered or the obtained solution has value larger than $12 \log n$ times the optimal value is at most $1/n^2 + 1/4 \leq 1/2$ for all $n \geq 2$. Thus we only have to run the whole algorithm at most two times in expectation to actually get a $12 \log n$-approximate solution.

Chapter 6

Satisfiability

The SATISFYABILITY problem asks if a certain given Boolean formula has a satisfying assignment, i.e., one that makes the whole formula evaluate to true. There is a related optimization problem called MAXIMUM SATISFIABILITY. The goal of this chapter is to develop a deterministic 3/4-approximation algorithm. We first give a corresponding randomized algorithm which will then be derandomized.

We are given the Boolean *variables* $X = \{x_1, \ldots, x_n\}$, where each $x_i \in \{0,1\}$. A *literal* ℓ_i of the variable x_i is either x_i itself, called a *positive* literal, or its negation $\bar{x}_i$ with truth value $1-x_i$, called a *negative* literal. A *clause* is a disjunction $C = (\ell_1 \vee \cdots \vee \ell_k)$ of literals ℓ_j of X; their number k is called the *size* of C. For a clause C let S_C^+ denote the set of its positive literals; similarly S_C^- the set of its negative literals. Let $\mathcal{C}$ denote the set of clauses. A Boolean formula in *conjunctive form* is a conjunction of clauses $F = C_1 \wedge \cdots \wedge C_m$. Each vector $x \in \{0,1\}^n$ is called a *truth assignment*. For any clause C and any such assignment x we say that x *satisfies* C if at least one of the literals of C evaluates to 1.

The problem MAXIMUM SATISFIABILITY is the following: We are given a formula F in conjunctive form and for each clause C a weight w_C, i.e., a weight function $w : \mathcal{C} \to \mathbb{N}$. The objective is to find a truth assignment $x \in \{0,1\}^n$ that maximizes the total weight of the satisfied clauses. As an important special case: If we set all weights w_C equal to one, then we seek to maximize the number of satisfied clauses.

Now we introduce for each clause C a variable $z_C \in \{0,1\}$ which takes the value one if and only if C is satisfied under a certain truth assignment x. Now we can formulate this problem as a mathematical program as follows:

Problem 6.1 MAXIMUM SATISFIABILITY

Instance. Formula $F = C_1 \wedge \cdots \wedge C_m$ with m clauses over the n Boolean variables $X = \{x_1, \ldots, x_n\}$. A weight function $w : \mathcal{C} \to \mathbb{N}$.

Task. Solve the problem

$$\begin{aligned} \text{maximize} \quad & \operatorname{val}(z) = \sum_{C \in \mathcal{C}} w_C z_C, \\ \text{subject to} \quad & \sum_{i \in S_C^+} x_i + \sum_{i \in S_C^-} (1 - x_i) \geq z_C \quad C \in \mathcal{C}, \\ & z_C \quad C \in \mathcal{C}, \\ & x_i \in \{0,1\} \quad i = 1, \ldots, n. \end{aligned}$$

The algorithm we aim for is a combination of two algorithms. One works better for small clauses, the other for large clauses. Both are initially randomized but can be *derandomized* using the method of conditional expectation, i.e., the final algorithm is deterministic.

6.1 Randomized Algorithm

For each variable x_i we define the random variable X_i that takes the value one with a certain probability p_i and zero otherwise. This induces, for each clause C, a random variable Z_C that takes the value one if C is satisfied under a (random) assignment and zero otherwise.

Algorithm for Large Clauses

Consider this algorithm Randomized Large: For each variable x_i with $i = 1, \ldots, n$, set $x_i = 1$ independently with probability 1/2 and $x_i = 0$ otherwise. Output x.

