Advanced Algorithms: Design and Analysis

Lecture Notes, Spring Term 2015

University of Zurich

Dr. Alexander Souza

Contents

Chapter 1

Introduction

1.1 Examples

We start with some examples of combinatorial optimization problems.

Example 1.1. The following problem is called the KNAPSACK problem. 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. Another example is a LOAD BALANCING problem: We have m machines and we have a set of n jobs that need to be done. Each job j has a processing time $p_{i,j}$ if executed by machine i. We can formulate our problem with the following mathematical program. We use the variables $x_{i,j} \in \{0, 1\}$ that indicate if job j is assigned to machine i. We want to minimize the time until all jobs are finished.

$$
\begin{array}{lll}
\text{minimize} & f, & \text{“minimize finishing time } f\text{”} \\
\text{subject to} & \displaystyle\sum_{j=1}^{n} p_{i,j} x_{i,j} \leq f, \quad i = 1, \ldots, m & \text{“}f \text{ is largest machine time”} \\
& \displaystyle\sum_{i=1}^{m} x_{i,j} = 1, \qquad j = 1, \ldots, n & \text{“each job gets done”} \\
& x_{i,j} \in \{0, 1\}, \qquad i = 1, \ldots, m,\ j = 1, \ldots .n & \text{“assignment”}
\end{array}
$$

1.2 Combinatorial Optimization Problems

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

U	the *solution space* of possible outputs,
P	the *feasibility predicate*,
value	the *value function* $\text{value} : U \to \mathbb{R}$,
extr	the desired *extremum*,i.e., max or min.

extr and value together define the *objective function*. The feasibility predicate P induces a set:

$$S \quad \text{the set of } \textit{feasible solutions}\text{: } S = \{X \in U : X \text{ satisfies } P\}.$$

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

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*.

Example 1.3. Let us investigate the problem in Example 1.1 in with this formalism.

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

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

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 I is called the (binary) *length* or *size* of the input and denoted $\text{size}(I)$.

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 $\alpha t(\text{size}(I))$ many elementary steps to compute a solution given any input I. 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 $I = (U, P, S, \text{value}, \text{extr})$. For any $I \in \Pi$, let $\text{OPT}(I) = \text{extr}_{X \in S(I)} \text{value}(X)$ 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 $\text{ALG}(I) = \text{value}(X)$. 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 Linear and Integer Linear Programs

Many COPs, like the ones above can be formulated in terms of a Integer Linear Program (ILP).

Let $A = (a_{i,j})_{i=1,\dots,m, j=1,\dots,n} \in \mathbb{R}^{m,n}$ be a matrix and let $b = (b_i)_{i=1,\dots,m} \in \mathbb{R}^m$ and $c = (c_j)_{j=1,\dots,n} \in \mathbb{R}^n$ be vectors. Further let $x = (x_j)_{j=1,\dots,n} \in \mathbb{Z}^n$ be variables that are allowed to take *integral* values, only. Our objective function is to minimize $c^\top x$ subject to $Ax \leq b$. More explicitly

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} c_j x_j, & & \text{“objective function”} \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{i,j} x_j \leq b_i, & i = 1, \dots, m \quad & \text{“constraints”} \\ & x_j \in \mathbb{Z}, & j = 1, \dots, n. \quad & \text{“integrality”} \end{aligned}$$

Solving an ILP is in general NP-hard. However, we will often replace the constraints $x_j \in \mathbb{Z}$ with $x_j \in \mathbb{R}$. This is then called a *relaxation* as a Linear Program (LP) and can be solved in polynomial time. The Ellipsoid method is one such algorithm, but is only interesting from theoretical perspective. In practice, the Simplex algorithm is frequently used, although it is not polynomial time in the worst case.

Of course, an LP solution is in general not feasible for the ILP, but we can sometimes "turn" it into a feasible solution, which is not "too bad".

1.5 Randomized Algorithms

Generally speaking, *randomized algorithms* have access to (arbitrarily many) random bits, and are allowed to decide based on their outcomes. This, of course, yields that either the output and/or the running time of the algorithm are random variables. As a consequence we can not expect that the algorithm behaves exactly the same, when given the same input. (Notice that deterministic algorithms do have this property.) Why would one want to allow such indefiniteness? There are several reasons: For example, approximation

algorithms are sometimes "fooled" by rather artifical counterexamples that rely on the specific strategy of the algorithm. In that perspective, randomizing strategies often yield improvements in approximation guarantee (in expectation). Furthermore, random choices sometimes yield significant speed-up of running time (in expectation), compared to worst-case running times.

There are two types of randomized algorithms: *Las Vegas* algorithms are allowed to use the random bits, but must always return correct answers. In contrast, *Monte Carlo* algorithms are allowed to return false answers, but this must not happen with "too large" probability.

If it is not desired to have a randomized algorithm, respectively a randomized construction method, then one can also try to *derandomize* a randomized algorithm. As the name suggests, this refers to the process of turning a randomized algorithm into a deterministic one. This is often achieved at the price of higher running times and/or deterioration of solution quality.

Chapter 2

Greedy Algorithms

Algorithms of the GREEDY type have in common that they construct solutions in an iterative manner and based on "locally optimal" criteria. Sometimes this strategy even yields a globally optimal solution, but mostly the final solution is non-optimal. In this chapter we give an example for optimal GREEDY algorithms (for the MINIMUM SPANNING TREE problem) and an example of a non-optimal one (for the NP-hard SET COVER problem).

2.1 Minimum Spanning Trees

Consider a telecommunications company that wants to rent a subset of an existing set of cables in a network, each of which connects two cities. The rented cables should suffice to connect all the cities and they should be as cheap as possible. This type of applications is formalized as follows:

Let G be an undirected, connected graph on the vertices $V(G) = V$ and the edges $E(G) = E$. Furthermore, let $c : E \to \mathbb{R}$ be a cost function. A graph F is a *forest* if F has no cycles. A forest F with $V(F) = V(G)$ and $E(F) \subseteq E(G)$ is a *spanning forest*. A connected forest is a *tree* and likewise a connected spanning forest is a *spanning tree*. For any vertex set C, called a *cut*, let $\delta(C) = \{e = uv \in E : u \in C, v \notin C\}$ denote the *cut-edges* over C.

Problem 2.1 MINIMUM SPANNING TREE

Instance. Undirected, connected graph G on the vertices $V(G)$ and the edges $E(G)$, cost function $c : E(G) \to \mathbb{R}$.

Task. Find a spanning tree T with cost value$(T) = \sum_{e \in E(T)} c(e)$ minimal.

See Figure 2.1 for an example. Maybe the first idea that comes to mind is to sort the edges according to non-decreasing cost and to add the edges one-by-one provided that each addition does not close a cycle. This is the invariant of the algorithm KRUSKAL.

