These course notes are based on a course »Advanced Combinatorial Algorithms« which was held in August/September 2003 at the Escuela Politécnica Nacional Quito, Ecuador, in the Programa Individual de Doctorado en Matemática Aplicada.

I would be happy to receive feedback, in particular suggestions for improvement and notifications of typos and other errors.

Sven O. Krumke

krumke@zib.de
## Contents

1 Introduction 3  
1.1 Acknowledgements 3  
1.2 Additional Literature 3  

1 Network Flows 5  

2 Fast Algorithms for Computing Maximum Flows 7  
2.1 An Elementary Problem 7  
2.2 Notation and Preliminaries 7  
2.3 Residual Networks and Augmenting Paths 9  
2.4 Maximum Flows and Minimum Cuts 10  
2.5 Network Flows and LP-Duality 14  
2.6 Flow Feasibility Problems 15  
2.7 A Review of Basic Algorithms 16  
2.8 Preflow-Push Algorithms 19  
2.8.1 Bounding the Number of Relabel Operations 24  
2.8.2 Bounding the Number of Pushes 25  
2.8.3 Time Complexity of the Generic Algorithm 26  
2.8.4 The FIFO-Preflow-Push Algorithm 27  
2.9 Exercises 28  

3 Minimum Cost Flows 31  
3.1 Introduction 31  
3.2 Disgression: Total Unimodularity and Applications of the Minimum Cost Flow Problem 32  
3.3 Flow Decomposition 34  
3.4 Disgression: Multicommodity Flows and Column Generation 35  
3.4.1 A Review of the Simplex Method 36  
3.4.2 Column Generation for the Multicommodity Flow Problem 37  
3.5 LP-Duality and Optimality Conditions 38  
3.6 The Cycle Cancelling Algorithm of Klein 41  
3.7 The Successive Shortest Path Algorithm 41  
3.8 The Capacity Scaling Algorithm 43  
3.9 The Successive Scaling Algorithm 46  
3.10 The Scale-and-Shrink Algorithm 48  
3.11 Exercises 54
## 4 Dynamic Network Flows

4.1 Basic Definitions ................................................. 55
4.2 Dynamic Flows and Cuts over Time .......................... 57
4.3 Temporally Repeated Flows .................................. 58
4.4 The Dynamic Max-Flow-Min-Cut-Theorem .................. 60
4.5 The Discrete Time Model and Time-Expanded Networks ... 62
4.6 Exercises .................................................. 64

## II Online Optimization

5 Introduction ................................................. 67
5.1 A Simple Problem? ........................................ 67
5.2 Online Computation ........................................... 68
5.3 Competitive Analysis ........................................ 71
5.3.1 Deterministic Algorithms ................................. 71
5.3.2 Randomized Algorithms ................................. 72
5.4 Exercises .................................................. 75

6 Examples and Elementary Techniques ............... 77
6.1 Cruel Adversaries ........................................ 77
6.2 Potential Functions ........................................ 78
6.3 Averaging over Adversaries ................................ 79
6.4 Yao’s Principle and Lower Bounds for Randomized Algorithms .................. 80
6.4.1 Two-Person Zero-Sum Games ............................ 81
6.4.2 Yao’s Principle and its Application to Online Problems ... 83
6.5 Exercises .................................................. 84

7 The Paging Problem ........................................ 87
7.1 An Optimal Offline Algorithm .............................. 87
7.2 Deterministic Algorithms .................................... 88
7.2.1 Phase Partitions and Marking Algorithms ............. 89
7.2.2 A Lower Bound for Deterministic Algorithms ....... 90
7.3 Randomized Algorithms ...................................... 91
7.4 The k-Server Problem ....................................... 93
7.5 Exercises .................................................. 96

8 Transportation Problems .................................... 99
8.1 Online Dial-a-Ride Problems ............................... 99
8.2 Minimizing the Makespan ................................... 100
8.2.1 Lower Bounds ......................................... 101
8.2.2 Two Simple Strategies .................................. 102
8.2.3 A Best-Possible Online-Algorithm ................. 106
8.3 Minimizing the Sum of Completion Times ............... 110
8.3.1 A Deterministic Algorithm ............................. 110
8.3.2 An Improved Randomized Algorithm ............... 113
8.4 Alternative Adversary Models ....................... 115
8.4.1 The Fair Adversary .................................... 115
8.4.2 The Non-Abusive Adversary ........................... 119
8.5 Exercises .................................................. 120
## III Scheduling

### 9 Introduction
- 9.1 A Hard Problem .................................................. 123
- 9.2 Scheduling Problems ........................................... 124
- 9.3 Approximation Algorithms and Schemes .................. 125
- 9.4 List-Scheduling .................................................. 125
- 9.5 Minimizing Project Duration ................................. 127
- 9.6 Just-in-Time Scheduling ......................................... 128
- 9.7 Exercises ......................................................... 129

### 10 The Single Machine
- 10.1 Scheduling Rules ................................................ 131
- 10.2 Using a Preemptive Schedule ............................... 133
- 10.3 Linear Programming ........................................... 134
- 10.4 $\alpha$-Point Scheduling ................................. 136
  - 10.4.1 Preemptive List Scheduling ......................... 136
  - 10.4.2 A Randomized $\alpha$-Scheduler .................. 138
- 10.5 Exercises ......................................................... 139

### 11 Parallel Machines
- 11.1 Scheduling Rules .............................................. 141
- 11.2 Application of Matching Techniques .................... 142
- 11.3 Linear Programming .......................................... 144
- 11.4 More Lower Bounds on the Optimal Schedule ........ 144
- 11.5 Relaxed Decision Procedures and Linear Programming 146
- 11.6 Preemptive and Fractional Schedules as Bases for Approximation 149
- 11.7 Preemptive scheduling on parallel machines .......... 150
- 11.8 Non-preemptive scheduling on parallel machines .... 151
- 11.9 Dynamic Programming and Polynomial Time Approximation Schemes 153

**Bibliography** .................................................. 157
Acknowledgements
Introduction

These lecture notes deal with selected topics from the theory of combinatorial algorithms. The notes are by no means intended to give a complete overview over the beauty of this area. Rather, the notes reflect some of my preferences while still keeping a thread to guide through the material. The course notes are divided into three parts.

Part I deals with network flows. Chapter 2 gives an introduction to the basic terminology of flows and then gives several algorithms to compute a maximum flow. As a »must« we also prove the famous Max-Flow-Min-Cut-Theorem which has numerous applications in combinatorics. In Chapter 3 we study minimum cost flows. Chapter 4 is concerned with dynamic flows (also known as flows over time).

Part II is about online optimization, that is, optimization where the input is provided piece-by-piece and decisions have to be made on the base of incomplete data. We study classical results about competitive analysis and its relation to game theory.

Part III presents results from the area of scheduling. In addition to studying the complexity and approximability of scheduling problems we will again be haunted by online-optimization since many scheduling problems are in fact online-problems.

The material in these lecture notes is based on a two week compact course taught at the Escuela Politécnica Nacional in Quito, Ecuador, in the Programa Individual de Doctorado en Matemática Aplicada. Each day consisted of three hours of lecturing plus three hours of exercises with discussions in the afternoon. I reckon that the present material would approximately suffice for teaching a »regular« one-semester course with four hours a week.

1.1 Acknowledgements

First of all my thanks go to Nicole Megow for providing immense feedback. Moreover, Nicole greatly contributed to the scheduling chapter in writing parts of and for guiding me through the jungle of machine scheduling. I would never have completed a first version of Part II in time without her help. Nicole also contributed a lot to the exercises and solutions that can be found in this document. I would also like to thank Polo Vaca from the Escuela Politécnica Nacional Quito for constructive advice from which I have benefited a lot. Last but not least, I thank all the participants of the course in Quito for enduring the course and providing an excellent atmosphere and teaching environment.

1.2 Additional Literature

There are numerous good text books available which cover more or less the topics presented in these notes. For network flows, the book of Ahuja, Magnanti and Orlin is a great source of information. Online-Optimization is a comparatively new area, but the books
5-10 provide an excellent treatment of the material. A classical article about scheduling is 12. You can also find a lot of useful stuff in the books 6-25, 13, 14.
Part I

Network Flows
2.1 An Elementary Problem

The Italian ice cream company DETOUR produces only one flavor of ice cream, vanilla. There are a couple of machines producing the ice and they are all connected via a pipe network to an output valve from which the product is shipped to the customers. The pipes have different capacities such that different amounts of ice cream can be pushed through them. Figure 2.1 illustrates the situation. Luigi wants to route the ice cream through the pipe network such that as much ice cream as possible reaches the loading dock. How can he do that?

![Network Diagram]

Figure 2.1: Production and pipe network of the ice cream company DETOUR.

It turns out that Luigi’s problem is an instance of the maximum flow problem which we are going to study in this chapter. As we will see there exist efficient algorithms for solving this problem, so Luigi’s current task can be accomplished quickly (if you know how). Stay tuned!

2.2 Notation and Preliminaries

Let $G = (V, A)$ be a directed graph with arc capacities $u: A \rightarrow \mathbb{R}_{\geq 0}$. In all what follows we assume that the following conditions are met:
1. We assume that $G$ does not contain parallel arcs.

2. For each arc $(i, j) \in A$ also the inverse arc $(j, i)$ is contained in the arcset $A$.

Both assumptions do not impose any loss of generality, but allow us to avoid clutter in notation. We can always enforce them by suitable network transformations. Figure 2.2 illustrates how parallel arcs can be removed from the network by adding new nodes. Each of the parallel arcs is »split« into two arcs with a new node in between. The second assumption can always be enforced trivially by adding non existent inverse arcs with capacity zero.

![Figure 2.2: Parallel arcs can be removed by adding new nodes.](image)

Let $f : A \to \mathbb{R}_{\geq 0}$ be a function assigning values to the arcs of the network. We imagine $f(a)$ to be the »flow value« on arc $a$. For a node $i \in V$ we define by

$$e_f(i) := \sum_{(j, i) \in A} f(j, i) - \sum_{(i, j) \in A} f(i, j)$$

(2.1)

the excess of $i$ with respect to $f$. The first term in (2.1) corresponds to the inflow into $i$, the second term is the outflow out of $i$.

**Definition 2.1 (Flow in a network)**

Let $s, t \in V$ be nodes in the capacitated network $G$. A feasible $(s, t)$-flow is a function $f : A \to \mathbb{R}$ that has the following properties:

(i) $f$ satisfies the **capacity constraints**, that is, for all $a \in A$ we have: $0 \leq f(a) \leq u(a)$.

(ii) $f$ satisfies the **mass balance constraints**, that is, for all $i \in V \setminus \{s, t\}$ we have:

$$e_i(f) = 0.$$

The nodes $s$ and $t$ are called the **source** and the **sink** of the flow, respectively.

If $f$ is a flow, then we have

$$e_f(s) + e_f(t) = \sum_{v \in V} e_f(v)$$

(since $e_f(v) = 0$ for all $v \in V \setminus \{s, t\}$)

$$= \sum_{v \in V} \left( \sum_{(j, i) \in A} f(j, i) - \sum_{(i, j) \in A} f(i, j) \right)$$

$$= 0.$$

To obtain the last equality we used that in the above sum each term $f(x, y)$ appears once positive in the first sum for $i = x$ and once negative for $j = y$. This shows that the net flow
into the sink \( t \) is equal to the net flow out of the source. We define \( \text{val}(f) := e(t) = -e(s) \) to be the value of the flow \( f \). A flow is a maximum \((s,t)\)-Flow, if it has maximum flow value among all feasible \((s,t)\)-flows.

In view of maximum flows (and also minimum cost flows in the following chapter) we can restrict ourselves to the case that at most one of the values \( f(i,j) \) and \( f(j,i) \) is not equal to zero. If \( f(i,j) > 0 \) and \( f(j,i) > 0 \) we can reduce both flow values by \( \epsilon = \min\{f(i,j), f(j,i)\} \) and obtain a flow of the same value (and equal or smaller cost if costs are nonnegative as in the next chapter).

In the following we will be concerned the maximum flow problem, that is, given a capacitated network we wish to find a flow of maximum value. This problem is an example of an (offline) optimization problem.

**Definition 2.2 (Optimization Problem)**

An optimization problem is specified by a set \( I \) of inputs (or instances), by a set \( \text{SOL}(I) \) for every instance \( I \in I \) and by an objective function \( f \) that specifies for every feasible solution \( x \in \text{SOL}(I) \) an objective value or cost. In case of a maximization problem one searches for a feasible solution with maximum cost, in case of a minimization problem one wants to determine a feasible solution of minimum cost.

We usually denote by \( \text{OPT}(I) \) the optimal objective value for instance \( I \).

### 2.3 Residual Networks and Augmenting Paths

An important concept in the context of flows is the notion of a residual network. This network \( G_f \), defined for a particular flow \( f \) indicates how much flow can still be sent along the arcs.

**Definition 2.3 (Residual network)**

Let \( f \) be a flow in the network \( G \) with capacities \( u \colon A \to \mathbb{R}_{\geq 0} \). The residual network \( G_f \) has the same node set as \( G \) and contains for each arc \((i,j) \in A\) up to two arcs:

- If \( f(i,j) < u(i,j) \), then \( G_f \) contains an arc \((i,j)^+\) of residual capacity \( r((i,j)^+) := u(i,j) - f(i,j) \).
- If \( f(i,j) > 0 \), then \( G_f \) contains an arc \((j,i)^-\) of residual capacity \( r((j,i)^-) := f(i,j) \).

Figure 2.3 shows an example of a residual network. Note that \( G_f \) could contain parallel arcs although \( G \) does not. If \( f(i,j) < u(i,j) \) and \( f(j,i) > 0 \), then \( G_f \) contains the arc \((i,j)^+\) and the arc \((i,j)^-\). To avoid notational difficulties we use the signs for the arcs in \( G_f \). Without the signs it could be unclear which of the arcs \((i,j)\) we refer to. In the sequel we use \( \delta \) as a generic sign, that is, each arc in \( G_f \) is of the form \((i,j)^\delta\).

Let \( p \) be a directed path from \( s \) to \( t \) in the residual network \( G_f \). We let \( \Delta(p) := \min_{(i,j)^\delta \in p} r((i,j)^\delta) > 0 \) the minimum residual capacity on the path (cf. 2.4).

We can now augment the flow \( f \) along the path \( p \): If \( (i,j)^+ \) is on \( p \), then we have \( f(i,j) < u(i,j) \) and we set \( f'(i,j) := f(i,j) + \Delta(p) \). If \( (j,i)^- \) is on \( p \), then \( f(j,i) > 0 \) and we set \( f'(j,i) := f(j,i) - \Delta(p) \). For all other arcs, \( f'\) coincides with \( f \). Figure 2.4 illustrates the procedure.

**Definition 2.4 (Augmenting Path)**

A directed \((s,t)\)-path in the residual network \( G \) is called augmenting path for the flow \( f \). The residual capacity of the path is defined to be the minimum residual capacity on its arcs.
Our considerations from above show the following lemma:

**Lemma 2.5** If there is an augmenting path for \( f \), then \( f \) is not a maximum flow. \( \square \)

### 2.4 Maximum Flows and Minimum Cuts

**Definition 2.6 (Cut in a directed graph, forward and backward part)**

Let \( G = (V, A) \) be a directed graph and \( S \cup T = V \) a partition of the node set \( V \). We call

\[
[S, T] := \{ (i, j) \in A : i \in S \text{ and } j \in T \} \cup \{ (j, i) \in A : j \in T \text{ and } i \in S \}
\]

the cut induced by \( S \) and \( T \). We also denote by

\[
(S, T) := \{ (i, j) \in A : i \in S \text{ and } j \in T \}
\]

\[
(T, S) := \{ (j, i) \in A : j \in T \text{ and } i \in S \}
\]

the forward part and the backward part of the cut. We have \([S, T] = (S, T) \cup (T, S)\).

The cut \([S, T]\) is an \((s, t)\)-cut if \( s \in S \) and \( t \in T \).

Figure 2.5 shows an example of a cut and its forward and backward part.

**Definition 2.7 (Capacity of a cut)**

If \( u : A \to \mathbb{R}_{\geq 0} \) is a capacity function defined on the arcs of the network \( G = (V, A) \) and \([S, T]\) is a cut, then the capacity of the cut is defined to be the sum of the capacities of its forward part:

\[
u[S, T] := \sum_{(u, v) \in (S, T)} u(u, v).
\]

Let \( f \) be an \((s, t)\)-flow and \([S, T]\) be an \((s, t)\)-cut in \( G \). Then we have:

\[
\text{val}(f) = -e(s) = -\sum_{i \in S} e(i)
\]

\[
= \sum_{i \in S} \left( \sum_{(i,j) \in A} f(i,j) - \sum_{(j,i) \in A} f(j,i) \right).
\]  

---

**Figure 2.3**: Illustration of residual networks.
2.4 Maximum Flows and Minimum Cuts

(a) The original network and a flow \( f \) with value \( \text{val}(f) = 5 \).

(b) An augmenting path \( p \) (dashed) in the residual network \( G_f \) with \( \Delta(p) = 1 \).

(c) Augmentation of \( f \) along \( p \) yields a new flow \( f' \) with \( \text{val}(f') = \text{val}(f) + \Delta(p) = \text{val}(f) + 1 \).

(d) The resulting residual network \( G_{f'} \) does not contain any path from \( s \) to \( t \). The nodes \( S \) reachable from \( s \) are shown in black.

(e) The nodes in \( S \) induce a cut \([S, T]\) with \( u(S, T) = \text{val}(f) \). The arcs in the forward part \( (S, T) \) of the cut are shown as dashed arcs.

Figure 2.4: A directed \((s, t)\)-path in a residual network can be used to increase the value of the flow.
(a) An \((s, t)\)-cut \([S, T]\) in a directed graph. The arcs in \([S, T]\) are shown as dashed arcs.

(b) The forward part \((S, T)\) of the cut: Arcs in \((S, T)\) are shown as dashed arcs.

(c) The backward part \((T, S)\) of the cut: arcs in \((T, S)\) are shown as dashed arcs.

Figure 2.5: A cut \([S, T]\) in a directed graph and its forward part \((S, T)\) and backward part \((T, S)\).
If for an arc \((x, y)\) both nodes \(x\) and \(y\) are contained in \(S\), then the term \(f(x, y)\) appears twice in the sum \(2.2\), once with a positive and once with a negative sign. Hence, \(2.2\) reduces to

\[
\text{val}(f) = \sum_{(i,j) \in (S,T)} f(i, j) - \sum_{(j,i) \in (T,S)} f(j, i). \tag{2.3}
\]

Using that \(f\) is feasible, that is, \(0 \leq f(i, j) \leq u(i, j)\) for all arcs \((i, j)\), we get from \(2.3\):

\[
\text{val}(f) = \sum_{(i,j) \in (S,T)} f(i, j) - \sum_{(j,i) \in (T,S)} f(j, i) \leq \sum_{(i,j) \in (S,T)} u(i, j) = u[S,T].
\]

Thus, the value \(\text{val}(f)\) of the flow is bounded from above by the capacity \(u[S,T]\) of the cut. We have proved the following lemma:

**Lemma 2.8** Let \(f\) be an \((s, t)\)-flow and \([S,T]\) an \((s, t)\)-cut. Then:

\[
\text{val}(f) \leq u[S,T].
\]

Since \(f\) and \([S,T]\) are arbitrary we deduce that:

\[
\max_{f \text{ is an } (s,t)\text{-flow in } G} \text{val}(f) \leq \min_{[S,T] \text{ is an } (s,t)\text{-cut in } G} u[S,T]. \tag{2.4}
\]

We now show that in fact we have equality in \(2.4\), that is, the maximum flow value equals the minimum cut capacity. To this end, let \(f^*\) be a maximum \((s,t)\)-flow (the existence of such a flow follows by continuity and compactness arguments). According to Lemma \(2.5\) there does not exist an augmenting path with respect to \(f^*\). Consequently, \(t\) is not reachable from \(s\) in \(G_{f^*}\) and the sets

\[
S := \{ i \in V : i \text{ is reachable from } s \text{ in } G_{f^*} \},
\]

\[
T := \{ i \in V : i \text{ is not reachable from } s \text{ in } G_{f^*} \}
\]

are both nonempty (we have \(s \in S\) and \(t \in T\)). In other words, \([S,T]\) is an \((s,t)\)-cut. Let \((i,j)\) be any arc in the forward part of the cut. Then, we must have that \(f(i,j) = u(i,j)\), since otherwise \((i,j)^+\) would be contained in \(G_{f^*}\) and \(j \in T\) would be reachable from \(s\) (by definition \(i\) is reachable from \(s\)). Hence:

\[
\sum_{(i,j) \in (S,T)} f(i,j) = \sum_{(i,j) \in (S,T)} u(v,w) = u[S,T]. \tag{2.5}
\]

Similarly, for any arc \((j,i) \in (T,S)\) we must have that \(f(j,i) = 0\), since otherwise \((i,j)^-\) \(\in G_{f^*}\) and \(j \in T\) would again be reachable from \(s\). Thus, we have:

\[
\sum_{(j,i) \in (T,S)} f(u,v) = 0. \tag{2.6}
\]

Combining \(2.5\) and \(2.6\) yields:

\[
\text{val}(f) = \sum_{(i,j) \in (S,T)} f(i,j) - \sum_{(j,i) \in (T,S)} f(j, i) = u[S,T] \tag{by \(2.3\)}.
\]

Lemma \(2.8\) now implies that \(f^*\) must be maximum flow and \([S,T]\) a minimum cut. Hence, the maximum value of a flow equals the minimum capacity of a cut. This result is known as the famous *Max-Flow-Min-Cut-Theorem*: 

\[
\text{Max-Flow-Min-Cut-Theorem:}
\]
**Theorem 2.9 (Max-Flow-Min-Cut-Theorem)** Let $G = (V, A)$ be a network with capacities $u: A \rightarrow \mathbb{R}_{\geq 0}$, then:
\[
\max_{f \text{ is an } (s,t)\text{-flow in } G} \text{val}(f) = \min_{[S, T] \text{ is an } (s,t)\text{-cut in } G} u[S, T].
\]

**Proof:** See above. \(\square\)

The Max-Flow-Min-Cut-Theorem has a large number of interesting combinatorial consequences and applications. We refer to [1, 7, 15, 23] for details.

Our proof of the Max-Flow-Min-Cut-Theorems above has another nice byproduct. Our arguments (applied to an arbitrary flow instead of the already known maximum flow $f$) show that, if there is no augmenting path for a flow $f$ then $f$ must be maximum. We had already shown the converse in Lemma 2.5. This result will prove to be useful for establishing correctness of our maximum flow algorithms. We thus note it for later reference:

**Theorem 2.10** A flow $f$ is a maximum $(s, t)$-flow if and only if there is no augmenting path, that is, there is no directed path in $G_f$ from $s$ to $t$. \(\square\)

### 2.5 Network Flows and LP-Duality

We can write the maximum flow problem as a linear program as follows:

\[
\begin{align*}
\max & \sum_{(i,t) \in A} f(i,t) - \sum_{(t,j) \in A} f(t,j) \\
0 & \leq f(i,j) \leq u(i,j) \quad \text{for all } (i,j) \in A
\end{align*}
\]

The dual of the Linear Program (2.7) is given by

\[
\begin{align*}
\min & \sum_{(i,j) \in A} u(i,j)z(i,j) \\
- y(i) + y(j) + z(i,j) & \geq 0 \quad \text{for all } (i,j) \in A \text{ with } i,j \notin \{s,t\} \\
y(j) + z(s,j) & \geq 0 \quad \text{for all } (s,j) \in A \\
- y(i) + z(i,s) & \geq 0 \quad \text{for all } (i,s) \in A \\
- y(i) + z(i,t) & \geq 1 \quad \text{for all } (i,t) \in A \\
y(j) + z(r,j) & \geq -1 \quad \text{for all } (t,j) \in A \\
z(i,j) & \geq 0 \quad \text{for all } (i,j) \in A
\end{align*}
\]

To simplify the situation we introduce two additional variables $y(s) = 0$ and $y(t) = -1$. Then, all constraints above ar of the form:

\[-y(i) + y(j) + z(i,j) \geq 0.\]

Observe that we can add 1 to all $y(i)$ without changing anything. This yields the following LP-formulation of the dual to the maximum flow problem:

\[
\begin{align*}
\min & \sum_{(i,j) \in A} u(i,j)z(i,j) \\
y(s) = 1, y(t) = 0 \\
- y(i) + y(j) + z(i,j) & \geq 0 \quad \text{for all } (i,j) \in A \\
z(i,j) & \geq 0 \quad \text{for all } (i,j) \in A
\end{align*}
\]
Theorem 2.11 If the Linear Program (2.9) has an optimal solution, then it has one of the following forms: $S \cup T = V$ is a partition of the vertex set with $s \in S$ and $t \in T$ and:

$$y(i) = \begin{cases} 
1 & \text{if } i \in S \\
0 & \text{if } i \in T 
\end{cases}$$

$$z(i, j) = \begin{cases} 
1 & \text{if } (i, j) \in (S, T) \\
0 & \text{otherwise} 
\end{cases}$$

Proof: Let $(y, z)$ be an optimal solution of (2.9). Without loss of generality assume that $y(1) \geq y(2) \geq \cdots \geq y(n)$. Suppose that $s = p$ and $t = q$. Clearly, $p < q$.

Consider the family of cuts $(S_i, T_i)$, $i = p, \ldots, q - 1$ where $S_i = \{y(1), \ldots, y(i)\}$, $T_i = V \setminus S_i$. The values $\lambda_i := y(i) - y(i + 1)$ are all nonnegative and their sum satisfies

$$\sum_{i=p}^{q-1} \lambda_i = y(p) - y(q) = y(s) - y(t) = 1 - 0 = 1.$$ 

Thus, we can view the $\lambda_i$ as a probability distribution over the family of cuts. We compute the expected value of a cut drawn according to this probability distribution. The value is given by

$$\frac{1}{q-p} \sum_{i=p}^{q-1} \lambda_i u(\delta(S_i)), \quad (2.10)$$

where $\delta(S_i)$ denotes the arcs emanating from $S_i$ and $u(X) = \sum_{x \in X} u(x)$ for any subset $X$ of the arcs. Any arc $(i, j)$ with $i > j$ does not contribute to the sum (2.10). On the other hand, any arc $(i, j)$ with $i < j$ contributes at most $y(i) - y(j)u(i, j)$ to (2.10). Hence, the expected value of a cut is at most

$$\sum_{(i, j) \in A \atop i < j} (y(i) - y(j))u(i, j) \leq \sum_{(i, j) \in A} (y(i) - y(j))u(i, j) \leq \sum_{(i, j) \in A} z(i, j)u(i, j). \quad (2.11)$$

The value on the right hand side of (2.11) is exactly the optimal objective value of $(y, z)$. Since the value of a random cut is at most this value, there must exist at least one of the cuts in our family having capacity $\sum_{(i, j) \in A} z(i, j)u(i, j)$.

2.6 Flow Feasibility Problems

In some applications it makes sense to have additional lower bounds $l(i, j)$ on the flow values on the arcs. In these problems one asks for a flow $f$ that satisfies $l(i, j) \leq f(i, j) \leq u(i, j)$ for any arc $(i, j)$ and which achieves balance at every node in the network, that is, $e_f(i) = 0$ for all $i \in V$. Such a flow feasibility problem appears for instance in the design of algorithms for the minimum cost flow problem in Chapter 3.

We can solve the flow feasibility problem by means of a maximum flow computation in a slightly enlarged network. Let $G = (V, A)$ be the network for the flow feasibility problem. We add two new nodes $s$ and $t$ to $G$ which are connected by arcs $(s, j)$ and $(i, t)$ for all $i, j \in V$ to the vertices in the old network (see Figure 2.6 for an illustration). The capacities $u'$ in the new network $G'$ are as follows:

- The capacity $u'(s, j)$ of arc $(s, j)$ is $\sum_{l:(i,j) \in A} l(i, j)$.
- The capacity $u'(i, t)$ of arc $(i, t)$ is $\sum_{j:(i,j) \in A} l(i, j)$.
• The capacity of an arc $(i, j) \in A$ is $u'(i, j) = u(i, j) - l(i, j)$.

We show that there is a maximum flow from $s$ to $t$ in $G'$ which saturates all arcs $(s, j)$ and $(i, j)$ if and only if there is a feasible flow in $G$.

Let $f$ be a feasible flow in $G$. If we define $f'(i, j) := f(i, j)$ for all arcs $(i, j) \in A$ a node $i \in V$ will have imbalance $\sum_{j:(i,j)\in A} l(i, j) - \sum_{j:(j,i)\in A} l(i, j)$. Thus, if we set $f'(s, j) := \sum_{i:(i,j)\in A} l(i, j)$ and $f'(i, t) := \sum_{j:(i,j)\in A} l(i, j)$ all nodes will be balanced and the corresponding flow $f'$ in $G'$ saturates all new arcs.

If on the other hand, $f'$ is a flow in $G'$ such that all of the new arcs are saturated, we can define a flow $f$ in $G$ by $f(i, j) := f'(i, j) + l(i, j)$. It is straightforward to see that $f$ is feasible in $G$.

Figure 2.6: Solving flow feasibility problems by maximum flow computation.

We will now consider a slightly different flow feasibility problem. Let $b: V \rightarrow \mathbb{R}$. We wish to find a flow $f$ such that

\[ e_f(i) = b(i) \quad \text{for all } i \in V \] (2.12a)

\[ 0 \leq f(i, j) \leq u(i, j) \quad \text{for all } (i, j) \in A \] (2.12b)

The proof of the following theorem can be deduced from the Max-Flow-Min-Cut-Theorem and is left as an easy exercise:

**Theorem 2.12** There exists a flow $f$ satisfying the conditions in (2.12) if and only if $\sum_{i \in V} b(i) = 0$ and for every $S \subseteq V$ we have

\[ \sum_{i \in S} b(i) \leq u(S, V \setminus S). \]

Moreover, if $b$ and $u$ are integral, then (2.12) has a solution if and only if it has an integral solution.

**Corollary 2.13** There exists a flow $f: A \rightarrow \mathbb{R}$ with $e_f(i) = b(i)$ for every $i \in V$ and $f(a) \geq 0$ for every arc $a \in A$ if and only if $\sum_{i \in V} b(i) = 0$ and for every $S \subseteq V$ with $(V \setminus S, S) = \emptyset$ we have $\sum_{i \in S} b(i) = 0$. If $b$ is integral, then there exists a solution if and only if there exists an integral solution.

### 2.7 A Review of Basic Algorithms

Theorem 2.10 on page 14 immediately suggest a simple algorithm for computing a maximum flow (see Algorithm 2.1). We start with the flow that is zero everywhere $f \equiv 0$. If $G_f$ contains an augmenting path $p$ with respect to $f$, we augment $f$ along $p$. We then update $G_f$ and continue until $G_f$ does not contain any augmenting path. This algorithm dates back to Ford and Fulkerson:

If all capacities are integer, then each time $f$ is augmented by an integral amount (at least 1) and all flows occurring during the run of the algorithm are integer.

Let $U := \max \{u(a) : a \in A\}$ be the maximum capacity of an arc. Then, the $(s, t)$-cut $((s), V \setminus \{s\})$ contains at most $n - 1$ arcs and thus, has capacity at most $(n - 1)U$. Hence, Algorithm 2.1 must terminate after at most $(n - 1)U$ iterations, since it can not find any augmenting path. By Theorem 2.10 the (integral) flow obtained upon termination must be maximum.
Algorithm 2.1 Generic algorithm based on augmenting paths.

**INPUT**: A directed graph $G = (V, A)$ in adjacency list representation; a nonnegative capacity function $u: A \rightarrow \mathbb{R}_{\geq 0}$, two nodes $s, t \in V$.

1. for all $(u, v) \in A$ do
   2. $f(u, v) \leftarrow 0$ \{ Start with the zero flow $f \equiv 0$. \}
3. end for
4. while there is an augmenting path in $G_f$ do
   5. Choose such a path $p$.
   6. $\Delta \leftarrow \min \{ r((i, j)^\delta) : (i, j)^\delta \in p \}$ \{ residual capacity of $p$. \}
   7. Augment $f$ along $p$ by $\Delta$ units of flow.
   8. Update $G_f$.
9. end while

Theorem 2.14 If all capacities are integral, then the generic algorithm based on augmenting paths, Algorithm 2.1, terminates after $O(nU)$ augmentation steps with an optimal integral flow.

Corollary 2.15 If all capacities are integral, then there always exists a maximum flow which is also integral.

The result of Corollary 2.15 is interesting and the basis of many combinatorial results. Be aware that the Corollary does not show that any maximum flow is integral, but rather that there is at least one maximum flow which happens also to be integral. Figure 2.7 shows an example of a flow which is maximum but not integral.

![Figure 2.7](image)

Figure 2.7: Even if capacities are integral, then there might be a maximum flow which is not integral. The dashed arc is the forward part of a minimum cut. By Corollary 2.15, there is at least one maximum flow which is also integral.

Algorithm 2.1 does not specify which augmenting path to use if there is more than one. The obvious choice would be to use a shortest one. Such a path can be determined in $O(n + m)$ time by Breadth-First-Search. The resulting algorithm is called the Edmonds-Karp Algorithm. We will shortly prove a strongly polynomial running time bound for this algorithm.

It should be noted that Algorithm 2.1 could be horribly slow if we choose bad augmenting paths. There are examples where in fact $\Omega(nU)$ augmentations are used. If $U = 2^n$ this leads to exponential time! Another theoretical disadvantage of the generic algorithm...
is that it might not terminate if capacities are non-integral. Although the residual capacities converge to zero, the flow obtained in the limit is not maximum. We refer to [1] for examples.

**Lemma 2.16** Let \( \delta(i, j) \) denote the distance between nodes \( i \) and \( j \) in the residual network. Then, during the Edmonds-Karp-Algorithm for any node \( i \) the value \( \delta(s, i) \) is monotonously nondecreasing.

**Proof:** We show the claim by induction on the number of augmentation steps. If no augmentation happens, then the claim is trivial. We now assume that the claim holds until the \( k \)th augmentation where flow \( f \) is augmented to flow \( f' \). Let \( \delta(s, i) \) denote the distances before the augmentation and \( \delta'(s, i) \) the distances after the augmentation, that is, in \( G_{f'} \). We must show that \( \delta'(s, i) \geq \delta(s, i) \) for all nodes \( i \).

![Figure 2.8: Proof of Lemma 2.16](image)

If \( i \) is predecessor of \( j \) on a shortest path from \( s \) to \( j \) in \( G_{f'} \), then \( \delta'(s, j) = \delta'(s, i) + 1 \).

Suppose that \( \delta'(s, j) < \delta(s, j) \) for some \( j \in V \). Then \( j \neq s \), since the distance of \( s \) to itself is always zero. We can assume that \( j \) is chosen in such a way that \( \delta'(s, j) \) is minimum among all nodes which violate the claim of the lemma. Let \( i \) be the predecessor of \( j \) on a shortest path from \( s \) to \( j \) in \( G_{f'} \) (see Figure 2.8). Then

\[
\delta'(s, j) = \delta'(s, i) + 1,
\]

since the path from \( s \) to \( i \) must be a shortest \( (s, i) \)-path. By the choice of \( j \)

\[
\delta'(s, i) \geq \delta(s, i).
\]

We know that the arc \((i, j)^\delta\) is contained in \( G_{f'} \). If \((i, j)^\delta\) was also in \( G_f \), then

\[
\begin{align*}
\delta(s, j) &\leq \delta(s, i) + 1 \\
&\leq \delta'(s, i) + 1 \quad \text{(by (2.14))} \\
&= \delta'(s, j) \quad \text{(by (2.13)).}
\end{align*}
\]

This contradicts the assumption that \( \delta'(s, j) < \delta(s, j) \).

Hence \((i, j)^\delta\) is not contained in \( G_f \). Since the arc is present in \( G_{f'} \), the \((k + 1)\)st augmentation must have used the inverse arc \((j, i)^{-\delta}\). Since the algorithm always augments along shortest paths, it follows that \( \delta(s, i) = \delta(s, j) + 1 \). Thus,

\[
\begin{align*}
\delta(s, j) &= \delta(s, i) - 1 \\
&\leq \delta'(s, i) - 1 \quad \text{(by (2.14))} \\
&= \delta'(s, j) - 2 \quad \text{(by (2.13))} \\
&< \delta'(s, j).
\end{align*}
\]

Again, we obtain a contradiction. \( \Box \)

We can now establish the running time of the Edmonds-Karp-Algorithm.
Theorem 2.17 Let $G$ be a network with integral, rational or real capacities. The Edmonds-Karp-Algorithm terminates after $O(nm)$ iterations with a maximum flow. The total running time is $O(nm^2)$.

Proof: We call an arc $(i, j)^\delta$ on an augmenting path $p$ a bottleneck arc if its residual capacity coincides with that of $p$. In any augmentation step there is at least one bottleneck arc, and all bottleneck arcs disappear from the residual network by the flow augmentation. To establish the claim, it suffices to show that any arc can be a bottleneck arc at most $O(n)$ times.

Let $(i, j)^\delta$ be a bottleneck arc of the current iteration and $f$ be the current flow. We have $\delta(s, j) = \delta(s, i) + 1$ since the algorithm augments along shortest paths. The arc $(i, j)^\delta$ can only re-appear in a residual network (a necessary condition for becoming a bottleneck arc again!) if some later augmentation uses arc $(j, i)^{-\delta}$. At this moment we have $\delta'(s, i) = \delta'(s, j) + 1$. By Lemma 2.16 it follows that $\delta'(s, i) \geq \delta(s, j) + 1 = \delta(s, i) + 2$. Hence, the distance of $i$ has increased by at least $2$ in between. Since the distance of any node from $i$ is bounded from above by $n - 1$ (if it is still finite), $(i, j)^\delta$ can be a bottleneck arc at most $(n - 1)/2 = O(n)$ times.

2.8 Preflow-Push Algorithms

In this section we investigate so called preflow-push algorithms for computing maximum flows. These algorithms are highly efficient both theoretically and practically.

A disadvantage of most algorithms based on augmenting paths is that they determine an augmenting path and in the next iteration search again for such a path throwing away all potentially useful information that is available from the previous search. Figure 2.9 illustrates such a situation. Preflow-push algorithms work more efficiently by augmenting flow not along a whole path but along single arcs.

The basic operation of all preflow-push algorithms is a flow push. To push $\varepsilon$ units of flow over an arc $(i, j)^\delta$ means to increase the flow on $(i, j)$ by $\varepsilon$ if $\delta = +$ and to decrease the flow by $\varepsilon$ units if $\delta = -$.

We call a function $d: V \rightarrow \mathbb{N}_0$ a distance labeling with respect to $G_f$, if it satisfies the following conditions:

\begin{align*}
d(t) &= 0 \\
d(i) &\leq d(j) + 1 \quad \text{for all arcs } (i, j)^\delta \in G_f.
\end{align*}

We will refer to conditions \ref{eq:distance_labeling1} and \ref{eq:distance_labeling2} as validity conditions and call $d(i)$ the distance label of node $i$.

Let $p = (v = v_0, v_1, \ldots, v_k = t)$ be a path from $v$ to $t$ in $G_f$. Then the validity conditions imply that

$$d(v) \leq d(v_1) + 1 \leq d(v_2) + 2 \leq \cdots \leq d(t) + k = k.$$ 

Hence, the distance label $d(v)$ is at most the length of the path (counted in number of arcs). Since $p$ was arbitrary, we can deduce that $d(v)$ is a lower bound for the distance of $v$ to $t$ in the residual network $G_f$. This property in conjunction with Theorem \ref{thrm:edmonds_karp} gives a useful condition to test maximality of flows by means of distance labelings:

Lemma 2.18 Let $d$ be a distance labeling with respect to $G_f$ and $d(s) \geq n$. Then $f$ is a maximum flow.

\footnote{In the case of real capacities we assume that we can carry out arithmetic operation on reals in constant time.}
Figure 2.9: Algorithms based on augmenting paths use the long prefix of the path in each iteration.

**Proof:** If there is any (s, t)-path in $G_f$, then there is one that does not contain cycles and thus has length at most $n - 1$. Since $d(s) \geq n$ we can deduce that there is no such (s, t)-path and by Theorem 2.10 the flow must be maximum.

In order to describe the new class of algorithms for computing maximum flows we need the concept of a *preflow*:

**Definition 2.19 (Preflow, active node)**

Let $s, t \in V$ be nodes in the capacitated network $G$. A **feasible preflow** is a function $f : A \to \mathbb{R}_{\geq 0}$ obeying the capacity constraints (see Definition 2.1) and which also satisfies $e_f(v) \geq 0$ for all $v \in V \setminus \{s, t\}$.

Any node $v \in V \setminus \{s, t\}$ with $e_f(v) > 0$ is called an **active node** (with respect to the preflow $f$).

The basic idea for all preflow-push-algorithms is as follows: we start with a preflow $f$ which ensures that $t$ is not reachable from $s$ in $G_f$. The property that $t$ is reachable from $s$ will be maintained throughout the algorithm. We now try to »push« flow from active nodes to other nodes »closer to the sink« (how this is done exactly will be described shortly). If at some point in time we reach the situation that $e_f(v) = 0$ for all nodes $v \in V \setminus \{s, t\}$ we have found a maximum flow.

In order to specify what we mean with »closer to the sink« we use distance labels as defined above.

**Definition 2.20 (Feasible arc)**

Let $G_f$ be the residual network for a preflow $f$ and $d$ a valid distance labeling with respect to $G_f$. An arc $(i, j) \in G_f$ is **feasible** if $d(i) = d(j) + 1$.

All of our algorithms will only push flow over feasible arcs. There is a neat visualization of the process. We can imagine the flow as water and the distance labels as heights. Then, flow is always sent downwards. Initially, the source will be lifted such that enough water fills the pipe system. At some point in time we might arrive at the situation that an active node $i$ is not able to get rid of its excess since all its successors in the network are at greater heights. In this case we lift $i$ just enough to let the excess flow out.
Algorithm 2.2 Generic preflow-push algorithm for computing a maximum flow.

**GENERIC-PREFLOW-PUSH**(G, u, s, t)

**Input:** A directed graph G = (V, A) in adjacency list representation; a nonnegative capacity function u: A → ℝ₀, two nodes s, t ∈ V.

**Output:** A maximum (s, t)-flow f.

1. for all (u, v) ∈ A do
2. \( f(u, v) \leftarrow 0 \) \{ Start with the zero preflow \( f \equiv 0 \). \}
3. end for
4. Compute the exact distances \( \delta(v, t) \) in \( G_f \) and set \( d[v] \leftarrow \delta(v, t) \) for all \( v \in V \).
   \{ This computation can be carried out by a reverse breadth first search in time \( O(n + m) \). \}
5. for all (s, v) ∈ A do
6. \( f(s, v) \leftarrow u(s, v) \)
7. end for
8. \( d[s] \leftarrow n \)
9. while there is an active node do
10. \{ active node \( i : e_f(i) > 0 \). \}
11. Choose an active node \( i \).
12. PUSH-RELABEL(i)
13. end while

**PUSH-RELABEL**(i)

1. if there is a feasible arc \((i, j)^{\delta}\) in \( G_f \) then \{ feasible arc: \( d[i] = d[j] + 1 \) \}
2. Push \( e \leftarrow \min\{e_f(i), r((i, j)^{\delta})\} \) units of flow from \( i \) to \( j \).
3. else
4. \( d[i] \leftarrow 1 + \min\{ d[j] : (i, j)^{\delta} \in G_f \} \) \{ Increase the label of \( i \) (relabel). \}
5. end if

Algorithms 2.2 shows pseudocode for the generic preflow-push algorithm. It starts with the zero flow which is modified by saturating all arcs that emanate from the source (Steps 4 to 7). From that moment on the sink node is no longer reachable from the source in the residual network. The source is lifted in Step 8 and exact distance labels are computed.

The main part of the algorithm consists of repeatedly calling the subroutine PUSH-RELABEL(i) for an active node \( i \). PUSH-RELABEL either pushes flow from \( i \) over a feasible arc to a neighboring node in \( G_f \) (Step 2) or increases the label of \( i \) (Step 3). We refer to the latter action as a relabel operation. The algorithm terminates if no active nodes are present. Figures 2.10 until 2.12 show an example of the execution of the algorithm.

We first address the correctness of the algorithm.

**Lemma 2.21** The node labels \( d[v] \) (\( v \in V \)) maintained by the algorithm are valid distance labels.

**Proof:** We show the claim by induction on the number of calls to PUSH-RELABEL. Clearly, before the first call \( d \) forms a valid distance labeling.

A relabel operation clearly preserves all conditions 2.15 and 2.16. A push over \((i, j)^{\delta}\) might add the inverse arc \((j, i)^{−\delta}\) to the network. We must verify that in this case the corresponding condition \( d[j] \leq d[i] + 1 \) holds. However, since the algorithm only pushes flow along feasible arcs, a push over \((i, j)^{\delta}\) implies that \( d[i] = d[j] + 1 \). Thus, again the validity is maintained. \(\square\)

**Lemma 2.22** If the generic preflow-push algorithm terminates, it terminates with a maximum flow.
22 Fast Algorithms for Computing Maximum Flows

(a) The initial network.

(b) The residual network for the preflow $f \equiv 0$ with exact distance labels. The heights of the nodes visualize the distance labels. The residual capacities are shown on the arcs.

(c) Initially, all arcs emanating from $s$ are saturated. In addition, the distance label of $s$ is set to $d[s] = n$. Infeasible arcs are shown as dashed arcs. The current iteration uses node 3. Arc $(3, 4)$ is the only feasible arc emanating from 3.

(d) Again, the active node 3 is chosen. This time, there are no feasible arcs emanating from 3. Thus, the label of 3 is increased to $1 + \min\{1, 4\} = 2$ erhöht. This way, arc $(3, 2)$ becomes feasible.

Figure 2.10: Computation of a maximum flow by the preflow-push algorithm.
2.8 Preflow-Push Algorithms

Figure 2.11: Computation of a maximum flow by the preflow-push algorithm (continued).
Figure 2.12: Computation of a maximum flow by the preflow-push algorithm (continued).

**Proof:** Upon termination, there is no active node. Hence, the algorithm must terminate with a flow. By Lemma 2.21 we get that $d[s] \geq n$ is a lower bound for the distance from $s$ to $t$ in the residual network. Lemma 2.18 now shows that the flow upon termination must be maximum.

In view of Lemma 2.22 we »only« need to show that our algorithm terminates after a finite number of steps and to bound its complexity.

### 2.8.1 Bounding the Number of Relabel Operations

**Lemma 2.23** Let $f$ be a preflow during the execution of the generic preflow-push algorithm. For any active node $i$ there is a path from $i$ to $s$ in $G_f$.

**Proof:** Let $S \subseteq V$ the set of nodes in $G_f$ from which $s$ can be reached and let $T := V \setminus S$. We must show that $T$ does not contain any active node.

There is no arc $(i, j)^S$ in $G_f$ with $i \in T$ and $j \in S$, since otherwise $s$ would be reachable from $i$ contradicting the fact that $i \in T$. Hence

$$
\sum_{(i,j) \in (S;T)} f(i, j) - \sum_{(j,i) \in (T;S)} f(j, i) = -\sum_{(j,i) \in (T;S)} u(j, i). \quad (2.17)
$$
2.8 Preflow-Push Algorithms

Damit ergibt sich

\[ 0 \leq \sum_{i \in T} e_f(i) \quad \text{(since } f \text{ is a preflow and } s \in S) \]

\[ = \sum_{i \in T} \left( \sum_{(j,i) \in A} f(j,i) - \sum_{(i,j) \in A} f(i,j) \right) \]

\[ = \sum_{(i,j) \in (S,T)} f(i,j) - \sum_{(j,i) \in (T,S)} f(j,i) \]

\[ = - \sum_{(j,i) \in (T,S)} u(j,i) \quad \text{(by (2.17))} \]

\[ \leq 0 \quad \text{(since } u \geq 0). \]

It follows that \( \sum_{i \in T} e_f(i) = 0 \) and by \( e_f(i) \geq 0 \) we get that \( e_f(i) = 0 \) for all \( v \in T \).

As a corollary of the above lemma we get that in Step 4 never minimizes over the empty set: since for all active nodes there is a path to \( s \), in particular there must be at least one emanating arc.

**Lemma 2.24** All distance labels \( d[i] \) remain bounded from above by \( 2n - 1 \) during the execution of the algorithm. The distance label of any node is increased at most \( 2n - 1 \) times. The total number of relabel operations is \( O(n^2) \).

**Proof:** Only the distance labels of active nodes are increased. Hence it suffices to show that the algorithm never increases a label of an active node to a value larger than \( 2n - 1 \).

Let \( i \) be an active node. By Lemma 2.23, there is a path from \( i \) to \( s \) in \( G_f \). This path can be assumed to be without cycles and hence has length at most \( n - 1 \). The validity of the distance labels (see Lemma 2.21) now implies that \( d[i] \leq (n - 1) + d[s] = (n - 1) + n = 2n - 1 \).

\[ \square \]

2.8.2 Bounding the Number of Pushes

We have already bounded the number of relabel operations. The number of pushes is a bit harder to bound. We divide the pushes into two classes:

**Definition 2.25 (Saturating and non-saturating push)**

A push of flow in Step 2 is called **saturating** if \( \epsilon = r((i,j)^A) \). Otherwise, the push is termed **non-saturating**.

A saturating push causes \((i,j)^A\) to appear from the residual network while the inverse arc \((i,j)^-A\) appears.

**Lemma 2.26** The generic preflow-push algorithm uses \( O(nm) \) saturating pushes.

**Proof:** Let \((i,j)^A\) be a (potential) arc of the residual network. We show that there can be only \( O(n) \) saturating pushes over \((i,j)^A\). This implies that the total number of saturating pushes is at most \( 2m \cdot O(n) = O(nm) \).

Suppose that there is a saturating push over \((i,j)^A\). Then, at this time \( d[i] = d[j] + 1 \). The push causes \((i,j)^A\) to disappear from the residual network. It can only re-appear after some flow has been pushed over the inverse arc \((j,i)^-A\). The moment, flow is pushed over \((j,i)^-A\) we have \( d'[j] = d'[i] + 1 \geq d[i] + 1 = d[j] + 2 \). Here, we have used the fact
that distance labels never decrease. Hence between two saturating pushes over \((i, j)^\delta\) the distance label of \(j\) must have increased by at least 2. Since by Lemma 2.24 this can happen at most \((2n - 1)/2 = \mathcal{O}(n)\) times, we have proved the claim.