Define the quantity

$$\alpha_k = 1 - 2^{-k}.$$

Lemma 6.1. *Let C be a clause. If $\operatorname{size}(C) = k$ then*

$$\mathbb{E}\left[Z_C\right] = \alpha_k.$$

Proof. A clause C is not satisfied, i.e., $z_C = 0$ if and only if all its literals are set to zero. By independence, the probability of this event is exactly 2^{-k} and thus

$$\mathbb{E}\left[Z_C\right] = 1 \cdot \Pr\left[Z_C = 1\right] + 0 \cdot \Pr\left[Z_C = 0\right] = 1 - 2^{-k} = \alpha_k$$

which was claimed. □

Theorem 6.2. *In expectation, the algorithm* Randomized Large *is a 1/2-approximation algorithm for* Maximum Satisfiability.

Proof. By linearity of expectation, Lemma 6.1, and $\operatorname{size}(C) \geq 1$ we have

$$\mathbb{E}\left[\operatorname{val}(Z)\right] = \sum_{C \in \mathcal{C}} w_C \mathbb{E}\left[Z_C\right] = \sum_{C \in \mathcal{C}} w_C \alpha_{\operatorname{size}(C)} \geq \frac{1}{2} \sum_{C \in \mathcal{C}} w_C \geq \frac{1}{2} \operatorname{val}(z')$$

where (x', z') is an optimal solution for Maximum Satisfiability. We have used the obvious bound $\operatorname{val}(z') \leq \sum_{C \in \mathcal{C}} w_C$. □

Algorithm for Small Clauses

Maybe the most natural linear programming relaxation of the problem is:

$$\begin{aligned}
\text{maximize} \quad & \operatorname{val}(z) = \sum_{C \in \mathcal{C}} w_C z_C, \\
\text{subject to} \quad & \sum_{i \in S_C^+} x_i + \sum_{i \in S_C^-} (1 - x_i) \geq z_C \quad C \in \mathcal{C}, \\
& 0 \leq z_C \leq 1 \quad C \in \mathcal{C} \\
& 0 \leq x_i \leq 1 \quad i = 1, \ldots, n.
\end{aligned}$$

In the sequel let (x^*, z^*) denote an optimum solution for this LP.

Consider this algorithm RANDOMIZED SMALL: Determine (x^*, z^*). For each variable x_i with $i = 1, \ldots, n$, set $x_i = 1$ independently with probability x_i^* and $x_i = 0$ otherwise. Output x.

Define the quantity

$$\beta_k = 1 - \left(1 - \frac{1}{k}\right)^k.$$

Lemma 6.3. *Let C be a clause. If* $\text{size}(C) = k$ *then*

$$\mathbb{E}\left[Z_C\right] = \beta_k z_C^*.$$

Proof. We may assume that the clause C has the form $C = (x_1 \vee \cdots \vee x_k)$; otherwise rename the variables and rewrite the LP.

The clause C is satisfied if $x_1, \ldots, x_k$ are not all set to zero. The probability of this event is

$$\begin{aligned} 1 - \Pi_{i=1}^k (1 - x_i^*) &\geq 1 - \left(\frac{\sum_{i=1}^k (1 - x_i^*)}{k}\right)^k \\ &= 1 - \left(1 - \frac{\sum_{i=1}^k x_i^*}{k}\right)^k \\ &\geq 1 - \left(1 - \frac{z_C^*}{k}\right)^k. \end{aligned}$$

Above we firstly have used the arithmetic-geometric mean inequality, which states that for non-negative numbers $a_1, \ldots, a_k$ we have

$$\frac{a_1 + \cdots + a_k}{k} \geq \sqrt[k]{a_1 \cdot \cdots \cdot a_k}.$$

Secondly the LP guarantees the inequality $x_1^* + \cdots + x_k^* \geq z_C^*$.

Now define the function $g(t) = 1 - (1 - t/k)^k$. This function is concave with $g(0) = 0$ and $g(1) = 1 - (1 - 1/k)^k$ which yields that we can bound

$$g(t) \geq t(1 - (1 - 1/k)^k) = t\beta_k$$

for all $t \in [0, 1]$.