Another idea is to maintain a minimum spanning tree of subsets of the vertex set of the graph and to extend the tree until it is a spanning tree of the graph. This is the idea behind the algorithm PRIM.

Now we will establish several structural properties of any optimal, i.e., minimum cost, spanning tree and derive the correctness of the two algorithms from those.

Theorem 2.1. *Let G be an undirected graph and $c : E \to \mathbb{R}$ be a cost function and let T be a spanning tree. Then we have*

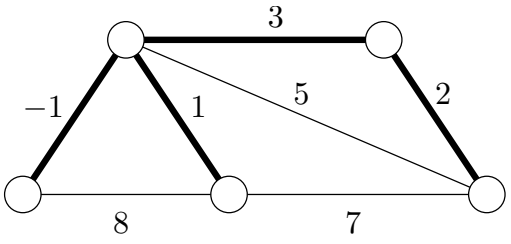


Figure 2.1: An example of an undirected, connected graph with a cost function on the edges. The bold edges indicate a minimum spanning tree in that graph.

Algorithm 2.1 KRUSKAL

Input. Undirected connected graph $G = (V, E)$, cost function $c : E \to \mathbb{R}$.

Output. Spanning tree T with minimum cost.

Step 1. Sort the edges such that $c(e_1) \leq \cdots \leq c(e_m)$ and let $T = (V, \emptyset)$.

Step 2. For $i = 1, \ldots, m$ do: If $T + e_i$ is a forest then $T = T + e_i$.

---

(1) T is a minimum spanning tree.

(2) For any $e = uv \in E(G) - E(T)$, no edge on the u-v-path in T has larger cost than e.

(3) For every $e \in E(T)$, e is a minimum cost edge over $\delta(C)$, where C is a connected component of $T - e$.

(4) We can order $E(T) = \{e_1, \ldots, e_{n-1}\}$ such that for each $i = 1, \ldots, n-1$, there is a set $X \subseteq V(G)$ such that e_i is a minimum cost edge over $\delta(X)$ and $e_j \notin \delta(X)$ for all $j = 1, \ldots, i-1$.

Proof. (1) $\Rightarrow$ (2): Suppose (2) is violated. Let $e = uv \in E(G) - E(T)$ and let e' be an edge on the u-v-path in T with $c(e') > c(e)$. Then $T' = T - e' + e$ is a spanning tree with lower cost than T.

(2) $\Rightarrow$ (3): Suppose (3) is violated. Let $e \in E(T)$, C a connected component of $T - e$ and $e' = uv \in \delta(C)$ with $c(e') < c(e)$. Observe that the u-v-path in T must contain an edge of $\delta(C)$. But the only such edge is e. So (2) is violated.

(3) $\Rightarrow$ (4): Take an arbitrary order and $X = V(C)$.

(4) $\Rightarrow$ (1): Suppose $E(T) = \{e_1, \ldots, e_{n-1}\}$ satisfies (4) and let T^* be an optimimum spanning tree such that $i = \inf\{h \in \{1, \ldots, n-1\} : e_h \notin E(T^*)\}$ is maximum. We show that $i = \infty$, i.e., $T = T^*$. Suppose not, then let $X \subseteq V(G)$ such that e_i is a minimum cost edge of $\delta(X)$ and $e_j \notin \delta(X)$ for all $j = 1, \ldots, i-1$. $T^* + e_i$ contains a cycle C. Since $e_i \in E(C) \cap \delta(X)$, at least one more edge e' (with $e' \neq e_i$) of C must belong to $\delta(X)$. Observe that $T + e_i - e'$ is a spanning tree. Since T^* is optimum $c(e_i) > c(e')$. But since $e' \in \delta(X)$, we also have $c(e') > c(e_i)$. Moreover, if $e' = e_j \in E(T)$, then $j > i$. So $c(e') = c(e_i)$ and $T^* + e_i - e'$ is another optimum spanning tree, which contradicts the maximality of i. □

Corollary 2.2. *The algorithm* KRUSKAL *computes a minimum spanning tree. It can be implemented to run in time $O(m \log m)$.*

Algorithm 2.2 PRIM

Input. Undirected connected graph $G = (V, E)$, cost function $c : E \to \mathbb{R}$.

Output. Spanning tree T with minimum cost.

Step 1. Choose $v \in V$ and let $T = (\{v\}, \emptyset)$.

Step 2. While $V(T) \neq V(G)$ do: Get minimum cost $e \in \delta(V(T))$ and let $T = T + e$.

Proof. By connectivity of G and since KRUSKAL maintains a spanning forest, it is clear that the algorithm computes a spanning tree T at termination. As it guarantees condition (2), T is optimal.

Sorting the edges requires time $O(m \log m)$. In order to decide if an edge closes a cycle in the current forest, we maintain a UNION-FIND datastructure, e.g., a BOTTOM-UP FOREST. This datastructure supports FIND operations $O(\alpha(m))$ time and UNION operations in $O(1)$ time. Here, α denotes the inverse of the Ackermann function, which grows strictly slower than log. Since there are m FIND operations and $n - 1$ UNION operations we have runningtime $O(m \log m + \alpha(m)m + n) = O(m \log m)$ in total. □

Corollary 2.3. *The algorithm* PRIM *computes a minimum spanning tree. It can be implemented to run in time* $O(m + n \log n)$.

Proof. The correctness follows from the fact that the algorithm ensures property (4).

To yield the claimed running time, it is useful to maintain a FIBONACCI HEAP priority queue. This enables EXTRACT-MIN in $O(\log n)$ amortized time and INSERT in $O(1)$ time. Since there are m INSERT and n EXTRACT-MIN operations, the claimed running time follows. □

2.2 Set Cover

The SET COVER problem this section deals with is a very simple to state – 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 2.4. 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.

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

Problem 2.2 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

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

---

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

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

Step 3. Return x.

Theorem 2.5. *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.

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 2.6. *For each $i \in \{1, \ldots, n\}$,* $\text{price}(e_i) \leq \text{value}(x^*)/(n - i + 1)$.

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

$$\text{price}(e_i) \leq \frac{\text{value}(x^*)}{|U - C|} \leq \frac{\text{value}(x^*)}{n - i + 1}$$

which was claimed. □

Proof of Theorem 2.5. Since the cost of each set is distributed evenly among the new elements covered, the total cost of the set cover picked is

$$\text{value}(x) = \sum_{i=1}^{n} \text{price}(e_i) \leq \text{value}(x^*) H_n,$$

where we have used Lemma 2.6. □

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. We often write $N = (G, c, s, t)$.

For any vertex v, let $\delta^-(v)$ be 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., $\delta^+(v) = \{vu \in A : u \in V\}$. Let $f : A \to \mathbb{R}^+$ be any function on the edges. Define the *balance* $\mathrm{bal}_f(v)$ of vertex v with respect to f by

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

The function f is called *conserving* at a vertex v if $\mathrm{bal}_f(v) = 0$.