**Lemma 2.27** The number of non-saturating pushes in the generic preflow-push algorithm is \(\mathcal{O}(n^2 m)\).

**Proof:** We use a potential function argument. Let \(I \subseteq V \setminus \{s, t\}\) the set of all active nodes. Our potential \(\Phi\) is defined as

\[
\Phi := \sum_{i \in I} d[i].
\]

Clearly, \(\Phi\) is nonnegative. Before the main part of the algorithm, that is, before the first call to PUSH-RELABEL, we have \(\Phi \leq (n - 1)(2n - 1) < 2n^2 = \mathcal{O}(n^2)\), since each of the at most \(n - 1\) successors of \(s\) have a distance label at most \(2n - 1\) (cf. Lemma 2.24). If at some point \(\Phi\) drops to zero, then by the nonnegativity of \(d\) the set of active nodes must be empty and the algorithm terminates.

A non-saturating push over an arc \((i, j)^\delta\) reduces the excess of \(i\) to zero, and possibly makes \(j\) active. The potential drops by at least \(d[j] - d[i] = 1\), since the arc was feasible and thus \(d[i] = d[j] + 1\). All increases in potential are thus due to either saturating pushes or relabel operations.

A saturating push over \((i, j)^\delta\) can increase the potential by at most \(d[j] \leq 2n - 1\). By Lemma 2.26 there are \(\mathcal{O}(nm)\) saturating pushes which let us bound the total potential increase due to saturating pushes by \((2n - 1) \cdot \mathcal{O}(nm) = \mathcal{O}(n^2 m)\).

Relabel operations are carried out only on active nodes. Since for any of the \(n - 2\) potential active nodes, its label can increase to at most \(2n - 1\), the total increase in potential due to relabel operations is \(\mathcal{O}(n^2)\).

We have shown that the initial potential is \(\mathcal{O}(n^2)\) and that the total increase in potential is \(\mathcal{O}(n^2 m) + \mathcal{O}(n^2) = \mathcal{O}(n^2 m)\). Since any non-saturating push leads to a decrease in potential, this bounds the number of non-saturating pushes to \(\mathcal{O}(n^2) + \mathcal{O}(n^2 m) = \mathcal{O}(n^2 m)\) as claimed.

### 2.8.3 Time Complexity of the Generic Algorithm

We now show that the total running time of the generic algorithm is in \(\mathcal{O}(n^2 m)\). We will see later how this can be improved by clever choices of the active node in Step 10.

All operations in the initialization until Step 1 can be carried out in \(\mathcal{O}(n + m)\) time. We have already shown that PUSH-RELABEL is called at most \(\mathcal{O}(n^2 m)\) times by the bounds on the number of relabel and push operations. However, it is not clear that each such call uses constant time (at least on average) which would be necessary to prove the \(\mathcal{O}(n^2 m)\) time bound.

We tackle the problem by a clever representation of the residual network \(G_f\). We store for each node \(i \in V\) a list of all possible arcs in a residual network that emanate from \(i\). The List \(L[i]\) contains all arcs of the set:

\[
\{ (i, j)^+: (i, j) \in A \} \cup \{ (i, j)^-: (j, i) \in A \}.
\]

Together with each arc \((i, j)^\delta\) we store its residual capacity and a pointer to the list entry of the corresponding inverse arc \((j, i)^{-\delta}\) in the list \(L[j]\). We have \(\sum_{i \in V} |L[i]| = 2m\).

We organize the active lists in a doubly-linked list \(L_{\text{active}}\). Recall that elements in a doubly-linked list can be inserted and deleted in constant time, provided a pointer to the element is at hand (see 7). Moreover, we can test in constant time whether \(L_{\text{active}}\) is empty.
For each list \( L[i] \) we keep a pointer \( \text{current}[i] \) on the «current» arc, which initially points to the head of the list. For each node \( i \in I \) we store its distance label \( d[i] \) and its excess \( e[i] \). It should be clear that all those (simple) data structures can be build in time \( O(n + m) \).

The data structures allow us to carry out a push in constant time: Since we can determine the residual capacity and the excess of the node in constant time we can determine the value \( \varepsilon \) in Step 2 in constant time. Moreover, the pointer to \( (j, i)^{-\delta} \) in the list \( L[j] \) lets us update the residual capacities affected by the push in constant time. If a new node becomes active due to the push, we add it to the list \( L_{\text{active}} \). Again, this can be done in constant time.

Suppose that \( \text{PUSH-RELABEL}(i) \) is called. We need to test whether there is a feasible arc emanating from \( i \). To this end we start at the entry pointed to by \( \text{current}[i] \) until we either find a feasible arc (with positive residual capacity) or reach the end of the list.

In the first case we advance the pointer to the respective entry, say \((i, j)^\delta\), and perform the push. We have already argued above that a push only needs constant time. Hence, the total time needed for pushes is \( O(n^2m) \) by the fact that \( O(n^2m) \) pushes are carried out.

In the second case where we reach the end of the list \( L[i] \), we perform a relabel operation on \( i \) and reset \( \text{current}[i] \) to the first element in \( L[i] \) (we will argue shortly that in this case there is no feasible arc emanating from \( i \) and we can justly perform a relabel). The new label of \( i \) can be determined by a full pass through \( L[i] \), which needs \( O(|L[i]|) \) time. Since the label of \( i \) is increased only \( 2n - 1 \) times (see Lemma 2.24), the total effort for all relabel operations is

\[
(2n - 1) \sum_{i \in V} |L[i]| = (2n - 1)2m = O(nm).
\]

It remains to show that if we reach the end of the list \( L[i] \), there is no feasible arc emanating from \( i \). Clearly, there can be no feasible arcs after the arc we started our search from, the one pointed to by \( \text{current}[i] \). Let \((i, j)^\delta\) be an arc before \( \text{current}[i] \). Since we moved the current arc pointer past \((i, j)^\delta\) at this moment \((i, j)^\delta\) was either infeasible or had zero residual capacity. Since the time we moved the pointer past \((i, j)^\delta\) there can not have been a relabel operation on \( i \) since in this case we would have reset the current arc pointer.

- If earlier \((i, j)^\delta\) was infeasible, then \( r(i, j)^\delta > 0 \), but \( d[i] \leq d[j] \). In order to make \((i, j)^\delta\) feasible there must be a relabel operation on \( i \) which we just argued was not performed. Hence, the arc is still infeasible.
- If earlier \((i, j)^\delta = 0 \), then the arc \((i, j)^\delta\) can only have positive residual capacity now, if in the mean time we performed a push on \((j, i)^{-\delta} \). But this is only possible if at that time \( d'[j] = d'[i] + 1 \). Again, we need a relabel on \( i \) to make \((i, j)^\delta\) feasible.

We have thus shown the following theorem:

**Theorem 2.28** The generic preflow-push algorithm can be implemented to run in time \( O(n^2m) \).

### 2.8.4 The FIFO-Preflow-Push Algorithm

The bottleneck of the generic preflow-push algorithm are the \( O(n^2m) \) non-saturating pushes. The time needed for all other operations is \( O(nm) \), which is considerably better. In this section we show how we can decrease the number of non-saturating pushes by a clever choice of the active node.

If the generic preflow-push algorithm calls \( \text{PUSH-RELABEL} \) on an active node \( i \) and performs a saturating push, then \( i \) possibly remains active. The next call to \( \text{PUSH-RELABEL} \) might use another active node. In contrast, the **FIFO-preflow-push algorithm** organizes the
list of active nodes as a first-in-first-out queue. New active nodes are added to the back of the list. If an active node $i$ is removed from the head of the list we repeatedly call \textsc{PUSH-RELABEL}($i$) until either $i$ becomes inactive or its label increases. In the latter case, $i$ is added again to the back of the list.

**Theorem 2.29** The FIFO-preflow-push algorithm finds a maximum flow in $O(n^3)$ time.

**Proof:** By our previous observations it suffices to show that there are only $O(n^3)$ non-saturating pushes. To this end, we partition the execution of the algorithm into phases. Phase 1 consists of the processing of all nodes that are active after the initialization. For $i \geq 1$, Phase $i + 1$ consists of the processing of all nodes that are added to the queue in Phase $i$.

Note that in any phase there are at most $n$ non-saturating pushes, since a non-saturating push causes the current active node to become inactive. Later considerations of the node involving pushes fall into later phases.

We use a potential function argument to bound the number of phases. Let again $I$ denote the set of active nodes. Our potential is defined as

$$
\Phi := \max \{ d[i] : i \in I \}.
$$

We call a phase an increasing phase if at the end of the phase $\Phi$ has not decreased compared to the value at the beginning. Otherwise, the phase is called a decreasing phase.

An increasing phase can only occur if at least one relabel operation is carried out during the phase: if no relabel operation takes place, then for all nodes that are active at the beginning of the phase, their excess has been successfully pushed to nodes with strictly smaller distance labels, which means that $\Phi$ decreases. Since by Lemma 2.24 there are $O(n^2)$ relabel operations, we have $O(n^3)$ increasing phases. Consequently, the number of non-saturating pushes in increasing phases is at most $n \cdot O(n^2) = O(n^3)$.

We obtain a bound on the number of decreasing phases if we sum up all increases in potential over all increasing phases (recall that a decreasing phase strictly decreases the potential).

Consider an increasing phase and let $i$ be an active node with maximum label $d'[i]$ at the end of the phase, that is, a node that determines the potential value. If $d[i]$ denotes the label of $i$ at the beginning of the phase, then the potential can not have increased by more than $d'[i] - d[i]$ during the phase. Hence, the total increase in potential over all increasing phases is bounded by the total increase in distance labels, which is at most $\sum_{i \in V} (2n-1) = O(n^2)$. Thus, there are only $O(n^2)$ decreasing phases. Again, since a phase contains at most $n$ non-saturating pushes, the claim of the theorem follows. \hfill $\Box$

### 2.9 Exercises

**Exercise 2.1 (Seat assignment as a Max Flow Problem)**

Some families go out for dinner together. For social reasons they would like to choose their seats such that no two members of the same family are sitting on the same table. Show that this problem of assigning seats can be formulated as a Max Flow Problem. Assume that there are $p$ families where the $i$-th family has $a(i)$ members which are to be placed around $q$ available tables where the $j$-th table has a capacity of $b(j)$ seats.

**Exercise 2.2 (Reductions to max flow problem)**

Show that the following extensions to the maximum flow problem can be reduced to the standard maximum flow problem:
2.9 Exercises

(a) Besides the edge capacities there are given node capacities.

(b) Multiple sources $s_1, \ldots, s_k$, for $k \geq 2$, and multiple sinks $t_1, \ldots, t_l$, for $l \geq 2$, are given. The goal is to maximize the total flow from the sources to the sinks.

Exercise 2.3 (Bit Scaling Algorithm (Gabow, 1985))

Let $U := \max\{u(a) : a \in A\}$ and $K = \lceil \log U \rceil$. In the bit-scaling algorithm for the maximum flow problem, we present each arc capacity as a $K$-bit binary number, adding leading zeros if necessary to make each capacity $K$ bits long. The problem $P_k$ considers the capacity of each arc as the $k$ leading bits. Let $x^*_k$ denote a maximum flow and let $v^*_k$ denote the maximum flow value in the Problem $P_k$. The algorithm solves a sequence of problems $P_1, P_2, P_3, \ldots, P_K$, using $2x^*_{k-1}$ as a starting solution for the problem $P_k$.

(a) Show that $2x^*_{k-1}$ is feasible for $P_k$ and that $v^*_k - 2v^*_{k-1} \leq m$.

(b) Show that the shortest augmenting path algorithm for solving problem $P_k$, starting with $2x^*_{k-1}$ as the initial solution, requires $O(m^2)$ time. Conclude that the bit-scaling algorithm solves the maximum flow problem in $O(m^2 \log U)$ time.

Exercise 2.4 (Marriage Theorem)

Consider the situation where we are given $n$ men $H$ and $n$ women $D$. For each woman $d \in D$ there is a subset $N(d) \subseteq H$ of the men who are willing to marry $d$. This exercise develops a necessary and sufficient condition such that under given compatibility relations the women and men can all be married to a feasible partner.

Suppose $G = (H \cup D, E)$ is an undirected bipartite graph. A matching is a subset $M \subseteq E$ such that $M$ contains no two incident edges. In a maximum matching it is not possible to add an edge without destroying the matching property. A matching $M$ is perfect if each node in $V = H \cup D$ is incident to an edge in $M$.

For a set $D' \subseteq D$ let $N(D') \subseteq H$ denote the set of nodes in $H$ that are adjacent to at least one node in $D'$. Show that the bipartite graph $G = (H \cup D, E)$ has a perfect matching if and only if $|N(D')| \geq |D'|$ for all $D' \subseteq D$.

Hint: Use the Max-Flow-Min-Cut-Theorem.

Exercise 2.5 (Stable Marriage)

This exercise concerns a variant of the marriage problem. Again, we are given men $H$, women $D$ with $n = |H| = |D|$. But this time, each man/woman has a sorted preference list of all the people of the other sex, that is, the preference list of a man (woman) is a permutation of the women (men). A perfect matching in the complete bipartite graph with vertex set $D \cup H$ is stable, if it does not contain an unstable pair. An unstable pair is a man $h$ and a woman $d$ which are not matched together and such that $h$ would prefer $d$ to his current partner while at the same time $d$ would also prefer $h$ to her current match. The surprising result that you should prove is that there always exists a stable perfect matching.

Hint: Consider the following algorithmic approach called the proposal algorithm: As long as there exists an unmatched man, choose such a man $h$. Then $h$ proposes to the first woman $d$ in his list which has not rejected him yet. The woman $d$ accepts the proposal if she is currently single or her current mate is worth less than $h$ (in her opinion). In the latter case she dumps her current partner and mates with $h$. Show that the proposal algorithm terminates with a stable marriage.

Food for thought: How many proposals does the algorithm use in the worst case? Can you prove something about the average case performance in the following setting: the lists of the women are arbitrary, but the lists of the men are chosen independently and uniformly at random.
3.1 Introduction

Let $G = (V, A)$ be a directed network with capacities $u: A \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. Notice that we allow some or all of the capacities to be infinite. In addition to the maximum flow problems treated in Chapter 2 we are also given costs $c: A \to \mathbb{R}_{\geq 0}$ on the arcs of $G$, where $c(i, j)$ denotes the cost of sending one unit of flow over the arc $(i, j)$. We associate with each node $i \in V$ a number $b(i)$ which indicates its desired supply or demand depending on whether $b(i) > 0$ or $b(i) < 0$. The minimum cost flow problem consists of finding a flow in $G$ such that all nodes are balanced, that is, every node $i \in V$ has excess $b(i)$ with respect to $f$. Thus, the minimum cost flow problem can be stated formally as follows:

$$
\min \sum_{(i,j) \in A} c(i, j)f(i, j) \quad (3.1a)
$$

$$
\sum_{j : (j, i) \in A} f(j, i) - \sum_{j : (i, j) \in A} f(i, j) = b(i) \quad \text{for all } i \in V \quad (3.1b)
$$

$$
0 \leq f(i, j) \leq u(i, j) \quad \text{for all } (i, j) \in A \quad (3.1c)
$$

We call a function $f$ satisfying (3.1b) a $b$-flow (for the imbalance vector $b$). If $f$ additionally satisfies the capacity constraints (3.1c), then it is a feasible $b$-flow. As in Chapter 2 we call the constraints (3.1b) the mass balance constraints and (3.1c) the capacity constraints.

Notice that we can restate the mass balance constraints using our notion of excess introduced in the previous chapter, namely (3.1b) is equivalent to

$$
e_f(i) = b(i) \quad \text{for all } i \in V.
$$

We refer to $e_f(i)$ again as the excess of node $i$. Nodes with $e_f(i) = 0$ will be referred to as balanced nodes.

The complexity of some our algorithms depends on the numbers given in the instance. We will denote by $U$ the maximum finite capacity in an instance. Similarly, $C := \max_{(i,j) \in A} c(i, j)$ and $B := \max_{i \in V} |b(i)|$ denote the maximum cost of an arc and the maximum absolute value of the supply/demand of a node. Finally, we let $\bar{U} = \max\{U, B\}$.

As in our presentation of the maximum flow problem we will make some assumptions about the structure of an instance which simplify the presentation while imposing no restrictions:

1. The network does not contain parallel arcs and for each arc $(i, j)$ the network also contains the inverse arc $(j, i)$.

These assumptions are just as in the maximum flow problems in the previous chapter.

2. All data (cost, supply/demand, and capacity) are integral.
3. The supplies/demands at the nodes satisfy \( \sum_{i \in V} b(i) = 0 \) and the minimum cost flow problem has a feasible solution.

Clearly, if the condition \( \sum_{i \in V} b(i) \) is not met, then the instance does not have a feasible solution. We can test for feasibility of an instance as follows: We add two new nodes \( s \) and \( t \). The source node \( s \) is connected via arcs \((s, i)\) of capacity \( b(i) \) to all nodes \( i \) with positive supply \( b(i) > 0 \). The sink \( t \) is connected to all nodes \( j \) with \( b(j) < 0 \) via arcs \((i, t)\) of capacity \( -b(i) \).

We now solve the problem of finding a maximum \((s, t)\)-flow. The original instance has a feasible solution if and only if the maximum flow in the augmented network saturates all arcs \((s, i)\) and \((i, t)\).

4. For any pair of nodes \( i \) and \( j \), there is a directed path from \( i \) to \( j \) consisting only of uncapacitated arcs.

We can enforce this property by adding artificial uncapacitated arcs of large cost. No such arc would be used in an optimal solution.

A problem very closely related to the minimum cost flow problem is the minimum cost circulation problem. In this problem all required node imbalances are zero, but costs may be negative. Standard techniques can be used to transform the minimum cost circulation problem into a minimum cost flow problem and vice versa, see e.g. [1, 6].

### 3.2 Disgression: Total Unimodularity and Applications of the Minimum Cost Flow Problem

**Definition 3.1 (Totally unimodular matrix)**

An \( p \times q \)-matrix \( A \) with integer elements is called **totally unimodular**, if the determinant of every square submatrix of \( A \) is either 0, -1 or +1.

**Theorem 3.2 (Unimodularity Theorem)** Let \( A \) be a matrix with integer entries and linearly independent rows. Then, the following three conditions are equivalent:

(i) \( A \) is unimodular.

(ii) Every basic feasible solution of \( \{ x : Ax = b, x \geq 0 \} \) is integral for every integer vector \( b \).

(iii) Every basis matrix \( B \) of \( A \) has an integer inverse \( B^{-1} \).

**Proof:** See Exercise 3.2 \( \square \)

**Theorem 3.3** Let \( M \) be any \( m \times n \) matrix with entries taken from \( \{0, +1, -1\} \) with the property that any column contains at most one \(+1\) and at most one \(-1\). Then \( M \) is totally unimodular.

**Proof:** See Exercise 3.3 \( \square \)

The node-arc incidence matrix of a directed network \( G = (V, A) \) is the \( n \times m \)-Matrix \( M(A) = (m_{xy}) \) such that

\[
m_{xy} = \begin{cases} 
+1 & \text{if } a = (i, j) \text{ and } x = i \\
-1 & \text{if } a = (i, j) \text{ and } y = j \\
0 & \text{otherwise}
\end{cases}
\]
Corollary 3.4 The node-arc incidence matrix of a directed network is totally unimodular.

Proof: Immediate from the definition of the node-arc incidence matrix and Theorem 3.2.

Theorem 3.5 (Consecutive ones Theorem) Let M be any \( m \times n \)-matrix with entries from \( \{0, 1\} \) and the property that the rows of \( M \) can be permuted in such a way that all 1s appear consecutively. Then, \( M \) is totally unimodular.

Proof: See Exercise 3.4.

Suppose that we are given a Linear Programming Problem of the form

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{subject to} & \quad Ax \leq b \\
& \quad x \geq 0,
\end{align*}
\]

where \( A \) is a \( p \times q \)-matrix with the consecutive ones property stated in Theorem 3.5. We assume that the rows of \( A \) have already been permuted such that the ones appear consecutively. By adding slack variables \( z \geq 0 \) we bring the constraints of (3.2) into equality form. We also add a redundant row \( 0 \cdot x + 0 \cdot z = 0 \) to the set of constraints. This yields the following equivalent version of (3.2):

\[
\begin{align*}
\text{min} & \quad c_0^T x \\
\text{subject to} & \quad A_0 x + z = b_0 \\
& \quad x \geq 0,
\end{align*}
\]

where \( A_0 \) is a matrix with entries from \( \{1, 0, +1\} \) such that each column contains exactly one \( -1 \) and one \( +1 \). Hence, (3.4) is a minimum cost flow problem with node-arc incidence matrix \( A_0 \) and required balances \( b_0 \). We have thus shown that any Linear Programming Problem of the form (3.2) and a matrix with the consecutive ones property can be solved by means of a minimum cost flow computation. Since we will learn efficient algorithms for computing minimum cost flows in this chapter, this means that we can also solve any LP of the special form efficiently, too. In particular the results in Section 3.10 show that these LPs can be solved by a strongly polynomial time algorithm. So far, no strongly polynomial time algorithm for general Linear Programs is known.
3.3 Flow Decomposition

In our formulation of the minimum cost flow problem we have used arc variables \( f(i, j) \) to specify the flow. There is an alternative formulation which turns out to be useful on several occasions. The path and cycle flow formulation starts with an enumeration \( P \) of all directed paths and \( W \) of all cycles in the network. The variables in the formulation will be \( f(P) \), the flow on path \( P \), and \( f(W) \), the flow on cycle \( W \). These are defined for every directed path and cycle.

Every set of path and cycle flows uniquely determines arc flows: the flow on arc \((i, j)\) is the sum of the flows \( f(P) \) and \( f(W) \) of all paths and cycles that contain the arc \((i, j)\). We will now show that the converse is also true in the sense that every arc flow induces path and cycle flows.

**Theorem 3.6 (Flow Decomposition Theorem)** Every path and cycle flow has a unique representation as nonnegative arc flows. Conversely, every nonnegative arc flow \( f \) can be represented as a path and cycle flow (though not necessarily uniquely) with the following properties:

(i) Every directed path with positive flow connects a node \( i \) with negative excess \( e_f(i) < 0 \) to a node \( j \) with positive excess \( e_f(j) > 0 \).

(ii) At most \( n + m \) paths and cycles have nonzero flow; out of these at most \( m \) cycles have nonzero flow.

**Proof:** We only need to prove the «decomposition» of an arc flow into a path and cycle flow. Our proof will in fact be an algorithmic one, that is, it provides a method of determining the decomposition.

Let \( f \) be such arc flow. Let \( i \) be a node with negative excess \( e_f(i) < 0 \) with respect \( f \). Then, some arc \((i, i_1)\) must carry positive flow. If \( i_1 \) is already a node with \( e_f(i_1) > 0 \), we stop. Otherwise, \( e_f(i_1) \leq 0 \) and it follows that there must be an arc \((i_1, i_2)\) with positive flow. We continue until we either arrive at a node \( j \) with positive excess or visit a previously visited node. One of these cases must occur after at most \( n \) steps, since the graph is finite and contains \( n \) nodes.

In the first case, we have a directed path \( P \) from \( i \) to a node \( j \) with \( e_f(j) > 0 \). We let

\[
 f(P) := \min \{ -e_f(i), e_f(j), \min \{ f(i, j) : (i, j) \in P \} \}.
\]

We redefine \( e_f(i) := e_f(i) + f(P) \), \( e_f(j) := e_f(j) - f(P) \) and \( f(i, j) := f(i, j) - f(P) \) for all arcs \((i, j) \in P\).

In the second case, we have found a directed cycle \( W \). We let \( f(W) := \min \{ f(i, j) : (i, j) \in W \} \) and redefine \( f(i, j) := f(i, j) - f(W) \) for all arcs \((i, j) \in W\).

Notice that each of the above cases leads to a new arc flow and that all arc flow values can only decrease. Moreover, the excess of a node strictly decreases (first case) or the flow on some arcs strictly decreases (first and second case). More precisely, in the first case either the excess/deficit of a node reduces to zero or the flow on an arc reduces to zero. In the second case the flow of at least one arc is reduced to zero. Thus, the procedure must terminate after at most \( n + m \) steps.

At termination, all node balances must be zero (since otherwise we could still repeat the process above). We now choose a node where at least one outgoing arc has positive flow. Carrying out the above procedure we find a directed cycle which gives us the flow value for another cycle. Again, at least one arc flow is reduced to zero and we must terminate with the flow \( f \equiv 0 \) after at most \( m \) steps. We can even bound the total number of paths and
flows collected by $n + m$, since everytime we collect a cycle, the flow of at least one arc goes down to zero. This also bounds the total number of cycles by $m$.

Clearly, the original arc flow equals the sum of the flows on the paths and cycles. Moreover, since we have collected at most $n + m$ paths and cycles among which are at most $m$ cycles, the bounds given in the theorem follow.

As for maximum flows, residual networks turn out to be quite helpful. We extend our definition from Section 2.3 to include costs.

**Definition 3.7 (Residual network)**

Let $f$ be a flow in the network $G$ with capacities $u: A \to \mathbb{R}_{\geq 0}$ and costs $c: A \to \mathbb{R}_{\geq 0}$. The residual network $G_f$ has the same node set as $G$ and contains for each arc $(i, j) \in A$ up to two arcs:

- If $f(i, j) < u(i, j)$, then $G_f$ contains an arc $(i, j)^+$ of residual capacity $r((i, j)^+) := u(i, j) - f(i, j)$ and cost $c((i, j)^+)$.
- If $f(i, j) > 0$, then $G_f$ contains an arc $(j, i)^-$ of residual capacity $r((j, i)^-) := f(i, j)$ and cost $-c((i, j))$.

Residual networks allow us to compare two solutions for the minimum cost flow problem. A directed cycle $W_f$ in the residual network $G_f$ induces an augmenting cycle $W$ in the original network $G$, that is, a cycle along which we can increase/decrease flow on the arcs by a small positive amount without violating the feasibility of the flow. The cost of the augmenting cycle $W$ is defined to be the cost of the corresponding directed cycle in $W_f$.

**Theorem 3.8** Let $f$ and $f_0$ be any two feasible solutions for the minimum cost flow problem. Then $f$ equals $f_0$ plus the flow on at most $m$ directed cycles in $G_{f_0}$. The cost of $f$ is equal to the cost of $f_0$ plus the cost of flow on these cycles.

**Proof:** The flow $f - f_0$ is a flow in $G_{f_0}$. By the flow decomposition theorem it can be decomposed into path and cycle flows. However, since any node is balanced with respect to $f - f_0$, only cycles can appear in the decomposition.

**3.4 Disgression: Multicommodity Flows and Column Generation**

The flow decomposition theorem turns out to be useful in several contexts. In this section we briefly diverge from the minimum cost flow problem and show how decomposition can be used to obtain efficient solutions for a multicommodity flow problem.

In the maximum multicommodity flow problem we are given source/sink pairs $(s_k, t_k)$, $k = 1, \ldots, K$ with demands $d_k$. The goal of the problem is to find a $(s_k, t_k)$-flow $f^k$ of value $d_k$ for each $k$ such that for each arc $(i, j) \in A$ the sum of the flows on it does not exceed its capacity, that is,

$$
\sum_{k=1}^{K} f^k(i, j) \leq u(i, j) \quad \text{for all } (i, j) \in A
$$

(3.5)
36 Minimum Cost Flows

We could formulate the problem as Linear Program in the obvious way. Thereby, we would search for flow vectors \( f^k \) subject to the following constraints:

\[
\begin{align*}
e_{f^k}(i) &= 0 & \text{for all } k \text{ and all } i \notin \{s_k, t_k\} \quad (3.6a) \\
e_{f^k}(t_k) &= d_k & \text{for all } k \quad (3.6b) \\
e_{f^k}(s_k) &= -d_k & \text{for all } k \quad (3.6c) \\
\sum_{k=1}^{K} f^k(i, j) &\leq u(i, j) & \text{for all } (i, j) \in A \quad (3.6d) \\
f^k(i, j) &\geq 0 & \text{for all } (i, j) \in A \quad (3.6e)
\end{align*}
\]

The disadvantage is the large number of variables in (3.6). We will present an alternative LP-formulation that uses even more variables but which can be solved more efficiently.

3.4.1 A Review of the Simplex Method

To make the presentation self-contained we briefly recall the Simplex method for solving an LP in standard form. Consider the following pair of Linear Programs:

\[
\begin{align*}
\text{(P)} \quad \max c^T x \quad &Ax = b \quad x \geq 0 \\
\text{(D)} \quad \min b^T y \quad &A^T y \geq c
\end{align*}
\]

A basis for (P) is an index set \( B \) of the variables of (P) such that the corresponding sub-matrix \( A_B \) of \( A \) is nonsingular. The basis is termed feasible if \( x_B := A_B^{-1}b \geq 0 \). Clearly, in this case \( (x_B, x_N) \) with \( x_N := 0 \) is a feasible solution of (P). The simplex method for solving (P) iterates from basis to basis never decreasing the objective function value. It is easy to see that a basic solution \( (x_B, x_N) \) of (P) is optimal if and only if the vector \( y \) defined by \( y^T A_B = c_B^T \) satisfies \( A^T y \leq c \). We only outline the basics and refer to standard textbooks on Linear Programming for details, e.g. [20, 6].

Given a basis \( B \), the Simplex method solves \( y^T A_B = c_B^T \) to obtain \( y \). If \( A^T y \geq c \), then we have an optimal solution for (P), since

\[
c^T x = c_B^T x_B = y^T A_B x_B = y^T b = b^T y,
\]

and for any pair \((x', y')\) of feasible solutions for (P) and (D), respectively, we have

\[
c^T x' \leq (A^T y')^T x' = (y')^T A x' = b^T y'.
\]

If on the other hand, \( a_i^T y < c_i \) for some \( i \), we can improve the solution by adding variable \( x_i \) to the basis and throwing out another index. We first express the new variable in terms of the old basis, that is, we solve \( A_B z = a_i \). Let \( x(\varepsilon) \) be defined by \( x_B(\varepsilon) := x_B - \varepsilon z, x_i(\varepsilon) := \varepsilon \) and zero for all other variables. Then,

\[
c_B^T x(\varepsilon) = c_B^T(x_B - \varepsilon z) + c_i \varepsilon \\
= c_B^T x_B + \varepsilon (c_i - c_B^T z) \\
= c_B^T x_B + \varepsilon (c_i - y^T A_B z) \\
= c_B^T x_B + \varepsilon (c_i - y^T A_B z) > 0
\]

Thus, for \( \varepsilon > 0 \) the new solution \( x(\varepsilon) \) is better than the old one \( x \). The Simplex method now chooses the largest possible value of \( \varepsilon \), such that \( x(\varepsilon) \) is feasible, that is \( x(\varepsilon) \geq 0 \). This operation will make one of the old basic variables \( j \) in \( B \) become zero. The selection of \( j \) is usually called the ratio test, since \( j \) is any index in \( B \) such that \( z_j > 0 \) and \( j \) minimizes the ratio \( x_k/z_k \) over all \( k \in B \) having \( z_k > 0 \).
3.4.2 Column Generation for the Multicommodity Flow Problem

We return to our multicommodity flow problem. For each commodity \( k \) we add an artificial sink \( t_0^k \) to the network which is connected to the sink \( t_k \) via an arc \((t_k, t_0^k)\) of capacity \( d_k \). The problem can now be reformulated as that of maximizing the sum of the \((s_k, t_0^k)\)-flows over all commodities.

Let \( P_k \) denote the collection of all directed \((s_k, t_0^k)\)-paths in \( G \) and \( P = \bigcup_{k=1}^{K} P_k \). Since the flow decomposition theorem allows us to express each flow defined on the arcs by a flow on paths, we can restate our multicommodity flow problem as follows:

\[
\begin{align*}
\max & \quad \sum_{P \in P} f(P) \\
\text{s.t.} & \quad f(P) \leq u(i, j) \quad \text{for all } (i, j) \in A \\
& \quad f(P) \geq 0 \quad \text{for all } P \in P
\end{align*}
\]

We add slack variables \( f(i, j) \) for each \((i, j) \in A\) to make the constraints (3.7b) equality constraints. Then, the LP (3.7) turns into an LP in standard form:

\[
\begin{align*}
\max & \quad w^T f \\
\text{s.t.} & \quad (A \ I)f = u \\
& \quad f \geq 0
\end{align*}
\]

with corresponding dual:

\[
\begin{align*}
\min & \quad u^T y \\
\text{s.t.} & \quad A^T y \geq w \\
& \quad \sum_{(i,j) \in P} y(i,j) \geq 1 \quad \text{for all } P \in P \\
& \quad y \geq 0 \\
& \quad y(i,j) \geq 0 \quad \text{for all } (i,j) \in A
\end{align*}
\]

We use the Simplex method as outlined in Section 3.4.1 to solve (3.8). A basis of (3.8) is a pair \((P', A')\) where \( P' \subseteq P \), \( A' \subseteq A \), \(|P'| + |A'| = |A|\) and the matrix \( B \) consisting of the corresponding columns of \((A \ I)\) is nonsingular. Note that \( P' = \emptyset \) and \( A' = A \) provides a feasible start basis.

In the Simplex algorithm we need to check for optimality of a current basic solution \( f \). To this end, we have to check whether \( y \), defined by \( y^T B = w_B^T \), is feasible for the dual. This amounts to checking the following two conditions:

\[
\begin{align*}
\text{(i)} & \quad \sum_{(i,j) \in P} y(i,j) \geq 1 \quad \text{for all } P \in P; \\
\text{(ii)} & \quad y(i,j) \geq 0 \quad \text{for all } (i,j) \in A.
\end{align*}
\]

The second condition is easy to check. The first condition involves checking an exponential number of paths. However, having a closer look at (i) reveals that (i) asks whether there is a path in \( P \) of length smaller than 1, where arc weights are given by \( y \geq 0 \). Thus, we can check the condition (i) by performing a shortest path computation on an auxiliary graph with nonnegative arc weights. This provides an efficient (in particular polynomial time) method of checking (i).

If we find that \( y \) is feasible, then we are done. In the other case, we have either \( y(i,j) < 0 \) for some arc \( a = (i,j) \) or \( \sum_{(i,j) \in P} y(i,j) < 1 \) for some path \( P \). In any case we have a new variable that we would like to have enter the basis. In the former case we solve \( Bz = 1_a \), in the latter we solve \( Bf = \chi_P \) in order to express the new variable in terms of the old basis. Here, \( 1_a \) denotes the unit vector having a one at position \( a \) and \( \chi_P \) is the characteristic vector of \( P \) in \( \mathbb{R}^{|A|} \). Once we have \( z \), we can select a basic variable to leave the basis by the standard ratio test.
3.5 LP-Duality and Optimality Conditions

We can obtain optimality conditions for minimum cost flows by considering the dual to (3.1). To make things more transparent, we rewrite (3.1) to:

\[
\begin{align*}
\min & \quad \sum_{(i,j) \in A} c(i,j) f(i,j) \\
\text{subject to} & \quad \sum_{j : (j,i) \in A} f(j,i) - \sum_{j : (i,j) \in A} f(i,j) = b(i) \quad \text{for all } i \in V \\
& \quad -f(i,j) \geq -u(i,j) \quad \text{for all } (i,j) \in A \\
& \quad f(i,j) \geq 0 \quad \text{for all } (i,j) \in A
\end{align*}
\]  

The dual of the Linear Program (3.10) is:

\[
\begin{align*}
\max & \quad \sum_{i \in V} b(i) \pi(i) - \sum_{(i,j) \in A} u(i,j) z(i,j) \\
\text{subject to} & \quad -\pi(i) + \pi(j) - z(i,j) \leq c(i,j) \quad \text{for all } (i,j) \in A \\
& \quad z(i,j) \geq 0 \quad \text{for all } (i,j) \in A
\end{align*}
\]

In the above we use the convention that, if \( u(i,j) = \infty \) there is no dual variable \( z(i,j) \) since the corresponding constraint \( f(i,j) \leq u(i,j) \) is missing from (3.10).

It will be convenient to call

\[ c^\pi(i,j) := c(i,j) + \pi(i) - \pi(j) \]

the reduced cost of arc \((i,j)\) with respect to the node potential \( \pi \). In this context we also allow \( \pi \) to be any real valued function defined on the node set.

**Observation 3.9** Let \( \pi \) be any node potential. The reduced costs \( \pi \) have the following properties:

(i) For any directed path \( P \) from node \( k \) to node \( l \) we have \( \sum_{(i,j) \in P} c^\pi(i,j) = \sum_{(i,j) \in P} c(i,j) + \pi(k) - \pi(l) \)

(ii) For any directed cycle \( W \), \( \sum_{(i,j) \in W} c^\pi(i,j) = \sum_{(i,j) \in W} c(i,j) \).

Let \((\pi, z)\) be any feasible solution for the dual (3.11). Then, for all arcs \((i,j)\) with \( u(i,j) = +\infty \) we have \( \pi(i) - \pi(j) \leq c(i,j) \) or \( c^\pi(i,j) \geq 0 \). If \( u(i,j) < \infty \), then the only condition on the variable \( z(i,j) \) is \( z(i,j) \geq -c^\pi(i,j) \). Clearly, the best (in terms of the objective function) we can do is choose \( z(i,j) = -c^\pi(i,j) \) in this case. In any case, given \( \pi \), the best choice for the \( z \)-variables is to set \( z(i,j) = \max\{0, -c^\pi(i,j)\} \). Thus, in some sense, the \( z \)-variables are superfluous. We will refer to a node potential \( \pi \) alone as a feasible (optimal) dual solution, if it can be extended by the above setting to a feasible (optimal) dual solution.

We will now write the complementary slackness conditions for the pair of dual problems (3.10) and (3.11):

\[
\begin{align*}
f(i,j) > 0 \Rightarrow c^\pi(i,j) = -z(i,j)
\end{align*}
\]  

Considering the fact that \( z(i,j) = \max\{0, -c^\pi(i,j)\} \) we can reformulate the condition \( c^\pi(i,j) = -z(i,j) \) in (3.13) as \( c^\pi(i,j) \leq 0 \). Similarly plugging \( z(i,j) = ...
max\{0, -c^\pi(i,j)\} into (3.13) gives that $-c^\pi(i,j) > 0$ implies $f(i,j) = u(i,j)$. We can thus reformulate (3.13) and (3.14) equivalently as
\begin{align*}
f(i,j) > 0 & \Rightarrow c^\pi(i,j) \leq 0 \\
c^\pi(i,j) < 0 & \Rightarrow f(i,j) = u(i,j).
\end{align*}
(3.15) (3.16)

Since (3.15) in turn is equivalent to $c^\pi(i,j) > 0 \Rightarrow f(i,j) = 0$, we obtain the following theorem:

**Theorem 3.10** A feasible solution $f$ for the minimum cost flow problem (3.1) is optimal if and only if there exists a node potential $\pi$ such that for all $(i,j) \in A$ the following conditions hold:
\begin{align*}
c^\pi(i,j) > 0 & \Rightarrow f(i,j) = 0 \\
c^\pi(i,j) < 0 & \Rightarrow f(i,j) = u(i,j)\, (\neq \infty)
\end{align*}
Moreover, every pair $(f, \pi)$ of optimal primal and dual solutions satisfies these conditions.

**Proof:** If $G_f$ contains a negative cost cycle $W$, we can augment the flow along $W$ by a positive amount without changing node imbalances but strictly decreasing the cost. Thus, $f$ cannot be optimal.

Assume conversely that $G_f$ does not contain a negative cost cycle. We denote by $c'$ the arc costs in $G_f$. We add a new node $r$ to $G_f$ which is connected via arcs $(r, i)$ to all nodes $i \in V$. The arcs $(r, i)$ have zero cost. Since $G_f$ does not contain a negative cost cycle the shortest path distances $d$ from $r$ are well defined and satisfy:
\begin{equation}
d(y) \leq d(x) + c'(x,y) \text{ for all } (x,y) \in G^f. (3.17)
\end{equation}

If $f(i,j) > 0$, then we have $(j, i^-) \in G_f$ and from (3.17) it follows that $d(i) \leq d(j) - c(i,j)$ or $c(i,j) + \pi(i) - \pi(j) \leq 0$. In other words, we have $c^\pi(i,j) \leq 0$. This means that condition (3.15) is satisfied for $\pi := d$.

On the other hand, if $f(i,j) < u(i,j)$, then $(i,j)^+ \in G_f$ and from (3.17) we obtain that $c^\pi(i,j) \geq 0$. This condition is equivalent to (3.16). Thus, by Theorem 3.10 $f$ must be an optimal solution.

Note that in our above constructive proof of Theorem 3.11 the potential $\pi := d$ constructed is integral, provided all arc costs are integral (since shortest paths preserve integrality). This shows the following interesting result:

**Theorem 3.12** Suppose that the minimum cost flow problem (3.1) has a feasible solution and all costs are integral. Then, the dual problem (3.11) has an optimal solution that is integral.

We have shown in Theorem 3.11 how to construct an optimal dual solution given an optimal flow. We now consider the converse problem, that is, given an optimal dual solution $\pi$ we want to find an optimal flow $f$. For convenience, we recall the conditions that $f$ needs to satisfy:
\begin{align*}
e_f(i) &= b(i) \text{ for all } i \in V \quad (3.18a) \\
0 &\leq f(i,j) \leq u(i,j) \text{ for all } (i,j) \in A \quad (3.18b) \\
f(i,j) &= 0 \text{ if } c^\pi(i,j) > 0 \quad (3.18c) \\
f(i,j) &= u(i,j) \text{ if } c^\pi(i,j) < 0 \quad (3.18d)
\end{align*}
Let $u'$ and $l'$ be defined by
\[
\begin{align*}
    u'(i, j) &= \begin{cases} 
        0 & \text{if } c^\pi(i, j) > 0 \\
        u(i, j) & \text{otherwise}
    \end{cases} \\
    l'(i, j) &= \begin{cases} 
        u(i, j) & \text{if } c^\pi(i, j) < 0 \\
        0 & \text{otherwise}
    \end{cases}
\end{align*}
\]

Then, the conditions (3.18) are equivalent to finding a flow subject to the conditions that
\[
\begin{align*}
    e_f(i) &= b(i) & \text{for all } i \in V \\
    l'(i, j) &\leq f(i, j) \leq u'(i, j) & \text{for all } (i, j) \in A
\end{align*}
\]

This can be achieved by the method described in Section 2.6 which uses one maximum flow computation. Since maximum flow computations preserve integrality, we obtain the following result:

**Theorem 3.13** Suppose that the minimum cost flow problem (3.1) has a feasible solution and that $b$ and $u$ are integral. Then, the problem has an optimal solution that is integral.

We conclude this section by summarizing the optimality conditions that we have obtained in this section for easier reference:

**Theorem 3.14** For a feasible flow, the following statements are equivalent:

(i) $f$ is a minimum cost flow.

(ii) The residual graph $G_f$ does not contain a negative cost cycle. (negative cycle optimality conditions)

(iii) There exists a node potential $\pi$ such that $c^\pi((i, j)^{\delta}) \geq 0$ for all $(i, j)^{\delta} \in G_f$. (reduced cost optimality conditions)

(iv) There exists a node potential $\pi$ such that for all $(i, j) \in A$ the following conditions hold:
\[
\begin{align*}
    c^\pi(i, j) > 0 &\Rightarrow f(i, j) = 0 \\
    c^\pi(i, j) < 0 &\Rightarrow f(i, j) = u(i, j)(\neq \infty)
\end{align*}
\]

(complementary slackness conditions (I))

(v) There exists a node potential $\pi$ such that for every arc $(i, j) \in A$ the following conditions hold:

- If $c^\pi(i, j) > 0$, then $f(i, j) = 0$.
- If $0 < f(i, j) < u(i, j)$, then $c^\pi(i, j) = 0$.
- If $c^\pi(i, j) < 0$, then $f(i, j) = u(i, j)$.

(complementary slackness conditions (II))

Proof:

(i)$\Leftrightarrow$(ii) This statement is proved in Theorem 3.11.

(ii)$\Leftrightarrow$(iii) This statement is an immediate consequence of Observation 3.9.

(i)$\Leftrightarrow$(iv) This statement is subject of Theorem 3.10.

(iv)$\Leftrightarrow$(v) The conditions in (v) are just a reformulation of the conditions in (iv).
3.6 The Cycle Cancelling Algorithm of Klein

The optimality condition given in Theorem 3.11 immediately suggests an algorithm for computing a minimum cost flow. We first compute an arbitrary feasible flow $f$. This can be achieved as described in Section 2.6. Then, as long as the residual network $G_f$ contains a negative cost cycle, we identify such a cycle $W$ and "cancel" it, by pushing $(W) := \min_{(i,j) \in W} R((i,j)^k)$ units of flow along the cycle. Recall that a negative cost cycle can be determined in $O(nm)$ time, e.g. by the Bellman-Ford-Algorithm (see [7]). By pushing $(W)$ units of flow along the cycle, the residual capacity of at least one arc in $W$ reduces to zero which makes the arc disappear from the residual network. Hence we can rightly speak of cancelling the cycle $W$.

If all data is integral (as we assumed), and we started with an integral flow (which we can obtain by the maximum flow computation), then all intermediate flows are also integral. Moreover, each cycle cancelling step decreases the cost of the current flow by an integral amount. Thus, the algorithm must terminate after a finite number of steps with a flow of minimum cost. In fact, we can estimate the number of iterations by $O(mUC)$, since any flow (in particular the initial one) has cost at most $mUC$.

**Theorem 3.15** The cycle cancelling algorithm terminates after at most $O(mUC)$ iterations with a feasible flow of minimum cost. The total running time of the algorithm is $O(nm^2UC)$ plus the time required for the computation of the initial flow. \hfill $\square$

The cycle cancelling algorithm provides another constructive proof of the integrality results stated in Theorems 3.13 and 3.12.

The running time we have derived for the cycle cancelling algorithm is not polynomial since $U$ and $C$ need not be polynomially bounded by in $n$ and $m$. The algorithm leaves the question open which negative cycle to cancel if there are many and, in fact, some choices lead to bad running times while others result in a polynomial time algorithm.

3.7 The Successive Shortest Path Algorithm

A *pseudoflow* in a capacitated network $G = (V, A)$ is a function $f : A \rightarrow \mathbb{R}$ satisfying the capacity constraints but not necessarily the mass balance constraints. For a pseudoflow $f$ we define the *imbalance* of a node $i$ with respect to $f$ by

$$\text{imbal}_f(i) := e_f(i) - b(i).$$

If $\text{imbal}_f(i) > 0$ we call $\text{imbal}_f(i)$ the *surplus* of node $i$. If $\text{imbal}_f(i) < 0$, the value $-\text{imbal}_f(i)$ will be called the *deficit* of $i$. A node with zero imbalance is termed a *satisfied node*.

Given a pseudoflow $f$ we let $S_f$ and $D_f$ be the set of surplus and deficit nodes. Observe that $\sum_{i \in S_f} \text{imbal}_f(i) = \sum_{i \in D_f} b(i) = 0$. Hence, $\sum_{i \in S_f} \text{imbal}_f(i) = -\sum_{i \in D_f} \text{imbal}_f(i)$. Consequently, if a network contains a surplus node, then it also must contain a deficit node. We define the residual network for a pseudoflow exactly as in the case of a flow (see Definition 3.7 on page 55).

**Lemma 3.16** Suppose that a pseudoflow $f$ satisfies the reduced cost optimality conditions with respect to some node potential $\pi$. Let $d(v)$ denote the shortest path distances from some node $s$ to all other nodes in $G_f$ with $c^f((i,j)^k)$ as arc lengths.

(i) The pseudoflow $f$ also satisfies the reduced cost optimality conditions with respect to the node potentials $\pi' = \pi + d$.\hfill $\square$
(ii) For each shortest path from node $s$ to a node $v$, the reduced costs $e^{\pi'}((i, j)^d)$ are zero for all arcs on the path.

**Proof:**

(i) By the assumptions stated in the lemma, we have $e^{\pi}((i, j)^d) \geq 0$ for all arcs $(i, j)^d \in G_f$. The shortest path distances $d$ satisfy for all $(i, j)^d \in G_f$ the condition $d(j) \leq d(i) + e^{\pi}((i, j)^d)$. In other words

$$d(j) \leq d(i) + c((i, j)^d) + \pi(i) - \pi(j)$$

$$\Leftrightarrow 0 \leq c((i, j)^d) + (\pi(i) + d(i)) - (\pi(j) + d(j))$$

$$\Leftrightarrow 0 \leq c((i, j)^d) + \pi'(i) - \pi'(j)$$

(ii) This follows from the fact that for each arc $(i, j)^d$ on a shortest path we have $d(j) = d(i) + c((i, j)^d)$.

We are now ready to prove an important property for the design of our next minimim cost flow algorithm.

**Lemma 3.17** Let the pseudoflow $f$ satisfy the reduced cost optimality conditions and let $f'$ be obtained from $f$ by sending flow along a shortest path in $G_f$ from a node $s$ to some other node $v$. Then $f'$ also satisfies the reduced cost optimality conditions.

**Proof:** Let $\pi$ be a node potential such that $e^{\pi}((i, j)^d) \geq 0$ for all $(i, j)^d \in G_f$. We define the new potential $\pi'$ as in the proof of Lemma 3.16. Then, by Lemma 3.16 the arcs on a shortest path from $s$ have all zero cost with respect to $\pi'$. Sending flow along such a shortest path can only influence the arcs on this path, possibly adding an inverse arc $(j, i)^{-d}$ for an arc $(i, j)^d$. However, all these inverse arcs have zero reduced cost and hence, $G_{f'}$ does not contain arcs with negative reduced cost (with respect to $\pi'$).

The results of the preceeding lemma suggest an algorithm for computing a minimum cost flow which is known as the *successive shortest path algorithm*. This algorithm starts with the pseudoflow $f \equiv 0$ and then always sends flow from a surplus node to a deficit node along a shortest path. Algorithm 3.1 describes the algorithm in pseudocode.