Therefore

$$\Pr\left[Z_C = 1\right] \geq 1 - \left(1 - \frac{z_C^*}{k}\right)^k \geq \beta_k z_C^*$$

and the claim follows. □

Theorem 6.4. *In expectation, the algorithm* RANDOMIZED SMALL *is a* $1 - 1/e$*-approximation algorithm for* MAXIMUM SATISFIABILITY.

Proof. The function β_k is decreasing with k. Therefore if all clauses are of size at most k, then by Lemma 6.3

$$\mathbb{E}\left[\text{val}(Z)\right] = \sum_{C \in \mathcal{C}} w_C \mathbb{E}\left[Z_C\right] \geq \beta_k \sum_{C \in \mathcal{C}} w_C z_C^* = \beta_k \text{val}(z^*) \geq \beta_k \text{val}(z'),$$

where (x', z') is an optimal solution for MAXIMUM SATISFIABILITY. The claim follows since $(1 - 1/k)^k > 1/e$ for all $k \in \mathbb{N}$. □

3/4-Approximation Algorithm

Consider the algorithm Randomized Combine: With probability 1/2 run Randomized Large otherwise run Randomized Small.

Lemma 6.5. *Let C be a clause, then*

$$\mathbb{E}[Z_C] = \frac{3z_C^*}{4}.$$

Proof. Let the random variable B take the value zero if the first algorithm is run, one otherwise. For a clause C let $\text{size}(C) = k$. By Lemma 6.1 and $z_C^* \leq 1$

$$\mathbb{E}[Z_C \mid B = 0] = \alpha_k \geq \alpha_k z_C^*.$$

and by Lemma 6.1

$$\mathbb{E}[Z_C \mid B = 1] \geq \beta_k z_C^*.$$

Combining we have

$$\mathbb{E}[Z_C] = \mathbb{E}[Z_C \mid B = 0]\Pr[B = 0] + \mathbb{E}[Z_C \mid B = 1]\Pr[B = 1] \geq \frac{z_C^*}{2}(\alpha_k + \beta_k).$$

It is easy to see that $\alpha_k + \beta_k \geq 3/2$ for all $k \in \mathbb{N}$. □

Theorem 6.6. *In expectation, the algorithm* Randomized Combine *is a* 3/4*-approximation algorithm for* Maximum Satisfiability.

Proof. This follows from Lemma 6.5 and linearity of expectation. □

6.2 Derandomization

The notion of *derandomization* refers to "turning" a randomized algorithm into a deterministic one (possibly at the cost of additional running time or deterioration of approximation guarantee). One of the several available techniques is the method of *conditional expectation.*

We are given a Boolean formula $F = C_1 \wedge \cdots \wedge C_m$ in conjunctive form over the variables $X = \{x_1, \ldots, x_n\}$. Suppose we set $x_1 = 0$, then we get a formula F_0 over the variables $x_2, \ldots, x_n$ after simplification; if we set $x_1 = 1$ then we get a formula F_1.

Example 6.7. Let $F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3) \wedge (x_1 \vee \bar{x}_4)$ where $X = \{x_1, \ldots, x_4\}$.

$$\begin{aligned} x_1 = 0: &\quad F_0 = (x_2) \wedge (x_4) \\ x_1 = 1: &\quad F_1 = (x_3) \end{aligned}$$

Applying this recursively, we obtain the tree $T(F)$ depicted in Figure 6.1. The tree $T(F)$ is a complete binary tree with height $n+1$ and $2^{n+1}-1$ vertices. Each vertex at level i corresponds to a setting for the Boolean variables $x_1, \ldots, x_i$. We label the vertices of $T(F)$ with their respective conditional expectations as follows. Let $X_1 = a_1, \ldots, X_i = a_i \in \{0, 1\}$ be the outcome of a truth assignment for the variables $x_1, \ldots, x_i$. The vertex corresponding to this assignment will be labeled