The Maximum Flow problem asks to transport as many units from the source to the sink without violating the edge capacities. More precisely, a function $f : A \to \mathbb{R}^+$ is called an *s-t-flow* if:

(1) edge capacities are respected, i.e.,

$$0 \le f(e) \le c(e) \text{ for all } e \in A, \text{ and}$$

(2) f is conserving, i.e.,

$$\mathrm{bal}_f(v) = 0 \text{ for } v \in V - \{s, t\}, \quad \mathrm{bal}_f(s) \ge 0, \quad \text{and} \quad \mathrm{bal}_f(t) \le 0.$$

Its *value* is defined by $\mathrm{value}(f) = \mathrm{bal}_f(s)$. 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.

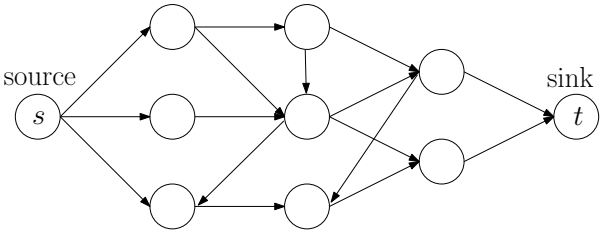


Figure 3.1: A network with source s and sink t.

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

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

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 any algorithm for solving LPs but we are not satisfied with this – we want a combinatorial algorithm (without solving an LP) with guaranteed polynomial running time.

Let S be a subset of the vertices, called a *cut*. The induced *cut-edges* is the set of *outgoing edges* $\delta^+(S) = \{uv \in A : u \in S, v \in V - S\}$ and *incoming edges* $\delta^-(S) = \{vu \in A : u \in S, v \in V - S\}$. Define its *capacity* by $\text{cap}(S) = \sum_{e\in\delta^+(S)} c(e)$. 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. We extend the definition of *balance* also for any cut S:

$$\text{bal}_f(S) = \sum_{e\in\delta^+(S)} f(e) - \sum_{e\in\delta^-(S)} f(e).$$

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. We will see soon 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) $\text{value}(f) = \text{bal}_f(S)$,

(2) $\text{value}(f) \le \text{cap}(S)$.

Proof. We use the flow conservation property, i.e., $\mathrm{bal}_f(v) = 0$ for all $v \in S - \{s\}$ to find

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

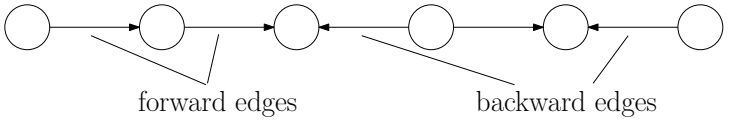
Furthermore we have $\mathrm{value}(f) \leq \sum_{e \in \delta^+(S)} c(e) = \mathrm{cap}(S)$ since $0 \leq f(e) \leq c(e)$. □

The following definitions and structural result are the basis for an algorithm. A *path* is a sequence $P = v_1, e_1, v_2, e_2, \ldots, e_k, v_{k+1}$ alternating between vertices and edges, such that $e_i = v_i v_{i+1} \in A$ or $e_i = v_{i+1} v_i \in A$ for $i = 1, \ldots, k$. The vertices v_1 and v_{k+1} are the *end-vertices* of the path. The number of edges in the path is called its *length*. A path is *simple*, if its vertices are pairwise disjoint. We will always assume that paths are simple, unless stated otherwise. A *v-w-path* P has the form $e_1 = v\cdot$ and $e_\ell = \cdot w$, i.e., it *starts* at v and *ends* at w. An edge $e = vw$ in a path is called *forward edge* if $vw \in A$; *backward edge* if $wv \in A$. (A v-v-path is called a *cycle*.)

An s-v-path P is called *f-augmenting* with respect to a flow f if

(1) $f(e) < c(e)$ for every forward edge $e \in P$,

(2) $f(e) > 0$ for every backward edge $e \in P$.



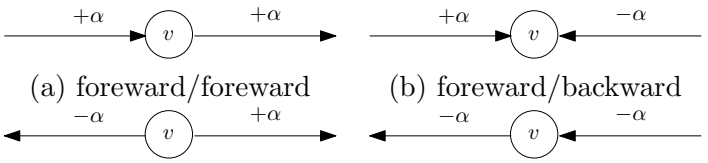
By how much can we increase the current flow value using a particular augmenting path P? Define the quantity

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

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 and because each edge occurs at most once in P, 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 $\mathrm{bal}_{f'}(s) \geq \mathrm{bal}_f(s) \geq 0$ and consequently $\mathrm{bal}_{f'}(t) \leq \mathrm{bal}_f(t) \leq 0$. Consider an augmentation along edges $e_i e_{i+1}$ with $e_i = v_i v_{i+1}$ and $e_{i+1} = v_{i+1} v_{i+2}$ for $i = 1, \ldots, \ell - 1$. Call $v = v_{i+1}$ and distinguish four cases:



(a) foreward/foreward (b) foreward/backward

(c) backward/foreward (d) backward/backward

This yields the claim. □

Algorithm 3.1 FORD-FULKERSON

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

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 value(f') > value(f), i.e., that f is not maximal.

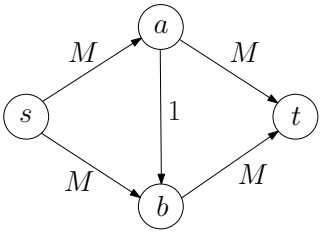
Now let there be no f-augmenting path from s to t. Consider the set S of vertices with augmenting paths from s, i.e., $S = \{v \in V : \text{there is an } f\text{-augmenting path from } s \text{ to } v\}$ and $t \notin S$. Thus S is an $s-t$-cut. By definition of augmenting paths, we must have $f(e)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

If the capacities are not integers, then FORD-FULKERSON might not even terminate. Especially, we have not yet specified how we actually choose the augmenting paths mentioned in Step 2 of the algorithm. This must be done carefully in order to obtain a polynomial time algorithm as the following instance illustrates. It turns out that choosing shortest augmenting paths guarantee termination after a polynomial number of augmentations; see the EDMONDS-KARP algorithm.

Example 3.5. 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.

Edmonds-Karp Algorithm

Example 3.5 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 $c : A \to \mathbb{R}^+$.

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 3.6. *The algorithm* EDMONDS-KARP *computes a maximum $s-t$-flow f in any network N with n vertices and m edges in time $O\left(nm^2\right)$.*

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 3.7. *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 3.6. 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 3.7 $x_v(h) \geq x_v(k)$ 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.

3.2 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. Thus a *network* is denoted $N = (G, c, w, b)$.