**Theorem 3.18** The successive shortest path algorithm correctly computes a minimum cost flow $f$. The algorithm performs at most $O(S(n, m, C) \log n)$ operations, where $S(n, m, C)$ denotes the time required to perform a shortest path computation in a network with $n$ nodes, $m$ arcs and nonnegative arc lengths bounded by $C$, then the total time complexity of the algorithm is $O(nBS(n, m, nC))$.

**Proof:** We show that any intermediate pseudoflow $f$ obtained during the algorithm satisfies the reduced cost optimality conditions with respect to the node potential $\pi$. This is clearly the case for the initial pseudoflow which is zero everywhere.

Note that as long as the set of surplus nodes is nonempty, the set of deficit nodes must be nonempty, too. Thus, in any iteration the algorithm can find a pair of nodes $s$ and $v$ as required. Moreover, by our assumption that the network always contains an uncapacitated path between any pair of nodes, there is always a path between $s$ and $v$ in $G_f$.

By Lemma 3.17 the pseudoflow obtained by augmenting flow along the shortest path will satisfy the reduced optimality conditions with respect to the updated node potential. Thus, if the algorithm terminates, it must terminate with a minimum cost flow.
Algorithm 3.1 Successive shortest path algorithm for computing a minimum cost flow.

**SUCCESSIVE-SHORTEST-PATH**\(G, u, c, b\)

**Input:** A directed graph \(G = (V, A)\) in adjacency list representation; a nonnegative capacity function \(u: E \rightarrow \mathbb{R}_{\geq 0}\), a nonnegative cost function \(c: E \rightarrow \mathbb{R}\); required node imbalances \(b\).

**Output:** A minimum cost flow \(f\)

1. Set \(f := 0\) and \(\pi := 0\)
2. Set \(\text{imbal}_f(i) := -b(i)\) for all \(i \in V\).
3. Compute the sets of surplus and deficit nodes:
   \[ S = \{ i \in V : \text{imbal}_f(i) > 0 \} \]
   \[ D = \{ i \in V : \text{imbal}_f(i) < 0 \} \]

4. **while** \(S \neq \emptyset\) **do**
   5. Select a node \(s \in S\) and a node \(v \in D\).
   6. Determine shortest path distances \(d\) from node \(s\) to all other nodes in \(G_f\) with respect to the reduced costs \(c^e((i,j))\) as arc lengths.
   7. Let \(P\) be a shortest path from \(s\) to \(v\).
   8. Update \(\pi := \pi + d\)
   9. \(\varepsilon := \min\{\text{imbal}_f(s), -\text{imbal}_f(v), \Delta(P)\}\).
   10. Augment \(\varepsilon\) units of flow along \(P\).
   11. Update \(f, G_f, S\) and \(D\).
5. **end while**

Each iteration of the algorithm decreases the imbalance of a surplus node. Thus, after at most \(nB\) iterations, no surplus node remains and the algorithm terminates. In each iteration, the algorithm solves a shortest path problem with nonnegative arc lengths. The reduced cost arc lengths in the residual network are bounded by \(nC\) (rather than \(C\) as in the original network). Hence, the time required for a single iteration is \(S(n, m, nC)\) from which the time bound claimed in the theorem follows. □

**Remark 3.19** Suppose that instead of initializing the successive shortest path algorithm with the zero pseudoflow and zero potential, we use an arbitrary pseudoflow \(f\) and a potential \(\pi\) such that \((f, \pi)\) satisfies the reduced cost optimality conditions, or equivalently, satisfies the complementary slackness optimality conditions.

Then, again the algorithm will terminate with an optimal flow by exactly the same arguments that we used in our analysis above. However, in this case we can bound the number of iterations by
\[
\sum_{i \in V \text{ s.t. } \text{imbal}_f(i)>e_f(i)} (b(i) - e_f(i)).
\]

This follows, since in any iteration the total \(f\)-surplus \(\sum_{i \in V \text{ s.t. } \text{imbal}_f(i)>e_f(i)} (b(i) - e_f(i))\) decreases by at least one.

### 3.8 The Capacity Scaling Algorithm

In Section 3.7, we derived a time bound of \(O(nBS(n, m, nC))\) for the successive shortest path algorithm. Since \(S(n, m, nC) \in O(m + n \log n)\) if we use Dijkstra’s algorithm with a Fibonacci-Heap implementation, the only reason why we do not have a polynomial time bound for the algorithm is the pseudopolynomial number \(nB\) of iterations.
In this section we will design a variant of the successive shortest path algorithm which ensures that in any iteration we augment a »sufficiently large« amount of flow, which helps us decrease the number of iterations and obtain a polynomial time algorithm.

Algorithm 3.2 Capacity scaling algorithm for computing a minimum cost flow.

**CAPACITY-SCALING-ALGORITHM**\((G, u, c, b)\)

**Input:** A directed graph \(G = (V, A)\) in adjacency list representation; a nonnegative capacity function \(u : E \rightarrow \mathbb{R}_{\geq 0}\), a nonnegative cost function \(c : E \rightarrow \mathbb{R}\); required node imbalances \(b\).

**Output:** A minimum cost flow \(f\)

1. Set \(f := 0\) and \(\pi := 0\)
2. Let \(\Delta := 2^{\lfloor \log_2 G \rfloor} \) \hspace{1cm} \{ Beginning of a \(\Delta\)-scaling phase \}
3. while \(\Delta \geq 1\) do
4. for all \((i, j) \in G_f\) do
5. if \(r((i, j)) > 0\) and \(c((i, j)) < 0\) then
6. Send \(r((i, j))\) units of flow along \((i, j)\); update \(f\) and the residual network \(G_f\)
7. end if
8. end for
9. Compute the sets of large surplus and deficit nodes:

   \[
   S(\Delta) = \{ i \in V : \text{imbal}_f(i) \geq \Delta \} \\
   D(\Delta) = \{ i \in V : \text{imbal}_f(i) < -\Delta \}
   \]

10. while \(S(\Delta) \neq \emptyset\) and \(T(\Delta) \neq \emptyset\) do
11. Select a node \(s \in S(\Delta)\) and a node \(v \in D(\Delta)\).
12. Determine shortest path distances \(d\) from node \(s\) to all other nodes in \(G_f(\Delta)\) with respect to the reduced costs \(c^\pi((i, j))\) as arc lengths.
13. Let \(P\) be a shortest path from \(s\) to \(v\) in \(G_f(\Delta)\).
14. Update \(\pi := \pi + d\)
15. Augment \(\Delta\) units of flow along \(P\).
16. Update \(f, G_f(\Delta), S(\Delta)\) and \(D(\Delta)\).
17. end while
18. \(\Delta := \Delta/2\)
19. end while

To specify what we mean precisely with »sufficiently large«, let \(\Delta \geq 1\) be a scaling parameter. We define the two sets

\[
S(\Delta) = \{ i \in V : \text{imbal}_f(i) \geq \Delta \} \\
D(\Delta) = \{ i \in V : \text{imbal}_f(i) < -\Delta \}
\]

of surplus and deficit nodes with large imbalance. In each iteration of the algorithm we augment \(\Delta\) units of flow from a node in \(S(\Delta)\) to a node in \(T(\Delta)\). Clearly, the augmentation must take place along a path of residual capacity at least \(\Delta\). The \(\Delta\)-residual network \(G_f(\Delta)\) contains only those arcs of \(G_f\) which have residual capacity no smaller than \(\Delta\). Since we have assumed that for any pair of nodes there is always a directed path of uncapacitated arcs between them, \(G_f(\Delta)\) will always contain a directed path for any pair of nodes.

The capacity scaling algorithm augments flow along paths in \(G_f(\Delta)\). Recall that in the successive-shortest-path algorithm we augmented flow using paths in \(G_f\). Since \(G_f(\Delta)\) does not contain all arcs of the residual network \(G_f\), a shortest path in \(G_f(\Delta)\) is not necessarily a shortest path in \(G_f\). Thus, we will not be able to use Lemma 3.17 and apply the reduced cost optimality conditions directly to prove correctness of the scaling algorithm.
3.8 The Capacity Scaling Algorithm

The scaling algorithm ensures that every arc in $G_f(\Delta)$ satisfies the reduced cost optimality conditions. This follows by the fact that all arcs on a shortest path in $G_f(\Delta)$ have zero reduced cost. We have seen already that arcs in $G_f$ might have negative reduced cost. The algorithm is described as Algorithm 3.2 in pseudocode.

The algorithm starts with a large value $\Delta := 2^{\lfloor \log_2 \bar{U} \rfloor}$ for the scaling parameter. We will refer to a period where $\Delta$ remains fixed as a $\Delta$-scaling phase. A $\Delta$-scaling phase begins by checking the reduced cost optimality conditions for all arcs in $G_f(\Delta)$. If such an arc has reduced cost $c^\delta((i,j)^\delta) < 0$, then it is removed from $G_f(\Delta)$ by saturating it completely (Step 6 of the algorithm). Then, the algorithm augments flow along shortest paths in $G_f(\Delta)$. Each augmentation starts in a node from $S(\Delta)$, ends in a node from $D(\Delta)$ and carries exactly $\Delta$ units of flow. The scaling phase ends, if either $S(\Delta)$ or $D(\Delta)$ becomes empty.

**Lemma 3.20** At the beginning of a $\Delta$-scaling phase the sum of the surpluses is bounded from above by $2(n + m)\Delta$.

**Proof:** The claim is clearly true for the first $\Delta$-phase with $\Delta := 2^{\lfloor \log_2 \bar{U} \rfloor}$, since $\Delta \geq B$ and $B$ is the maximum requirement for a node.

At the end of a $2\Delta$-scaling phase either the surplus of all nodes is at strictly smaller than $2\Delta$ or the deficit of all nodes is strictly smaller than $2\Delta$. Since the sum of the surpluses equals the sum of the deficits, the total surplus must be bounded from above by $2n\Delta$. Consider the next scaling phase with $\Delta$ as parameter.

At the beginning of the $\Delta$-scaling phase, the algorithm first checks whether every arc $(i,j)^\delta$ in the network $G_f(\Delta)$ has negative reduced cost. Each such arc is eliminated from the residual network by saturating it immediately in Step 6. Notice that each such arc $(i,j)^\delta$ has residual capacity strictly smaller than $2\Delta$ (since all arcs in $G_f(2\Delta)$ had nonnegative reduced costs). Thus, by saturating $(i,j)^\delta$ the imbalance of each of its endpoints changes by at most $2\Delta$. As a consequence, after all such arcs have been saturated, the total sum of the surpluses is bounded by $2n\Delta + 2m\Delta = 2(n + m)\Delta$.  

From Lemma 3.20 it follows that at most $2(n + m)$ augmentations occur within each $\Delta$-scaling phase, since each augmentation carries $\Delta$ units of flow and thus reduces the total excess by $\Delta$.

After $O(\log \bar{U})$ phases, we have $\Delta = 1$. After this phase, by the integrality of the data, every node imbalance must be zero. Moreover, in this phase $G_f(\Delta) = G_f$ and every arc in the residual network satisfies the reduced cost optimality condition. Consequently, the capacity scaling algorithm terminates after $O(\log \bar{U})$ phases with a minimum cost flow.

**Theorem 3.21** The capacity scaling algorithm solves the minimum cost flow problem in time $O((n + m) \log \bar{U} S(n, m, nC))$.

**Proof:** The time bound follows from our considerations above and the fact that in each phase at most $2(n + m) = O(n + m)$ augmentations are carried out, each of which requires a shortest path computation of time $S(n, m, nC)$.  

Theorem 3.21 shows that the scaling technique turns the successive-shortest-path algorithm into a polynomial time algorithm.
3.9 The Successive Scaling Algorithm

For the presentation of the successive scaling algorithm in this section and the scale and shrink algorithm in the following section we will first assume that all capacities in the network are infinite. While this assumption may sound weird at first, we can achieve this property by a suitable network transformation which is depicted in Figure 3.1.

The idea of the transformation is to make the capacity constraint on an arc \((i, j)\) the mass balance constraint for some new nodes. It is straightforward to see that the transformation is indeed valid. The special case of the minimum cost flow problem where all capacities are infinite is usually called the transshipment problem.

Our presentation makes one more assumption about the instance of the transshipment problem to solve. We require the existence of a special node \(r\) such that \(b(r) = 0\), \((i, r) \in A\) with \(c(i, r) = 0\) and \((r, i) \notin A\) for all \(i \in V \setminus \{r\}\). This assumption merely simplifies the presentation of the algorithm, since we can enforce this property by adding a new node \(r\) and the required arcs without changing anything.

The successive scaling algorithm was in fact the first polynomial time algorithm for the minimum cost flow problem. To scale an instance \((G, c, b)\) of the transshipment problem by a factor of \(\Delta > 0\) means the following:

- We replace \(b(i)\) by \(b'(i) = \lfloor b(i)/\Delta \rfloor\) for all \(i \neq r\).
- We set \(b'(r) = -\sum_{i \neq r} b'(i)\).

The latter definition ensures that \(\sum_{i \in V} b'(i) = 0\) so that feasibility of an instance is preserved.

**Lemma 3.22** Let \((G, c, b)\) be an instance of the transshipment problem and \((G, c, b')\) be obtained by scaling with a factor of \(\Delta\). If \((G, c, b)\) has an optimal solution, then \((G, c, b')\) has an optimal solution.

**Proof:** Since we have assumed that costs are nonnegative and costs are not affected by scaling, it suffices to show that \((G, c, b')\) has a feasible solution. By Corollary 2.13 we must show that for every \(S \subseteq V\) with \((V \setminus S, S) = \emptyset\) we have \(\sum_{i \in S} b'(i) \leq 0\).
This condition is trivial if $S = V$. Thus $S \neq V$. The special node $r$ can not be contained in $S$ since $(i, r) \in A$ for all $i \in V$. Thus,
\[
\sum_{i \in S} b'(i) = \sum_{i \in S} \left( \frac{b(i)}{\Delta} \right) \leq \sum_{i \in V} \frac{b(i)}{\Delta} = \Delta \sum_{i \in V} b(i).
\]
Since $(G, c, b)$ had a feasible solution, $\sum_{i \in V} b(i) \leq 0$ and the claim follows. \hfill \qedsymbol

The successive scaling algorithm solves an instance $I = (G, c, b)$ of the transshipment problem by solving a series of scaled instances $I_K, I_{K-1}, \ldots, I_0$. Instance $I_k$ is the instance $I$ scaled by $2^k$. Observe that $I_0 = I$.

Suppose that we are given a pair $(f, \pi)$ of optimal primal and dual solutions for $I_{k+1} = (G, c, b_{k+1})$. Then, $(f, \pi)$ satisfy the reduced cost optimality conditions $c^*(i, j)^b \geq 0$ for all $(i, j)^b \in G_f$. Since all capacities are infinite also $(2f, \pi)$ satisfy the reduced cost optimality conditions. Hence, we can use $(2f, \pi)$ as an initialization for the successive shortest path algorithm to solve $I_k = (G, c, b_k)$ as described in Remark 3.19. The number of iterations of the algorithm is bounded by
\[
\sum_{i \in V} (b_k(i) - e_{2f}(i)) \leq n - 1,
\]
since $e_{2f}(i) = 2b_{k+1}(i)$ and $2b_{k+1}(i) \in \{b_k(i), b_k(i) - 1\}$. Hence, the total time required to solve $I_k$ is $O(nS(m, n, nC))$ provided we are already given an optimal primal-dual pair for $I_{k+1}$.

Consider the instance $I_K$ obtained by scaling with a factor of $K = 2^{\lceil \log_2 B \rceil}$, where $B := \sum_{i, b(i) > 0} b(i)$. Then, all required node balances in $I_K$ are in $\{-1, 0, +1\}$ such that we can start with the zero flow and corresponding optimal zero potential and apply once more the successive shortest path algorithm which will terminate after at most $n - 1$ augmentations. The successive-scaling algorithm is displayed in Algorithm 3.3. Observe that we have to use $B$ in the definition of $K$ since we have assumed the existence of the special node $r$.

**Algorithm 3.3** Successive scaling algorithm for solving the minimum cost flow problem

**SUCCESSIVE-SCALING($G, c, b$)

**Input:** A directed graph $G = (V, A)$ in adjacency list representation; a nonnegative cost function $c : E \rightarrow \mathbb{R}$; required node imbalances $b$.

**Output:** A minimum cost flow $f$.

1. Let $f := 0$ and $\pi := 0$
2. Let $K := 2^{\lceil \log_2 B \rceil}$, where $B := \sum_{i, b(i) > 0} b(i)$.
3. for $k = K, K - 1, \ldots, 0$ do
   1. Let $I_k$ be the instance obtained by scaling $I$ by $2^k$.
   2. Solve $I_k$ by means of the successive shortest path algorithm which is initialized by $(2f, \pi)$ (cf. Remark 3.19).
   3. Update $f$ to be the obtained optimal flow and $\pi$ the corresponding optimal dual solution.
4. end for

**Theorem 3.23** The successive scaling algorithm solves a transshipment instance in time $O(n \log BS(n, m, nC))$, where $B := \sum_{i, b(i) > 0} b(i)$.

**Proof:** We have already seen that the algorithm solves each instance $I_k$ in time $O(nS(n, m, nC))$. Since there are $O(\log B)$ instances, the claimed running time follows. The correctness of the result is a consequence of the fact that $I_0 = I$. \hfill \qedsymbol
3.10 The Scale-and-Shrink Algorithm

Although the running time of the capacity scaling algorithm in Section 3.8 and the successive scaling algorithm in Section 3.9 are polynomial, they are not strongly polynomial. In this section we will present and analyze an algorithm which achieves a strongly polynomial running time. It is based on scaling techniques similar to the capacity scaling algorithm of Section 3.8.

We will present the algorithm, the scale and shrink algorithm under the same assumptions as the successive scaling algorithm, namely, all capacities are infinite and there exists a special node \( r \) such that \( b(r) = 0 \), \((i, r) \in A\) with \( c(i, r) = 0 \) and \((r, i) \notin A\) for all \( i \in V \setminus \{r\} \).

We restate the transshipment problem as a Linear Programming Problem together with its dual, which will turn out to be extremely useful in the analysis later on:

\[
\begin{align*}
\min & \quad \sum_{(i, j) \in A} c(i, j)f(i, j) \\
\sum_{j: (i, j) \in A} f(j, i) - \sum_{j: (i, j) \in A} f(i, j) &= b(i) \quad \text{for all } i \in V \\
f(i, j) &\geq 0 \quad \text{for all } (i, j) \in A
\end{align*}
\]

\[
\begin{align*}
\max & \quad \sum_{i \in V} b(i)\pi(i) \\
\pi(j) - \pi(i) &\leq c(i, j) \quad \text{for all } (i, j) \in A
\end{align*}
\]

We first start with an elementary property that also helps us refresh our knowledge about LP-duality in the case of the minimum cost flow problem.

**Lemma 3.24** Let \((G, c, b)\) be a transshipment problem with \( b = 0 \), that is \( b(i) = 0 \) for all \( i \in V \). Then any feasible solution for the dual Linear Program (3.20) is an optimal solution.

**Proof:** By the nonnegativity of the costs \( c \), the flow \( f \equiv 0 \) is an optimal solution of the instance, i.e., of the Linear Program (3.19). Since the objective function value of the dual (3.20) is zero for any feasible solution, any feasible solution to (3.20) must be also optimal for the dual. \( \square \)

The notion of a tight arc will be essential for the algorithm.

**Definition 3.25 (Tight arc)**

An arc \( a = (i, j) \in A \) is tight if there exists an optimal solution for the transshipment problem with \( f(a) > 0 \).

Suppose that \( a = (p, q) \) is a tight arc for an instance \( I \) of the transshipment problem. The complementary slackness optimality conditions of Theorem 3.14 (iv) (or Theorem 3.10) tell us that \( c^*(p, q) = 0 \) for any optimal solution \( \pi \) for the dual problem. In other words,

\[
\pi(q) - \pi(p) = c(p, q)
\]
for any optimal dual solution. Equation 3.21 tells us that we can remove the variable \( \pi(q) \) from the dual Linear Program by replacing it by \( c(p, q) + \pi(p) \). Let us rewrite 3.21 by marking the spots where \( p \) and \( q \) appear:

\[
\begin{align*}
\text{max} & \quad b(p)\pi(p) + b(q)\pi(q) + \sum_{i \in V \setminus \{p, q\}} b(i)\pi(i) \\
\pi(q) - \pi(i) & \leq c(i, q) \quad \text{for all } (i, q) \in A \quad (3.22a) \\
\pi(i) - \pi(q) & \leq c(q, i) \quad \text{for all } (q, i) \in A \quad (3.22b) \\
\pi(j) - \pi(i) & \leq c(i, j) \quad \text{for all } (i, j) \in A \text{ with } i, j \neq q \quad (3.22c)
\end{align*}
\]

Using \( \pi(q) = c(p, q) + \pi(p) \) reduces 3.22 to the following:

\[
\begin{align*}
\text{max} & \quad (b(p) + b(q))\pi(p) + \sum_{i \in V \setminus \{q, p\}} b(i)\pi(i) \\
\pi(p) - \pi(i) & \leq c(i, q) - c(a) \quad \text{for all } (i, q) \in A \quad (3.23a) \\
\pi(i) - \pi(p) & \leq c(q, i) - c(a) \quad \text{for all } (q, i) \in A \quad (3.23b) \\
\pi(j) - \pi(i) & \leq c(i, j) \quad \text{for all } (i, j) \in A \text{ with } i, j \neq q \quad (3.23c)
\end{align*}
\]

Now, let us have a closer look at 3.23. Consider the graph \( G/a \) obtained from \( G \) by 

\textit{contracting} the arc \((p, q)\), that is, replacing the nodes \( p \) and \( q \) by the node \( p \) and the following operations:

- Each arc \((i, q)\) is replaced by an arc \((i, p)\) with cost \( \bar{c}(i, p) = c(i, q) - c(a) \).
- Each arc \((q, i)\) is replaced by an arc \((p, i)\) with cost \( \bar{c}(p, i) = c(q, i) - c(a) \).
- Each arc \((i, j)\) with \( i, j \neq q \) stays and has cost \( \bar{c}(i, j) = c(i, j) \).
- Node \( p \) has required balance \( \bar{b}(p) = b(p) + b(q) \)
- Each node \( i \neq q \) has balance \( \bar{b}(i) = b(i) \).

Then 3.23 is the dual of the transshipment problem defined by \((G/a, \bar{c}, \bar{b})\). We call this instance \((G/a, \bar{c}, \bar{b})\) the instance obtained from \( I \) by 

\textit{contracting} arc \( a \) and denote it by \( I/a \). (The procedure of contracting an arc is illustrated in Figure 3.2.)

![Figure 3.2: Contraction of an arc.](image)

(a) A directed graph \( G = (V, A) \). The arc \( a = (1, 2) \) to be contracted is shown as a dashed arc.

(b) The resulting graph \( G/a \) after the contraction of \( a = (1, 2) \) in \( G \).
Since any optimal dual solution $\pi$ for $I$ has $\pi(q) = \pi(p) + c(p, q)$, we can retrieve an optimal dual solution for $I$ from an optimal dual solution for $I/a$.

**Lemma 3.26** Let $I = (G, c, b)$ be an instance of the transshipment problem and $a = (p, q)$ be a tight arc. Let $\tilde{\pi}$ be an optimal solution for the instance $I/a$ obtained by contracting $a$. Then an optimal solution $\pi$ for $I$ is given by

$$
\pi(i) = \tilde{\pi}(i) \quad \text{for } i \neq q \\
\pi(q) = \tilde{\pi}(p) + c(p, q).
$$

**Proof:** Immediately from the fact that $\pi(q) = \pi(p) + c(p, q)$ for any optimal dual solution of $I$ (that is of (3.22)) and the relations between the Linear Programs (3.22) and (3.23).

A note of caution: Lemma 3.26 does not tell us that an optimal flow for $I/a$ determines an optimal flow of $I$. It just tells us that we can extend an optimal dual solution for $I/a$ to an optimal dual solution for $I$. However, recall that given an optimal dual solution for $I$ we can find an optimal flow by solving a maximum flow problem as shown on page 39 in Section 3.5.

Contracting an arc preserves feasibility of the problem and the existens of the problem, since there is a correspondence between optimal solutions of the instances $I$ and $I/a$. Observe also that contracting an arc might introduce self loops, which we can safely delete without changing anything.

Assume for the moment that we are given an oracle that helps us identify a tight arc. Then, Lemma 3.26 suggests the following method for computing a minimum cost flow for an instance $I = (G, c, b)$.

If $b = 0$, then the zero flow $f \equiv 0$ is optimal. An optimal dual solution $\pi$ can be obtained as shown in Section 3.5. If $b \neq 0$, then then there must be a tie arc. We call the oracle to identify such a tight arc $a$. Then, we continue with the instance $I/a$. Observe that $I/a$ has one node less than $I$. Again, if all required balances in $I/a$ are zero, then we can find an optimal primal and dual solution easily. Otherwise, we identify again a tight arc and contract it. After at most $n - 1$ contractions we have a network with a single node and the minimum cost flow problem is trivial to solve. By Lemma 3.26 we can reverse the contractions by getting an optimal dual solution for $I$ from an optimal dual solution for $I/a$. As already noted, we then can get an optimal flow from the dual.

The algorithm just described is depicted as Algorithm 3.4 in pseudocode. What remains is to implement the oracle that determines a tight arc.

This algorithm, FIND-TIGHT-ARC is shown in Algorithm 3.5. It uses the concept of a tree solution for the transshipment problem (3.19). A tree solution is a function $f : A \to \mathbb{R}$ satisfying the mass balance constraints (3.19) and such that there exists a tree $T$ in $G = (V, A)$ with $f(a) = 0$ for all $a \notin T$.

**Lemma 3.27** A tree $T$ uniquely determines its tree solution.

**Proof:** See Exercise 3.5.

**Lemma 3.28** If there is a feasible solution to the transshipment problem, then there is also a feasible solution which is a tree solution. If there exists an optimal solution, then there is also an optimal solution which is a tree solution.

**Proof:** Let $f$ be any feasible solution. If $f$ is not a tree solution, then there is a cycle $W$ in $G$ such that each arc of $W$ has nonzero flow.
Algorithm 3.4 Scale-and-Shrink algorithm for solving the minimum cost flow problem

SCALE-AND-SHRINK(G, c, b)

Input: A directed graph G = (V, A) in adjacency list representation; a nonnegative cost function c: E → R; required node imbalances b.

Output: A minimum cost flow f.

1. Set k := 1 and I_1 = (G, c, b)
2. while b ≠ 0 do
3. Call FIND-TIGHT-ARC(I_k) to find a tight arc a of I_k and set I_{k+1} := I_k/a.
4. k := k + 1
5. end while
6. Find a feasible solution π for the dual of I_k.
7. while k > 1 do
8. Extend π to an optimal dual solution of I_{k-1}
9. k := k - 1
10. end while
11. Find a feasible flow f of (G, c, b) such that c^π(a) = 0 whenever f(a) = 0.

If W does not contain a backward arc, we can decrease the flow on all arcs by the same amount without violating the mass balance constraints. Hence, decreasing the flow by the minimum flow value on the arcs of W will reduce the flow of one arc to zero.

If W does contain backward arcs, we can increase the flow on any forward arc of W by a positive amount and simultaneously decrease the flow on any backward arc of W by a positive amount such that the resulting flow still satisfies the mass balance constraints. Increasing/decreasing the flow by the minimum flow on any backward arc will again reduce the flow of one (backward) arc to zero.

By the above procedure we obtain a new flow f’ which is nonzero on at least one more arc than f. After at most m steps we must arrive at the situation that the flow is a tree solution.

Finally suppose that f is an optimal solution. If there is a cycle W such that each arc carries positive flow, then from Theorem 3.14 on page 40 it follows that W must have zero cost. Hence, the procedure given above will not increase the cost of the flow and will terminate with an optimal solution which then is also a tree solution. □

Tree solutions play an important role in the construction of a very popular algorithm for solving the transshipment problem (respective the minimum cost flow problem). The Network Simplex Method maintains feasible tree solutions and looks for special negative cost cycles. We refer to the books [1, 6] for this method.

Algorithm 3.5 Subroutine to find a tight arc.

FIND-TIGHT-ARC(G, c, b)

1. Find a feasible tree solution f for (G, c, b)
2. Scale the instance (G, c, b) by Δ = \frac{1}{n(n-1)} \max\{ f(a) : a ∈ A \} to obtain a new instance (G, c, b') with
   • b’(i) = \frac{b(i)}{Δ} for all i ≠ r
   • b’(r) = - \sum_{i \neq r} b’(i)
3. Find an optimal tree solution f’ for the instance (G, c, b’).
4. Find a ∈ A such that f’(a) ≥ n - 1.
**Lemma 3.29** Let \( a \in A \) be any arc such that in Step 4 of the FIND-TIGHT-ARC algorithm we have \( f'(a) \geq n - 1 \). Then, \( a \) is a tight arc.

**Proof:** Suppose that we solve the instance \((G, c, b)\) by means of the successive shortest path algorithm, initializing the algorithm with \((\Delta f', \pi')\), where \((f', \pi')\) is a pair of optimal primal and dual solutions for \((G, c, b')\) (cf. Remark 3.19). Here, we use \( \Delta f' \) to denote the flow with flow \( \Delta f'(i, j) \) on an arc \((i, j)\).

Observe that \((\Delta f', \pi')\) satisfy the reduced cost optimality conditions for \((G, c, b)\), since the residual network does not change in the transition from \((G, c, b')\) to \((G, c, b)\). Hence, \((\Delta f', \pi')\) are a valid initialization. The algorithm terminates with an optimal flow \( f \) and a corresponding optimal potential \( \pi \).

Since \( f' \) is feasible for \((G, c, b)\), we have \( e_{f'}(i) = b'(i) \) for all \( i \). Thus, we get \( e_{\Delta f'}(i) = \Delta b'(i) \) for all \( i \). Since for \( i \neq r \) we have \( b'(i) = \lfloor b(i) \Delta \rfloor \), it follows that \( e_{\Delta f'}(i) \in (b(i) - \Delta, b(i)) \). Hence, for any node \( i \) with \( b(i) > e_{\Delta f'}(i) \), it follows that \( b(i) - e_{\Delta f'}(i) < \Delta \).

As noted in Remark 3.19, the total value of all augmentations is bounded from above by

\[
\sum_{i \in V} (b(i) - e_{\Delta f'}(i)) < (n - 1) \Delta.
\]

Hence, the value of all augmentations is strictly smaller than \((n - 1) \Delta\). Consequently, since \( \Delta f'(a) \geq \Delta(n - 1) \) by the assumption that \( f'(a) \geq n - 1 \), it follows that after all augmentations \( a \) must still carry positive flow. Hence, we have found an optimal solution where \( a \) has positive flow, in other words, \( a \) is a tight arc. \(\square\)

**Lemma 3.30** FIND-TIGHT-ARC is able to find a tight arc.

**Proof:** In view of Lemma 3.29, it suffices to show that there will always be an arc \( a \) with \( f'(a) \geq n - 1 \). Moreover, it suffices to consider the situation that \( b \neq 0 \), since FIND-TIGHT-ARC is only called in this case.

Let \( a \) be an arc such that \( f(a) = \max\{a' \in A : f(a')\} \). Then \( \Delta = \frac{f(a)}{n(n-1)} \) and \( f(a) > 0 \) since \( b \neq 0 \). We now exploit the fact that \( f \) is a tree solution. Let \( T \) be the corresponding tree. Removing \( a \) from \( T \) cuts the tree into two parts which form a partition \( X \cup Y = V \) of the vertex set of \( G \). Assume without loss of generality that \( a = (x, y) \) with \( x \in X \) and \( y \in Y \). Then, the total net inflow into \( Y \) must equal \( f(x, y) \) and the total net outflow out of \( X \) must equal \( f(x, y) \). Thus, we have \( f(a) = \sum_{i \in Y} b(i) \).

Consider the cut \((X, Y)\). Then, the flow \( \sum_{e \in (X, Y)} f'(e) \) from \( X \) into \( Y \) across the cut is at least \( \sum_{i \in Y} b'(i) \) (since the netflow into \( Y \) must be exactly \( \sum_{i \in Y} b'(i) \)) that is,

\[
\sum_{i \in Y} b'(i) \leq \sum_{e \in (X, Y)} f'(e).
\] (3.24)

Since \( f' \) is a tree solution, there is one arc \( a' \) in \((X, Y)\) with \( f(a') \geq \frac{1}{n-1} \sum_{e \in (X, Y)} f'(e) \).
Thus, this arc a' satisfies:

\[ f'(a') \geq \frac{1}{n-1} \sum_{e \in (X,Y)} f'(e) \]
\[ \geq \frac{1}{n-1} \sum_{i \in Y} b'(i) \]
\[ \geq \frac{1}{n-1} \sum_{i \in Y} \left( \frac{b(i)}{\Delta} - 1 \right) \]
\[ \geq \frac{1}{n-1} \left( \frac{n(n-1)}{f(a)} \sum_{i \in Y} b(i) - |Y| \right) \]
\[ = n \frac{|Y|}{n-1} - 1 \]
\[ \geq n - 1. \]

Hence from Lemma 3.29 it follows that a' is a tight arc.

We have shown the correctness of SCALE-AND-SHRINK. We complete our analysis with the running time analysis:

**Theorem 3.31** Algorithm SCALE-AND-SHRINK finds an optimal solution for the transshipment problem in time \( O(n^2 \log nS(n, m, nC)) \).

**Proof:** We first analyze the running time of the FIND-TIGHT-ARC routine which is called at most \( n \) times by the algorithm. The first time FIND-TIGHT-ARC is called, we compute a feasible tree solution as follows:

- We compute a feasible flow by one maximum flow computation as shown in Section 2.6. This can be accomplished in \( O(n^3) \) time by the FIFO-preflow-push algorithm of Section 2.6.4.
- We then convert the solution into a tree solution as in the proof of Lemma 3.28. This needs time \( O(n^2) \).

In all subsequent calls to FIND-TIGHT-ARC we are faced with the situation that we already have a tree solution \( f \) for \( I \), but need a tree solution for \( I/a \). If \( a \) is not in the tree, then \( f \) is clearly again a feasible tree solution for \( I \). Otherwise, the contraction of \( a \) causes a cycle \( W \) in the flow pattern of \( f \). However, by sending flow along \( W \) (as in the proof of Lemma 3.28) we can reduce the flow on at least one arc of \( W \) to zero. This procedure needs \( O(n^2) \) time. We conclude that finding a feasible tree solution needs a total of \( O(n^3) \) time for all calls to FIND-TIGHT-ARC.

The only other major effort in FIND-TIGHT-ARC is to solve the transshipment instance \((G, c, b')\) in Step 6. We can use the successive scaling algorithm of Section 3.9 for this purpose. Its running time on \((G, c, b')\) is \( O(n \log BS(n, m, nC)) \), where \( B := \sum_{i: b'(i) > 0} b'(i) \).

We bound \( b'(i) \) for all \( i \) with \( b'(i) > 0 \) in an appropriate way. First consider a node \( i \neq r \). Since \( f \) is a tree solution, there are at most \( n-1 \) arcs where \( f \) is nonzero. In particular, at most \( n-1 \) arcs \((j, i)\) can have nonzero flow. Since \( b(i) = \sum_{j: (j, i) \in A} f(j, i) \), we conclude that there is an arc \((j_0, i)\) with \( f(j_0, i) \geq b(i)/(n-1) \). Hence the scaling parameter \( \Delta \) satisfies \( \Delta \geq \frac{b(i)}{n(n-1)^2} \) and

\[ b'(i) \leq \frac{b(i)}{n(n-1)^2} = n(n-1)^2. \]
The special node \( r \) has required balance \( b'(r) \leq n - 1 \). Hence, \( B = \sum_{i : b'(i) > 0} b'(i) \leq n \cdot n(n - 1)^2 = \mathcal{O}(n^4) \). This yields \( \mathcal{O}(n \log B S(n, m, nC)) = \mathcal{O}(n \log n S(n, m, nC)) \) for solving the instance \((G, c, b')\).

In summary, the running time for a single call to FIND-TIGHT-ARC is in \( \mathcal{O}(n \log S(n, m, nC)) \) and there will be an additional effort of \( \mathcal{O}(n^3) \) for the first call. These running times dominate all other steps in the main part of the algorithm. Since FIND-TIGHT-ARC is called \( n \) times, the claimed running time follows.

### 3.11 Exercises

**Exercise 3.1**
Show that the minimum cost flow problem contains the shortest path problem as a special case.

**Exercise 3.2**
Let \( A \) be a matrix with integer entries and linearly independent rows. Prove that the following three conditions are equivalent:

(i) \( A \) is unimodular.

(ii) Every basic feasible solution of \( \{ x : Ax = b, x \geq 0 \} \) is integral for every integer vector \( b \).

(iii) Every basis matrix \( B \) of \( A \) has an integer inverse \( B^{-1} \).

**Exercise 3.3**
Let \( M \) be any \( m \times n \) matrix with entries taken from \( \{0, 1\} \) with the property that any column contains at most one \( +1 \) and at most one \( -1 \). Prove that \( M \) is totally unimodular. Show that as a corollary, the node-arc incidence matrix of a network is totally unimodular.

**Exercise 3.4**
Let \( M \) be any \( m \times n \)-matrix with entries \( \{0, 1\} \) and the property that the rows of \( M \) can be permutated in such a way that all \( 1 \)s appear consecutively. Prove that \( M \) is totally unimodular.

**Exercise 3.5**
Prove that a tree \( T \) uniquely determines its tree solution.

**Exercise 3.6 (Negative costs)**
Why can we assume that we have given only non-negative costs for the min cost flow problem. What happens if costs are negative?

**Exercise 3.7 (Budget problem)**
Consider the following problem: There is given a budget and a network with arc costs and capacities. The objective is to find a maximum flow with costs that are not exceeding the budget.

Give an algorithm for that problem. Is it polynomial?
Dynamic Network Flows

4.1 Basic Definitions

We consider again a network $G = (V, A)$ with capacities $u$ and costs $c$ on the arcs as in the minimum cost flow problem. Additionally, the arcs $(i, j) \in A$ are assigned transit times (or traversal times) $\tau(i, j) \in \mathbb{N}$. At each time at most $u(i, j)$ units of flow can be sent over $(i, j)$ and it takes $\tau(i, j)$ time to traverse the arc $(i, j)$.

**Definition 4.1 (Dynamic Flow)**

A dynamic flow in $G = (V, A)$ is a function $f : A \times \mathbb{R} \to \mathbb{R}_0^+$ satisfying the capacity constraints

$$f(a, \theta) \leq u(a) \quad \text{for all } a \in A \text{ and all } \theta \in \mathbb{R}. \quad (4.1)$$

We interpret $f(a, \theta)$ to be the flow rate over arc $a$ at time $\theta$. The cost of a dynamic flow is defined as

$$c(f) = \sum_{a \in A} c(a) \int_{-\infty}^{+\infty} f(a, \theta)d\theta. \quad (4.2)$$

Let $\delta^+(i) := \{(j, i) : (j, i) \in A\}$ and $\delta^-(i) := \{(i, j) : (i, j) \in A\}$ denote the set of arcs emanating and ending in $i$, respectively. We say that a dynamic flow satisfies the flow conservation constraints in $i \in V$ at time $\theta$ if

$$\sum_{a \in \delta^+(i)} \int_{-\infty}^{\theta} f(a, z - \tau(a))dz \geq \sum_{a \in \delta^+(i)} \int_{-\infty}^{\theta} f(a, z)dz. \quad (4.3)$$

Constraint (4.3) means that at any moment $\theta$ in time, the amount of flow that has left a node $i$ (right hand side of the inequality) is at most the flow that has reached $i$ until time $\theta$ (left hand side of the inequality). Observe that we allow flow to »wait« at vertex $i$ until there is capacity enough to be transmitted further.

The dynamic flow satisfies the strict flow conservation constraints in $i \in V$ at time $\theta$ if we have equality in (4.3), that is,

$$\sum_{a \in \delta^+(i)} \int_{-\infty}^{\theta} f(a, z - \tau(a))dz = \sum_{a \in \delta^+(i)} \int_{-\infty}^{\theta} f(a, z)dz. \quad (4.4)$$

A dynamic flow with time horizon $T \geq 0$ is a dynamic flow $f$ such that

$$f(a, \theta) = 0 \quad \text{for all } a \in A \text{ and all } \theta \notin [0, T - \tau(a)]. \quad (4.5)$$

Condition (4.5) means that no flow is sent before time 0 or after time $T$. 
Definition 4.2 (Dynamic \((s, t)\)-flow with and without waiting)
Let \(s, t \in V\) be two nodes. A dynamic \((s, t)\)-flow with waiting is a dynamic \(f\-bw\) which satisfies the \(f\-bw\) conservation constraints in all nodes \(i \in V \setminus \{s\}\) at all times. A dynamic \((s, t)\)-flow without waiting must additionally satisfy all strict \(f\-bw\) conservation constraints in all nodes \(i \in V \setminus \{s, t\}\) at all times.

A dynamic \((s, t)\)-\(f\-bw\) with time horizon \(T\) must satisfy the strict \(f\-bw\) conservation constraints in all nodes \(i \in V \setminus \{s, t\}\) at time \(T\).

The value of a dynamic \((s, t)\)-flow is

\[
\text{val}(f) := \sum_{a \in \delta^+(s)} \int_0^T f(a, \theta) d\theta - \sum_{a \in \delta^-(s)} \int_0^T f(a, \theta) d\theta, \tag{4.6}
\]

which is the amount of flow that has left the source node \(s\) by time \(T\). Let \(i \in V\) and write as a shorthand

\[
\bar{e}_f(i) := \sum_{a \in \delta^+(i)} \int_0^T f(a, \theta) d\theta - \sum_{a \in \delta^-(i)} \int_0^T f(a, \theta) d\theta.
\]

Since \(f\) satisfies the strict flow conservation constraints in \(i \in V \setminus \{s, t\}\) at time \(T\) and \(f\) has time horizon \(T\) we have for \(i \in V \setminus \{s, t\}\)

\[
0 = \sum_{a \in \delta^+(i)} \int_0^T f(a, z) dz - \sum_{a \in \delta^-(i)} \int_0^T f(a, z - \tau(a)) dz \tag{4.7a}
\]

\[
= \sum_{a \in \delta^+(i)} \int_0^T f(a, z) dz - \sum_{a \in \delta^-(i)} \int_0^T f(a, z) dz \tag{4.7b}
\]

\[
= \bar{e}_f(i). \tag{4.7c}
\]

where we have used the fact that \(f(a, z) = 0\) for \(z > T - \tau(a)\) and \(z < 0\). Thus,

\[
\bar{e}_f(s) + \bar{e}_f(t) = \sum_{i \in V} \bar{e}_f(i)
\]

\[
= \sum_{i \in V} \left( \sum_{a \in \delta^+(i)} \int_0^T f(a, x) dx - \sum_{a \in \delta^-(i)} \int_0^T f(a, x) dx \right)
\]

\[
= 0.
\]

The last inequality stems from the fact that each term \(\int_0^T f(a, x) dx\) appears twice, once positive and once negative. Hence, we get that

\[
\text{val}(f) = \sum_{a \in \delta^+(s)} \int_0^T f(a, \theta) d\theta - \sum_{a \in \delta^-(s)} \int_0^T f(a, \theta) d\theta
\]

\[
= \sum_{a \in \delta^-(t)} \int_0^T f(a, \theta) d\theta - \sum_{a \in \delta^+(t)} \int_0^T f(a, \theta) d\theta,
\]

which is our intuition: the amount of flow that has left \(s\) equals the amount of flow that has reached \(t\).

The maximum dynamic \((s, t)\)-flow problem with time horizon \(T\) (or shorter the maximum dynamic flow problem) is to find a dynamic \((s, t)\)-flow with time horizon \(T\) of maximum value.
4.2 Dynamic Flows and Cuts over Time

As in the case of static flows cuts play an important role. The concept of a cut, however, has to be refined in order to reflect the dynamic behavior of the flow.

Definition 4.3 (Dynamic cut with time horizon $T$)
A dynamic cut with time horizon $T$ is given by a function $X : [0, T) \rightarrow 2^V$ such that:

(i) $s \in X(\theta) \subseteq V \setminus \{t\}$ for all $\theta \in [0, T)$;
   (this just means that $(X(\theta), V \setminus X(\theta))$ is an $(s, t)$-cut for all $\theta \in [0, T)$)

(ii) $X(\theta_1) \subseteq X(\theta_2)$ for $\theta_1 \leq \theta_2$;

Let $X$ be a dynamic cut with time horizon $T$. For a node $i \in V$ we define $\alpha_i \in [0, T]$ as follows:

$$\alpha_i := \min \{ \theta : i \in X(\theta') \text{ for all } \theta' \geq \theta \}$$

(4.8)

if the set in (4.8) is nonempty and $\alpha_i := T$ otherwise. Clearly $\alpha_s = 0$ and $\alpha_t = T$.

Let $X$ be a dynamic $(s, t)$-cut in $G$. In a dynamic $(s, t)$-flow, every unit of flow must cross $X$ on some arc in the cut $X$ at some time. Consider such an arc $a = (i, j)$. In order to cross the cut on $a$ from left to right, flow must leave node $i$ after time $\alpha_i$ and arrive at $j$ before time $\alpha_j$, that is, it must leave $i$ before time $\alpha_j - \tau(i, j)$. Hence, we can say that arc $(i, j)$ is in the cut $X$ during the time interval $[\alpha_i, \alpha_j - \tau(i, j))$.

Definition 4.4 (Capacity of a dynamic cut)
An arc $a = (i, j)$ is in the dynamic cut $X$ during the time interval $[\alpha_i, \alpha_j - \tau(a))$. The capacity of the dynamic cut equals the amount of flow which could be sent over the arcs while they are in the cut, that is,

$$\sum_{(i,j) \in A} \int_{\alpha_i}^{\alpha_j - \tau(i, j)} u(i, j) d\theta = \sum_{(i,j) \in A} \max \{0, \alpha_j - \tau(i, j) - \alpha_i\} \cdot u(i, j).$$

(4.9)

In the following sections we will establish an analogon to the famous Max-Flow-Min-Cut-Theorem (see Theorem 2.9 on page 14) for static flows.

Lemma 4.5 Let $f$ be a dynamic $(s, t)$-flow with time horizon $T$ and let $X$ be a dynamic cut with the same time horizon $T$ in the graph $G = (V, A)$. Then the value of $f$ is at most the capacity of the cut $X$.

Proof: Let $i \in V \setminus \{s, t\}$. Any dynamic $(s, t)$-flow must satisfy the flow conservation constraints at $i$ at all times. Hence, in particular we have

$$\sum_{a \in \delta^+ (i)} \int_0^{\alpha_i} f(a, \theta) d\theta - \sum_{a \in \delta^- (i)} \int_0^{\alpha_i - \tau(a)} f(a, \theta) d\theta \leq 0.$$  (4.10)

Moreover, if $f$ satisfies the strict flow conservation then we have an actual equality in (4.10). Moreover, if $i = t$ we have $\alpha_i = T$ and hence

$$\sum_{a \in \delta^+ (t)} \int_0^T f(a, \theta) d\theta - \sum_{a \in \delta^- (t)} \int_{\alpha_i - \tau(a)}^T f(a, \theta) d\theta = 0.$$  (4.11)
We can now use these results to bound the value of the flow:

\[
\text{val}(f) = \sum_{a \in \delta^+(s)} \int_0^T f(a, \theta) d\theta - \sum_{a \in \delta^-(s)} \int_0^T f(a, \theta) d\theta
\]

\[
\leq \sum_{i \in V \setminus \{s, t\}} \left( \sum_{a \in \delta^+(i)} \int_{\alpha_i}^{\alpha_i - \tau(a)} f(a, \theta) d\theta - \sum_{a \in \delta^-(i)} \int_{\alpha_i - \tau(a)}^{\alpha_i} f(a, \theta) d\theta \right)
\geq 0 \text{ for } i \neq s, t \text{ by \ref{eq:flow}} \text{ and } = 0 \text{ for } i = t \text{ by \ref{eq:flow}}.
\]

\[
= \sum_{a \in A} \int_{\alpha_i}^{\alpha_j - \tau(a)} f(a, \theta) d\theta
\]

\[
\leq \sum_{a \in A} \int_{\alpha_i}^{\alpha_j - \tau(a)} u(a) d\theta
\]

\[
= \sum_{(i, j) \in A} \max\{0, \alpha_j - \tau(i, j) - \alpha_i\} \cdot u(i, j).
\]

This proves the claim. \(\square\)

Observe that inequality \((*)\) is in fact an equality if \(f\) satisfies the strict flow conservation constraints. We will make use of this fact later on.

### 4.3 Temporally Repeated Flows

An important concept in particular for the computation of maximal dynamic flows is the notion of a temporally repeated flow. Let \(f : A \to \mathbb{R}_{\geq 0}\) be a (static) \((s, t)\)-flow in \(G\). As shown in Section \ref{sec:static}, we can decompose \(f\) into path flows. To avoid confusions, we will denote by \(\omega : P \to \mathbb{R}_{\geq 0}\) the corresponding path flow decomposition, where \(P\) is the set of all \((s, t)\)-paths in \(G\) and \(\omega(P) \geq 0\) is the flow on path \(P\).

For a path \(P \in \mathcal{P}\) denote by \(\tau(P) := \sum_{a \in P} \tau(P)\) its traversal time.

**Definition 4.6 (Temporally repeated dynamic flow)**

For a flow \(f\) with path decomposition \(\omega\) the **temporally repeated dynamic flow with time horizon** \(T\) induced by \(f\) is denoted by \(f^T\) and defined as follows: We start at time \(0\) to send \(\omega(P)\) units of flow along path \(P\) and continue to do so until the latest moment in time \(T - \tau(P)\) where flow can still reach the sink.

We will illustrate temporally repeated dynamic flows with an example. Consider the network given in Figure \ref{fig:temp_repeated}.

**Figure 4.1: Example of a temporally repeated dynamic flow**

Suppose that \(f^T\) is the temporally repeated dynamic flow induced by \(f\). Let \(a \in A\) be arbitrary. At any moment in time \(\theta\), the amount of flow sent by \(f^T\) over \(a\) is bounded from above as follows:

\[
f^T(a, \theta) \leq \sum_{P \in \mathcal{P}, a \in P} \omega(P) = f(a) \leq u(a).
\]
4.3 Temporally Repeated Flows

Hence, \( f^T \) satisfies the capacity constraints for dynamic flows. Moreover, by construction \( f^T \) also satisfies the strict flow conservation constraints for any node in \( V \setminus \{s, t\} \) and any moment in time (recall that at any moment in time \( f^T \) is the sum of static flows). Hence, \( f^T \) is in fact a dynamic flow with time horizon \( T \).

