MATTHIAS GERDTS

COMBINATORIAL OPTIMISATION
MSM 3M02b

Address of the Author:

Matthias Gerdts

Computational Optimisation Group
School of Mathematics
University of Birmingham
Edgbaston
Birmingham B15 2TT

E-Mail: gerdtsm@maths.bham.ac.uk
WWW: web.mat.bham.ac.uk/M.Gerdts

Preliminary Version: March 16, 2009

Contents

Lecture plan

Lectures:

Date	Hours	Pages
12.1.2009	L1	1–8
14.1.2009	L1	8–11
19.1.2009	L1	11–15
21.1.2009	L1	15–18
26.1.2009	L1	19–23
28.1.2009	L1	24–29
02.2.2009	L1	29–33
04.2.2009	L1	33–38
09.2.2009	L1	38–41
11.2.2009	L1	41–42
16.2.2009	L1	43
18.2.2009	L1	CT
23.2.2009	L1	44–46
25.2.2009	L1	47–50
02.3.2009	L1	51–53
04.2.2009	L1	54–57
09.3.2009	L1	58–59
11.3.2009	L1	59–63
16.3.2009	L1	64–
18.3.2009	L1	CT
23.3.2009	L1	
25.3.2009	L1	

Chapter 1

Notation

Numbers: $\mathbb{R}$ is the set of reel numbers, $\mathbb{Z}$ is the set of integers, $\mathbb{N}$ is the set of positive integers, $\mathbb{N}_0$ is the set of non-negative integers.

Vectors and matrices: A vector $x \in \mathbb{R}^n$ is a column vector with components $x_1, x_2, \ldots, x_n$:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}.$$

$A \in \mathbb{R}^{m \times n}$ is a matrix with m rows and n columns and entries a_{ij}, $i = 1, \ldots, m$, $j = 1, \ldots, n$:

$$A = (a_{ij})_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \vdots & a_{mn} \end{pmatrix}.$$

$A^\top$ denotes the transposed matrix of the matrix A.
$\|\cdot\| = \|\cdot\|_2$ denotes the Euclidean norm in $\mathbb{R}^n$, i.e. for $x = (x_1, \ldots, x_n)^\top$ it holds $\|x\| = \sqrt{x^\top x} = \sqrt{\sum_{i=1}^n x_i^2}$.

Chapter 2

Introduction

Combinatorial optimisation is a branch of optimisation in applied mathematics and computer science, related to operations research, algorithm theory and computational complexity theory. Combinatorial optimisation algorithms solve instances of problems that are believed to be hard in general. Usually, the set of feasible solutions of combinatorial optimisation problems is discrete or can be reduced to a discrete one, and the goal is to find the best possible solution. Most often, the feasible set contains a finite number of points only, but the number grows very rapidly with increasing problem size.
There are very many examples of combinatorial optimisation problems such as

- routing and shortest path problems
- matching and assignment problems
- VLSI design problems
- traveling salesman problems
- knapsack problems
- scheduling problems
- network optimisation problems (network flows, transportation problems)
- sequence alignment problems in DNA sequencing
- (mixed-)integer linear or nonlinear programming
- and many more...

2.1 Efficiency and Complexity

Our main aim is to construct efficient algorithms for combinatorial optimisation problems – if possible at all. However, it will turn out that for certain problems no efficient algorithm is known as yet. Even worse, for these problems an efficient algorithm is unlikely to exist. What is an efficient algorithm?

Example 2.1.1 (Polynomial and exponential complexity)
Suppose a computer with 10^{10} ops/sec (10 GHz) is given. A problem P (e.g. a linear program) has to be solved on the computer. Let n denote the size of the problem (e.g. the number of variables). Let five different algorithms be given, which require n, n^2, n^4, 2^n, and $n!$ operations (e.g. steps in the simplex method), respectively, to solve the problem P of size n.
The following table provides an overview on the time which is needed to solve the problem.

ops \ size n	*20*	*60*	$\cdots$	*100*	*1000*
n	0.002 μs	0.006 μs	$\cdots$	0.01 μs	0.1 μs
n^2	0.04 μs	0.36 μs	$\cdots$	1 μs	0.1 ms
n^4	16 μs	1.296 ms	$\cdots$	10 ms	100 s
2^n	0.1 ms	3 yrs	$\cdots$	10^{12} yrs	$\cdot$
$n!$	7.7 yrs	$\cdot$	$\cdot$	$\cdot$	$\cdot$

Clearly, only the algorithms with polynomial complexity n, n^2, n^4 can be regarded as efficient while those with exponential complexity 2^n and $n!$ are very inefficient.

We don't want to define the complexity or effectiveness of an algorithm in a mathematically sound way at this stage. Instead we use the following informal but intuitive definition:

Definition 2.1.2 (Complexity (informal definition))
A method for solving a problem P has polynomial complexity if it requires at most a polynomial (as a function of the size) number of operations for solving any instance of the problem P. A method has exponential complexity, if it has not polynomial complexity.

To indicate the complexity of an algorithm we make use of the $\mathcal{O}$-notation. Let f and g be real-valued functions. We write

$$f(n) = \mathcal{O}(g(n))$$

if there is a constant C such that

$$|f(n)| \leq C|g(n)| \qquad \text{for every } n.$$

In complexity theory the following problem classes play an important role. We don't want to go into details at this stage and provide informal definitions only:

- The class $\mathcal{P}$ consists of all problems for which an algorithm with polynomial complexity exists.
- The class $\mathcal{NP}$ (Non-deterministic Polynomial) consists of all problems for which an algorithm with polynomial complexity exists that is able to certify a solution of

the problem. In other words: If a solution x of the problem P is given, then the algorithm is able to certify in polynomial time that x is actually a solution of P. On the other hand, if x is not a solution of P then the algorithm may not be able to recognise this in polynomial time.

Clearly, $\mathcal{P} \subseteq \mathcal{NP}$, but it is an open question whether $\mathcal{P} = \mathcal{NP}$ holds or not. If you are the first to answer this question, then the Clay Institute (www.claymath.org) will pay you one million dollars, see http://www.claymath.org/millennium/ for more details. It is conjectured that $\mathcal{P} \neq \mathcal{NP}$. Good luck!

2.2 Examples

Example 2.2.1 (Assignment Problem)

Given:

- *n employees $E_1, E_2, \dots, E_n$*
- *n tasks $T_1, T_2, \dots, T_n$*
- *cost c_{ij} for the assignment of employee E_i to task T_j*

Goal:

Find an assignment of employees to tasks at minimal cost!

Mathematical formulation: Let $x_{ij} \in \{0,1\}$, $i,j = 1,\dots,n$, be defined as follows:

$$x_{ij} = \begin{cases} 0, & \text{if employee } i \text{ is not assigned to task } j, \\ 1, & \text{if employee } i \text{ is assigned to task } j \end{cases}$$

The assignment problem reads as follows:

$$\begin{aligned} \text{Minimise} \quad & \sum_{i,j=1}^{n} c_{ij} x_{ij} \\ \text{subject to} \quad & \sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1,\dots,n, \\ & \sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1,\dots,n, \\ & x_{ij} \in \{0,1\}, \qquad i,j = 1,\dots,n. \end{aligned}$$

As $x_{ij} \in \{0,1\}$, the constraints

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1,\dots,n,$$

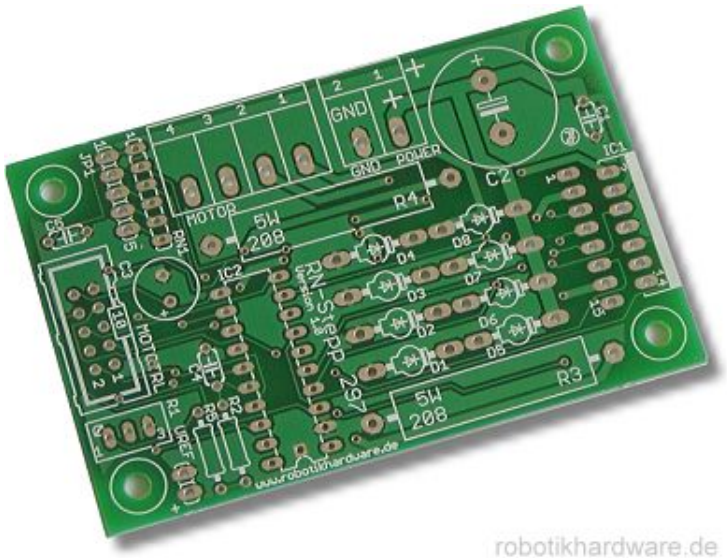
guarantee that each employee i does exactly one task. Likewise, the constraints

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, \ldots, n,$$

guarantee that each task j is done by one employee.
From a theoretical point of view, this problem seems to be rather simple as only finitely many possibilities for the choice of x_{ij}, $i, j = 1, \ldots, n$ exist. Hence, in a naive approach one could enumerate all feasible points and choose the best one. How many feasible points are there? Let $x_{ij} \in \{0, 1\}$, $i, j = 1, \ldots, n$, be the entries of an $n \times n$-matrix. The constraints of the assignment problem require that each row and column of this matrix has exactly one non-zero entry. Hence, there are n choices to place the 1 in the first row, $n - 1$ choices to place the 1 in the second row and so on. Thus there are $n! = n(n-1)\cdot(n-2)\cdots 2\cdot 1$ feasible points. The naive enumeration algorithm needs to evaluate the objective function for all feasible points. The following table shows how rapidly the number of evaluations grows:

n	10	20	30	50	70	90
eval	$3.629 \cdot 10^6$	$2.433 \cdot 10^{18}$	$2.653 \cdot 10^{32}$	$3.041 \cdot 10^{64}$	$1.198 \cdot 10^{100}$	$1.486 \cdot 10^{138}$

Example 2.2.2 (VLSI design, see Korte and Vygen [5])
A company has a machine which drills holes into printed circuit boards.



Since it produces many boards and as time is money, it wants the machine to complete one board as fast as possible. The drilling time itself cannot be influenced, but the time needed to move from one position to another can be minimised by finding an optimal route. The drilling machine can move in horizontal and vertical direction simultaneously.

Hence, the time needed to move from a position $p_i = (x_i, y_i)^\top \in \mathbb{R}^2$ *to another position* $p_j = (x_j, y_j)^\top \in \mathbb{R}^2$ *is proportional to the* l_∞*-distance*

$$\|p_i - p_j\|_\infty = \max\{|x_i - x_j|, |y_i - y_j|\}.$$

Given a set of points $p_1, \dots, p_n \in \mathbb{R}^2$*, the task is to find a permutation* $\pi : \{1, \dots, n\} \to \{1, \dots, n\}$ *of the points such that the total distance*

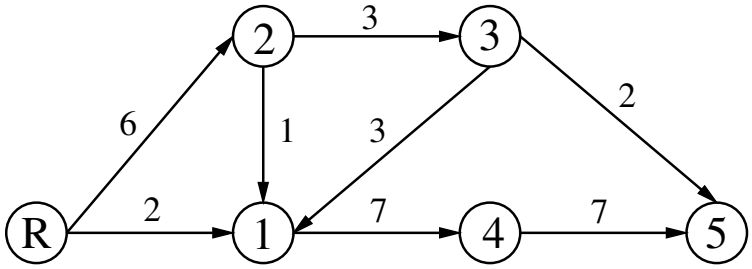
$$\sum_{i=1}^{n-1} \|p_{\pi(i)} - p_{\pi(i+1)}\|$$

becomes minimal.

Again, the most naive approach to solve this problem would be to enumerate all $n!$ *permutations of the set* $\{1, \dots, n\}$*, to compute the total distance for each permutation and to choose that permutation with the smallest total distance.*

Example 2.2.3 **(Network flow)**

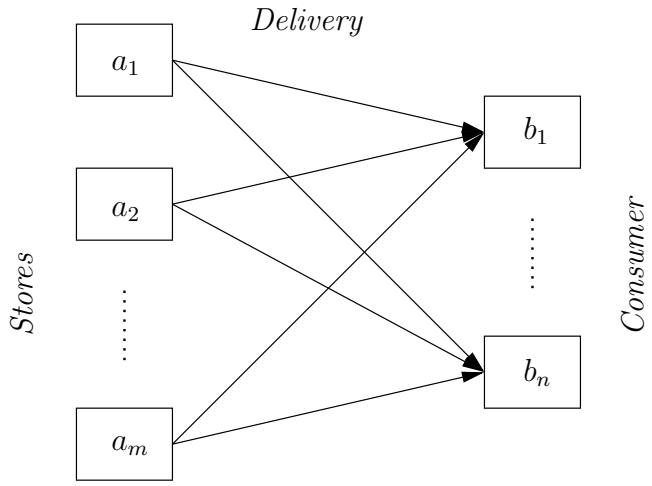
An oil company intends to transport as much oil as possible through a given system of pipelines from node R to node 5, see the graph below. Depending on the diameter of each pipeline the capacity is limited by the numbers (in million barrel per hour) next to the edges of the graph.



This problem is a so-called maximum flow problem or max-flow problem.

Example 2.2.4 **(Transportation problem)**

A transport company has m *stores and wants to deliver a product from these stores to* n *consumers. The delivery of one item of the product from store* i *to consumer* j *costs* c_{ij} *pound. Store* i *has stored* a_i *items of the product. Consumer* j *has a demand of* b_j *items of the product. Of course, the company wants to satisfy the demand of all consumers. On the other hand, the company aims at minimising the delivery costs.*



Let x_{ij} denote the amount of products which are delivered from store i to consumer j. In order to find the optimal transport plan, the company has to solve the following linear program:

$$
\begin{aligned}
&\textit{Minimise} && \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} && && \textit{(minimise delivery costs)}\\
&\textit{s.t.} && \sum_{j=1}^{n} x_{ij} && \leq\ a_i, \quad i=1,\dots,m, && \textit{(can't deliver more than it is there)}\\
& && \sum_{i=1}^{m} x_{ij} && \geq\ b_j, \quad j=1,\dots,n, && \textit{(satisfy demand)}\\
& && x_{ij} && \geq\ 0, \quad i=1,\dots,m,\ j=1,\dots,n. && \textit{(can't deliver negative amount)}
\end{aligned}
$$

In practical problems the variables x_{ij} often may only assume non-negative integer values, i.e. $x_{ij} \in \mathbb{N}_0$.

Example 2.2.5 (Facility location problem)

Let a set $I = \{1,\dots,m\}$ of clients and a set $J = \{1,\dots,n\}$ of potential depots be given. Opening a depot $j \in J$ causes fixed maintainance costs c_j.

Each client $i \in I$ has a total demand b_i and every depot $j \in J$ has a capacity u_j. The costs for the transport of one unit from depot $j \in J$ to customer $i \in I$ are denoted by h_{ij}.

The facility location problem aims at minimising the total costs (maintainance and transportation costs) and leads to a mixed-integer linear program:

Minimise

$$\sum_{j\in J} c_j x_j + \sum_{\substack{i\in I\\ j\in J}} h_{ij}y_{ij}$$

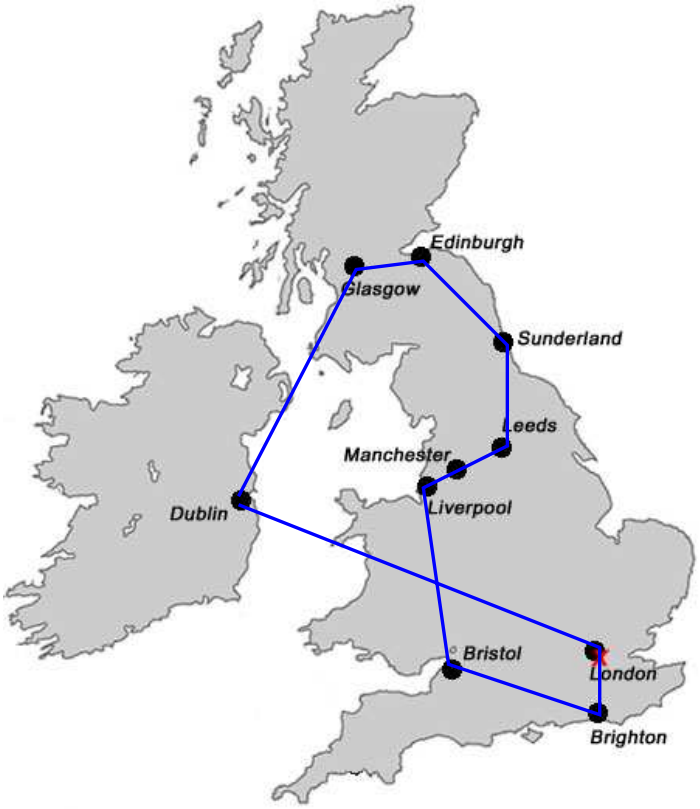
subject to the constraints

$$\begin{aligned} x_j \in \{0,1\}, &\qquad j \in J, \\ y_{ij} \geq 0, &\qquad i \in I,\ j \in J, \\ \sum_{j \in J} y_{ij} = b_i, &\qquad i \in I, \\ \sum_{i \in I} y_{ij} \leq x_j u_j, &\qquad j \in J. \end{aligned}$$

Herein, the variable $x_j \in \{0,1\}$ is used to decide whether depot $j \in J$ is opened ($x_j = 1$) or not ($x_j = 0$). The variable y_{ij} denotes the demand of client i satisfied from depot j.

Example 2.2.6 (Traveling salesman problem)

Let a number of cities $V = \{1, \ldots, n\}$ and a number of directed connections between the cities $E \subset V \times V$ be given.
Let c_{ij} denote the length (or time or costs) of the connection $(i,j) \in E$. A tour is a closed directed path which meets each city exactly once. The problem is to find a tour of minimal length.



Let the variables

$$x_{ij} \in \{0,1\}$$

be given with

$$x_{ij} = \begin{cases} 1, & \text{if } (i,j) \text{ is part of a tour,} \\ 0 & \text{otherwise} \end{cases}$$

for every $(i,j) \in E$.
These are finitely many vectors only, but usually very many!
The constraint that every city is met exactly once can be modeled as

$$\sum_{\{i:(i,j)\in E\}} x_{ij} = 1, \qquad j \in V, \tag{2.1}$$

$$\sum_{\{j:(i,j)\in E\}} x_{ij} = 1, \qquad i \in V. \tag{2.2}$$

These constraints don't exclude disconnected sub-tours (actually, these are the constraints of the assignment problem).
Hence, for every disjoint partition of V *in non-empty sets*

$$\begin{aligned} U &\subset V \\ U^c &\subset V \end{aligned}$$

we postulate: There exists a connection

$$(i,j) \in E \text{ with } i \in U,\ j \in U^c$$

and a connection

$$(k,\ell) \in E \text{ with } k \in U^c,\ \ell \in U\ .$$

The singletons don't have to be considered according to (2.1) and (2.2), respectively. We obtain the additional constraints

$$\sum_{\{(i,j)\in E: i\in U, j\in V\setminus U\}} x_{ij} \geq 1 \tag{2.3}$$

for every $U \subset V$ *with* $2 \leq |U| \leq |V| - 2$.
The Traveling Salesman Problem reads as:
Minimise

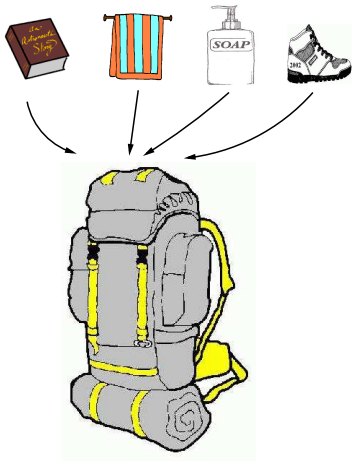
$$\sum_{(i,j)\in E} c_{ij} x_{ij}$$

subject to

$$x_{ij} \in \{0,1\}, \qquad (i,j) \in E,$$

and (2.1), (2.2), (2.3).

Example 2.2.7 (Knapsack Problems)



There is one knapsack and N items. Item j has weight a_j and value c_j for $j = 1, \ldots, N$. The task is to create a knapsack with maximal value under the restriction that the maximal weight is less than or equal to A. This leads to the optimization problem

$$\text{Maximise} \sum_{j=1}^{N} c_j x_j \qquad \text{subject to} \qquad \sum_{j=1}^{N} a_j x_j \leq A,\ x_j \in \{0,1\},\ j = 1, \ldots, N,$$

where

$$x_j = \begin{cases} 1, & \text{item } j \text{ is put into the knapsack,} \\ 0, & \text{item } j \text{ is not put into the knapsack.} \end{cases}$$

A naive approach is to investigate all 2^N possible combinations.

2.3 A Linear Programming Reminder

In this section we restrict the discussion to linear programs in standard form. This is not a restriction as any linear program can be transformed to standard form using well-known transformation techniques that can be found in every textbook on linear programming. In the sequel it is assumed that the reader is familiar with these techniques.

Definition 2.3.1 **(Standard Linear Program** (LP)**)**

Let

$$c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \in \mathbb{R}^n, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \in \mathbb{R}^m, \quad A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

with Rank$(A) = m$ be given. The standard linear program reads as follows: Find $x = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n$ such that the objective function

$$c^\top x = \sum_{i=1}^{n} c_i x_i$$

becomes minimal subject to the constraints

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \qquad i = 1, \dots, m, \tag{2.4}$$

$$x_j \geq 0, \qquad j = 1, \dots, n. \tag{2.5}$$

In matrix notation:

$$\textit{Minimise} \quad c^\top x \quad \textit{subject to} \quad Ax = b,\ x \geq 0. \tag{2.6}$$

We need some notation.

Definition 2.3.2 **(objective function, feasible set, feasible points, optimality)**

(i) The function $f(x) = c^\top x$ is called objective function.

(ii) The set

$$M := \{x \in \mathbb{R}^n \mid Ax = b,\ x \geq 0\}$$

is called feasible (or admissible) set (of LP*). M is a convex set.*

(iii) A vector $x \in M$ is called feasible (or admissible) (for LP*).*

(iv) $\hat{x} \in M$ is called optimal (for LP*), if*

$$c^\top \hat{x} \leq c^\top x \quad \forall x \in M.$$

The feasible set M is the intersection of the affine set $\{x \in \mathbb{R}^n \mid Ax = b\}$ and the non-negative orthant $\{x \in \mathbb{R}^n \mid x \geq 0\}$ and has vertices. Those vertices play an important role in linear programming. The following theorem states that it is sufficient to visit only the vertices of the feasible set in order to find one (not every!) optimal solution.