$$\mathbb{E}[\text{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i].$$

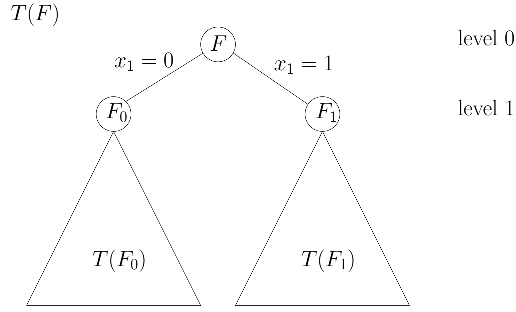


Figure 6.1: Derandomization tree for a formula F.

If $i = n$, then this conditional expectation is simply the total weight of clauses satisfied by the truth assignment $x_1 = a_1, \ldots, x_n = a_n$.

The goal of the remainder of the section is to show that we can find deterministically in polynomial time a path from the root of $T(F)$ to a leaf such that the conditional expectations of the vertices on that path are at least as large as $\mathbb{E}[\mathrm{val}(Z)]$. Obviously, this property yields the desired: We can construct determistically a solution which is at least as good as the one of the randomized algorithm in expectation.

Lemma 6.8. *The conditional expectation*

$$\mathbb{E}[\mathrm{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i]$$

of any vetex in $T(F)$ can be computed in polynomial time.

Proof. Consider a vertex $X_1 = a_1, \ldots, X_i = a_i$. Let F' be the Boolean formula obtained from F by setting $x_1, \ldots, x_i$ accordingly. F' is in the variables $x_{i+1}, \ldots, x_n$.

Clearly, by linearity of expectation, the expected weight of any clause of F' under any random truth assignment to the variables $x_{i+1}, \ldots, x_n$ can be computed in polynomial time. Adding to this the total weight of clauses satisfied by $x_1, \ldots, x_i$ gives the answer. □

Theorem 6.9. *We can compute in polynomial time a path from the root to a leaf in $T(F)$ such that the conditional expectation of each vertex on this path is at least $\mathbb{E}[\mathrm{val}(Z)]$.*

Proof. Consider the conditional expectation at a certain vertex $X_1 = a_1, \ldots, X_i = a_i$ for setting the next variable X_{i+1}. We have that

$$\begin{aligned}
&\mathbb{E}[\mathrm{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i] \\
&\qquad = \mathbb{E}[\mathrm{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = 0] \Pr[X_{i+1} = 0] \\
&\qquad\qquad + \mathbb{E}[\mathrm{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = 1] \Pr[X_{i+1} = 1].
\end{aligned}$$

We show that the two conditional expectations with X_{i+1} can *not* be both strictly smaller than $\mathbb{E}[\mathrm{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i]$. Assume the contrary, then we have

$$\begin{aligned}
&\mathbb{E}[\mathrm{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i] \\
&\qquad < \mathbb{E}[\mathrm{val}(Z) \mid X_1 = a_1, \ldots, X_i = a_i] (\Pr[X_{i+1} = 0] + \Pr[X_{i+1} = 1])
\end{aligned}$$

which is a contradiction since $\Pr[X_{i+1} = 0] + \Pr[X_{i+1} = 1] = 1$.

This yields the existence of such a path can by Lemma 6.8 it can be computed in polynomial time. □

The derandomized version of a randomized algorithm now simply executes these proofs with the probability distribution as given by the randomized algorithm.

Chapter 7

Facility Location

The Facility Location problem was popular in operations research in the 1960s but no constant factor approximation algorithms were known until 1997. The discovery of these is due to LP-rounding techniques and the primal-dual schema. In this section we present a 3-approximate primal-dual algorithm.