What about the value of \( f^T \)? Let \( P \) be a path in the path decomposition. We sent flow over \( P \) from time 0 until time \( T - \tau(P) \). Thus, the amount of flow sent over \( P \) equals \( \omega(P)(T - \tau(P)) \). Summing over all paths in the decomposition yields:

\[
\text{val}(f^T) = \sum_{P \in P} \omega(P)(T - \tau(P))
\]

\[
= T \sum_{P \in P} \omega(P) - \sum_{P \in P} \sum_{a \in P} \tau(a)
\]

\[
= T \text{val}(f) - \sum_{a \in A} \tau(a) \sum_{P \in P: a \in P} \omega(P)
\]

\[
= T \text{val}(f) - \sum_{a \in A} \tau(a) f(a).
\]

(4.12)

Equation (4.12) has an important consequence: the value \( f^T \) is independent of the paths used in the flow decomposition. We summarize our results for later reference:

**Lemma 4.7** Let \( f^T \) be a temporally repeated dynamic flow induced by the flow \( f \). Then, \( f^T \) satisfies the strict flow conservation constraints at any node in \( V \setminus \{s, t\} \) at any moment in time and \( f^T \) has value:

\[
\text{val}(f^T) = T \text{val}(f) - \sum_{a \in A} \tau(a) f(a).
\]

(4.13)

Lemma 4.7 allows us to reduce the problem of finding a temporally repeated dynamic flow of maximum value to a static flow problem: We need to find a flow \( f \) in \( G \) which maximizes \( T \text{val}(f) - \sum_{a \in A} \tau(a) f(a) \), or equivalently, which minimizes \( \sum_{a \in A} \tau(a) f(a) - T \text{val}(f) \). This problem can be formulated as follows:

\[
\min \sum_{a \in A} \tau(a) f(a) - \sum_{i \in (s, i)} \Delta f(s, i)
\]

(4.14a)

\[
\sum_{j \in (j, i) \in A} f(j, i) - \sum_{j \in (i, j) \in A} f(i, j) = 0 \quad \text{for all } i \in V \setminus \{s, t\}
\]

(4.14b)

\[
0 \leq f(i, j) \leq u(i, j) \quad \text{for all } (i, j) \in A
\]

(4.14c)

Conditions (4.14b) and (4.14c) state that any feasible solution must be an \((s, t)\)-flow. The problem (4.14) is equivalent to a minimum cost flow problem with cost \( c(i, s) = \tau(i, s) - T \) for all arcs \((i, s) \in A\) and \( c(i, j) = \tau(i, j) \) for all other arcs. Observe that in this minimum cost flow problem the cost of some of the arcs might be negative. Still, the methods of Chapter 3 can be extended to solve such a minimum cost flow problem. We invite the reader to go through the material in Chapter 3 and check that, for instance, the successive scaling algorithm still works in this setting and provides the same time bounds.

Alternatively, we can reformulate the problem of finding a flow which minimizes \( \sum_{a \in A} \tau(a) f(a) - T \text{val}(f) \) as a minimum cost circulation problem in an augmented network: add the arc \((t, s)\) with cost \( -T \) and infinite capacity. All other arcs have capacities given by \( u \) and costs given by \( \tau \). We will adopt this second view of the problem in the sequel.
4.4 The Dynamic Max-Flow-Min-Cut-Theorem

We assume without loss of generality that each node in \( G \) lies on an \((s, t)\)-path of length (with respect to the transit times \( \tau \)) at most \( T \). Nodes which do not satisfy this condition are useless, since flow can never pass through them in a dynamic \((s, t)\)-flow with time horizon \( T \).

We denote by \( \bar{G} \) the augmented network constructed in the previous section which allowed us to solve the maximum repeated dynamic flow problem to a minimum cost circulation problem.

For a solution \( x \) to the minimum cost circulation problem in \( \bar{G} \) we can consider the residual network \( \bar{G}_x \). If \( x \) is of minimum cost, then \( \bar{G}_x \) can not contain negative length cycles, where the length of \((i, j)\) in \( \bar{G}_x \) is defined via the cost \( \tau \) as in Definition 3.7. Hence, the shortest path distances \( d_x(i, j) \) between nodes in \( \bar{G}_x \) are well defined.

Lemma 4.8 Let \( x \) be a minimum cost circulation in \( \bar{G} \). The shortest path distances in \( \bar{G} \) satisfy the following conditions:

(i) \( d_x(s, t) \geq T \)
(ii) \( 0 \leq d_x(s, i) \) for all \( i \in V \).
(iii) \( d_x(s, i) \leq d_x(s, j) - \tau(i, j) \) for all \( (i, j) \in A \) with \( x(i, j) > 0 \).

Proof:

(i) Since \( \bar{G} \) does not contain a negative length cycle and \((t, s)\) has cost \(-T\), the distance \( d_x(s, t) \) can not be shorter than \( T \) (otherwise the path of length smaller than \( T \) and the arc \((t, s)\) would form a cycle of negative length).

(ii)

(iii) If \( x(i, j) > 0 \), then \( \bar{G}_x \) contains an arc \((j, i)\) of cost \(-\tau(i, j)\). A shortest path from \( s \) to \( j \) can thus be extended by this arc to get a path from \( s \) to \( i \) of length \( d_x(s, j) - \tau(i, j) \). The shortest \((s, i)\) path can only have smaller length.

\[ \Box \]

Theorem 4.9 Let \( f^T \) be a maximum repeated dynamic fbw with time horizon \( T \). Then \( f^T \) is also a maximum dynamic fbw with time horizon \( T \).

Proof: Let \( f \) be a static \((s, t)\)-flow maximizing \( T \text{val}(f) = \sum_{a \in A} \tau(a) f(a) \) which induces \( f^T \) and let \( x \) denote the corresponding minimum cost circulation in \( \bar{G} \). Denote again by \( d_x(s, v) \) the shortest path distances in the residual network \( \bar{G}_x \).

For \( i \in V \) let \( \alpha_i := d_x(s, i) \). Clearly, \( \alpha_s = 0 \) and by Lemma 4.8(i) \( \alpha_t \geq T \). Moreover, by Lemma 4.8 we have \( \alpha_i \geq 0 \) for all \( i \in V \). We can define a dynamic \((s, t)\)-cut \( X \) with time horizon with the help of these threshold values:

\[ X(\theta) := \{ i \in V : \alpha_i \leq \theta \} \quad \text{for} \quad 0 \leq \theta < T. \]

By \( \alpha_s = 0 \) and \( \alpha_t \geq T \) the set \( X(\theta) \) will in fact induce an \((s, t)\)-cut for all \( 0 \leq \theta \leq T \) and thus \( X \) is in fact a dynamic cut.

We compare the value \( \text{val}(f^T) \) to the capacity of the dynamic cut \( X \). The intuition is to achieve equality in each inequality used in the proof of the upper bound in Lemma 4.3.
4.4 The Dynamic Max-Flow-Min-Cut-Theorem

Claim 4.10 For \( f^T \) and the constructed cut \( X \) we have: \( f^T(a, \theta) = u(a) \) for all \( a = (i, j) \) and \( \theta \in [\alpha_i, \alpha_j - \tau(a)) \).

We first show how Claim 4.10 implies the claim of the theorem. As in Lemma 4.5 but now using the fact that \( f \) satisfies the strict flow conservation constraints, we have:

\[
\text{val}(f) = \sum_{a \in \delta^+(s)} \int_0^T f(a, \theta)d\theta - \sum_{a \in \delta^-(s)} \int_0^T f(a, \theta)d\theta \\
\leq \sum_{x \in V} \left( \sum_{a \in \delta^+(s)} \int_{\alpha_i}^{\alpha_j - \tau(a)} f(a, \theta)d\theta - \sum_{a \in \delta^-(s)} \int_{\alpha_i}^{\alpha_j - \tau(a)} f(a, \theta)d\theta \right) \\
= \sum_{a \in A} \int_{\alpha_i}^{\alpha_j - \tau(a)} f(a, \theta)d\theta \\
\leq \sum_{a \in A} \int_{\alpha_i}^{\alpha_j - \tau(a)} u(a)d\theta \\
= \sum_{(a, j) \in A} \max\{0, \alpha_j - \tau(i, j) - \alpha_i\} \cdot u(i, j).
\]

The last equality comes from Claim 4.10 and Lemma 4.8(iii).

We complete the proof by establishing Claim 4.10. Let \( a = (i, j) \). If \( x(a) < u(a) \), then \( (i, j)^+ \in \delta_x \).

Then,

\[
\alpha_j = d_x(s, j) \leq d_x(s, i) + \tau(i, j) = \alpha_i + \tau(a).
\]

In this case the interval \([\alpha_i, \alpha_j - \tau(a))\) is empty and there is nothing to show since the claim is trivial. Hence, we can assume for the remainder of the proof that \( x(a) = u(a) \). Consider a flow decomposition \( \omega \) of the \((s, t)\)-flow \( f \) (which is the same as the circulation \( x \) with the only exception of the value on the artificial arc \((t, s)) \).

We know that \( f(a) = u(a) \). If we can show that each path \( P \) in the decomposition containing \( a \) induces an \((s, i)\)-path of length at most \( \alpha_i \) and a \((j, t)\)-path of length of length at most \( T - \alpha_j \), then by construction of \( f^T \), there will be flow \( \omega(P) \) on \( a \) at each moment in time in the interval \([\alpha_i, \alpha_j - \tau(a))\). Since the sum of the flows on the paths equals \( f(a) = u(a) \), this establishes the claim.

Let \( P = (s = v_1, \ldots, v_k = i, v_{k+1} = j, \ldots, v_z = t) \) be such a path in the decomposition and \( P' = (s = v_1, \ldots, v_k = i, v_{k+1} = j, \ldots, v_z = t) \) will be contained in \( \delta_x \). We have \( \tau(P') = -\tau_x(-P') \), where \( \tau_x \) denotes the lengths in \( \delta_x \). We have

\[
\tau(P') = -\tau_x(-P') \leq -(d_x(s, s) - d_x(s, i)) = d_x(s, i) = \alpha_i.
\]

The fact that \((v_{k+1} = j, \ldots, v_z = t)\) has length at most \( T - \alpha_j \) is along the same lines. This establishes Claim 4.10 and completes the proof of the theorem.

Notice that as a byproduct of the proof of Theorem 4.9 we have shown that the maximum dynamic flow value equals the minimum dynamic cut capacity.

Theorem 4.11 (Dynamic Max-Flow-Min-Cut-Theorem) The maximum value of a dynamic \((s, t)\)-flow with time horizon \( T \) equals the minimum capacity of a dynamic \((s, t)\)-cut with time horizon \( T \).
4.5 The Discrete Time Model and Time-Expanded Networks

So far we have considered the flow on an arc as a continuous function \( f(\cdot, \cdot) : [0, T) \to \mathbb{R}_{\geq 0} \). For many applications also the discrete variant is of interest. In this variant \( f(\cdot, \cdot) : \{0, \ldots, T-1\} \to \mathbb{R}_{\geq 0} \), that is, flow can only be sent at integral moments \( \theta \in \{0, \ldots, T-1\} \) in time over an arc \( a \) and it reaches the end of the arc at time \( \theta + \tau(a) \).

**Definition 4.12 (Discrete time dynamic flow with time horizon \( T \))**

A discrete time dynamic flow with time horizon \( T \) is a function \( f(\cdot, \cdot) : \{0, \ldots, T\} \to \mathbb{R}_{\geq 0} \) satisfying the capacity constraints

\[
(f(a, \theta) \leq u(a)) \quad \text{for all } a \in A \text{ and all } \theta \in \{0, \ldots, T-1\}.
\]  

(4.16)

The following definitions are easy carryovers from the continuous case. The cost of a discrete dynamic flow is defined as

\[
c(f) = \sum_{a \in A} c(a) \sum_{\theta=0}^{T-1} f(a, \theta).
\]

The (strict) flow conservation constraints at \( i \in V \) at time \( \theta \in \{0, \ldots, T-1\} \) are:

\[
\sum_{a \in \delta^-(i)} \sum_{z=0}^{T-\tau(a)} f(a, z) = \sum_{a \in \delta^+(i)} \sum_{z=0}^{T-1} f(a, z).
\]  

(4.17)

The value of a discrete dynamic flow is

\[
\text{val}(f) := \sum_{a \in \delta^+(s)} \sum_{\theta=0}^{T-1} f(a, \theta) - \sum_{a \in \delta^-(t)} \sum_{\theta=0}^{T-1} f(a, \theta).
\]

There is one major new aspect of discrete dynamic flows compared to the continuous versions. The discrete maximum dynamic flow problem can be reduced to a maximum flow problem in a *time-expanded network* \( G^T = (V^T, A^T) \). The idea is to have a copy of each node in \( G \) at each moment in time. More precisely,

\[
V^T := \{ i_\theta : i \in V, \theta = 0, \ldots, T-1 \}
\]

\[
E^T := \{ (i_\theta, j_{\theta+\tau(i,j)}), (i, j) \in V, \theta = 0, \ldots, T-1 - \tau(i,j) \}
\]

If we allow waiting, then \( G^T \) also contains the waiting arcs \( (i_\theta, i_{\theta+1}) \). The capacities and costs for the arcs are derived from the original network \( G \) in the obvious way. The construction of the time-expanded network is illustrated in Figure 4.2.

We can add an artificial source \( s^* \) and an artificial sink \( t^* \) to \( G^T \). For each \( i \in V \) and all \( \theta = 0, \ldots, T-1 \), we add the arcs \( (s^*, i_\theta) \) and \( (i_\theta, t^*) \) of infinite capacity. Call the augmented time expanded network \( G^T \). By construction the discrete maximum dynamic flow problem in \( G \) is equivalent to finding a maximum flow from \( s^* \) to \( t^* \) in the time-expanded network \( G^T \). Since \( G^T \) has \( \Theta(nT) \) nodes, the maximum dynamic flow problem can be solved in time \( O(n^3T^3) \) for instance by means of the FIFO-preflow-push algorithm of Section 2.8.4.

However, the time bound \( O(n^3T^3) \) is not polynomial! Since we time horizon \( T \) will be encoded in binary in an instance of the maximum dynamic flow problem, it uses only \( \Theta(\log T) \) bits.
Figure 4.2: Network $G$ and time-expanded version $G^T$ of $G$. The dashed arcs are only included if we allow waiting. Capacities are not shown in the networks.
Theorem 4.13 Let $G = (V, A)$ be a network with capacities, costs and transit times, and let $T \in \mathbb{N}$. Every (continuous) dynamic flow with time horizon $T$ corresponds to an equivalent discrete dynamic flow with time horizon $T$ and vice versa. In particular, the costs are the same and the waiting/non-waiting property also carries over.

Proof: Let $f: A \times [0, T) \to \mathbb{R}_{\geq 0}$ be a dynamic flow. We define a discrete flow $g: A \times \{0, \ldots, T - 1\} \to \mathbb{R}_{\geq 0}$ as follows:

$$g(a, \theta) := \int_{\theta}^{\theta + 1} f(a, x) \, dx \quad \text{for } a \in A \text{ and } \theta = 0, \ldots, T - 1.$$ 

Conversely, given $g: A \times \{0, \ldots, T - 1\} \to \mathbb{R}_{\geq 0}$, we can define a continuous discrete flow by

$$f(a, x) := g(a, \theta) \quad \text{for } a \in A \text{ and } x \in [\theta, \theta + 1).$$ 

It is straightforward to verify that the claimed properties are satisfied. 

Theorem 4.13 allows us to use the methods of Section 4.3 to compute maximum dynamic flows also in the discrete case without using the time expanded network. We remark, however, that time-expanded networks can be used in a variety of contexts to model other dynamic network flow problems where either a »compact« problem formulation is not known or provably not available.

4.6 Exercises

Exercise 4.1
Let $f$ be a dynamic $(s, t)$-flow with time horizon $T$ and let $X$ be a dynamic cut with the same time horizon $T$ in the graph $G = (V, A)$. Show that the value of $f$ is at most the capacity of the cut $X$.

Exercise 4.2

Consider the following network $G$. The arc labels indicate transit times and capacities, i.e., the label at arc $e$ is $(\tau_e, u_e)$.

(a) Use the augmenting path algorithm of Ford and Fulkerson in order to determine a maximal $s$-$t$-flow over time with time horizon $T = 18$ in $G$.

(b) Instead of applying the algorithm of Ford and Fulkerson, one can also use the time-expanded network $G_{18}$ and compute a maximal $s_0$-$t_{T-1}$-flow there. Count the number of nodes and arcs in $G_{18}$. 

Diagram representation of the network: 

- Source node $s$ with label $(5, 25)$
- Node $(5, 5)$
- Node $(5, 4)$
- Node $(6, 4)$
- Intermediate nodes labeled with transit times and capacities
- Destination node $t$ with label $(1, 20)$
Part II

Online Optimization
5.1 A Simple Problem?

A situation which many of us know: You overslept, you are already late for the morning meeting, all traffic lights are on red, and once you finally reach the office building it takes ages for the elevator to arrive. Who on earth designed this elevator control system? There must be a way to craft a perfect elevator control with a little bit of mathematics!

Let us consider the situation of an elevator that runs in a 25 floor building and which currently is waiting at the fourth floor. Person \( A \) requests a transportation from the penthouse in the 25th floor down to the second floor, \( B \) wants to be moved from the second floor to the first, and \( C \) wants to go upwards from the first floor to the penthouse, see Figure 5.1.

Our current task is easy: we obtain a shortest transportation, if we first pick up \( B \), then \( C \), and finally take care of \( A \).

However, just a second after we pass the third floor on our way down, suddenly \( D \) appears at the third floor and wants to be carried to the second floor. Hmm... what shall we do? Should we first complete our initial transportation schedule and care about \( D \) later? Or should we reverse direction, and pick up \( D \) first?

In any case we waste valuable time since we travel unnecessary distance which we could have avoided if we had known in advance that (and when) \( D \) showed up.

We have just discovered the online aspect of the elevator problem. We are facing incomplete information, and even if every time a new request becomes known we compute a new “optimal” schedule this does not necessarily lead to an overall optimal solution. Suppose that in our particular case \( D \) were actually the last transportation request and our goal would be to finish serving requests as early as possible. Then, in hindsight (or with clairvoyance) the best solution would have been to wait at the third floor for one second until \( D \) arrived.
At the moment our most promising option looks like reversing gear and handling $D$ first (after all, we have just lost one second right now compared to a clairvoyant controller). But what if $E$ shows up on the fourth floor, just before we pick up $D$? Should we then serve $E$ first?

Perhaps the elevator problem is not so trivial!

\[\text{Figure 5.2: } D \text{ appears} \ldots\]

\[\text{Figure 5.3: Is this the future?}\]

### 5.2 Online Computation

In general, traditional optimization techniques assume complete knowledge of all data of a problem instance. However, in reality it is unlikely that all information necessary to define a problem instance is available beforehand. Decisions may have to be made before complete information is available. This observation has motivated the research on online optimization. An algorithm is called online if it makes a decision (computes a partial solution) whenever a new piece of data requests an action.

Let us illustrate online computation with a few more examples.

**Example 5.1 (Ski Rental Problem)** Mabel goes skiing for the first time in her life. She is faced with the question of whether to buy skis for $B$ \(B \in \mathbb{R} \geq 1\) or to rent skis at the cost of $1 \in \mathbb{R}$ per day. Of course, if Mabel knew how many times she would go skiing in the future, her decision would be easy. But unfortunately, she is in an online situation where the number of skiing days only becomes known at the very last day.

**Example 5.2 (Ice Cream Problem)** Luigi owns a small ice cream shop where he only sells the two most popular ice cream flavors, vanilla and chocolate in dishes of one liter. His ice cream machine has two modes, $V$ and $C$ for producing vanilla and chocolate ice
5.2 Online Computation

cream, respectively. Switching between modes requires Luigi to clean the machine which amounts to costs of \(1 \, \text{€}\). Upon the press of a button, the machine produces the sort of ice cream which corresponds to its current mode. The corresponding cost is given in the second column of Table 5.1.

Instead of using the machine, Luigi may also produce ice cream manually, which costs him a bit more, since it eats up a substantial amount of his time. The cost is given in the third column of Table 5.1.

<table>
<thead>
<tr>
<th>Flavor</th>
<th>Machine Cost</th>
<th>Manual Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>1 €</td>
<td>2 €</td>
</tr>
<tr>
<td>S</td>
<td>2 €</td>
<td>4 €</td>
</tr>
</tbody>
</table>

Table 5.1: Cost for producing ice cream by machine and manually. The cost for switching the mode at the machine is \(1 \, \text{€}\).

People queue in front of Luigi’s cafe. Hence, Luigi can only see the first ice cream request in the line and must serve this request before learning the next order. Which requests should Luigi serve by using the machine and which ones should he serve manually in order to minimize his total cost?

Our third problem concerns a more serious problem from the area of scheduling.

Example 5.3 (Online Machine Scheduling with Jobs Arriving Over Time) In scheduling one is concerned with the assignment of jobs (activities) to a number of machines (the resources). In our example, one is given \(m\) identical machines and is faced with the task of scheduling independent jobs on these machines. The jobs become available at their release dates, specifying their processing times. An online algorithm learns the existence of a job only at its release date. Once a job has been started on a machine the job may not be preempted and has to run until completion. However, jobs that have been scheduled but not yet started may be rescheduled. The objective is to minimize the average flow time of a job, where the flow time of a job is defined to be the difference between the completion time of the job and its release date.

All of our problems have in common that decisions have to be made without knowledge of the future. Notice that there is a subtle difference between the ski rental problem and the ice cream problem on the one hand and the elevator and scheduling problems on the other hand: In the the ski rental and ice cream problems a request demands an immediate answer which must be given before the next request is revealed. In the other two problems, an online algorithm is allowed to wait and to revoke decisions. Waiting incurs additional costs, typically depending on the elapsed time. Previously made decisions may, of course, only be revoked as long as they have not been executed.

A general model that comprises many online problems is the concept of a request-answer game:

Definition 5.4 (Request-Answer Game)

A request-answer game \((R, A, C)\) consists of a request set \(R\), a sequence of nonempty answer sets \(A = A_1, A_2, \ldots\) and a sequence of cost functions \(C = C_1, C_2, \ldots\) where \(C_j : R^j \times A_1 \times \cdots \times A_j \rightarrow \mathbb{R}_+ \cup \{+\infty\}\).

We remark here that in the literature one frequently requires each answer set \(A_j\) to be finite. This assumption is made to avoid difficulties with the existence of expected values when studying randomization. However, the finiteness requirement is not of conceptual importance. As remarked in [5 Chapter 7] an infinite or even continuous answer set can be “approximated” by a sufficiently large finite answer set.
Definition 5.5 (Deterministic Online Algorithm)

A deterministic online algorithm $\text{ALG}$ for the request-answer game $(R, A, C)$ is a sequence of functions $f_j : R^j \to A_j$, $j \in \mathbb{N}$. The output of $\text{ALG}$ on the input request sequence $\sigma = r_1, \ldots, r_n$ is

$$\text{ALG}[\sigma] := (a_1, \ldots, a_m) \in A_1 \times \cdots \times A_m,$$

where $a_j := f_j(r_1, \ldots, r_j)$.

The cost incurred by $\text{ALG}$ on $\sigma$, denoted by $\text{ALG}(\sigma)$, is defined as

$$\text{ALG}(\sigma) := C_m(\sigma, \text{ALG}[\sigma]).$$

We now define the notion of a randomized online algorithm.

Definition 5.6 (Randomized Online Algorithm)

A randomized online algorithm $\text{RALG}$ is a probability distribution over deterministic online algorithms $\text{ALG}_x$ ($x$ may be thought of as the coin tosses of the algorithm $\text{RALG}$). The answer sequence $\text{RALG}[\sigma]$ and the cost $\text{RALG}(\sigma)$ on a given input $\sigma$ are random variables.

In terms of game theory we have defined a randomized algorithm as a mixed strategy (where a deterministic algorithm is then considered to be a pure strategy). There is no harm in using this definition of a randomized algorithm since without memory restrictions all types of randomized strategies, mixed strategies, behavioral strategies, and general strategies, are equivalent (see e.g. [5, Chapter 6]).

We illustrate request-answer games by two simple examples. The first example is the classical paging problem, which we will study in more detail in Chapter 7.

Example 5.7 (Paging Problem) Consider a two-level memory system (e.g., of a computer) that consists of a small fast memory (the cache) with $k$ pages and a large slow memory consisting of a total of $N$ pages. Each request specifies a page in the slow memory, that is, $r_j \in R := \{1, \ldots, N\}$. In order to serve the request, the corresponding page must be brought into the cache. If a requested page is already in the cache, then the cost of serving the request is zero. Otherwise one page must be evicted from the cache and replaced by the requested page at a cost of 1. A paging algorithm specifies which page to evict. Its answer to request $r_j$ is a number $a_j \in A_j := \{1, \ldots, k\}$, where $a_j = p$ means to evict the page at position $p$ in the cache.

![Figure 5.4: Paging problem with cache size $k = 3$.](image)

The objective in the paging problem is to minimize the number of page faults. The cost function $C_j$ simply counts the number of page faults which can be deduced easily from the request sequence $r_1, \ldots, r_j$ and the answers $a_1, \ldots, a_j$.

In the paging problem the answer to a request implies an irrevocable decision of how to serve the next request, that is, the paging problem is formulated within the sequence model. We will now provide a second example of a request-answer game which specifies a problem in the time stamp model, where decisions can be revoked as long as they have not been executed yet.
Example 5.8 (Online TCP Acknowledgment) In the online TCP acknowledgment problem introduced in [8] a number of packets are received. Each packet $j$ has a receipt time $t_j \geq 0$ and a weight $w_j \geq 0$. An online algorithm learns the existence of a packet only at its receipt time.

![Figure 5.5: Online TCP acknowledgement problem.](image)

All packets must be acknowledged at some time after their receipt. A single acknowledgment acknowledges all packets received since the last acknowledgment. There is a cost of 1 associated with each acknowledgment. Moreover, if packet $j$ is acknowledged at time $t \geq t_j$, this induces a latency cost of $w_j(t - t_j)$. The goal in the online TCP acknowledgment problem is to minimize the sum of the acknowledgment cost and the latency cost.

At time $t_j$ when packet $r_j = (t_j, w_j)$ is received, an online algorithm $ALG$ must decide when to acknowledge all yet unconfirmed packets. Hence, the answer to request $r_j \in R := \mathbb{R}_+ \times \mathbb{R}_+$ is a real number $a_j \in A_j := \mathbb{R}_+$ with $a_j \geq t_j$. If an additional packet is received before time $a_j$, then $ALG$ is not charged for the intended acknowledgment at time $a_j$, otherwise it incurs an acknowledgment cost of one. The cost function $C_j$ counts the number of actual acknowledgments and adds for each packet $r_j$ the latency cost resulting from the earliest realized acknowledgment in the answer sequence. The condition that packets can not be acknowledged before they are received can be enforced by defining the value $C_j(r_1, \ldots, r_j, a_1, \ldots, a_j)$ to be $+\infty$ if $a_j < t_j$.

5.3 Competitive Analysis

We now define competitiveness of deterministic and randomized algorithms. While the deterministic case is straightforward, the randomized case is much more subtle.

5.3.1 Deterministic Algorithms

Suppose that we are given an online problem as a request-answer game. We define the optimal offline cost on a sequence $\sigma \in R^m$ as follows:

$$\text{OPT}(\sigma) := \min \{ C_m(\sigma, a) : a \in A_1 \times \cdots \times A_m \}.$$  

The optimal offline cost is the yardstick against which the performance of a deterministic algorithm is measured in competitive analysis.

Definition 5.9 (Deterministic Competitive Algorithm)

Let $c \geq 1$ be a real number. A deterministic online algorithm $ALG$ is called $c$-competitive if

$$ALG(\sigma) \leq c \cdot \text{OPT}(\sigma)$$  

holds for any request sequence $\sigma$. The competitive ratio of $ALG$ is the infimum over all $c$ such that $ALG$ is $c$-competitive.
Introduction

Observe that, in the above definition, there is no restriction on the computational resources of an online algorithm. The only scarce resource in competitive analysis is information.

We want to remark here that the definition of $c$-competitiveness varies in the literature. Often an online algorithm is called $c$-competitive if there exists a constant $b$ such that

$$\text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + b$$

holds for any request sequence. Some authors even allow $b$ to depend on some problem or instance specific parameters. In these lecture notes will stick to the definition given above.

Since for a $c$-competitive algorithm ALG we require inequality (5.1) to hold for any request sequence, we may assume that the request sequence is generated by a malicious adversary.

Competitive analysis can be thought of as a game between an online player (an online algorithm) and the adversary. The adversary knows the (deterministic) strategy of the online player, and can construct a request sequence which maximizes the ratio between the player’s cost and the optimal offline cost.

Example 5.10 (A Competitive Algorithm for the Ski Rental Problem) Due to the simplicity of the Ski Rental Problem all possible deterministic online algorithms can be specified. A generic online algorithm $\text{ALG}_j$ rents skis until the woman has skied $k - 1$ times for some $j \geq 1$ and then buys skis on day $j$. The value $j = \infty$ is allowed and means that the algorithm never buys. Clearly, each such algorithm is online. Notice that on a specific request sequence $\sigma$ algorithm $\text{ALG}_j$ might not get to the point that it actually buys skis, since $\sigma$ might specify less than $j$ skiing days. We claim that $\text{ALG}_j$ for $j = B$ is $c$-competitive with $c = 2 - 1/B$.

Let $\sigma$ be any request sequence specifying $n$ skiing days. Then our algorithm has cost $\text{ALG}_B(\sigma) = n$ if $n \leq B - 1$ and cost $\text{ALG}_B(\sigma) = B - 1 + B = 2B - 1$ if $j \geq B$. Since the optimum offline cost is given by $\text{OPT}(\sigma) = \min\{n, B\}$, it follows that our algorithm is $(2 - 1/B)$-competitive.

Example 5.11 (Lower Bounds for the Ski Rental Problem) In Example 5.10 we derived a $(2 - 1/B)$-competitive algorithm for the ski rental problem with buying cost $B$. Can we do any better?

The answer is »no«! Any competitive algorithm $\text{ALG}$ must buy skis at some point in time, say, day $i$. The adversary simply presents skiing requests until the algorithm buys and then ends the sequence. Thus, the online cost is $\text{ALG}(\sigma) = j + B$, whereas the optimal offline cost is $\text{OPT}(\sigma) = \min\{j, B\}$.

If $j \leq B - 1$, then

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} = \frac{j + B}{\min\{j + 1, B\}} = \frac{j + B}{j + 1} = 1 + \frac{B - 1}{j + 1} \geq 1 + \frac{B - 1}{B} = 2 - \frac{1}{B}.$$  

Of $j \geq B - 1$, then

$$\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} = \frac{j + B}{\min\{j + 1, B\}} \geq \frac{2B - 1}{B} = 2 - \frac{1}{B}.$$  

Hence, the ratio of the online cost and the offline cost is at least $2 - 1/B$.  

5.3.2 Randomized Algorithms

For randomized algorithms we have to be precise in defining what kind of information about the online player is available to the adversary. This leads to different adversary models which are explained below. For an in-depth treatment we refer to [5][22].
If the online algorithm is randomized then according to intuition, the adversary has less power since the moves of the online player are no longer certain. The weakest kind of adversary for randomized algorithms is the oblivious adversary:

**Definition 5.12 (Oblivious Adversary)**

An oblivious adversary \( \text{OBL} \) must construct the request sequence in advance based only on the description of the online algorithm but before any moves are made.

We can now define competitiveness against this adversary:

**Definition 5.13 (Competitive Algorithm against an Oblivious Adversary)**

A randomized algorithm \( \text{RALG} \), distributed over a set \( \{\text{ALG}_y\} \) of deterministic algorithms, is \( c \)-competitive against an oblivious adversary for some \( c \geq 1 \), if

\[
\mathbb{E}_Y [\text{ALG}_y(\sigma)] \leq c \cdot \text{OPT}(\sigma).
\]

for all request sequences \( \sigma \). Here, the expression \( \mathbb{E}_Y [\text{ALG}_y(\sigma)] \) denotes the expectation with respect to the probability distribution \( Y \) over \( \{\text{ALG}_y\} \) which defines \( \text{RALG} \).

In case of a deterministic algorithm the above definition collapses to that given in Definition 5.9.

**Example 5.14 (Ski Rental Problem Revisited)** We look again at the Ski Rental Problem given in Example 5.1. We now consider the following randomized algorithm \( \text{RANDSKI} \) against an oblivious adversary. Let \( \rho := B/(B-1) \) and \( \alpha := \frac{\rho-1}{\rho^n - 1} \). At the start \( \text{RANDSKI} \) chooses a random number \( k \in \{0, \ldots, B-1\} \) according to the distribution \( \Pr[k = x] := \alpha \rho^x \). After that, \( \text{RANDSKI} \) works completely deterministic, buying skis after having skied \( k \) times. We analyze the competitive ratio of \( \text{RANDSKI} \) against an oblivious adversary. Note that it suffices to consider sequences \( \sigma \) specifying at most \( B \) days of skiing. For a sequence \( \sigma \) with \( n \leq B \) days of skiing, the optimal cost is clearly \( \text{OPT}(\sigma) = n \). The expected cost of \( \text{RANDSKI} \) can be computed as follows

\[
\mathbb{E} [\text{RANDSKI}(\sigma)] = \sum_{k=0}^{n-1} \alpha \rho^k (k + B) + \sum_{k=n}^{B-1} \alpha \rho^k n.
\]

We now show that \( \mathbb{E} [\text{RANDSKI}(\sigma)] = c \rho := \frac{\rho^n}{\rho^n - 1} \text{OPT}(\sigma) \).

\[
\mathbb{E} [\text{RANDSKI}(\sigma)] = \sum_{j=0}^{n-1} p_j (j + B) + \sum_{j=n}^{B-1} p_j n
\]

\[= \sum_{j=0}^{n-1} \alpha \rho^j (j + B) + \sum_{i=n}^{B-1} \alpha \rho^i n
\]

\[= \alpha \sum_{i=0}^{n-1} \rho^i j + \alpha \sum_{j=0}^{B-1} \rho^j + \alpha n \sum_{j=n}^{B-1} \rho^j
\]

\[= \alpha \left( (n-1) \rho^{n+1} - n \rho^n + \rho \right) + \alpha B \rho^n - 1
\]

\[+ \alpha n \rho^B - \rho^n
\]

\[= \alpha \frac{(n-1) \rho^{n+1} - n \rho^n + \rho}{(\rho - 1)^2}
\]

\[+ \alpha B \rho^n - 1 + \alpha n \rho^B - \rho^n
\]
\[ \alpha \left( \frac{1}{\rho - 1} \right)^2 \left( (n - 1)\rho^{n+1} - \rho^n + \rho^n \right) \]
\[ = \frac{\alpha}{(\rho - 1)^2} \left( \rho B - \rho^n + \rho - 1 \right) \]
\[ = \frac{\alpha}{(\rho - 1)^2} \left( (\rho - 1)\rho B - (\rho - 1)(\rho^n - 1) \right) \]
\[ = \frac{\alpha}{\rho - 1} \left( \rho B - \rho^n + 1 \right) \]
\[ \leq \frac{\alpha}{\rho - 1} \cdot \rho B \quad \text{(since } \rho \geq 1) \]
\[ = \frac{\rho B}{\rho - 1} \cdot n \]
\[ = \frac{\rho^B}{\rho - 1} \cdot \text{OPT}(\sigma). \]

Hence, RANDSKI is \( c_B \)-competitive with \( c_B = \frac{\rho B}{\rho - 1} \). Since \( \lim_{B \to \infty} c_B = \epsilon/\epsilon = 1.58 \), this algorithm achieves a better competitive ratio than any deterministic algorithm whenever \( 2B - 1 > \epsilon/(\epsilon - 1) \), that is, when \( B > (2\epsilon - 1)/(\epsilon - 1) \).

In contrast to the oblivious adversary, an adaptive adversary can issue requests based on the online algorithm’s answers to previous ones.

The adaptive offline adversary (ADOFF) defers serving the request sequence until he has generated the last request. He then uses an optimal offline algorithm. The adaptive online adversary (ADON) must serve the input sequence (generated by himself) online. Notice that in case of an adaptive adversary \( \text{ADV} \), the adversary’s cost \( \text{ADV}(\sigma) \) for serving \( \sigma \) is a random variable.

**Definition 5.15 ( Competitive Algorithm against an Adaptive Adversary )**

A randomized algorithm \( \text{RALG} \), distributed over a set \( \{\text{ALG}_y\} \) of deterministic algorithms, is called \( c \)-competitive against an adaptive adversary \( \text{ADV} \) for some \( c \geq 1 \), where \( \text{ADV} \in \{\text{ADON}, \text{ADOFF}\} \), if

\[ \mathbb{E}_{\sigma} \left[ \text{ALG}_y(\sigma) - c \cdot \text{ADV}(\sigma) \right] \leq 0. \]

for all request sequences \( \sigma \). Here, \( \text{ADV}(\sigma) \) denotes the adversary cost which is a random variable.

The above adversaries differ in their power. Clearly, an oblivious adversary is the weakest of the three types and an adaptive online adversary is no stronger than the adaptive offline adversary. Moreover, as might be conjectured, the adaptive offline adversary is so strong that randomization adds no power against it. More specifically, the following result holds:

**Theorem 5.16 ( \cite{3} )** Let \( (R, A, C) \) be a request-answer game. If there exists a randomized algorithm for \( (R, A, C) \) which is \( c \)-competitive against any adaptive offline adversary, then there exists also a \( c \)-competitive deterministic algorithm for \( (R, A, C) \).

The strength of the adaptive online adversary can also be estimated:

**Theorem 5.17 ( \cite{3} )** Let \( (R, A, C) \) be a request-answer game. If there exists a randomized algorithm for \( (R, A, C) \) which is \( c \)-competitive against any adaptive online adversary, then there exists a \( c^2 \)-competitive deterministic algorithm for \( (R, A, C) \).

For more information about the relations between the various adversaries we refer to \cite{3} and the textbooks \cite{5, 22}.
5.4 Exercises

Exercise 5.1
Given an optimal offline-algorithm for the ice cream problem in Example 5.2.

Exercise 5.2
Formulate the ski rental problem, the ice cream problem as request answer games.

Exercise 5.3
Formulate the elevator problem and the scheduling problem from Example 5.3 as request answer problems (using infinite answer sets).

Exercise 5.4
Show that against an adaptive offline adversary RANDSKI does not achieve a competitive ratio smaller than \( 2 - \frac{1}{B} \) without using Theorem 5.16.

Exercise 5.5 (Bin packing)
In bin packing, a finite set of items of size \( s_i \in (0, 1] \) is supposed to be packed into bins of unit capacity using the minimum possible number of bins. In online bin packing, an item \( i \) has to be packed before the next item \( i + 1 \) becomes known. Once an item is packed it cannot be removed and put in another bin.

(a) Show that any \( c \)-competitive deterministic algorithm for the online Bin Packing Problem has \( c \geq 4/3 \).

(b) The FIRSTFIT-Algorithm puts an item \( i \) always in the first bin that has still enough space to fit in \( i \). If there is no bin left with enough space then a new bin is opened. Show that FIRSTFIT is \( 2 \)-competitive.
Examples and Elementary Techniques

In this section we present elementary techniques that are used in the design and analysis of algorithms. Our toy problem, so as to speak, will be the list accessing problem. In this problem we are given a linear list \( L \) containing a finite number of elements. Requests for elements in the list arrive online. Given a request for an element \( x \) the cost of answering the request is the position of \( x \) in \( L \). In order to minimize the cost, one can reorganize the list as follows:

**Free exchanges** Directly after a request to \( x \), the element \( x \) can be moved to an arbitrary position further to the front of the list at no cost.

**Paid exchanges** Other exchanges of adjacent elements in the list are possible at unit cost.

The motivation for studying the above cost model is the situation of a linked list data structure. During a search from the front of the list to a specific element \( x \) one could store a pointer to an arbitrary position further to the front. Thus, a move of \( x \) could be carried out at »no cost«. The goal of the list accessing problem is to minimize the total cost, which equals the sum of the access and reorganization cost. An online algorithm for the list accessing problem must answer request \( r_i \) before the next request \( r_{i+1} \) is revealed.

We consider the following online strategies:

**Algorithm MTF (move to the front)** After request \( r_i \) the element \( r_i \) is moved to the front of the list (using only free exchanges).

**Algorithm TRANS (transpose)** After request \( r_i \) the element \( r_i \) is moved to the front one position by exchanging it with its predecessor in the list (using a free exchange).

**Algorithm FC (frequency count)** The algorithm maintains for each element in the list \( L \) a counter \( \text{count}[x] \), indicating how often \( x \) has been requested. Upon a request \( r_i \) the counter \( \text{count}[r_i] \) is updated and the list is resorted according to decreasing counter values afterwards.

### 6.1 Cruel Adversaries

The cruel adversary concept tries to enforce a bad behavior of an online algorithm by hurting it as much as possible in every single step. We use this adversary for proving that FC is not competitive.
Theorem 6.1 For any $\varepsilon > 0$ there are arbitrary long request sequences $\sigma$ such that $\text{TRANS}(\sigma)/\text{OPT}(\sigma) \geq 2m/3 - \varepsilon$.

Proof: The cruel adversary always requests the last element in the list of $\text{TRANS}$. This amounts to requesting the elements that were in the last two positions of the initial list alternatingly. The cost of $\text{ALG}$ on such a sequence of length $n$ are $\text{TRANS}(\sigma) = nm$, where as usual $m$ denotes the number of elements in the list.

The adversary can serve the request sequence as follows. After the first two requests, both relevant elements are moved to the first two positions in its list. Hence, from this moment on, the cost of the adversary for any pair of requests to these elements is 3. Thus, the total cost of $\text{OPT}$ can be bounded from above as $\text{OPT}(\sigma) \leq 2m + 3(n - 2)/2$. Comparing this value with the cost of $nm$ for $\text{TRANS}$ shows the theorem.

6.2 Potential Functions

Potential functions are an important tool for proving competitiveness results. In our example, we use such a potential function for the analysis of $\text{MTF}$.

Let $\text{ALG}$ be an online algorithm for some online minimization problem. The potential $\Phi$ maps the current configurations of $\text{ALG}$ and $\text{OPT}$ to a nonnegative value $\Phi \geq 0$. We denote by $\Phi_i$ the potential after request $i$. Let $c_i$ be the cost incurred by $\text{MTF}$ on request $r_i$. The amortized cost $a_i$ for serving request $r_i$ is defined as

$$a_i = \frac{c_i}{\text{cost of \text{MTF} for } r_i} + \frac{\Phi_i - \Phi_{i-1}}{\text{change in potential}}.$$

The intuition behind a potential function and the amortized cost is to measure »how well« the current configuration of the online algorithm is compared to an optimal offline algorithm. The potential can be viewed as a bank account. If the difference $\Phi_i - \Phi_{i-1}$ is negative, then the amortized cost underestimate the real cost $c_i$. The difference is covered by withdrawal from the account. We have:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i + (\Phi_0 - \Phi_n) \leq \sum_{i=1}^{n} a_i + \Phi_0. \quad (6.1)$$

Thus, up to an additive constant, the real cost is bounded from above by the amortized cost. If we can show that $a_i \leq c \text{OPT}(r_i)$, where $\text{OPT}(r_i)$ is the cost incurred by the optimal offline algorithm on request $r_i$, then it follows that the online algorithm $\text{ALG}$ is $c$-competitive.

In our particular application, we define the potential function by

$$\Phi_i = \text{number of inversions} = |\{(a, b) : a \text{ is before } b \text{ in } \text{MTF}'s \text{ list, but } b \text{ is before } a \text{ in the list of } \text{OPT}\}|.$$

When bounding the amortized cost, we imagine that request $r_i$ is first served by $\text{MTF}$ and then by $\text{OPT}$.

Let $x$ be the element requested in $r_i$, which is at position $k$ in $\text{MTF}$’s list and at position $j$ in the list organized by $\text{OPT}$ (see Figure 6.1 for an illustration). We denote by $f$ and $p$ the number of free respective paid exchanges used by $\text{OPT}$ on this request.

Let $v$ be the number of elements that are in front of $x$ in $\text{MTF}$’s list but behind $x$ in $\text{OPT}$’s list before the request (these elements are indicated by stars * in Figure 6.1). By moving $x$ to
6.3 Averaging over Adversaries

Figure 6.1: Configurations of the lists organized by MTF and OPT at the time of request \( r_i = x \).

the front of the list, \( v \) inversions are removed and at most \( j - 1 \) new inversions are created. Thus, the change in potential due to actions of MTF is at most \( -v + j - 1 \).

By reorganizing the list, OPT can increase the potential by at most \( p - f \) (every free exchange decreases the number of inversions, every paid exchange can increase the number of inversions by at most one). Thus, the amortized cost satisfies:

\[
\begin{align*}
\alpha_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= k + \Phi_i - \Phi_{i-1} \\
&\leq k - v + j - 1 + p - f \\
&= j + p + k - v - 1. 
\end{align*}
\]

(6.2)

Recall that \( v \) is the number of elements in MTF’s list that were in front of \( x \) but came after \( x \) in OPT’s list. Thus \( k - v \) elements are before \( x \) in both lists. Since OPT contained \( x \) at position \( j \), we can conclude that \( k - v - 1 \leq j - 1 \). Using this inequality in (6.2) gives

\[
\alpha_i \leq j + p + (j - 1) \leq 2(j + p) - 1 = 2\text{OPT}(r_i) - 1.
\]

Summing over all requests \( r_1, \ldots, r_n \) results in \( \sum_{i=1}^{n} \alpha_i \leq 2\text{OPT}(\sigma) - n \). Since \( \text{OPT}(\sigma) \leq nm \), we can conclude that

\[
\sum_{i=1}^{n} \alpha_i \leq \left( 2 - \frac{1}{m} \right) \text{OPT}(\sigma). \quad (6.3)
\]

We finally use that MTF and OPT start with identical list configurations such that \( \Phi_0 = 0 \). From (6.3) and (6.1) we get the following theorem:

**Theorem 6.2** MTF achieves a competitive ratio of \( 2 - 1/m \) for the list accessing problem. \( \square \)

### 6.3 Averaging over Adversaries

Usually, the optimal offline cost is hard to bound both from above and below. For proving lower bound results on the competitive ratio of online algorithm averaging over adversaries can help. The basic idea is to have a set \( M \) of algorithms each of which serves the same request sequence \( \sigma \). If the sum of the costs of all algorithms in \( M \) is at most \( C \), then there must be some algorithm in \( M \) which has cost at most \( C/|M| \), the average cost. Hence, we get that \( \text{OPT}(\sigma) \leq C/|M| \).

**Theorem 6.3** Any deterministic algorithm for the list accessing problem has a competitive ratio at least \( 2 - 2/(m + 1) \), where \( m = |L| \) denotes the length of the list.
**Proof:** We use a cruel adversary in conjunction with the averaging idea outlined above. Given a deterministic algorithm \( \text{ALG} \), the cruel adversary chooses a sequence length \( n \) and always requests the last element in \( \text{ALG} \)'s list. The online cost is then \( \text{ALG}(\sigma) = nm \).

Consider the \( m! \) static offline algorithms which correspond to the \( m! \) permutations of the list elements. Each of these algorithm initially sorts the list according to its permutation and then keeps the list fixed for the rest of the sequence. The sorting cost of each algorithm can be bounded by a constant \( b \) which depends only on \( m \).

We now bound the sum of the costs of the \( m! \) static algorithms on a request \( r_i = x \). For each of the \( m \) possible positions of \( x \) there are exactly \( (m - 1)! \) permutations which have \( x \) at this positions. Hence, the sum of the costs is

\[
\sum_{j=1}^{m} j(m - 1)! = (m - 1)! \frac{m(m + 1)}{2}.
\]

Thus, the average cost of the static algorithms is at most

\[
b + \frac{m + 1}{2}.
\]

This yields:

\[
\frac{\text{ALG}(\sigma)}{\text{OPT}(\sigma)} \geq \frac{nm}{n \frac{m+1}{2} + b} = \frac{2m}{m + 1 + 2b/n} \to_{n \to \infty} \frac{2m}{m + 1} = \left(2 - \frac{2}{m}\right).
\]

This completes the proof. \( \square \)

### 6.4 Yao’s Principle and Lower Bounds for Randomized Algorithms

A lower bound on the competitive ratio is usually derived by providing a set of specific instances on which no online algorithm can perform well compared to an optimal offline algorithm. Here, again, we have to distinguish between deterministic and randomized algorithms.

For deterministic algorithms finding a suitable set of request sequences is in most cases comparatively easy. For randomized algorithms, however, it is usually very difficult to bound the expected cost of an arbitrary randomized algorithm on a specific instance from below. As an illustration, consider the extremely simple ski rental problem. How would you fool an arbitrary randomized algorithm?

In this section we derive a lower bound technique called Yao’s principle which ha become the standard tool for proving lower bounds for randomized algorithms.

**Theorem 6.4 (Yao’s Principle)** Let \( \{ \text{ALG}_y : y \in \mathcal{Y} \} \) denote the set of deterministic online algorithms for an online minimization problem. If \( X \) is a probability distribution over input sequences \( \{ \sigma_x : x \in \mathcal{X} \} \) and \( \bar{c} \geq 1 \) is a real number such that

\[
\inf_{y \in \mathcal{Y}} \mathbb{E}_X [\text{ALG}_y(\sigma_x)] \geq \bar{c} \mathbb{E}_X [\text{OPT}(\sigma_x)], \tag{6.4}
\]

then \( \bar{c} \) is a lower bound on the competitive ratio of any randomized algorithm against an oblivious adversary.
Yao’s principle allows to trade randomization in an online algorithm for randomization in the input. Basically, it says the following: if we have a probability distribution \( X \) over the input sequences such that on average (with respect to \( X \)) every deterministic algorithm performs badly (compared to an optimal offline algorithm), then also for every randomized algorithm there exists a bad input sequence.