Theorem 2.3.3 **(Fundamental Theorem of Linear Programming)**

Let a standard LP 2.3.1 be given with $M \neq \emptyset$. Then:

(a) Either the objective function is unbounded from below on M or Problem 2.3.1 has an optimal solution and at least one vertex of M is among the optimal solutions.

(b) If M is bounded, then an optimal solution exists and $x \in M$ is optimal, if and only if x is a convex combination of optimal vertices.

The geometric definition of a vertex is as follows: $x \in M$ is a vertex of M if and only if x cannot be expressed as a strict convex combination of feasible points in M. This definition is not very useful for numerical computations. Fortunately, there is an equivalent characterisation of a vertex as a feasible basic solution.
Notation:

- The columns of A are denoted by

$$a^j := (a_{1j}, \dots, a_{mj})^\top \in \mathbb{R}^m, \qquad j = 1, \dots, n,$$

  i.e.

$$A = \left(\begin{array}{c|c|c|c} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{array} \right) = \begin{pmatrix} a^1 & a^2 & \cdots & a^n \end{pmatrix}.$$

- Let $B \subseteq \{1, \dots, n\}$ be an index set.
  - Let $x = (x_1, \dots, x_n)^\top$ be a vector. Then, x_B is defined to be the vector with components x_i, $i \in B$.
  - Let A be a $m \times n$-matrix with columns a^j, $j = 1, \dots, n$. Then, A_B is defined to be the matrix with columns a^j, $j \in B$.

Definition 2.3.4 **(feasible basic solution)**
Let $A \in \mathbb{R}^{m \times n}$, $\operatorname{rank}(A) = m$, $b \in \mathbb{R}^m$, and $M = \{x \in \mathbb{R}^n \mid Ax = b,\ x \geq 0\}$.
Let $B \subseteq \{1, \dots, n\}$ be an index set with $|B| = m$, $N = \{1, \dots, n\} \setminus B$, and let the columns a^i, $i \in B$, be linearly independent. Then, x is called a basic solution (of M) if

$$A_B x_B = b, \qquad x_N = 0.$$

x is called feasible basic solution (of M) if x is a basic solution and $x_B \geq 0$.

The following theorem states that a feasible basic solution is a vertex of the feasible set and vice versa.

Theorem 2.3.5
$x \in M$ is a vertex of M if and only if x is a feasible basic solution of M.

We need some more definitions.

Definition 2.3.6

- *(Basis)*
  Let rank$(A) = m$ and let x be a feasible basic solution of the standard LP. Every system $\{a^j \mid j \in B\}$ of m linearly independent columns of A, which includes those columns a^j with $x_j > 0$, is called basis of x.

- *((Non-)basis index set, (non-)basis matrix, (non-)basic variable)*
  Let $\{a^j \mid j \in B\}$ be a basis of x. The index set B is called basis index set, the index set $N := \{1, \dots, n\} \setminus B$ is called non-basis index set, the matrix $A_B := (a^j)_{j \in B}$ is called basis matrix, the matrix $A_N := (a^j)_{j \in N}$ is called non-basis matrix, the vector $x_B := (x_j)_{j \in B}$ is called basic variable and the vector $x_N := (x_j)_{j \in N}$ is called non-basic variable.

The fundamental theorem of linear programming suggests to compute the feasible basic solutions of the feasible set M. This is the basic idea of the simplex method.

Algorithm 2.3.7 (Revised Simplex Method)

(0) Phase 0:
Transform the linear program into standard form 2.3.1, if necessary at all.

(1) Phase 1:
Determine a feasible basic solution (vertex) x for the standard LP 2.3.1 with basis index set B, non-basis index set N, basis matrix A_B, basic variable $x_B \geq 0$ and non-basic variable $x_N = 0$.

If no feasible solution exists, STOP. The problem is infeasible.

(2) Phase 2:

(i) Compute $\beta = (\beta_i)_{i \in B}$ as the solution of the linear equation $A_B \beta = b$.
(ii) Solve the linear equation $A_B^\top \lambda = c_B$ for $\lambda \in \mathbb{R}^m$ and compute $\zeta = (\zeta_j)_{j \in N}$ by

$$\zeta^\top = \lambda^\top A_N - c_N^\top.$$

(ii) Check for optimality:
If $\zeta_j \leq 0$ for every $j \in N$, then STOP. The current feasible basic solution $x_B = \beta$, $x_N = 0$ is optimal. The objective function value is $d = c_B^\top \beta$.
(iii) Determine pivot column: Choose an index q with $\zeta_q > 0$ (according to Bland's rule).
(iv) Compute pivot column: Solve the linear equation $A_B \gamma = a^q$, where a^q denotes column q of A.
(v) Check for unboundedness:
If $\gamma \leq 0$, then the linear program does not have a solution and the objective function is unbounded from below. STOP.

(vi) *Determine pivot row:*
Choose an index p (according to Bland's rule) with

$$\frac{\beta_p}{\gamma_p} = \min\left\{ \frac{\beta_i}{\gamma_i} \;\middle|\; \gamma_i > 0,\ i \in B \right\}.$$

(vii) *Perform basis change:*
Set $B := (B \setminus \{p\}) \cup \{q\}$ and $N := (N \setminus \{q\}) \cup \{p\}$.

(viii) *Go to (i).*

The revised simplex algorithm requires to solve three linear equations only, namely two linear equations with A_B in steps (i) and (iv) and one linear equation with $A_B^\top$ in (ii):

$$A_B\beta = b, \qquad A_B\gamma = a^q, \qquad A_B^\top\lambda = c_B.$$

Bland's rule [3]
Among all possible choices for the pivot element, always choose the pivot column $q \in N$ and the pivot column $p \in B$ in steps (iii) and (vi) of Algorithm 2.3.7 with the smallest indices q and p.

Finally, we state the main result of this section:

Theorem 2.3.8 **(Finite termination)**
If the pivot element is chosen according to Bland's rule, the revised simplex algorithm 2.3.7 either finds an optimal solution or it detects that the objective function is unbounded from below. In both cases Algorithm 2.3.7 terminates after a finite number of steps.

2.3.1 Duality

Many algorithms for combinatorial optimisation problems exploit dual linear programs.

Definition 2.3.9 **(Primal problem (P))**
The standard LP

$$(P) \qquad \text{Minimise} \quad c^\top x \quad \text{subject to} \quad Ax = b,\ x \geq 0$$

with $A \in \mathbb{R}^{m\times n}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ is called primal problem.

Each primal problem can be associated another linear program – the dual problem.

Definition 2.3.10 **(Dual problem (D))**
The linear program

$$(D) \qquad \text{Maximise} \quad b^\top\lambda \quad \text{subject to} \quad A^\top\lambda \leq c$$

is called the dual problem of (P).

Example 2.3.11

Let the primal LP be given by the following linear program:
Minimise $-3x_1 - 4x_2$ *subject to*

$$\begin{aligned} 2x_1 + x_2 + x_3 &= 8, \\ 4x_1 + x_2 + x_4 &= 10, \\ x_1, x_2, x_3, x_4 &\geq 0. \end{aligned}$$

The dual problem reads as:
Maximise $8\lambda_1 + 10\lambda_2$ *subject to*

$$\begin{aligned} 2\lambda_1 + 4\lambda_2 &\leq -3, \\ \lambda_1 + \lambda_2 &\leq -4, \\ \lambda_1 &\leq 0, \\ \lambda_2 &\leq 0. \end{aligned}$$

By application of the well-known transformation techniques, which allow to transform a general LP into a standard LP, and writing down the dual problem of the resulting standard problem, it is possible to formulate the dual problem of a general LP. The following dualisation scheme applies. Notice in the scheme below, that the primal problem is supposed to be a minimisation problem.

primal constraints (minimise $c^\top x$)	dual constraints (maximise $b^\top \lambda$)
$x \geq 0$	$A^\top \lambda \leq c$
$x \leq 0$	$A^\top \lambda \geq c$
x free	$A^\top \lambda = c$
$Ax = b$	λ free
$Ax \leq b$	$\lambda \leq 0$
$Ax \geq b$	$\lambda \geq 0$

The same scheme applies component-wise if primal constraints of the different types

$$x_i \left\{ \begin{array}{c} \geq 0 \\ \leq 0 \\ \text{free} \end{array} \right\}, \ i = 1, \ldots, n, \qquad \sum_{j=1}^{n} a_{ij} x_j \left\{ \begin{array}{c} = \\ \leq \\ \geq \end{array} \right\} b_i, \ i = 1, \ldots, m,$$

occur simultaneously in a general LP. More precisely, the dual variable λ_i is associated

with the i-th primal constraint according to the scheme

$$\sum_{j=1}^{n} a_{ij}x_j \left\{ \begin{array}{c} = \\ \leq \\ \geq \end{array} \right\} b_i \qquad \leftrightarrow \qquad \lambda_i \left\{ \begin{array}{c} \text{free} \\ \leq 0 \\ \geq 0 \end{array} \right\}.$$

Moreover, the i-th primal variable x_i is associated with the i-th component of the dual constraint according to the scheme

$$x_i \left\{ \begin{array}{c} \geq 0 \\ \leq 0 \\ \text{free} \end{array} \right\} \qquad \leftrightarrow \qquad (A^\top \lambda)_i \left\{ \begin{array}{c} \leq \\ \geq \\ = \end{array} \right\} c_i.$$

In the latter, $(A^\top\lambda)_i$ denotes the i-th component of the vector $A^\top\lambda$, where $A = (a_{ij})$ denotes the matrix of coefficients of the primal constraints (excluding sign conditions of course). Again, the above relations assume that the primal problem is a minimisation problem.

It holds

Theorem 2.3.12

Dualisation of the dual problem yields the primal problem again.

Using this theorem, the above scheme can be read backwards, i.e. the primal and dual problems can be interchanged.

We summarise important relations between the primal problem and its dual problem.

Theorem 2.3.13 (Weak Duality Theorem)

Let x be feasible for the primal problem (P) (i.e. $Ax = b$, $x \geq 0$) and let λ be feasible for the dual problem (D) (i.e. $A^\top\lambda \leq c$). Then it holds

$$b^\top\lambda \leq c^\top x.$$

The weak duality theorem provides a motivation for the dual problem: dual feasible points provide lower bounds for the optimal objective function value of the primal problem. Vice versa, primal feasible points provide upper bounds for the optimal objective function value of the dual problem. This property is very important in the context of Branch & Bound methods for integer programs.

Moreover, it holds

Theorem 2.3.14 (Sufficient Optimality Criterion)

Let x be feasible for the primal problem (P) and let λ be feasible for the dual problem (D).

(i) If $b^\top\lambda = c^\top x$, then x is optimal for the primal problem (P) and λ is optimal for the dual problem (D).

(ii) $b^\top \lambda = c^\top x$ *holds if and only if the complementary slackness condition holds:*

$$x_j \left(\sum_{i=1}^{m} a_{ij}\lambda_i - c_j \right) = 0, \qquad j = 1, \dots, n.$$

Remark 2.3.15

The complementary slackness condition is equivalent with the following: For $j = 1, \dots, n$ *it holds*

$$x_j > 0 \qquad \Rightarrow \qquad \sum_{i=1}^{m} a_{ij}\lambda_i - c_j = 0$$

and

$$\sum_{i=1}^{m} a_{ij}\lambda_i - c_j < 0 \qquad \Rightarrow \qquad x_j = 0.$$

This means: Either the primal constraint $x_j \geq 0$ *is active (i.e.* $x_j = 0$*) or the dual constraint* $\sum_{i=1}^m a_{ij}\lambda_i \leq c_i$ *is active (i.e.* $\sum_{i=1}^m a_{ij}\lambda_i = c_i$*). It cannot happen that both constraints are inactive at the same time (i.e.* $x_j > 0$ *and* $\sum_{i=1}^m a_{ij}\lambda_i < c_i$*).*

The following theorem is the main result of this section.

Theorem 2.3.16 (Strong Duality Theorem)

The primal problem (P) has an optimal solution x *if and only if the dual problem (D) has an optimal solution* λ*. Moreover, the primal and dual objective function values coincide if an optimal solution exists, i.e.* $c^\top x = b^\top \lambda$*.*

Combining the strong duality theorem and the sufficient optimality criterion we obtain

Corollary 2.3.17

Let x *be feasible for the primal problem (P) and let* λ *be feasible for the dual problem (D). Then the following statements are equivalent.*

- x *is optimal for (P) and* λ *is optimal for (D).*
- *It holds* $c^\top x = b^\top \lambda$*.*
- *The complementary slackness condition holds.*

Remark 2.3.18

The primal (revised) simplex method, compare Algorithm 2.3.7, computes vertices x*, which are primally feasible, and dual variables* λ*, which satisfy* $c^\top x = b^\top \lambda$ *in each step. The primal simplex method stops as soon as* λ *becomes feasible for the dual problem, i.e. dual feasibility is the optimality criterion for the primal simplex method.*

2.4 A Primal-Dual Algorithm

We consider the primal linear program

$$(P) \qquad \text{Minimise} \qquad c^\top x \qquad \text{subject to} \qquad Ax = b,\ x \geq 0$$

and the corresponding dual problem

$$(D) \qquad \text{Maximise} \qquad b^\top \lambda \qquad \text{subject to} \qquad A^\top \lambda \leq c$$

with $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, and $\lambda \in \mathbb{R}^m$. Without loss of generality we assume $b \geq 0$. Moreover, let a^j, $j = 1, \ldots, n$, denote the columns of A.
Our aim is to construct an algorithm that computes a feasible x for (P) and a feasible λ for (D) such that the complementary slackness conditions

$$x_j \left((A^\top \lambda)_j - c_j \right) = 0, \qquad j = 1, \ldots, n,$$

hold, where $(A^\top \lambda)_j = (a^j)^\top \lambda$ denotes the j-th component of the vector $A^\top \lambda$. Theorem 2.3.14 guarantees then the optimality of x and λ.

Construction of the algorithm:

- Let a feasible $\lambda \in \mathbb{R}^m$ for (D) be given (if $c \geq 0$, we can choose $\lambda = 0$; otherwise we have to solve an auxiliary problem first).

  Define the index set

  $$B := \{j \in \{1, \ldots, n\} \mid (A^\top \lambda)_j = c_j\}$$

  of active dual constraints.

- The complementary slackness conditions will be satisfied if we can find a feasible x for (P) with

  $$x_j \underbrace{\left((A^\top \lambda)_j - c_j \right)}_{<0 \text{ for } j \notin B} = 0, \qquad \forall j \notin B,$$

  which is the case if and only if $x_j = 0$ holds for all $j \notin B$. Summarising, we have to find $x \in \mathbb{R}^n$ with

  $$\begin{aligned} \sum_{j\in B} a^j x_j = A_B x_B &= b, \\ x_j &\geq 0, \qquad \forall j \in B, \\ x_j &= 0, \qquad \forall j \notin B. \end{aligned}$$

  This can be achieved by solving an auxiliary linear program which was already useful to find an initial feasible basic solution for the simplex method (Phase 1).

This problem is obtained by introducing an artificial slack variable $y \in \mathbb{R}^m$ with non-negative components $y_i \geq 0$, $i = 1, \ldots, m$, and is referred to as Restricted Primal:

$$(RP) \qquad \begin{array}{lll} \text{Minimise} & \sum_{i=1}^{m} y_i & \\ \text{subject to} & A_B x_B + y = b, & \\ & x_j \geq 0, & j \in B, \\ & x_j = 0, & j \notin B, \\ & y_i \geq 0, & i = 1, \ldots, m. \end{array}$$

This linear program can be solved by, e.g., the simplex method. Notice that (RP) always has an optimal solution as the objective function is bounded below by 0 on the feasible set and $x = 0$ and $y = b \geq 0$ is feasible.

Let f_{opt} denote the optimal objective function value of (RP) and let (x_{opt}, y_{opt}) be an optimal solution of (RP).

- Two cases may occur:

  (i) If $f_{opt} = 0$, then the corresponding solution x_{opt} is feasible for (P) and satisfies the complementary slackness conditions. Thus, x_{opt} is optimal for (P) and λ is optimal for (D).

  (ii) If $f_{opt} > 0$, then the corresponding solution x_{opt} is not feasible for (P) as it only satisfies $Ax_{opt} + y_{opt} = b$ with a non-zero vector y_{opt}. How can we proceed?
  To answer this question, we will exploit the dual problem of (RP) which is given by

$$(DRP) \qquad \begin{array}{lll} \text{Maximise} & b^\top \lambda & \\ \text{subject to} & A_B^\top \lambda \leq 0, & \\ & \lambda_j \leq 1, & j = 1, \ldots, m. \end{array}$$

  Let $\bar{\lambda}$ denote an optimal solution of this problem. Owing to the strong duality theorem we have $b^\top \bar{\lambda} = f_{opt} > 0$.
  We intend to improve the dual objective function by computing a better dual vector

$$\lambda^* = \lambda + t\bar{\lambda}$$

  for some $t \in \mathbb{R}$. The dual objective function value then computes to

$$b^\top \lambda^* = b^\top \lambda + t \underbrace{b^\top \bar{\lambda}}_{>0}$$

  and it will increase for $t > 0$.

Moreover, we want to maintain dual feasibility of λ^*, i.e.

$$(A^\top \lambda^*)_j = \underbrace{(A^\top \lambda)_j}_{\leq c_j} + t(A^\top \bar{\lambda})_j \leq c_j, \qquad \forall j = 1, \ldots, n.$$

As $\bar{\lambda}$ satisfies $A_B^\top \bar{\lambda} \leq 0$ resp. $(a^j)^\top \bar{\lambda} = (A^\top \bar{\lambda})_j \leq 0$ for $j \in B$, the above inequality is satisfied for every $t \geq 0$ for $j \in B$.

It remains to check dual feasibility for $j \notin B$. Two cases:

(iia) If $(A^\top \bar{\lambda})_j \leq 0$ for all $j \notin B$, then dual feasibility is maintained for every $t \geq 0$. For $t \to \infty$ the dual objective function is unbounded because $b^\top \lambda^* \to \infty$ for $t \to \infty$. Hence, in this case the dual objective function is unbounded above and the dual does not have a solution. According to the strong duality theorem the primal problem does not have a solution as well.

(iib) If $(A^\top \bar{\lambda})_j > 0$ for some $j \notin B$, then dual feasibility is maintained for

$$t_{max} := \min \left\{ \frac{c_j - (A^\top \lambda)_j}{(A^\top \bar{\lambda})_j} \;\middle|\; j \notin B,\ (A^\top \bar{\lambda})_j > 0 \right\}.$$

Hence, $\lambda^* = \lambda + t_{max} \bar{\lambda}$ is feasible for (D) and improves the dual objective function value.

These considerations lead to the following algorithm.

Algorithm 2.4.1 (Primal-Dual Algorithm)

(0) Let λ be feasible for (D).

(1) Set $B = \{j \in \{1, \ldots, n\} \mid (A^\top \lambda)_j = c_j\}$.

(2) Solve (RP) (for the time being by the simplex method). Let f_{opt} denote the optimal objective function value and let $\bar{\lambda}$ be an optimal solution of (DRP).

(3) If $f_{opt} = 0$, then STOP: optimal solution found; x is optimal for (P) and λ is optimal for (D).

(4) If $f_{opt} > 0$ and $(A^\top \bar{\lambda})_j \leq 0$ for every $j \notin B$, then STOP: (D) is unbounded and (P) is infeasible.

(5) Compute

$$t_{max} := \min \left\{ \frac{c_j - (A^\top \lambda)_j}{(A^\top \bar{\lambda})_j} \;\middle|\; j \notin B,\ (A^\top \bar{\lambda})_j > 0 \right\}$$

set $\lambda := \lambda + t_{max} \bar{\lambda}$, and go to (1).

Example 2.4.2

Consider the standard LP

$$(P) \qquad \text{Minimise} \quad c^\top x \quad \text{subject to} \quad Ax = b,\ x \geq 0,$$

with

$$c = \begin{pmatrix} -2 \\ -3 \\ 0 \\ 0 \end{pmatrix}, \quad b = \begin{pmatrix} 8 \\ 9 \end{pmatrix}, \quad A = \begin{pmatrix} 1 & 2 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix}.$$

The dual is given by

$$(D) \qquad \text{Maximise} \quad b^\top \lambda \quad \text{subject to} \quad A^\top \lambda \leq c,$$

i.e. by

$$\begin{aligned} \text{Maximise} \quad & 8\lambda_1 + 9\lambda_2 \\ \text{subject to} \quad & \lambda_1 + 2\lambda_2 \leq -2, \\ & 2\lambda_1 + \lambda_1 \leq -3, \\ & \lambda_1 \leq 0, \\ & \lambda_2 \leq 0. \end{aligned}$$

Iteration 1:

$\lambda = (-1,-1)^\top$ *is feasible for (D) because* $A^\top\lambda = (-3,-3,-1,-1)^\top \leq c$ *and only the second constraint is active, i.e.* $B = \{2\}$. *The restricted primal reads as follows:*

$$\begin{aligned} \text{Minimise} \quad & y_1 + y_2 \\ \text{subject to} \quad & 2x_2 + y_1 = 8, \\ & x_2 + y_2 = 9, \qquad x_2, y_1, y_2 \geq 0. \end{aligned}$$

This LP has the solution $x_2 = 4, y_1 = 0, y_2 = 5$ *and* $f_{opt} = 5 > 0$. *A solution of the dual of the restricted primal is* $\bar\lambda = (-1/2, 1)^\top$. *We find* $A^\top\bar\lambda = (3/2, 0, -1/2, 1)^\top$ *and thus the unboundedness criterion is not satisfied. We proceed with computing*

$$t_{max} = \min\left\{\frac{-2-(-3)}{3/2}, \frac{0-(-1)}{1}\right\} = \frac{2}{3}$$

and update $\lambda = \lambda + t_{max}\bar\lambda = (-4/3, -1/3)^\top$.