Facility Location is the following problem: We are given a complete bipartite graph $G = (V, E)$ with bipartition $V = F \cup C$, where F refers to the set of (potential) *facilities* and C to the set of *cities*. Establishing a facility i causes *opening cost* f_i. Attaching city j to an (opened) facility i yields *connection cost* c_{ij}. We assume that the c_{ij} satisfy the triangle inequality. So, now, the problem is to find a subset $I \subseteq F$ of facilities to open and a mapping $a : C \to I$ for assigning cities to open facilities in a way that each city is connected to at least one facility as to minimize the total opening and connection cost. We write this task as a mathematical program, where y_i indicates if facility i is open and x_{ij} if city j is assigned to facility i.

Problem 7.1 Facility Location

Instance. Complete bipartite graph $G = (F \cup C, E)$, weight functions $f : F \to \mathbb{N}$, and $c : E \to \mathbb{N}$.

Task. Solve the problem

$$\begin{aligned} \text{minimize} \quad & \text{val}(x, y) = \sum_{i \in F} \sum_{j \in C} c_{ij} x_{ij} + \sum_{i \in F} f_i y_i \\ \text{subject to} \quad & \sum_{i \in F} x_{ij} \geq 1 \quad j \in C \quad \text{“each city connects”} \\ & y_i - x_{ij} \geq 0 \quad i \in F, j \in C \quad \text{“facility must be open”} \\ & x_{ij} \in \{0, 1\} \\ & y_i \in \{0, 1\}. \end{aligned}$$

The problem Facility Location is NP-hard. Here we will show the following main result.

Theorem 7.1. *There is a* 3*-approximation algorithm for* Facility Location *that runs in* $O(m \log m)$ *time, where* $m = |F||C|$.

An obvious way of relaxing this problem is to replace the constraints $x_{ij} \in \{0, 1\}$ and

$y_i \in \{0,1\}$ by $0 \le x_{ij} \le 1$ and $0 \le y_i \le 1$ respectively. For sake of completeness:

$$\begin{aligned} \text{minimize} \quad & \mathrm{val}(x,y) = \sum_{i\in F}\sum_{j\in C} c_{ij}x_{ij} + \sum_{i\in F} f_i y_i \\ \text{subject to} \quad & \sum_{i\in F} x_{ij} \ge 1 \quad j \in C \\ & y_i - x_{ij} \ge 0 \quad i \in F, j \in C \\ & 0 \le x_{ij} \le 1 \\ & 0 \le y_i \le 1. \end{aligned} \tag{P}$$

The dual of this LP can be written as:

$$\begin{aligned} \text{maximize} \quad & \overline{\mathrm{val}}(\alpha,\beta) = \sum_{j\in C} \alpha_j \\ \text{subject to} \quad & \alpha_j - \beta_{ij} \le c_{ij} \quad i \in F, j \in C \\ & \sum_{j\in C} \beta_{ij} \le f_i \quad i \in F \\ & \alpha_j \ge 0 \quad j \in C \\ & \beta_{ij} \ge 0 \quad i \in F, j \in C. \end{aligned} \tag{D}$$

7.1 Complementary Slackness

One of the important steps in designing a primal-dual algorithm is to get an intuition what the dual LP "is doing". This usually induces a schema how dual variables can "pay" for primal ones.

We begin with a recapitulation of the *complementary slackness conditions*, which refers to Corollary 2.13 of the strong duality theorem and gives one way of proving optimality:

Corollary 7.2. *Let* $\max\{c^\top x : Ax \le b, x \ge 0\}$ *and* $\min\{y^\top b : y^\top A \ge c, y \ge 0\}$ *be a primal-dual pair and let* x *and* y *be respective feasible solutions. Then the following statements are equivalent:*

(1) x *and* y *are both optimum solutions.*

(2) $c^\top x = y^\top b$.

(3) $(y^\top A - c)^\top x = 0$.

(4) $y^\top (b - Ax) = 0$.

For us, the third and the fourth condition are particularly interesting as they relate the primal and dual variables at optimal points. The third is called *primal complementary slackness*, the forth *dual complementary slackness*. One way of looking at this dual condition is to say that "either $y_i = 0$ or $b_i - (Ax)_i = 0$", i.e., "if the dual variable y_i is not zero, then the corresponding primal constraint is satisfied with equality". Similarly for the primal condition.

