Summary Optimiz

Adrian Willi

January 31, 2023

Chapter 1

Introduction

1.1 Optimization

1.1.1 Quantitative vs. Qualitative

Quantitative analysis and optimization (only this is covered in this course) is based on quantifiable information and knowledge like measurable data and mathematical models. Qualitative analysis and optimization is based on non-quantifiable information and knowledge like "informal" facts and verbal descriptions of processes and procedures.

1.1.2 Continuous optimization

- Infinite number of solutions represented by continuous variables
- Graph of the objective function is arbitrary "landscape" (i.e. nonlinear)
- Main part of the theory: **local optimization**
- Methods mainly based on differentiability information (1st and 2nd derivatives)
- Very difficult in case of non-differentiable or even non-continuous functions
- Constrained optimization (with constraints) more difficult than optimization without constraints
- **Global optimization**: mostly "stochastic search"

1.1.3 Discrete optimization

- Number of solutions is finite (or countable): represented by integer variables
- There exists a trivial and finite algorithm: Enumeration
- Major part can be formulated as discrete problems
- Goal: to solve (exactly or approximately) a real life problem efficiently (i.e. in "reasonable" time)
- Question: What is a good (i.e. efficient) algorithm for a given problem?
- Development of Complexity Theory (see runtime of algorithms)

1.1.4 Runtime of Algorithms

Obviously the size of the problem instance matters. But instead of measuring time, we count the number of elementary operations to be executed on a computer.

Worst-case analysis: Estimate number of elementary operations needed to solve the most difficult instance of a given size n. We only want to estimate the **order of magnitude (order of growth)** of runtime function f(n), i.e. approximative behaviour when n becomes large. Also known as Big-O notation.
Example: $7n^3 + 4n^2 + 3$ is $O(n^3)$

Polynomial Algorithms, "Good" and "Bad"

An algorithm is said to be polynomial if its runtime is $O(n^k)$ for some fixed positive number k.

"Good" vs. "Bad" Algorithms:

- Good: Polynomial runtime (considered as "effective", "tractable")
- Bad: Non-polynomial runtime (i.e. exponential, considered "intractable")

"Good" vs. "Bad" Problems:

- Good: Polynomial algorithm is known
- Bad: No polynomial algorithm is known (i.e. only exponential alg. known)

Chapter 2

Mathematical Models

2.1 Descriptive Models

Also called Evaluation models. Question: **"What if?"**. Given an alternative, calculate the resulting consequences.

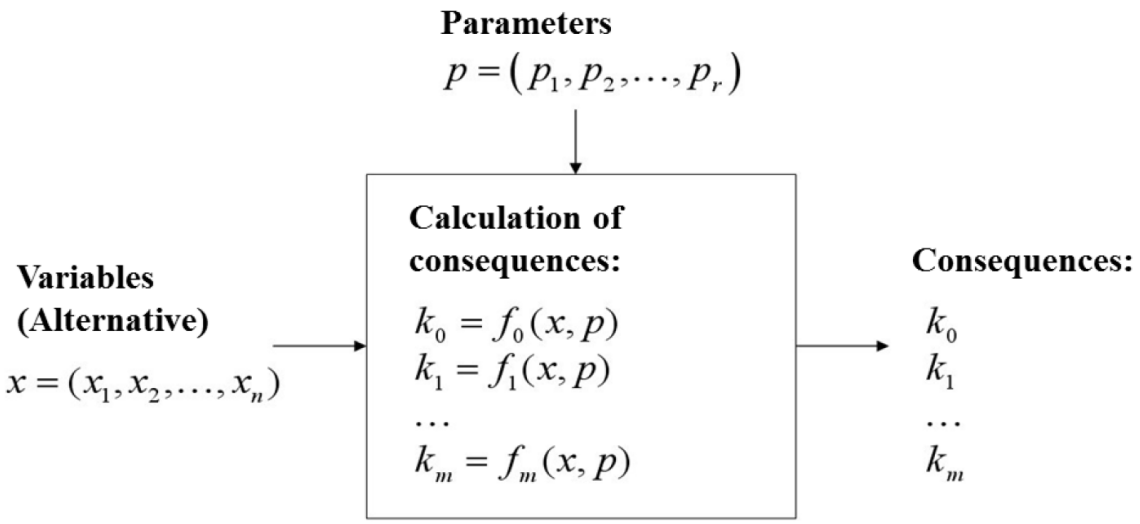


Figure 2.1: Descriptive model

How does it work?

- User **specifies** the alternative x
- Model calculates consequences $f_i(x, p)$.

Functions $f_i(\mathbf{x}, \mathbf{p})$ can be given **explicitly** or **implicitly** ($\rightarrow$ algorithm)

Example: Formulation of a descriptive model

Sets:

I Set of locations, $I = 1, 2, \ldots, 20$

Parameters:

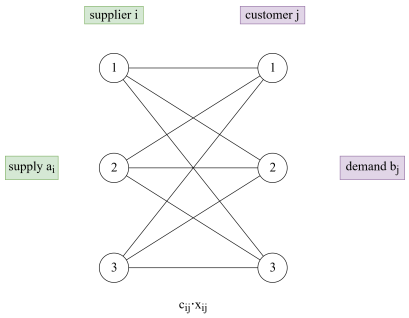
a_i Current stock of vehicles in branch i, $i \in I$

b_j Needed stock of vehicles in branch j, $j \in J$

c_{ij} Travelling distance (in km) from branch i to branch $j; i, j \in I$

Variables:

x_{ij} Number of transferred vehicles from branch i to branch $j; i, j \in I$



Consequences:

k_0	Travelling distance (in km) of all transfers
k_i^{Out}	Number of vehicles leaving branch i, $i \in I$
k_j^{In}	Number of vehicles arriving at branch j, $j \in I$

Model:

$$k_0 = \sum_{i \in I} \sum_{j \in I} c_{ij} \cdot x_{ij}$$

$$k_i^{Out} = \sum_{j \in I} x_{ij} \qquad i \in I$$

$$k_j^{In} = \sum_{i \in I} x_{ij} \qquad j \in I$$

2.2 Optimization Models

Also called Prescriptive models. Question: **"What's Best?"**. Among all **feasible** alternatives an **optimal** alternative is calculated. Set of all feasible solutions: **solution space, feasible set**

- defined by **constraints**, i.e. conditions for consequences (satisfaction)
- typically inequalities and equations

consequences to be optimized: **objective function**.