Iteration 2:

$A^\top\lambda = (-2,-3,-4/3,-1/3)^\top$ *and thus* $B = \{1,2\}$. *The restricted primal is*

$$\begin{aligned} \text{Minimise} \quad & y_1 + y_2 \\ \text{subject to} \quad & x_1 + 2x_2 + y_1 = 8, \\ & 2x_1 + x_2 + y_2 = 9, \qquad x_1, x_2, y_1, y_2 \geq 0. \end{aligned}$$

This LP has the solution $x_1 = 10/3, x_2 = 7/3, y_1 = y_2 = 0$ *and* $f_{opt} = 0$. *Hence,* $x = (10/3, 7/3, 0, 0)^\top$ *is optimal for (P) and* $\lambda = (-4/3, -1/3)^\top$ *is optimal for (D).*

Chapter 3

Minimum Spanning Trees

Many combinatorial optimisation problems like routing problems, network flow problems, traveling salesman problems, and transportation problems use graphs and networks. For instance, a roadmap is a graph which defines connections between cities.
This chapter will be used to

- introduce basic notations and terminologies from graph theory
- discuss minimum spanning tree problems
- introduce a greedy strategy for minimum spanning tree problems

As a motivation for this chapter, consider the following practically relevant problem.
A telephone company wants to rent a subset of an existing telephone network. The subset of the network should be large enough to connect all cities, but at the same time renting it should be as cheap as possible. Figure 3.1 illustrates the situation with 6 cities.

How to model this problem? The telephone network is naturally modeled by a graph or, more precisely, by a network. Intuitively, we think of a graph as a set of nodes and edges, which describe connections between nodes in terms of undirected lines or directed lines (arrows), compare Figure 3.1.

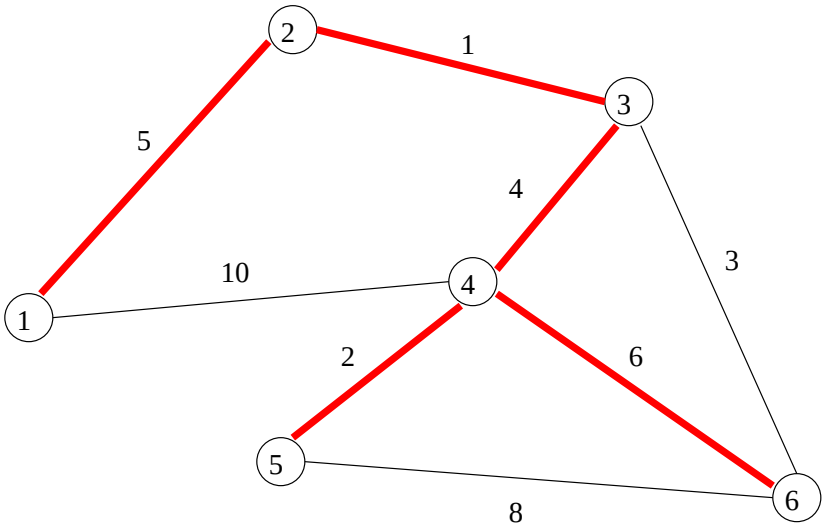


Figure 3.1: Spanning tree in a (undirected) graph.

In our example the nodes of the graph correspond to the cities and the edges correspond to the telephone connections between cities. The weight of each edge corresponds to the cost of renting the connection. We are looking for a so-called spanning tree of minimum weight which will be defined below.
Finding a spanning tree of minimum weight in a network belongs to the oldest problems in combinatorial optimisation and a first algorithm was given by Boruvka in 1926.

3.1 Elements from Graph Theory

Many combinatorial optimisation problems like the spanning tree problem, routing problems, network flow problems, traveling salesman problems, and transportation problems use graphs and networks. For instance, a roadmap or an electricity network is a graph which defines connections between cities. Basic notations and terminologies are summarised in the sequel.

Definition 3.1.1 **(Graph)**
A graph G is a tupel $G = (V, E)$, where $V \neq \emptyset$ is a finite set of nodes (or vertices) and $E \subseteq V \times V$ are the edges of G.
Let $e = (v, w) \in E$ be an edge with $v, w \in V$. We say:

- *$e = (v, w)$ joins the nodes v and w and v and w are adjacent.*
- *v is a neighbour of w and vice versa. The set of all neighbours of a node v is denoted by $N(v)$.*
- *v and w are the endpoints of e.*
- *v and w are incident with e.*

Sometimes, the edges of a graph are just used to indicate that some kind of relation between two nodes exists while the orientation of the edge is not important. But often, the edges of a graph describe flow directions or one-way street connections. In this case, the orientation of an edge is actually important as it is only possible to reach some node from another node following the connecting edge in the right direction. Hence, it is useful and necessary to distinguish directed and undirected graphs.

Definition 3.1.2 **(directed graph, undirected graph, underlying undirected graph)**

- *A graph (V, E) is called undirected, if all edges $e \in E$ are undirected, i.e. all tupels $(v, w) \in E$ are not ordered.*
- *A graph (V, E) is called digraph, if all edges $e \in E$ are directed, i.e. all tupels $(v, w) \in E$ are ordered.*

- *The underlying undirected graph of a digraph (V, E) is the undirected graph $(V, \tilde{E})$ with $(v, w) \in E \Leftrightarrow (v, w) \in \tilde{E}$, i.e. the underlying undirected graph has the same nodes and edges as the directed graph but the orientation of the edges are omitted.*

We introduce some terminologies for digraphs only.

Definition 3.1.3

Let $e = (v, w) \in E$ with $v, w \in V$ be an edge of a digraph $G = (V, E)$. We say:

- *e leaves the node v and enters the node w.*
- *v is the tail of e and w is the head of e.*
- *v is the predecessor of w. The set of all predecessors of a node $w \in V$ is denoted by $P(w)$ and $\delta^-(w) := |P(w)|$ denotes the in-degree of w.*
- *w is the successor of v. The set of all successors of a node $v \in V$ is denoted by $S(v)$ and $\delta^+(v) := |S(v)|$ denotes the out-degree of v.*

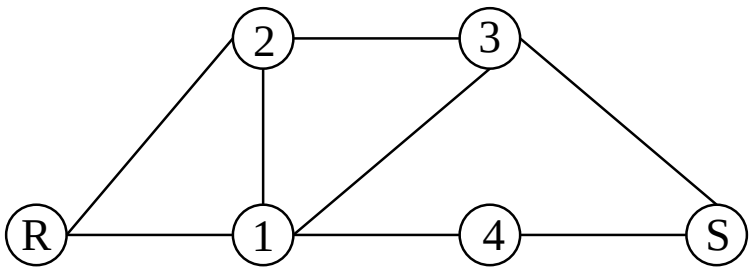
In a digraph a node without predecessor is called source and a node without successor is called sink.

In the sequel we always assume that the number of nodes in a graph is finite. Moreover, we exclude parallel edges, i.e. edges having the same endpoints (undirected graphs) or the same tail and head (directed graphs), respectively. Finally, we exclude sweeps, i.e. edges of type (i, i) with $i \in V$.

Usually, nodes are visualised by circles. Edges in an undirected graph are visualised by lines connecting two nodes. Edges in a digraph are visualised by arrows and indicate, e.g., flow directions or one-way street connections.

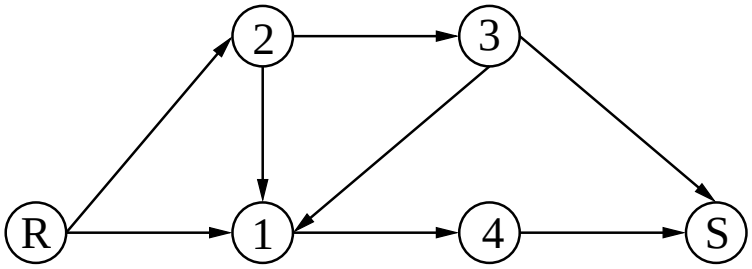
Example 3.1.4

Undirected graph $G = (V, E)$ with nodes $V = \{R, 1, 2, 3, 4, S\}$ and edges $E = \{(R, 1), (R, 2), (1, 2), (1, 3), (1, 4), (2, 3), (3, S), (4, S)\}$:



The neighbours of the node 1 are $N(1) = \{R, 2, 3, 4\}$.

Directed graph $G = (V, E)$ *with nodes* $V = \{R, 1, 2, 3, 4, S\}$ *and edges* $E = \{(R,1), (R,2), (2,1), (1,4), (2,3), (3,1), (3,S), (4,S)\}$*:*



The successors of node 2 *are* $S(2) = \{1, 3\}$. *The predecessors of node* 1 *are* $P(1) = \{R, 2, 3\}$. *Node* R *is the only source and* S *is the only sink of the digraph.*

Caution: Let i and j be nodes and let $e = (i, j)$ be an edge. In an undirected graph the edges (i, j) and (j, i) are considered to be the same. Whereas in a digraph the ordering of the nodes i and j in the edge (i, j) is highly relevant, i.e. (i, j) and (j, i) denote different edges and only one of which may be present in the directed graph.

A graph $G = (V, E)$ is called complete, if for any choice of distinct nodes $v, w \in V$ it holds $(v, w) \in E$. Notice that a complete digraph includes both, the edges (v, w) and (w, v). It is easy to see that a complete graph with n nodes has $n(n-1)/2$ edges and a digraph with n nodes has $n(n-1)$ edges.
A digraph $G = (V, E)$ is called symmetric, if $(v, w) \in E$ implies $(w, v) \in E$. It is called antisymmetric if $(v, w) \in E$ implies $(w, v) \notin E$.
Often it is important to find a way through a given graph starting at some node and ending at another node.

Definition 3.1.5 (Walks and Paths)
Let $G = (V, E)$ *be a graph.*

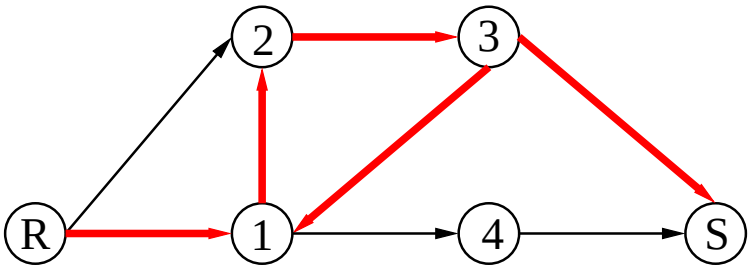
- *Each sequence*
$$W = v_1, e_1, v_2, e_2, \ldots, v_k, e_k, v_{k+1}, \qquad k \geq 0, \tag{3.1}$$
of nodes $v_1, \ldots, v_{k+1} \in V$ *and edges* $e_1, \ldots, e_k \in E$ *with* $e_i = (v_i, v_{i+1})$ *for* $i = 1, \ldots, k$ *is called edge progression (in* G*) with initial node* v_1 *and final node* v_{k+1}.
We use the shorthand notation $W = [v_1, v_2, \ldots, v_k, v_{k+1}]$.

- *An edge progression (3.1) is called walk (in* G*), if* $e_i \neq e_j$ *for* $1 \leq i < j \leq k$. *A walk is closed if* $v_1 = v_{k+1}$.

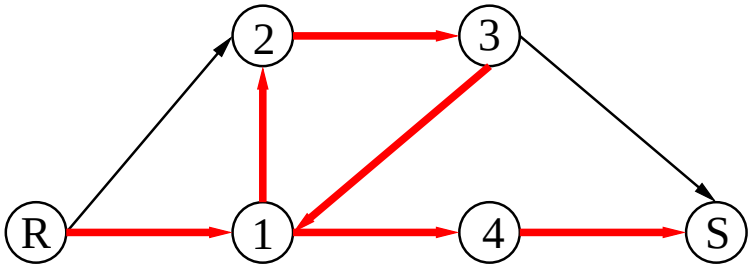
- *A walk (3.1) is called* $v_1 - v_{k+1}$*−path (in* G*), if* $v_i \neq v_j$ *for* $1 \leq i < j \leq k+1$. *The length of a path is the number of its edges.*

- *A circuit is a closed walk (3.1) with $v_i \neq v_j$ for $1 \leq i < j \leq k$.*
- *A node w is said to be reachable from a node v, if a $v-w-$path exists.*

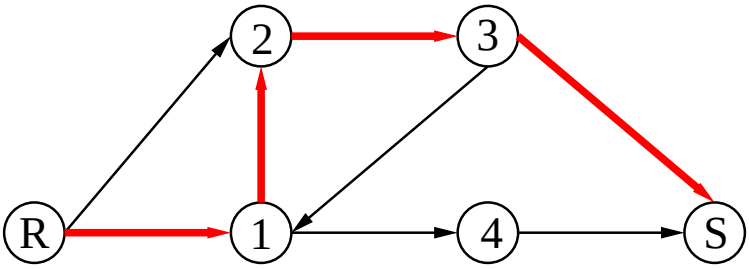
Example 3.1.6



Edge progression W=R,(R,1),1,(1,2),2,(2,3),3,(3,1),1,(1,2),2,(2,3),3,(3,S),S
not a walk as the edges (1,2) and (2,3) are visited twice!



Walk W=R,(R,1),1,(1,2),2,(2,3),3,(3,1),1,(1,4),4,(4,S),S
not a path as vertex 1 is visited twice!
The walk contains a circuit: 1,(1,2),2,(2,3),3,(3,1),1



Path W=R,(R,1),1,(1,2),2,(2,3),3,(3,S),S
S is reachable from every other vertex,
vertices R,1,2,3 are not reachable from vertex 4

Often, we are interested in connected graphs.

Definition 3.1.7 (connected graphs)

An undirected graph G is called connected, if there is a $v-w-$path for every $v, w \in V$.

A digraph G is called connected, if its underlying undirected graph is connected.

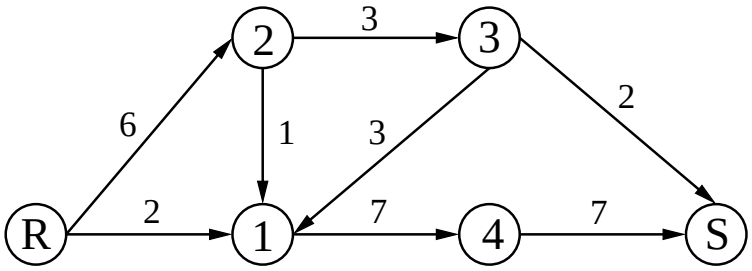
Often, edges in graphs are weighted, for instance moving along an edge causes costs or an edge corresponds to the length of a street or an edge has a certain capacity.

Definition 3.1.8 **(Weighted Graph, Network)**

A weighted graph or network N is a tripel $N = (V, E, c)$, where (V, E) is a graph and $c : E \to \mathbb{R}$ is a weight function with $e = (i, j) \in E \mapsto c(e) = c_{ij}$. The network is called directed (undirected), if the underlying graph (V, E) is directed (undirected).

Example 3.1.9

Network $N = (V, E, c)$ with nodes $V = \{R, 1, 2, 3, 4, S\}$, edges $E = \{(R, 1), (R, 2), (2, 1), (1, 4), (2, 3), (3, 1), (3, S), (4, S)\}$, and mapping $c : E \to \mathbb{R}$ with $c(R, 1) = c_{R1} = 2$, $c(R, 2) = c_{R2} = 6$, $c(2, 1) = c_{21} = 1$, $c(1, 4) = c_{14} = 7$, $c(2, 3) = c_{23} = 3$, $c(3, 1) = c_{31} = 3$, $c(3, S) = c_{3S} = 2$, $c(4, S) = c_{4S} = 7$.



3.2 Minimum Spanning Tree Problem

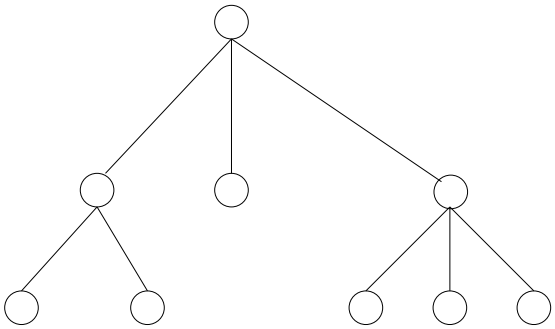
Before we formulate the problem in a formal way we need to clarify what a tree and a spanning tree are.

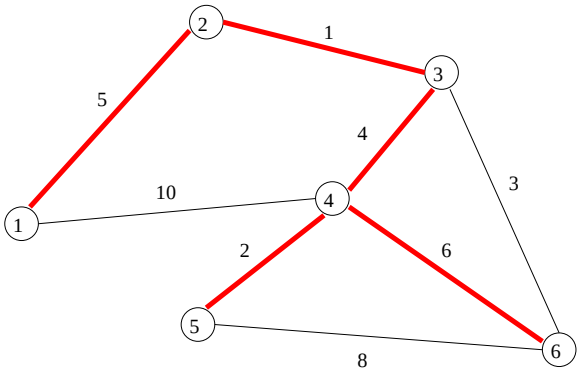
Definition 3.2.1 **(Tree, spanning tree)**

- *$G' = (V', E')$ is a subgraph of the graph $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G' = (V', E')$ of G is called spanning, if $V' = V$.*
- *An undirected graph which is connected and does not have a circuit (as a subgraph) is called a tree.*
- *A spanning tree of an undirected graph is a spanning subgraph which is a tree.*

Example 3.2.2

A tree and a spanning tree (red) in a graph:





Theorem 3.2.3

Let G be an undirected graph with n nodes. Then the following statements are equivalent:

(a) G is a tree.

(b) G has $n-1$ edges and no circuits.

(c) G has $n-1$ edges and is connected.

(d) G is a maximal circuit-free graph, i.e. any additional edge creates a circuit.

(e) For any two distinct nodes v, w there exists a unique $v-w-$path.

With these definitions we can state the

Minimum Spanning Tree Problem:
Let an undirected graph $G = (V, E)$ with nodes $V = \{v_1, \ldots, v_n\}$ and edges $E = \{e_1, \ldots, e_m\}$, and a weight function $c : E \to \mathbb{R}$ be given. The task is either to find a spanning tree $T = (V, E')$ in G with $E' \subseteq E$ such that $\sum_{e \in E'} c(e)$ becomes minimal or to decide that G is not connected.

Note that a graph has at least one spanning tree, if and only if it is connected.

3.3 Kruskal's Algorithm: A Greedy Algorithm

How can we solve the problem? We know already that the result will be a spanning tree (provided such a tree exists) and thus by definition the nodes of the spanning tree are the nodes of G. So, all we have to do is to find the subset of edges $E' \subset E$ which defines the spanning tree and minimises the costs

$$\sum_{e \in E'} c(e).$$

How can the set of edges E' be constructed? A straightforward idea is to construct E' step by step by adding successively edges from E to E'. Let's do this in a greedy fashion: as a spanning tree with minimum weight is sought, in each step we prefer edges $(v_i, v_j) \in E$ with small costs $c(v_i, v_j) = c_{ij}$. Of course, we have to check whether adding an edge creates a circuit. This outline of an algorithm is essentially Kruskal's Algorithm.

Algorithm 3.3.1 **(Kruskal's Algorithm)**

(0) Let an undirected Graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and $|E| = m$ and a weight function $c : E \to \mathbb{R}$ be given.

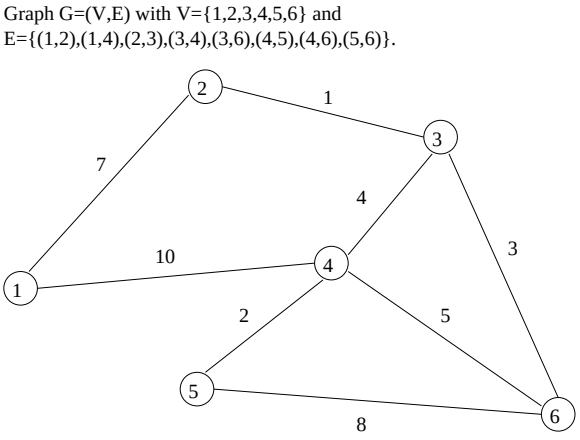
(1) Sort the edges in E such that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$, where $e_i \in E$, $i = 1, \ldots, m$, denote the edges after sorting.

(2) Set $E' = \emptyset$ and $T = (V, E')$.

(3) **For** $i = 1$ **to** m **do**
If $(V, E' \cup \{e_i\})$ *contains no circuit* **then** *set* $E' := E' \cup \{e_i\}$ *and* $T := (V, E')$.

Example 3.3.2

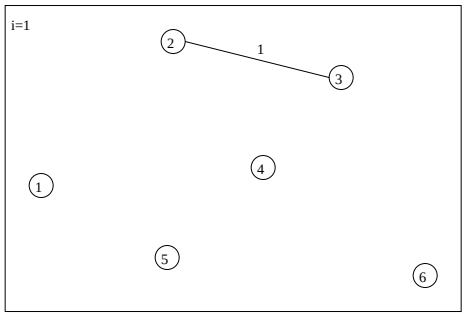
Consider the following example:



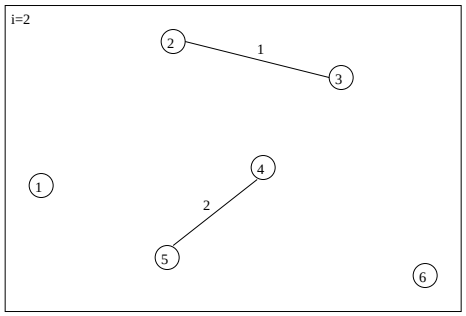
Sorting the edges in step (1) leads to the ordering

$$e_1 = (2,3), e_2 = (4,5), e_3 = (3,6), e_4 = (3,4), e_5 = (4,6), e_6 = (1,2), e_7 = (5,6), e_8 = (1,4).$$

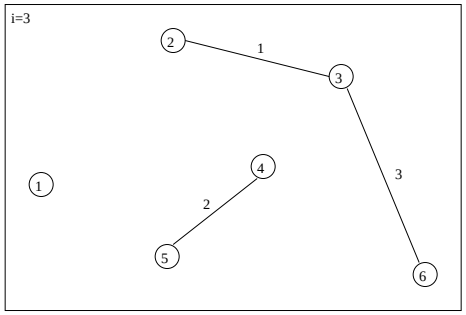
Kruskal's algorithm produces the following subgraphs:



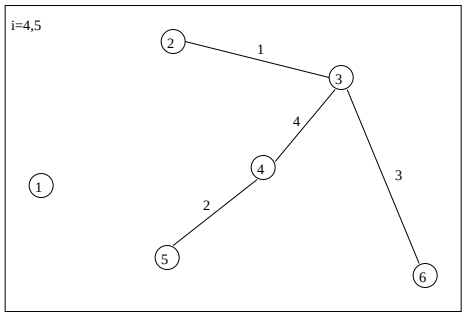
Add edge $e_1 = (2,3)$ *with weight* $c(e_1) = 1$.