Now we define a modified notion of a flow. For any mapping $b : V \to \mathbb{R}$ with $\sum_{v\in V} b(v) = 0$ 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

(1) $0 \leq f(e) \leq c(e)$ for all $e \in A$ and

(2) $b(v) = \mathrm{bal}_f(v) = \sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e)$.

A 0-flow is called a *circulation*.

The *cost* of any flow f is

$$\mathrm{value}(f) = \sum_{e\in A} f(e)w(e).$$

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

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 $c(vt) =$

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.

$\max\{0, -b(v)\}$ 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.

The MINIMUM COST FLOW problem can be solved in polynomial time with an approach 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 3.8. *There is an algorithm which solves the* MINIMUM COST FLOW *problem on any network with n vertices and m edges in time $O\left(m \log m(m + n \log n)\right)$.*

3.3 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.,

$$\text{value}(M) = \sum_{e \in M} w(e),$$

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

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 $\text{value}(M) = \sum_{e \in M} w(e)$ or conclude that no such matching exists.

Theorem 3.9. *The* ASSIGNMENT *problem is a* MINIMUM COST FLOW *problem.*

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. Now every integral b-flow f in N with $b = (b(s), b(v_1), \ldots, b(v_n), b(t)) = (n, 0, \ldots, 0, -n)$ corresponds to a perfect matching in G with the same weight, and vice versa. □

In most applications the requirement $|L| = |R|$ is disturbing, but can usually be handeled by adding artificial vertices and edges.

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.

Chapter 4

Matchings

The theory of matchings is one of the classical topics in graph theory and combintorial optimization. The task of finding a matching occurs frequently as a subproblem of some other problem.

Let G be an undirected graph on the vertices $V(G)$ and the edges $E(G)$. A *matching* is a set $M \subseteq E(G)$ with the property that $e \cap e' = \emptyset$ for all $e \neq e' \in M$.

Problem 4.1 MATCHING

Instance. Undirected graph G on the vertices $V(G)$ and edges $E(G)$.

Task. Find a matching M in G with maximum cardinality, i.e., value$(M) = |M|$.

Let M be any matching in G. Any edge $e \in M$ is called *M-matched*, otherwise *M-free*. A vertex $v \in V(G)$ is *M-covered* if there is an edge $e = v\cdot \in M$. Otherwise v is *M-exposed*.

A matching M is called *maximal* if there is no edge e such that $M \cup \{e\}$ is a matching in G. A matching M is *maximum* if it has largest cardinality among all matchings of G. A matching covering all vertices of a graph is called *perfect*.

4.1 Maximum Matchings and Augmenting Paths

A *path* is a sequence $P = v_1, e_1, v_2, e_2, \ldots, e_k, v_{k+1}$ alternating between vertices and edges, such that $e_i = v_i v_{i+1} \in E$ for $i = 1, \ldots, k$. The vertices v_1 and v_{k+1} are the *end-vertices* of the path. The number of edges in the path is called its *length*. A path is *simple*, if its vertices are pairwise disjoint. We will always assume that paths are simple, unless stated otherwise. For any matching M, a path in which the edges alternate between matched and free edges is called *M-alternating path*. An alternating path where both end-vertices are M-exposed is called an *M-augmenting path*.

Theorem 4.1. *Let G be an undirected graph and M a matching in G. Then M is a maximum matching if and only if there is no M-augmenting path in G.*

Proof. If there is an M-augmenting path P in G, the symmetric difference $M \triangle E(P)$ is a matching and has greater cardinality than M. Thus M is not a maximum matching.

On the other hand, if there is a matching M' with $|M'| > |M|$, the symmetric difference $M \triangle M'$ is a vertex-disjoint union of alternating cycles (having even length) and paths, where at least one path must be M-augmenting. □

4.2 Blossom Algorithm

Theorem 4.1 tells us that one way to find a maximum matching is to find augmenting paths until no such exists. It is not clear how to organize the search such that a polynomial time algorithm is achieved. The central concepts behind the Blossom algorithm of Edmonds are alternating forests and blossoms. These will be introduced shortly. The main result is the following:

Theorem 4.2. *The* Blossom *algorithm computes a maximum matching in* $O(nm)$ *time.*

The main difficulty is the treatment of cycles of odd length. Observe that in any such cycle C, for any matching M, there is always at least one $(M \cap E(C))$-exposed vertex. Surprisingly, it suffices to get rid of an odd cycle by shrinking it to a single vertex.

A graph G is called *factor-critical* if $G - v$ has a perfect matching for each $v \in V(G)$. A matching M is called *near perfect*, if it covers all but one vertex. Observe that a cycle of odd length is factor critical and has a near-perfect matching.

Let G be a graph and M a matching in G. A *blossom* in G with respect to M is a factor-critical subgraph C of G with $|M \cap E(C)| = (|V(C)| - 1)/2$. The vertex of C exposed by $M \cap E(C)$ is called *base* of C.

Lemma 4.3. *Let G be a graph, M a matching in G, and C a blossom in G (with respect to M). Let $v \in V(G)$ be any M-exposed vertex and let r be the base of C. Suppose that there is an M-alternating v-r-path Q of even length with $E(Q) \cap E(C) = \emptyset$.*

Let G' and M' result from G and M by shrinking $V(C)$ to a single vertex. Then M is a maximum matching in G if and only if M' is a maximum matching in G'.

Proof. Suppose that M is not a maximum matching in G. The set $N = M \triangle E(Q)$ is a matching of the same cardinality, so it is not maximum either. By Theorem 4.1 there then exists an N-augmenting path P in G. Note that N does not cover r.

At least one of the end-vertices of P, say x does not belong to C. If P and C are disjoint, let y be the other end-vertex of P. Otherwise, let y be the first vertex on P – when traversed from x – belonging to C. Let G' result from G when shrinking $V(C)$. Let P' and N' in G' correspond to P and N in G. The end-vertices of P' are exposed by N'. Hence P' is an N' augmenting path in G'. So N' is not a maximum matching in G', and nor is M' (as it has the same cardinality).

To prove the converse, suppose that M' is not a maximum matching in G'. Let N' be a larger matching in G'. N' corresponds to a matching $\bar{N}$ in G which covers at most one vertex C in G. Since C is factor critical, $\bar{N}$ can be extended by $k = (|V(C)| - 1)/2$ many edges to a matching N in G, where

$$|N| = |\bar{N}| + k = |N'| + k > |M'| + k = |M|.$$

This shows that M is not a maximum matching in G. □

Given a graph G and a matching M in G. An *M-alternating forest* in G is a forest F in G with the following properties:

(1) $V(F)$ contains all the M-exposed vertices. Each connected component of F contains exactly one exposed vertex, called its *root*.

(2) We call a vertex $v \in V(F)$ an *outer* (*inner*) vertex, if it has even (odd) distance to the root of the connected component containing v. (In particular, the roots are outer vertices.) All inner vertices have degree 2 in F.