We will prove Yao’s principle only in the special case of a finite set of input sequences and a finite set of possible deterministic algorithms. This way we avoid a lot of subtle issues about the existence of expectations and measure theory. Since Yao’s principle is derived from game theoretic concepts, we briefly review the necessary material to explain the ideas behind the theorem.

### 6.4.1 Two-Person Zero-Sum Games

Consider the following children’s game: Richard and Chery put their hands behind their backs and make a sign for one of the following: stone (closed fist), paper (open palm), and scissors (two fingers). Then, they simultaneously show their chosen sign. The winner is determined according to the following rules: stone beats scissors, scissors beats paper and paper beats stone. If both players show the same sign, the game is a draw. The loser pays \( 1 \epsilon \) to the winner. In case of a draw, no money is payed.

We can represent the scissors-paper-stone game as a matrix \( M \), where the rows correspond to Richard’s choices and the columns represent the choices of Chery. The entry \( m_{ij} \) is the amount of money Richard wins if he chooses \( i \) and Chery chooses \( j \). Table 6.1 shows the corresponding matrix.

<table>
<thead>
<tr>
<th></th>
<th>Stone</th>
<th>Scissors</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stone</td>
<td>0</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>Scissors</td>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Paper</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: Matrix for the scissors-paper-stone game

The scissors-paper-stone game is an example of a two-person zero sum game. The matrix associated with the game is called payoff matrix and the game is a zero-sum game since the net amount by Richard and Chery is always exactly zero.

**Definition 6.5 (Two-person zero-sum game)**

Given any matrix \( n \times m \)-Matrix \( M = (m_{ij}) \), the \( n \times m \) two-person zero-sum game \( \Gamma_M \) with payoff matrix \( M \) is played by two persons \( R \) and \( C \). The set of possible strategies for the row player \( R \) is in correspondence with the rows of \( M \), and likewise for the strategies of the column player \( C \). These strategies are called pure strategies of \( R \) and \( C \). The entry \( m_{ij} \) is the amount paid by \( C \) to \( R \) when \( R \) chooses strategy \( i \) and \( C \) chooses strategy \( j \). The matrix is known to both players.

The goal of the row player \( R \) is to maximize his profit, while the column player \( C \) seeks to minimize her loss.

You may imagine the situation of an online player and the offline adversary in the game theoretic setting as follows: the rows correspond to all possible deterministic online algorithms, the columns are in correspondence with the possible input sequences. If the online player chooses \( \text{ALG}_i \) and the adversary \( \sigma_j \), then the payoff of the adversary (who is the row player) is \( \text{ALG}_i(\sigma_j)/\text{OPT}(\sigma_j) \).

---

1 This imagination is not exactly how we will handle things later, but for the moment it gives a good motivation.
Suppose that \( R \) chooses strategy \( i \). Then, no matter what \( C \) chooses, he is assured a win of \( \min_j m_{i,j} \). An optimal (pure) strategy for \( R \) is to choose \( i^* \) such that \( \min_j m_{i^*,j} = \max_i \min_j m_{i,j} \). We set

\[
V_R := \max_i \min_j m_{i,j} \quad (6.5)
\]

\[
V_C := \min_j \max_i m_{i,j}. \quad (6.6)
\]

Here, Equation (6.6) gives the value corresponding to the optimal strategy of the row player. It is easy to see that \( V_R \leq V_C \) for any two-person zero sum game (see Exercise 6.2). In fact, strict inequality may occur (see Exercise 6.3). If \( V_R = V_C \) then the common value is called the value of the game.

Now let us do the transition to randomized or mixed strategies. A mixed strategy for a player is a probability distribution over his/her deterministic strategies. Thus, a mixed strategy for \( R \) is a vector \( p = (p_1, \ldots, p_n)^T \) and for \( C \) it is a vector \( q = (q_1, \ldots, q_m) \) such that all entries are nonnegative and sum to one.

The payoff to \( R \) becomes a random variable with expectation

\[
\sum_{i=1}^n \sum_{j=1}^m p_i m_{i,j} q_j = p^T M q.
\]

How do optimal mixed strategies look like? \( R \) can maximize his expected profit by choosing \( p \) such as to maximize \( \min_q p^T M q \). Likewise, \( C \) can choose \( q \) to minimize \( \max_p p^T M q \). We define

\[
v_R := \max_p \min_q p^T M q \quad (6.7)
\]

\[
v_C = \min_q \max_p p^T M q. \quad (6.8)
\]

The minimum and maximum in the above equations are taken over all possible probability distributions.

Observe that once \( p \) is fixed in (6.7), then \( p^T M q \) becomes a linear function of \( q \) and is minimized by setting to one the entry \( q_j \) with the smallest coefficient in this linear function. A similar observation applies to (6.8). We thus have

\[
v_R = \max_p \min_j p^T M e_j \quad (6.9a)
\]

\[
v_C = \min_q \max_i e_i^T M q. \quad (6.9b)
\]

The famous Minimax Theorem von von Neumann states that with respect to mixed strategies, any two-person zero-sum game has a value:

**Theorem 6.6 (von Neumann’s Minimax Theorem)** For any two-person zero-sum game specified by a matrix \( M \)

\[
\max_p \min_q p^T M q = \min_q \max_p p^T M q. \quad (6.10)
\]

A pair of of mixed strategies \((p^*, q^*)\) which maximizes the left-hand side and minimizes the right-hand side of (6.10) is called a saddle-point and the two distributions are called optimal mixed strategies.

**Proof:** By (6.9) it suffices to show that

\[
\max_p \min_j p^T M e_j = \min_q \max_i e_i^T M q. \quad (6.11)
\]
Consider the left-hand side of (6.11). The goal is to choose $p$ such that $\min_j \sum_{i=1}^n m_{ij} p_i$ is maximized. This can be equivalently expressed as the following Linear Program:

\[
\begin{align*}
\max \quad & z \\
\text{s.t.} \quad & z - \sum_{i=1}^n m_{ij} p_i \leq 0, \quad j = 1, \ldots, m \\
\quad & \sum_{i=1}^n p_i = 1 \\
\quad & p \geq 0.
\end{align*}
\]

Similarly, we can reformulate the right-hand side of (6.11) as another Linear Program:

\[
\begin{align*}
\min \quad & w \\
\text{s.t.} \quad & w - \sum_{j=1}^m m_{ij} q_j \geq 0, \quad i = 1, \ldots, n \\
\quad & \sum_{j=1}^m q_j = 1 \\
\quad & q \geq 0.
\end{align*}
\]

Observe that (6.12) and (6.13) are dual to each other. Clearly, both Linear Programs are feasible, hence the claim of the theorem follows by the Duality Theorem of Linear Programming.

Using our observation in (6.9) we can reformulate the Minimax Theorem as follows:

**Theorem 6.7 (Loomi’s Lemma)** For any two-person zero-sum game specified by a matrix $M$

\[
\max_p \min_j p^T M e_j = \min_q \max_i e_i^T M q.
\]

Loomi’s Lemma has an interesting consequence. If $C$ knows the distribution $p$ used by $R$, then here optimal mixed strategy is in fact a pure strategy. The analogous result applies vice versa.

From Loomi’s Lemma we can easily derive the following estimate, which is known as Yao’s Principle: Let $\tilde{p}$ be a fixed distribution over the set of pure strategies of $R$. Then

\[
\min_q \max_i e_i^T M q = \max_p \min_j p^T M e_j \quad \text{(by Loomi’s Lemma)}
\]

\[
\geq \tilde{p}^T \min_j M e_j.
\]

The inequality

\[
\min_q \max_i e_i^T M q \geq \tilde{p}^T \min_j M e_j
\]

is referred to as Yao’s Principle.

### 6.4.2 Yao’s Principle and its Application to Online Problems

We now prove the finite version of Yao’s Principle for Online-Problems (cf. Theorem 6.3 on page 80). Let II be an online problem with a finite set of deterministic online algorithms $\{\text{ALG}_j : j = 1, \ldots, m\}$ and a finite set $\{\sigma_i : i = 1, \ldots, n\}$ of input instances.
Suppose we want to prove that no randomized algorithm can achieve a competitive ratio smaller than \( c \) against an oblivious adversary. Then we need to show that for any distribution \( p \) over \( \{ \text{ALG}_j \} \) there is an input sequence \( \sigma_i \) such that

\[
E_q[\text{ALG}_j(\sigma_i)] \geq c \cdot \text{OPT}(\sigma_i).
\]

This is equivalent to proving that

\[
\min_q \max_i \{ E_q[\text{ALG}_j(\sigma_i)] - c \cdot \text{OPT}(\sigma_i) \} \geq 0.
\] \hspace{1cm} (6.15)

Consider following two-person zero-sum game with payoff matrix \( M = (m_{ij}) \), where \( m_{ij} := \text{ALG}_j(\sigma_i) - c \cdot \text{OPT}(\sigma_j) \). The online player (column player) wants to minimize her loss, the adversary (row player) wants to maximize his win. We have for any distribution \( p \) over the set of input instances \( \{ \sigma_i \} \):

\[
\min_q \max_i \{ E_q[\text{ALG}_j(\sigma_i)] - c \cdot \text{OPT}(\sigma_i) \} = \min_q \max_i \left( \sum_j m_{ij} q_j \right) = \min_j \sum_q E_p[\text{ALG}_j(\sigma_i)] - c \cdot E_p[\text{OPT}(\sigma_i)],
\] \hspace{1cm} (6.16)

where \( E_p[\text{ALG}_j(\sigma_i)] \) denotes the expected cost of the deterministic algorithm \( \text{ALG}_j \) with respect to the distribution \( p \) over the input. Hence, if \( E_p[\text{ALG}_j(\sigma_i)] \geq c \cdot E_p[\text{OPT}(\sigma_i)] \) for all \( j \), the term in (6.16) will be nonnegative. This shows (6.15).

**Example 6.8** Once more we consider the ski rental problem and show how Yao’s Principle can be applied to get a lower bound for randomized algorithms.

Let \( B := 10 \). We construct a probability distribution over a set of two possible input instances: With probability \( \frac{1}{2} \), there will be only one day of skiing, with probability \( \frac{1}{2} \), there will be 20 days of skiing.

The expected optimal cost under this distribution is given by \( E[\text{OPT}(\sigma)] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 10 = \frac{11}{2} \).

Let \( \text{ALG}_j \) be the online algorithm which buys skis on day \( j \). Then we have:

\[
E[\text{ALG}_j(\sigma)] = \begin{cases} 10 & \text{for } j = 1 \\ \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (j - 1 + 10) & \text{for } j = 2, \ldots, 20. \\ \frac{1}{2} + \frac{11}{2} & \text{for } j > 20. \end{cases}
\]

Hence we obtain that

\[
\frac{E[\text{ALG}_j(\sigma)]}{E[\text{OPT}(\sigma)]} \geq \frac{6}{11} = \frac{12}{22} = \frac{12}{11}
\]

for all \( j \). Thus, no randomized algorithm can beat the competitive ratio of 12/11. \( \triangleq \)

### 6.5 Exercises

**Exercise 6.1 (Potential function and amortised costs)**

Recall the application of potential functions and amortized costs in the proof of competitiveness of MTF. Potential functions are a very useful and elegant tool as we want to illustrate with the following examples from the area of data structures.
Consider the data structure $D_0$ on which $n$ operations can be executed. For $i = 1, \ldots, n$ let $c_i$ denote the cost of an algorithm $\text{ALG}$ caused by the $i$th operation. Let $D_i$ denote the data structure after the $i$th operation. A potential function $\Phi$ assigns a real number $\Phi(D_i)$ to a data structure $D_i$. The amortized costs $\hat{c}_i$ for the $i$th operation are

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

The intuition for a potential function and the amortized costs is the following: $\Phi(D_i)$ measures/estimates, «how well» the current state of the structure is. Imagine $\Phi(D_i)$ as a bank account: if the difference $\Phi(D_i) - \Phi(D_{i-1})$ is negative, then $\hat{c}_i$ underestimates the actual costs $c_i$. The difference can be balanced by withdrawing the potential-loss/difference from the account.

For the amortized cost holds:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i + \Phi(D_0) - \Phi(D_n).$$

In the following we analyze stack operations. A stack is a Last-in-First-Out memory/buffer $S$, on which the following operations are defined:

- $\text{PUSH}(S, x)$ puts the object $x$ on top of the stack.
- $\text{POP}(S)$ returns the topmost object of the stack and removes it. (If the stack is empty then the operation is aborted with an error message.)

Both operations cost $\mathcal{O}(1)$ time units. Additionally we allow the operation $\text{MULTIPOP}(S, k)$, which removes the topmost $k$ objects from the stack. This operation takes $\mathcal{O}(k)$ units of time.

(a) Apply the common worst case analysis and show how to obtain a time bound of $\mathcal{O}(n^2)$ for a sequence of $n$ stack operations (beginning with an empty stack). Assume that for the $\text{MULTIPOP}$-operations there are always sufficiently many elements on the stack.

(b) Use the potential $\Phi(S) := |S|$ and amortized costs in order to improve the worst case bound and obtain a time bound of $\mathcal{O}(n)$.

**Exercise 6.2**
Consider an arbitrary two-person zero-sum game defined by the $n \times m$-Matrix $M$. Let

$$V_R := \max_i \min_j m_{ij},$$

$$V_C := \min_j \max_i m_{ij}.$$  

Show that $v_R \leq v_C$.

**Exercise 6.3**
Give a two-person zero-sum game such that $V_R < V_C$. 
The Paging Problem

The paging problem is one of the fundamental and oldest online problem. In fact, this problem inspired the notion of competitive analysis in [25].

In the paging problem we are given a memory system consisting of two levels. Each level can store memory pages of a fixed size. The second level consists of slow memory which stores a fixed set $P = \{p_1, \ldots, p_N\}$ of $N$ pages. The first level is the fast memory (cache), which can store an arbitrary $k$ element subset of $P$. Usually $k \ll N$.

The memory system is faced with a sequence $\sigma = r_1, \ldots, r_n$ of page requests. Upon a request to page $p_i$ the page is accessed and must be presented in the cache. If $p_i$ is already in the cache, we have a cache hit and the system does not need to do anything. If $p_i$ is not in the cache, then we speak of a cache miss and the system must copy $p_i$ into the cache thereby replacing another page in the cache. A paging algorithm must therefore decide which page to evict from the cache upon a page fault. An online algorithm for the paging problem must process request $r_i$ before $r_{i+1}$ is revealed to the algorithm.

In this section we consider a simple cost model for the paging problem, the page fault model. Under this model, the goal is simply to minimize the number of page faults. This model can be justified by the fact that a cache hit means fast (almost negligible) access times, whereas a page fault involves access to the slow memory and evicting operations. Typical applications of the paging problem include the organization of CPU/memory systems and caching of disk accesses.

7.1 An Optimal Offline Algorithm

The paging problem is one of the few online problems where an optimal offline algorithm is known and, moreover, this algorithm has efficient (polynomial) running time. We first consider the offline problem, where we are given a known sequence $\sigma = r_1, \ldots, r_n$ of page requests.

**Algorithm LFD (longest forward distance)** Upon a page fault the algorithm evicts that page from the cache whose next request is furthest into the future.

Algorithm LFD is clearly an offline algorithm since it requires complete knowledge of future page requests.

**Lemma 7.1** Let $\text{ALG}$ be a deterministic paging algorithm and let $\sigma = r_1, \ldots, r_n$ be an arbitrary request sequence. Then for $i = 1, \ldots, n$ there is an algorithm $\text{ALG}_i$ with the following properties:

1. $\text{ALG}_i$ serves the first $i - 1$ requests in the same way as $\text{ALG}$. 

2. If the $i$th request amounts to a page fault, then ALG$_i$ evicts that page from the cache whose next request is furthest into the future (that is, using the LFD-rule).

3. ALG$_i$(σ) ≤ ALG(σ).

Proof: We construct ALG$_i$ from ALG. If the $i$th request does not involve a page fault, then there is nothing to prove. Thus, assume for the remainder of the proof that $r_i = f$ produces a page fault for ALG.

We let $p$ be the page which is replaced by ALG by $f$ and $q$ the page which would have been replaced according to the LFD-rule. It suffices to treat the case $p \neq q$.

Algorithm ALG$_i$ replaces $q$ by $f$. The contents of the cache after $r_i$ are $X \cup \{p\}$ for ALG$_i$ and $X \cup \{q\}$ for ALG, where $X$ is a set of $k - 1$ common pages in the cache. Note that $f \in X$ and that the next request to page $p$ must occur before the next request to $q$, since the LFD-rule caused $q$ to be ejected and not $p$.

After request $r_i$, the algorithm ALG$_i$ processes the remaining requests as follows until page $p$ is requested again (page $p$ must be requested again, since $q$ is requested after $p$ and we assumed that $p \neq q$).

- If ALG evicts $q$ from the cache, then ALG$_i$ evicts $p$.
- If ALG evicts $x \neq q$ from the cache, then ALG$_i$ also evicts $x$.

It is easy to see that until ALG evicts $q$ the caches of the two algorithms are of the form $X' \cup \{p\}$ and $X' \cup \{q\}$ respectively. Thus, the first case causes both caches to unify. From this moment on, both algorithms work the same and incur the same number of page faults. Hence, if the first case occurs before the next request to $p$ there is nothing left to show.

Upon the next request to $p$ (which as we recall once more must occur before the next request to $q$), ALG incurs a page fault, while ALG$_i$ has a cache hit. If ALG ejects $q$ from the cache, then the caches unify and we are done. Otherwise, the next request to $q$ causes ALG$_i$ to have a page fault whereas ALG has a cache-hit. At this moment ALG$_i$ ejects $q$ and both caches unify again. Clearly, the number of page faults of ALG$_i$ is at most that of ALG.

Corollary 7.2 LFD is an optimal offline algorithm for paging.

Proof: Let OPT be an optimal offline algorithm and consider any request sequence $\sigma = r_1, \ldots, r_n$. Lemma 7.1 gives us an algorithm OPT$_i$ with $OPT_i(\sigma) \leq OPT(\sigma)$. We re-apply the Lemma with $i = 2$ to OPT$_i$ and obtain OPT$_2$ with $OPT_2(\sigma) \leq OPT_1(\sigma) \leq OPT(\sigma)$. Repeating this process yields OPT$_n$, which serves $\sigma$ just as LFD.

Lemma 7.3 Let $N = k + 1$. Then we have LFD(σ) ≤ $\lceil |\sigma| / k \rceil$, that is, LFD has at most one page fault for every $k$th request.

Proof: Suppose that LFD has a page fault on request $r_i$ and ejects page $p$ of its cache $X$. Since $N = k + 1$, the next page fault of LFD must be on a request to $p$. When this request occurs, due to the LFD-rule all $k - 1$ pages of $X \setminus \{p\}$ must have been requested in between.

7.2 Deterministic Algorithms

An online algorithm must decide which page to evict from the cache upon a page fault. We consider the following popular caching strategies:
FIFO (First-In/First-Out) Eject the page which has been in the cache for the longest amount of time.

LIFO (Last-In/First-Out) Eject the page which was brought into cache most recently.

LFU (Least-Frequently-Used) Eject a page which has been requested least frequently.

LRU (Least-Recently-Used) Eject a page whose last request is longest in the past.

We first address the competitiveness of LIFO.

Lemma 7.4 LIFO is not competitive.

Proof: Let \( p_1, \ldots, p_k \) be the initial cache content and \( p_{k+1} \) be an additional page. Fix \( \ell \in \mathbb{N} \) arbitrary and consider the following request sequence:

\[
\sigma = p_1, p_2, \ldots, p_k, (p_{k+1}, p_k)^\ell \\
= p_1, p_2, \ldots, p_k, (p_{k+1}, p_k), \ldots, (p_{k+1}, p_k)
\]

Starting at the \((k+1)\)st request, LIFO has a page fault on every request which means \( \text{LIFO}(\sigma) = 2\ell \). Clearly, \( \text{OPT}(\sigma) = 1 \). Since we can choose \( \ell \) arbitrarily large, there are no constants \( c \) and \( \alpha \) such that \( \text{LIFO}(\sigma) \leq c \text{OPT}(\sigma) + \alpha \) for every sequence. \( \square \)

The strategy LFU is not competitive either:

Lemma 7.5 LFU is not competitive.

Proof: Let \( p_1, \ldots, p_k \) be the initial cache content and \( p_{k+1} \) be an additional page. Fix \( \ell \in \mathbb{N} \) arbitrary and consider the request sequence:

\[
\sigma = p_1^\ell, \ldots, p_k^\ell, (p_{k+1}, p_k)^\ell.
\]

After the first \((k-1)\ell\) requests LFU incurs a page fault on every remaining request. Hence, \( \text{LFU}(\sigma) \geq 2(\ell - 1) \). On the other hand an optimal offline algorithm can evict \( p_1 \) upon the first request to \( p_{k+1} \) and thus serve the sequence with one page fault only. \( \square \)

7.2.1 Phase Partitions and Marking Algorithms

To analyze the performance of LRU and FIFO we use an important tool which is known as \( k \)-phase partition. Given a request sequence \( \sigma \), phase 0 of is defined to be the empty sequence. Phase \( i + 1 \) consists of the maximal subsequence of \( \sigma \) after phase \( i \) such that the phase contains requests to at most \( k \) different pages.

\[
\sigma = \underbrace{r_1, \ldots, r_j}_{k \text{ different pages}}, \underbrace{r_{j+1}, \ldots, r_{j+k'}}_{k \text{ different pages}}, \ldots, \underbrace{r_{j+k'}, \ldots, r_n}_{\leq k \text{ different pages}}
\]

The \( k \)-phase partition is uniquely defined and independent of any algorithm. Every phase, except for possibly the last phase, contains requests to exactly \( k \) different pages.

Given the \( k \)-phase partition we define a marking of all pages in the universe as follows. At the beginning of a phase, all pages are unmarked. During a phase, a page is marked upon the first request to it. Notice that the marking does not depend on any particular algorithm. We call an algorithm a marking algorithm if it never ejects a marked page.
Theorem 7.6 Any marking algorithm is $k$-competitive for the paging problem with cache size $k$.

**Proof:** Let ALG be a marking algorithm. We consider the $k$-phase partition of the input sequence $\sigma$. ALG incurs at most $k$ page faults in a phase: upon a fault to page $f$, the marked page is brought into the cache. Since ALG never evicts a marked page, it can incur at most $k$ page faults in a single phase.

We slightly modify our partition of the input sequence to show that an optimal offline algorithm must incur as many page faults as there are phases in the partition. Segment $i$ of the input sequence starts with the second request in phase $i$ and ends with the first request of phase $i+1$. The last request in segment $i$ is a request that has not been requested in phase $i$, since otherwise phase $i$ would not be maximal.

At the end of segment $i$, the cache of OPT contains the first request of phase $i+1$, say $p$. Until the end of phase $i$ there will be requests to $k$ other pages, and, until the end of the segment requests to $k$ other pages. This means that there will be requests to $k+1$ pages until the end of segment $i$. Since OPT can only keep $k$ pages in the cache, it must incur a page fault.

Lemma 7.7 LRU is a marking algorithm.

**Proof:** Suppose that LRU ejects a marked page $p$ during a phase on request $r_i$. Consider the first request $r_j$ ($j < i$) to $p$ during the phase which caused $p$ to be marked. At this point in time $p$ is the page in the cache of LRU whose last request has been most recently. Let $Y = X \cup \{p\}$ denote the cache contents of LRU after request $r_j$. In order to have LRU eject $p$, all other pages in $X$ must have been requested during the phase. Thus, there have been requests to all $k$ pages in $Y$ during the phase. Clearly, $r_i \notin Y$, since otherwise LRU would not incur a page fault on request $r_i$. But this means that during the phase $k+1$ different pages have been requested.

**Corollary 7.8** LRU is $k$-competitive for the paging problem with cache size $k$.

In contrast, FIFO is not a marking algorithm. Nevertheless, FIFO can be shown to be competitive by a slight modification of the proof of Theorem 7.6

Theorem 7.9 FIFO is $k$-competitive for the paging problem with cache size $k$.

**Proof:** See Exercise 7.1.

### 7.2.2 A Lower Bound for Deterministic Algorithms

Theorem 7.10 Let ALG be any deterministic online algorithm for the paging problem with cache size $k$. If ALG is $c$-competitive, then $c \geq k$.

**Proof:** Let $S = \{p_1, \ldots, p_k, p_{k+1}\}$ be an arbitrary $(k+1)$ element subset of pages. We use only requests to pages in $S$. Without loss of generality we assume that the initial cache contents is $\{p_1, \ldots, p_k\}$.

The request sequence is defined inductively such that ALG has a page fault on every single request. We start with $r_1 = p_{k+1}$. If upon request $r_i$ ALG ejects page $p$, then $r_{i+1} := p$. Clearly, $\text{ALG}(\sigma) = |\sigma|$. On the other hand, by Lemma 7.3 $\text{OPT}(\sigma) \leq |\sigma|/k$ which proves the claim.
7.3 Randomized Algorithms

In this section we present a randomized algorithm which beats the lower bound for deterministic algorithms stated in Theorem 7.10. More specifically, it achieves a competitive ratio of $2H_k$, where

$$H_k = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{k}$$

denotes the $k$th Harmonic number. Observe that $\ln k < H_k \leq 1 + \ln k$. Hence, randomization gives an exponential boost in competitiveness for the paging problem.

**RANDMARK** Initially, all pages are unmarked.

Upon a request to a page $p$ which is not in the cache, $p$ is brought marked into the cache. The page to be evicted is chosen uniform at random among all unmarked pages in the cache (if all pages in the cache are already marked, then all marks are erased first).

Upon a request to a page $p$ which is already in the cache, $p$ is marked.

**Theorem 7.11** RANDMARK is $2H_k$-competitive against an oblivious adversary.

**Proof:** We can assume that RANDMARK and the adversary start with the same cache contents. For the analysis we use again the $k$-phase partition. Observe that RANDMARK never evicts a marked page and that a marked page remains in the cache until the end of the phase. Hence, RANDMARK deserves its name (it is a randomized marking algorithm). If $P_i$ denotes the pages requested in phase $i$, then the cache contents of RANDMARK at the end of the phase is exactly $P_i$.

We call a page that is requested during a phase and which is not in the cache of RANDMARK at the beginning of the phase a new page. All other pages requested during a phase will be called old pages.

Consider an arbitrary phase $i$ and denote by $m_i$ the number of new pages in the phase. RANDMARK incurs a page fault on every new page. Notice that RANDMARK has exactly $m_i$ faults on new pages since a marked page is never evicted until the end of the phase. We now compute the expected number of faults on old pages.

There will requests to at most $k - m_i$ different old pages in a phase (the last phase might not be complete). Let $\ell$ denote the number of new pages requested before the $j$th request to an old page. When the $j$th old page is requested, RANDMARK has evicted $k - (j - 1)$ unmarked old pages from the cache and these have been chosen uniformly at random. Hence, the probability that the $j$th page is not in the cache is $\ell / (k - (j - 1)) \leq m_i / (k - j + 1)$. Hence, the number $X_i$ of page faults in phase $i$ satisfies:

$$E[X_i] \leq m_i + \sum_{j=1}^{k-m_i} \frac{m_i}{k - j + 1} = m_i(H_k - H_{m_i} + 1) \leq m_i H_k.$$

By linearity of expectation $E[RANDMARK] = \sum_i E[X_i]$.

We complete the proof by showing that $OPT(\sigma) \geq \sum_i m_i / 2$. Together phase $i - 1$ and phase $i$ contain requests to at least $k + m_i$ different pages. Since OPT can keep only $k$ pages in the cache, it will incur at least $k + m_i$ page faults during these two phases. We thus get $OPT(\sigma) \geq \sum_i m_{2i}$ and $OPT(\sigma) \geq \sum_i m_{2i+1}$, which amounts to

$$OPT(\sigma) \geq \frac{1}{2} \left( \sum_i m_{2i} + \sum_i m_{2i+1} \right) = \sum_i \frac{m_i}{2}.$$

This proves the claim.
We now address the question how well randomized algorithms can perform.

**Theorem 7.12** Any randomized algorithm for the paging problem with cache size $k$ has competitive ratio at least $H_k$ against an oblivious adversary.

**Proof:** Our construction uses a subset of the pages $P$ of size $k+1$ consisting of the initial cache contents and one additional page. Let $\rho$ be a distribution over the input sequences with the property that the $i$th request is drawn uniformly at random from $P$ independent of all previous requests.

Clearly, any deterministic paging algorithm faults on each request with probability $1/(k+1)$. Hence, we have

$$\mathbb{E}_\rho [\text{ALG}(\sigma^n)] = \frac{n}{k+1}. \quad (7.1)$$

where $\sigma^n$ denotes request sequences of length $n$. By Yao’s Principle the number

$$r := \lim_{n \to \infty} \frac{n}{(k+1)\mathbb{E}_\rho [\text{OPT}(\sigma^n)]}$$

is a lower bound on the competitive ratio of any randomized algorithm against an oblivious adversary. Thus, our goal is to show that $r \geq H_k$. The desired inequality will follow if we can show that

$$\lim_{n \to \infty} \frac{n}{\mathbb{E}_\rho [\text{OPT}(\sigma^n)]} = (k+1)H_k. \quad (7.2)$$

We have a closer look at the optimal offline algorithm. We partition the sequence $\sigma^n$ into stochastic phases as follows: the partition is like the $k$-phase partition with the sole difference that the starts/ends of the phases are given by random variables.

Let $X_i, i = 1, 2, \ldots$ be the sequence of random variables where $X_i$ denotes the number of requests in phase $i$. Notice that the $X_i$ are all independent and uniformly distributed. The $j$th phase starts with request $r_{S_{j-1} + 1}$, where $S_j := \sum_{i=1}^{j-1} X_j$. Let us denote by $N(n)$ the number of complete phases:

$$N(n) := \max \{ j : S_j \leq n \}.$$

Our proof proceeds in three steps:

(i) We show that

$$\text{OPT}(\sigma^n) \leq N(n) + 1. \quad (7.3)$$

(ii) We prove that

$$\lim_{n \to \infty} \frac{n}{\mathbb{E}_\rho [N(n)]} = \mathbb{E}_\rho [X_i]. \quad (7.4)$$

(iii) We establish that the expected value $\mathbb{E}[X_i]$ satisfies:

$$\mathbb{E}_\rho [X_i] = (k+1)H_k - 1. \quad (7.5)$$

The inequalities \(7.3\)–\(7.5\) then imply \(7.2\).

(i) At the end of phase $j$ the cache of the optimal offline algorithm $\text{OPT}$ consists of all pages consisted in the phase (with the possible exception of the last phase which might be incomplete). Phase $j + 1$ starts with a page fault. If $\text{OPT}$ now removes the page from the cache which is requested in the first request of phase $j + 2$, then each phase will be served with at most one page fault (this argument heavily uses the fact that we are working on a subset of $k + 1$ pages).
(ii) The family of random variables \( \{ N(n) : n \in \mathbb{N} \} \) forms a renewal process. Equation (7.4) is a consequence of the Renewal Theorem (see [5, Appendix E]).

(iii) Our problem is an application of the coupon collector’s problem (see Exercise 7.2). We are given \( k + 1 \) coupons (corresponding to the pages in \( P \)) and a phase ends one step, before we have collected all \( k + 1 \) coupons. Thus, the expected value \( \mathbb{E}[X_i] \) is one less than the expected number of rounds in the coupon collectors problem, which by Exercise 7.2 is \( (k + 1)H_k \)

\[ \square \]

### 7.4 The \( k \)-Server Problem

The famous \( k \)-server problem is a natural generalization of the paging problem. Let \((X, d)\) be a metric space, that is, a set \( X \) endowed with a metric \( d \) satisfying:

\[
\begin{align*}
d(x, y) &= 0 \iff x = y \\
d(x, y) &= d(y, x) \\
d(x, z) &\leq d(x, y) + d(y, z).
\end{align*}
\]

In the \( k \)-server problem an algorithm moves \( k \) mobile servers in the metric space. The algorithm gets a sequence \( \sigma = r_1, \ldots, r_n \) of requests, where each request \( r_i \) is a point in the metric space \( X \). We say that a request \( r_i \) is answered, if a server is placed on \( r_i \). The algorithm must answer all requests in \( \sigma \) by moving the servers in the metric space. His cost \( \text{ALG}(\sigma) \) is the total distance travelled by its servers.

![Figure 7.1: When request \( r_i \) is issued, one server must be moved to \( r_i \). Only then the next request \( r_{i+1} \) becomes known to an online algorithm.](image1)

The paging problem is the special case of the \( k \)-server problem when \( d(x, y) = 1 \) for all \( x, y \in X \). The \( k \) servers represent the \( k \) pages in cache, \( N = |X| \) is the total number of pages in the system.

![Figure 7.2: Modelling the paging problem with \( N = 5 \) and \( k = 2 \) as a \( k \)-server problem](image2)
This observation already shows that for some metric spaces there can be no deterministic algorithm for the k-server problem with competitive ratio \( c < k \). We will show now that this result holds true for any metric space with at least \( k + 1 \) points.

We call an algorithm lazy, if at any request it moves at most one server, and this only if the point requested is not already occupied by one of its servers.

**Lemma 7.13** Let \( \text{ALG} \) be any algorithm for the k-server problem. There is a lazy algorithm \( \text{ALG}' \) such that \( \text{ALG}'(\sigma) \leq \text{ALG}(\sigma) \) for any request sequence \( \sigma \). If \( \text{ALG} \) is an online algorithm, then \( \text{ALG}' \) is also an online algorithm.

**Proof:** The proof is obtained by a simple induction in conjunction with the triangle inequality.

The modified algorithm \( \text{ALG}' \) works as follows: suppose that \( \text{ALG} \) is lazy until request \( i \) and then unnecessarily moves a server \( s \) from \( x \) to \( y \). The cost incurred is \( d(x, y) \). The new algorithm \( \text{ALG}' \) works just as \( \text{ALG} \) until request \( i \) but then skips the movement of \( s \). Let \( z \) be the point of the next request served by the server \( s \) of \( \text{ALG} \). Then, \( \text{ALG}' \) directly moves the server \( s \) from \( x \) to \( z \). Since \( d(x, z) \leq d(x, y) + d(y, z) \) the cost of \( \text{ALG}' \) is not larger than the one incurred by \( \text{ALG} \).

Lemma 7.13 allows us to restrict ourselves to lazy online algorithm when proving lower bounds.

**Theorem 7.14** Let \( M = (X, d) \) be any metric space with \( |X| \geq k + 1 \). Then, any deterministic \( c \)-competitive algorithm for the k-server problem in \( M \) satisfies \( c \geq k \).

**Proof:** Let \( \text{ALG} \) be any deterministic \( c \)-competitive algorithm. By Lemma 7.13 we can assume that \( \text{ALG} \) is lazy. We show that there are arbitrary long request sequences such that \( \text{ALG}(\sigma) \geq k \text{OPT}(\sigma) \).

We construct \( \sigma \) and \( k \) algorithms \( B_1, \ldots, B_k \), such that \( \text{ALG}(\sigma) = \sum_{j=1}^{k} B_j(\sigma) \). By averaging there must be a \( B_{j_0} \) with \( B_{j_0}(\sigma) \leq k \text{ALG}(\sigma) \).

Let \( S \) be the set of points occupied initially by \( \text{ALG}' \)'s servers plus an additional point in \( X \). We can assume that \( |S| = k + 1 \). Our sequences only use requests to points in \( S \).

The construction of the sequence \( \sigma = r_1, r_2, \ldots \) is inductively. The first request \( r_1 \) will be to that point in \( S \) where \( \text{ALG} \) does not have a server. Suppose that at request \( r_i \) \( \text{ALG} \) moves a server from \( x \) to \( r_i \). Then, the next request \( r_{i+1} \) will be to \( x \). The cost of \( \text{ALG} \) on \( \sigma \) is thus

\[
\text{ALG}(\sigma) = \sum_{i=1}^{n} d(r_{i+1}, r_i) = \sum_{i=1}^{n} d(r_i, r_{i+1}).
\]  

Let \( x_1, \ldots, x_k \) be the points in \( S \) initially occupied by the servers operated by \( \text{ALG} \). For \( j = 1, \ldots, k \) algorithm \( B_j \) starts with servers in all points except for \( x_j \). If at some point in time a point \( r_i \) is requested where \( B_j \) does not have a server, it moves a server from \( r_{i-1} \) to \( r_i \).

Denote by \( S_j \) the points where \( B_j \) has its servers. Initially, all \( S_j \) are different. Suppose that these sets are still different until request \( r_1 \). We show that they will still all be different after the request. A set \( S_j \) only changes, if \( r_i \not\in S_j \), since only in this case \( B_j \) moves a server. Thus, if two sets \( S_j \) both contain \( r_i \) or both do not contain \( r_i \), they will still be different after \( r_i \).

It remains to treat the case that \( r_i \in S_j \) but \( r_i \not\in S_k \). Since all algorithms served request \( r_{i-1} \), they all have a server on \( r_{i-1} \). Hence, after \( r_i \), \( S_j \) will still contain \( r_{i-1} \) while
7.4 The $k$-Server Problem

$S_k$ will not (since a server has been moved from $r_{i-1}$ to $r_i$ to serve the request). Hence, even in this case, the sets $S_j$ will remain different.

We have shown that all $S_j$ are different throughout the whole request sequence. This means, that upon request $r_i$ exactly one of the algorithms must move a server at cost $d(r_{i-1}, r_i)$. Thus, the total cost of all $B_j$ combined is

$$\sum_{j=1}^{k} B_j(\sigma) = \sum_{i=2}^{n} d(r_{i-1}, r_i) = \sum_{i=1}^{n-1} d(r_i, r_{i+1}).$$

Up to an additive constant, this is the cost of ALG as shown in (7.6).

The famous $k$-server conjecture states that the lower bound from the previous theorem can in fact be obtained:

**Conjecture 7.15 ($k$-Server Conjecture)** For any metric space there is a $k$-competitive algorithm for the $k$-server problem.

Until today, this conjecture remains unproven. In fact, it is not a priori clear that there should be an algorithm whose competitive ratio solely depends on $k$ and not on the number of points in $X$. The first extremely important result was achieved by Fiat et al. [9], who found a $O((k!)^3)$-competitive algorithm. The most celebrated result in online computation to date is the result of Koutsoupias and Papadimitriou [16] who almost settled the $k$-server conjecture and showed that the »work function algorithm« achieves a competitive ratio of $(2k - 1)$.

While the proof of Koutsoupias and Papadimitriou [16] presents some clever techniques it is simply too long for these lecture notes. Hence, we settle for an easier case, the $k$-server problem on the real line $\mathbb{R}$ with the Euclidean metric $d(x, y) = |x - y|$.

**Algorithm DC (Double Coverage)**

If request $r_i$ is outside of the convex hull of the servers, then serve the request with the closest server.

Otherwise, the request is between two servers. Move both servers at the same speed towards $r_i$ until the first one reaches $r_i$ (if initially two servers occupy the same point, then an arbitrary one is chosen).

Algorithm DC is not a lazy algorithm. However, we can easily make it lazy by the techniques of Lemma 7.13. For the analysis, the non-lazy version given above turns out to be more appropriate.

**Theorem 7.16** DC is $k$-competitive for the $k$-server problem in $\mathbb{R}$.

**Proof:** Our proof uses a potential function argument similar to the one for the list accessing problem in Section 6.2. Define $M_i$ to be the minimum weight matching between the servers of DC and OPT after request $r_i$. We also let $S_i$ be the sum of all distances between the servers of DC at this point in time. Our potential is as follows:

$$\Phi_i := k \cdot M_i + S_i.$$ 

Clearly, $\Phi_i \geq 0$. We imagine that at request $r_i$ first OPT moves and then DC.

**Claim 7.17** (i) If OPT moves a server by $\delta$ units, then the potential increases by at most $k\delta$. 
The Paging Problem

(ii) If DC moves two servers by a total amount of \( \delta \), then the potential decreases by at least \( \delta \).

Before we prove Claim 7.17, we show how it implies the Theorem. We consider the amortized cost \( a_i \) incurred by DC for request \( r_i \), defined by

\[
a_i := DC(r_i) + \Phi_i - \Phi_{i-1}.
\]

As in Section 6.2, it suffices to show that \( a_i \leq k \cdot \text{OPT}(r_i) \), since then

\[
DC(\sigma) \leq \Phi_0 + \sum_{i=1}^{n} a_i \leq \Phi_0 + \sum_{i=1}^{n} k \cdot \text{OPT}(r_i) = k \cdot \text{OPT}(\sigma) + \Phi_0.
\]

To see that \( a_i \leq k \cdot \text{OPT}(r_i) \) note that from Properties 7.17 and 7.17 we have

\[
\Phi_i - \Phi_{i-1} \leq k \cdot \text{OPT}(r_i) - DC(r_i).
\]

It remains to prove Claim 7.17. If OPT moves a server by \( \delta \) units, then the matching component of the potential can increase by at most \( k \delta \) (just keep the minimum cost matching, one of the edge weights increases by \( \delta \)). Since the sum component \( S_i \) remains fixed, this implies 7.17.

To show 7.17, we distinguish between the two different cases which determine how DC reacts to a request. If \( r_i \) is outside of the convex hull of DC’s servers, then DC just moves one server \( s \), which is either the far most left or the far most right server. Moving \( s \) by \( \delta \) units increases the distance to each of the \( k-1 \) remaining servers by \( \delta \), such that \( S_i \) increases by \( (k-1)\delta \). On the other hand, it is easy to see that in a minimum cost matching \( s \) must be matched to one of OPT’s servers who is already at \( r_i \). Hence, moving \( s \) closer to its matching partner decreases \( M_i \) by at least \( \delta \). Hence, we get a net decrease in potential of \( (k-1)\delta - k\delta = \delta \) as claimed.

If \( r_i \) is between two servers \( s \) and \( s' \), then both of them will be moved by \( \delta/2 \), if \( \delta \) is the total amount of movement by DC in reaction to request \( r_i \). Again, it is straightforward to see that one of the servers \( s \) and \( s' \) must be matched to OPT’s server that is already at \( r_i \). Hence, one server gets \( \delta/2 \) closer to its partner, while the other server moves at most \( \delta/2 \) away from its matching partner. Thus, the matching component of the potential does not increase. Consider now \( S_i \). For any server \( s'' \neq s, s' \) the distance \( d(s'', s) + d(s'', s') \) remains constant: one term increases by \( \delta/2 \), the other decreases by \( \delta/2 \). However, the distance between \( s \) and \( s' \) decreases by a total amount of \( \delta/2 + \delta/2 = \delta \), so that \( S_i \) decreases by \( \delta \). Hence, also in the second case \( \Phi \) must decrease by at least \( \delta \). This completes the proof of Claim 7.17.

7.5 Exercises

Exercise 7.1
Prove Theorem 7.9

Exercise 7.2
In the coupon collector’s problem, there are \( n \) types of coupons and at each trial a coupon is chosen at random. Each random coupon is equally likely to be any of the \( n \) types, and the random choice of the coupons are mutually independent. Let \( X \) be a random variable defined to be the number of trials required to collect at least one of each type of coupon. Show that \( \mathbb{E}[X] = nH_n \).

Hint: Let \( C_1, \ldots, C_X \) denote the sequence of trials, where \( C_i \in \{1, \ldots, n\} \) denotes the type of coupon drawn in the \( i \)th trial. Call the \( i \)th trial a success, if the type \( C_i \) was not drawn.
in any of the first $i - 1$ selections. Divide the sequence into epochs, where epoch $i$ begins with the trial following the $i$th success. Consider the random variables $X_i$, $0 \leq i \leq n - 1$, where $X_i$ denotes the number of trials in the $i$th epoch.

**Exercise 7.3 (Algorithms for the $k$-server problem)**

Show that the online-algorithm which always moves a closest server to the requested point is not competitive for the $k$-server problem.

**Exercise 7.4 (Cow path problem)**

A cow is standing in front of a fence with green yummy grassland behind. You understand that the cow is desperately looking for the hole in the fence. Unfortunately, it does not know which way to go: left or right? What would be the best strategy for the hungry cow in order to find the way through the fence as fast as possible?

This problem can be modeled as the search for an unknown point $a \in \mathbb{R}$ starting from the origin $0$. The optimal offline strategy knows the place $a$ of the hole in the fence and, thus, the value of the optimal solution (move straight from the origin to $a$) is simply the distance $|a|$.

(a) Does there exist a strictly $c$-competitive algorithm?

(b) Assume that $|a| \geq 1$. Try to construct an $9$-competitive algorithm for that problem.

Hint: Consider an algorithm that moves first $\alpha > 1$ units of length to the right, then goes back to the origin, from where it heads $\alpha^2$ units of length to the left. The $i$th turning point of that algorithm is $(-1)^{i+1} \alpha^i$.

What can you say about the competitiveness of that algorithm? How would you choose $\alpha$?

**Exercise 7.5 (Online graph matching)**

Consider the following online variant of the matching Problem. Given is a bipartite graph $G = (H \cup D, R)$, i.e., each directed edge $r \in R$ is of the form $r = (h, d)$ where $h \in H$ and $d \in D$ and $H \cap D = \emptyset$.

Suppose $G = (V, E)$ is an undirected graph. A matching is a subset $M \subseteq E$ such that $M$ contains no two incident edges. In a maximum matching it is not possible to add an edge without destroying the matching property. A matching $M$ is perfect if each node in $V$ is incident to an edge in $M$.

The dating service Online-Matching received data from $n$ men $H = \{h_1, \ldots, h_n\}$ who are interested in a wonderful lady. The service organizes a dance party where these men can find their love. Therefore invitations has been sent to $n$ promising women $D = \{d_1, \ldots, d_n\}$. With the invitation they received an overview of all interested men. Each lady creates a list of her preferred men. At the entrance to the party each girl is assigned a dance partner from her list. If there is no man from a girls list left without a partner, then she is paid some financial compensation and is sent home. Of course, the goal of the dating service is to partner up as many people as possible.

This problem can be modeled as an online version of the problem of finding a perfect matching. We are given a bipartite graph $G = (H \cup D, E)$ with $2n$ nodes, and we assume that $G$ has a perfect matching.

An online algorithm knows all men $H$ from the beginning. A request $r_i$ consists of a neighborhood $N(d_i) = \{h \in H : (h, d_i) \in E\}$ of a woman-node $d_i$. The online algorithm has to decide upon the arrival of the request to which man in $N(d_i)$ this girl should be assigned to (if possible). The sequence $\sigma = r_1, \ldots, r_n$ consists of a permutation of ladies and the objective is to obtain as many couples as possible.
Consider, now, the following very simple algorithm: as soon as a lady arrives she is assign to any still single guy who is on her preference list. If there is no guy left then she is sent home.

(a) Prove that this algorithm is 2-competitive.

(Hint: Assume that $M$ is a maximum matching with $|M| < n/2$. Denote by $H'$ the set of men who got a partner through $M$. Then, $|H'| < n/2$. Make use of the fact that $G$ has a perfect matching.)

(b) Show that every deterministic online algorithm (and even each randomized online algorithm against an adaptive offline adversary) has a competitive ratio of no less than 2 for the problem above.
8.1 Online Dial-a-Ride Problems

Let $M = (X, d)$ be a metric space with distinguished origin $o \in X$. We assume that $M$ is “connected and smooth” in the following sense: for all pairs $(x, y)$ of points from $M$, there is a rectifiable path $\gamma : [0, 1] \rightarrow X$ in $X$ with $\gamma(0) = x$ and $\gamma(1) = y$ of length $d(x, y)$ (see e.g. [2]). Examples of metric spaces that satisfy the above condition are the Euclidean space $\mathbb{R}^p$ and a metric space induced by an undirected connected edge-weighted graph.

An instance of the basic online dial-a-ride problem $OLDARP$ in the metric space $M$ consists of a sequence $\sigma = r_1, \ldots, r_n$ of requests. Each request is a triple $r_j = (t_j, \alpha_j, \omega_j) \in \mathbb{R} \times X \times X$ with the following meaning: $t_j = t(r_j) \geq 0$ is a real number, the time where request $r_j$ is released (becomes known), and $\alpha_j = \alpha(r_j) \in X$ and $\omega_j = \omega(r_j) \in X$ are the source and destination, respectively, between which the object corresponding to request $r_j$ is to be transported.

It is assumed that the sequence $\sigma = r_1, \ldots, r_n$ of requests is given in order of non-decreasing release times, that is, $0 \leq t(r_1) \leq t(r_2) \leq \cdots \leq t(r_n)$. For a real number $t$ we denote by $\sigma_{\leq t}$ the subsequence of requests in $\sigma$ released up to and including time $t$. Similarly, $\sigma_{\leq t}$ and $\sigma_{< t}$ denote the subsequences of $\sigma$ consisting of those requests with release time exactly $t$ and strictly smaller than $t$, respectively.

A server is located at the origin $o \in X$ at time 0 and can move at constant unit speed. We will only consider the case where the server has capacity 1, i.e., it can carry at most one objects at a time. We do not allow preemption: once the server has picked up an object, it is not allowed to drop it at any other place than its destination.
An online algorithm for OLDARP does neither have information about the release time of the last request nor about the total number of requests. The online algorithm must determine the behavior of the server at a certain moment \( t \) of time as a function of all the requests released up to time \( t \) (and the current time \( t \)). In contrast, an offline algorithm has information about all requests in the whole sequence \( \sigma \) already at time 0.

A feasible online/offline solution, called transportation schedule, for a sequence \( \sigma \) is a sequence of moves for the server such that the following conditions are satisfied: (i) the server starts in the origin at time 0, (ii) each request in \( \sigma \) is served, but picked up not earlier than the time it is released. If additionally (iii) the server ends its work at the origin, then the transportation schedule is called closed. Depending on the specific variant of OLDARP only closed schedules may be feasible.

Given an objective function \( C \), the problem \( C \)-OLDARP consists of finding a feasible schedule \( S^* \) minimizing \( C(S^*) \). The problem \( C \)-OLDARP can be cast into the framework of request-answer games. We refer to [17] for details. For the purposes in these lecture notes, the more informal definition above will be sufficient.

### 8.2 Minimizing the Makespan

This section is dedicated to the problem \( C_{\text{max}}^o \)-OLDARP with the objective to minimize the makespan. The makespan \( C_{\text{max}}(S) \) of a transportation schedule \( S \) is defined to be the time when \( S \) is completed.