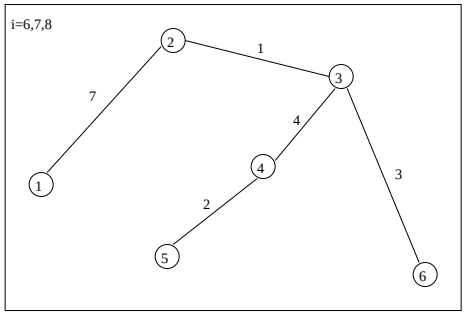
Add edge $e_2 = (4,5)$ *with weight* $c(e_2) = 2$.



Add edge $e_3 = (3,6)$ *with weight* $c(e_3) = 3$.



Add edge $e_4 = (3,4)$ *with weight* $c(e_4) = 4$. *Adding edge* $e_5 = (4,6)$ *with weight* $c(e_5) = 5$ *would lead to a cycle, so* e_5 *is not added.*



Add edge $e_6 = (1,2)$ *with weight* $c(e_6) = 7$. *Adding edges* $e_7 = (5,6)$ *with weight* $c(e_7) = 8$ *and* $e_8 = (1,4)$ *with weight* $c(e_8) = 10$ *would lead to cycles, so* e_7 *and* e_8 *are not added.*

Kruskal's algorithm is a so-called greedy algorithm. Greedy algorithms exist for many other applications. All greedy algorithms have in common that they use locally optimal

decisions in each step of the algorithm in order to construct a hopefully globally optimal solution. With other words: Greedy algorithms choose the best possible current decision in each step of an algorithm. In Kruskal's algorithm in each step the edge with the smallest cost was used. In general, it is not true that this strategy leads to an optimal solution. But for the minimum spanning tree problem it turns out to work successfully.

Theorem 3.3.3

Kruskal's algorithm constructs a spanning tree with minimal costs for the graph G, provided G has a spanning tree.

Proof: a proof can be found in Korte and Vygen [5], Theorem 6.3, p. 129. □

The complexity of Kruskal's algorithm depends on the complexity of the sorting algorithm used in step (1) and the complexity of the circuit checking algorithm in step (3) of Kruskal's algorithm.

A well-known sorting algorithm is the Merge-Sort algorithm. It divides the list $a_1, a_2, \ldots, a_n$ of n real numbers to be sorted into two sublists of approximately the same size. These sublists are sorted recursively by the same algorithm. Finally, the sorted sublists are merged together. This strategy is known as a divide-and-conquer strategy.

The MERGE-SORT algorithm below computes a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}.$$

It can be shown that the Merge-Sort algorithm needs $\mathcal{O}(n \log n)$ operations, compare Korte and Vygen [5], Theorem 1.5, page 10.

Algorithm 3.3.4 (MERGE-SORT)

(0) Let a list $a_1, \ldots, a_n$ of real numbers be given.

(1) If $n = 1$ then set $\pi(1) = 1$ and STOP.

(2) Set $m = \lfloor \frac{n}{2} \rfloor$. Let $\rho =$ MERGE-SORT$(a_1, \ldots, a_m)$ and $\sigma =$ MERGE-SORT$(a_{m+1}, \ldots, a_n)$.

(3) Set $k = 1$ and $\ell = 1$.

While $k \leq m$ *and* $\ell \leq n - m$ **do**
  If $a_{\rho(k)} \leq a_{m+\sigma(\ell)}$ **then**
    set $\pi(k + \ell - 1) = \rho(k)$ *and* $k = k + 1$.
  else
    set $\pi(k + \ell - 1) = m + \sigma(\ell)$ *and* $\ell = \ell + 1$.
  endif
end do
While $k \leq m$ **do**

set $\pi(k+\ell-1) = \rho(k)$ *and* $k = k+1$.
end do
While $\ell \leq n-m$ **do**
set $\pi(k+\ell-1) = m + \sigma(\ell)$ *and* $\ell = \ell+1$.
end do

Detecting a cycle in a graph can be done by the following labeling algorithm. Herein, every node initially gets assigned the label 'not explored'. Using a recursive depth first search, those nodes which are adjacent to the current node and are labeled 'not explored', are labeled 'active'. This is done until no adjacent node exists anymore or in the function CircDetectRec a node labeled 'active' is adjacent to the current 'active' node. In the latter case a circuit has been detected. The second argument u in CircDetectRec denotes the predecessor of v w.r.t. the calling sequence of CircDetectRec.

Algorithm 3.3.5 (CircDetect)

(0) Let a graph $G = (V, E)$ *with* n *nodes be given.*

(1) Label every node $v \in V$ *to be 'not explored'.*

(2) **For each** $v \in V$ **do**
If *v is 'not explored'* **then**
If CircDetectRec*(v,v)=false* **then** *STOP: Graph contains cycle.*
end do

*Function: result=*CircDetectRec*(v,u):*

(0) Label node v as 'active'.

(1) **For each** $w \in V$ *adjacent to* v *and* $w \neq u$ **do**
If *w is 'active'* **then**
result=false
return
else if *w is 'not explored'* **then**
If CircDetectRec*(w,v)=false* **then**
result=false
return
end if
end if
end do
Label node v to be 'explored', set result=true, and return.

The complexity of the CircDetect algorithm for a graph with n nodes and m edges is $\mathcal{O}(n+m)$. As every graph $(V, E' \cup \{e_i\})$ in step (3) of Kruskal's algorithms has at most n edges (finally it will be a spanning tree which has $n-1$ edges), the complexity of the circuit detection algorithm is given by $\mathcal{O}(n)$. Step (3) is executed m times and together with the complexity $\mathcal{O}(m \log m)$ of the Merge-Sort algorithm we obtain

Corollary 3.3.6 (Complexity of Kruskal's algorithm)
Kruskal's algorithm can be implemented in $\mathcal{O}(m\log(m) + mn)$ operations and Kruskal's algorithm has polynomial complexity. The minimum spanning tree problem belongs to the class $\mathcal{P}$ of polynomially time solvable problems.

Remark 3.3.7
There are even more efficient implementations of Kruskal's algorithm which use better data structures and need $\mathcal{O}(m \log n)$ operations only. Details can be found in Korte and Vygen [5], page 130, and in Papadimitriou and Steiglitz [7], section 12.2, page 274.

3.4 Prim's Algorithm: Another Greedy Algorithm

Kruskal's algorithm added successively edges to a graph with node set V. Prim's algorithm starts with a single node and constructs successively edges and adjacent nodes. This leads to a growing tree. Again, the greedy idea is employed as the edge with smallest cost is chosen in each step.

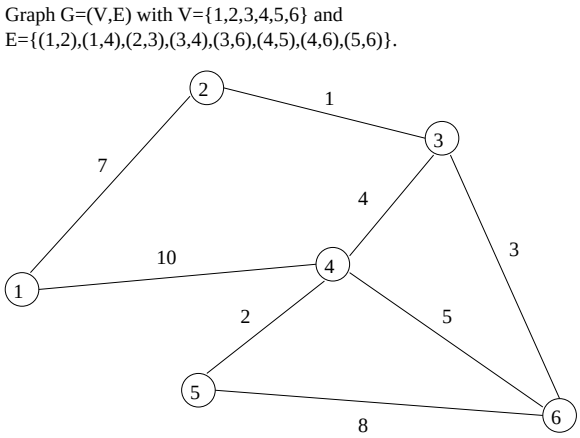
Algorithm 3.4.1 (Prim's Algorithm)

(0) Let an undirected Graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ and a weight function $c : E \to \mathbb{R}$ be given.

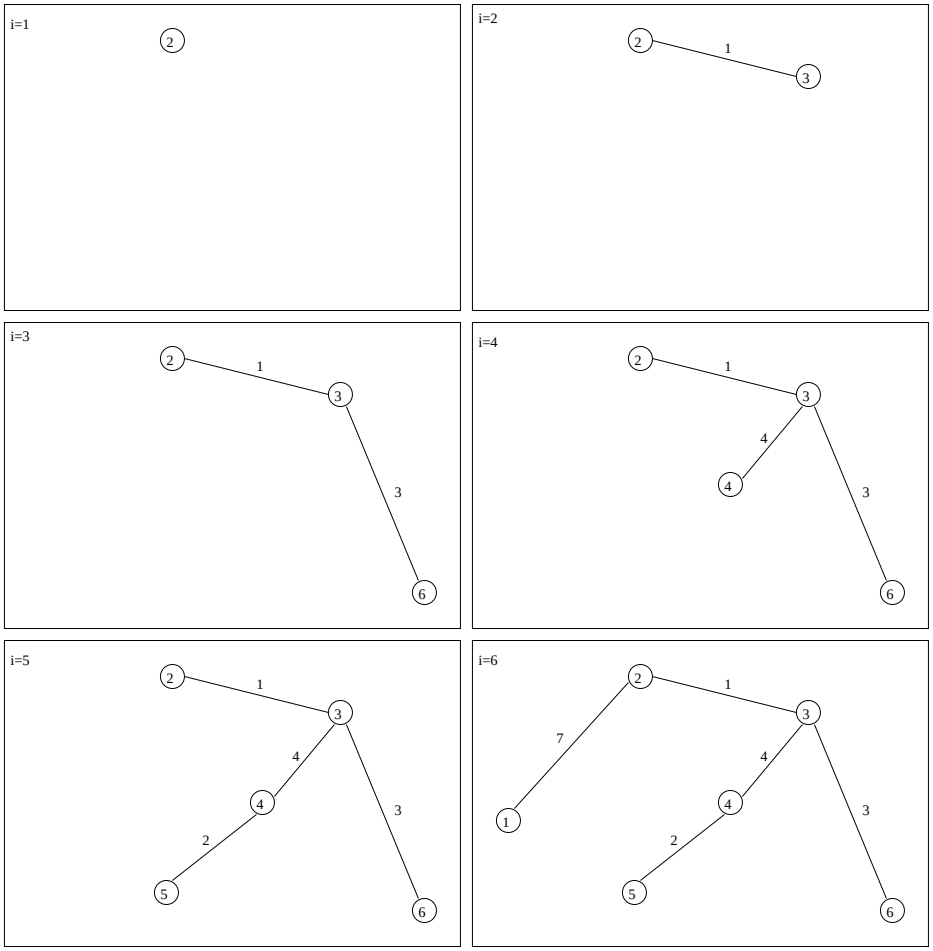
(1) Choose $v \in V$. Set $V' := \{v\}$, $E' := \emptyset$, and $T := (V', E')$.

(2) **While** $V' \neq V$ **do**
Choose an edge $e = (v, w)$ with $v \in V'$ and $w \in V \setminus V'$ of minimum weight $c(e)$. Set $V' := V' \cup \{w\}$, $E' := E' \cup \{e\}$, and $T := (V', E')$.
end do

Example 3.4.2
Consider the following example:



Prim's algorithm produces the following subgraphs, starting with node 2:



The correctness of the algorithm is established in

Theorem 3.4.3

Prim's algorithm constructs a spanning tree with minimal costs for the graph G, provided G has a spanning tree. Prim's algorithm can be implemented in $\mathcal{O}(n^2)$ operations

(polynomial complexity).

Proof: a proof can be found in Korte and Vygen [5], Theorem 6.5, p. 131. □

Remark 3.4.4

There are even more efficient implementations of Prim's algorithm which use better data structures (Fibonacci heap) and need $\mathcal{O}(m+n\log n)$ operations only. Details can be found in Korte and Vygen [5], pages 131-133.

Chapter 4

Shortest Path Problems

Many combinatorial optimisation problems can be formulated as linear optimisation problems on graphs or networks. Among them are

- transportation problems
- maximum flow problems
- assignment problems
- shortest path problems

In this chapter we will discuss shortest path problems and the algorithms of Dijkstra and Floyd-Warshall. We will see that these algorithms have polynomial complexity.
Typical applications for shortest path problems arise for instance in satellite navigation systems for cars and routing tasks in computer networks.
Let a directed network $N = (V, E, c)$ with set of nodes $V = \{Q = 1, 2, \ldots, n = S\}$ and set of edges E, $|E| = m$, be given with:

- Let Q be a source (i.e. Q has no predecessor and no edge enters Q) and let S be a sink (i.e. S has no successor and no edge leaves S).
- Let $c : E \to \mathbb{R}$ with $(i, j) \in E \mapsto c(i, j) = c_{ij}$ denote the length of the edge (i, j).
- N is connected.

The task is to find a shortest path from Q to S that is a path of minimal length. Herein, the length of a path $P = [v_1, \ldots, v_k, v_{k+1}]$ in the network N with $v_i \in V$, $i = 1, \ldots, k+1$, is defined by

$$\sum_{i=1}^{k} c(v_i, v_{i+1}) = \sum_{i=1}^{k} c_{v_i v_{i+1}}.$$

Shortest path problem:
Find a shortest $Q - S-$path in the network N or decide that S is not reachable from Q.

4.1 Linear Programming Formulation

The shortest path problem can be written as a linear program. The idea is to transport one unit from Q to S through the network. You may think of a car that is driving on the road network. The task is then to find a shortest $Q-S-$path for this car.

Let's formalise this idea and let $x : E \to \mathbb{R}$, $(i,j) \in E \mapsto x(i,j) = x_{ij}$, denote the amount which moves along the edge (i,j). Of course we want x_{ij} to be either one or zero indicating whether an edge (i,j) is used or not. But for the moment let's neglect this additional restriction. We will see later that our linear program will always have a zero-one solution. We supply one unit at Q and demand one unit at S. Moreover, nothing shall be lost at intermediate nodes $i = 2, 3, \ldots, n-1$, i.e. the so-called conservation equations

$$\underbrace{\sum_{k:(j,k)\in E} x_{jk}}_{\text{outflow out of node } j} - \underbrace{\sum_{i:(i,j)\in E} x_{ij}}_{\text{inflow into node } j} = 0, \quad j \in V\backslash\{Q,S\}.$$

have to hold at every interior node $j = 2, \ldots, n-1$. As we supply one unit at source Q and demand one unit at sink S, the following conservation equations have to hold:

$$\begin{aligned} \underbrace{\sum_{k:(Q,k)\in E} x_{Qk}}_{\text{outflow out of node } Q} &= 1, \\ - \underbrace{\sum_{i:(i,S)\in E} x_{iS}}_{\text{inflow into node } S} &= -1. \end{aligned}$$

These linear equations conservation equations can be written in compact form as

$$Hx = d,$$

where $d = (1, 0, \ldots, 0, -1)^\top$ and H is the so-called node-edge incidence matrix.

Definition 4.1.1 **(Node-edge incidence matrix)**

Let $G = (V, E)$ be a digraph with nodes $V = \{1, \ldots, n\}$ and edges $E = \{e_1, \ldots, e_m\}$. The matrix $H \in \mathbb{R}^{n\times m}$ with entries

$$h_{ij} = \begin{cases} 1, & \text{if } i \text{ is the tail of edge } e_j, \\ -1, & \text{if } i \text{ is the head of edge } e_j, \\ 0, & \text{otherwise} \end{cases}, \qquad i = 1, \ldots, n,\ j = 1, \ldots, m,$$

is called (node-edge) incidence matrix of G.

Remark 4.1.2

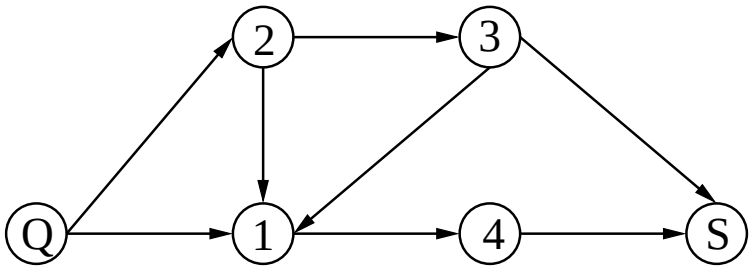
- *Every column of H refers to an edge $(i,j) \in E$ and is defined as follows:*

$$\begin{array}{c} (i,j) \\ \begin{pmatrix} \\ 1 \\ \\ -1 \\ \\ \end{pmatrix} \end{array} \begin{array}{ll} \\ \leftarrow \text{ row } i \\ \\ \leftarrow \text{ row } j \\ \\ \end{array} \qquad \begin{array}{ll} \text{consequence:} & \text{outflow is positive} \\ & \text{inflow is negative} \end{array}$$

- *Some authors define the incidence matrix to be the negative of H.*

Example 4.1.3

Directed graph $G = (V,E)$ with nodes $V = \{Q,1,2,3,4,S\}$ and edges $E = \{(Q,1),(Q,2),(2,1),(1,4),(2,3),(3,1),(3,S),(4,S)\}$:



Incidence matrix H:

$$H = \begin{array}{c} \begin{matrix} (Q,1) & (Q,2) & (2,1) & (1,4) & (2,3) & (3,1) & (3,S) & (4,S) \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \end{pmatrix} \end{array} \begin{array}{l} \text{node } Q \\ \text{node } 1 \\ \text{node } 2 \\ \text{node } 3 \\ \text{node } 4 \\ \text{node } S \end{array}$$

Each row of H corresponds to a node in V and each column of H corresponds to an edge in E according to the ordering in V and E.
We can immediately see that the conservation equation is equivalent to $Hx = d$ where each component d_i of d denotes a demand ($d_i < 0$) or a supply ($d_i \geq 0$). For instance, with

$$x = (x_{Q1}, x_{Q2}, x_{21}, x_{14}, x_{23}, x_{31}, x_{3S}, x_{4S})^\top$$

conservation at the intermediate node 2 *with* $d_2 = 0$ *reads as*

$$-x_{Q2} + x_{21} + x_{23} = 0 (= d_2),$$

which on the left is just the row corresonding to node 2 *in* H *multiplied by* x*.*

Summarising, the shortest path problem can written as the following linear program.

$$\text{Minimise} \quad \sum_{(i,j)\in E} c_{ij}x_{ij} \qquad \text{subject to} \qquad Hx = d,\ x_{ij} \geq 0,\ (i,j) \in E. \tag{4.1}$$

Herein, the vector x has the components x_{ij}, $(i,j) \in E$, in the same order as in E.
What can we say about solvability of the problem? The problem is feasible, if there exists a $Q-S-$path. If, in addition, $c_{ij} \geq 0$ holds, then the objective function is bounded below on the feasible set and thus an optimal solution exists.
There remains one question: Does the problem always have a solution with $x_{ij} \in \{0,1\}$? Fortunatel yes, because the matrix H is a so-called totally unimodular matrix and d is integral.

Definition 4.1.4 **(unimodular, totally unimodular)**

- *An integer matrix* $A \in \mathbb{Z}^{n\times n}$ *is called unimodular if* $\det(A) = \pm 1$*.*
- *An integer matrix* H *is called totally unimodular, if every square, non-singular submatrix of* H *is unimodular.*

It can be shown that the incidence matrix H as defined above is totally unimodular, see Papadimitriou and Steiglitz [7], Section 13.2, Corollary on page 318. Moreover, the following result holds:

Theorem 4.1.5

Let $H \in \mathbb{Z}^{m\times n}$ *be totally unimodular and let* $b \in \mathbb{Z}^m$ *be integral. Then all vertices of the sets*

$$\{x \in \mathbb{R}^n \mid Hx = b,\ x \geq 0\}$$

and

$$\{x \in \mathbb{R}^n \mid Hx \leq b,\ x \geq 0\}$$

are integral.

Proof: see Papadimitriou and Steiglitz [7], Section 13.2.

This theorem has a very important consequence. Recall that the simplex method always terminates in an optimal vertex of the feasible set (assuming that an optimal solution

exists). Hence, the simplex method, when applied to the above LP formulation (4.1) of the shortest path problem, automatically terminates in an integer solution with $x_{ij} \in \{0,1\}$. Hence, constraints of type $x_{ij} \in \{0,1\}$ or $x_{ij} \in \mathbb{Z}$ can be omitted.
More generally, the above holds for every LP formulation of network optimisation problems which involve the node-edge incidence matrix H and involve integer data only, for instance transportation problems, assignment problems, and maximum flow problems.

Remark 4.1.6

Caution: The above does not imply that every optimal solution is integral! It just states that optimal vertex solutions are integral, but there may be non-vertex solutions as well which are not integral.

4.2 Dijkstra's Algorithm

In this section we restrict the discussion to shortest paths problems with non-negative weights:

Assumption 4.2.1 (Non-negative weights)
It holds $c_{ij} \geq 0$ for every $(i,j) \in E$ in the network N.

It can be shown that the rank of the $n \times m$ incidence matrix H is $n-1$ if the graph (V,E) is connected. In particular, the last constraint in $Hx = d$ which corresponds to the sink S is linearly dependent (redundant) and can be omitted. This can be seen as follows: As each column of H has one entry 1 and one entry -1, summing up all rows of H yields a zero vector. Hence, the rank of H is at most $n-1$. As (V,E) is supposed to be connected the underlying undirected graph has a spanning tree T with $n-1$ edges. As S is a sink, there exists exactly one edge connecting a node v with S in T. Deleting S from the tree (i.e. deleting the last row of H) shows that in the corresponding column in H only one entry 1 is left. By row and column permutations we can move this entry 1 into the left upper position of the matrix. Now we can apply the same procedure to the node v connected to S and after $n-1$ steps we end up with a permuted matrix which has entries ± 1 on the diagonal and zeros below the diagonal and thus the matrix is non-singular.
Using this reasoning, (4.1) is equivalent to the problem

$$\text{Minimise} \quad \sum_{(i,j)\in E} c_{ij}x_{ij} \quad \text{subject to} \quad \tilde{H}x = \tilde{d},\ x_{ij} \geq 0,\ (i,j) \in E, \tag{4.2}$$

where $\tilde{d} = (1,0,\ldots,0)^\top \in \mathbb{R}^{n-1}$ and $\tilde{H}$ is the incidence matrix H with the last row omitted.
The dual of (4.2) reads as

$$\text{Maximise} \quad \lambda_1 \quad \text{subject to} \quad \lambda_i - \lambda_j \leq c_{ij}, \quad (i,j) \in E. \tag{4.3}$$

As $c_{ij} \geq 0$ we can easily provide a feasible dual point: $\lambda_i = 0$ for every $i = 1, \ldots, n-1$. We intend to apply the Primal-Dual Algorithm 2.4.1 to (4.2) and define the index set

$$B = \{(i,j) \in E \mid \lambda_i - \lambda_j = c_{ij}\}.$$

The restricted primal problem with artificial slack variables $y = (y_1, \ldots, y_{n-1})^\top$ reads as

$$\text{Minimise} \quad \sum_{i=1}^{n-1} y_i \qquad \text{subject to} \qquad \tilde{H}_B x_B + y = \tilde{d},\ x_{ij} \geq 0,\ (i,j) \in B,\ y \geq 0. \tag{4.4}$$

The dual of the restricted primal is given by

$$\begin{aligned} &\text{Maximise} && \lambda_1 \\ &\text{subject to} && \lambda_i - \lambda_j \leq 0, \qquad (i,j) \in B, \\ &&& \lambda_i \leq 1, \qquad i = 1, \ldots, n-1, \\ &&& \lambda_n = 0. \end{aligned} \tag{4.5}$$

The constraint $\lambda_n = 0$ is due to the fact, that we deleted the last row of H. The problem (4.5) can be easily solved. As $\lambda_1 = \lambda_Q \leq 1$ has to be maximised, we can set $\lambda_1 = 1$ and propagate the 1 to all nodes reachable from node 1 in order to satisfy the constraints $\lambda_i - \lambda_j \leq 0$. Hence, an optimal solution of (4.5) is given by

$$\bar{\lambda}_i = \begin{cases} 1, & \text{if node } i \text{ is reachable from node } Q = 1 \text{ using edges in } B, \\ 0, & \text{if node } S = n \text{ is reachable from node } i \text{ using edges in } B, \\ 1, & \text{otherwise,} \end{cases} \qquad i = 1, \ldots, n-1.$$

Notice that there may be other solutions to (4.5). We then proceed with the primal-dual algorithm by computing the step-size

$$t_{max} = \min_{(i,j) \notin B : \bar{\lambda}_i - \bar{\lambda}_j > 0} \{c_{ij} - (\lambda_i - \lambda_j)\},$$

updating $\lambda = \lambda + t_{max}\bar{\lambda}$, and performing the next iteration of the primal-dual algorithm.