Functionality:

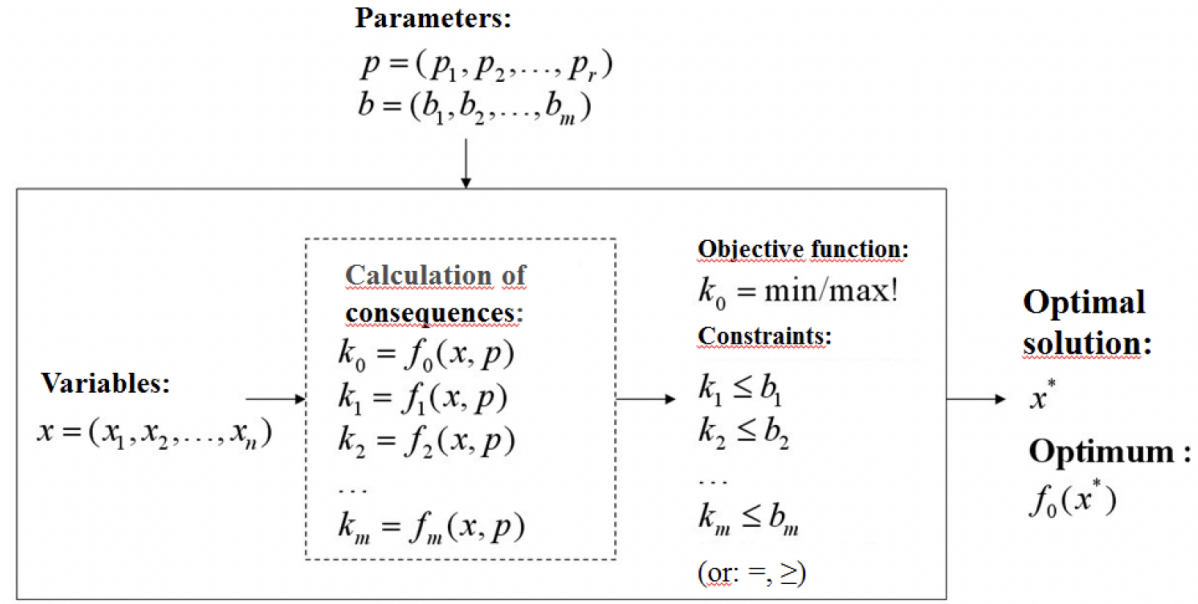


Figure 2.2: Optimization model

- No alternatives need to be specified

  feasible alternatives given by description feasible set
- Model itself provides no solution: **optimization algorithm** is necessary!

Example: Formulation of an optimization model

Sets:

I Set of locations, $I = 1, 2, \ldots, 20$

Parameters:

a_i Current stock of vehicles in branch i, $i \in I$

b_j Needed stock of vehicles in branch j, $j \in J$

c_{ij} Travelling distance (in km) from branch i to branch $j; i$, $j \in I$

Variables:

x_{ij} Number of transferred vehicles from branch i to branch $j; i$, $j \in I$

Constraints:

$$\sum_{j \in I} x_{ij} \leq a_i \qquad i \in I$$

$$\sum_{i \in I} x_{ij} \geq b_j \qquad j \in I$$

$$x_{ij} \geq 0, x_{ij} \in \mathbb{Z}, \qquad i \in I, j \in I$$

Objective Function:

$$\min \sum_{i \in I} \sum_{j \in I} c_{ij} x_{ij}$$

2.2.1 Notice

- Optimization model includes the mathematical definition of the feasible set
  - No alternatives need to be specified
  - Information about "construction" of alternatives integrated in model
- Optimization model includes no methods for finding optimal solutions
  - **Optimization algorithms** are needed for solving a model
  - Many different algorithms depending on the model
- Theory of optimization models and algorithms: **Operations Research**

2.2.2 Example - Product Mixture in an Oil Refinery

The problem can be formulated as a Linear Program as follows:

Sets:

I	Set of raw fuel types, $I = \{1, \ldots, m\}$.
J	Set of jet fuel types, $J = \{1, \ldots, n\}$.

Parameters:

a_i^{PN}	Performance number PN of raw fuel i, $i \in I$.
a_i^{RVP}	Vapor pressure RVP of raw fuel i, $i \in I$.
b_i	Production output (in barrel) of raw fuel i, $i \in I$.
c_i	Price (\$ per barrel) for raw fuel i, $i \in I$.
d_j^{PN}	Minimal performance number PN required for jet fuel j, $j \in J$.
d_j^{RVP}	Maximal vapor pressure RVP required for jet fuel j, $j \in J$.
f_j	Price (\$ per barrel) for jet fuel j, $j \in J$.

Variables:

x_i	Amount (in barrels) of raw fuel i sold directly, $i \in I$.
y_{ij}	Amount (in barrels) of raw fuel i used to produce jet fuel j, $i \in I$, $j \in J$.

Objective function and constraints:

$$\max \sum_{i \in I} c_i x_i + \sum_{j \in J} f_j \sum_{i \in I} y_{ij} \tag{A.10}$$

$$\sum_{i \in I} a_i^{PN} y_{ij} \geq d_j^{PN} \sum_{i \in I} y_{ij}, \qquad j \in J \tag{A.11}$$

$$\sum_{i \in I} a_i^{RVP} y_{ij} \leq d_j^{RVP} \sum_{i \in I} y_{ij}, \qquad j \in J \tag{A.12}$$

$$x_i + \sum_{j \in J} y_{ij} \leq b_i, \qquad i \in I \tag{A.13}$$

$$x_i \geq 0, \qquad i \in I \tag{A.14}$$

$$y_{ij} \geq 0, \qquad i \in I, j \in J \tag{A.15}$$

Consider constraints A.11 and A.12. According to the problem definition, we know that the values PN and RVP of a raw fuel mixture are given by the weighted average of the corresponding values of the individual components. For PN, we obtain the following constraint:

$$\frac{\sum_{i \in I} a_i^{PN} y_{ij}}{\sum_{i \in I} y_{ij}} \geq d_j^{PN}, \qquad j \in J$$

Figure 2.3: Formulation of optimization problem

2.2.3 Example - Shift Planning in a Department Store

1.2.4 Problem 4: Shift Planning in a Department Store

The sales staff in a department store are working in a shift system. During a business reorganization the management are planning to re-engineer the shift system. According to the new regulations, every shift has to include seven or eight working hours. For eight-hours-shifts, one or two hours of rest time have to be granted after four hours of working. For seven-hours-shifts, one hour of rest time has to be granted after three or four hours of working.

The gross costs per working hour are determined as follows: for an eight-hours-shift with one hour rest – 60 francs; for an eight-hours-shift with two hours rest – 70 francs; for a seven-hours-shift – 65 francs.

The opening hours of the department store are from 8:00 till 21:00. The demand for sales staff (number of persons per one hour interval) has been determined basing on customer numbers depending on time and can be found in Table 1.7.

Our task is to find a shift plan covering staff demand with minimal costs.

Figure 2.4: Formulation of optimization problem

A.4 Problem 4: Shift planning in a department store

Sets:

I	Set of time periods, $I = \{1, \ldots, m\}$.
J	Set of shifts, $J = \{1, \ldots, n\}$.

Parameters:

b_i	Demand (number of persons) in time period i, $i \in I$.
c_j	Costs for shift j, $j \in J$. They are determined by using costs per working hour.
a_{ij}	Binary indicator with value 1 if and only if time period i is included in shift j, $i \in I$, $j \in J$.

Variables:

x_j	Number of persons planned for shift j, $j \in J$.

Objective function and constraints:

$$\min \sum_{j \in J} c_j x_j$$

$$\sum_{j \in J} a_{ij} x_j \geq b_i \qquad i \in I$$

$$x_j \in \mathbb{Z}_0 \qquad j \in J$$

Figure 2.5: Formulation of optimization problem

2.2.4 Example - Feeds production

A company produces different types of feed for farm animals by mixing several ingredients. Each ingredient contains a certain amount of protein and calcium (given in gram per kg), and each type of feed requires a minimum total amount of protein and calcium (given in gram per kg). Furthermore, the purchase price for each ingredient is given (in dollar per kg), and the sales price for each type of feed is given (in dollar per kg). Finally, the production quantity of each feed type should not exceed a specified limit (in kg).