**Definition 8.1 (Online Dial-a-Ride Problem \( C_{\text{max}}^o \)-OLDARP)**

The problem \( C_{\text{max}}^o \)-OLDARP consists of finding a closed transportation schedule which starts at the origin and minimizes the closed makespan \( C_{\text{max}}^o \).

The \( C_{\text{max}}^o \)-OLDARP comprises the online traveling salesman problem (OLTSP) which was introduced in [2] as an online variant of the famous traveling salesman problem. In the OLTSP cities (requests) arrive online over time while the salesman is traveling. The requests are to be handled by a salesman-server that starts and ends his work at a designated origin. The cost of such a route is the time when the server has served the last request and has returned to the origin (if the server does not return to the origin at all, then the cost of such a route is defined to be infinity).

Notice that the OLTSP differs from its famous relative, the traveling salesman problem, in certain aspects: First, the cost of a feasible solution is not the length of the tour but the total travel-time needed by the server. The total travel time is obtained from the tour length plus the time where the server remains idle. Second, due to the online nature of the problem it may be unavoidable that a server reaches a certain point in the metric space more than once.
8.2 Minimizing the Makespan

Definition 8.2 (Online Traveling Salesman Problem (OLTSP))

The online traveling salesman problem (OLTSP) is the special case of the \( C_{max}^{\alpha} \)-OLDARP, when for each request \( r \) its source and destination coincide, that is, \( \alpha(r) = \omega(r) \) for all \( r \).

If \( \sigma = r_1, \ldots, r_n \) is a sequence of requests for the OLTSP we write briefly \( r_j = (t_j, \alpha_j) \) instead of \( r_j = (t_j, \alpha_j, \alpha_j) \). Observe that the capacity of the server is irrelevant in case of the OLTSP. We will investigate more aspects of the OLTSP in Section 8.4.1.

8.2.1 Lower Bounds

Ausiello et al. established the following lower bounds for the OLTSP:

**Theorem 8.3** ([2]) Any deterministic online algorithm for the OLTSP in general metric spaces has a competitive ratio greater or equal to 2. Any deterministic online algorithm for the OLTSP on the real line has competitive ratio at least \((9 + \sqrt{17})/8\). \( \square \)

Since \( C_{max}^{\alpha} \)-OLDARP generalizes the OLTSP, the above theorem yields as a corollary a lower bound of 2 on the competitive ratio of any deterministic algorithm for \( C_{max}^{\alpha} \)-OLDARP in general metric spaces.

**Theorem 8.4** Any deterministic online algorithm for the \( C_{max}^{\alpha} \)-OLDARP in general metric spaces has a competitive ratio greater or equal to 2. For the case of the real line, any deterministic algorithm has a competitive ratio at least \( c \geq 1 + \sqrt{2}/2 \).

**Proof:** As noted above, the general lower bound is an immediate consequence of Theorem 8.3. We now address the case of the real line (with server capacity equal to one). Suppose that ALG is a deterministic online algorithm with competitive ratio \( c \leq 1 + \sqrt{2}/2 \). We show that also \( c \geq 1 + \sqrt{2}/2 \), which proves the claim of the theorem.

At time 0, the algorithm ALG is faced with two requests \( r_1 = (0, o, 2) \) and \( r_2 = (0, 2, o) \). The optimal offline cost to serve these two requests is 4.

Figure 8.5: Lower bound construction in Theorem 8.4.
The server operated by ALG must start serving request $r_2$ at some time $2 \leq T \leq 4c - 2$, because otherwise ALG could not be $c$-competitive. At time $T$ the adversary issues another request $r_3 = (T, T, 2)$. Then $\text{OPT}(r_1, r_2, r_3) = 2T$. On the other hand, $\text{ALG}(r_1, r_2, r_3) \geq 3T + 2$. Thus, the competitive ratio $c$ of ALG satisfies

$$c \geq \frac{3T + 2}{2T} = \frac{3}{2} + \frac{1}{T} \geq \frac{3}{2} + \frac{1}{4c - 2}.$$ 

The smallest value $c \geq 1$ such that $c \geq 3/2 + 1/(4c - 2)$ is $c = 1 + \sqrt{2}/2$. This completes the proof. \hfill $\Box$

We continue with a lower bound for randomized algorithms. The lower bound will be shown for the OLTS on the real line endowed with the usual Euclidean metric.

**Theorem 8.5** Any randomized algorithm for the OLTS on the real line has competitive ratio greater or equal to $3/2$ against an oblivious adversary.

**Proof:** We use Yao’s Principle (Theorem 6.4 on page 80) as a tool for deriving the lower bound. No request will be released before time 1. At time 1 with probability $1/2$ there is a request at 1, and with probability $1/2$ a request at $-1$. This yields a probability distribution $X$ over the two request sequences $\sigma_1 = (1, 1)$ and $\sigma_2 = (1, -1)$.

Since $\text{OPT}(\sigma_1) = \text{OPT}(\sigma_2) = 2$ it follows that $\mathbb{E}_X[\text{OPT}(\sigma_x)] = 2$. We now calculate the expected cost of an arbitrary deterministic algorithm. Consider the deterministic online algorithm $\text{ALG}_y$ which has its server at position $y \in \mathbb{R}$ at time 1 (clearly, any deterministic online algorithm is of this form). With probability $1/2$, $y$ is on the same side of the origin as the request which is released at time 1, with probability $1/2$ the position $y$ and the request are on opposite sides of the origin. In the first case, $\text{ALG}_y(\sigma_x) \geq 1 + (2 - y)$ (starting at time 1 the server has to move to 1 and back to the origin which needs time at least $2 - y$). In the other case, $\text{ALG}_y(\sigma_x) \geq 1 + (2 + y)$. This yields

$$\mathbb{E}_X[\text{ALG}_y(\sigma_x)] = \frac{1}{2}(3 - y) + \frac{1}{2}(3 + y) = 3.$$

Hence $\mathbb{E}_X[\text{ALG}_y(\sigma_x)] \geq 3/2 \cdot \mathbb{E}_X[\text{OPT}(\sigma_x)]$ and the claimed lower bound follows by Yao’s Principle. \hfill $\Box$

**Corollary 8.6** Any randomized algorithm for the $C_{\text{max}}$-OLDARP on the real line has competitive ratio greater or equal to $3/2$ against an oblivious adversary. \hfill $\Box$

### 8.2.2 Two Simple Strategies

We now present and analyze two very natural online-strategies for $C_{\text{max}}$-OLDARP.
Algorithm 8.1 Algorithm REPLAN

As soon as a new request \( r_j \) arrives the server stops and replans: it computes a schedule with minimum length which starts at the current position of the server, takes care of all yet unserved requests (including those that are currently carried by the server), and ends at the origin. Then it continues using the new schedule.

Algorithm 8.2 Algorithm IGNORE

The server remains idle until the point in time \( t \) when the first requests become known. The algorithm then serves the requests released at time \( t \) immediately, following a shortest schedule \( S \) which starts and ends at the origin.

All requests that arrive during the time when the algorithm follows \( S \) are temporarily ignored. After \( S \) has been completed and the server is back in the origin, the algorithm computes a shortest schedule for all unserved requests and follows this schedule. Again, all new requests that arrive during the time that the server is following the schedule are temporarily ignored. A schedule for the ignored requests is computed as soon as the server has completed its current schedule.

The algorithm keeps on following schedules and temporarily ignoring requests in this way.

Both algorithms above repeatedly solve “offline instances” of the \( C_{\text{max}}^{\alpha \text{-OLDARP}} \). These offline instances have the property that all release times are no less than the current time. Thus, the corresponding offline problem is the following: given a number of transportation requests (with release times all zero), find a shortest transportation for them.

For a sequence \( \sigma \) of requests and a point \( x \) in the metric space \( M \), let \( L^*(t, x, \sigma) \) denote the length of a shortest schedule (i.e., the time difference between its completion time and the start time) which starts in \( x \) at time \( t \), serves all requests from \( \sigma \) (but not earlier than their release times) and ends in the origin.

Observation 8.7 The function \( L^* \) has the following properties:

1. \( L^*(t', x, \sigma) \leq L^*(t, x, \sigma) \) for all \( t' \geq t \);
2. \( L^*(t, x, \sigma) \leq d(x, y) + L^*(t, y, \sigma) \) for all \( t \geq 0 \) and all \( x, y \in X \);
3. \( \text{OPT}(\sigma) = L^*(0, o, \sigma) \);
4. \( \text{OPT}(\sigma) \geq L^*(t, o, \sigma) \) for any time \( t \geq 0 \).

We now derive another useful property of \( L^* \) for the special case that the server has unit-capacity. This result will be used in the proof of the competitiveness of the REPLAN-strategy.

Lemma 8.8 Let \( \sigma = r_1, \ldots, r_m \) be a sequence of requests for the \( C_{\text{max}}^{\alpha \text{-OLDARP}} \). Then for any \( t \geq t_m \) and any request \( r \) from \( \sigma \),

\[
L^*(t, \omega(r), \sigma\backslash \{r\}) \leq L^*(t, o, \sigma) - d(\alpha(r), \omega(r)) + d(\alpha(r), o).
\]

Here \( \sigma \backslash \{r\} \) denotes the sequence obtained from \( \sigma \) by deleting the request \( r \).

Proof: Consider a transportation schedule \( S^* \) which starts at the origin \( o \) at time \( t \), serves all requests in \( \sigma \) and has length \( L^*(t, o, \sigma) \). It suffices to construct another schedule \( S \)
which starts in \(\omega(r)\) no earlier than time \(t\), serves all requests in \(\sigma \setminus \{r\}\) and has length at most \(L^*(t, o, \sigma) - d(\alpha(r), \omega(r)) + d(\alpha(r), o)\).

Let \(S^*\) serve the requests in the order \(r_{j_1}, \ldots, r_{j_m}\) and let \(r = r_{j_k}\). Notice that if we start in \(\omega(r)\) at time \(t\) and serve the requests in the order

\[ r_{j_k+1}, \ldots, r_{j_m}, r_{j_1}, \ldots, r_{j_{k-1}} \]

and move back to the origin, we obtain a schedule \(S\) with the desired properties.

Let \(\sigma = r_1, \ldots, r_n\) be any request sequence for \(C^{\omega}_{\max}\)-OLDARP. Since the optimal offline algorithm can not serve the last request \(r_m = (t_m, \alpha_m, \omega_m)\) from \(\sigma\) before this request is released we get that

\[
\text{OPT}(\sigma) \geq \max \{ L^*(t, o, \sigma), t_m + d(\alpha_m, \omega_m) + d(\omega_m, o) \} \quad (8.1)
\]

for any \(t \geq 0\).

**Theorem 8.9** REPLAN is 5/2-competitive for the \(C^{\omega}_{\max}\)-OLDARP.

**Proof:** Let \(\sigma = r_1, \ldots, r_n\) be any sequence of requests. We distinguish between two cases depending on the current load of the REPLAN-server at the time \(t_m\), i.e., the time when the last request is released.

If the server is currently empty it recomputes an optimal schedule which starts at its current position, denoted by \(s(t_m)\), serves all unserved requests, and returns to the origin. This schedule has length at most \(L^*(t_m, s(t_m), \sigma) \leq d(o, s(t_m)) + L^*(t_m, o, \sigma)\). Thus,

\[
\text{REPLAN}(\sigma) \leq t_m + d(o, s(t_m)) + L^*(t_m, o, \sigma) \\
\leq t_m + d(o, s(t_m)) + \text{OPT}(\sigma) \quad \text{by (8.1)} \quad (8.2)
\]

Since the REPLAN server has traveled to position \(s(t_m)\) at time \(t_m\), there must be a request \(r \in \sigma\) where either \(d(o, \alpha(r)) \geq d(o, s(t_m))\) or \(d(o, \omega(r)) \geq d(o, s(t_m))\). By the triangle inequality this implies that the optimal offline server will have to travel at least twice the distance \(d(o, s(t_m))\) during its schedule. Thus, \(d(o, s(t_m)) \leq \text{OPT}(\sigma)/2\). Plugging this result into inequality (8.2) we get that the total time the REPLAN server needs is no more than 5/2 \(\text{OPT}(\sigma)\).

It remains the case that the server is serving a request \(r\) at the time \(t_m\) when the last request \(r_m\) becomes known. The time needed to complete the current move is \(d(s(t_m), \omega(r))\). A shortest schedule starting at \(\omega(r)\) serving all unserved requests has length at most \(L^*(t_m, \omega(r), \sigma \setminus \{r\})\). Thus, we have
Figure 8.8: Case 1: The server is empty at time $t_m$ when the last request is released.

Figure 8.9: $\text{OPT}(\sigma) \geq 2d(o, s(t_m))$

Figure 8.10: Case 2 (C=1): The server is serving request $r$.

Hence, inequality $8.2$ also holds in case that the server is carrying an object at time $t_m$. As argued above, $t_m + d(o, s(t_m)) + \text{OPT}(\sigma) \leq 5/2 \text{OPT}(\sigma)$. This completes the proof.

**Theorem 8.10** Algorithm $\text{IGNORE}$ is $5/2$-competitive for the $C_{\text{max}}$-OLDARP.
Proof: We consider again the point in time $t_m$ when the last request $r_m$ becomes known. If the \textsc{ignore}-server is currently idle at the origin $o$, then it completes its last schedule no later than time $t_m + L^*(t_m, o, \sigma_{=t_m})$, where $\sigma_{=t_m}$ is the set of requests released at time $t_m$. Since $L^*(t_m, o, \sigma_{=t_m}) \leq \text{OPT}(\sigma)$ and $\text{OPT}(\sigma) \geq t_m$, it follows that in this case \textsc{ignore} completes no later than time $2 \cdot \text{OPT}(\sigma)$.

It remains the case that at time $t_m$ the \textsc{ignore}-server is currently working on a schedule $S$ for a subset $\sigma_S$ of the requests. Let $t_S$ denote the starting time of this schedule. Thus, the \textsc{ignore}-server will complete $S$ at time $t_S + L^*(t_S, o, \sigma_S)$. Denote by $t_{\geq t_S}$ the set of requests presented after the \textsc{ignore}-server started with $S$ at time $t_S$. Notice that $t_{\geq t_S}$ is exactly the set of requests that are served by \textsc{ignore} in its last schedule. The \textsc{ignore}-server will complete its total service no later than time $t_S + L^*(t_S, o, \sigma_S) + L^*(t_m, o, \sigma_{\geq t_S})$.

Let $r_f \in \sigma_{\geq t_S}$ be the first request from $\sigma_{\geq t_S}$ served by \text{OPT}. Thus

$$\text{OPT}(\sigma) \geq t_f + L^*(t_f, o, \sigma_{\geq t_S}) \geq t_S + L^*(t_m, o, \sigma_{\geq t_S}).$$

Now, $L^*(t_m, o, \sigma_{\geq t_S}) \leq d(o, \sigma_f) + L^*(t_m, o, \sigma_{\geq t_S})$ and $L^*(t_S, o, \sigma_S) \leq \text{OPT}(\sigma)$. Therefore,

$$\text{IGNORE}(\sigma) \leq t_S + \text{OPT}(\sigma) + d(o, \sigma_f) + L^*(t_m, o, \sigma_{\geq t_S})$$

by (8.3)

$$\leq 2 \cdot \text{OPT}(\sigma) + d(o, \sigma_f)$$

This completes the proof. 

8.2.3 A Best-Possible Online-Algorithm

In this section we present and analyze our algorithm \textsc{smartstart} which achieves a best-possible competitive ratio of 2 (cf. the lower bound given in Theorem 8.3). The idea of the algorithm is basically to emulate the \textsc{ignore}-strategy but to make sure that each sub-transportation schedule is completed “not too late”: if a sub-schedule would take “too long” to complete then the algorithm waits for a specified amount of time. Intuitively this construction tries to avoid the worst-case situation for \textsc{ignore} where right after the algorithm starts a schedule a new request becomes known.

\textsc{smartstart} has a fixed “waiting scaling” parameter $\theta > 1$. From time to time the algorithm consults its “work-or-sleep” routine: this subroutine computes an (approximately) shortest schedule $S$ for all unserved requests, starting and ending in the origin. If this
schedule can be completed no later than time $\theta t$, i.e., if $t + l(S) \leq \theta t$, where $t$ is the current time and $l(S)$ denotes the length of the schedule $S$, the subroutine returns $(S, \text{work})$, otherwise it returns $(S, \text{sleep})$.

In the sequel it will be convenient to assume that the “work-or-sleep” subroutine uses a $\rho$-approximation algorithm for computing a schedule: the approximation algorithm always finds a schedule of length at most $\rho$ times the optimal one. While in online computation one is usually not interested in time complexity (and thus in view of competitive analysis we can assume that $\rho = 1$), employing a polynomial-time approximation algorithm will enable us to get a practical algorithm.

The server of algorithm SMARTSTART can assume three states:

**idle** In this case the server has served all known requests, is sitting in the origin and waiting for new requests to occur.

**sleeping** In this case the server is sitting at the origin and knows of some unserved requests but also knows that they take too long to serve (what “too long” means will be formalized in the algorithm below).

**working** In this state the algorithm (or rather the server operated by it) is following a computed schedule.

We now formalize the behavior of the algorithm by specifying how it reacts in each of the three states.

**Algorithm 8.3 Algorithm SMARTSTART**

If the algorithm is idle at time $T$ and new requests arrive, calls “work-or-sleep”. If the result is $(S, \text{work})$, the algorithm enters the working state where it follows schedule $S$. Otherwise the algorithm enters the sleeping state with wakeup time $t'$, where $t' \geq T$ is the earliest time such that $t' + l(S) \leq \theta t'$ and $l(S)$ denotes the length of the just computed schedule $S$, i.e., $t' = \min\{ t \geq T : t + l(S) \leq \theta t \}$.

In the sleeping state the algorithm simply does nothing until its wakeup time $t'$. At this time the algorithm reconsults the “work-or-sleep” subroutine. If the result is $(S, \text{work})$, then the algorithm enters the working state and follows $S$. Otherwise the algorithm continues to sleep with new wakeup time $\min\{ t \geq t' : t + l(S) \leq \theta t \}$.

In the working state, i.e., while the server is following a schedule, all new requests are (temporarily) ignored. As soon as the current schedule is completed the server either enters the idle-state (if there are no unserved requests) or it reconsults the “work-or-sleep” subroutine which determines the next state (sleeping or working).
Theorem 8.11 For all real numbers $\theta \geq \rho$ with $\theta > 1$, Algorithm SMARTSTART is $c$-competitive for the $C_{\max}^\sigma$-OLDRP with

$$c = \max \left\{ \theta, \rho \left( 1 + \frac{1}{\theta - 1} \right) \frac{\theta}{2} + \rho \right\}.$$ 

Moreover, the best possible choice of $\theta$ is $\frac{1}{2} \left( 1 + \sqrt{1 + 8\rho} \right)$ and yields a competitive ratio of $c(\rho) := \frac{1}{4} \left( 4\rho + 1 + \sqrt{1 + 8\rho} \right)$.

![Figure 8.13: Competitive ratio $c(\rho)$ of SMARTSTART for $\rho \geq 1$.](image)

**Proof:** Let $\sigma_{=t_m}$ be the set of requests released at time $t_m$, where $t_m$ denotes again the point in time when the last requests becomes known. We distinguish between different cases depending on the state of the SMARTSTART-server at time $t_m$:

**Case 1:** The server is idle.

In this case the algorithm consults its “work-or-sleep” routine which computes an approximately shortest schedule $S$ for the requests in $\sigma_{=t_m}$. The SMARTSTART-server will start its work at time $t' = \min \left\{ t \geq t_m : t + l(S) \leq \theta t \right\}$, where $l(S) \leq \rho L^*(t_m, o, \sigma_{=t_m})$ denotes the length of the schedule $S$.

If $t' = t_m$, then by construction the algorithm completes no later than time $\theta t_m \leq \theta \OPT(\sigma)$. Otherwise $t' > t_m$ and it follows that $t' + l(S) = \theta t'$. By the performance guarantee $\rho$ of the approximation algorithm employed in “work-or-sleep”, we have that $\OPT(\sigma) \geq l(S)/\rho = \frac{\theta - 1}{\rho}$. Thus, it follows that

$$\text{SMARTSTART}(\sigma) = t' + l(S) \leq \theta t' \leq \theta \cdot \frac{\rho \OPT(\sigma)}{\theta - 1} = \rho \left( 1 + \frac{1}{\theta - 1} \right) \OPT(\sigma).$$

**Case 2:** The server is sleeping.

Note that the wakeup time of the server is no later than $\min \left\{ t \geq t_m : t + l(S) \leq \theta t \right\}$, where $S$ is now a shortest schedule for all the requests in $\sigma$ not yet served by SMARTSTART at time $t_m$, and we can proceed as in Case 1.

**Case 3:** The algorithm is working.

If after completion of the current schedule the server enters the sleeping state, then the arguments given above establish that the completion time of the SMARTSTART-server does not exceed $\rho \left( 1 + \frac{1}{\theta - 1} \right) \OPT(\sigma)$.

The remaining case is that the SMARTSTART-server starts its final schedule $S'$ immediately after having completed $S$. Let $t_S$ be the time when the server started $S$ and denote by $\sigma_{\geq t_S}$ the set of requests presented after the server started $S$ at time $t_S$. Notice that $\sigma_{\geq t_S}$ is exactly the set of requests that are served by SMARTSTART in its last schedule $S'$.

$$\text{SMARTSTART}(\sigma) = t_S + l(S) + l(S').$$ (8.4)
Here, $l(S)$ and $l(S')$ denote the length of the schedule $S$ and $S'$, respectively. We have that
\[ t_S + l(S) \leq \theta t_S, \]
(8.5)
since the SMARTSTART only starts a schedule at some time $t$ if it can complete it no later than time $\theta t$. Let $r_f \in \sigma_{\geq t_S}$ be the first request from $\sigma_{\geq t_S}$ served by OPT.

Using the arguments given in the proof of Theorem 8.10 we conclude as in (8.3) that
\[ \text{OPT} \geq t_S + L^*(t_m, \alpha_f, \sigma_{\geq t_S}). \]
(8.6)

Moreover, since the tour of length $L^*(t_m, \alpha_f, \sigma_{\geq t_S})$ starts in $\alpha_f$ and returns to the origin, it follows from the triangle inequality that
\[ L^*(t_m, \alpha_f, \sigma_{\geq t_S}) \geq d(o, \alpha_f). \]
Thus, from (8.6) we get
\[ \text{OPT} \geq t_S + d(o, \alpha_f). \]
(8.7)

On the other hand
\[ l(S') \leq \rho (d(o, \alpha_f) + L^*(t_m, \alpha_f, \sigma_{\geq t_S})) \]
\[ \leq \rho (d(o, \alpha_f) + \text{OPT} - t_S) \]
by (8.6). \hspace{1cm} (8.8)

Using (8.5) and (8.8) in (8.4) and the assumption that $\theta \geq \rho$, we obtain
\[ \text{SMARTSTART} \leq \theta t_S + l(S') \]
\[ \leq (\theta - \rho) t_S + \rho d(o, \alpha_f) + \rho \text{OPT} \]
\[ \leq \theta \text{OPT} + (2\rho - \theta) d(o, \alpha_f) \]
by (8.7). \hspace{1cm} (8.9)

This completes the proof.

For “pure” competitive analysis we may assume that each schedule $S$ computed by “work-or-sleep” is in fact an optimal schedule, i.e., that $\rho = 1$. The best competitive ratio for SMARTSTART is then achieved for that value of $\theta$ where the three terms $\theta$, $1 + \frac{1}{\theta - 1}$ and $\frac{\theta}{2} + 1$ are equal. This is the case for $\theta = 2$ and yields a competitive ratio of 2. We thus obtain the following corollary.

**Corollary 8.12** For $\rho = 1$ and $\theta = 2$, Algorithm SMARTSTART is 2-competitive for the $C_{\max}^\infty$-OLDARP.
8.3 Minimizing the Sum of Completion Times

In this section we address a different objective function for the online dial-a-ride problem, which is motivated by the traveling repairman problem, a variant of the traveling salesman problem.

In the traveling repairman problem (TRP) a server must visit a set of $m$ points $p_1, \ldots, p_m$ in a metric space. The server starts in a designated point $0$ of the metric space, called the origin, and travels at most at unit speed. Given a tour through the $m$ points, the completion time $C_j$ of point $p_j$ is defined as the time traveled by the server on the tour until it reaches $p_j$ ($j = 1, \ldots, m$). Each point $p_j$ has a weight $w_j$, and the objective of the TRP is to find the tour that minimizes the total weighted completion time $\sum_{j=1}^m w_j C_j$. This objective is also referred to as the latency.

Consider the following online version of the TRP called the online traveling repairman problem (OLTRP). Requests for visits to points are released over time while the repairman (the server) is traveling. In the online setting the completion time of a request $r_j$ at point $p_j$ with release time $t_j$ is the first time at which the repairman visits $p_j$ after the release time $t_j$. The online model allows the server to wait. However, waiting yields an increase in the completion times of the points still to be served. Decisions are revocable as long as they have not been executed.

In the dial-a-ride generalization $\sum w_j C_j$-OLDARP (for “latency online dial-a-ride problem”) each request specifies a ride from one point in the metric space, its source, to another point, its destination. The server can serve only one ride at a time, and preemption of rides is not allowed: once a ride is started it has to be finished without interruption.

8.3.1 A Deterministic Algorithm

It is not clear how to construct a competitive algorithm for the $\sum w_j C_j$-OLDARP. Exercise 8.1 asks you to show that the “natural REPLAN-approach“ does not work. The approach we present in this section divides time into exponentially increasing intervals and computes transportation schedules for each interval separately. However, each of these subschedules is not chosen such as to minimize the overall objective function. The goal is to maximize the total weight of requests completed within this interval.
To justify the correctness of the algorithm first notice that \( \beta \geq 1 \) for any \( \alpha \geq 1 + \sqrt{2} \). Moreover, it holds that the transportation schedule \( S_i \) computed at time \( B_i \) can actually be finished before time \( \beta B_{i+1} \), the time when transportation schedule \( S_{i+1} \), computed at time \( B_{i+1} \), needs to be started: \( S_1 \) is finished latest at time \( B_1 + B_1 = 2B_1 \leq \frac{\alpha + 1}{\alpha} B_1 = \beta B_2 \) since \( \alpha \leq 3 \). For \( i \geq 2 \), schedule \( S_i \) is also finished in time: By condition (iii), \( l(S_i) \leq B_i + B_{i-1} = (1 + \frac{1}{\alpha})B_i \). Hence, schedule \( S_i \) is finished latest at time \( \beta B_i + (1 + \frac{1}{\alpha})B_i = \frac{\alpha + 1}{\alpha - 1} B_i = \beta B_{i+1} \).

**Lemma 8.13** Let \( R_i \) be the set of requests served by schedule \( S_i \), computed at time \( B_i \), \( i = 1, 2, \ldots \), and let \( R_i^\ast \) be the set of requests in the optimal offline solution which are completed in the time interval \([B_{i-1}, B_i]\). Then

\[
\sum_{i=1}^{k} w(R_i) \geq \sum_{i=1}^{k} w(R_i^\ast) \quad \text{for} \quad k = 1, 2, \ldots
\]

**Proof:** We first argue that for any \( k \geq 1 \) we can obtain from the optimal offline solution \( S^\ast \) a schedule \( S \) which starts at the origin, has length at most \( B_k \), ends with an empty server at a point with distance at most \( B_k \) from the origin, and which serves all requests in \( \bigcup_{i=1}^{k} R_i^\ast \).

Consider the optimal offline transportation schedule \( S^\ast \). Start at the origin and follow \( S^\ast \) for the first \( B_k \) time units with the modification that, if a request is picked up in \( S^\ast \) before time \( B_k \) but not delivered before time \( B_k \), omit this action. Observe that this implies that the server is empty at the end of this schedule. We thereby obtain a schedule \( S \) of length at most \( B_k \) which serves all requests in \( \bigcup_{i=1}^{k} R_i^\ast \). Since the server moves at unit speed, it follows that \( S \) ends at a point with distance at most \( B_k \) from the origin.

We now consider phase \( k \) and show that by the end of phase \( k \), at least requests of weight \( \sum_{i=1}^{k} w(R_i^\ast) \) have been scheduled by \( \text{INTERVAL}_\alpha \). If \( k = 1 \), the transportation schedule \( S \) obtained as outlined above satisfies already all conditions (i)–(iii) required by \( \text{INTERVAL}_\alpha \). If \( k \geq 2 \), then condition (i) might be violated, since \( S \) starts in the origin. However, we can obtain a new schedule \( S' \) from \( S \) starting at the endpoint \( x_{k-1} \) of the schedule from

---

**Algorithm 8.4** Algorithm \( \text{INTERVAL}_\alpha \)

**Phase 0:** In this phase the algorithm is initialized.

Set \( L \) to be the earliest time when a request could be completed by \( \text{OPT} \). We can assume that \( L > 0 \), since \( L = 0 \) means that there are requests released at time 0 with source and destination \( o \). These requests are served at no cost. For \( i = 0, 1, 2, \ldots \), define \( B_i := \alpha^{i-1} L \), where \( \alpha \in [1 + \sqrt{2}, 3] \) is fixed.

**Phase \( i \), for \( i = 1, 2, \ldots \):** At time \( B_i \) compute a transportation schedule \( S_i \) for the set of yet unserved requests released up to time \( B_i \) with the following properties:

(i) Schedule \( S_i \) starts at the endpoint \( x_{i-1} \) of schedule \( S_{i-1} \) (we set \( x_0 := o \)).

(ii) Schedule \( S_i \) ends at a point \( x_i \) with an empty server such that \( d(o, x_i) \leq B_i \).

(iii) The length of schedule \( S_i \), denoted by \( l(S_i) \), satisfies

\[
l(S_i) \leq \begin{cases} B_i & \text{if } i = 1, \\ B_i + B_{i-1} & \text{if } i \geq 2. \end{cases}
\]

(iv) The transportation schedule \( S_i \) maximizes the sum of the weights of requests served among all schedules satisfying (i)–(iii).

If \( i = 1 \), then follow \( S_1 \) starting at time \( B_1 \). If \( i \geq 2 \), follow \( S_i \) starting at time \( \beta B_i \) until \( \beta B_{i+1} \), where \( \beta := \frac{\alpha + 1}{\alpha (\alpha - 1)} \).
the previous phase, moving the empty server from $x_{k-1}$ to the origin and then following $S$. Since $d(B_{k-1}, o) \leq B_{k-1}$, the new schedule $S'$ has length at most $B_{k-1} + l(S) \leq B_{k-1} + B_k$ which means that it satisfies all the properties (i)–(iii) required by INTERVAL$_\alpha$.

Recall that schedule $S$ and thus also $S'$ serves all requests in $\bigcup_{i=1}^{k} R^*_i$. Possibly, some of the requests from $\bigcup_{i=1}^{k} R^*_i$ have already been served by INTERVAL$_\alpha$ in previous phases. As omitting requests can never increase the length of a transportation schedule, in phase $k$, INTERVAL$_\alpha$ can schedule at least all requests from $S_k$.

Consequently, the weight of all requests served in schedules $S_1, \ldots, S_k$ of INTERVAL$_\alpha$ is at least $w(\bigcup_{i=1}^{k} R^*_i) = \sum_{i=1}^{k} w(R^*_i)$ as claimed.

The previous lemma gives us the following bound on the number of phases that INTERVAL$_\alpha$ uses to process a given input sequence $\sigma$.

**Corollary 8.14** Suppose that the optimum offline schedule is completed in the interval $(B_{p-1}, B_p]$ for some $p \geq 1$. Then the number of phases of the Algorithm INTERVAL$_\alpha$ is at most $p$. Schedule $S_p$ computed at time $B_p$ by INTERVAL$_\alpha$ is completed no later than time $\beta B_{p+1}$.

**Proof:** By Lemma 8.13 the weight of all requests scheduled in the first $p$ phases equals the total weight of all requests. Hence all requests must be scheduled within the first $p$ phases. Since, by construction of INTERVAL$_\alpha$, schedule $S_p$ computed in phase $p$ completes by time $\beta B_{p+1}$, the claim follows.

To prove competitiveness of INTERVAL$_\alpha$ we need an elementary lemma which can be proven by induction.

**Lemma 8.15** Let $a_i, b_i \in \mathbb{R}_{\geq 0}$ for $i = 1, \ldots, p$, for which

(i) $\sum_{i=1}^{p} a_i = \sum_{i=1}^{p} b_i$;

(ii) $\sum_{j=1}^{p'} a_i \geq \sum_{j=1}^{p'} b_i$ for all $1 \leq p' \leq p$.

Then $\sum_{i=1}^{p} r_i a_i \leq \sum_{i=1}^{p} r_i b_i$ for any nondecreasing sequence $0 \leq r_1 \leq \cdots \leq r_p$.

**Proof:** See Exercise 8.2.

**Theorem 8.16** Algorithm INTERVAL$_\alpha$ is $\frac{\alpha(\alpha+1)}{\alpha-1}$-competitive for the $\sum_{j} w_j C_j$-OLDARP for any $\alpha \in [1 + \sqrt{2}, 3]$. For $\alpha = 1 + \sqrt{2}$, this yields a competitive ratio of $(1 + \sqrt{2})^2 < 5.8285$.

**Proof:** Let $\sigma = r_1, \ldots, r_n$ be any sequence of requests. By definition of INTERVAL$_\alpha$, each request served in schedule $S_i$ completes no later than time $\beta B_{i+1} = \frac{\alpha+1}{\alpha(\alpha-1)} B_{i+1}$.

Summing over all phases $1, \ldots, p$ yields

$$\text{INTERVAL}_\alpha(\sigma) \leq \frac{\alpha+1}{\alpha(\alpha-1)} \sum_{i=1}^{p} B_{i+1} w(R_i) = \alpha \cdot \frac{\alpha+1}{\alpha-1} \sum_{i=1}^{p} B_{i+1} w(R_i).$$

(8.9)

From Lemma 8.13 we know that $\sum_{i=1}^{p} w(R_i) \geq \sum_{i=1}^{p} w(R^*_i)$ for $k = 1, 2, \ldots$ and from Corollary 8.14 we know that $\sum_{i=1}^{k} w(R_i) = \sum_{i=1}^{k} w(R^*_i)$. Therefore, application of
Lemma 8.15 to the sequences \( a_i := w(R_i) \) and \( b_i := w(R_i^*) \) with the weighing sequence \( \tau_i := B_{i-1} \), \( i = 1, \ldots, p \), gives

\[
\alpha \cdot \frac{\alpha + 1}{\alpha - 1} \sum_{i=1}^{P} B_{i-1} w(R_i) \leq \alpha \cdot \frac{\alpha + 1}{\alpha - 1} \sum_{i=1}^{P} B_{i-1} w(R_i^*). \tag{8.10}
\]

Denote by \( C_j^* \) the completion time of request \( r_j \) in the optimal offline solution \( \text{OPT}(\sigma) \). For each request \( r_j \) denote by \( (B_{\phi_j}, B_{\phi_j+1}) \) the interval that contains \( C_j^* \). Then

\[
\alpha \cdot \frac{\alpha + 1}{\alpha - 1} \sum_{i=1}^{P} B_{i-1} w(R_i^*) = \alpha \cdot \frac{\alpha + 1}{\alpha - 1} \sum_{j=1}^{m} B_{\phi_j} w_j \leq \alpha \cdot \frac{\alpha + 1}{\alpha - 1} \sum_{j=1}^{m} w_j C_j^*. \tag{8.11}
\]

(8.9), (8.10), and (8.11) together yield

\[
\text{INTERVAL}_\alpha(\sigma) \leq \alpha \cdot \frac{\alpha + 1}{\alpha - 1} \text{OPT}(\sigma).
\]

The value \( \alpha = 1 + \sqrt{2} \) minimizes the function \( f(\alpha) := \frac{\alpha(\alpha + 1)}{\alpha - 1} \) in the interval \([1 + \sqrt{2}, 3]\), yielding \((1 + \sqrt{2})^2 < 5.8285\) as competitive ratio. \( \square \)

**Corollary 8.17** For \( \alpha = 1 + \sqrt{2} \), algorithm \( \text{INTERVAL}_\alpha \) is \((1 + \sqrt{2})^2\)-competitive for the OLTSP. \( \square \)

### 8.3.2 An Improved Randomized Algorithm

In this section we use randomization to improve the competitiveness result obtained in the previous section. At the beginning, \( \text{RANDINTERVAL}_\alpha \) chooses a random number \( \delta \in [0, 1] \) according to the uniform distribution. From this moment on, the algorithm is completely deterministic, working in the same way as the deterministic algorithm \( \text{INTERVAL}_\alpha \) presented in the previous section. For \( i \geq 0 \) define \( B_i^\prime := \alpha^{i-1+\delta} L \), where again \( L \) is the earliest time that a request could be completed by \( \text{OPT} \). As stated before in the case of \( \text{INTERVAL}_\alpha \) we can assume that \( L > 0 \).

The difference between \( \text{RANDINTERVAL}_\alpha \) and \( \text{INTERVAL}_\alpha \) is that all phases are defined using \( B_i^\prime := \alpha^{i-1+\delta} L \) instead of \( B_i := \alpha^{i-1} L \), \( i \geq 1 \). To justify the accuracy of \( \text{RANDINTERVAL}_\alpha \), note that in the correctness proof for the deterministic version, we only made use of the fact that \( B_{i+1} = \alpha B_i \) for \( i \geq 0 \). This also holds for the \( B_i^\prime \). Hence, any choice of the parameter \( \alpha \in [1 + \sqrt{2}, 3] \) yields a correct version of \( \text{RANDINTERVAL}_\alpha \). We will show later that the optimal choice for \( \text{RANDINTERVAL}_\alpha \) is \( \alpha = 3 \).

The proof of Lemma 8.13 also holds also with \( B_i \) replaced by \( B_i^\prime \) for each \( i \geq 0 \). We thus obtain the following lemma.

**Lemma 8.18** Let \( R_i \) be the set of requests scheduled in phase \( i \geq 1 \) of Algorithm \( \text{RANDINTERVAL}_\alpha \) and denote by \( R_i^* \) the set of requests that are completed by \( \text{OPT} \) in the time interval \([B_i^\prime-1, B_i^\prime]\). Then

\[
\sum_{i=1}^{k} w(R_i) \geq \sum_{i=1}^{k} w(R_i^*) \quad \text{for } k = 1, 2, \ldots.
\]

**Proof:** We only have to ensure that schedule \( S_1 \) is finished before time \( \beta B_i^\prime \). The rest of the proof is the same as that for Lemma 8.13. The proof for the first phase follows from the fact that \( \beta B_2^\prime - L = \left( \frac{\alpha + 1}{\alpha - 1} \right) \cdot \alpha^{1+\delta} - L > \alpha^{\delta} L = B_1^\prime \). \( \square \)
We can now use the proof of Theorem 8.16 with Lemma 8.13 replaced by Lemma 8.18. This enables us to conclude that for a sequence \( \sigma = r_1, \ldots, r_n \) of requests the expected objective function value of RANDINTERVAL\( \alpha \) satisfies

\[
E [\text{RANDINTERVAL}\alpha(\sigma)] \leq E [\alpha - 1 \sum_{j=1}^{m} B'_{r_j}] \cdot w_j = \alpha \cdot \sum_{j=1}^{m} w_j E [B'_{r_j}],
\]

where \( (B'_{r_j}, B'_{r_{j+1}}) \) is the interval containing the completion time \( C_j^* \) of request \( r_j \) in the optimal solution \( \text{OPT}(\sigma) \).

To prove a bound on the performance of RANDINTERVAL\( \alpha \) we compute \( E [B] \). Notice that \( B_{r_i} \) is the largest value \( k + \frac{L}{\alpha} \), \( k \in \mathbb{Z} \), which is strictly smaller than \( C_j^* \).

**Lemma 8.19** Let \( z \geq L \) and \( \delta \in (0, 1] \) be a random variable uniformly distributed on \((0, 1]\). Define \( B := \max\{ k + \frac{L}{\alpha} : \alpha k + \delta L < z \text{ and } k \in \mathbb{Z} \} \). Then, \( E [B] = \frac{\alpha - 1}{\alpha \ln \alpha} z \).

**Proof:** Suppose that \( \alpha k L < z < \alpha k + 1 \) for some \( k \geq 0 \). Observe that

\[
B = \begin{cases} 
\alpha k - 1 + \delta L & \text{if } \delta \geq \frac{\ln \alpha - \frac{z}{\alpha L}}{\alpha} \\
\alpha k + \delta L & \text{otherwise}.
\end{cases}
\]

Hence

\[
E [B] = \int_{0}^{\log_{\alpha} \frac{z}{\alpha L}} \alpha k + \delta L d\delta + \int_{\log_{\alpha} \frac{z}{\alpha L}}^{1} \alpha k - 1 + \delta L d\delta
\]

\[
= \alpha k L \left[ \frac{1}{\ln \alpha} \log_{\alpha} \frac{z}{\alpha L} \right]_{0}^{\log_{\alpha} \frac{z}{\alpha L}} + \alpha k - 1 \left[ \frac{1}{\ln \alpha} \log_{\alpha} \frac{z}{\alpha L} \right]_{0}^{1} = \frac{\alpha - 1}{\alpha \ln \alpha} z.
\]

This completes the proof. \( \square \)

From Lemma 8.19 we can conclude that \( E [B'_{r_i}] = \frac{\alpha - 1}{\alpha \ln \alpha} C_j^* \). Using this result in inequality (8.12), we obtain

\[
E [\text{RANDINTERVAL}\alpha(\sigma)] \leq \alpha \cdot \frac{\alpha - 1}{\alpha \ln \alpha} \cdot \sum_{j=1}^{m} w_j C_j^* = \frac{\alpha + 1}{\alpha \ln \alpha} \cdot \text{OPT}(\sigma).
\]

Minimizing the function \( g(\alpha) := \frac{\alpha + 1}{\alpha \ln \alpha} \) over the interval \([1 + \sqrt{2}, 3]\), we conclude that the best choice is \( \alpha = 3 \).

**Theorem 8.20** Algorithm RANDINTERVAL\( \alpha \) is \( \frac{\alpha + 1}{\ln \alpha} \)-competitive for the \( \sum w_j C_j \)-OLDARP against an oblivious adversary, where \( \alpha \in [1 + \sqrt{2}, 3] \). Choosing \( \alpha = 3 \) yields a competitive ratio of \( \frac{1}{\ln 3} < 3.6410 \) for RANDINTERVAL\( \alpha \) against an oblivious adversary. \( \square \)

**Corollary 8.21** For \( \alpha = 3 \), algorithm RANDINTERVAL\( \alpha \) is \( \frac{1}{\ln 3} \)-competitive for the OLTARP against an oblivious adversary. \( \square \)
8.4 Alternative Adversary Models

8.4.1 The Fair Adversary

We consider the OLTSP on $\mathbb{R}_{\geq 0}$ in case that the offline adversary is the conventional (omnipotent) opponent.

**Theorem 8.22** Let $\text{ALG}$ be any deterministic algorithm for OLTSP on $\mathbb{R}_{\geq 0}$. Then the competitive ratio of $\text{ALG}$ is at least $3/2$.

**Proof:** At time 0 the request $r_1 = (0, 1)$ is released. Let $T \geq 1$ be the time that the server operated by $\text{ALG}$ has served the request $r_1$ and returned to the origin $o$. If $T \geq 3$, then no further request is released and $\text{ALG}$ is no better than $3/2$-competitive since $\text{OPT}(r_1) = 2$. Thus, assume that $T < 3$.

![Figure 8.16: Lower bound construction of Theorem 8.22](image)

In this case the adversary releases a new request $r_2 = (T, T)$. Clearly, $\text{OPT}(r_1, r_2) = 2T$. On the other hand $\text{ALG}(r_1, r_2) \geq 3T$, yielding a competitive ratio of $3/2$.

The following simple strategy achieves a competitive ratio that matches this lower bound (as we will show below):

**Algorithm 8.5** Algorithm MRIN (“Move-Right-If-Necessary”)

If a new request is released and the request is to the right of the current position of the server operated by MRIN, then the MRIN-server starts to move right at unit speed. The server continues to move right as long as there are yet unserved requests to the right of the server. If there are no more unserved requests to the right, then the server moves towards the origin $o$ at unit speed.

**Theorem 8.23** MRIN is a $3/2$-competitive algorithm for the OLTSP on $\mathbb{R}_{\geq 0}$.

**Proof:** See Exercise 8.3

The adversary used in Theorem 8.22 abused his power in the sense that he moves to points where he knows a request will pop up without revealing the request to the online server before reaching the point.

The model of the *fair adversary* defined formally below allows improved competitive ratios for the OLTSP on $\mathbb{R}_{\geq 0}^+$. Under Recall that $\sigma_{<t}$ is the subsequence of $\sigma$ consisting of those requests with release time strictly smaller than $t$.

**Definition 8.24 (Fair Adversary)** An offline adversary for the OLTSP in the Euclidean space $(\mathbb{R}^n, ||.||)$ is fair, if at any moment $t$, the position of the server operated by the adversary is within the convex hull of the origin $o$ and the requested points from $\sigma_{<t}$.
In the special case of $\mathbb{R}_{\geq 0}$ a fair adversary must always keep its server in the interval $[0, F]$, where $F$ is the position of the request with the largest distance to the origin $o = 0$ among all requests released so far.

The following lower bound results show that the OLTSP against a fair adversary is still a non-trivial problem.

**Theorem 8.25** Let $\text{ALG}$ be any deterministic algorithm for OLTSP on $\mathbb{R}$. Then the competitive ratio of $\text{ALG}$ against a fair adversary is at least $(5 + \sqrt{57})/8$.

**Proof:** See [17, 4].

**Theorem 8.26** Let $\text{ALG}$ be any deterministic algorithm for OLTSP in $\mathbb{R}_{\geq 0}$. Then the competitive ratio of $\text{ALG}$ against a fair adversary is at least $(1 + \sqrt{17})/4$.

**Proof:** See [17, 4].

We now show that $\text{MRIN}$ performs better against the fair adversary.

**Theorem 8.27** Algorithm $\text{MRIN}$ is $4/3$-competitive for the OLTSP on $\mathbb{R}_{\geq 0}$ against a fair adversary.

**Proof:** We use induction on the number of requests in the sequence $\sigma$ to establish the claim. The claim clearly holds if $\sigma$ contains at most one request. The induction hypothesis states that the claim of the theorem holds for any sequence of $m - 1$ requests.

Let $\sigma = r_1, \ldots, r_n$ be any sequence of requests. We consider the time $t := t_m$ when the last set of requests $\sigma_{=t_m}$ is released. If $t = 0$, then the claim obviously holds, so we will
assume for the remainder of the proof that \( t > 0 \). Let \( r = (t, x) \) be that request of \( \sigma_{=t_m} \) which is furthest away from the origin.

In the sequel we denote by \( s(t) \) and \( s^*(t) \) the positions of the MRIN-server and the fair adversary server at time \( t \), respectively.

Let \( r_f = (t_f, f) \) be the furthest unserved request by MRIN of the subsequence \( \sigma_{<t} \) at time \( t \), that is, the unserved request from \( \sigma_{<t} \) most remote from the origin \( o \). Finally, let \( r_F = (t_F, F) \) be the furthest request in \( \sigma_{<t} \). Notice that by definition \( f \leq F \).

We distinguish three different cases depending on the position \( x \) of the request \( r \) relative to \( f \) and \( F \).

**Case 1: \( x \leq f \)**

Since the MRIN-server still has to travel to \( f \), all the requests in \( \sigma_{=t} \) will be served on the way back to the origin and the total completion time of the MRIN-server will not increase by releasing the requests \( \sigma_{=t} \). Since new requests can never decrease the optimal offline solution value, the claim follows from the induction hypothesis.

![Figure 8.18: Case 1: If \( x \leq f \), MRIN’s cost does not increase.](image)

**Case 2: \( f \leq x < F \)**

If \( s(t) \geq x \), again MRIN’s completion time does not increase compared to the situation before the requests in \( \sigma_{=t} \) were released, so we may assume that \( s(t) \leq x \).

![Figure 8.19: Case 2: \( f \leq x < F \) and \( s(t) \leq x \).](image)

The MRIN-server will now travel to point \( x \) which needs time \( d(s(t), x) \), and then return to the origin. Thus, \( \text{MRIN}(\sigma) = t + d(s(t), x) + x \). On the other hand \( \text{OPT}(\sigma) \geq t + x \). It follows that \[
\frac{\text{MRIN}(\sigma)}{\text{OPT}(\sigma)} \leq 1 + \frac{d(s(t), x)}{\text{OPT}(\sigma)} \tag{8.13}
\]

We now show that \( \text{OPT}(\sigma) \) is at least 3 times \( d(s(t), x) \), this will establish the claimed ratio of 4/3. Notice that \( f < F \) (since \( f \leq x < F \)) and the fact that \( f \) is the furthest unserved request at time \( t \) implies that the MRIN-server must have already visited \( F \) at time \( t \) (otherwise the furthest unserved request would be at \( F \) and not at \( f < F \)). Therefore, \( t \geq F + d(F, s(t)) \), and

\[ \text{OPT}(\sigma) \geq t + x \geq F + d(F, s(t)) + x. \tag{8.14} \]

Clearly, each of the terms on the right hand side of inequality (8.14) is at least \( d(s(t), x) \).

**Case 3: \( f \leq F \leq x \)**

First recall that the MRIN-server always moves to the right if there are yet unserved requests to the right of his present position.

Since the last request \( (t, x) \) is at least as far away from the origin as \( F \), the optimal offline server will only move left after it has served the furthest request in \( \sigma \), in this case at \( x \). In fact, the optimal fair offline strategy is as follows: as long as there are unserved requests to
the right of the server, move right, otherwise wait at the current position. As soon as the
last request \((t, x)\) has been released and the offline server has reached \(x\), it moves to the
origin and completes its work.

Hence, at any time in the interval \([0, t]\), the fair adversary’s server is to the right of the
MRIN-server or at the same position.