Now we return to Facility Location. Assume for the moment that there is an integral solution, say (x, y), which is optimal for (P). This solution corresponds to a set $I \subseteq F$ and a mapping $a : C \to I$. Thus, under this solution, $y_i = 1$ if and only if $i \in I$ and $x_{ij} = 1$ if and only if $a(j) = i$. Let (α, β) be an optimal solution for (D).

Now, for (P) and (D) the primal-dual complementary slackness conditions are:

(1) For all $i \in F$ and $j \in C$: either $x_{ij} = 0$ or $\alpha_j - \beta_{ij} = c_{ij}$.

(2) For all $i \in F$: either $y_i = 0$ or $\sum_{j \in C} \beta_{ij} = f_i$.

(3) For all $i \in C$: either $\alpha_j = 0$ or $\sum_{i \in F} x_{ij} = 1$.

(4) For all $i \in F$ and $j \in C$: either $\beta_{ij} = 0$ or $y_i = x_{ij}$.

By (2) each open facility i must be "paid" by the dual variables β_{ij}, i.e.,

$$\sum_{j \in C} \beta_{ij} = f_i.$$

By condition (4), if facility i is open, but city j is not assigned to it, i.e., $a(j) \neq i$, then we must have $y_i \neq x_{ij}$ and thus $\beta_{ij} = 0$. This means that no city contributes to a facility it is not connecting to.

By condition (1), if for some city j and facility i we have $a(j) = i$ then we must have $\alpha_j - \beta_{ij} = c_{ij}$. Thus we can think of $\alpha_j = \beta_{ij} + c_{ij}$ as the total price paid by city j, where β_{ij} is its opening cost share and c_{ij} its connection cost (paid exclusively).

7.2 Primal-Dual Algorithm

Here we apply the primal-dual schema, i.e., we carry out the following steps: Relax the primal slackness conditions and use these for an algorithm which ensures dual feasibility and improves primal optimality.

Relaxing the Slackness Conditions

We will relax the primal slackness conditions as follows: The cities are partitioned into two sets, *directly connected* and *indirectly connected*. Only directly connected cities will pay for opening facilities, i.e., β_{ij} can be non-zero only if j is a directly connected city and $a(j) = i$. For an indirectly connected city j, we relax the primal slackness condition to

$$\frac{1}{3} c_{a(j)j} \leq \alpha_j \leq c_{a(j)j}.$$

So, with the above intuition this reads: The total price paid by an indirectly connecting city is at most its direct connection cost, but at least one third of this cost. All other primal conditions are maintained, i.e., for a directly connectiong city j we have

$$\alpha_j - \beta_{a(j)j} = c_{a(j)j},$$

and for each open facility i,

$$\sum_{j : a(j) = i} \beta_{ij} = f_i.$$

Algorithm

The algorithm consists of two phases. In the first phase, the algorithm operates in a primal-dual fashion. It finds a dual feasible solution and also determines a set of tight edges and temporarily open facilities F_t. In the second phase the algorithm chooses a subset $I \subseteq F_t$ of facilities to open permanently, and a mapping $a : C \rightarrow I$.

Phase 1. We would like to find as large a dual solution as possible. This motivates the following underlying process: Each city j raises its dual variable α_j until it gets connected to an open facility, i.e., until $\alpha_j = c_{ij}$ for some open facility i. All other primal and dual variables simply respond to this change, trying to maintain feasibility or satisfying complementary slackness conditions.

A notion of *time* is defined in this phase, so that each event can be associated with the time at which it happened; the phase starts at time zero. Initially each city is defined to be *unconnected*. Throughout this phase, the algorithm raises the dual variable α_j for each unconnected city at unit rate, i.e., α_j will grow by one in unit time. When $\alpha_j = c_{ij}$ for some edge ij, the algorithm will declare this edge to be *tight*. Henceforth, the dual variable β_{ij} will also be raised uniformly, thus ensuring that the constraint $\alpha_j - \beta_{ij} \leq c_{ij}$ in (D) is never violated. At this point in time the connection cost are paid and the variable β_{ij} goes towards paying for the opening cost of facility i. Each edge ij such that $\beta_{ij} > 0$ is called *special*.