Formulate a linear programming model which calculates an optimal production plan, i.e. a production plan that maximizes total profit.

Figure 2.6: Formulation of optimization problem

Sets :

I Set of ingredients, $I = \{1,\ldots,m\}$

J Set of feed types, $J = \{1,\ldots,n\}$

Parameters :

a_i^{Prot} Amount of protein (gram per kg) contained in ingredient i, $i \in I$

a_i^{Calc} Amount of calcium (gram per kg) contained in ingredient i, $i \in I$

d_i^{Prot} Total amount of protein (gram per kg) required for feed type j, $j \in J$

d_i^{Calc} Total amount of calcium (gram per kg) required for feed type j, $j \in J$

f_i Purchase price (dollar per kg) for ingredient i, $i \in I$

c_j Sales price (dollar per kg) for feed type j, $j \in J$

b_j Maximum production quantity (kg) for feed type j, $j \in J$

Variables :

x_{ij} Amount (kg) of ingredient i mixed into feed type j, $i \in I, j \in J$

$$\max \sum_{j\in J} c_j \sum_{i\in I} x_{ij} - \sum_{i\in I} f_i \sum_{j} x_{ij}$$

$$\sum_{i\in I} a_i^{\text{Prot}} x_{ij} \geq d_j^{\text{Prot}} \sum_{i\in I} x_{ij}, \quad j \in J$$

$$\sum_{i\in I} a_i^{\text{Calc}} x_{ij} \geq d_j^{\text{Calc}} \sum_{i\in I} x_{ij}, \quad j \in J$$

$$\sum_{i\in I} x_{ij} \leq b_j, \quad j \in J$$

$$x_{ij} \geq 0, \quad i \in I, j \in J$$

Figure 2.7: Formulation of optimization problem

2.3 Statement of the Model

- **General optimization problem:**

  $\prod : \max\{f(\mathbf{x}) : \mathbf{x} \in S\}$, where $S \subseteq \mathbb{R}^n$

- decision variables (Entscheidungsvariablen): $\mathbf{x} = (x_1, x_2, ..., x_n)^T \in \mathbb{R}^n$
- (feasible) solution (zulässige Lösung): $\mathbf{x} \in S$
- feasible set (Lösungsmenge): $S \subseteq \mathbb{R}^n$
- objective function (Zielfunktion): $f : S \to \mathbb{R}$
- **optimal solution:**

  $\mathbf{x}^* \in S$ such that $f(\mathbf{x}^*) \geq f(\mathbf{x})$ for all $\mathbf{x} \in S$

  This means basically, that the objective value of the optimal solution $x^* \in S$ can not be worse than the objective value of any other feasible solution ($x \in S$). Note that an optimization problem can have multiple optimal solutions, but only one optimum.

- **optimum, optimal value:** $f(\mathbf{x}^*)$

2.3.1 Conditions ensuring the Existence of an Optimal Solution

- **Feasibility:** $S \neq \emptyset$
  There must be at least one feasible solution.

  Example for infeasibility: $\max\{x_1 : 2x_1 + 4x_2 = 5, \mathbf{x} \in \mathbb{Z}^2\}$

- **Boundedness:** $\exists \omega : f(\mathbf{x}) \leq \omega$ for all $\mathbf{x} \in S$
  There exists an upper bound for the feasible solutions. For example: Tell me the highest number above 100!

  Example unbounded problem: $\max\{x_1 : 2x_1 + 4x_2 = 5, \mathbf{x} \in \mathbb{R}^2\}$

- **Closedness of S:** S is a closed set
  Example S not closed: $\max\{x_1 : x_1 < 1, x_1 \in \mathbb{R}\}$

- **Continuity of f:** f is a continuous function
  f not continuous: $\max\{f(x) : x \in \mathbb{R}, x \geq 0\}$ with

$$f(x) = \begin{cases} 3 - x, & \text{if } x > 0 \\ 2, & \text{if } x = 0 \end{cases} \tag{2.1}$$

Definition of the feasible set:

i) **Functional Constraints**: The solution space S is defined by a set of *inequalities* of the form $g_i(\mathbf{x}) \leq 0, i = 1, 2, \ldots, p(p \geq 0)$, and a set of *equalities* $h_j(\mathbf{x}) = 0, j = 1, 2, \ldots, q(q \geq 0)$:

$S = \{\mathbf{x} \in G_1 \times \cdots \times G_n : g_i(\mathbf{x}) \leq 0, h_j(\mathbf{x}) = 0, i = 1, \ldots, p, j = 1, \ldots, q\}$
where $G_i = \mathbb{R}, \mathbb{Q}, \mathbb{Z}, \mathbb{R}_0, \mathbb{Q}_0, \mathbb{Z}_0, \{0, 1\}, \ldots$

ii) **Non-Functional Constraints**: The definition of the solution space S may contain "non-functional" constraints, for instance logical predicates which select the feasible elements from a given set, based on certain properties:

$S = \{\mathbf{x} \in G_1 \times \cdots \times G_n : \mathbf{x}$ has certain properties$\}$

The following example shows a non-functional definition of a solution space:

$S = \{x \in \mathbb{Z}^n : \mathbf{x}$ is a permutation of the numbers $1, 2, \ldots, n\}$

It is theoretically possible in most cases to find, for a set defined by non-functional constraints, a corresponding formulation with functional constraints. However, finding such a formulation can be very challenging.

Maximization and Minimization

Optimization problems can be maximization or minimization problems. Every maximization problem can easily be transformed into an "equivalent" minimization problem (and vice versa), since we have

$\max\{f(\mathbf{x}) : \mathbf{x} \in S\} = -\min\{-f(\mathbf{x}) : \mathbf{x} \in S\}$

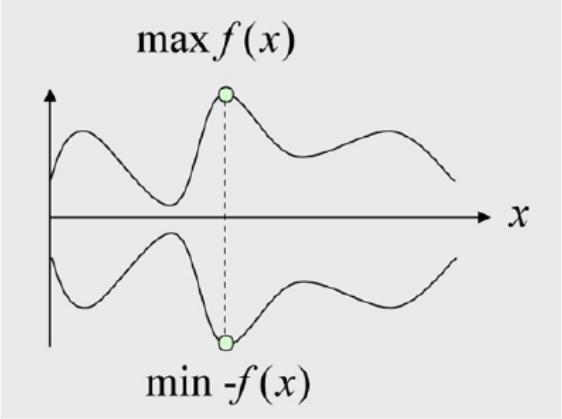


Figure 2.8: Maximization and Minimization

2.3.2 Some Notions and Concepts

Problem and Problem Instances

Problem:

$$\max\{\sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} a_j x_j \leq b, \mathbf{x} \in \mathbb{R}^n\}$$

Instance: Defining numerical parameters n, a, b, c

$$\max\{4x_1 + 7x_2 : 3x_1 + 5x_2 \leq 17, \mathbf{x} \in \mathbb{R}^2\}$$

Neighborhood Notions

Let $S \subseteq \mathbb{R}^n S$ be an arbitrary set of "points" in $\mathbb{R}^n$, and $P(S) = \{S' : S' \subseteq\}$ the corresponding power set, i.e. the set of all subsets of S. A *neighborhood* defined on S is a (set-valued) function of the form $N : S \to P(S)$ that assigns to each point $\mathbf{x} \in S$ a set of neighboring points $N(\mathbf{x})$, assuming always $\mathbf{x} \in N(x)$, i.e. $\mathbf{x}$ is a neighbor of itself. $N(\mathbf{x}$ is called a neighborhood of $\mathbf{x}$.