Because the offline server does not move left as long as there will be new requests released
to the right of its current position, the distance between the MRIN-server and the offline
server increases only if the offline server is waiting at some point. Let \(W^*(t)\) be the total
waiting time of the offline server at the moment \(t\) when the last request \(x\) is released. Then
we know that
\[
d(s(t), s^*(t)) \leq W^*(t). \tag{8.15}
\]
Moreover, the following relation between the current time and the waiting time holds:
\[
t = d(o, s^*(t)) + W^*(t). \tag{8.16}
\]
Since the adversary is fair, its position \(s^*(t)\) at time \(t\) can not be to the right of \(F\). Thus,
\[
d(s^*(t), x) = d(s^*(t), F) + d(F, x) \tag{8.17}
\]
which gives us
\[
\begin{align*}
\text{OPT}(\sigma) & \geq t + d(s^*(t), F) + d(F, x) + x \\
& = d(o, s^*(t)) + W^*(t) + d(s^*(t), F) + d(F, x) + x \\
& = W^*(t) + F + d(F, x) + x \\
& = W^*(t) + 2x \\
& \geq W^*(t) + 2d(s(t), s^*(t)) \\
& \geq 3d(s(t), s^*(t)) \tag{8.18}
\end{align*}
\]
At time \(t\) MRIN’s server has to move from its current position \(s(t)\) to \(x\) and from there to
move to the origin:
\[
\text{MRIN}(\sigma) = t + d(s(t), x) + x \\
= t + d(s(t), s^*(t)) + d(s^*(t), F) + d(F, x) + x.
\]
Hence,

$$\frac{\text{MRIN}(\sigma)}{\text{OPT}(\sigma)} = \frac{t + d(s^*(t), F) + d(F, x) + x}{\text{OPT}(\sigma)} + \frac{d(s(t), s^*(t))}{\text{OPT}(\sigma)}$$

$$\leq 1 + \frac{d(s(t), s^*(t))}{\text{OPT}(\sigma)} \quad \text{by (8.17)}$$

$$\leq \frac{4}{3} \quad \text{by (8.18)}.$$ 

This proves the claim. 

Can one improve the competitiveness of $4/3$ or is this already best possible? Let us think for a moment: The problem with Algorithm MRIN is that shortly after it starts to return towards the origin from the furthest previously unserved request, a new request to the right of its server is released. In this case the MRIN-server has to return to a position it just left. Algorithm WS presented below avoids this pitfall successfully.

**Algorithm 8.6** Algorithm WS (“Wait Smartly”)

The WS-server moves right if there are yet unserved requests to the right of its present position. Otherwise, it takes the following actions. Suppose it arrives at its present position $s(t)$ at time $t$.

1. Compute the optimal offline solution value $\text{OPT}(\sigma_{\leq t})$ for all requests released up to time $t$.
2. Determine a waiting time $W := \alpha \, \text{OPT}(\sigma_{\leq t}) - s(t) - t$, with $\alpha = (1 + \sqrt{17})/4$.
3. Wait at point $s(t)$ until time $t + W$ and then start to move back to the origin.

One can prove the following result about the competitiveness of the more clever strategy WS:

**Theorem 8.28** WS is $\alpha$-competitive for the OLTSp on $\mathbb{R}_{\geq 0}$ against a fair adversary with $\alpha = (1 + \sqrt{17})/4 < 1.2808$.

**Proof:** See [174].

### 8.4.2 The Non-Abusive Adversary

An objective function that is very interesting in view of applications is the maximum flow time, which is defined to be the maximum time span from the release time of a request until the time it is served. We denote the OLTSp with this objective function as the $F_{\text{max}}$-OLTSp. Unfortunately, there can be no competitive algorithm, not even deterministic for the $F_{\text{max}}$-OLTSp as very simple constructions show.

The following lower bound result shows that even the fairness restriction on the adversary introduced in Section 8.4.1 is still not strong enough to allow for competitive algorithms in the $F_{\text{max}}$-OLTSp.
Theorem 8.29 No randomized algorithm for the $F_{\text{max}}$-OLTSP on $\mathbb{R}$ can achieve a constant competitive ratio against an oblivious adversary. This result still holds, even if the adversary is fair, i.e., if at any moment in time $t$ the server operated by the adversary is within the convex hull of the origin and the requested points from $\sigma_{\leq t}$.

Proof: Let $\varepsilon > 0$ and $k \in \mathbb{N}$. We give two request sequences $\sigma_1 = (\varepsilon, \varepsilon, (2\varepsilon, 2\varepsilon), \ldots, (ke, ke), (T, 0))$ and $\sigma_2 = (\varepsilon, \varepsilon, (2\varepsilon, 2\varepsilon), \ldots, (ke, ke), (T, ke))$, each with probability $1/2$, where $T = 4k\varepsilon$.

The expected cost of an optimal fair offline solution is at most $\varepsilon$. Any deterministic online algorithm has cost at least $k\varepsilon/2$. The Theorem follows by applying Yao’s Principle.

The fair adversary is still too powerful in the sense that it can move to points where it knows that a request will appear without revealing any information to the online server before reaching the point. A non-abusive adversary does not possess this power.

Definition 8.30 (Non-Abusive Adversary)
An adversary $\text{ADV}$ for the OLTSP on $\mathbb{R}$ is non-abusive, if the following holds: At any moment in time $t$, where the adversary moves its server from its current position $p_{\text{ADV}}(t)$ to the right (left), there is a request from $\sigma_{\leq t}$ to the right (left) of $p_{\text{ADV}}(t)$ which $\text{ADV}$ has not served yet.

The following result shows that finally, the non-abusive adversary allows a distinction between different algorithms. The proof of the theorem is beyond the scope of these lecture notes and can be found in the cited paper.

Theorem 8.31 (\cite{18}) There exists a deterministic algorithm for the $F_{\text{max}}$-OLTSP which is $8$-competitive against a non-abusive adversary.

8.5 Exercises

Exercise 8.1
Show that the »natural REPLAN-approach« is not competitive for the $\sum w_j C_j$-OLDARP (this approach always recomputes an optimal solution with respect to the objective function of minimize the sum of completion times, if a new request becomes known).

Exercise 8.2
Prove Lemma 8.15
Let $a_i, b_i \in \mathbb{R}_{\geq 0}$ for $i = 1, \ldots, p$, for which

(i) $\sum_{i=1}^{p} a_i = \sum_{i=1}^{p} b_i$;

(ii) $\sum_{i=1}^{p'} a_i \geq \sum_{i=1}^{p'} b_i$ for all $1 \leq p' \leq p$.

Then $\sum_{i=1}^{p} \tau_i a_i \leq \sum_{i=1}^{p} \tau_i b_i$ for any nondecreasing sequence $0 \leq \tau_1 \leq \cdots \leq \tau_p$.

Exercise 8.3
Prove that MRIN is $3/2$-competitive for the OLTSP in $\mathbb{R}_{\geq 0}$. 
Part III

Scheduling
9.1 A Hard Problem

Since students are usually very short on money you decided to work in Luigi’s ice cream shop. Since Example 5.2 on page 68 Luigi’s business has expanded a lot. His ice cream has become famous and he now sells ice cream to many different places all over the world. In particular, his ice cream cakes have become sought after a lot. Luigi owns 10 ice cream cake machines to produce the cakes. The machines are identical in the sense that each of them can produce all of Luigi’s recipes at the same cost. Although processing times are the same among the machines, the fancier cakes need more time in the machines than the simpler ones.

It is 08:00 and you arrive a little bit sleepy at Luigi’s place. Luigi tells you that there is a total of 280 cakes that need to be produced today. As a big fan of AS Roma Luigi wants to watch the soccer cup final AS Roma vs. Juve at 18:00 and needs to prepare mentally for that big game so that he will have to leave in a few minutes. Luigi is afraid that anyone might steal his famous recipes for the ice cakes (which possibly could be deduced from the running production process) so he wants you to watch the machines until all the ice cakes are finished. »You can assign the cakes to the machines as you want«, he shouts as he slips into his AS Roma soccer shirt and heads out of the shop.

You are left with 280 cake orders and 10 ice cream cake machines. Needless to say that you would also hate to miss the soccer game at 18:00 (although you are a Juve fan and not a supporter of AS Roma). You want to assign jobs to machines so that you can get home as early as possible. In scheduling terms that means you want to minimize the makespan on identical parallel machines ($P||C_{\text{max}}$). Since the machines are currently being serviced and won’t be available until 09:00 you still have some time to think about a schedule that will get you to the soccer game...

You first think about a fancy combinatorial algorithm to schedule the cakes on the machines. But after a while, you find out that your problem is NP-hard (see Theorem 9.6 below). Given this fact, it is not impossible that there exists an efficient algorithm but you can be sure that it is quite unlikely. So, finding one until 6 p.m. seems out of reach.

Now, one option would be to formulate the problem as an Integer Linear Program and hand it to the next best solver such as CPLEX. However, there is no guarantee that even CPLEX will find a solution until 18:00 which means that you might not have anything to start with before the kickoff of the soccer cup final.

So branch-and-bound via CPLEX seems to be out of the race for the moment. What about some rules of thumb?

A reasonable strategy might be to place a cake order on a machine as soon as the machine becomes available. This rule is known as list scheduling. A quick calculation shows you

---

1 This mostly holds true for university professors, too.
that the list schedule will not end before 21:00. That is not good enough for you. However, there is one positive thing about the list schedule. You can prove (see Theorem 9.4) that this algorithm will never give a makespan greater than twice the optimal makespan. Thus, the optimal makespan has length at least \((21 - 9)/2 = 6\) hours. Thus, no matter how you schedule the jobs, you won’t be able to leave before 15:00.

Are there any other algorithms that will allow you to reach the cup final?

### 9.2 Scheduling Problems

Scheduling theory is concerned with the optimal allocation of scarce resources to activities over time. Scheduling problems arise in a variety of settings. In this part of the lecture notes we will explore the complexity and solvability of different scheduling problems. For many problems we will show that they can be solved efficiently in polynomial time. Our means of designing algorithms will be all kinds of combinatorial tricks. For NP-hard problems, we will take the approach of approximation algorithms which are very closely related to competitive online algorithms.

A scheduling problem is defined by three separate elements: the **machine environment**, the **optimality criterion** (or **objective function**), and a set of **side constraints and characteristics**.

We use the now-standard notation of \([\alpha|\beta|\gamma]\) for scheduling problems. A problem is denoted by \(\alpha|\beta|\gamma\), where \(\alpha\) denotes the machine environment, \(\beta\) denotes various side constraints and characteristics, and \(\gamma\) denotes the objective function.

For a single machine, the environment is \(\alpha = 1\). Other more complex environments involve parallel machines and shops. In the case of parallel machines we are given \(m\) machines. A job \(j\) with processing requirement \(p_j\) can be processed on any of these machines, or, if preemption is allowed, started on one machine, and when preempted potentially resumed on another machine. A machine can process at most one job at a time and a job can be processed by at most one machine at a time.

\(\alpha = P\) This denotes the case of **identical parallel machines**. A job \(j\) requires \(p_j\) units of processing time when processed on any machine.

\(\alpha = Q\) In the **uniformly related machines** environment each machine \(i\) has a speed \(s_i > 0\). A job \(j\), if processed entirely on machine \(i\), would take a total of \(p_j/s_i\) time to process.

\(\alpha = R\) In the **unrelated parallel machines** the relative performances of the machines is unrelated. In other words, the speed of machine \(i\) on job \(j\), \(s_{ij}\) depends on both the machine and the job; job \(j\) requires \(p_j/s_{ij}\) time on machine \(i\). We define \(p_{ij} := p_j/s_{ij}\).

As mentioned above, the parameter \(\beta\) denotes various side constraints which include the following:

- The presence of release dates is indicated by \(r_j\). A job \(j\) with release date \(r_j\) may not be scheduled before time \(r_j\).
- If preemption is allowed the entry includes pmtn. In a preemptive schedule a job may be interrupted and resumed later possibly on another machine.
- Precedence constraints are indicated by prec.

We turn to objective functions. The most common objective functions are
9.3 Approximation Algorithms and Schemes

\( C_{\text{max}} \) the maximum completion time, also called the makespan;
\[
\sum w_j C_j \quad \text{the average (weighted) completion time;}
\]
\( F_{\text{max}} = \sum F_j \); the maximum and average flow time; here the flow time of a job \( j \) is defined to be the time difference \( C_j - r_j \) between its release time \( r_j \) and its completion time \( C_j \).
\( L_{\text{max}} \) the maximum lateness; for this objective to make sense, one is given deadlines \( d_j \) for the jobs and the lateness of \( j \) is defined to be \( C_j - d_j \).

For example, the problem of scheduling the ice cream cakes on Luigi’s machines is described as \( P_j | r_j, \text{prec} | \sum C_j \).

Example 9.1 (Luigi and the Ice Cream Cakes for the Mafia) Another day at Luigi’s you find yourself in a new situation. Again, there are a couple of cake orders that have to be processed, but this time the situation has become more complex: Firstly, Luigi is short of some ingredients, and will get deliveries only over the day. This means that for every cake order \( j \) he knows a time \( r_j \) before which production can not start. Secondly, even the \textit{cosa nostra} has started to order ice cream at Luigi’s. Since the \textit{cosa nostra} has higher and lower \textit{mafiosi}, this induces precedences between the cake orders: if the \textit{mafioso} who ordered cake \( j \) is ranked higher than the one who ordered \( k \), then production of cake \( k \) must not begin before cake \( j \) is completed (otherwise Luigi risks ending up in the Tiber with a block of concrete at his feet). Luigi wants to make the \textit{mafiosi} as happy as possible, thus he wants to minimize the average completion time of a cake. Unfortunately, all except for one of his machines are currently being repaired, hence, he needs to process all jobs on just a single ice cream cake machine. His problem is thus \( 1|r_j, \text{prec} | \sum C_j \).

9.3 Approximation Algorithms and Schemes

Definition 9.2 (Approximation Algorithm)
Let \( \Pi \) be a minimization problem. A polynomial-time algorithm, \( ALG \), is said to be a \( \rho \)-approximation algorithm for \( \Pi \), if for every problem instance \( I \) with optimal solution value \( \text{OPT}(I) \), the solution returned by the algorithm satisfies:
\[
ALG(I) \leq \rho \cdot \text{OPT}(I).
\]

Definition 9.3 (Polynomial Time Approximation Scheme)
A family \{\( ALG_\varepsilon \)\}_\varepsilon of approximation algorithms for a problem \( \Pi \), is called a polynomial approximation scheme or PTAS, if algorithm \( ALG_\varepsilon \) is a \((1 + \varepsilon)\)-approximation algorithm for \( \Pi \). The scheme is called a fully polynomial time approximation scheme or FPAS, if the running time of \( ALG_\varepsilon \) is polynomial in the size of the input and \( 1/\varepsilon \).

9.4 List-Scheduling

Consider the problem of scheduling a sequence \( J = (r_1, \ldots, r_n) \) of jobs on a collection of \( m \) identical machines. Each job has a nonnegative processing requirement \( p(r_j) \) and must be processed on a single machine without interruptions. Each machine can run at most one job at a time (see Figure 9.1).

The goal of the scheduling problem in this section is to minimize the maximum load of a machine, or equivalently, to minimize the length of the schedule. This objective is usually referred to as minimizing the makespan.
Let us consider the following list-scheduling algorithm that was proposed by Graham in 1964. This algorithm is also called Graham’s Algorithm:

**Algorithm LS** Consider the jobs in the order $r_1, \ldots, r_n$. When processing job $r_i$ assign job $r_i$ to a machine which currently has the least load.

Will this algorithm give an optimal solution? Apparently not, if we consider the case of two machines and three jobs with processing times $1, 1, 2$. Then, LS will assign the first two jobs to different machines and after the assignment of the third job end up with a makespan of 3. On the other hand, the optimal solution has a makespan of 2.

Although LS does not always yield an optimal solution we can still prove a bound on its performance:

**Theorem 9.4** Given any input instance $I$, the algorithm LS will produce a schedule of length at most $(2 - 1/m)$ times the optimal solution.

**Proof:** Set $I = (r_1, \ldots, r_n)$ be any sequence of jobs. Without loss of generality we assume that machine 1 is the machine on which the maximum load is achieved by LS. We consider the first time during the processing of $I$ when this load appears on machine 1, say, when job $r_i$ with processing time $w$ is assigned to 1. Let $l$ denote the load of machine 1 prior to the assignment. With these notations, we have $LS(I) = l + w$.

By construction of the algorithm, at the time when job $r_i$ is assigned, all machines different from 1 must have load at least $l$. Hence, the sum of all processing times is at least $ml + w$. This gives us the lower bound

$$OPT \geq \frac{ml + w}{m} = l + \frac{w}{m}.$$ 

Thus, using $OPT \geq w$ we get.

$$LS(I) = l + w \leq OPT + (1 - \frac{1}{m})w \leq OPT + (1 - \frac{1}{m})OPT = (2 - \frac{1}{m})OPT.$$ 

This proves the claim. 

Observe that LS is in fact an online algorithm for the following online scheduling problem: Jobs are given in a request sequence one by one and job $r_i$ must be scheduled before job $r_{i+1}$ becomes known. Theorem 9.4 results in the following competitiveness bound for this online-scheduling problem.

**Corollary 9.5** LS is also $(2 - 1/m)$-competitive for the online-problem of scheduling jobs on identical machines such as to minimize the makespan.
Let us return to the offline problem again. Is there a better algorithm than LS? The following construction shows that we can not hope to find a polynomial time algorithm which always finds an optimal solution, since this result would imply that \( P = NP \).

**Theorem 9.6** The scheduling problem of minimizing the makespan is \( NP \)-hard even for the case of \( m = 2 \) machines.

**Proof:** The claim is established by a simple reduction from PARTITION, a well known \( NP \)-complete problem, see e.g. [11]. An instance of PARTITION is given by integers \( a_1, \ldots, a_n \).

The question asked is whether there there exists a partition of \( \{1, \ldots, n\} \) into two parts \( X \cup Y = \{1, \ldots, n\} \), \( X \cap Y = \emptyset \) such that

\[
\sum_{i \in X} a_i = \sum_{i \in Y} a_i.
\]

Let \( B := \frac{1}{2} \sum_{i=1}^{n} a_i \).

Given an instance \( I \) of PARTITION we construct an instance \( I' \) of the scheduling problem on two uniform machines as follows. The jobs \( (r_1, \ldots, r_n) \) correspond to the numbers \( a_1, \ldots, a_n \). Job \( r_i \) has processing time \( a_i \). It is immediate to see that \( I' \) has an optimal solution of cost at most \( B \) if and only if the instance \( I \) of PARTITION has a solution. \( \square \)

### 9.5 Minimizing Project Duration

In the basic project scheduling problem we are given a project consisting of a set \( J \) of jobs with known durations \( p(j) \) \( (j \in J) \). Moreover, we are given precedence relations \( A \) between the jobs with the following meaning: if \( (i, j) \in A \), then job \( i \) must be completed before job \( j \) is started. The goal is to finish the whole project as early as possible.

A feasible project schedule consists of a starting time \( s(j) \) for each job \( j \in J \) such that:

\[
\begin{align*}
  s(j) &\geq s(i) + p(i) & \text{for all } (i, j) \in A \\
  s(j) &\geq 0 & \text{for all } j \in J
\end{align*}
\]

Constraint (9.1) enforces the precedence constraints and (9.2) makes the schedule start not earlier than the current time. The time when the schedule is completed is \( \max_{j \in J} (s(j) + p(j)) \).

We can model the project scheduling problem as a shortest path problem in a network as follows. We define a project network \( G \) whose nodes correspond to the jobs in \( J \). Whenever \( (i, j) \in A \), there is a directed arc \( (i, j) \) in the network of cost \( c(i, j) = p(i) \). We add two artificial nodes \( s \) and \( t \) corresponding to the start of the project. Node \( s \) is connected to every node \( j \) that has no incoming arc via an arc \( (s, j) \), and for every node \( i \) with no outgoing arc, we add the arc \( (i, t) \). All arcs incident to \( s \) have zero cost, while an arc \( (i, t) \) has cost \( p(i) \). The construction is illustrated in Figure 9.2.

We can assume that \( G \) as constructed above is acyclic, since a cycle would imply that the project can never be finished. Moreover, a cycle can be detected easily in advance, so that we do not have to worry about this case from now on.

Figure 9.2: Transformation of the project scheduling problem to a shortest path problem.

We claim that the project scheduling problem is equivalent to the problem of finding a longest path from \( s \) to \( t \) in \( G \). To prove this claim, we first formulate the project scheduling problem as a Linear Program. Let \( \pi(i) \) denote the starting time of job \( i \in J \cup \{s, t\} \) and
\( A_t \) denote the union of the precedence arcs \( A \) and the new arcs \((s, j)\) and \((i, t)\) with their respective costs. Note that we have added the artificial jobs \( s \) and \( t \) corresponding to the start and the end of the project. Then, we can write the problem as follows:

\[
\begin{align*}
\min & \quad \pi(t) - \pi(s) \\
\pi(j) - \pi(i) & \geq c(i, j) \quad \text{for all } (i, j) \in \bar{A} \\
\end{align*}
\]

The inequalities \(9.3b\) model the precedence constraints just as \(9.1\) did. The objective \( \pi(t) - \pi(s) \) measures the project duration.

Now consider the dual of \(9.3\) which is given by

\[
\begin{align*}
\max & \quad \sum_{i: (i, j) \in \bar{A}} c(i, j) f(i, j) \\
- \sum_{j: (j, i) \in \bar{A}} f(i, j) - \sum_{i: (i, j) \in \bar{A}} f(i, j) & = \begin{cases} 
-1 & \text{for } i = s \\
0 & \text{for } j \in J \\
1 & \text{for } i = t 
\end{cases} \\
f(i, j) & \geq 0 \quad \text{for all } (i, j) \in \bar{A}
\end{align*}
\]

The problem \(9.3\) is a minimum cost flow problem (see the LP formulation of the minimum cost flow problem in \(3.10\)) with costs given by \(-c(i, j)\). In fact, it is a transshipment problem (see \(3.19\)), since all capacities are infinite. It asks to send one unit of flow from \( s \) to \( t \) at minimum cost \(-\sum c(i, j) f(i, j)\). According to the result of Exercise \(3.1\) this particular task is equivalent to computing a shortest path from \( s \) to \( t \) in \( G \) where arc lengths are \(-c(i, j)\). This in turn means nothing else but computing a longest path in \( G \) from \( s \) to \( t \).

Since \( G \) is acyclic, a longest path can be computed in linear time, for instance using a topological sort of the graph \( G \) (see Exercises \(9.1, 9.2\)). This allows us to solve the project scheduling problem in linear time, too.

### 9.6 Just-in-Time Scheduling

The just-in-time scheduling problem is an extension of the project scheduling problem in Section \(9.5\). We are additionally given a subset \( B \subseteq A \) and numbers \( \beta(i, j) \) for all \((i, j) \in B\) such that, if \((i, j) \in B\) then job \( j \) must start no later than \( \beta(i, j) \) plus the starting time of \( i \). The »just-in-time« component here is that \( j \) is started »just early enough« in order not to delay the whole process.

We can formulate this problem again as a Linear Program:

\[
\begin{align*}
\min & \quad \pi(t) - \pi(s) \\
\pi(j) - \pi(i) & \geq c(i, j) \quad \text{for all } (i, j) \in \bar{A} \\
\pi(i) - \pi(j) & \geq -\beta(i, j) \quad \text{for all } (i, j) \in B
\end{align*}
\]

The problem \(9.5\) can be solved just as the project scheduling problem in the previous section by a shortest path computation. The arcs in the network \( G' \) (besides the arcs incident to the start- and endnode) are

- for each \((i, j) \in A\) an arc \((i, j)\) of cost \(-c(i, j)\), and
- for each \((i, j) \in B\) an arc \((j, i)\) of cost \(\beta(i, j)\).
Notice that it might be the case that the network $G'$ contains cycles! However, it can be shown (see Exercise 9.3) that a negative length cycle is an indicator of infeasibility of the problem. We can use the Bellman-Ford-Algorithm to determine shortest path distances in $G'$ and detect a negative length cycle. Hence, once more a shortest path computation helps in solving the scheduling problem.

9.7 Exercises

Exercise 9.1

A topological sort of a directed graph $G = (V, A)$ is a bijective mapping $f: A \rightarrow \{1, \ldots, n\}$, where $n := |V|$ with the following property:

$$f(i) < f(j) \quad \text{for all } (i, j) \in A.$$

(i) Prove that $G$ has a topological sort if and only if $G$ is acyclic.

(ii) Show how to construct a topological sort in $O(|V| + |A|)$ time (if one exists, otherwise the algorithm should correctly inform that there is no topological sort).

Hint: You might have shown in (i) that $G$ must contain a source node, that is, a node where no arc ends. This node can be given number 0.

Exercise 9.2

Let $G = (V, A)$ be an acyclic graph and $c: A \rightarrow \mathbb{R}$ an arbitrary arc weight function (which might be negative). Let $s, t \in V$ be two nodes. Show how to compute a shortest $(s, t)$-path in $G$ in linear time.

Hint: Use a topological sort in conjunction with a dynamic programming approach.

Exercise 9.3

Show that if the network $G'$ constructed in Section 9.6 contains a negative length cycle, then the just-in-time scheduling problem has no solution.
The Single Machine

The single machine is the most simple scheduling environment and thus provides a good starting point. In this chapter we are concerned with the solvability and complexity of several scheduling problems on a single machine.

In all of our scheduling problems we begin with a set $J$ of $n$ jobs, numbered $1, \ldots, n$. Each job $j$ has a processing requirement $p_j$, i.e., the job requires a total of $p_j$ units of time on the machine. If each job must be processes in an uninterrupted fashion, we speak of a *nonpreemptive scheduling environment*. On the other hand, if a job may be processed for some time, then interrupted and resumed later (maybe being interrupted again), we have a *preemptive environment*.

A schedule $S$ for the set $J$ specifies for each job $j$ which $p_j$ units of time the machine uses to process $j$. Given a schedule $S$, we denote by $C_j^S$ the *completion time* of job $j$ in $S$, that is, the moment in time when job $j$ has been completely processed in the schedule.

The overall goal of scheduling is to provide »good« schedules, where the exact objective function depends on the particular application.

### 10.1 Scheduling Rules

In this section we review simple *scheduling rules* (so as to speak: »rules of thumb«) which in some cases can be used to solve scheduling problems.

Perhaps the simplest scheduling problem on a single machine consists of minimizing the sum of completion times $\sum C_j$. Intuitively, it makes sense to schedule the largest job at the end of the schedule to ensure that it does not contribute to the delay of any other job. This is formalized in the following simple algorithm:

**Shortest processing time (SPT)** Order the jobs by nondecreasing processing time (breaking ties arbitrarily) and schedule in that order.

**Theorem 10.1** SPT determines an optimal solution for $1||\sum C_j$.

**Proof:** We use an interchange argument to prove the theorem. Consider an arbitrary optimal schedule and suppose that the jobs in this schedule are not scheduled in non-decreasing order of processing time. Then, there is a pair of jobs $j$ and $k$ such that $j$ is scheduled immediately before $k$ but $p_j > p_k$.

Consider the effect of interchanging $j$ and $k$ (see Figure 10.1). All other jobs keep their starting and completion time. All that changes, it the completion times of $j$ and $k$. Let $\ell$ be the time where $j$ started in the initial schedule. Then, the objective function changes by

$$-(t + p_j) - (t + p_j + p_k) + (t + p_k) + (t + p_k + p_j) = p_k - p_j < 0.$$


Hence, the new schedule is better than the initial one which was assumed to be optimal. This is a contradiction.

The algorithm given above and its analysis generalize to the optimization of the average weighted completion time \( \sum w_j C_j \). Again, intuitively we would like to schedule jobs in order of nonincreasing value of \( w_j/p_j \). This is usually called Smith’s rule:

**Smith’s rule (SMITH)** Order the jobs by nonincreasing ratio \( w_j/p_j \) (breaking ties arbitrarily) and schedule in that order.

**Theorem 10.2** SMITH determines an optimal solution for \( 1|| \sum w_j C_j \).

**Proof:** See Exercise 10.1.

**Shortest Remaining Processing Time (SRPT)** At each point in time, schedule the job with shortest remaining processing time, preempting when jobs of shorter processing time are released.

**Theorem 10.3** SRPT is an exact algorithm for \( 1|\pmtn| \sum C_j \).

**Proof:** Our proof uses again an exchange argument. Consider a schedule in which an available job \( j \) with the shortest remaining processing time is not being processed at time \( t \), and instead an available job \( k \) with strictly larger remaining processing time is run on the machine. We denote by \( p'_j \) and \( p'_k \) the remaining processing times of \( j \) and \( k \) at time \( k \). With this notation \( p'_j < p'_k \).

After time \( t \), a total of \( p'_j + p'_k \) units of time is spent on the processing of \( j \) and \( k \). We replace the first \( p'_j \) units of time that were allocated to either of jobs \( j \) and \( k \) after time \( km \) and use them to process \( j \) until completion. Then, take the remaining \( p'_k \) units of time that were spent processing jobs \( j \) and \( k \), and use them to run \( k \). This exchange is possible, since both jobs were available at time \( t \).

In the new schedule, all jobs different from \( j \) and \( k \) complete at the same time. Job \( k \) finishes when job \( j \) completed in the original schedule. But job \( j \), which needed \( p'_j < p'_k \) additional time, finishes before job \( k \) originally completed. Thus, the objective function strictly decreases.

The weighted version of the previous problem \( 1|\pmtn| \sum w_j C_j \) turns out to be NP-hard. We have seen that the version without release dates \( 1|\pmtn| \sum w_j C_j \) can be solved optimally (even online) by Smith’s rule (see Theorem 10.2). Moreover, the unweighted problem \( 1|\pmtn| \sum C_j \) can also be solved optimally be the shortest remaining processing time algorithm SRPT (see Theorem 10.3).

**Algorithm WSPT** At any time \( t \) schedule the job with highest ratio \( w_j/p_j \) among the available not yet completed jobs. Interrupt the processing of a currently active job, if necessary.
10.2 Using a Preemptive Schedule

**Theorem 10.4** WSPT is 2-competitive for the online problem \(1|r_j, pmtn| \sum w_j C_j\).

**Proof:** Consider a job \(j\). Then, the completion time \(C_j\) of \(j\) in the schedule produced by WSPT is at most \(r_j\) plus the sum of the processing times of all jobs \(k\) with higher ratio \(w_k/p_k\) plus the processing time of \(j\):

\[
C_j \leq r_j + \sum_{k: \frac{w_k}{p_k} > \frac{w_j}{p_j}} p_k + p_j. \quad (10.1)
\]

Summing over all jobs \(j = 1, \ldots, n\) we get from (10.1):

\[
\sum_{j=1}^{n} w_j C_j \leq \sum_{j=1}^{n} w_j (r_j + p_j) + \sum_{j=1}^{n} w_j \sum_{k: \frac{w_k}{p_k} > \frac{w_j}{p_j}} p_k \leq \text{OPT}(\sigma) \leq 2 \text{OPT}(\sigma).
\]

We now consider the problem of minimizing the maximum lateness \(L_{\text{max}}\). A simple greedy algorithm attempts to schedule the job that is closest to being late.

**Earliest due date (EDD)** Order the jobs by nondecreasing deadlines (due dates) and schedule in that order.

**Theorem 10.5** EDD is an optimal algorithm for \(1||L_{\text{max}}\).

**Proof:** Assume without loss of generality that \(d_1 \leq d_2 \leq \cdots \leq d_n\). For a schedule define the number of inversions to be the number pairs \(j, k\) such that \(j < k\) but \(j\) is scheduled immediately after \(k\). Let \(S\) be an optimal schedule with the fewest number of inversions. If there are no inversions in \(S\), then this is the EDD-schedule. Assume for the sake of a contradiction that \(S\) has an inversion \(j, k\).

Consider the effect of exchanging \(j\) and \(k\) and call the resulting schedule \(S'\). The completion times of all other jobs remain unchanged. Moreover, the exchange operation strictly decreases the number of inversions. We have \(L^S_j = C^S_j - d_j \leq C^S_k - d_k \leq C^S_j - d_j\). Consequently, \(\max\{L^S_j, L^S_k\} = C^S_j - d_j\). On the other hand, \(L^S_j \leq L^S_j\), since the completion time of \(j\) decreases. Moreover, \(L^S_k = C^S_j - d_k = C^S_j - d_k < C^S_j - d_j\). Hence, the schedule \(S'\) has maximum lateness no larger than \(S\). This contradicts the assumption that \(S\) was optimal with the fewest number of inversions. \(\square\)

### 10.2 Using a Preemptive Schedule

In this section we discuss techniques for inferring an ordering of jobs from a relaxation of the problem. Our example with be the problem \(1|r_j| \sum C_j\) which is known to be NP-hard to solve. However, we can solve the relaxation \(1|r_j, pmtn| \sum C_j\) in polynomial time by the SRPT-rule (see Theorem 10.3).

Our algorithm CONV\-PREEMPTIVE works as follows. Given an instance of \(1|r_j| \sum C_j\) it runs SRPT to get an optimal solution for the relaxation \(1|r_j, pmtn| \sum C_j\). Then, it converts this schedule back to a non-preemptive schedule as follows. Suppose that the completion times of the optimal preemptive schedule \(P\) satisfy \(C^P_1 \leq C^P_2 \leq \cdots \leq C^P_n\). CONV\-PREEMPTIVE schedules the jobs nonpreemptively in this order. If at some point in time the next job in the order has not been released, we wait idly until its release date and then schedule it.
Theorem 10.6 \textsc{convert-preemptive} is a 2-approximation algorithm for $1|r_j|\sum C_j$.

\textbf{Proof:} Let $S$ be the non-preemptive schedule produced by \textsc{convert-preemptive}. We can view $S$ as follows. Consider the last scheduled piece of a job $j$. Suppose that this piece has length $k_j$ and is scheduled from time $C^P_j - k_j$ to $C^P_j$ in the preemptive schedule. Insert $p_j - k_j$ extra units of time in the schedule at time $C^P_j$ (delaying by an additional amount of $p_j - k_j$ time the part of the schedule after $C^P_j$), and schedule $j$ nonpreemptively in the resulting available block of length $p_j$. We then remove all pieces of $j$ that were processed before $C^P_j$ and push all jobs forward in time as much as possible without changing the scheduled order or violating the release date constraint (see Figure 10.2 for an illustration).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure10_2.png}
\caption{Illustration of algorithm \textsc{convert-preemptive}}
\end{figure}

In the conversion process, job $j$ can only be moved back by processing times associate with jobs that finish earlier in $P$. Thus,

$$C^S_j \leq C^P_j + \sum_{k : C^P_k \leq C^P_j} p_k.$$  \hfill (10.2)

Since all jobs from the set \{ $k : C^P_k \leq C^P_j$ \} completed before $C^P_j$, the sum of their processing times can not be larger than $C^P_j$. Hence, from (10.2), we obtain $C^S_j \leq 2C^P_j$. Since the objective function value of the optimal preemptive schedule is a lower bound for the optimal non-preemptive schedule, the claim follows. \hfill \Box

\section{10.3 Linear Programming}

Linear Programming is an important tool for obtaining exact an approximate algorithms for scheduling problems. As an example, we consider the problem $1|r_j, prec| \sum C_j$.

In a first step, we describe a linear programming relaxation of the problem. This relaxation is not just the continuous relaxation of an exact integer programming formulation. Instead, we add a class of inequalities that would be satisfied by any feasible solution to $1|r_j, prec| \sum C_j$. The variables in our formulation are the $C_j$:

$$\min \sum_{j=1}^{n} w_j C_j$$ \hfill (10.3a)

$$C_j \geq r_j + p_j \quad \text{for} \quad j = 1, \ldots, n$$ \hfill (10.3b)

$$C_k \geq C_j + p_k \quad \text{for all} \quad j, k \text{ such that } j < k$$ \hfill (10.3c)

$$C_k \geq C_j + p_k \quad \text{or} \quad C_j \geq C_k + p_j \quad \text{for all} \quad j, k$$ \hfill (10.3d)

Unfortunately (10.3) is not a Linear Programming formulation yet, since the constraint (10.3d) is not linear. Hence, we will replace (10.3d) by a relaxation. To this end,
define for a subset $S \subseteq \{1, \ldots, n\} = J$ of the jobs

$$p(S) := \sum_{j \in S} p_j$$

$$p^2(S) := \sum_{j \in S} p_j^2.$$

**Lemma 10.7** Let $C_1, \ldots, C_n$ be the completion times of jobs in any feasible schedule for a single machine. Then, the $C_j$ satisfy the inequalities:

$$\sum_{j \in S} p_j C_j \geq \frac{1}{2} (p(S)^2 + p^2(S)) \quad \text{for all } S \subseteq J \quad (10.4)$$

**Proof:** Without loss of generality assume that $C_1 \leq C_2 \leq \cdots \leq C_n$. We first consider the case $S = J$. Then, $C_j \geq \sum_{k=1}^n p_k$. Multiplying by $p_j$ and summing over $j$ gives

$$\sum_{j=1}^n p_j C_j \geq \sum_{j=1}^n p_j \sum_{k=1}^j p_k = \frac{1}{2} (p^2(S) + p(S)^2).$$

Hence, the claim is shown for $S = J$. The case $S \neq J$ can be reduced to this case by the following argument: The schedule for $J$ yields a feasible schedule for any subset $S$ (just ignore the jobs from $J \setminus S$). Hence, we may view $S$ as the entire set of jobs and get back to the case handled above. \hfill \Box

We now replace (10.3b) by (10.2). Although the new set of constraints form an exponentially large set, the resulting Linear Program can still be solved in polynomial time, since the constraints can be separated in polynomial time [13].

Assume now that we have a solution $\tilde{C}_1 \leq \tilde{C}_2 \leq \cdots \leq \tilde{C}_n$ for the Linear Program constructed above. We show that we can use the information from the LP to get a provably good schedule. For simplicity, we restrict ourselves to the case of $1|\text{prec}| \sum C_j$. The case of $1|r_j, \text{prec}| \sum C_j$ is along the same lines but a bit more technical.

The algorithm SCHEDULE-BY-$\tilde{C}_j$ simply schedules the jobs in the order $1, \ldots, n$ according to the values $\tilde{C}_1 \leq \tilde{C}_2 \leq \cdots \tilde{C}_n$ obtained from the LP. Observe that this ordering respects the precedence constraints since $\tilde{C}_k \geq \tilde{C}_j + p_j$ if $j < k$.

**Theorem 10.8** Algorithm SCHEDULE-BY-$\tilde{C}_j$ is a 2-approximation for $1|\text{prec}| \sum C_j$.

**Proof:** Let $C_1 \leq \cdots \leq C_n$ denote the completion times obtained by SCHEDULE-BY-$\tilde{C}_j$. Note that $C_j = \sum_{k=1}^j p_j = p(S)$ for $S = \{1, \ldots, j\}$. We use inequality (10.4) for $S = \{1, \ldots, j\}$. This results in

$$\sum_{k=1}^j p_k \tilde{C}_k \geq \frac{1}{2} (p(S)^2 + p^2(S)) \geq \frac{1}{2} p(S)^2 = \frac{1}{2} p(S) C_j. \quad (10.5)$$

On the other hand we have $\tilde{C}_k \leq \tilde{C}_j$ for $k \leq j$. Thus,

$$\tilde{C}_j p(S) = C_j \sum_{k=1}^j p_k \geq \sum_{k=1}^j \tilde{C}_k p_k \geq \frac{1}{2} p(S) C_j. \quad (10.6)$$

Dividing both sides of (10.6) by $p(S)$ gives $C_j \leq 2 \tilde{C}_j$ and thus $\sum_{j=1}^n w_j C_j \leq 2 \sum_{j=1}^n w_j \tilde{C}_j \leq 2 \text{OPT}$. \hfill \Box
10.4 \( \alpha \)-Point Scheduling

In Sections 10.2 and 10.3 we have learned two techniques for constructing schedules with provably good approximation ratios: the conversion of preemptive schedules and Linear Programming. In this section we will introduce another powerful concept, namely, the notion of \( \alpha \)-points.

Recall again our 2-approximation algorithm for \( 1|r_j| \sum C_j \): We first computed an optimal schedule \( P \) for the preemptive problem \( 1|r_j, pmtn| \sum C_j \) via the SRPT-rule and then converted the preemptive schedule back to a non-preemptive one by scheduling jobs in the order of their completion times in the preemptive schedule. We were able to show that in the final schedule \( S \) each job completed no later than twice its completion time in \( P \), that is, \( C_S^j \leq 2C_P^j \) for each job \( j \).

The notion of an \( \alpha \)-schedule generalizes the above idea. Given a preemptive schedule \( P \) and \( 0 < \alpha \leq 1 \), we define the \( \alpha \)-point \( C_P^j(\alpha) \) of a job \( j \) as the first point in time at which an \( \alpha \)-fraction of job \( j \) has been completed in \( P \). In other words, at time \( C_P^j(\alpha) \) the job \( j \) has been processed on the machine(s) for \( \alpha p_j \) time units. In particular we have, \( C_P^j(1) = C_P^j \).

We also define \( C_P^j(\alpha) \) as the starting time of job \( j \) in \( P \).

An \( \alpha \)-schedule is a non-preemptive schedule obtained by list scheduling jobs in order of increasing \( C_P^j(\alpha) \). Clearly, an \( \alpha \)-scheduler is an online algorithm; moreover for \( \alpha = 1 \), we obtain or algorithm CONVERT-PREEMPTIVE which we showed to be a 2-approximation.

**Lemma 10.9** Given an instance of one-machine scheduling with release dates, for any \( 0 < \alpha \leq 1 \), an \( \alpha \)-schedule satisfies \( C_S^j \leq (1 + 1/\alpha)C_P^j \).

**Proof:** The proof is similar to Theorem 10.6. Assume without loss of generality that \( C_P^1(\alpha) \leq C_P^2(\alpha) \leq \cdots \leq C_P^n(\alpha) \). Let \( r_{j_{\max}}^{\max} = \max_{1 \leq k \leq j} r_k \) be the latest release time among jobs with \( \alpha \)-points no greater than that of \( j \). By definition at time \( r_{j_{\max}}^{\max} \) all jobs \( 1, \ldots, j \) have been released and thus

\[
C_S^j \leq r_{j_{\max}}^{\max} + \sum_{k=1}^j p_k. \tag{10.7}
\]

On the other hand, \( C_P^j \geq r_{j_{\max}}^{\max} \), since only an \( \alpha \)-fraction of \( j \) has finished by \( r_{j_{\max}}^{\max} \). Since the \( \alpha \)-fractions of jobs \( 1, \ldots, j \) all have been processed by time \( C_P^j \), we also get \( C_P^j \geq \alpha \sum_{k=1}^j p_k \). Using these two inequalities in (10.7) gives \( C_S^j \leq (1 + 1/\alpha)C_P^j \).

Although Lemma 10.9 is interesting, it may appear useless for obtaining approximation ratios less than 2: the term \( 1 + 1/\alpha \) is at least 2 for all \( 0 < \alpha \leq 1 \). However, there is one crucial observation: A worst-case instance that induces a performance ratio \( 1 + 1/\alpha \) for one (fixed) value of \( \alpha \) is possibly not a worst-case instance for some other value of \( \alpha \). We could pick \( \alpha \) randomly to »smooth« over the worst-case instances. In the offline setting we can even try different values of \( \alpha \) deterministically, finally outputting the best solution found. In the sequel we will investigate this intuition. To this end, we first study the structure of preemptive schedules and generalize the list scheduling algorithm to a preemptive environment.

### 10.4.1 Preemptive List Scheduling

One way to construct a feasible preemptive schedule from a given list of jobs representing some order is preemptive list scheduling:
Preemptive list scheduling (PLS) Schedule at any point in time the first available job in the given list. Here, a job is available if it release time has elapsed.

**Lemma 10.10** Let \( P \) be a feasible preemptive schedule and \( P' \) the schedule resulting from applying PLS to the list of jobs sorted in nondecreasing order of completion times in \( P \). Then, \( C_j^{P'} \leq C_j^P \) for all jobs \( j \).

**Proof:** Let \( j \) be any job and define \( t \geq 0 \) be the earliest point in time such that there is no idle time in \( P' \) during \( (t, C_j^{P'}) \) and only jobs \( k \) with \( C_k^P \leq C_j^P \) are processed. Call the set of these jobs \( K \). Then

\[
C_j^{P'} = t + \sum_{k \in K} p_k. \tag{10.8}
\]

All jobs in \( K \) must have been released by time \( t \), whence \( r_k \geq t \) for all \( k \in K \). Thus,

\[
C_j^P \geq t + \sum_{k \in K} p_k. \tag{10.9}
\]

Comparing (10.9) and (10.8) yields the claim. \( \square \)

As a first consequence of Lemma 10.10 we consider the problem \( 1|\text{pmtn}| \sum w_j C_j \). Recall that the non-preemptive problem \( 1|| \sum w_j C_j \) could be solved optimally by Smith’s ratio rule (Theorem 10.2 on page 132). Take an optimal preemptive schedule \( P \) for \( 1|\text{pmtn}| \sum w_j C_j \) and apply preemptive list scheduling according to non-increasing completion times as in Lemma 10.10. The new schedule \( S \) is not worse than \( P \) according to the lemma. The crucial observation now is that \( S \) does not contain any preemptions! A job would only be preempted if a new job with smaller completion time in \( P \) became available. But since there are no release dates, this is not the case. Thus, there is always an optimal schedule for \( 1|\text{pmtn}| \sum w_j C_j \) which is not-preemptive. That is, preemption is not necessary in \( 1|\text{pmtn}| \sum w_j C_j \). As a corollary, Smith’s rule also solves \( 1|\text{pmtn}| \sum w_j C_j \).

**Corollary 10.11** Smith determines an optimal solution for \( 1|\text{pmtn}| \sum w_j C_j \).

A natural generalization of Smith’s rule to \( 1|r_j, \text{pmtn}| \sum w_j C_j \) (which is \( \text{NP-hard} \)) is preemptive list scheduling in order of non-increasing ratios \( w_j/p_j \). Observe that this algorithm also works online since at any point in time only knowledge about the jobs released up to this point is required!

**Theorem 10.12** PLS in order of non-increasing ratios \( w_j/p_j \) is a 2-approximation algorithm for \( 1|r_j, \text{pmtn}| \sum C_j \).

**Proof:** Let \( C_j \) be the completion time of job \( j \) in the schedule constructed by PLS and denote by \( C_j^\ast \) the completion time in the optimal schedule. Without loss of generality assume that \( w_1/p_1 \geq \cdots \geq w_n/p_n \). Clearly,

\[
\sum_{j=1}^n w_j C_j^\ast \geq \sum_{i=1}^n w_j r_j, \tag{10.10}
\]

since not even the optimal solution can run jobs before their release times. Another lower bound on the optimal value is obtained by making all release times zero. Hence, we relax \( 1|r_j, \text{pmtn}| \sum C_j \) to \( 1|\text{pmtn}| \sum C_j \), which according to Corollary 10.11 can be solved by Smith’s rule. Hence, the completion time of job \( j \) in the relaxation is \( \sum_{k \leq j} p_k \). This gives:

\[
\sum_{j=1}^n w_j C_j^\ast \geq \sum_{j=1}^n w_j \sum_{k \leq j} p_k. \tag{10.11}
\]
Note that $C_j$ is bounded from above by $r_j + \sum_{k \leq j} P_k$ by construction of the preemptive list schedule in order of non-increasing ratios $w_j/p_j$. Hence $\sum w_j C_j$ is bounded from above by the sum of the terms on the right hand sides in (10.10) and (10.11).

10.4.2 A Randomized $\alpha$-Scheduler

Theorem [10.12] showed us that preemptive list scheduling in order of non-increasing ratios $p_j/w_j$ yields a 2-approximation algorithm. We will now improve this result by preemptive list scheduling in order of non-decreasing $\alpha$-points.

Algorithm 10.1 Randomized algorithm for $1|r_j, pmtn| \sum w_j C_j$

RAND$\alpha$

1. Draw $\alpha$ randomly from $[0, 1]$ according to the density function $f$.
2. Construct the preemptive list schedule $P$ in order of non-increasing ratios $w_j/p_j$.
3. Apply preemptive list scheduling in order of non-decreasing $\alpha$-points $C_j^P(\alpha)$.

In the sequel we are going to show that the randomized $\alpha$-scheduler RAND$\alpha$ (Algorithm 10.1) obtains an approximation ratio $4/3$. Observe that this algorithm is still online, so it also achieves the same competitive ratio.

The analysis of the algorithm is divided into three parts:

1. We construct an Integer Linear Programming Relaxation of $1|r_j, pmtn| \sum w_j C_j$ to get a lower bound for the optimal solution.
2. We bound the completion time of each job in the schedule of RAND$\alpha$ by a value which depends on $\alpha$.
3. We compare the expected completion times to the lower bound computed in the first step.

\begin{align}
\min \quad & \sum_{j=1}^{n} w_j C_j^{ILP} \\
\text{s.t.} \quad & \sum_{t=r_j}^{T} y_{jt} = p_t \quad \text{for } j = 1, \ldots, n \quad (10.12b) \\
& \sum_{j=1}^{n} y_{jt} \leq 1 \quad \text{for } t = 0, \ldots, T \quad (10.12c) \\
& C_j^{ILP} = \frac{p_j}{2} + \frac{1}{p_j} \sum_{t=0}^{T} y_{jt}(t + \frac{1}{2}) \quad \text{for } j = 1, \ldots, n \quad (10.12d) \\
& y_{jt} = 0 \quad \text{for all } j \text{ and } t = 0, \ldots, r_j - 1 \quad (10.12e) \\
& y_{jt} \in \{0, 1\} \quad \text{for all } j \text{ and } t \quad (10.12f)
\end{align}

Lemma 10.13 Consider an arbitrary preemptive schedule $P$ that is finished before time $T+1$, and assign to the ILP $\{10.12\}$ variables $y_{jt}$ by $y_{jt} = 1$ if job $j$ is being processed in the interval $(t, t+1]$, and $y_{jt} = 0$ otherwise. Then,

\[ \int_{0}^{1} C_j(\alpha) d\alpha = \frac{1}{p_j} \sum_{t=0}^{T} y_{jt}(t + \frac{1}{2}) \leq C_j^P - \frac{p_j}{2} \quad (10.13) \]

for each job $j$ with equality if and only if $j$ is never preempted from the machine.
Corollary 10.14 The optimal value of the ILP (10.12) is a lower bound on the value of an optimal preemptive schedule.

Lemma 10.15 The ILP (10.12) can be solved in $O(n \log n)$ time.