Facility i is said to be *paid for* if $\sum_j \beta_{ij} = f_i$. If so, the algorithm declares the facility *temporarily open*, i.e., $i \in F_t$. Furthermore, all unconnected cities having tight edges to this facility are declared *connected* and facility i is declared the *connecting witness* for each of these cities. (Notice that the dual variables α_j of these cities are not raised any more.) In the future, as soon as an unconnected city j gets a tight edge to i, j will also be declared connected and i the connection witness for j. (Notice that $\beta_{ij} = 0$ and the edge ij is not special.) When all cities are connected, the first phase terminates. If several events happen simultaneously, the algorithm executes them in arbitrary order.

As a side remark, at the end of this phase, a city may have paid towards temporarily opening several facilities. However, we want to ensure that a city pays only for the facility that it is eventually connected to. This is ensured in the second phase, which chooses a set of temporarily open facilities for opening permanently.

Phase 2. Let F_t denote the set of temporarily open facilities and T denote the subgraph of G induced by all special edges. Let T^2 denote the graph that has an edge uv if and only if there is a path of length at most two between u and v in T, and let H be the subgraph of T^2 induced by F_t. Find any maximal independent set in H, say I. All facilities in the set I are declared *open*.

For city j, define $\mathcal{F}_j = \{i \in F_t : ij \text{ is special}\}$. Since I is an independent set, at most one of the facilities in $\mathcal{F}_j$ is opened. If there is a facility $i \in \mathcal{F}_j$ that is opened, then set $a(j) = i$ and declare the city j *directly connected*. Otherwise, consider the tight edge $i'j$ such that i' was the connecting witness for j. If $i' \in I$, again set $a(j) = i'$ and declare the city j *directly connected* (notice that in this case $\beta_{i'j} = 0$). In the remaining case that $i' \notin I$, let i be any neighbor of i' in the graph H such that $i \in I$. Set $a(j) = i$ and declare city j *indirectly connected*.

The set I and the mapping $a : C \to I$ define a primal integral solution: $x_{ij} = 1$ if and only if $a(j) = i$ and $y_i = 1$ if and only if $i \in I$. The values for α_j and β_{ij} obtained at the end of the first phase form a dual feasible solution.

Analysis

The crucial result for the analysis, which directly gives the approximation guarantee, is the following.

Theorem 7.3. *The primal and dual solutions constructed by the algorithm satisfy*

$$\sum_{i\in F}\sum_{j\in C} c_{ij}x_{ij} + 3\sum_{i\in F} f_i y_i \le 3\sum_{j\in C}\alpha_j.$$

We will show how the dual variables α_j pay for the primal costs of opening facilities and connecting cities to facilities. Denote by α_j^f and α_j^c the contributions of city j to these two costs respectively; $\alpha_j = \alpha_j^f + \alpha_j^c$. If j is indirectly connected then $\alpha_j^f = 0$ and $\alpha_j^c = \alpha_j$. If j is directly connected then the following must holds:

$$\alpha_j = c_{ij} + \beta_{ij},$$

where $i = a(j)$. Now let $\alpha_j^f = \beta_{ij}$ and $\alpha_j^c = c_{ij}$.

Lemma 7.4. *Let $i \in I$ then*

$$\sum_{j:a(j)=i} \alpha_j^f = f_i.$$

Proof. Since i is temporarily open at the end of phase one, it is completely paid for, i.e.,

$$\sum_{j:ij \text{ is special}} \beta_{ij} = f_i.$$

The critical observation is that each city j that has contributed to f_i must be directly connected to i. For each such city, $\alpha_j^f = \beta_{ij}$. Any other city j' that is connected to facility i must satisfy $\alpha_{j'}^f = 0$. □

Corollary 7.5. $\sum_{i\in I} f_i = \sum_{j\in C}\alpha_j^f$.