- Neighborhood: $N : S \to P(S)$ where $S \subseteq \mathbb{R}^2$
  $\mathbf{x} \mapsto N(\mathbf{x})$
- Neighborhood solutions: $N(\mathbf{x}) \subseteq S$ where $\mathbf{x} \in N(\mathbf{x})$

Euclidean Neighborhood, ε-neighborhood

A special neighborhood in $\mathbb{R}^n$ is the so-called *Euclidean Neighborhood*, or $\varepsilon - neighborhood$.

- $N_\varepsilon(\mathbf{x}) = \{\mathbf{y} \in \mathbb{R}^n : \|\mathbf{y} - \mathbf{x}\| < \varepsilon\}$ where $\varepsilon > 0$
- Euclidean norm: $\|\mathbf{x}\| = \sqrt{x_1^2 + \cdots + x_n^2}$

In $\mathbb{R}^2$ (or $\mathbb{R}^3$), the neighborhood $N_\varepsilon(\mathbf{x})$ corresponds to the set of all points $\mathbf{y}$ located inside a circle (or ball) with center $\mathbf{x}$ and radius $\varepsilon > 0$, with "boundary" points excluded. An illustration for $\mathbb{R}^2$ is given in Figure 2.9.

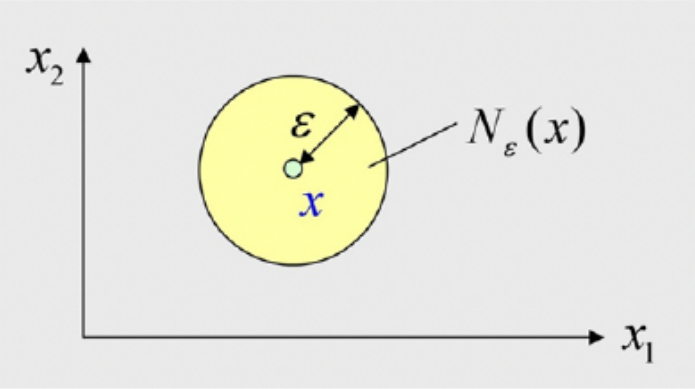


Figure 2.9: Euclidean Neighborhood, ε-neighborhood

Notions: consider $S \subseteq \mathbb{R}^n$:

- interior point $\mathbf{x}$: $N_\varepsilon(\mathbf{x}) \subseteq S$ for a suitable $\varepsilon > 0$
- boundary point $\mathbf{x}$: $N_\varepsilon(\mathbf{x}) \cap S \neq \emptyset$ and $N_\varepsilon(\mathbf{x}) \cap (\mathbb{R}^n - S) \neq \emptyset$ for all $\varepsilon > 0$
- S closed: all boundary points of S belong to S
- S open: all $\mathbf{x} \in S$ are interior points of S
- S bounded: $S \subseteq \{\mathbf{x} \in \mathbb{R}^n : \mathbf{a} \leq \mathbf{x} \leq \mathbf{b}\}$ for suitable $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$

Theorem 1. ***Extreme Value Theorem of Weierstrass***
Let $S \subseteq \mathbb{R}^n$ be a non-empty, closed, bounded set and $f : S \to \mathbb{R}$ be continuous.
Then $\max\{f(\boldsymbol{x}) : \boldsymbol{x} \in S\}$ *has a finite optimum,*
i.e. $\exists \mathbf{x}^* \in S$ *such that* $f(\mathbf{x}^*) = \max\{f(\mathbf{x} : \mathbf{x} \in S\}$.

Example: Combinatorial neighborhood
$N(\mathbf{x}) = \{\mathbf{y} \in \mathbb{Z}^n : \mathbf{y}$ differs form $\mathbf{x}$ in at most one component$\}$

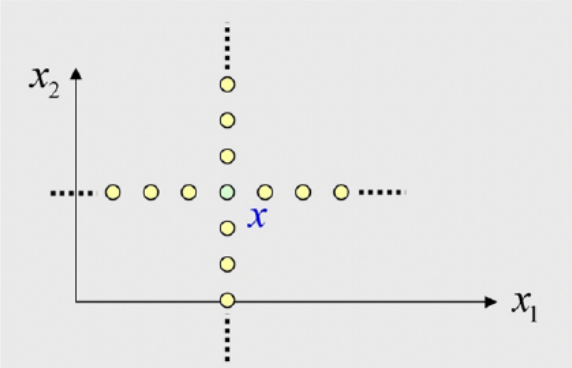


Figure 2.10: Combinatorial neighborhood

Global and Local Optima

Consider an optimization problem of the form $\prod : \max\{f(\mathbf{x}) : \mathbf{x} \in S\}$ with $S \subseteq \mathbb{R}^n$. As already mentioned, $\mathbf{x}^* \in S$ is called an optimal solution of $\prod$ if

- global optimal solution. $\mathbf{x}^* : f(\mathbf{x}^*) \geq f(\mathbf{x})$ for all $\mathbf{x} \in S$ (in case of maximization)

$\mathbf{x}^*$ is also called a *globally optimal solution*, and the optimum $f(\mathbf{x}^*)$ is called the *global optimum*.

Besides the global optimum of a problem, there may exist so-called *local optima* which also play an important role in optimization.

- local optimal solution $\mathbf{x}^* : \exists N(\mathbf{x}^*)$ such that $f(\mathbf{x}^*) \geq f(\mathbf{x})$ for all $\mathbf{x} \in N(\mathbf{x}^*) \cap S$.

Expressed in words, a solution is locally optimal if it is as least as good as all its neighbors in a certain neighborhood. Note that every globally optimal solution is also a locally optimal solution **(but not conversely)**.

Local optimal solution is referring to Euclidean neighborhood $N_\varepsilon(\mathbf{x})$ or to different neighborhoods. For different neighborhoods different local optimal solutions are possible.

Figure 2.11 shows an example in $\mathbb{R}^1$. Note that in $\mathbb{R}^1$, the ε-neighborhoods to be considered for checking local optimality of a solution $x \in \mathbb{R}^1$ correspond to the open intervals $]x - \varepsilon, x + \varepsilon[$.