Optimality:

If it turns out that there is a path from the source Q to the sink S using edges in the index set B, then $\lambda_1 = 0$ and the optimal objective function values of the dual of the restricted primal and the restricted primal are both zero and the primal-dual algorithm stops with an optimal solution.

Observations and interpretations:

The following theorem gives an important explanation of what is going on in the primal-dual algorithm for the shortest path problem. We use the set

$$W = \{i \in V \mid S \text{ is reachable from } i \text{ by edges in } B\} = \{i \in V \mid \bar{\lambda}_i = 0\}.$$

Theorem 4.2.2

Let $N = (V, E, c)$ with $|V| = n$ be a network with positive weight $c_{ij} > 0$ for every $(i, j) \in E$.

(a) If the primal-dual algorithm starts with the feasible dual point $\lambda_i = 0$, $i = 1, \ldots, n$, then the shortest path from Q to S is found after at most $n - 1$ iterations.

(b) In every iteration an edge $(i, j) \in E$ with $i \notin W$ and $j \in W$ is added to the index set B. This edge will stay in B for all following iterations.

(c) The value of the variable λ_i with $i \in W$ is the length of a shortest path from node i to node $S = n$.

Proof: The assertions can be shown by induction.

Remark 4.2.3

- *The assumption of positive weights c_{ij} in the theorem is essential for the interpretation of the dual variables as lengths of shortest paths.*
- *The algorithm not only finds the lengths of shortest paths but also for every node in W the shortest path to S.*
- *If t_{max} is not uniquely determined by some $(i, j) \notin B$, then every such (i, j) can be added to B. Notice that the above theorem only provides an interpretation for those dual variables λ_i for which i belongs to W (there may be other variables in B).*

This essentially is the working principle of Dijkstra's algorithm. An efficient implementation looks as follows (in contrast to the primal-dual algorithm we start with node Q instead of S):

Algorithm 4.2.4 (Dijkstra's Algorithm)

(0) Initialisation: Set $W = \{1\}$, $d(1) = 0$, and $d(i) = \infty$ for every $i \in V \setminus \{1\}$.

(1) For every $i \in V \setminus \{1\}$ with $(1, i) \in E$ set $d(i) = c_{1i}$ and $p(i) = 1$.

(2) **While** $W \neq V$ **do**
Find $k \in V \setminus W$ with $d(k) = \min\{d(i) \mid i \in V \setminus W\}$.
Set $W = W \cup \{k\}$.
For every $i \in V \setminus W$ *with* $(k, i) \in E$ **do**

If $d(i) > d(k) + c_{ki}$ **then**
Set $d(i) = d(k) + c_{ki}$ *and* $p(i) = k$.
end if
end do
end do

Output:

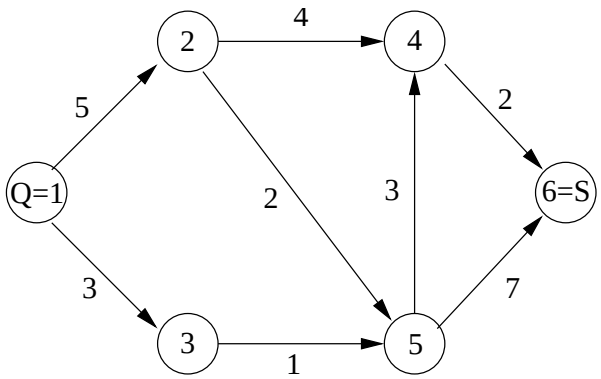
- shortest paths from node $Q = 1$ to all $i \in V$ and their lengths.
- for every $i \in V$, $d(i)$ is the length of the shortest $Q - i-$path. $p(i)$ denotes the predecessor of node i on the shortest path.
- if i is not reachable from Q, then $d(i) = \infty$ and $p(i)$ is undefined.

Complexity:

The While-loop has $n - 1$ iterations. Each iteration requires at most n steps in the For-Loop within the While-loop to update d. Likewise at most n steps are needed to find the minimum $d(k)$ within the While-loop. Hence, the overall complexity of Dijkstra's algorithm is $\mathcal{O}(n^2)$ where n denotes the number of nodes in the network.

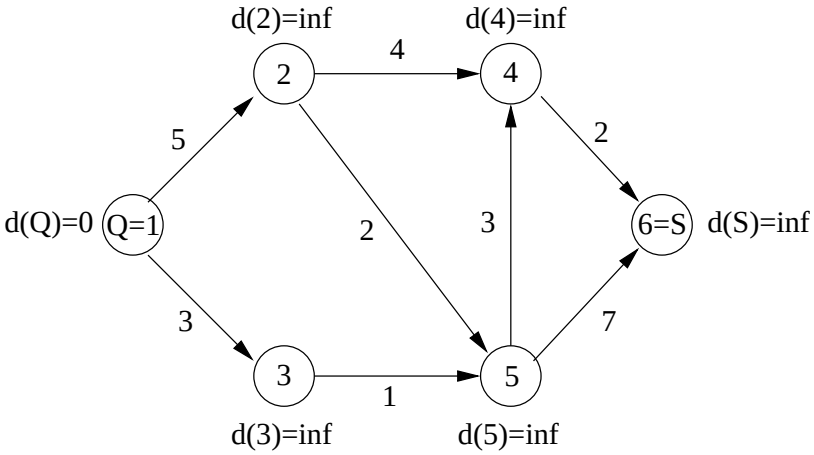
Example 4.2.5

Find the shortest path from $Q = 1$ *to* $S = 6$ *using Dijkstra's algorithm for the following network.*

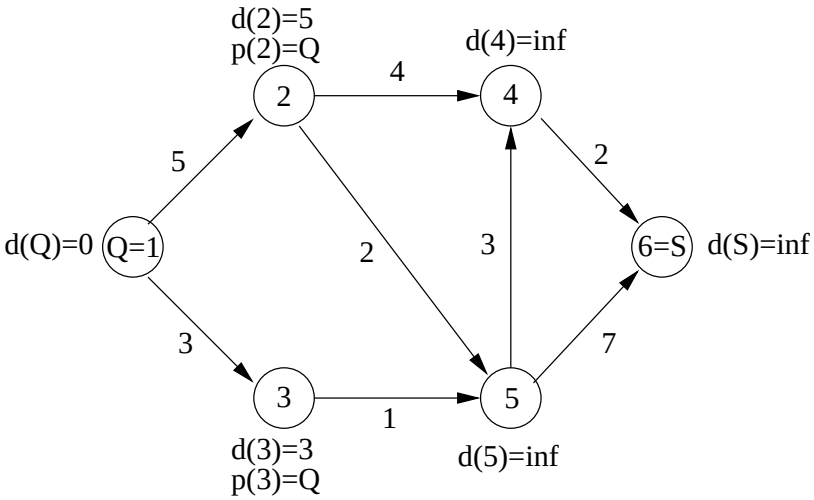


Dijkstra's algorithm produces the following intermediate steps:

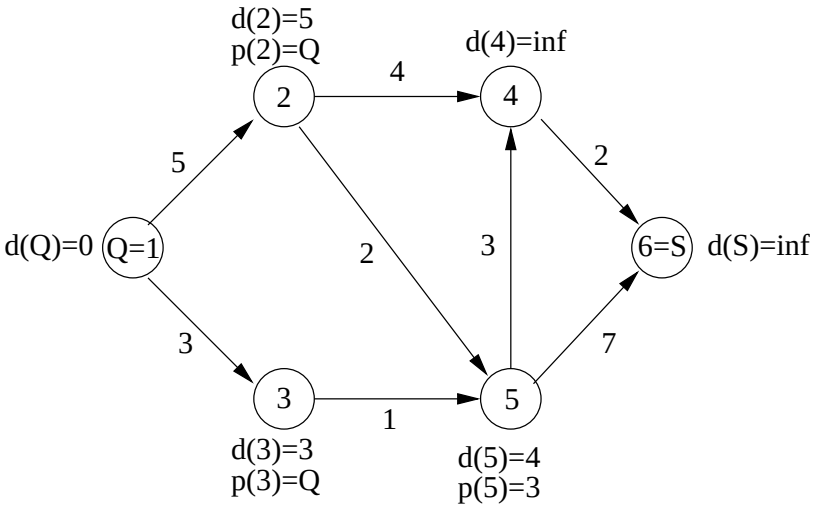
Initialisation: W={1}



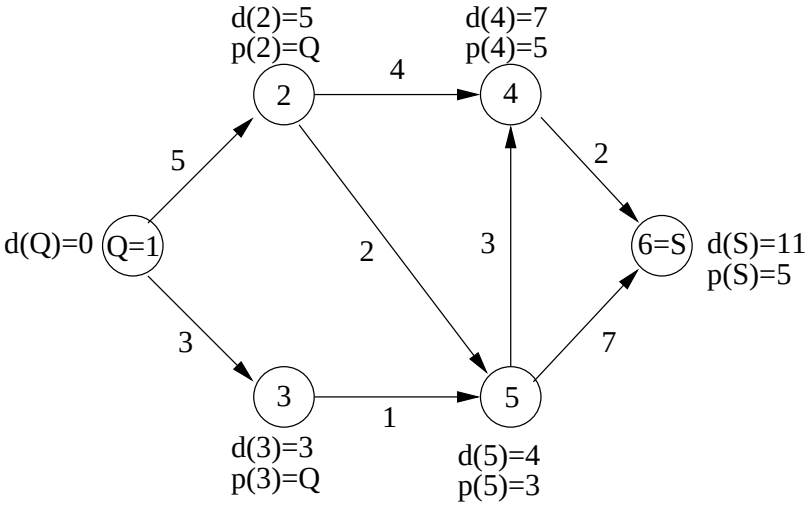
Step (1): W={1}



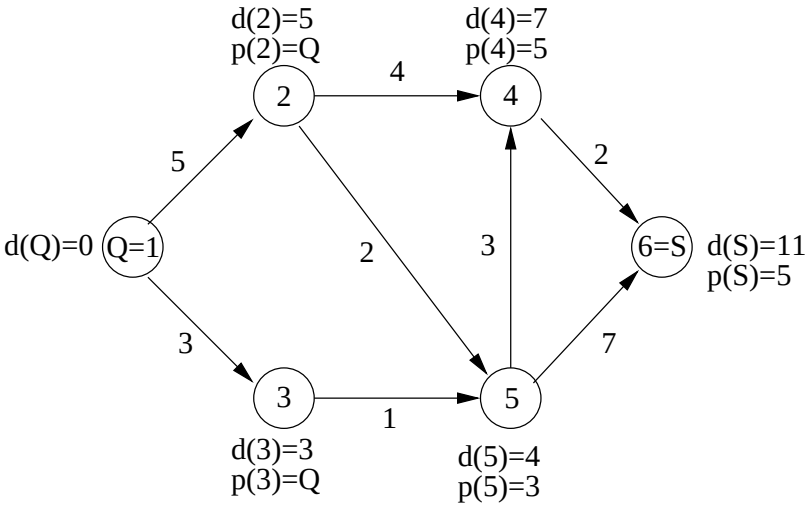
Step (2), Iteration 1: W={1,3}, k=3



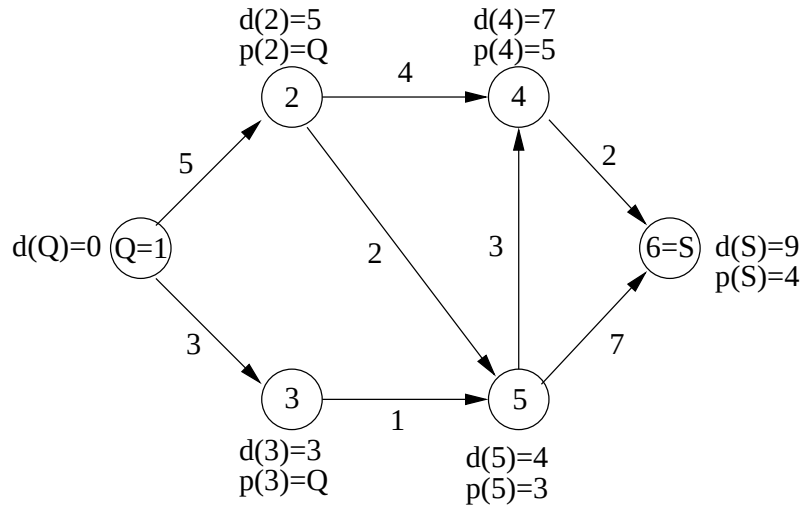
Step (2), Iteration 2: W={1,3,5}, k=5



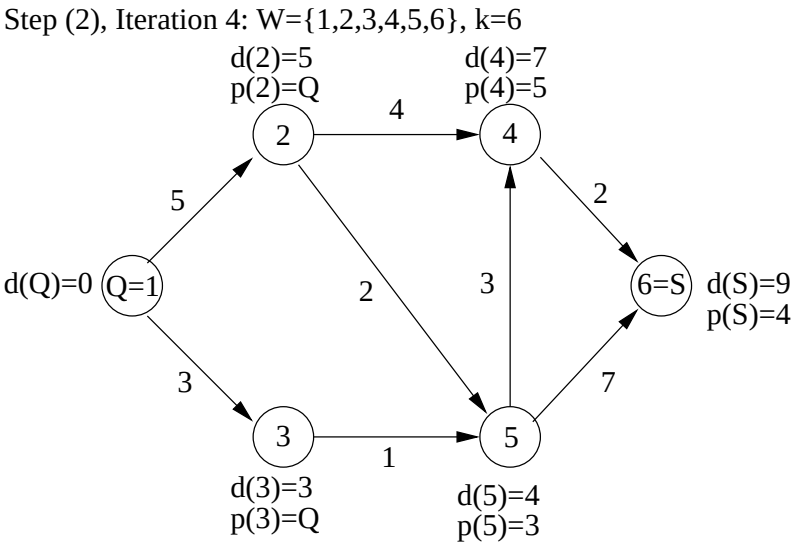
Step (2), Iteration 3: W={1,2,3,5}, k=2



Step (2), Iteration 3: W={1,2,3,4,5}, k=4



Final result: The shortest path from Q to S is the path $[Q, 3, 5, 4, S]$ with length 9.



Remark 4.2.6

- *Dijkstra's algorithm only works for non-negative weights $c_{ij} \geq 0$. In the presence of negative weights it may fail.*
- *The assumptions that Q be a source and S be a sink are not essential and can be dropped.*

4.3 Algorithm of Floyd-Warshall

In contrast to the previous section, we now allow negative weights $c_{ij} < 0$, but with the restriction that the network does not contain a circuit of negative length.

Assumption 4.3.1

The network N does not contain a circuit of negative length.

The algorithm of Floyd-Warshall computes not only the shortest paths from Q to any other node $j \in V$ (as Dijkstra's algorithm) but also the shortest paths between any two nodes $i \in V$ and $j \in V$. Moreover, it is very simple to implement. Finally, the algorithm works for negative weights and it detects circuits of negative length. It is not a primal-dual algorithm.

Algorithm 4.3.2 (Floyd-Warshall)

(0) Initialisation:
Set $d_{ij} = \infty$ for every (i,j), $i,j = 1,\ldots,n$.
Set $d_{ij} = c_{ij}$ for every $(i,j) \in E$.

Set $d_{ii} = 0$ *for every* $i = 1, \dots, n$.
Set $p_{ij} = i$ *for every* $i, j \in V$.

(1) **For** $j = 1, \dots, n$ **do**
  For $i = 1, \dots, n$, $i \neq j$ **do**
    For $k = 1, \dots, n$, $k \neq j$ **do**
      If $d_{ik} > d_{ij} + d_{jk}$ **then**
        Set $d_{ik} = d_{ij} + d_{jk}$ *and* $p_{ik} = p_{jk}$.
      end if
    end do
  end do
end do

Output:

- Distance matrix
$$D = \begin{pmatrix} d_{11} & \cdots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{n1} & \cdots & d_{nn} \end{pmatrix},$$
where d_{ij} is the length of a shortest path from i to j.

- Predecessor matrix
$$P = \begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{n1} & \cdots & p_{nn} \end{pmatrix},$$
where (p_{ij}, j) is the final edge of a shortest path from i to j, that is p_{ij} is the predecessor of the node j on a shortest $i-j-$path (if such a path exists).

Complexity:
Each For-loop has at most n iterations. Hence, the overall complexity of the algorithm of Floyd-Warshall is $\mathcal{O}(n^3)$ where n denotes the number of nodes in the network.

The correctness of the algorithm is established in the following theorem.

Theorem 4.3.3

For every $i, j \in V$ *the algorithm of Floyd-Warshall correctly computes the length* d_{ij} *of a shortest path from* i *to* j*, if no circuit of negative length exists in the network* N.

Proof: We will show the following: After the outer loop for $j = 1, \dots, j_0$ is completed, the variable d_{ik} (for all i and k) contains the length of a shortest $i-k-$path with intermediate nodes $v \in \{1, \dots, j_0\}$ only.

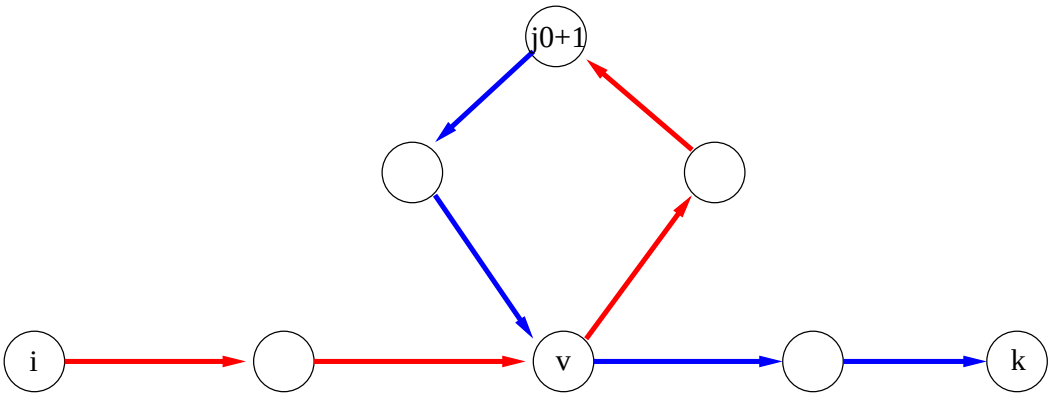
Induction for j_0: For $j_0 = 0$ the assertion is true owing to the initialisation in step (0).

Let the assertion hold for some $j_0 \in \{0, \ldots, n-1\}$. We have to show the assertion to be true for $j_0 + 1$ as well.

Let i and k be arbitrary nodes. According to the induction assumption, d_{ik} contains the length of an $i-k-$path with intermediate nodes $v \in \{1, \ldots, j_0\}$ only. In the loop for $j = j_0 + 1$, d_{ik} is replaced by $d_{i,j_0+1} + d_{j_0+1,k}$, if this value is smaller.

It remains to show that the corresponding $i-(j_0+1)-$path P and the $(j_0+1)-k-$path $\tilde{P}$ have no inner nodes in common (otherwise the combined way $P \cup \tilde{P}$ would not be a path by definition).

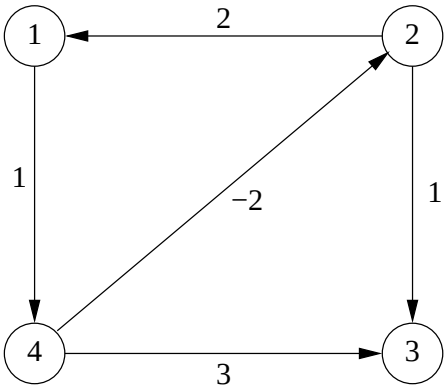
Suppose that P and $\tilde{P}$ have an inner node v in common, see figure below (P is colored red and $\tilde{P}$ is blue).



Then the combined walk $P \cup \tilde{P}$ contains at least one circuit from v to $j_0 + 1$ (in P) and back to v (in $\tilde{P}$). By assumption any such circuit does not have negative length and thus they can be shortcut. Let R denote the shortcut $i-k-$path. This path uses nodes in $\{1, \ldots, j_0\}$ only and the length is no longer than $d_{i,j_0+1} + d_{j_0+1,k} < d_{ik}$. But this contradicts the induction assumption as d_{ik} was supposed to be the length of a shortest $i-k-$path using nodes in $\{1, \ldots, j_0\}$ only. □

Example 4.3.4

Consider the following network which contains a circuit. Notice that the circuit does not have negative length.