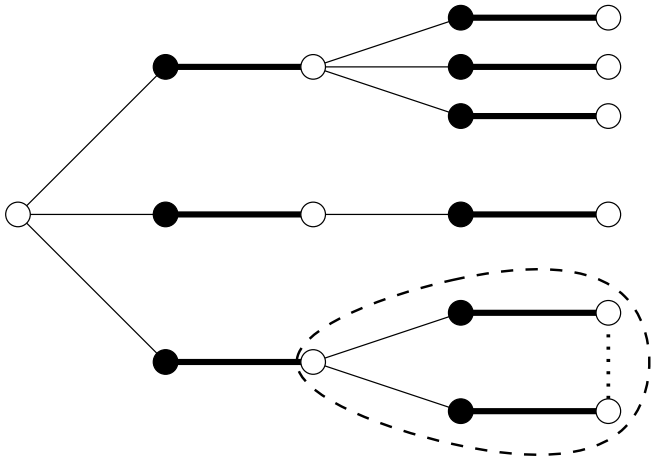


Figure 4.1: An alteranting forest (consisting of a single tree) with root r. The dotted edge forms a blossom (indicated by the dashed line) with base b. Observe that the unique path from b to r is even and alternating.

(3) For any $v \in V(F)$, the unique path from v to the root of the connected component containing v is M-alternating.

See Figure 4.1 for an illustration of an alternating forest and a blossom.

Observation 4.4. *In any alternating forest, the number of outer vertices that are not root equals the number of inner vertices.*

Proof. Each outer vertex that is not a root has exactly one neighbor which is an inner vertex and whose distance to the root is smaller. This is obviously a bijection between the outer vertices that are not a root and the inner vertices. □

The Blossom algorithm works as follows: Given some matching M, we build up an M-alternating forest F. We start with the set S of exposed vertices and no edges.

At any stage of the algorithm, we consider a neighbor v of an outer vertex u. Let $P(u)$ denote the unique path in F from u to a root. There are three interesting cases, corresponding to three operations "grow", "augment", and "shrink":

Case 1. $v \notin V(F)$. Then we add the edge uv and the matching edge $vw \in M$ to F. Thus the forest will grow.

Case 2. v is an outer vertex in a different component than u. Then we augment along the path $P(u) \cup \{uv\} \cup P(v)$.

Case 3. v is an outer vertex in the same connected component of F (with root r, say). Let ℓ be the first vertex of $P(u)$ (starting at u) which also belongs to $P(v)$. The vertex ℓ can be one of u and v. If ℓ is not a root, then it must have degree at least three. Thus it is an outer vertex then. If ℓ is a root, it is an outer vertex by definition. Therefore $C = P(u)[u, \ell] \cap \{uv\}P(v)[v, \ell]$ is a blossom with at least three vertices. We shrink C.

If none of the cases applies, all neighbors of outer vertices are inner. Then the algorithm terminates and returns the matching M found.

Proof of Theorem 4.2. We claim that M is maximum. Consider an alternating forest F that belongs to a maximum matching. Let S be the set of outer vertices and let T be the set of inner vertices of F. Denote $s = |S|$ and $t = |T|$. $G - T$ has t odd components (each outer vertex is isolated in $G - T$). Hence any matching must leave at least $s - t$ vertices uncovered. But on the other hand, the number of vertices exposed by M, i.e., the number of roots in F, is exactly $s - t$ by Observation 4.4.

For the running time observe that there are at most n augmentations taking time $O(m)$ each. Growing the forest takes time $O(1)$, which can occur at most n times between two augmentations. Shrinking a blossom C takes time $O(|V(C)|)$ which yields that the forest has $|V(C)| - 1 > 1$ vertices less. Hence the total effort for shrinkings between two augmentations is at most $O(n)$ (with a suitable implementation). Checking the other edges, that do not correspond to any of the three cases, takes total time $O(m)$ between two augmentations. □

The implementation of the Blossom algorithm is not an easy task. For this reason, we do not go into the details here.

Chapter 5

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 j comes along with a profit c_j and a weight w_j. We are asked to choose a subset of the items as to maximize total profit but the total weight not exceeding W.

Example 5.1. 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-weight 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 5.1 KNAPSACK

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

Task. Solve the problem

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

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

The number w_j is called the *weight* of item j. The *weight* of a vector $x \in \{0,1\}^n$ is given by $\text{weight}(x) = \sum_{j=1}^n w_j x_j$. 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 5.1 we derive a simple 1/2-approximation algorithm. In Section 5.3 we can even improve on this by giving a polynomial-time $1-\varepsilon$-approximation algorithm (for every fixed $\varepsilon > 0$).

5.1 Fractional Knapsack and Greedy

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

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

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

Observation 5.2. *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*

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

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 5.2 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 5.3. 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 value$(x) = 1$. The optimum solution is $x^* = (0, 1)$ and thus value$(x^*) = W - 1$. 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 5.1 GREEDY

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

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

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 $j \neq k$.

Step 3. If value$(x) \geq$ value(y) return x otherwise return y.

Theorem 5.4. *The algorithm* GREEDY *is a* 1/2*-approximation for* KNAPSACK.

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

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

$$\text{value}(x^*) \leq \text{value}(z^*),$$

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

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

which implies the approximation ratio of 1/2. □

5.2 Pseudo-Polynomial Time Algorithm

Here we give a pseudo-polynomial time algorithm that solves KNAPSACK optimally by using dynamic programming. The term *pseudo-polynomial* means polynomial if the input is given in unary encoding (and thus exponential if the input is given in binary encoding).

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 $\text{weight}(x) \leq W$ and $\text{value}(x) = k$. That is, after defining the set $W_{j,k} = \{\text{weight}(x) : \text{weight}(x) \leq W, \text{value}(x) = k, x \in \{0,1\}^j 0^{n-j}\}$ 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 5.2 DYNAMIC PROGRAMMING KNAPSACK

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

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

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

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

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 5.5. *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 $W_{j,k} = \{\text{weight}(x) : \text{weight}(x) \leq W, \text{value}(x) = k, x \in \{0,1\}^j 0^{n-j}\}$ 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 $\text{value}(x) = k$. By definition of the algorithm we have $\text{weight}(x) = \text{weight}(y) + w_j = m_{j-1,k-c_j} + w_j \leq W$ 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. □

5.3 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 $O(nC)$. 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 to $O(nC/t)$ but may yield suboptimal solutions.

Algorithm 5.3 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 $\text{value}(x) = 0$ then return x.