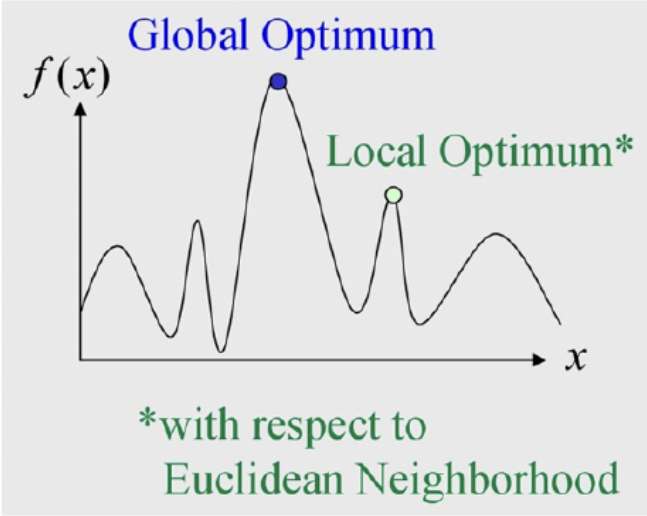


Figure 2.11: Global and Local Optima

Level Sets

Level sets of functions play an important role in analysis and optimization, notably for the graphical representation of 2- and 3-dimensional functions. Recall that the graph H of a function $f : S \to$ with $S \subseteq \mathbb{R}^n$ is the $n+1$ dimensional subset $H \subseteq \mathbb{R}^{n+1}$ defined by

- $H = \{(\mathbf{x}, f(\mathbf{x})) : \mathbf{x} \in S\}$
- Level set for level α is:
  $L_\alpha = \{\mathbf{x} \in S : f(\mathbf{x}) = \alpha\}$.

  When the number of variables is two, a level set is generally a curve and called a level curve or contour line. For n = 3, level sets are typically "surfaces" in the 3-dimensional space.

  By means of level sets, graphs of 2- and 3-dimensional functions (which are 3- and 4-dimensional sets, respectively) can be represented in lower dimension as 2- and 3- dimensional images, respectively. An example is given in Figure2.12.

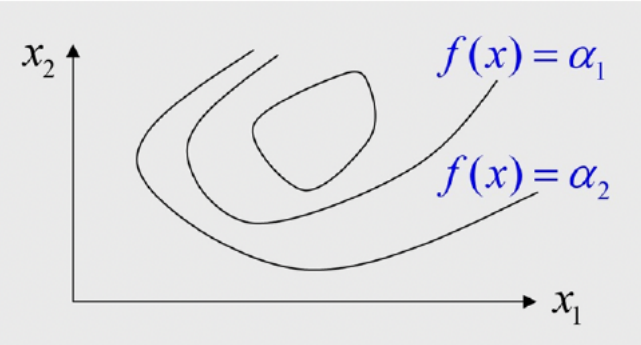


Figure 2.12: Level set

- **Special case: linear functions**
  If $f : \mathbb{R}^2 \to \mathbb{R}$ is a linear function, i.e. if $f(\mathbf{x}) = c_1x_1 + c_2x_2$ for some $\mathbf{c} \in \mathbb{R}^2$, the level sets are parallel lines in $\mathbb{R}^2$, and the vector $\mathbf{c}$ (the gradient of f) is orthogonal to these lines. The vector $\mathbf{c}$ points in the direction where the level α (i.e. function value) increases if level lines are "shifted in parallel" in this direction. See Figure 2.13 for an example.

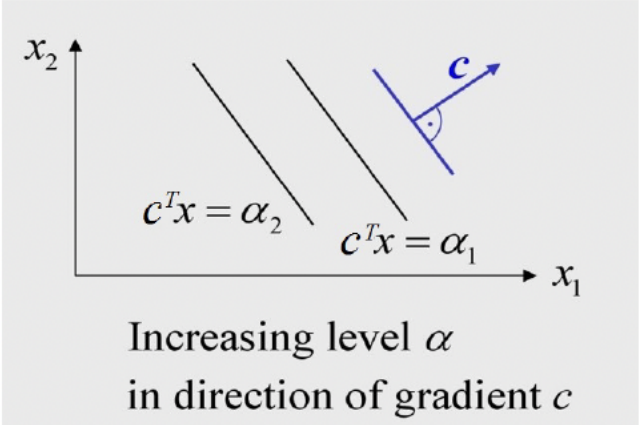


Figure 2.13: Special case: linear functions

2.3.3 1. Example for Neighborhoods

Consider a base set

$$S = \{x \in \{0,1\}^5 : \sum_{i=1}^{5} x_i = 3\}$$

and a function given by

$$f: S \to \mathbb{R} \text{ with } f(x) = c^T x, \text{ where } c^T = (5,3,7,1,2).$$

Further, we define on S a neighbourhood notion N as follows:

$$N(x) = \{x' \in S : x'_i = x_{i \bmod 5+1} \text{ for } i = 1,\ldots,5\} \cup$$
$$\{x' \in S : x'_i = x_{(i+3) \bmod 5+1} \text{ for } i = 1,\ldots,5\} \cup \{x\}$$

a) Determine all elements of S and the corresponding function values of f.

b) Determine all local (with regard to N) and all global maximal and minimal solutions of f.

c) Choose any local minimal solution and prove that it is indeed a local minimal solution.

Hint: To understand the terms $x'_i = x_{i \bmod 5+1}$ and $x'_i = x_{(i+3) \bmod 5+1}$ enter some values of i into the formulas. The operator "mod" describes the modulo operator, which calculates the remainder when performing integer division. For instance, $5 \bmod 5 = 0$ and $6 \bmod 5 = 1$.

Figure 2.14: 1. Example for Neighborhoods

a) $S = \{x \in \{0,1\}^5 : \sum_{i=1}^{5} x_i = 3\}$

All elements of S consists of 5 numbers whereby 3 of them are 1 and two of them are 0. All possible combinations of this are elements of S. The solution space consists of exactly 10 solutions because the cardinality of S is given by

$$|S| = \binom{5}{3} = \frac{5!}{3!\,(5-3)!} = \frac{5!}{3!\,2!} = \frac{5 \cdot 4}{2 \cdot 1} = 10$$

$f: S \to \mathbb{R}$ with $f(x) = c^T x$, where $c^T = (5,3,7,1,2)$

We now just need to put these values into function f to get the corresponding function values. All elements and values of the function are listed below.

b) The neighborhood $N(x)$ is defined as the union of three sets where the last set just contains the vector x itself.

$N(x) = \{x' \in S : x'_i = x_{i \bmod 5+1} \text{ for } i = 1,\ldots,5\} \cup \{x' \in S : x'_i = x_{(i+3) \bmod 5+1} \text{ for } i = 1,\ldots,5\} \cup \{x\}$ Now we insert the values $1,\ldots,5$:

$x'_1 = x_{1 \bmod 5+1} = x_2$ $\quad$ $x'_1 = x_{(1+3) \bmod 5+1} = x_5$

$x'_2 = x_{2 \bmod 5+1} = x_3$ $\quad$ $x'_2 = x_{(2+3) \bmod 5+1} = x_1$

$x'_3 = x_{3 \bmod 5+1} = x_4$ $\quad$ $x'_3 = x_{(3+3) \bmod 5+1} = x_2$

$x'_4 = x_{4 \bmod 5+1} = x_5$ $\quad$ $x'_4 = x_{(4+3) \bmod 5+1} = x_3$

$x'_5 = x_{5 \bmod 5+1} = x_1$ $\quad$ $x'_5 = x_{(5+3) \bmod 5+1} = x_4$

cyclic right shift: $x = (1,1,1,0,0)^T \to x^{right} = (1,1,0,0,1)^T$

cyclic left shift: $x = (1,1,1,0,0)^T = (1,1,0,0,1)^T$

All local/global Max/Min points are listed below in the table.