For a job $j$ we denote by $J'$ the set of jobs that start before $j$ in the preemptive list schedule $P$. Observe that no job $k \in J'$ is processed between the start and the completion time of $j$. We denote the fraction of $k \in J'$ completed by time $C_j^P(0)$ by $\eta_k$. The start time $C_j^P(0)$ of job $j$ equals the amount of idle time before $j$ is started plus the sum of fractional processing times of jobs in $J'$, thus

$$C_j^P(0) \geq \sum_{k \in J'} \eta_k p_k.$$  \hfill (10.14)

Lemma 10.16 Let $\alpha \in [0, 1]$ be fixed and let $C_j$ be the completion time of $j$ in the schedule computed by (the deterministic version of) Algorithm RAND$\alpha$. Then,

$$C_j \leq C_j^P(\alpha) + (1 - \alpha)p_j + \sum_{k \in J': \eta_k \geq \alpha} (1 - \eta_k)p_k.$$  \hfill (10.15)

Lemma 10.17 Let $f$ be a density function on $[0, 1]$ and $\alpha$ drawn randomly according to $\alpha$ so that $\mathbb{E}_f[\alpha] = \int_0^1 \alpha f(\alpha) \, d\alpha$. Assume that $\gamma > 0$ and

(i) $\max_{\alpha \in [0,1]} f(\alpha) \leq 1 + \gamma$,

(ii) $1 - \mathbb{E}_f[\alpha] \leq \frac{1 + \gamma}{2}$,

(iii) $(1 - \eta) \int_0^\eta f(\alpha) \, d\alpha \leq \gamma \eta$ for all $\eta \in [0, 1]$.

Moreover, the expected completion time of every job $j$ in the schedule constructed by algorithm RAND$\alpha$ is at most $(1 + \gamma)C_j^{ILP}$, where $C_j^{ILP}$ denotes the completion time in the Integer Linear Program (10.12).

Corollary 10.18 If one chooses the density function $f$, defined by

$$f(\alpha) = \begin{cases} \frac{1}{\gamma}(1 - \alpha)^{-2} & \text{if } \alpha \in [0, \frac{1}{2}] \\ \frac{1}{\gamma} & \text{otherwise}, \end{cases}$$

then algorithm RAND$\alpha$ achieves a competitive ratio of $4/3$.

10.5 Exercises

Exercise 10.1
Show that Smith’s rule yields an optimal algorithm for $1|| \sum w_j C_j$.

Exercise 10.2
Prove that, given a preemptive schedule $P$ for $1|\text{pmtn}| \sum w_j C_j$, there is a non-preemptive schedule $S$ such that $\sum w_j C_j^S \leq \sum w_j C_j^P$. 
11.1 Scheduling Rules

In Section 9.4 we studied the problem \( P || C_{\text{max}} \) and showed a performance result for the simple list scheduling algorithm of Graham which we recall here for completeness:

**Theorem 11.1** \( \text{LS} \) is a 2-approximation algorithm for \( P || C_{\text{max}} \).

**Proof:** See Theorem 9.4 on page 126 in Section 9.4. \( \square \)

We also showed in Theorem 9.6 that \( P || C_{\text{max}} \) is NP-hard to solve. Hence, we can not expect to obtain an exact algorithm that always runs in polynomial time. We will explore how the result of Theorem 11.1 can be improved. To this end, we first investigate the preemptive version of \( P || C_{\text{max}} \).

Let \( D := \max \{ \sum_j p_j / m, \max_j p_j \} \). Clearly, the optimal schedule can not be shorter than \( D \), not even if it uses preemption. McNaughton’s wrap-around rule produces a schedule of length \( D \), which must be optimal by this observation.

**McNaughton’s wrap-around rule (MCNWRAP)** Order the jobs arbitrarily. Then begin placing jobs on the machines, in order, filling machine \( i \) up until time \( D \) before starting machine \( i + 1 \).

Observe that MCNWRAP uses at most \( m - 1 \) preemptions. A job \( j \) might be split, assigned to the last \( t \) units of time of machine \( i \) and the first \( p_j - t \) units of time on machine \( i + 1 \). MCNWRAP is in fact able to schedule each job, since there are no more than \( mD \) units of processing time and \( D - t \geq p_j - t \) for any \( t \).

**Theorem 11.2** MCNWRAP is an exact algorithm for \( P | \text{pmtn} | C_{\text{max}} \). \( \square \)

Let us recall the analysis of LS. We denoted the maximum load by \( l + w \), where \( w \) was the processing time of some job. Since \( ml + w \leq \sum_j p_j \) we can think of the term \( l + w \) as follows. Each job starts begin processed before time \( \frac{l}{m} \sum_j p_j \), and the total schedule is no longer than \( \frac{l}{m} \sum_j p_j \) plus the length of the longest job that is running at time \( \frac{l}{m} \sum_j p_j \).

If we want to get a good schedule (at least a better one than for LS), it makes sense to run the longer jobs earlier and the shorter jobs later. This is formalized in the following algorithm:

**Longest processing time (LPT)** Sort the jobs according to nonincreasing processing time and then use LS to schedule the jobs.
Observe that in the transition of LS to LPT we use one property of the algorithm: LS could be applied to the online version of $P||C_{max}$, but LS can not, since it needs to know all jobs in advance.

**Theorem 11.3** LPT is a $4/3$-approximation algorithm for $P||C_{max}$.

**Proof:** We first show that we can assume without loss of generality that the job that completes last in the schedule of LPT (the job that determines the makespan) also is started last.

![Figure 11.1: Removal of the jobs that start after $j$ in the schedule produced by LPT does not decrease the objective function value.](image)

Suppose that job $j$ completes last in the schedule of LPT but is not the last job to start. Then, we can remove all jobs that start after the start time $s_j$ of $j$ without decreasing the makespan for LPT (see Figure 11.1). LPT would produce the same schedule if the removed jobs had not been present in the instance. Moreover, removing jobs can only decrease the length of the optimal schedule.

Hence, assume that the last job to complete is also the last job to start. The analysis of LS tells us that the length of the LPT-schedule is no larger than $OPT + \min_j p_j$.

If $\min_j p_j \leq OPT/3$, then the total schedule length is no larger than $4/3 \cdot OPT$ as claimed.

If, on the other hand, $\min_j p_j > OPT/3$, then all jobs have processing times strictly larger than $OPT/3$ and in the optimal schedule there can be at most two jobs per machine. Hence, it follows that $n \leq 2m$.

If $n \leq m$, then the LPT-schedule distributes the jobs to the machines so that each machine contains at most one job. Clearly, this is optimal. Hence, assume for the remainder of the proof that $m < n \leq 2m$.

Assume that $p_1 \geq p_2 \geq \cdots \geq p_n$. It is easy to see that an optimal schedule pairs job $j$ with job $2m + 1 - j$ if $2m + 1 - j \leq n$ and places $j$ by itself otherwise. However, this is exactly the schedule constructed by LPT. 

### 11.2 Application of Matching Techniques

In this section we consider the objective function $\sum C_j$. We start by investigating the problems $R||\sum C_j$ and $P||\sum C_j$ and show that both problems can be solved in polynomial time. While the algorithm for $P||\sum C_j$ is much simpler, it is advantageous to consider the more general problem $R||\sum C_j$ first, since the solution techniques will yield the simple algorithm for the special case in a nice way.

We consider the problem $R||\sum C_j$. For a schedule $S$ let $z_{ik}^S$ be the $k$th to last job that runs on machine $i$, and let $t_{ik}^S$ be the number of jobs that run on machine $i$. Suppose $j = z_{ik}^S$. 

Then, the completion time of \( j \) satisfies:

\[
C_j^B = C_{z_{ik}}^B = \sum_{x=k}^{\ell_i} p_{i,x} z_{ik}. 
\]

Hence, we can rewrite the objective function value as

\[
\sum_j C_j = \sum_{i=1}^{m} \sum_{k=1}^{\ell_i} C_{z_{ik}} = \sum_{i=1}^{m} \sum_{k=1}^{\ell_i} p_{i,x} z_{ik} = \sum_{i=1}^{m} \sum_{k=1}^{\ell_i} k p_{i,x} z_{ik}. \tag{11.1}
\]

In other words, the \( k \)th from last job run on a machine contributes exactly \( k \) times its processing time to the sum of completion times. This observation can be used to formulate \( R|| \sum C_j \) as a bipartite matching problem.

We define a complete bipartite graph \( G = (J, E) \) with \( J = \{1, \ldots, n\} \) contains a vertex for each job. The set \( B \) contains \( nm \) nodes \( w_{ik} \), where vertex \( w_{ik} \) represents the \( k \)th-from-last position on machine \( i \). The edge \((j, w_{ik})\) has weight \( k p_{ij} \).

**Lemma 11.4** A minimum weight matching in \( G = (J, E) \) subject to the constraint that every vertex in \( J \) is matched corresponds to an optimal schedule. Conversely, for every schedule there is a matching of the same weight.

**Proof:** The second part of the claim follow immediately from (11.1): If we construct the matching from the assignment of jobs to schedule positions, the objective function value will be exactly the cost of the matching.

For the other direction observe that not every matching which matches all vertices in \( J \) corresponds to a schedule, since a job might be assigned to the \( k \)th from last position while less than \( k \) jobs are assigned to that machine. However, such a matching is not of minimum cost. Hence, it follows that every minimum cost matching must »fill the first slots first« and hence, in fact, gives a schedule whose cost corresponds to that of the matching. \( \Box \)

**Corollary 11.5** The problem \( R|| \sum C_j \) can be solved in polynomial time. \( \Box \)

Since \( P|| \sum C_j \) is a special case of \( R|| \sum C_j \), Corollary 11.5 also shows that \( P|| \sum C_j \) is solvable in polynomial time. But let us have a closer look at this problem. In the identical parallel machines case a job needs the same amount of processing time no matter on which machine it is run. Hence, (11.1) simplifies to

\[
\sum_j C_j = \sum_{i=1}^{m} \sum_{k=1}^{\ell_i} k p_{i,x} z_{ik}. \tag{11.2}
\]

Equation (11.2) shows that a job as the \( k \)th from last job contributes exactly \( k \) times its processing time (which now is independent of the machine it is run) to the objective function value. Hence, the optimal schedule must look like as follows: schedule the \( m \) largest jobs last on each machine, schedule the next \( m \) largest jobs next to last, etc. This schedule constructed is exactly that constructed by the generalization of the SPT algorithm (see Theorem 10.1 on page 131 to the parallel machines case:

**Shortest processing time (SPT)** Order the jobs by nondecreasing processing time (breaking ties arbitrarily) and schedule in that order.

**Corollary 11.6** SPT is an exact algorithm for \( P|| \sum C_j \). \( \Box \)
11.3 Linear Programming

We have already seen that in the case of a single machine, Linear Programming can be used to obtain approximation algorithms with provable performance guarantees. We show another application in the parallel machine case. Consider the problem \( R_j | C_{\max} \).

We formulate the problem as an Integer Linear Program. The binary variable \( x_{ij} \) is one if an only if job \( j \) is assigned to machine \( i \). Observe that it does not matter in which position the job is. Our Integer Linear Program is:

\[
\begin{align*}
\text{min } D & \quad \text{(11.3a)} \\
\sum_{i=1}^{m} x_{ij} = 1 & \quad \text{for } j = 1, \ldots, n \quad \text{(11.3b)} \\
\sum_{j=1}^{n} p_{ij} x_{ij} & \leq D \quad \text{for } i = 1, \ldots, m \quad \text{(11.3c)} \\
x_{ij} \in \{0, 1\} & \quad \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n \quad \text{(11.3d)}
\end{align*}
\]

Equations (11.3b) ensure that each job is scheduled, Inequalities (11.3c) ensure that the makespan on every machine is at most \( D \). Of course, (11.3) is still NP-hard to solve. However, we can solve the Linear Programming Relaxation obtained by replacing (11.3d) by \( x_{ij} \geq 0 \).

Consider a solution \( \tilde{x} \) of the LP-relaxation. Without loss of generality we can assume that \( \tilde{x} \) is already a basic feasible solution which by well known results from Linear Programming (see e.g. [20]) has at most \( n + m \) positive variables. Since these \( n + m \) positive variables must be distributed among the \( n \) jobs, it follows that there are at most \( m \) jobs that are assigned to machines in a fractional way to more than one machine.

If \( x_{ij} = 1 \), we assign job \( j \) to machine \( i \). Call the schedule of these jobs \( S_1 \). Since we have solved a relaxation of the original problem, the length of \( S_1 \) is at most \( \text{OPT} \). As argued above, there are at most \( m \) jobs remaining. For these jobs we find an optimal schedule \( S_2 \) by complete enumeration in time at most \( O(m^m) \). The final solution consists of the composition of \( S_1 \) and \( S_2 \). The makespan of this solution is at most \( 2\text{OPT} \), since \( S_2 \) is an optimal schedule for a subset of all the jobs.

**Theorem 11.7** There is a 2-approximation algorithm for \( R_j | C_{\max} \) which runs in time \( O(m^m) \).

The drawback of our algorithm is that the running time is prohibitive, if \( m \) is large. Still, for a fixed number of machines, this running time is polynomial. We will show how to improve the running time of the above algorithm (which is dominated by the enumeration for the split jobs) in the next section by a new technique.

11.4 More Lower Bounds on the Optimal Schedule

By now it should have become clear that for proving approximation results (as well as competitiveness results) one inevitably needs good lower bounds for the optimal solution. In the previous section we used Linear Programming to get such bounds. We will now introduce more combinatorial lower bounds for parallel machine scheduling problems with the objective function of minimizing the weighted sum of completion times \( \sum_j w_j C_j \).

**Observation 11.8** Trivially, \( \sum_j w_j (r_j + p_j) \) is a lower bound on the value of an optimal solution since the completion time of a job \( j \) is always at least as large as its release date plus its processing time.
The second lower bound is based on the following single machine relaxation. Suppose that we are given an instance $I$ of the problem $P|r_j, pmtn| \sum w_j C_j$. We define an instance $I_1$ for a single machine and with the same set of jobs as in $I$. The single machine is processing with a speed $m$ times faster than the original machines, such that each job in instance $I_1$ has processing time $p'_j = p_j/m$. The release date and weight of each job in $I_1$ is equal to the ones of the corresponding job in $I$.

We claim that $I_1$ is a relaxation of $I$ in the sense stated as in the following lemma.

**Lemma 11.9** The value of an optimal schedule to $I_1$ is a lower bound on the value of an optimal schedule to $I$.

**Proof:** We can convert any feasible schedule for input $I$ to a feasible schedule to input $I_1$, without increasing the total weighted completion time by the following construction: Take a feasible schedule to instance $I$ and consider a time period $[t, t+1)$. We schedule an $1/m$ fraction of the processing or idle time that is scheduled on each of the parallel machines in that time period on the single machine in an arbitrary order in the same time interval. Figure 11.2 illustrates this mapping procedure for two machines.

![Conversion of a feasible non-preemptive parallel machine schedule $S(I)$ into a feasible preemptive single machine schedule $S(I_1)$](image)

Figure 11.2: Conversion of a feasible non-preemptive parallel machine schedule $S(I)$ into a feasible preemptive single machine schedule $S(I_1)$.

Obviously, this schedule is feasible, and its value is not greater than the value of the parallel machine schedule. This conversion of schedules is clearly feasible for an optimal parallel machine schedule, as well, and since the value of an optimal schedule is at most as large as the value of any feasible schedule, the claim follows.

Furthermore, we will need the following very fundamental result by McNaughton [21], who proved that no finite number of preemptions can improve the value of a schedule in absence of release dates.

**Lemma 11.10** If $P$ is a schedule for an instance of problem $P|pmtn| \sum w_j C_j$ with at least one preempted job, then there is a schedule $S$ for the same instance without any preempted job and with a value equal to the value of $P$.

In the following we assume that jobs are numbered in the non-increasing order of the values of their ratios $w_j/p_j$ such that $w_1/p_1 \geq \cdots \geq w_n/p_n$. We introduce our lower bound on the value of the optimal schedule for the parallel machine problem.
Lemma 11.11 The value of the optimal schedule for any instance $I$ of the problem $P[r_j, \text{pmtn}] \sum w_j C_j$ is bounded from below by

$$\sum_{j \in I} w_j \sum_{k \leq j} \frac{p_k}{m}.$$ 

Proof: Consider an instance $I$ of the problem $P[r_j, \text{pmtn}] \sum w_j C_j$. With Lemma 11.9 we can relax the parallel machine problem to a preemptive single machine problem on a machine with speed $m$ times faster than the original parallel machines without increasing the optimal objective value. After relaxing the release dates, McNaughton’s result in Lemma 11.10 applied to our special case $m = 1$ states that the values of the optimal schedules for the problems $1|\text{pmtn}| \sum w_j C_j$ and $1|| \sum w_j C_j$ are equal. The problem $1|| \sum w_j C_j$ can be solved optimally with the SMITH-rule (Theorem 10.2) achieving an objective value of $\sum_{j \in I} w_j \sum_{k \leq j} P_k$. We replace the modified processing times of the single machine instance $I$ by the regarding data of the original instance $I$. Then, the value of the optimal schedule to instance $I$ is bounded from below by $\sum_{j \in I} w_j \sum_{k \leq j} \frac{p_k}{m}$.

Trivially, allowing preemption relaxes the non-preemptive problem $P[r_j] \sum w_j C_j$. Thus, the lower bound above is a lower bound on the optimal value of the non-preemptive problem, as well.

11.5 Relaxed Decision Procedures and Linear Programming

A very useful definition for the design of approximation algorithms was introduced by Hochbaum and Shmoys in [13]:

Definition 11.12 (Relaxed Procedure)

A $\rho$-relaxed procedure $\text{TEST}$ for a minimization problem $\Pi$ has the following features. Let $\text{OPT}(I)$ denote the optimum value of any feasible solution for an instance $I$ of $\Pi$. Given a candidate value $M$ to be $\text{OPT}(I)$, the procedure $\text{TEST}(M)$ has two possible outcomes, namely “certificate of failure” ("COF") or success. These two outcomes have the following two implications:

1. If $\text{TEST}$ returns a COF then it is certain that $\text{OPT}(I) > M$.
2. If $\text{TEST}$ succeeds, it outputs a solution whose objective function value is at most $\frac{\rho \cdot M}{\rho}$. 

Observe that a $\rho$-relaxed procedure, called with a test value $\frac{\text{OPT}(I)}{\rho} \leq M < \text{OPT}(I)$, has the choice of returning a COF or delivering an answer of objective value at most $\rho \cdot M$.

Let $\Pi$ be a minimization problem. Suppose that, given an instance $I$ of $\Pi$ we know that $\text{OPT}(I)$ is an integer contained in the interval $[m_I, M_I]$. Then the existence of a $\rho$-relaxed procedure $\text{TEST}$ suggests the following approach to obtain an approximation algorithm for $\Pi$: We search the interval $[m_I, M_I]$ for $\text{OPT}(I)$ by examining possible candidate values $M$ with the help of Procedure $\text{TEST}$. If the test fails, we can conclude that $M$ is strictly smaller than the optimum value. On the other hand, if $\text{TEST}$ succeeds, it guarantees a solution of value at most $\rho \cdot M$. Thus, if we find the minimum candidate value in the interval such that $\text{TEST}$ succeeds, we will obtain a $\rho$-approximation for the optimum solution.

Of course, the above sketched idea is by no means a complete guide to a polynomial time approximation algorithm. To guarantee polynomiality of the algorithm obtained this way,
we will have to bound the running time of \textsc{Test} and the number of calls to the relaxed procedure \textsc{Test} appropriately. Also, if \textsc{Opt}(I) is not necessarily an integer, one will have to undertake appropriate steps. Nevertheless, the idea of relaxed decision procedures often proves to be a very useful tool.

One way to keep the number of calls to a relaxed test procedure small is to use \textit{binary search}. However, we must make sure that such a binary search will work. Similarly as above, assume that \textsc{Opt}(I) is an integer in the interval \([m_I, M_I]\). The possible problem is that for a test parameter \(M\) satisfying \(\textsc{Opt}(I)/\rho < M < \textsc{Opt}(I)\) we have no knowledge about whether the \(\rho\)-relaxed procedure \textsc{Test} will deliver a solution or output COF.

Nevertheless, it turns out that this is not grave. We can still run a binary search as shown in Algorithm 11.1 simply ignoring the uncertainty caused by the relaxed procedure.

**Algorithm 11.1 Binary search guided by a relaxed procedure.**

Input: A lower bound \(m_I\) and an upper bound \(M_I\) for the optimum value

1. if \textsc{Test}(\(m_I\)) \(\neq\) COF then
2. return \textsc{Test}(\(m_I\))
3. end if
4. \(high \leftarrow M_I\)
5. \(low \leftarrow m_I\)
6. while \((high - low) > 1\) do
7. \(i \leftarrow \left\lceil \frac{high + low}{2} \right\rceil\)
8. if \textsc{Test}(\(i\)) = COF then
9. \(low \leftarrow i\)
10. else
11. \(high \leftarrow i\)
12. end if
13. end while
14. return \textsc{Test}(\(high\))

**Lemma 11.13** Suppose that \textsc{Opt}(I) is an integer in the interval \([m_I, M_I]\) and \textsc{Test} is a \(\rho\)-relaxed procedure. Then the "binary search" in Algorithm 11.1 terminates with a value of \(high\) satisfying \(\textsc{Opt}(I) \leq high\) and \textsc{Test}(\(high\)) \(\neq\) COF. The objective function value of the solution output at the end of the binary search is at most \(\rho \cdot \textsc{Opt}(I)\).

**Proof:** If the test in Step 1 succeeds, i.e. if \textsc{Test}(\(m_I\)) \(\neq\) COF, then by the assumption that \textsc{Opt}(I) \(\in\) \([m_I, M_I]\) we have \textsc{Opt}(I) = \(m_I\) and Algorithm 11.1 returns an optimal solution. Thus, we will assume for the rest of the proof that \textsc{Test}(\(m_I\)) = COF.

As an invariant throughout the algorithm we have \textsc{Test}(\(low\)) = COF and \textsc{Test}(\(high\)) \(\neq\) COF. In particular, upon termination \textsc{Test}(\(high\)) is also not a COF.

To establish the rest of the lemma we distinguish between two cases. In the first case, the procedure \textsc{Test} is called for some \(M\) in the critical range, i.e. \(\textsc{Opt}/\rho \leq M < \textsc{Opt} = \textsc{Opt}(I)\) and does not return a COF for this test value. Then, by construction of the algorithm, \(high \leq M \leq \textsc{Opt}\) upon termination, which shows the claim. In the second case, \textsc{Test}(\(M\)) = COF for all \(\textsc{Opt}/\rho \leq M < \textsc{Opt}\) that are probed during the search. In this case, it is easy to see that \(high = \textsc{Opt}\), when the algorithm terminates.

The fact that the objective function value of the solution output at the end of the search is at most \(\rho \cdot \textsc{Opt}\) is a direct consequence of \textsc{Test}(\(high\)) \(\neq\) COF and the fact that \textsc{Test} is \(\rho\)-relaxed. \(\square\)
Lemma 11.13 enables us to perform a binary search even if we are dealing with relaxed decision procedures instead of exact ones. Thus, in the sequel we can restrict ourselves to showing that a certain procedure is relaxed (for some value $\rho$) and then simply refer to the binary search technique in order to make a fast search of a certain range possible.

Let us go back to the problem $R\|C_{\text{max}}$. Our first step in designing a relaxed decision procedure for the problem is to reconsider the Integer Linear Program (11.3). Instead of solving the »standard relaxation« we will add additional constraints to the relaxation in order to ensure that the Linear Program carries enough information. Specifically, our relaxation looks as follows:

\[ \sum_{i=1}^{m} x_{ij} = 1 \quad \text{for } j = 1, \ldots, n \]  \\
\[ \sum_{j=1}^{n} p_{ij} x_{ij} \leq D \quad \text{for } i = 1, \ldots, m \]  \\
\[ x_{ij} = 0 \quad \text{if } p_{ij} \geq D \]  \\
\[ x_{ij} \geq 0 \quad \text{for } i = 1, \ldots, m \text{ and } j = 1, \ldots, n \]

Constraint (11.4c) is actually implicit in the Integer Linear Program (11.3), but is not automatically true for the solution of the »standard relaxation«.

Our relaxed decision procedure works as follows. We solve the Linear Program (11.4). If (11.4) has no feasible solution, then we can be sure that (11.3) has no feasible solution and thus $\text{OPT} > D$, that is, the optimal makespan is strictly larger than $D$. What happens if the relaxation is feasible? We show that in this case we can round the solution of (11.3) to a feasible schedule of length at most $2D$. Hence, the above construction yields a 2-relaxed decision procedure for $R\|C_{\text{max}}$.

Consider a solution $\tilde{x}$ of (11.4). Again, without loss of generality we can assume that $\tilde{x}$ is already a basic feasible solution which by well known results from Linear Programming (see e.g. [20]) has at most $n + m$ positive variables. Since these $n + m$ positive variables must be distributed among the $n$ jobs, it follows that there are at most $m$ jobs that are assigned to machines in a fractional way to more than one machine.

If $\tilde{x}_{ij} = 1$, we assign job $j$ to machine $i$. Call the schedule of these jobs $S_1$. As argued above, there are at most $m$ jobs remaining. Let these jobs be $J_2$. We construct a bipartite graph $G = (V, E)$ with $V = J_2 \cup \{1, \ldots, m\}$. An edge $(j, i)$ is present if $0 < \tilde{x}_{ij} < 1$.

**Lemma 11.14** The bipartite graph $G = (V, E)$ as defined above has a matching in which each job $j \in J_2$ is matched.

**Proof:** See [19].

We use the matching according to Lemma 11.14 to schedule the jobs in $J_2$. Call the resulting schedule $S_2$. The overall schedule consists of the composition of $S_1$ and $S_2$ and has length at most the sum of the makespans of $S_1$ and $S_2$.

Clearly, the makespan of $S_1$ is at most $D$, since the $\tilde{x}$ form a feasible solution to (11.4). On the other hand, the makespan of $S_2$ is also at most $D$, since in $S_2$ each machine gets at most one job and constraint (11.4c) ensured that $x_{ij} = 0$ if $p_{ij} > D$. Hence, we have in fact a 2-relaxed decision procedure. We can search for the optimal schedule length by binary search and transform this 2-relaxed procedure into a 2-approximation.

**Theorem 11.15** There is a 2-approximation algorithm for $R\|C_{\text{max}}$. □
11.6 Preemptive and Fractional Schedules as Bases for Approximation

Recall that in Section 10.5 we used an optimal solution for the preemptive problem 1|[ri,j,pmnt]Cj to obtain a 2-approximation for 1|[ri]Cj. The problem 1|[ri,j,pmnt]Cj could be solved easily in polynomial time by the SRPT-rule.

In this section we apply similar techniques to the problem \( P|\{r_j\}, \text{pmtn}| \sum w_j C_j \). As a warmup, let us recall the conversion algorithm and prove a bound for parallel machines.

**Convert preemptive schedule (CONVERT-PREEMPTIVE)** Form a list \( L \) of the jobs, ordered by their preemptive completion times \( C_j^P \). Use the list scheduling algorithm LS with the constraint that no job starts before its release date.

**Lemma 11.16** Given a preemptive schedule \( P \) for \( P|\{r_j\}, \text{pmtn}| \sum w_j C_j \), algorithm CONVERT-PREEMPTIVE produces a nonpreemptive schedule \( S \) such that for each job \( j \):

\[
C_j^S \leq 3C_j^P.
\]

**Proof:** Suppose that \( C_j^P \leq C_j^P \leq \cdots \leq C_j^P \). We simplify the analysis of CONVERT-PREEMPTIVE by specifying that the list scheduling algorithm will not start the \( j \)th job in \( L \) until the \((j-1)\)st job is started. Clearly, this additional restriction can only worsen the schedule.

Job \( j \) is not added to schedule \( S \) until jobs \( 1, \ldots, j-1 \) are added. Thus \( j \) will not start before \( 1, \ldots, j-1 \) have started. Let \( r_j^* \) be the maximum release time of a job in \( 1, \ldots, j-1 \). Since \( r_j^* \) is the release date of a job that finished no later than \( C_j^P \) in the preemptive schedule, we have \( r_j^* \leq C_j^P \). By definition, all jobs \( 1, \ldots, j-1 \) have been released by time \( r_j^* \).

Even if no processing was done in \( S \) before time \( r_j^* \), at least one machine will have no more processing to do on \( 1, \ldots, j-1 \) at time \( \frac{1}{m} \sum_{i=1}^{j-1} p_i \). Observe that, since all jobs \( 1, \ldots, j-1 \) completed by time \( C_j^P \), we have \( \frac{1}{m} \sum_{i=1}^{j-1} p_i \leq C_j^P \). Hence, job \( j \) is started in the nonpreemptive schedule no later than time

\[
r_j^* + \frac{1}{m} \sum_{i=1}^{j-1} p_i \leq r_j^* + C_j^P.
\]

The completion time of \( j \) in \( S \) thus satisfies \( C_j^S \leq r_j^* + C_j^P + p_j \leq 3C_j^P \).

Unfortunately, Lemma 11.16 is not directly useful, since \( P|\{r_j\}, \text{pmtn}| \sum w_j C_j \) is still an NP-hard problem. To get around this difficulty, we refine our conversion techniques.

Define a fractional preemptive schedule as one in which a job \( j \) can be scheduled simultaneously with several other jobs on one machine, where a job receives some fraction of the machines resources and the sum of all the fractions assigned to a machine sum to at most one at any moment in time. More formally, let \( \gamma_j^i(t) \) be the fractional amount of machine \( i \) assigned to \( j \) at time \( t \). We view time as continuous and set \( T \) to be the completion time of the schedule. We require the following conditions for a fractional preemptive schedule:

\[
\sum_{j=1}^{n} \gamma_j^i(t) \leq 1 \quad \text{for all } t \in [0, T] \text{ and } i = 1, \ldots, m \tag{11.5a}
\]

\[
\sum_{i=1}^{m} \int_{0}^{T} \gamma_j^i(t) dt = p_j \quad \text{for all } j = 1, \ldots, n \tag{11.5b}
\]
We also require that for any pair \( j, t \) at most one value \( \gamma_j^t(t) \) is positive among the \( m \) values and that all \( \gamma_j^t(t) \) are zero if \( t \leq r_j \). The completion time of a job in a fractional preemptive schedule is the last moment \( t \) in time, when one of the \( \gamma_j^t(t) \) is positive.

The proof of the following Lemma is exactly the same as that for Lemma 11.16 (where we did not exploit the fact that \( P \) was not fractional at all!).

**Lemma 11.17** Given a fractional preemptive schedule for \( P | r_j, \text{pmtn} | \sum w_j C_j \), algorithm CONVERT-PREEMPTIVE produces a nonpreemptive schedule \( S \) such that for each job \( j \):

\[
C_S^j \leq 3C_P^j.
\]

In the next section we design a 2-approximation algorithm for the problem \( P | r_j, \text{pmtn} | \sum w_j C_j \). Combining this algorithm with the conversion algorithm yields a 6-approximation for \( P | r_j | \sum w_j C_j \).

### 11.7 Preemptive scheduling on parallel machines

We consider the following extension of the preemptive WSPT rule (see Section 10.1) for the single machine problem to the parallel machine environment.

**Algorithm 11.2** Algorithm P-WSPT

At any time, schedule the \( m \) jobs with the smallest indices among the available, not yet completed jobs (or fewer if less than \( m \) unscheduled jobs are available). Interrupt the processing of currently active jobs, if necessary.

Algorithm P-WSPT works on-line since the decision about which job to run is at any time just based on the set of available jobs.

**Theorem 11.18** Algorithm P-WSPT produces a solution of objective function value at most twice the optimal value for the off-line problem \( P | r_j, \text{pmtn} | \sum w_j C_j \).

**Proof:** Assume again that jobs are indexed in the non-increasing order of the values of their ratios \( w_j/p_j \). Consider the time interval \( [r_j, C_j] \) of an arbitrary but fixed job \( j \). We partition this interval into two disjunctive sets of subintervals: \( I(j) \) contains the subintervals in which job \( j \) is being processed, and \( \bar{I}(j) \) denotes the set of remaining subintervals, in which other jobs than \( j \) are being processed. Note that no machine can be idle during the subintervals belonging to \( I(j) \). Since the algorithm processes job \( j \) after its release date \( r_j \) whenever a machine is idle, we obtain

\[
C_j \leq r_j + |I(j)| + |\bar{I}(j)|,
\]

where \(| \cdot |\) denotes the sum of the lengths of the subintervals in the corresponding set.

The overall length of \( I(j) \) is clearly \( p_j \). Only jobs with a higher ratio of weight to processing time than \( j \) can be processed during the intervals of the set \( I(j) \), because the algorithm gives priority to \( j \) before scheduling jobs with lower ratio. In the worst case, that is when \( |\bar{I}(j)| \) is maximal, all jobs with higher priority than \( j \) are being processed in the subintervals of this set. Then \( |\bar{I}(j)| = (\sum_{k<j} p_k)/m \), and we can bound the value of the P-WSPT schedule as follows:

\[
\sum_j w_j C_j \leq \sum_j w_j (r_j + p_j) + \sum_j w_j \sum_{k<j} \frac{p_k}{m}.
\]

Applying the lower bounds introduced in Section 11.4 we can conclude that the P-WSPT algorithm is 2-competitive. □
This result is best possible for that algorithm.

11.8 Non-preemptive scheduling on parallel machines

In the single machine setting (see Section 10.2) the conversion technique CONVERT-PREEMPTIVE turned out to be a very useful tool for coping with non-preemption. We apply the same idea in the parallel setting and give a very general result, first. Let $C^P_j$ denote the completion time of job $j$ in the preemptive schedule, and let $C^S_j$ denote its completion time in the non-preemptive schedule. Additionally, we assume a different ordering of jobs in this Section: let them be numbered according to their completion time $C^P_j$ in the preemptive schedule.

Lemma 11.19 Given a preemptive schedule for instance $I$ of problem $P|\mathit{r}_j, \mathit{pmtn}| \sum \mathit{w}_j \mathit{C}_j$, any conversion algorithm (that uses a preemptive schedule in order to obtain a non-preemptive one by list scheduling according to the completion times $C^P_j$) generates a non-preemptive $m$-machine schedule $S$ for instance $I$ in which for each job $j$ its completion is bounded from above as follows:

$$C^S_j \leq 2 \cdot C^P_j + (1 - \frac{1}{m}) p_j.$$  

Proof: Consider a particular job $j$ and its completion time $C^S_j$ in the non-preemptive schedule. The algorithm starts processing job $j$ at time $C^P_j$, if a machine is idle and $j$ is the available job with the highest priority in the waiting list. In this list, only jobs $1, 2, \ldots, j-1$ precede $j$, and hence, only these jobs start processing in the non-preemptive schedule before job $j$. Since these preceding jobs are being processed non-preemptively, at least one machine will be idle after $\sum_{k<j} \frac{p_k}{m}$ units of processing time. Clearly, all jobs $1, \ldots, j$ have been released by $C^P_j$. Job $j$ will complete then by

$$C^S_j \leq C^P_j + \sum_{k<j} \frac{p_k}{m} + p_j.$$  \hspace{1cm} (11.6)

All jobs $1, \ldots, j$ have completed by time $C^P_j$ in the preemptive schedule and thus, $\sum_{k<j} \frac{p_k}{m} \leq C^P_j$. Hence, the claim follows. \hfill $\Box$

Note, that the above result holds for any off-line and on-line conversion algorithm.

A straightforward combination of the conversion technique with the 2-competitive Algorithm $P$-$\mathit{SRPT}$ (which is the parallel machine version of the single machine-algorithm $\mathit{SRPT}$ from Theorem 10.3 on page 132) leads to a 5-competitive online algorithm for the non-preemptive problem $P|\mathit{r}_j|\sum \mathit{C}_j$. We denote this algorithm by CONVERT-$P$-$\mathit{SRPT}$:

Algorithm 11.3 Algorithm CONVERT-$P$-$\mathit{SRPT}$

- Simulate sequencing jobs preemptively by the $P$-$\mathit{SRPT}$-Algorithm (parallel version of Algorithm $\mathit{SRPT}$). Convert the preemptive $m$-machine schedule into a non-preemptive one by scheduling jobs non-preemptively in order of their completion times in the preemptive schedule.

In the simulation of the $P$-$\mathit{SRPT}$-Algorithm we respect the release dates and also for the conversion of the preemptive schedule into a non-preemptive one we do not need any knowledge of jobs arriving in the future. Thus, this algorithm is online.
Lemma 11.20 Algorithm CONVERT-P-SRPT is $(5 - 1/m)$-competitive for the problem $P_{|r_j|} \sum C_j$.

Proof: We apply Lemma 11.19 for each job $j \in I$ and obtain for the value of the CONVERT-P-SRPT-schedule $S$

$$\sum_{j \in I} C_j^S \leq \sum_{j \in I} 2 \cdot C_j^P + \sum_{j \in I} (1 - \frac{1}{m}) p_j.$$ 

The preemptive schedule is a P-SRPT-schedule and we know that its value is at most twice the optimum value for the preemptive problem. The preemptive problem is a simple relaxation of the non-preemptive problem. Thus, the optimal value $Z^*$ to our problem $P_{|r_j|} \sum C_j$ is bounded from below by two times the optimal value to the relaxed preemptive problem. Furthermore, we use the trivial lower bound $\sum p_j \leq Z^*$ and obtain

$$\sum_{j \in I} C_j^m \leq 2 \cdot (2 \cdot Z^*) + (1 - \frac{1}{m}) Z^* = (5 - \frac{1}{m}) \cdot Z^*.$$ 

This completes the proof. $\square$

In the parallel machine setting, we have more options in the design of conversion algorithms than in the single machine environment. Recall the idea of simulating sequencing on a fast single machine with speed $m$ times faster than the original $m$ parallel machines which was introduced in Section 11.4 as a relaxation.

If we convert a preemptive schedule that was obtained by simulating sequencing jobs preemptively on a fast single machine then we obtain a $(3 - 1/m)$-competitive on-line algorithm CONVERT-SRPT for the non-weighted problem $P_{|r_j|} \sum C_j$. Note, that the refinement compared to the previous Algorithm CONVERT-P-SRPT lays in the preemptive schedule: instead of considering a preemptive schedule on $m$ machines we simulate, now, scheduling on a fast single machine.

Algorithm 11.4 Algorithm CONVERT-SRPT

We consider modified processing times $p_j' = \frac{p_j}{m}$ and simulate sequencing jobs preemptively by their shortest remaining modified processing times, known as SRPT-order, obtaining a preemptive single machine schedule. Convert the preemptive schedule into a non-preemptive one by applying list scheduling in the order of non-decreasing completion times in preemptive schedule.

The algorithm CONVERT-SRPT is an online algorithm, if we only simulate the preemptive schedule which respects the original release dates. We are able to prove that an adjusted version of Lemma 11.19 (using the same notation) still holds:

Lemma 11.21 Given a preemptive single machine schedule to instance $I$ with modified processing times, any conversion algorithm that applies list scheduling in order of non-increasing completion times of jobs in the preemptive single machine schedule generates a non-preemptive $m$-machine schedule to instance $I$ in which for each job $j$ its completion time is $C_j^m \leq 2 \cdot C_j^P + (1 - 1/m) p_j$.

Proof: By the same arguments as in the proof of Lemma 11.19 we obtain inequality (11.6) for our setting

$$C_j^S \leq C_j^P + \sum_{k<j} \frac{p_k}{m} + p_j$$

$$= C_j^P + \sum_{k \leq j} \frac{p_k}{m} + (1 - \frac{1}{m}) p_j.$$
Since $C^P_j$ is at least as big as the processing times of the jobs that precede it in the non-preemptive schedule, we have

$$C^P_j \geq \sum_{k < j} p'_k = \sum_{k < j} \frac{p_k}{m}.$$  

Moreover, it follows for the completion time of $j$ in the non-preemptive schedule that

$$C^S_j \leq 2 \cdot C^P_j + \frac{1}{m} \sum_{j \in I} p_j.$$  

This is what we wanted to show.

Now, we can prove the competitiveness result for Algorithm CONVERT-SRPT.

**Theorem 11.22** Algorithm CONVERT-SRPT is $(3 - 1/m)$-competitive for the problem $P|r_j| \sum C_j$.

**Proof:** We can apply Lemma 11.21 to each job $j \in I$ and obtain as a value of the non-preemptive schedule $S$ produced by CONVERT-SRPT:

$$\sum_{j \in I} C^S_j \leq 2 \cdot \sum_{j \in I} C^P_j + \frac{1}{m} \sum_{j \in I} p_j.$$  

The preemptive schedule has been obtained by the SRPT-rule and is, thus, optimal for the relaxed problem on the fast single machine. Therefore, the optimal objective value $Z^*$ of our parallel machine problem $P|r_j| \sum C_j$ is bounded $Z^* \geq \sum_{j \in I} C^P_j$. Applying the trivial lower bound $Z^* \geq \sum p_j$, as well, completes the proof.

## 11.9 Dynamic Programming and Polynomial Time Approximation Schemes

In this section we illustrate the design of polynomial time approximation schemes on the example of $P|j|C_{\text{max}}$. We first consider a special case of the problem which might sound very special at first, but which will come in handy a few moments later:

**Lemma 11.23** Given an instance of $P|j|C_{\text{max}}$ in which the processing times $p_j$ take on at most $s$ different values. Then, there exists an algorithm which finds an optimal solution in time $O(s \log n \cdot n^2)$.

**Proof:** Let $L$ be a target schedule length. We first show how to decide in time $O(n^2)$ by dynamic programming whether $\text{OPT} \leq L$. Let the different processing times by $v_1, \ldots, v_s$. Since the order of jobs on a machine does not matter in $P||C_{\text{max}}$ we can describe the assignment of jobs to a specific machine as a vector $A = (a_1, \ldots, a_s) \in \{0, \ldots, n\}^s$ where $a_i$ denotes the number of jobs with processing time $v_i$. The load (that is, the makespan) of the machine under this assignment is $\sum_{i=1}^s a_i v_i$.  

Let $\mathcal{A}$ be the set of all such vectors whose load does not exceed $L$. Observe that $|\mathcal{A}| = O(n^s)$. If $\text{OPT} \leq L$, then each machine can be assigned a vector from $\mathcal{A}$ (subject to the constraint that all jobs are assigned to machines).

For a vector $(x_1, \ldots, x_s) \in \{0, \ldots, n\}^s$ define $M(x_1, \ldots, x_s)$ be the minimum number of machines necessary to schedule $x_i$ jobs of size $v_i$ for $i = 1, \ldots, s$ such that no machine gets load larger than $L$. Then,

$$M(x_1, \ldots, x_s) = 1 + \min_{(a_1, \ldots, a_s) \in \mathcal{A}} M(x_1 - a_1, \ldots, x_s - a_s).$$  

(11.7)
The recursion (11.7) follows from the fact that in order to schedule the job vector 
\((x_1, \ldots, x_s)\) on \(k\) machines one machine must get some jobs according to some vector 
\((a_1, \ldots, a_s) \in A\).

Hence, with the help of (11.7) each table entry \(M(x_1, \ldots, x_s)\) can be computed in time 
\(O(n^s)\). Since there are \(O(n^s)\) table entries, this yields a total time of \(O(n^{2s})\). At the 
end of the dynamic program we can easily decide whether \(OPT \leq L\) by simply inspecting 
whether \(M(n_1, \ldots, n_s) \leq m\), where \(n_i\) is the number of jobs in the input instance with 
processing time \(v_i\).

So far, we have developed a method to decide in time \(O(n^{2s})\) whether \(OPT \leq L\) for a 
given value of \(L\). If \(OPT \leq L\) the dynamic program also yields a schedule with makespan 
at most \(L\). Observe that in an optimal solution the makespan is determined by the load 
on one machine which in turn corresponds to a vector \((a_1, \ldots, a_s) \in \{0, 1, \ldots, n\}^s\) such 
that \(OPT = \sum_{i=1}^s a_i v_i\). Hence, there are at most \(O(n^s)\) possible values for the optimum 
makespan and we can search for the smallest \(L\) among those such that \(OPT \leq L\) by binary 
search using \(O(\log n^s) = O(s \log n)\) calls to the dynamic program above.

We now turn to the general case of \(P||C_{\text{max}}\). As noted in Section 11.5 on page 146 it 
suffices to give a \((1 + \varepsilon)\)-relaxed decision procedure for any \(\varepsilon > 0\) which, given a target 
schedule length \(L\)

- either correctly outputs that \(OPT > L\); or
- produces a schedule of length at most \((1 + \varepsilon)L\).

We assume for the rest of the presentation that \(\varepsilon L, \varepsilon^2 L, 1/\varepsilon\) and \(1/\varepsilon^2\) are all integers. The 
proofs can easily be modified to handle the case of arbitrary rational numbers.

Call a job \(j\) large if \(p_j > \varepsilon L\). We assume for the moment that we have already a 
\((1 + \varepsilon)\)-relaxed decision procedure \textsc{BigJob-Relaxed-Decision} if the input instance 
consists only of big jobs. We then work as shown in Algorithm 11.6. We call \textsc{BigJob-Relaxed-Decision} 
for the set \(B = \{j : p_j > \varepsilon L\}\) of big jobs and the target schedule 
length \(L\). Clearly, if \textsc{BigJob-Relaxed-Decision} outputs a COF (certificate of failure) 
we know that \(OPT > L\), since adding the small jobs can only increase the optimal value. 
If \textsc{BigJob-Relaxed-Decision} does give us a schedule of length at most \((1 + \varepsilon)L\), then we 
try to add the small jobs to that schedule such that no machine gets load larger than 
\((1 + \varepsilon)L\).

**Lemma 11.24** RELAXED-DECISION SCHEME is a \((1 + \varepsilon)\)-relaxed decision procedure for 
\(P||C_{\text{max}}\).

**Proof:** We have already argued that if the algorithm returns a COF in Step 4 it is certain 
that \(OPT > L\). If the algorithm outputs a COF in Step 10 then at this moment in time every 
machine has load strictly larger than \(L\). Thus, the total processing time is more than \(mL\), 
which implies that \(OPT > L\).

If the algorithm does not output a COF, it is able to schedule all the small jobs in the for-loop. 
Before entering the loop, every machine has load at most \((1 + \varepsilon)L\) and small jobs 
are assigned only to machines with load at most \(L\). Since by definition, a small job has 
processing time at most \(\varepsilon L\), each machine ends up with load at most \((1 + \varepsilon)L\).

It remains to implement the relaxed decision procedure \textsc{BigJobs-Relaxed-Decision} 
for instances consisting only of big jobs. This algorithm works as shown in Algorithm 11.6. 
We can assume that \(\max_j p_j \leq L\), since otherwise it is immediately clear that \(OPT > L\). 
We scale the processing times \(p_j\) of all jobs down to the nearest integer multiple of \(\varepsilon^2 L\). 
This yields a new instance \(I'\) of \(P||C_{\text{max}}\). The following properties are satisfied:
Algorithm 11.5 \((1 + \varepsilon)\)-relaxed decision procedure for \(P|^C_{\text{max}}\)

\begin{algorithm}
\textsc{Relaxed-Decision-Scheme}(\(J, p_1, \ldots, p_n, L, \varepsilon\))
\begin{algorithmic}[1]
\State Let \(B := \{ j : p_j > \varepsilon L \}\) and \(K := \{ j : p_j \leq \varepsilon L \}\) be the set of big and small jobs, respectively.
\State Call \textsc{BigJob-Relaxed-Decision} for the set \(B\) of big jobs and the target schedule length \(L\).
\If {\textsc{BigJob-Relaxed-Decision} outputs a COF}
\State \textbf{return} COF
\Else
\For {each small job \(j\) in \(K\)}
\If {there is a machine \(i\) with load at most \(L\)}
\State assign job \(j\) to machine \(i\)
\Else
\State \textbf{return} COF
\EndIf
\EndFor
\EndIf
\end{algorithmic}
\end{algorithm}

(i) \(0 \leq p_j - p'_j \leq \varepsilon^2 L\) for all \(j\).

(ii) There are at most \(\frac{L}{\varepsilon^2 L} = \frac{1}{\varepsilon^2}\) different processing times.

(iii) In any schedule for \(I'\) with makespan at most \(L\), any machine has at most \(\frac{L}{\varepsilon^2 L} = \frac{1}{\varepsilon}\) jobs.

Property 11.9 enables us to compute an optimal solution for the instance \(I'\) by the dynamic programming algorithm of Lemma 11.23 in polynomial time (provided \(\varepsilon > 0\) is fixed). If the makespan \(D\) for \(I'\) is larger than \(L\), then we know that \(\text{OPT} > L\), since \(p'_j \leq p_j\) for all \(j\). This is the only place in which we output a COF.

If the optimal makespan for \(I'\) has length at most \(L\), we use the schedule for \(I'\) to schedule the jobs. Of course, this will increase the makespan, since each processing time increases by at most \(\varepsilon^2 L\). However, due to property 11.9 no machine in the optimal schedule for \(I'\) had more than \(1/\varepsilon\) jobs and thus, the total load on each machine increases by at most \(1/\varepsilon \cdot \varepsilon^2 L = \varepsilon L\). Hence, we end up with a makespan of at most \((1 + \varepsilon)L\) and \textsc{BigJob-Relaxed-Decision} is in fact a \((1 + \varepsilon)\)-relaxed decision procedure for the case of only big jobs.
Algorithm 11.6 \((1 + \varepsilon)\)-relaxed decision procedure for \(P||C_{\text{max}}\) provided only big jobs with \(p_j > \varepsilon L\) exist.

\textsc{BigJob-Relaxed-Decision}(J, p_1, \ldots, p_n, L, \varepsilon)

1. For each job \(j\) define a new processing time \(p'_j\) by rounding down \(p_j\) to the nearest integer multiple of \(\varepsilon^2 L\).
2. Use the dynamic programming algorithm from Lemma 11.23 to obtain an optimal solution for the instance \(I'\) with processing times \(p'_j\). Let \(D\) be its makespan.
3. \textbf{if} \(D > L\) \textbf{then}
4. \hspace{1em} \textbf{return} COF
5. \textbf{else}
6. \hspace{1em} Create a schedule for the original instance by scheduling the jobs as in the optimal schedule for \(I'\) (shifting jobs due to the expanded processing times) and return this schedule.
7. \textbf{end if}