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

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

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

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

Theorem 5.6. *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 $\max\{\text{value}(x), \text{value}(y)\}$. Let x^* be an optimum solution for the instance W, w, c. By Theorem 5.4 we have $2\text{value}(x) \geq \text{value}(x^*)$ and hence the choice $C = 2\text{value}(x)/t$ is a legal upper bound for the optimum value of the rounded instance W, w, c'. By Theorem 5.5 y is an optimum solution for this instance and we have

$$\begin{aligned}\text{value}(y) &= \sum_{j=1}^{n} c_j y_j \geq \sum_{j=1}^{n} t c'_j y_j = t \sum_{j=1}^{n} c'_j y_j \\ &\geq t \sum_{j=1}^{n} c'_j x^*_j = \sum_{j=1}^{n} t c'_j x^*_j > \sum_{j=1}^{n} (c_j - t) x^*_j \geq \text{value}(x^*) - nt.\end{aligned}$$

If $t = 1$ then y is optimal by Theorem 5.5. Otherwise the above inequality and the choice of t yields $\text{value}(y) \geq \text{value}(x^*) - \varepsilon\text{value}(x)$ and hence

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

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

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

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

where we have used the definition of t: If $t = 1$ then $\text{value}(x) \leq n/\varepsilon$ and otherwise $t = \varepsilon\text{value}(x)/n$. This running time dominates the time needed for the other steps. □

Chapter 6

Makespan Scheduling

In this chapter, we consider the classical Makespan Scheduling problem. We are given m machines for scheduling, indexed by the set $M = \{1, \ldots, m\}$. There are furthermore given n jobs, indexed by the set $J = \{1, \ldots, n\}$, where job j takes $p_{i,j}$ units of time if scheduled on machine i. Let J_i be the set of jobs scheduled on machine i. Then $\ell_i = \sum_{j \in J_i} p_{i,j}$ is the *load* of machine i. The maximum load $\ell_{\max} = c_{\max} = \max_{i \in M} \ell_i$ is called the *makespan* of the schedule.

The problem is NP-hard, even if there are only two identical machines. However, we will derive several constant factor approximations and a PTAS for identical machines and a 2-approximation for the general case.

6.1 Identical Machines

In the special case of *identical machines*, we have that $p_{i,j} = p_j$ for all $i \in M$ and all $j \in J$. Here p_j is called the *length* of job j.

List Scheduling

As a warm-up we consider the following two heuristics for Makespan Scheduling. The List Scheduling algorithm works as follows: Determine any ordering of the job set J, stored in a list L. Starting with all machines empty, determine the machine i with the currently least load and schedule the respective next job j in L on i. The load of i before the assignment of j is called the *starting time* s_j of job j and the load of i after the assignment is called the *completion time* c_j of job j. In the Sorted List Scheduling algorithm we execute List Scheduling, where the list L consists of the jobs in decreasing order of length.

Theorem 6.1. *The* List Scheduling *algorithm is a* 2*-approximation for* Makespan Scheduling *on identical machines.*

Proof. Let T^* be the optimal makespan of the given instance. We show that $s_j \leq T^*$ for all $j \in J$. This implies $c_j = s_j + p_j \leq T^* + p_j \leq 2 \cdot T^*$ for all $j \in J$, since we clearly must have $T^* \geq p_j$ for all $j \in J$.

Assume that $s_j > T^*$ for some $j \in J$. Then we have that the load *before* the assignment of j is $\ell_i > T^*$ for all $i \in M$. Thus the jobs $J' \subseteq J$ scheduled before j by the algorithm have total length $\sum_{j' \in J'} p_{j'} > m \cdot T^*$. On the other hand, since the optimum solution schedules all jobs J until time T^* we have $\sum_{j \in J} p_j \leq m \cdot T^*$. A contradiction and the List Scheduling algorithm must start all jobs not later than time T^*. □

Here we show that Sorted List Scheduling is a 3/2-approximation, but one can actually prove that the algorithm is a 4/3-approximation.

Theorem 6.2. *The* Sorted List Scheduling *algorithm is a* 3/2*-approximation for* Makespan Scheduling *on identical machines.*

Proof. Let T^* be the optimal makespan of the given instance. Partition the jobs $J_L = \{j \in J : p_j > T^*/2\}$ and $J_S = J - J_L$, called *large* and *small* jobs. Notice that there can be at most m large jobs: Assume that there are more than m such jobs. Then, in any schedule, including the optimal one, there must be at least two such jobs scheduled on some machine. Since the length of a large job is more than $T^*/2$, this contradicts that T^* is the optimal makespan.

Since there are at most m large jobs and the algorithm schedules those first and hence on individual machines, we have that each large job completes not later than T^*, i.e., $c_j \leq T^*$ for all $j \in J_L$. Thus, if a job completes later than T^* it must be a small job having length at most $T^*/2$. Since each job starts not later than T^* we have $c_j \leq T^* + p_j \leq 3/2 \cdot T^*$ for every small job $j \in J_S$. □

Polynomial Time Approximation Scheme

In this section we give a *polynomial time approximation scheme (PTAS)* for Makespan Scheduling on identical machines. This means, for any error parameter $\varepsilon > 0$, there is an algorithm which determines a $(1+\varepsilon)$-approximate solution with running time polynomial in the input size, but arbitary in $1/\varepsilon$. We will give a PTAS with running time $O\left(n^{2k} \cdot \lceil \log_2 1/\varepsilon \rceil\right)$, where $k = \lceil \log_{1+\varepsilon} 1/\varepsilon \rceil$.

The two main ingredients of the algorithm are these:

(1) Firstly, assume that we are given the optimal makespan T^* at the outset. Then we can try to construct a schedule with makespan at most $(1+\varepsilon) \cdot T^*$. But how do we determine the number T^*? It turns out that we can perform *binary search* in an interval $[\alpha, \beta]$, where α is any lower bound on T^* and β any upper bound on T^*. This binary search will enable us to eventually find a number B, which is within $(1+\varepsilon)$ times T^* and where the number of binary search iterations depends on the error parameter ε.

(2) Secondly, assume that the number of *distinct* values of job lengths is a constant k, say. Then we can determine all configurations of jobs that do not violate a load bound of t if scheduled on a single machine. This is the basis of a dynamic programming scheme to determine a schedule on m machines. Of course, this approach involves rounding the original job lengths to constantly many values, which introduces some error. The error can be controled by adjusting the constant k of distinct job lengths at the expense of running time and space requirement for the dynamic programming table.

Consider the instance J with jobs of lengths $p_1, \ldots, p_n$. Let t be a parameter and let $m(J,t)$ be the smallest number of machines required to schedule the jobs J having makespan at most t. With this definition, the minimum makespan T^* is given by $T^* = \min\{t : m(J,t) \leq m\}$. We will later perform binary search on the parameter t, in the interval $[\alpha, \beta]$, where $\alpha = \max\{\max_{j \in J} p_j, 1/m \cdot \sum_{j \in J} p_j\}$ and $\beta = 2 \cdot \alpha$. Notice that α is a lower bound on T^* and β an upper bound on T^*.