x:	$f(x)$:	x^{right}:	$f(x^{right})$:	x^{left}:	$f(x^{left})$:	Max:	Min:
(1,1,1,0,0)	15	(0,1,1,1,0)	11	(1,1,0,0,1)	10	global	—
(0,1,1,1,0)	11	(0,0,1,1,1)	10	(1,1,1,0,0)	15	—	—
(0,0,1,1,1)	10	(1,0,0,1,1)	8	(0,1,1,1,0)	11	—	—
(1,0,0,1,1)	8	(1,1,0,0,1)	10	(0,0,1,1,1)	10	—	local
(1,1,0,0,1)	10	(1,1,1,0,0)	15	(1,0,0,1,1)	8	—	—
(1,1,0,1,0)	9	(0,1,1,0,1)	12	(1,0,1,0,1)	14	—	local
(0,1,1,0,1)	12	(1,0,1,1,0)	13	(1,1,0,1,0)	9	—	—
(1,0,1,1,0)	13	(0,1,0,1,1)	6	(0,1,1,0,1)	12	local	—
(0,1,0,1,1)	6	(1,0,1,0,1)	14	(1,0,1,1,0)	13	—	global
(1,0,1,0,1)	14	(1,1,0,1,0)	9	(0,1,0,1,1)	6	local	—

c) When we look at the local minima $x^*(1,0,0,1,1)$ we need to verify that the values in the neighborhood are bigger. This is clearly the case because $8 \leq 10$ and $8 \leq 10$.

Figure 2.15: 1. Example for Neighborhoods

2.3.4 2. Example for Neighborhoods

Consider the base set

$$S = \{\boldsymbol{x} \in \{0,1\}^4 : x_i - x_{i+1} \geq 0 \text{ for } i = 1,\ldots,3\}$$

and a function defined on this set given by

$$f : S \to \mathbb{R} \quad \text{with } f(\boldsymbol{x}) = \boldsymbol{c}^\top \boldsymbol{x} \text{ where } \boldsymbol{c} = (-5,-2,9,-4)^\top .$$

Further, for all $\boldsymbol{x} \in S$ a neighbourhood notion is defined as follows:

$$N(\boldsymbol{x}) = \{\boldsymbol{x}' \in S : |\{i \in \{1,\ldots,4\} : x_i' - x_i \neq 0\}| \leq 1\} .$$

Hint: The neighbourhood solutions $N(\boldsymbol{x})$ are the vectors $\boldsymbol{x}' \in S$ which differ from $\boldsymbol{x}$ in at most one component.

a) (3 points)
Specify all elements $\boldsymbol{x}$ of the base set S and the corresponding function values $f(\boldsymbol{x})$.

b) (4 points)
For all elements $\boldsymbol{x} \in S$, specify the set of neighbourhood solutions $N(\boldsymbol{x})$.

c) (3 points)
Specify for every $\boldsymbol{x} \in S$, whether the following statements are true:

- $\boldsymbol{x}$ is a local maximum with regard to $N(\boldsymbol{x})$
- $\boldsymbol{x}$ is a local minimum with regard to $N(\boldsymbol{x})$
- $\boldsymbol{x}$ is a global maximum
- $\boldsymbol{x}$ is a global minimum

Figure 2.16: 2. Example for Neighborhoods

$S = \{x \in \{0,1\}^4 : x - x_{i+1} \geq 0 \text{ for } i = 1,\ldots,3\}$

and $f : S \to \mathbb{R}$ with $f(x) = c^\top x$ where $c = (-5,-2,5,-4)^\top$

Neighborhood $N(x) = \{x' \in S : |\{i \in \{1,\ldots,4\} : x_i' - x_i \neq 0| \leq 1\}$

a) x	a) $f(x)$	b) $N(x) - \{x\}$	c) Min/Max
$(0,0,0,0)$	0	$(1,0,0,0)$	Local max
$(1,0,0,0)$	-5	$(0,0,0,0)$, $(1,1,0,0)$	—
$(1,1,0,0)$	-7	$(1,0,0,0)$, $(1,1,1,0)$	global min
$(1,1,1,0)$	2	$(1,1,0,0)$, $(1,1,1,1)$	global max
$(1,1,1,1)$	-2	$(1,1,1,0)$	local min

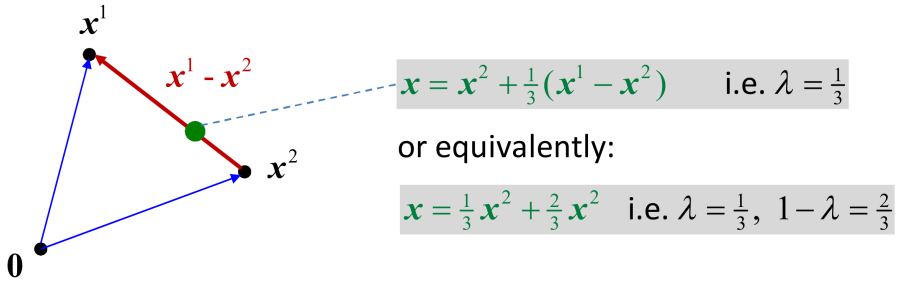
Figure 2.17: 2. Example for Neighborhoods

2.3.5 Convex Optimization

Convex combination of two points $\mathbf{x}^1, \mathbf{x}^2 \in \mathbb{R}^n$:

$$\mathbf{x} = \lambda \mathbf{x}^1 + (1-\lambda)\mathbf{x}^2 \qquad \text{for some } \lambda \in \mathbb{R} \text{ with } 0 \leq \lambda \leq 1$$
$$\mathbf{x} = \mathbf{x}^2 + \lambda(\mathbf{x}^1 - \mathbf{x}^2) \qquad \text{for some } \lambda \in \mathbb{R} \text{ with } 0 \leq \lambda \leq 1$$

Geometrically this can be understood as a convex combination of $\mathbf{x}^1$ and $\mathbf{x}^2$ that lies on the line between $\mathbf{x}^1$ and $\mathbf{x}^2$.



A **convex combination of multiple points** $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^k \in \mathbb{R}^n$ takes the form:

$$\mathbf{x} = \sum_{i=1}^{k} \lambda_i \mathbf{x}^i \qquad \text{for some } \boldsymbol{\lambda} \in \mathbb{R}^k \text{ with } 0 \leq \lambda \leq 1 \text{ and } \sum_{i=1}^{k} \lambda_i = 1$$

Expressed in words: A convex combination is a linear combination with non-negative coefficients that sum up to 1.

Convex set $S \subseteq \mathbb{R}^n$:

$$\lambda \mathbf{x}^1 + (1-\lambda)\mathbf{x}^2 \in S \quad \forall \mathbf{x}^1, \mathbf{x}^2 \in S \text{ and all } \lambda \in \mathbb{R}, 0 \leq \lambda \leq 1$$

Expressed in words: A set S is convex if the line between two arbitrary points of S is entirely contained in S.