Recall that α_j^f was defined to be 0 for indirectly connected cities. Thus, only the directly connected cities pay for the cost of opening facilities.

Lemma 7.6. *For an indirectly connected city j, $c_{ij} \le 3\alpha_j^c$, where $i = a(j)$.*

Proof. Let i' be the connecting witness for city j. Since j is indirectly connected to i, the edge ii' must be an edge in H. In turn, there must be a city, say j', such that ij and $i'j'$ are both special edges. Let t_1 and t_2 be the times at which i and i' were declared temporarily open during phase 1.

Since edge $i'j'$ is tight, $\alpha_j \ge c_{i'j}$. We will show that $\alpha_j \ge c_{ij'}$ and $\alpha_j \ge c_{i'j'}$. Then, the lemma will follow by using the triangle inequality.

Since edges $i'j'$ and ij' are tight, $\alpha_{j'} \ge c_{ij'}$ and $\alpha_{j'} \ge c_{i'j'}$. Since both these edges are special, they must both have gone tight before either i or i' is declared temporarily open. Consider the time $\min\{t_1, t_2\}$. Finally, since i' is the connecting witness for j, $\alpha_j \ge t_2$. Therefore $\alpha_j \ge \alpha_{j'}$, and the required inequalities follow. □

Proof of Theorem 7.3. For a directly connected city j, $c_{ij} = \alpha_j^c \le 3\alpha_j^c$, where $i = a(j)$. With Lemma 7.6 we get

$$\sum_{i\in F}\sum_{j\in C} c_{ij}x_{ij} \le 3\sum_{j\in C}\alpha_j^c.$$

Adding to this the equality given in Corollary 7.5 multiplied by 3 yields the claim. □

Running Time

Clearly, the total number of edges of the complete bipartite graph $G = (F \cup C, E)$ is $m = |F||C|$. For the implementation of the algorithm sort all the edges by increasing cost. This is the ordering in which they go tight. For each facility i, we maintain the number of cities that are currently contributing to it, and the *anticipated time* t_i, at which it would be completely paid for if no other event happens on the way. Initially all t_i's are infinite and each facility has 0 cities contributing to it. The t_i's are maintained in a binary heap, so we can update each one and find the current minimum in $O(\log |F|)$ time. Two types of events happen and they lead to the following updates:

An edge ij goes tight. If facility i is not temporarily open, then it gets one more city contributing towards its cost. The corresponding amount can easily be computed. Thus, the anticipated time for facility i to be paid for can be computed in constant time. The heap can be updated in $O(\log |F|)$ time.

If facility i is already temporarily open, city j is declared connected, and α_j is not raised anymore. For each facility i' that was counting j as a contributor, we need to decrease the number of contributors by 1 and recompute the anticipated time at which it gets paid for.

Facility i is completely paid for. In this event, i will be declared temporarily open, and all cities contributing to i will be declared connected. For each of these cities, we will execute the second case of the previous event, i.e., update facilities that they were contributing towards.

Observe that each edge ij will be considered at most twice. First, when it goes tight, and second when city j is declared connected. For each consideration of this edge, we will do $O(\log |F|)$ work. This discussion of the running time together with Theorem 7.3 yields Theorem 7.1.

Tight Example

The following familty of examples shows that the analysis of the algorithm is tight.

Example 7.7. There are two facilities with opening cost $f_1 = \varepsilon$ and $f_2 = (n+1)\varepsilon$. There are n cities where city one is at distance one from facility one, but the remaining cities are at distance three from it, and each city is at distance one from facility two. The optimal solution is to open facility two at total cost $(n+1)\varepsilon + n$. The algorithm will open facility one and connect all cities to it at total cost $\varepsilon + 1 + 3(n-1)$.