The algorithm of Floyd-Warshall produces the following result:

Initialisation:

D :

0	∞	∞	1
2	0	1	∞
∞	∞	0	∞
∞	-2	3	0

P :

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

$j = 1$*:*

D :

0	∞	∞	1
2	0	1	3
∞	∞	0	∞
∞	-2	3	0

P :

1	1	1	1
2	2	2	1
3	3	3	3
4	4	4	4

$j = 2$*:*

D :

0	∞	∞	1
2	0	1	3
∞	∞	0	∞
0	-2	-1	0

P :

1	1	1	1
2	2	2	1
3	3	3	3
2	4	2	4

$j = 3$*:*

D :

0	∞	∞	1
2	0	1	3
∞	∞	0	∞
0	-2	-1	0

P :

1	1	1	1
2	2	2	1
3	3	3	3
2	4	2	4

$j = 4$*:*

D :

0	-1	0	1
2	0	1	3
∞	∞	0	∞
0	-2	-1	0

P :

1	4	2	1
2	2	2	1
3	3	3	3
2	4	2	4

From the predecessor matrix P we can easily reconstruct an optimal path. For instance: The shortest path from node 1 to node 3 has length $d_{13} = 0$. The entry $p_{13} = 2$ in P tells

us the predecessor of node 3 on that path, that is node 2. In order to get the predecessor of node 2 on that path we use the entry $p_{12} = 4$. Now, the predecessor of node 4 on that path is $p_{14} = 1$. Hence, we have reconstructed an optimal path from node 1 to node 3 by backtracking. The path is $[1, 4, 2, 3]$.

The reasoning behind this backtracking method is as follows: Suppose the $v_1 - v_{k+1}-$path $[v_1, v_2, \ldots, v_k, v_{k+1}]$ has minimal length. Then for every $1 \leq j < \ell \leq k+1$ the $v_j - v_\ell-$path $[v_j, \ldots, v_\ell]$ connecting v_j and v_ℓ has minimal length as well. In other words: Subpaths of optimal paths remain optimal.

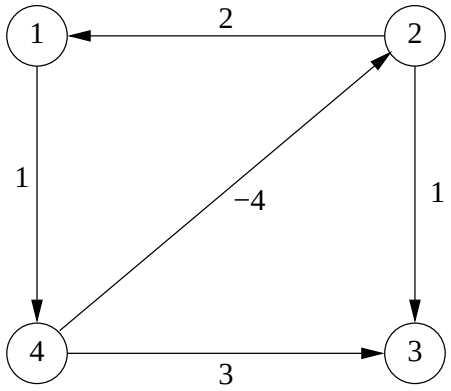
Remark 4.3.5 **(Circuit Detection)**

It can be shown that the diagonal elements d_{ii}, $i = 1, \ldots, n$, remain non-negative if and only if the network does not contain a circuit of negative length.

Hence, the algorithm of Floyd-Warshall is able to detect circuits of negative length. It can be halted if a diagonal element in D becomes negative.

Example 4.3.6 **(Exercise)**

Apply the algorithm of Floyd-Warshall to the following network which contains a circuit of negative length.



Chapter 5

Maximum Flow Problems

In this chapter we will discuss a special class of network optimisation problems – maximum flow problems – and the algorithm of Ford and Fulkerson. We will see that this algorithm has polynomial complexity. We have seen a specific example of a maximum flow problem already: the oil company problem in Example 2.2.3.

5.1 Problem Statement

Let a directed network $N = (V, E, u)$ with set of nodes $V = \{Q = 1, 2, \ldots, n = S\}$ and set of edges E, $|E| = m$, be given. Let $u : E \to \mathbb{R}$ with $(i, j) \in E \mapsto u(i, j) = u_{ij}$ denote maxmimal capacities. Let $x : E \to \mathbb{R}$ with $(i, j) \in E \mapsto x(i, j) = x_{ij}$ denote the amount of goods (e.g.oil) transported on edge $(i, j) \in E$.

Let Q and S be two distinct nodes (source and sink, respectively). The remaining nodes $2, 3, \ldots, n-1$ are called interior nodes.

Assumption 5.1.1

(i) Let Q be a source (i.e. Q has no predecessor and no edge enters Q) and let S be a sink (i.e. S has no successor and no edge leaves S).

(ii) The amount x_{ij} transported on edge $(i, j) \in E$ is subject to capacity constraints

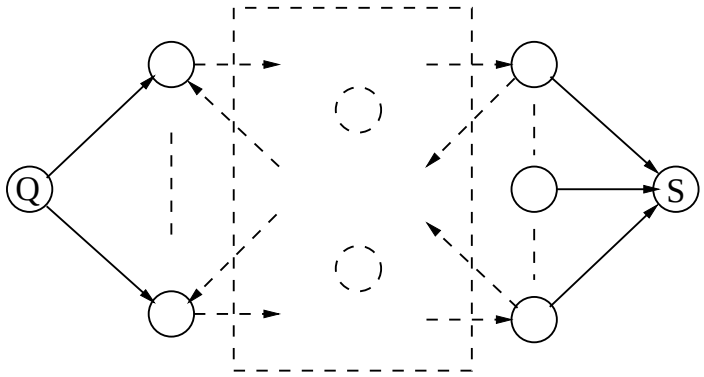
$$0 \leq x_{ij} \leq u_{ij}, \qquad u_{ij} \geq 0, \ \forall (i, j) \in E. \tag{5.1}$$

Herein, $u_{ij} = \infty$ is permitted.

(iii) The conservation equations

$$\underbrace{\sum_{k:(j,k)\in E} x_{jk}}_{\text{outflow out of node } j} - \underbrace{\sum_{i:(i,j)\in E} x_{ij}}_{\text{inflow into node } j} = 0, \quad j \in V \backslash \{Q, S\}. \tag{5.2}$$

hold at every interior node $j = 2, \ldots, n-1$.



Remark 5.1.2

Problems with multiple sources or sinks are not excluded by the above assumptions. In fact, such problems can be transformed into problems with only one source and one sink. In order to transform a problem with multiple sources, an artificial 'super source' and edges without capacity restrictions leaving the super source and entering the original sources can be introduced. Likewise, a 'super sink' and edges without capacity restrictions leaving the original sinks and entering the super sink can be introcuded for problems with multiple sinks. For the transformed problem, assumption (i) is satisfied.

Definition 5.1.3 **(Flow, feasible flow, volume, maximum flow)**

- *A flow $x : E \to \mathbb{R}$ through the network is an assignment of values $x(i,j) = x_{ij}$, such that the conservation equations (5.2) hold.*
- *A flow x is feasible, if the capacity constraints (5.1) are satisfied.*
- *The value*

$$v = \sum_{(Q,j)\in E} x_{Qj} \left(= \sum_{(i,S)\in E} x_{iS} \right) \tag{5.3}$$

  is called volume of the flow x.
- *A feasible flow with maximum volume is called maximum flow through the network.*

Using these definitions, the maximum flow problem reads as follows.

Maximum flow problem:
Find a maximum flow x through the network.

Solvability:

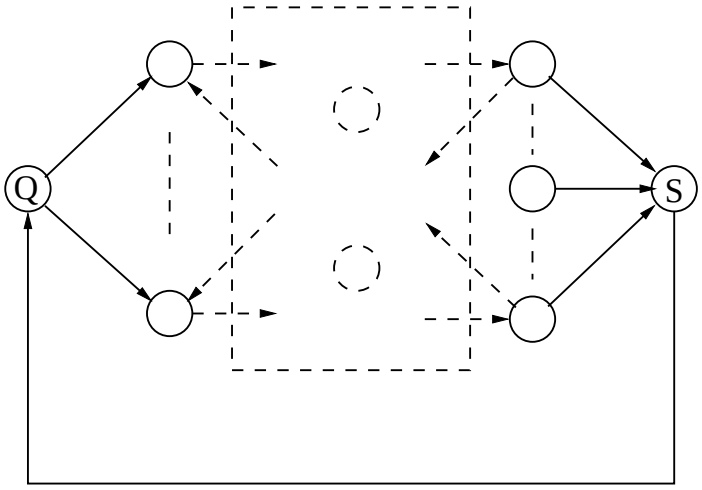
- The maximum flow problem is feasible as $x_{ij} = 0$ for every $(i, j) \in E$ satisfies all constraints.

- If all capacity bounds u_{ij} satisfy $0 \leq u_{ij} < \infty$, then the maximum flow problem has an optimal solution as in this case the feasible set is compact and non-empty.

5.2 Linear Programming Formulation

We intend to write the maximum flow problem as a linear program and note that digraphs can be uniquely described by matrices.

The maximum flow problem can be formulated as a linear program, if an additional edge (S, Q) from the sink to the source with no capacity restriction is introduced, cf. figure below.



The volume v is transported along edge (S, Q) and the conservation equations are extended to the source Q and the sink S. The conservation equations can be written in a compact way using the incidence matrix H (compare Definition 4.1.1) of the original network as

$$Hx + v \cdot d = 0,$$

where $d = (-1, 0, \ldots, 0, 1)^\top \in \mathbb{R}^n$ is the incidence column for the additional edge (S, Q). The maximum flow problem is equivalent to the following linear program:

$$\begin{array}{lll} \text{Maximise} & v & \\ \text{subject to} & Hx + v \cdot d = 0, & \\ & 0 \leq x_{ij} \leq u_{ij}, & (i,j) \in E, \\ & 0 \leq v. & \end{array}$$

The dual problem reads as

$$\begin{aligned} \text{Minimise} \quad & \sum_{(i,j)\in E} z_{ij}u_{ij} \\ \text{subject to} \quad & y_j - y_i - z_{ij} \leq 0, \quad (i,j)\in E, \\ & y_S - y_Q \geq 1, \\ & z_{ij} \geq 0, \quad (i,j)\in E. \end{aligned}$$

where z_{ij}, $(i,j)\in E$, and y_i, $i\in V$, denote the dual variables. The dual variables y_i, $i\in V$, are known as node numbers.
For the maximum flow problem the complementary slackness conditions read as

$$\begin{aligned} 0 &= x_{ij}(y_j - y_i - z_{ij}) \qquad \text{for } (i,j)\in E, \\ 0 &= z_{ij}(u_{ij} - x_{ij}) \qquad \text{for } (i,j)\in E, \\ 0 &= v(-1 + y_S - y_Q). \end{aligned} \tag{5.4}$$

5.3 Minimal cuts and maximum flows

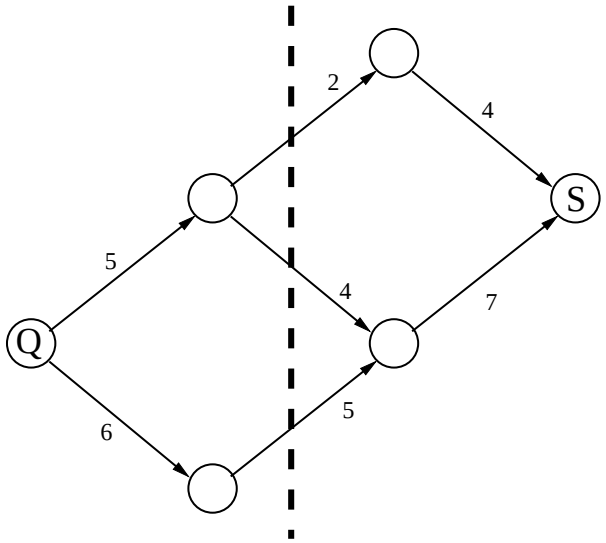
Definition 5.3.1 **(Cut, capacity of a cut)**

- *A subset $C\subset V$ of nodes is called cut in a network, if $Q\in C$ and $S\notin C$.*
- *The capacity of a cut C is defined to be the value*

$$\sum_{j\in C, k\notin C} u_{jk}.$$

Example 5.3.2

The capacity of the cut is $2+4+5=11$.



Corollary 5.3.3

Let C be a cut and x a flow with volume v. Then

$$v = \sum_{j\in C, k\notin C} x_{jk} - \sum_{i\notin C, j\in C} x_{ij}. \tag{5.5}$$

Proof: Summation of the conservation equations w.r.t. $j \neq Q, j \in C$ and adding the identy $v = \sum_{(Q,k)\in E} x_{Qk}$ yields

$$v + \sum_{j\in C, j\neq Q} \sum_{(i,j)\in E} x_{ij} = \sum_{(Q,k)\in E} x_{Qk} + \sum_{j\in C, j\neq Q} \sum_{(j,k)\in E} x_{jk}.$$

Changing the order of summation leads to

$$\begin{aligned} v + \sum_{j\in C, i\in C, (i,j)\in E} x_{ij} + \sum_{j\in C, i\notin C, (i,j)\in E} x_{ij} &= \sum_{(Q,k)\in E} x_{Qk} + \sum_{j\in C, j\neq Q, k\in C} x_{jk} + \sum_{j\in C, j\neq Q, k\notin C} x_{jk} \\ &= \sum_{j\in C, k\in C} x_{jk} + \sum_{j\in C, k\notin C} x_{jk} \end{aligned}$$

The assertion follows immediately:

$$v = \sum_{j\in C, k\notin C} x_{jk} - \sum_{j\in C, i\notin C, (i,j)\in E} x_{ij}.$$

□

Remark 5.3.4

If $C = \{Q\}$ and $C = V\backslash\{S\}$, respectively, are chosen in the preceding corollary, then (5.3) is obtained immediately.

In particular, it follows that the volume of a flow is bounded by the capacity of a cut:

$$v = \sum_{j\in C, k\notin C} x_{jk} - \sum_{i\notin C, j\in C} x_{ij} \overset{x_{jk}\leq u_{jk}, x_{ij}\geq 0}{\leq} \sum_{j\in C, k\notin C} u_{jk} \tag{5.6}$$

This fact can be described by the dual problem:

Theorem 5.3.5

Every cut C defines a feasible point of the dual problem according to

$$z_{ij} = \begin{cases} 1, & \text{if } i \in C,\ j \notin C, \\ 0, & \text{otherwise} \end{cases}$$

$$y_i = \begin{cases} 0, & \text{if } i \in C, \\ 1, & \text{otherwise} \end{cases}$$

The value of the dual problem at this point equals the capacity of the cut.

Proof: Feasibility immediately follows by computing the inequality constraints for the cases $i, j \in C$, and $i, j \notin C$, and $i \in C, j \notin C$, and $i \notin C, j \in C$. The values of the inequality constraints for these cases turn out to be $0, 0, 0, -1$, respectively. As $z_{ij} = 1$ holds exactly for those edges that define the capacity of the cut, the assertion follows. □

Remark 5.3.6

The theorem states that the capacity of a cut is greater than the volume v of a flow x, because owing to the weak duality theorem, the objective function value of the dual problem is always greater than or equal to the objective function value v of the primal problem.

Theorem 5.3.7 (Max-Flow Min-Cut)

Every maximum flow problem has exactly one of the following properties:

(a) There exist feasible flows with arbitrarily large volume and every cut has an unbounded capacity.

(b) There exists a maximum flow whose volume is equal to the minimal capacity of a cut.

Proof: We have seen that every maximum flow problem can be transformed into a linear program. According to the fundamental theorem of linear programming, every linear program satisfies exactly one of the following alternatives:

(i) It is unbounded.

(ii) It has an optimal solution.

(iii) It is infeasible.

The third alternative does not apply to maximum flow problems as the zero flow $x = 0$ is feasible. Consequently, the problem is either unbounded or it has an optimal solution. If the problem is unbounded, then there exists a flow with unbounded volume. According to (5.6) for every cut C and every flow x with volume v it holds

$$v \leq \sum_{j \in C, k \notin C} u_{jk}. \tag{5.7}$$

As the volume is unbounded, the capacity of every cut has to be infinity.
If the problem has an optimal solution, the simplex method finds an optimal solution x with node numbers y_k such that

$$0 > y_j - y_i \quad \Rightarrow \quad x_{ij} = 0, \tag{5.8}$$

$$0 < y_j - y_i \quad \Rightarrow \quad x_{ij} = u_{ij} \tag{5.9}$$

hold owing to the complementary slackness conditions (5.4) and the first constraint of the dual problem. The second constraint of the dual problem implies $y_S - 1 \geq y_Q$, hence $y_S > y_Q$. Hence, the set C of nodes k with $y_k \leq y_Q$ defines a cut. Notice that y_S does not belong to C as $y_S \not\leq y_Q$ holds. (5.9) implies $x_{ij} = u_{ij}$ for every original edge (i, j) with $i \in C$, $j \notin C$. (5.8) implies $x_{ij} = 0$ for every original edge (i, j) with $i \notin C$, $j \in C$. For the cut C we obtain with (5.5) the relation

$$v = \sum_{j \in C, k \notin C} u_{jk}.$$

On the other hand, (5.6) holds for an arbitrary cut. Thus, C is a cut with minimal capacity. As x is a maximum flow the assertion follows. □

Theorem 5.3.8

If every finite capacity u_{ij} is a positive integer and there exists a maximum flow, then there exists an integral maximum flow.

5.4 Algorithm of Ford and Fulkerson (Augmenting Path Method)

The algorithm of Ford and Fulkerson is motivated by the primal-dual algorithm. It turns out that the dual of the restricted primal problem corresponds to finding a so-called augmenting path from Q to S. The primal-dual algorithm then aims at increasing the volume of the flow by adapting the current flow such that it meets either a lower or an upper capacity constraint on this path.

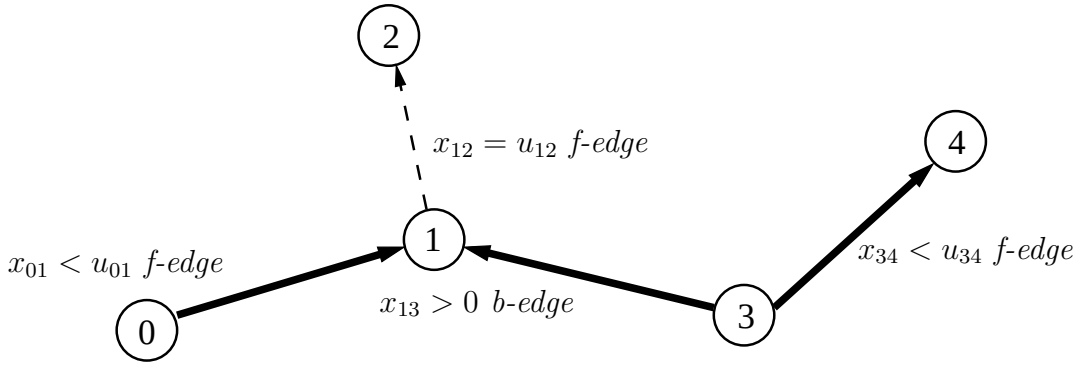
Definition 5.4.1 (useable path, augmenting path)

Let x be a feasible flow with volume v.

- *Let a $v_0 - v_r-$path $[v_0, v_1, \ldots, v_r]$ in the underlying undirected network with nodes $v_0, v_1, \ldots, v_r$ be given. The path is called useable, if $x_{ij} < u_{ij}$ holds for every forward edge and $x_{ij} > 0$ for very backward edge. Herein, an edge (i, j) is called forward edge (f-edge), if $i = v_k$ and $j = v_{k+1}$ hold for some k. Similarly, an edge (i, j) is called backward arc (b-edge), if $i = v_{k+1}$ and $j = v_k$ hold for some k.*

- *An useable $Q - S-$path is called augmenting path.*

Example 5.4.2 **(Useable path)**



The idea of the algorithm of Ford and Fulkerson is to create augmenting paths. It is exploited that a flow is maximal if and only if no augmenting path with useable edges exists.

Algorithm 5.4.3 **(Ford and Fulkerson (Augmenting Path Method))**

(0) Let a feasible flow x be given (e.g. $x_{ij} = 0$ for every $(i,j) \in E$).

(1) Find an augmenting path $W = [Q = v_0, v_1, \ldots, v_r = S]$. If no augmenting path exists, STOP: x is a maximum flow.

(2) Compute the flow $\bar{x}$ according to

$$\bar{x}_{ij} = \begin{cases} x_{ij} + d, & \text{if } (i,j) \text{ is an f-edge in the path } W, \\ x_{ij} - d, & \text{if } (i,j) \text{ is a b-edge in the path } W, \\ x_{ij}, & \text{if } (i,j) \text{ does not appear in the path } W. \end{cases}$$

Herein, d is defined as

$$d = \min\{d_v, d_r\}$$

with

$$\begin{aligned} d_v &= \min\{u_{ij} - x_{ij} \mid (i,j) \text{ is an f-edge in the path } W\}, \\ d_r &= \min\{x_{ij} \mid (i,j) \text{ is a b-edge in the path } W\}. \end{aligned}$$

(3) If $d = \infty$, STOP: the problem is unbounded.

(4) Goto (1) with x replaced by $\bar{x}$.

It holds:

- As the path W is augmenting it holds $d > 0$.
- $\bar{x}$ is a feasible flow because the conservation equations still hold at inner nodes and d is chosen such that $0 \le x_{ij} \le x_{ij} + d \le u_{ij}$ (f-edges) and $u_{ij} \ge x_{ij} \ge x_{ij} - d \ge 0$ (b-edges) hold.
- Let $\bar{v}$ be the volume of $\bar{x}$ and v the volume of x. Then $\bar{v} = v + d > v$, because

$$\bar{v} = \sum_{(Q,j)\in E} \bar{x}_{Qj} = (x_{Qv_1} + d) + \sum_{(Q,j)\in E, j\neq v_1} x_{Qj} = v + d \overset{d>0}{>} v.$$

It remains to clarify how an augmenting path can be constructed. Ford and Fulkerson developed a labeling algorithm which divides the nodes in 'labeled' and 'unlabeled'. Let the set C denote the set of labeled nodes. The labeled nodes are once more divided into 'explored' and 'not explored'.
Exploring a labeled node $i \in C$ means the following:

- Consider every edge (i, j). If the conditions $x_{ij} < u_{ij}$ and $j \notin C$ are satisfied, add node j to C.
- Consider every edge (j, i). If the conditions $x_{ji} > 0$ and $j \notin C$ are satisfied, add node j to C.

The following algorithm constructs an augmenting path needed in step (1) of the Algorithm of Ford and Fulkerson.

Algorithm 5.4.4 (Labeling Algorithm of Ford and Fulkerson)

(0) Label the source Q and set $C = \{Q\}$. The remaining nodes are unlabeled.

(1) If all labeled nodes have been explored, STOP. Otherwise choose a labeled node $i \in C$ which is not explored.

(2) Explore i. If the sink S has been labeled, STOP. Otherwise go to step (1).