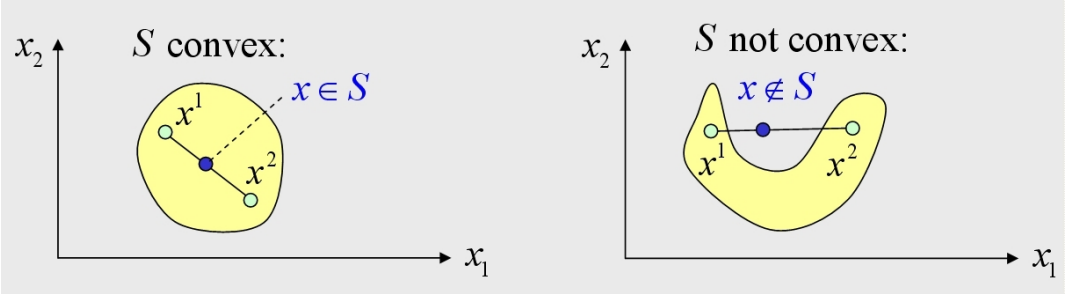


Figure 2.18: Illustration of convex and non-convex sets

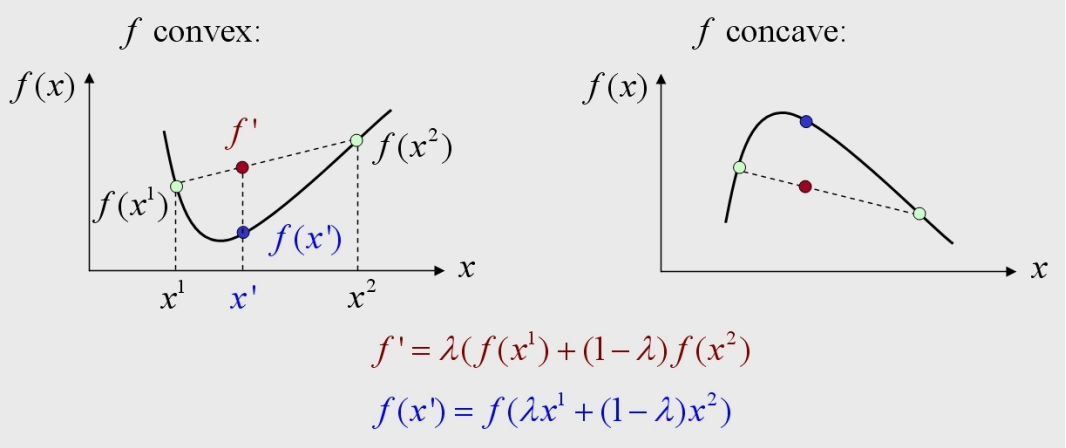
Theorem 2. *Any intersection of convex sets is a convex set.*

Convex function $f : S \to \mathbb{R}$, where S convex:

$$f(\lambda \mathbf{x}^1 + (1-\lambda)\mathbf{x}^2) \leq \lambda f(\mathbf{x}^1) + (1-\lambda) f(\mathbf{x}^2) \qquad \forall \mathbf{x}^1, \mathbf{x}^2 \in S, \lambda \in [0,1]$$

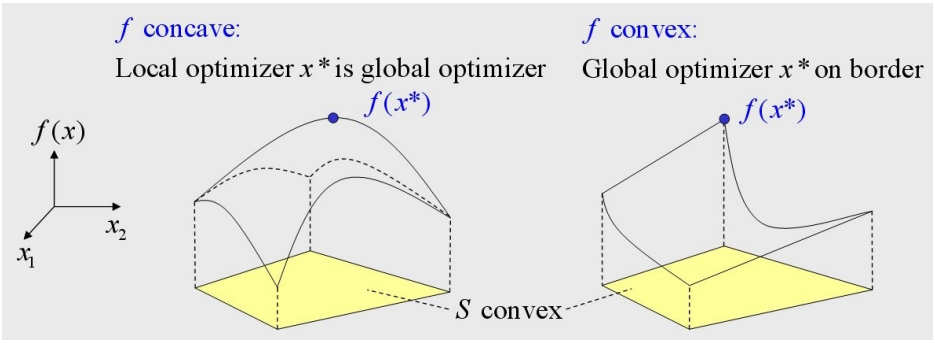
Concave function $f : S \to \mathbb{R}$, where S concave:

$$f(\lambda \mathbf{x}^1 + (1-\lambda)\mathbf{x}^2) \geq \lambda f(\mathbf{x}^1) + (1-\lambda) f(\mathbf{x}^2) \qquad \forall \mathbf{x}^1, \mathbf{x}^2 \in S, \lambda \in [0,1]$$

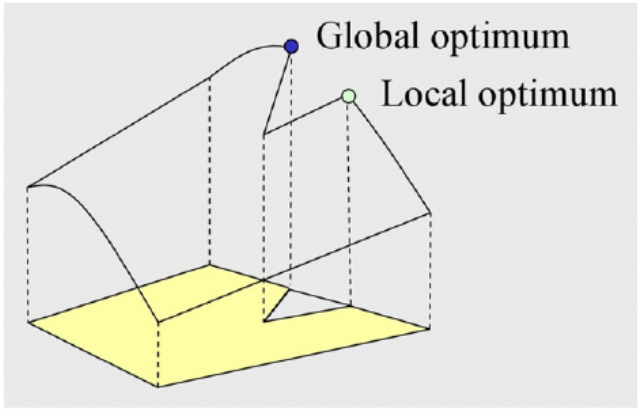


- **Linear function**: convex and concave
- Convex optimization problem: $\max\{f(\mathbf{x}) : \mathbf{x} \in S\}$ with f **convex** and let S be **convex**
- "Convex" optimization problem as well: $\min\{f(\mathbf{x}) : \mathbf{x} \in S\}$ with f **concave** and S **convex**

Theorem 3. *For a convex optimization problem any **local optimum** is a **global optimum**.*



Theorem 4. *Consider the problem $\max\{f(\mathbf{x}) : \mathbf{x} \in S\}$ with f **convex** and let S be **convex**, closed and bounded (alternatively consider $\min\{f(\mathbf{x}) : \mathbf{x} \in S\}$ with f **concave**). Further, let f not be constant in S. Then any **local optimum** is a **boundary point** of S.*



Convex programming :

$$\max\{f(\mathbf{x}) : g_i(\mathbf{x}) \leq b_i, i \in I, \mathbf{x} \in \mathbb{R}^n\}$$

where f concave and $g_i, i \in I$ are all convex.