Dynamic Programming. Assume for now that $|\{p_1, \ldots, p_n\}| = k$, i.e., there are k distinct job lengths. Fix an ordering of the jobs lengths. Then a k-tuple $(i_1, \ldots, i_k)$ describes for any $\ell \in \{1, \ldots, k\}$ the number i_ℓ of jobs having the respective length. For any k-tuple $(i_1, \ldots, i_k)$ let $m(i_1, \ldots, i_k, t)$ be the smallest number of machines needed to schedule these jobs having makespan at most t. For a given parameter t and an instance $(n_1, \ldots, n_k)$ with $\sum_{\ell=1}^{k} n_\ell = n$, we first compute the set Q of all k-tuples $(q_1, \ldots, q_k)$ such that $m(q_1, \ldots, q_k, t) = 1$, $0 \leq q_\ell \leq n_\ell$ for $\ell = 1, \ldots, k$, i.e., all sets of jobs that can be scheduled on a single machine with makespan at most t. Clearly, Q contains at most $O\left(n^k\right)$ elements. Having these numbers computed, we determine the entries $m(i_1, \ldots, i_k, t)$ for every $(i_1, \ldots, i_k) \in \{0, \ldots, n_1\} \times \cdots \times \{0, \ldots, n_k\}$ of a k-dimensional table as follows: The table is initialized by setting $m(q, t) = 1$ for every $q \in Q$. Then we use the following recurrence to compute the remaining entries:

$$m(i_1, \ldots, i_k, t) = 1 + \min_{q \in Q} m(i_1 - q_1, \ldots, i_k - q_k, t).$$

Computing each entry takes $O\left(n^k\right)$ time. Thus the entire table can be computed in $O\left(n^{2k}\right)$ time, thereby determining $m(n_1, \ldots, n_k, t)$ in polynomial time provided that k is a constant.

Rounding. Let $\varepsilon > 0$ be an error parameter and let $t \in [\alpha, \beta]$ as defined above. We say that a job j is small if $p_j < \varepsilon \cdot t$. Small jobs are removed from the instance for now. The rest of the job lengths are rounded down as follows: If a job j has length $p_j \in [t \cdot \varepsilon \cdot (1+\varepsilon)^i, t \cdot \varepsilon \cdot (1+\varepsilon)^{i+1})$ for $i \geq 0$, it is replaced by $p'_j = t \cdot \varepsilon \cdot (1+\varepsilon)^i$. Thus there can be at most $k = \lceil \log_{1+\varepsilon} 1/\varepsilon \rceil$ many distinct job lengths. Now we invoke the above dynamic programming scheme and determine a the optimal number of machines for scheduling these jobs if the makespan is at most t. Since the rounding reduces the length of each job by a factor of at most $(1+\varepsilon)$, the computed schedule has makespan at most $(1+\varepsilon) \cdot t$ when considering the original job lengths. Now we schedule the small jobs greedily in leftover space and open new machines if needed. Clearly, whenever a new machine is opened, all previous machines must be loaded to an extent of at least t. Denote by $a(J, t, \varepsilon)$ the number of machines used by this algorithm. Recall that the makespan is at most $(1+\varepsilon) \cdot t$.

Lemma 6.3. *We have that $a(J, t, \varepsilon) \leq m(J, t)$.*

Proof. If the algorithm does not open any new machines for small jobs, then the assertion clearly holds since the rounded down jobs have been scheduled optimally with makespan t. In the other case, all but the last machine are loaded to the extent of t. Hence, the optimal schedule of J having makespan t must also use at least $a(J, t, \varepsilon)$ machines. □

Corollary 6.4. *We have that $T = \min\{t : a(J, t, \varepsilon) \leq m\} \leq \min\{t : m(J, t) \leq m\} = T^*$.*

Binary Search. If T could be determined with no additional error during the binary search, then clearly we could use the above algorithm to obtain a schedule with makespan at most $(1+\varepsilon) \cdot T^*$. Next, we will specify the details of the binary search and show how to control the error it introduces. The binary search is performed in the interval $[\alpha, \beta]$ as defined above. Thus, the length of the available interval is $\beta - \alpha = \alpha$ at the start of the search and it reduces by a factor of two in each iteration. We continue the search until it drops to a length of at most $\varepsilon \cdot \alpha$. This will require $\lceil \log_2 1/\varepsilon \rceil$ many iterations. Let B be the right endpoint of the interval $[A, B]$ we terminate with.

Lemma 6.5. *We have that* $B \leq (1+\varepsilon) \cdot T^*$.

Proof. Clearly $T = \min\{t : a(J,t,\varepsilon) \leq m\}$ must be in the interval $[B - \varepsilon \cdot \alpha, B]$. Hence

$$B \leq T + \varepsilon \cdot \alpha \leq (1+\varepsilon) \cdot T^*,$$

where we have used $T^* \geq \alpha$ and Corollary 6.4. □

For $t \leq B$ this directly gives the result we wanted to show:

Theorem 6.6. *For any* $0 < \varepsilon \leq 1$ *the algorithm produduces a schedule with makespan at most* $(1+\varepsilon)^2 \cdot T^* \leq (1+3\varepsilon) \cdot T^*$ *within running time* $O\left(n^{2k} \cdot \lceil \log_2 1/\varepsilon \rceil\right)$, *where* $k = \lceil \log_{1+\varepsilon} 1/\varepsilon \rceil$.

6.2 Unrelated Machines

Here we give a 2-approximation algorithm for Makespan Scheduling on *unrelated machines*, which means that job j takes time $p_{i,j}$ if scheduled on machine i. The algorithm is based on a suitable LP-formulation and a procedure for rounding the LP.

An obvious integer program for this problem is the following: Let $x_{i,j}$ be a variable indicating if job j is assigned to machine i. The objective is to minimize the makespan. The first set of constraints ensures that each job is scheduled on one of the machines and the second ensures that each machine has a load of at most t.

$$\begin{aligned} \text{minimize} \quad & t \\ \text{subject to} \quad & \sum_{i \in M} x_{i,j} = 1 \quad j \in J, \\ & \sum_{j \in J} p_{i,j} x_{i,j} \leq t, \quad i \in M, \\ & x_{i,j} \in \{0,1\}. \end{aligned}$$

If we relax the constraints $x_{i,j} \in \{0,1\}$ to $x_{i,j} \in [0,1]$, it turns out that this formulation has unbounded integrality gap. (It is left as an exercise to show this.) The main cause of the problem is an "unfair" advantage of the LP-relaxation: If $p_{i,j} > t$, then we must have $x_{i,j} = 0$ in any feasible integer solution, but we might have $x_{i,j} > 0$ in feasible fractional solutions. However, we can not formulate the statement "if $p_{i,j} > t$ then $x_{i,j} = 0$" in terms of linear constraints.