The labeling algorithm stops if either an augmenting path is found (the sink is labeled) or no useable path from the source to the sink exists. In the latter case the set C of all labeled nodes defines a cut, because $Q \in C$ and $S \notin C$.
The cut C by construction has the following properties:

$$\begin{aligned} x_{jk} &= u_{jk}, && \text{for every edge } (j,k) \text{ with } j \in C,\ k \notin C, \\ x_{ij} &= 0, && \text{for every edge } (i,j) \text{ with } i \notin C,\ j \in C. \end{aligned}$$

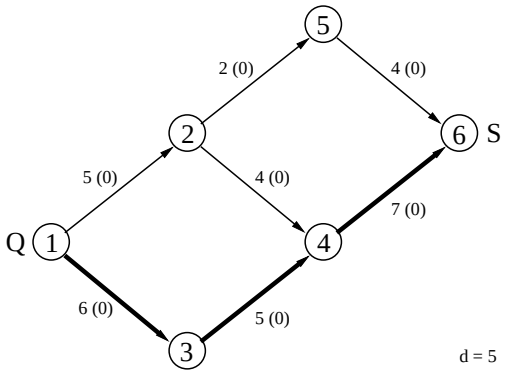
Particularly, C is a cut of minimal capacity, because with (5.5) we find

$$v = \sum_{j\in C, k\notin C} \underbrace{x_{jk}}_{=u_{jk}} - \sum_{i\notin C, j\in C} \underbrace{x_{ij}}_{=0} = \sum_{j\in C, k\notin C} u_{jk}.$$

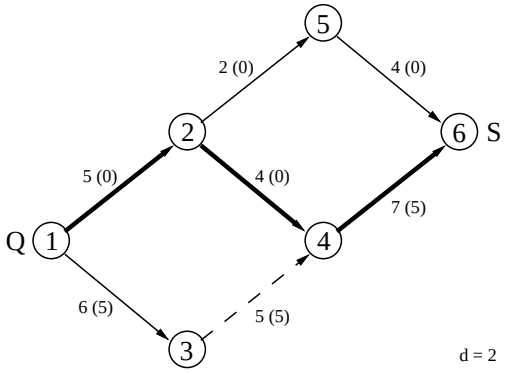
It follows that x is a flow with maximum volume v, because v is always bounded by the capacity of a cut.

Example 5.4.5 (Passenger Scheduling)

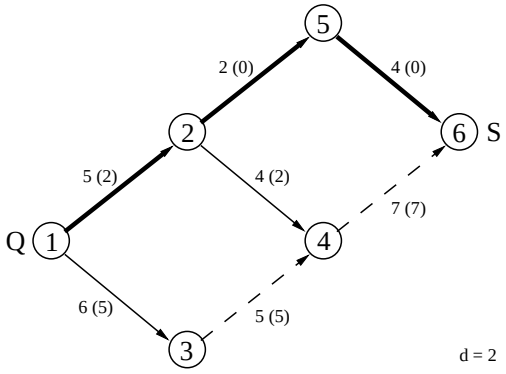
As many passengers as possible shall be transported from Q to S. As direct flights are not available anymore, flights via alternative cities have to be considered. For the respective connections only a limited number of free seats are available.



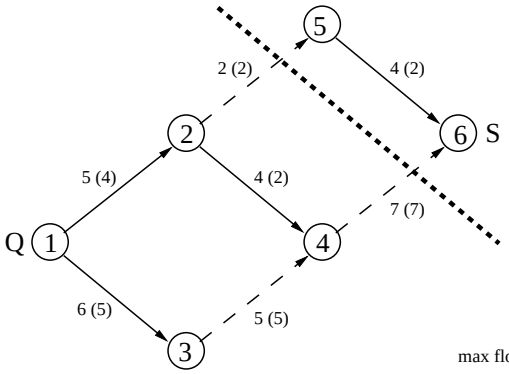
d = 5



d = 2



d = 2



max flow = 4+5=9

min cut = 2+7=9

C={1,2,3,4} defines a cut of minimal capacity. It is the result of the labeling procedure.

5.4.1 Finiteness and Complexity

So far, we haven't specified in which order the labeled nodes should be investigated. This flaw may lead to cycles in the algorithm of Ford and Fulkerson and in this case the algorithm does not terminate. Fortunately, there are certain conditions that guarantee finite termination.

Theorem 5.4.6

Under the assumptions

- *Every finite u_{ij} is a positive integer.*
- *At least one cut C possesses a finite capacity M.*
- *The flow, which is used to initialise the algorithm, is integral.*

the algorithm of Ford and Fulkerson terminates after at most $M+1$ steps.

Proof: Let the initial flow be integral. Then in each step an integral flow is generated because $d > 0$ is integral. In particular, the volume of the flow is increased by at least 1 in each iteration. The flow at the beginning of iteration k has at least volume $k-1$. On the other hand, the volume of every feasible flow is bounded by the finite capacity M of the cut. Hence, the k-th iteration takes place only if $k-1 \leq M$, i.e. the algorithm terminates after at most $M+1$ iterations. □

Remark 5.4.7

- *The algorithm may fail if at least one of the above conditions fails to hold.*
- *A variant of the algorithm of Ford and Fulkerson exists which terminates even for infinite and/or non-integral capacity bounds. This version considers the labeled nodes according to the first-labeled, first scanned principle, compare Edmonds and Karp (1972). This version determines a maximum flow in at most $nm/2$ iterations and has the total complexity $\mathcal{O}(m^2n)$, see Korte and Vygen [5], Th. 8.14, Cor. 8.15, page 173.*

Chapter 6

Dynamic Programming

Many combinatorial optimisation problems can be formulated as so-called dynamic optimisation problems, among them are

- inventory problems
- knapsack problem
- assignment problem
- reliability problem
- DNA sequence alignment
- machine scheduling problems
- ...

6.1 What is a Dynamic Optimisation Problem?

The dynamical behavior of the state of many technical, economical, and biological problems can be described by (discrete) dynamic equations. For instance we might be interested in the development of the population size of a specific species during a certain time period, or we want to describe the dynamical behavior of chemical processes or mechanical systems or the development of the profit of a company during the next five years, say.
Usually, the dynamical behavior of a given system can be influenced by the choice of certain control variables. For instance, the breeding of rabbits can be influenced by the incorporation of deseases or natural predators. A car can be controlled by the steering wheel, the accelerator pedal, and the brakes. A chemical process can be controlled, e.g., by increasing or decreasing the temperature. The profit of a company is influenced, e.g., by the prices of its products or the number of employees.
Very often, the state variables and/or the control variables cannot assume any value, but are subject to certain restrictions. These restrictions may result from certain safety regulations or physical limitations, e.g. the temperature in a nuclear reactor has to be lower than a specific threshold or the altitude of an airplane should be larger than ground level or the steering angle of a car is limited by a maximum steering angle.

In addition, we are particularly interested in those state and control variables which fulfill all restrictions and furthermore minimize or maximize a given objective function. For example, the objective of a company is to maximize the profit or to minimize operational costs.

Summarizing, we have the following ingredients for a discrete dynamic optimization problem, cf. Figures 6.1, 6.2:

- The state variables $y(t_j)$ describe the state of a process at certain time points t_j.
- The control variables $u(t_j)$ allow to influence the dynamical behavior of the state variables at time point t_j.
- The discrete dynamic equation $y(t_{j+1}) = g(t_j, y(t_j), u(t_j))$ describes the transition of the state from time point t_j to the subsequent time point t_{j+1} dependent on the current state $y(t_j)$, the control variable $u(t_j)$, and the time point t_j.
- The objective function has to be minimized (or maximized).
- The restrictions on the state and/or control variables have to be fulfilled.

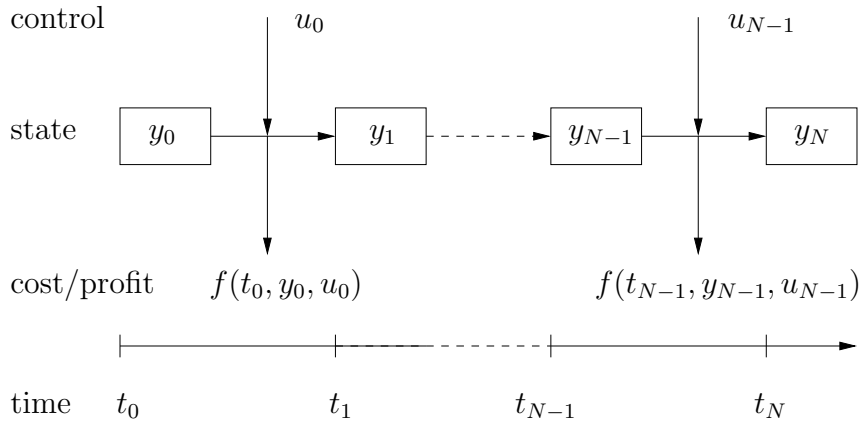


Figure 6.1: Schematic representation of a discrete dynamic optimization problem.

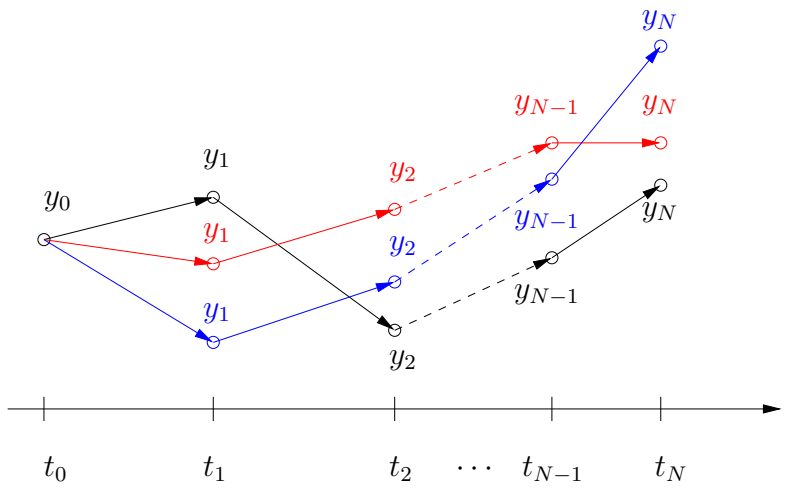


Figure 6.2: Examples of discrete trajectories for different control functions.

6.2 Examples and Applications

We will discuss several areas of applications leading to discrete dynamic optimization problems.

6.2.1 Inventory Management

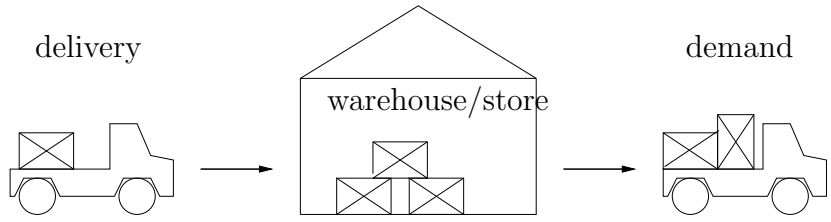


Figure 6.3: Inventory Management

A company has to determine a minimum cost inventory plan for a fixed number of time periods $t_0 < t_1 < \ldots < t_N$. A product is stored during these time periods $t_0 < t_1 < \ldots < t_N$ in a store. Let $u_j \geq 0$ be a delivery at time point t_j, $r_j \geq 0$ the demand in $[t_j, t_{j+1})$, and y_j the amount of stored products at time point t_j (just before delivery). Then the balance equations

$$y_{j+1} = y_j + u_j - r_j, \qquad j = 0, 1, \ldots, N-1$$

hold. In addition, we postulate that the demand can be satisfied always, that is $y_{j+1} \geq 0$ for all $j = 0, 1, \ldots, N-1$. Without loss of generality, at the beginning and at the end of the time period the stock level is zero, that is $y_0 = y_N = 0$. The delivery costs at time point t_j are modelled by

$$B(u_j) = \begin{cases} K + cu_j, & \text{if } u_j > 0, \\ 0, & \text{if } u_j = 0, \end{cases}$$

where K denotes the fixed costs and c is the cost per unit. The inventory costs become effective at the end of each time period and are given by hy_{j+1}, $j = 0, 1, \ldots, N-1$. Thus, the total costs are

$$\sum_{j=0}^{N-1} (K\delta(u_j) + cu_j + hy_{j+1}),$$

where

$$\delta(u_j) = \begin{cases} 1, & \text{if } u_j > 0, \\ 0, & \text{if } u_j = 0, \end{cases}$$

It holds

$$\begin{aligned} \sum_{j=0}^{N-1} u_j &= \sum_{j=0}^{N-1} (y_{j+1} - y_j + r_j) = (y_N - y_0) + \sum_{j=0}^{N-1} r_j = \sum_{j=0}^{N-1} r_j, \\ \sum_{j=0}^{N-1} y_{j+1} &= \sum_{j=0}^{N-1} (y_j + u_j - r_j) = \sum_{j=0}^{N-1} y_j + \sum_{j=0}^{N-1} u_j - \sum_{j=0}^{N-1} r_j = \sum_{j=0}^{N-1} y_j. \end{aligned}$$

Taking these relations into account, we can formulate the following inventory problem:

$$\begin{aligned} \text{Minimize} \quad & \sum_{j=1}^{N} K\delta(u_j) + hy_j \\ \text{subject to} \quad & y_{j+1} = y_j + u_j - r_j, \quad j = 0, \ldots, N-1, \\ & y_0 = y_N = 0, \\ & y_j \geq 0, \qquad\qquad j = 0, \ldots, N, \\ & u_j \geq 0, \qquad\qquad j = 0, \ldots, N. \end{aligned}$$

Example 6.2.1

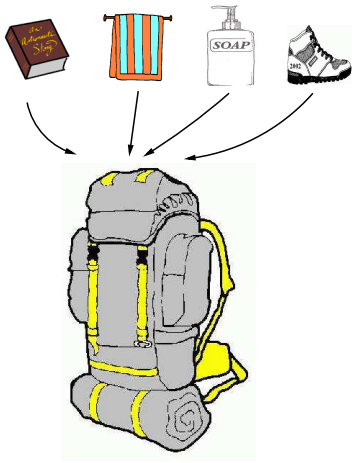
There are three different methods at different costs available in a production line within one time period. The costs are given by the following table:

Method	*I*	*II*	*III*
stock	0	15	30
costs	0	500	800

The fixed costs for a positive production volume amounts to 300 *dollar for each time period, the inventory costs are* 15 *dollar for each item and time period. The demand for*

each time period is given by 25 *items. The task is to determine an optimal production plan for the next three time periods, if the initial inventory is* 35 *items and the final inventory is restricted to* 20 *items. It is assumed that the inventory costs are valid at the beginning of each time period.*

6.2.2 Knapsack Problem



There is one knapsack together with N items. Item i has weight a_j and value c_j for $j = 1, \dots, N$. The task is to create a knapsack with maximal value under the restriction that the maximal weight is less or equal A. This leads to the optimization problem:

Maximize

$$\sum_{j=1}^{N} c_j u_j$$

s. t.

$$\sum_{j=1}^{N} a_j u_j \le A,\ u_j \in \{0,1\},\ j = 1, \dots, N,$$

where

$$u_j = \begin{cases} 1, & \text{item } j \text{ is put into the knapsack,} \\ 0, & \text{item } j \text{ is not put into the knapsack.} \end{cases}$$

This is a linear optimization problem with integer variables. It can be transformed into an equivalent discrete dynamic optimization problem. Let y_j denote the remaining weight after items $i = 1, \dots, j-1$ have been included (or not). Then, the discrete dynamic

optimization problem arises:

$$\begin{array}{lll} \textit{Maximize} & \sum_{j=1}^{N} c_j u_j & \\ \textit{subject to} & y_{j+1} = y_j - a_j u_j, & j = 1, \ldots, N, \\ & y_1 = A, & \\ & 0 \le y_j, & j = 1, \ldots, N, \\ & u_j \begin{cases} \in \{0,1\}, & \text{if } y_j \ge a_j, \\ = 0, & \text{if } y_j < a_j, \end{cases} & j = 1, \ldots, N. \end{array}$$

y_{N+1} is the remaining space in the knapsack.

Example 6.2.2

A manager has to choose co-workers for a project. There are four eligible co-workers to choose from. Each of them has a number assigned that indicates the capability of the respective co-worker. The respective numbers are $3, 5, 2$, and 4. The costs for the co-workers are $30, 50, 20$, and 40 thousand dollar, respectively. The manager has a maximal amount of 90 thousand dollars available for the personal expenditure. Which co-workers should the manager choose for the project?

The problem is a knapsack problem. The co-workers 1-4 correspond to $N = 4$ items, which can be put into the knapsack. The personal expenditure $a_j, j = 1, \ldots, N$ corresponds to the weight of the respective item, whereas the capability of the respective co-worker corresponds to the value c_j of item j. The maximal amount A is the maximal weight of the knapsack. The task is to create a knapsack with maximum value subject to the weight restriction.

6.2.3 Assignment Problems

A number A of resources has to be assigned to N projects. Assigning u_j resources to project j results in the profit $f_j(u_j)$. The goal is to maximize the total profit $\sum_{j=1}^{N} f_j(u_j)$. This leads to the optimization problem:

Maximize

$$\sum_{j=1}^{N} f_j(u_j)$$

s. t.

$$\sum_{j=1}^{N} u_j \le A,\ u_j \ge 0,\ j = 1, \ldots, N.$$

This problem can be transformed into a discrete dynamic optimization problem. For that purpose, we introduce the state y_j, which denotes the remaining resources after

assigning resources to the projects $i = 1, \dots, j-1$. Then the above optimization problem is equivalent with

$$\begin{aligned} \text{Maximize} \quad & \sum_{j=1}^{N} f_j(u_j) \\ \text{subject to} \quad & y_{j+1} = y_j - u_j, && j = 1, \dots, N, \\ & u_j \in U_j(y_j) = \{0, 1, \dots, y_j\}, && j = 1, \dots, N, \\ & y_1 = A. \end{aligned}$$

Example 6.2.3

A company has to assign four sales representatives to four sales regions A,B,C,D. The achievable sales volume depends on the number of assigned representatives according to the following table.

Region/Number	0	1	2	3	4
A	0	25	48	81	90
B	0	35	48	53	65
C	0	41	60	75	92
D	0	52	70	85	95

The company intends to maximize the total sales volume. How does the optimal assignment look like?

6.2.4 Reliability Problems

Let us assume that a computer works if and only if three components A,B and C work properly. In order to increase the reliability of the computer system it is possible to add certain emergency systems to each component. It costs 100 dollars to add such an emergency system to the first component, 300 dollars for the second component, and 200 dollars for the third component. Furthermore, it is assumed, that at most two emergency systems for each component can be added. The probability, that a component works properly, depends on the number of emergency systems added to the component according to the following table.

Number/System	A	B	C
0	0.85	0.60	0.70
1	0.90	0.85	0.90
2	0.95	0.95	0.98

We are looking for a configuration, that maximizes the reliability of the computer subject to the additional restriction that at most 600 dollars can be spent for additional emergency systems.

Formulation as a dynamic optimization problem:
The components A,B,C are denoted by 1,2,3. Let y_j be the amount remaining after emergency systems have been added to the components $1, \ldots, j-1$.
Let u_j be the number of emergency systems for component j subject to the restriction $u_j \in U = \{0, 1, 2\}$.
Let $p_j(u_j)$ be the probability that component j works properly, if u_j emergency systems have been added. c_j are the costs to add an emergency system to component j, that is $c_1 = 100, c_2 = 300, c_3 = 200$.
Thus, we obtain

$$\begin{aligned}
&\textit{Maximize} \quad \prod_{j=1}^{3} p_j(u_j) \\
&\textit{subject to} \\
&\qquad y_{j+1} = y_j - c_j \cdot u_j, \quad j = 1, 2, 3 \ , \\
&\qquad y_1 = 600 \ , \\
&\qquad u_j \in U_j(y_j) = \begin{cases} \{0\}, & \text{if } y_j < c_j \ , \\ \{0, 1\}, & \text{if } c_j \le y_j < 2c_j \ , \\ \{0, 1, 2\}, & \text{if } 2c_j \le y_j \ , \end{cases} \\
&\qquad y_{j+1} \in \{0, 100, 200, 300, 400, 500, 600\} \ .
\end{aligned}$$

Unfortunately, the objective function is a product and not a sum. But, we can simply transform the above problem by applying the logarithm to the objective function:

$$\begin{aligned}
&\textit{Maximize} \quad \sum_{j=1}^{3} \log p_j(u_j) \\
&\textit{subject to} \\
&\qquad y_{j+1} = y_j - c_j \cdot u_j, \quad j = 1, 2, 3 \ , \\
&\qquad y_1 = 600 \ , \\
&\qquad u_j \in U_j(y_j) = \begin{cases} \{0\}, & \text{if } y_j < c_j \ , \\ \{0, 1\}, & \text{if } c_j \le y_j < 2c_j \ , \\ \{0, 1, 2\}, & \text{if } 2c_j \le y_j \ , \end{cases} \\
&\qquad y_{j+1} \in \{0, 100, 200, 300, 400, 500, 600\} \ .
\end{aligned}$$

6.3 Mathematical Formulation of Discrete Dynamic Optimization Problems

Let

$$\mathbb{G} := \{t_j \mid j = 0, 1, \ldots, N\}$$

denote a grid with $N + 1$ fixed (time) points

$$t_0 < t_1 < \ldots < t_N.$$

The task is to determine a state grid function

$$y : \mathbb{G} \to \mathbb{R}^n, \qquad t_j \mapsto y(t_j),$$

and a control grid function

$$u : \mathbb{G} \to \mathbb{R}^m, \qquad t_j \mapsto u(t_j),$$

such that the objective function

$$\sum_{j=0}^{N} f(t_j, y(t_j), u(t_j)),$$

where

$$f : \mathbb{G} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$$

is minimized subject to the dynamic equations

$$y(t_{j+1}) = g(t_j, y(t_j), u(t_j)), \qquad j = 0, 1, \ldots, N-1,$$

where

$$g : \mathbb{G} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n.$$

In addition, the state constraints

$$y(t_j) \in Y(t_j), \qquad j = 0, 1, \ldots, N,$$

with nonempty sets $Y(t_j) \subseteq \mathbb{R}^n$, and the control constraints

$$u(t_j) \in U(t_j, y(t_j)), \qquad j = 0, 1, \ldots, N,$$

with nonempty sets $U(t_j, y) \subseteq \mathbb{R}^m$ for $y(t_j) \in Y(t_j)$ and $j = 0, 1, \ldots, N$ have to be fulfilled.
Summarizing we consider the

Discrete Dynamic Optimization Problem (DOP)

$$\begin{aligned}
&\text{Minimize} && \sum_{j=0}^{N} f(t_j, y(t_j), u(t_j)) \\
&\text{w.r.t.} && y \in \{y \mid y : \mathbb{G} \to \mathbb{R}^n\}, \; u \in \{u \mid u : \mathbb{G} \to \mathbb{R}^m\} \\
&\text{subject to} && y(t_{j+1}) = g(t_j, y(t_j), u(t_j)), \quad j = 0, 1, \ldots, N-1, \\
& && y(t_j) \in Y(t_j), \qquad j = 0, 1, \ldots, N, \\
& && u(t_j) \in U(t_j, y(t_j)), \qquad j = 0, 1, \ldots, N.
\end{aligned}$$

Remark 6.3.1

- *Very often, the sets $Y(t_j)$ are expressed in terms of inequality and equality constraints, that is*

$$Y(t_j) = \{y \in \mathbb{R}^n \mid r(t_j, y) \leq 0,\ h(t_j, y) = 0\}.$$

Similarly, the sets $U(t_j, y)$ often are given by

$$U(t_j, y) = \{u \in \mathbb{R}^m \mid \tilde{r}(t_j, y, u) \leq 0,\ \tilde{h}(t_j, y, u) = 0\}.$$

More specifically, the control may be restricted by box constraints only:

$$u(t_j) \in \{v = (v_1, \ldots, v_m)^\top \in \mathbb{R}^m \mid a_j \leq v_j \leq b_j,\ j = 1, \ldots, m\}.$$

6.4 Dynamic Programming Method and Bellman's Principle

The upcoming dynamic programming method is based on the Bellman's optimality principle for (DOP). The dynamic programming method is one of the earliest methods to solve discrete dynamic optimization problems.
For a more detailed discussion we refer to the monographs [1], [2], [4], [6]. Many applications and examples can be found in [8].

6.5 Bellman's Optimality Principle

Let $t_k \in \{t_0, t_1, \ldots, t_N\}$ be a fixed time point, $\mathbb{G}_k := \{t_j \mid j = k, k+1, \ldots, N\}$, and $\hat{y} \in Y(t_k)$ an admissible state. Consider the

Discrete dynamic optimization problem $(P(t_k, \hat{y}))$

$$\begin{aligned}
&\text{Minimize} && \sum_{j=k}^{N} f(t_j, y(t_j), u(t_j)) \\
&\text{w.r.t.} && y \in \{y \mid y : \mathbb{G}_k \to \mathbb{R}^n\},\ u \in \{u \mid u : \mathbb{G}_k \to \mathbb{R}^m\} \\
&\text{subject to} && y(t_{j+1}) = g(t_j, y(t_j), u(t_j)), \quad j = k, 1, \ldots, N-1, \\
&&& y(t_k) = \hat{y}, \\
&&& y(t_j) \in Y(t_j), \qquad j = k, k+1, \ldots, N, \\
&&& u(t_j) \in U(t_j, y(t_j)), \qquad j = k, k+1, \ldots, N.
\end{aligned}$$

Definition 6.5.1 **(Optimal Value Function)**
Let $t_k \in \mathbb{G}$. For $\hat{y} \in Y(t_k)$ let $V(t_k, \hat{y})$ denote the optimal value of the problem $(P(t_k, \hat{y}))$.

For $\hat{y} \notin Y(t_k)$ we set $V(t_k, \hat{y}) = \infty$. The function $V : \mathbb{G} \times \mathbb{R}^n \to \mathbb{R}$, $(t_k, \hat{y}) \mapsto V(t_k, \hat{y})$ is called **optimal value function**.

Theorem 6.5.2 **(Bellman's Optimality Principle)**
Let $\hat{y}(\cdot)$ and $\hat{u}(\cdot)$ be an optimal solution of (DOP). Then $\hat{y}|_{\mathbb{G}_k}$ and $\hat{u}|_{\mathbb{G}_k}$ is an optimal solution of $(P(t_k, \hat{y}(t_k)))$.

Proof: Assume, that $\hat{y}|_{\mathbb{G}_k}$ and $\hat{u}|_{\mathbb{G}_k}$ are not optimal for $(P(t_k, \hat{y}(t_k)))$. Then there exist feasible trajectories $\tilde{y} : \mathbb{G}_k \to \mathbb{R}^n$ and $\tilde{u} : \mathbb{G}_k \to \mathbb{R}^m$ for $(P(t_k, \hat{y}(t_k)))$ with

$$\sum_{j=k}^{N} f(t_j, \tilde{y}(t_j), \tilde{u}(t_j)) < \sum_{j=k}^{N} f(t_j, \hat{y}(t_j), \hat{u}(t_j))$$

and $\tilde{y}(t_k) = \hat{y}(t_k)$. Hence, the trajectories $y : \mathbb{G} \to \mathbb{R}^n$ and $u : \mathbb{G} \to \mathbb{R}^m$ with

$$y(t_j) := \begin{cases} \hat{y}(t_j), & \text{for } j = 0, 1, \ldots, k-1, \\ \tilde{y}(t_j), & \text{for } j = k, k+1, \ldots, N, \end{cases}$$

$$u(t_j) := \begin{cases} \hat{u}(t_j), & \text{for } j = 0, 1, \ldots, k-1, \\ \tilde{u}(t_j), & \text{for } j = k, k+1, \ldots, N, \end{cases}$$

are feasible for (DOP) and satisfy

$$\sum_{j=0}^{k-1} f(t_j, \hat{y}(t_j), \hat{u}(t_j)) + \sum_{j=k}^{N} f(t_j, \tilde{y}(t_j), \tilde{u}(t_j)) < \sum_{j=0}^{N} f(t_j, \hat{y}(t_j), \hat{u}(t_j)).$$

This contradicts the optimality of $\hat{y}(\cdot)$ and $\hat{u}(\cdot)$. □

The optimality principle states: The decisions in the periods $k, k+1, \ldots, N$ of the Problem (DOP) for a given state y_k are independent of the decisions in the periods $t_0, t_1, \ldots, t_{k-1}$, compare Figure 6.4.

Remark 6.5.3
For the validity of the optimality principle it is essential that the discrete dynamic optimization problem can be divided into stages, e.g. the state y at t_{j+1} only depends on the values of y and u at the previous stage t_j and not on the respective values at, e.g., t_0 and t_N. Similarly, the objective function is separable and the constraints only restrict y and u at t_j. This allows to apply a stagewise optimization procedure.

6.6 Dynamic Programming Method

The optimality principle allows to derive a recursion for the optimal value function. In the sequel, we use the convention $f(t_j, y, u) = \infty$, if $y \notin Y(t_j)$.

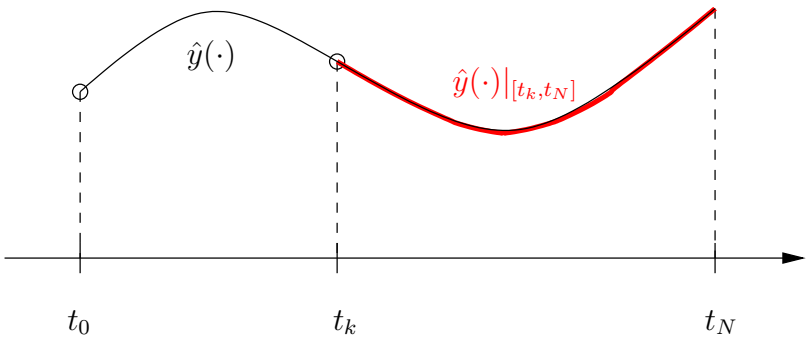


Figure 6.4: Bellman's optimality principle: Rest trajectories of optimal trajectories remain optimal.

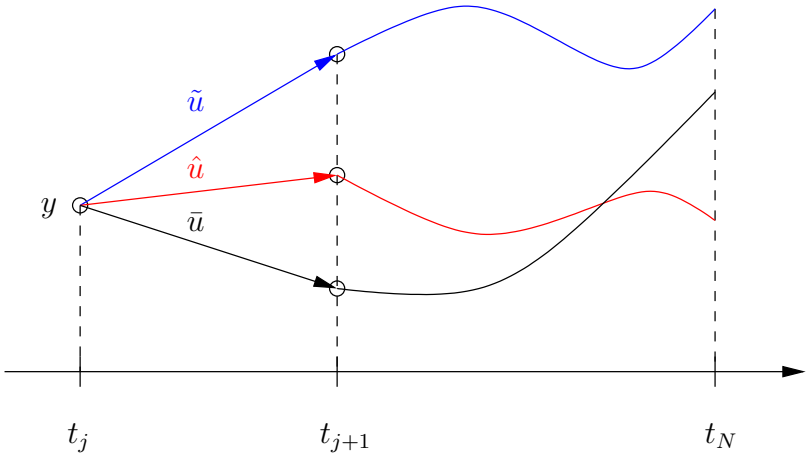


Figure 6.5: Bellman's dynamic programming method: Recursion of the optimal value function.

We exploit the fact, that the optimal value for $(P(t_N, y))$ is given by

$$V(t_N, y) = \min_{u \in U(t_N, y)} f(t_N, y, u). \tag{6.1}$$

Suppose now, that we already know the optimal value function $V(t_{j+1}, y)$ for any $y \in \mathbb{R}^n$. From the optimality principle we obtain

$$V(t_j, y) = \min_{u \in U(t_j, y)} \{f(t_j, y, u) + V(t_{j+1}, g(t_j, y, u)\}, \quad j = 0, 1, \ldots, N-1. \tag{6.2}$$

Equations (6.1) and (6.2) enable us to compute the optimal value function backward in time starting at t_N.
The optimal initial state $\hat{y}(t_0)$ of (DOP) is given by

$$\hat{y}(t_0) = arg \min_{y \in Y(t_0)} V(t_0, y). \tag{6.3}$$

Equations (6.1)-(6.3) form the basis for

Algorithm 6.6.1 **(Bellman's Dynamic Programming Method I)**

(i) **Backward computation**

1. *Let $V(t_N, y)$ be given by (6.1).*
2. *For $j = N-1, \dots, 0$: Calculate $V(t_j, y)$ as in (6.2).*

(ii) **Forward computation**

1. *Let $\hat{y}(t_0)$ be given by (6.3).*
2. *For $j = 0, 1, \dots, N-1$: Determine*

$$\hat{u}(t_j) = \arg \min_{u \in U(t_j, \hat{y}(t_j))} \{f(t_j, \hat{y}(t_j), u) + V(t_{j+1}, g(t_j, \hat{y}(t_j), u))\}$$

and set $\hat{y}(t_{j+1}) = g(t_j, \hat{y}(t_j), \hat{u}(t_j))$.
3. *Determine $\hat{u}(t_N) = \arg\min_{u \in U(t_N, \hat{y}(t_N))} f(t_N, \hat{y}(t_N), u)$.*

Algorithm 6.6.2 **(Bellman's Dynamic Programming Method II)**

(i) **Backward computation**

1. *Let $V(t_N, y)$ be given by (6.1) and $u^*(t_N, y)$ the corresponding optimal control.*
2. *For $j = N-1, \dots, 0$: Calculate $V(t_j, y)$ as in (6.2).*
   Let $u^*(t_j, y)$ denote the optimal control at t_j for y (feedback control).

(ii) **Forward computation**

1. *Let $\hat{y}(t_0)$ be given by (6.3).*
2. *For $j = 0, 1, \dots, N-1$: Determine $\hat{u}(t_j) = u^*(t_j, \hat{y}(t_j))$ and*

$$\hat{y}(t_{j+1}) = g(t_j, \hat{y}(t_j), \hat{u}(t_j)).$$

3. *Determine $\hat{u}(t_N) = u^*(t_N, \hat{y}(t_N))$.*

Remark 6.6.3

Both versions of Bellman's dynamic programming method yield an optimal solution of the discrete dynamic programming problem (DOP). Version II is preferable for hand calculations because it provides an optimal feedback control u^* as a function of time and state. Version I is more convenient for computer implementations, since it does not require to

store the feedback control u^* for each t_j and y and thus saves memory space. It only computes the optimal trajectories for y and u as functions of time.

Example 6.6.4

Solve the discrete dynamic optimization problem

$$\begin{aligned} \textit{Minimize} \quad & -\sum_{j=0}^{N-1} c(1-u_j)y(j) \\ \textit{s.t.} \quad & y(j+1) = y(j)(0.9+0.6u(j)), \quad j=0,1,\dots,N-1 \\ & y(0) = k>0, \\ & 0 \le u(j) \le 1, \quad j=0,1,\dots,N-1 \end{aligned}$$

for $k>0, c>0, b=0.6$ and $N=5$ by the dynamic programming method. Since $k>0$ and $u(j)\ge 0$ it holds $y(j)>0$ for all j. In the sequel we use the abbreviation $y_j := y(j)$. The recursive equations (6.1) and (6.2) for $N=5$ are given by $V(5,y_5)=0$ and

$$V(j,y_j) = \min_{0\le u_j\le 1}\{-cy_j(1-u_j)+V(j+1,y_j(0.9+0.6u_j))\}, \quad 0\le j\le N-1.$$

Evaluation of the recursion and observation of $c>0, y_j>0$ yields

$$\begin{aligned} V(4,y_4) &= \min_{0\le u_4\le 1}\{-cy_4(1-u_4)+\underbrace{V(5,y_4(0.9+0.6u_4)}_{=0}\} = -cy_4, \qquad \hat{u}_4=0, \\ V(3,y_3) &= \min_{0\le u_3\le 1}\{-cy_3(1-u_3)+V(4,(y_3(0.9+0.6u_3))\} \\ &= \min_{0\le u_3\le 1}\{-cy_3(1-u_3)-cy_3(0.9+0.6u_3)\} \\ &= cy_3 \min_{0\le u_3\le 1}\{-1.9+0.4u_3\} = -1.9cy_3, \qquad \hat{u}_3=0, \\ V(2,y_2) &= \min_{0\le u_2\le 1}\{-cy_2(1-u_2)+V(3,y_2(0.9+0.6u_2)\} \\ &= \min_{0\le u_2\le 1}\{-cy_2(1-u_2)-1.9cy_2(0.9+0.6u_2)\} \\ &= cy_2 \min_{0\le u_2\le 1}\{-2.71-0.14u_2\} = -2.85cy_2, \qquad \hat{u}_2=1, \\ V(1,y_1) &= \min_{0\le u_1\le 1}\{-cy_1(1-u_1)+V(2,y_1(0.9+0.6u_1)\} \\ &= \min_{0\le u_1\le 1}\{-cy_1(1-u_1)-2.85cy_1(0.9+0.6u_1)\} \\ &= cy_1 \min_{0\le u_1\le 1}\{-3.565-0.71u_1\} = -4.275cy_1, \qquad \hat{u}_1=1, \\ V(0,y_0) &= \min_{0\le u_0\le 1}\{-cy_0(1-u_0)+V(1,y_0(0.9+0.6u_0)\} \\ &= \min_{0\le u_0\le 1}\{-cy_0(1-u_0)-4.275cy_0(0.9+0.6u_0)\} \\ &= cy_0 \min_{0\le u_0\le 1}\{-4.8475-1.565u_0\} = -6.4125cy_0, \qquad \hat{u}_0=1, \end{aligned}$$

Hence, the optimal control is $\hat{u}_0=\hat{u}_1=\hat{u}_2=1, \hat{u}_3=\hat{u}_4=0$. Forward evaluation leads

to $\hat{y}_0 = k, \hat{y}_1 = 1.5 \cdot k, \hat{y}_2 = 2.25 \cdot k, \hat{y}_3 = 3.375 \cdot k, \hat{y}_4 = 3.0375 \cdot k, \hat{y}_5 = 2.73375 \cdot k$. *The optimal objective value is* $-c\hat{y}_3 - c\hat{y}_4 = -c(\hat{y}_3 + \hat{y}_4) = -6.4125 \cdot c \cdot k$.

6.7 Implementation

We discuss an implementable algorithm for the special discrete dynamic otimization problem

$$\begin{aligned} \text{Minimize} \quad & \sum_{j=0}^{N} f(t_j, y_j, u_j) \\ \text{s.t.} \quad & \begin{aligned} y_{j+1} &= g(t_j, y_j, u_j), && j = 0, \ldots, N-1, \\ y_0 &= y_a, \\ y_j &\in [y_l, y_u], && j = 1, \ldots, N, \\ u_j &\in [u_l(t_j, y_j), u_u(t_j, y_j)], && j = 0, \ldots, N. \end{aligned} \end{aligned}$$

The interval $[y_l, y_u]$ is divided into M equidistant sections of length $h = (y_u - y_l)/M$:

$$Y = \{y_l + i \cdot h \mid i = 0, \ldots, M\}.$$

Y denotes the feasible region for the state variables. Similarly, the control region $[u_l(t_j, y_j), u_u(t_j, y_j)]$ is divided into M_j equidistant sections of length $h_j = (u_u(t_j, y_j) - u_l(t_j, y_j))/M_j$:

$$U(t_j, y_j) = \{u_l(t_j, y_j) + i \cdot h_j \mid i = 0, \ldots, M_j\}, \quad j = 0, \ldots, N.$$

$U(t_j, y_j)$ denotes the feasible region for the control variables in step j.

Algorithm 6.7.1 (Bellman's Dynamic Programming Method)

(i) **Backward computation**

1. *For all* $y \in Y$ *let*
$$V(t_N, y) = \min_{u \in U(t_N, y)} f(t_N, y, u).$$
2. *For* $j = N-1, \ldots, 0$*: For all* $y \in Y$ *determine*
$$V(t_j, y) = \min_{u \in U(t_j, y) g(t_j, y, u) \in [y_l, y_u]} \{f(t_j, y, u) + V(t_{j+1}, g(t_j, y, u))\}. \tag{6.4}$$

(ii) **Forward computation**

1. *Let* $\hat{y}_1 = y_a$.
2. *For* $j = 0, 1, \ldots, N-1$*: Determine*
$$\hat{u}_j = \arg \min_{u \in U(t_j, \hat{y}_j) g(t_j, \hat{y}_j, u) \in [y_l, y_u]} \{f(t_j, \hat{y}_j, u) + V(t_{j+1}, g(t_j, \hat{y}_j, u))\} \tag{6.5}$$
and set $\hat{y}_{j+1} = g(t_j, \hat{y}_j, \hat{u}_j)$.

3. *Determine* $\hat{u}_N = \arg\min_{u \in U(t_N, \hat{y}_N)} f(t_N, \hat{y}_N, u)$.

Remark 6.7.2

The evaluation of (6.4) and (6.5), respectively, requires the values $V(t_{j+1}, y_{j+1})$ *with* $y_{j+1} = g(t_j, y, u) \in [y_l, y_u]$. *It may happen, that* y_{j+1} *is not a grid point in* Y *such that* $\bar{y} := y_l + i \cdot h < y_{j+1} < y_l + (i+1) \cdot h = \bar{y} + h$ *holds for some index* i. *Then, the value of the optimal value function at* y_{j+1} *is determined by linear interpolation of the values* $V(t_{j+1}, \bar{y})$ *and* $V(t_{j+1}, \bar{y} + h)$*:*

$$V(t_{j+1}, y_{j+1}) \approx V(t_{j+1}, \bar{y}) + \frac{y_{j+1} - \bar{y}}{h} \left(V(t_{j+1}, \bar{y} + h) - V(t_{j+1}, \bar{y}) \right).$$

Example 6.7.3 (Test example)

Knapsack problem ($M = 1000$, $M_j = 1$, $j = 1, \ldots, N$)

No.	*Item*	*weight*	*value*
1.	*knapsack*	*1400 g*	*1.00*
2.	*tent*	*2600 g*	*0.88*
3.	*camping mat*	*1200 g*	*0.92*
4.	*sleeping bag*	*1500 g*	*0.94*
5.	*stove*	*1600 g*	*0.79*
6.	*4 soups*	*each 30 g*	*0.79*
7.	*bottle of water*	*1150 g*	*0.98*
8.	*clothes*	*800 g*	*0.71*
9.	*washbag*	*300 g*	*0.74*
10.	*towel*	*350 g*	*0.81*
11.	*handy*	*550 g*	*0.5*
12.	*wallet*	*500 g*	*0.99*
13.	*pencils*	*300 g*	*0.52*
14.	*map of trails*	*80 g*	*0.98*
15.	*city guide*	*200 g*	*0.58*
16.	*bar of chocolate*	*100 g*	*0.98*

The maximum weight is $A = 10$ $[kg]$.

Remark 6.7.4

The main drawback of Bellman's dynamic programming method is the so called 'curse of dimension'. As it can be seen from formula (6.4) the method requires to compute and to store the values $V(t_j, y)$ *for each* $j = N, N-1, \ldots, 0$ *and each* $y \in Y$. *Depending on the value* N *this can become a really huge number. In the worst case each discrete*

trajectorie emanating from y_a has to be considered. Nevertheless, for certain applications, e.g. assignment problems, knapsack problems, and inventory problems with integral data, the dynamic programming method performs quite well.

Chapter A

Software

Available software in the world wide web:

- SCILAB: A Free Scientific Software Package; http://www.scilab.org
- GNU OCTAVE: A high-level language, primarily intended for numerical computations; http://www.octave.org/octave.html
- GAMS: Guide to Available Mathematical Software; http://gams.nist.gov/
- NETLIB: collection of mathematical software, papers, and databases; http://www.netlib.org/
- Decision Tree for Optimization Software; http://plato.la.asu.edu/guide.html
- NEOS GUIDE: www-fp.mcs.anl.gov/otc/Guide
- COPS: Large-Scale Optimization Problems; http://www-unix.mcs.anl.gov/∼more/cops/
- GALIB: set of C++ genetic algorithm objects; http://lancet.mit.edu/ga/
- test problems: http://www.princeton.edu/∼rvdb/ampl/nlmodels/

Bibliography

[1] R. E. Bellman. *Dynamic Programming*. University Press, Princeton, New Jersey, 1957.

[2] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. University Press, Princeton, New Jersey, 1971.

[3] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977.

[4] I. M. Bomze and W. Grossmann. *Optimierung - Theorie und Algorithmen*. BI-Wissenschaftsverlag, Mannheim, 1993.

[5] B. Korte and J. Vygen. *Combinatorial Optimization – Theory and Algorithms*, volume 21 of *Algorithms and Combinatorics*. Springer Berlin Heidelberg, fourth edition edition, 2008.

[6] K. Neumann and M. Morlock. *Operations Research*. Carl Hanser Verlag, München Wien, 2002.

[7] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization–Algorithms and Complexity*. Dover Publications, 1998.

[8] W. L. Winston. *Operations Research: Applications and Algorithms*. Brooks/Cole–Thomson Learning, Belmont, 4 edition, 2004.