Parametric Pruning. We will make use of a technique called *parametric pruning* to overcome this difficulty. Let the parameter t be a "guess" of a lower bound for the actual makespan T^*. Of course, we will do binary search on t in order to determine a suitable value in an outside loop. However, having a value for t fixed, we are now able to enforce constraints $x_{i,j} = 0$ for all machine-job pairs i, j for which $p_{i,j} > t$. Define $S_t = \{(i,j) : p_{i,j} \leq t\}$. We now define a family $\text{LP}(t)$ of linear programs, one for each value of the parameter t. $\text{LP}(t)$ uses the variables $x_{i,j}$ for which $(i,j) \in S_t$ and asks if there is a feasible

solution using the restricted assignment possibilities, only.

$$\begin{aligned} \text{minimize} \quad & 0 && (\text{LP}(t)) \\ \text{subject to} \quad & \sum_{i:(i,j)\in S_t} x_{i,j} = 1 \quad j \in J, \\ & \sum_{j:(i,j)\in S_t} p_{i,j} x_{i,j} \leq t, \quad i \in M, \\ & x_{i,j} \geq 0 \quad (i,j) \in S_t. \end{aligned}$$

Extreme Point Solutions. With a binary search, we find the smallest value for t such that LP(t) has a feasible solution. Let T be this value and observe that $T^* \geq T$, i.e., the actual makespan is bounded from below by T. Our algorithm will "round" an extreme point solution of LP(T) to yield a schedule with makespan at most $2 \cdot T^*$. Extreme point solutions to LP(T) have several useful properties.

Lemma 6.7. *Any extreme point solution to* LP(T) *has at most* $n + m$ *many non-zero variables.*

Proof. Let $r = |S_T|$ represent the number of variables on which LP(T) is defined. Recall that a feasible solution is an extreme point solution to LP(T) if and only if it sets r many linearly independent constraints to equality. Of these r linearly independent constraints, at least $r - (n + m)$ must be chosen from the third set of constraints, i.e., of the form "$x_{i,j} \geq 0$". The corresponding variables are set to zero. So, any extreme point solution has at most $n + m$ many non-zero variables. □

Let x be an extreme point solution to LP(T). We will say that job j is *integrally set* if $x_{i,j} \in \{0, 1\}$ for all machines i. Otherwise, i.e., $x_{i,j} \in (0, 1)$ for some machine i, job j is said to be *fractionally set*.

Corollary 6.8. *Any extreme point solution to* LP(T) *must set at least* $n - m$ *many jobs integrally.*

Proof. Let x be an extreme point solution to LP(T) and let α and β be the number of jobs that are integrally and fractionally set by x, respectively. Each job of the latter kind is assigned to at least 2 machines and therefore results in at least 2 non-zero entries in x. Hence we get $\alpha + \beta = n$ and $\alpha + 2\beta \leq n + m$. Therefore $\beta \leq m$ and $\alpha \geq n - m$. □

Algorithm. The algorithm starts by computing the range in which it finds the right value for T. For this it constructs a greedy schedule, in which each job is assigned to the machine on which it has the smallest length. Let α be the makespan of this schedule. Then the range is $[\alpha/m, \alpha]$ (and it is an exercise to show that α/m is indeed a lower bound on T^*).

The LP-rounding algorithm is based on several interesting properties of extreme point solutions of LP(T), which we establish now. For any extreme point solution x for LP(T) define a bipartite graph $G = (M \cup J, E)$ such that $(i, j) \in E$ if and only if $x_{i,j} > 0$. Let $F \subseteq J$ be the fractionally set jobs in x and let H be the subgraph of G induced by the vertex set $M \cup F$. Clearly $(i, j) \in E(H)$ if $0 < x_{i,j} < 1$. A matching in H is called *perfect* if it matches every job $j \in F$. We will show and use that the graph H admits perfect matchings.

We say that a connected graph on a vertex set V is a *pseudo tree* if it has at most $|V|$ many edges. Since the graph is connected, it must have at least $|V| - 1$ many edges. So,

Algorithm 6.1 SCHEDULE UNRELATED

Input. J, M, $p_{i,j}$ for all $i \in M$ and $j \in J$

Output. $x_{i,j} \in \{0,1\}$ for all $i \in M$ and $j \in J$

Step 1. By binary search in $[\alpha/m, \alpha]$ compute smallest value T of the parameter t such that LP(t) has a feasible solution.

Step 2. Let x be an extreme point solution for LP(T).

Step 3. Construct graph H and find perfect matching P.

Step 4. Round in x all fractionally set jobs according to the matching P.

---

it is either a tree or a tree with an additional single edge (closing exactly one cycle). A graph is a *pseudo forrest* if each of its connected components is a pseudo tree.

Lemma 6.9. *We have that G is a pseudo forrest.*

Proof. We will show that the number of edges in each connected component of G is bounded by the number of vertices in it. Hence, each connected component is a pseudo tree.

Consider a connected component G_c. Restrict LP(T) and the extreme point solution x to the jobs and machines of G_c, only, to obtain LP$_c$(T) and x_c. Let $x_{\bar{c}}$ represent the rest of x. The important observation is that x_c must be an extreme point solution for LP$_c$(T). Suppose that this is not the case. Then, x_c is a convex combination of two feasible solutions to LP$_c$(T). Each of these, together with $x_{\bar{c}}$ form a feasible solution for LP(T). Therefore x is a convex combination of two feasible solutions to LP(T). But this contradicts the fact that x is an extreme point solution. With Lemma 6.7 G_c is a pseudo tree. □

Lemma 6.10. *Graph H has a perfect matching P.*

Proof. Each job that is integrally set in x has exactly one edge incident at it in G. Remove these jobs together with their incident edges from G. The resulting graph is clearly H. Since an equal number of edges and vertices have been removed from the pseudo forrest G, H is also a pseudo forrest.

In H, each job has a degree of at least two. So, all leaves in H must be machines. Keep matching a leaf with the job it is incident to and remove them both from the graph. (At each stage all leaves must be machines.) In the end we will be left with even cycles (since we started with a bipartite pseudo forrest.) Match alternating edges of each cycle. This gives a perfect matching P. □

Theorem 6.11. *Algorithm* SCHEDULE UNRELATED *is a 2-approximation for* MAKESPAN SCHEDULING *on unrelated machines.*

Proof. Clearly $T \leq T^*$ since LP(T^*) has a feasible solution. The extreme point solution x to LP(T) has a fractional makespan of at most T. Therefore, the restriction of x to integrally set jobs has an integral makespan of at most T. Each edge (i,j) of H satisfies $p_{i,j} \leq T$. The perfect matching found in H schedules at most one extra job on each machine. Hence, the total makespan is at most $2 \cdot T \leq 2 \cdot T^*$ as claimed. The algorithm clearly runs in polynomial time. □

It is an exercise to show that the analysis is tight for the algorithm.