Theorem 5. $S = \{\mathbf{x} \in \mathbb{R}^n : g_i(\mathbf{x}) \leq b_i, i \in \mathbf{I}\}$ *is convex.*

2.4 Types of optimization problems and methods

2.4.1 Optimization models:

- Constrained vs. Unconstrained
- Global vs. Local
- Differentiable vs. Non-differentiable
- Discrete vs. Continuous
- Convex vs. Non-convex
- Linear vs. Nonlinear

2.4.2 Optimization methods:

- Exact vs. Heuristic
- General vs. Problem specified

Types of heuristics:

- Constructive
- Improving heuristics
  - Local Search
  - general Meta Heuristics

Chapter 3

Linear Programming (LP)

3.1 Problem Formulation

Assumptions and notation:
- $\mathbf{x} \in \mathbb{R}^n$
- $\mathbf{A} \in \mathbb{R}^{m \times n}$
- $\mathbf{b} \in \mathbb{R}^m$
- $\mathbf{c} \in \mathbb{R}^n$
- $I = \{1, \ldots, m\}$
- $J = \{1, \ldots, n\}$
- $\mathbf{a}^i \in \mathbb{R}^n$ is i-th row of $\mathbf{A}$
- $\mathbf{A}^j \in \mathbb{R}^m$ is j-th column of $\mathbf{A}$

Linear function: $f : \mathbb{R}^n \to \mathbb{R}$

$$f(\mathbf{x}) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \sum_{j=1}^{n} a_j x_j = \mathbf{a}^T \mathbf{x} \quad (\mathbf{a} \in \mathbb{R}^n)$$

Linear inequality:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i, i \in I \text{ resp. } \mathbf{a}^i \mathbf{x} \leq b_j, i \in I \text{ resp. } \mathbf{A}\mathbf{x} \leq \mathbf{b}$$

This is analogous for linear inequalities of type "$\geq$" and linear equations "$=$". The general definition of linear functions f is as follows:

$$f(\alpha \mathbf{x} + \beta \mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y}) \quad (\alpha, \beta \in \mathbb{R}) \quad \textbf{Linearity}$$

A **linear program (LP)** should minimise a linear objective function subject to linear constraints (linear inequalities or linear equalities)

3.1.1 General Form of a Linear Program

$\min / \max \mathbf{c}^T \mathbf{x}$
$\mathbf{a}^i \mathbf{x} \leq b_i, i \in I_1$
$\mathbf{a}^i \mathbf{x} = b_i, i \in I_2$
$\mathbf{a}^i \mathbf{x} \geq b_i, i \in I_3$
$x_j \geq 0, j \in J_1$
x_j free, $j \in J_2$
$x_j \leq 0, j \in J_3$

3.1.2 Canonical Form of a Linear Program

$\max \mathbf{c}^T \mathbf{x}$
$\mathbf{a}^i \mathbf{x} \leq b_i, i \in I$
$x_j \geq 0, j \in J$

$\min \mathbf{c}^T \mathbf{x}$
$\mathbf{a}^i \mathbf{x} \geq b_i, i \in I$
$x_j \geq 0, j \in J$

3.1.3 Standard Form of a Linear Program

$\min / \max \mathbf{c}^T \mathbf{x}$
$\mathbf{a}^i \mathbf{x} = b_i, i \in I$
$x_j \geq 0, j \in J$

3.1.4 Inequality Form of a Linear Program

$\max \mathbf{c}^T \mathbf{x}$
$\mathbf{a}^i \mathbf{x} \leq b_i, i \in I$

$\min \mathbf{c}^T \mathbf{x}$
$\mathbf{a}^i \mathbf{x} \geq b_i, i \in I$

3.1.5 Transformations

- **Inequalities to Inequalities**
  $\mathbf{a}^i \mathbf{x} \leq b_i \quad \leftrightarrow \quad -\mathbf{a}^i \mathbf{x} \geq -b_i$

- **Equalities to inequalities**
  $\mathbf{a}^i \mathbf{x} = b_i \quad \rightarrow \quad \mathbf{a}^i \mathbf{x} \leq b_i, \mathbf{a}^i \mathbf{x} \geq b_i$

- **Inequalities to equalities**
  $\mathbf{a}^i \mathbf{x} \leq b_i \quad \rightarrow \quad \mathbf{a}^i \mathbf{x} + x_i^s = b_i, x_i^s \geq 0$ (slack variable)
  $\mathbf{a}^i \mathbf{x} \geq b_i \quad \rightarrow \quad \mathbf{a}^i \mathbf{x} - x_i^s = b_i, x_i^s \geq 0$ (surplus variable)

- **Non-positive to non-negative variables**
  $x_j \leq 0 \quad \rightarrow \quad x_j := -\bar{x}_j, \bar{x}_j \geq 0$

- **Free to non-negative variables**
  x_j free $\quad \rightarrow \quad x_j := x_j^+ - x_j^- \qquad x_j^+, x_j^- \geq 0$

Exercise 3

Replace non-positive variable by non-negative variable: $x_j \leq 0 \rightsquigarrow x_j := -\bar{x}_j,\ \bar{x}_j \geq 0$

$$\begin{aligned} \min\ & x_1 - 2x_2 - 3x_3 \\ & x_1 + 2x_2 + 4x_3 \geq 12 \\ & x_1 - x_2 + x_3 = 2 \\ & x_1 + 2x_2 + x_3 \leq 14 \\ & x_1 \geq 0 \\ & x_2 \text{ free} \\ & x_3 \leq 0 \end{aligned} \qquad \begin{aligned} \min\ & x_1 - 2x_2 + 3\bar{x}_3 \\ & x_1 + 2x_2 - 4\bar{x}_3 \geq 12 \\ & x_1 - x_2 - \bar{x}_3 = 2 \\ & x_1 + 2x_2 - \bar{x}_3 \leq 14 \\ & x_1 \geq 0 \\ & x_2 \text{ free} \\ & \bar{x}_3 \geq 0 \end{aligned}$$

Replace free variable by the difference of two non-negative variables: x_j free $\rightsquigarrow x_j := x_j^+ - x_j^-,\ x_j^+, x_j^- \geq 0$

$$\begin{aligned} \min\ & x_1 - 2(x_2^+ - x_2^-) + 3\bar{x}_3 \\ & x_1 + 2(x_2^+ - x_2^-) - 4\bar{x}_3 \geq 12 \\ & x_1 - (x_2^+ - x_2^-) - \bar{x}_3 = 2 \\ & x_1 + 2(x_2^+ - x_2^-) - \bar{x}_3 \leq 14 \\ & x_1 \geq 0 \\ & x_2^+ \geq 0 \\ & x_2^- \geq 0 \\ & \bar{x}_3 \geq 0 \end{aligned} \quad = \quad \begin{aligned} \min\ & x_1 - 2x_2^+ + 2x_2^- + 3\bar{x}_3 \\ & x_1 + 2x_2^+ - 2x_2^- - 4\bar{x}_3 \geq 12 \\ & x_1 - x_2^+ + x_2^- - \bar{x}_3 = 2 \\ & x_1 + 2x_2^+ - 2x_2^- - \bar{x}_3 \leq 14 \\ & x_1 \geq 0 \\ & x_2^+ \geq 0 \\ & x_2^- \geq 0 \\ & \bar{x}_3 \geq 0 \end{aligned}$$

Maximizing problem in canonical form

$$\begin{aligned} \max\ & -x_1 + 2x_2^+ - 2x_2^- - 3\bar{x}_3 \\ & -x_1 - 2x_2^+ + 2x_2^- + 4\bar{x}_3 \leq -12 \\ & x_1 - x_2^+ + x_2^- - \bar{x}_3 \leq 2 \\ & -x_1 + x_2^+ - x_2^- + \bar{x}_3 \leq -2 \\ & x_1 + 2x_2^+ - 2x_2^- - \bar{x}_3 \leq 14 \\ & x_1 \geq 0 \\ & x_2^+ \geq 0 \\ & x_2^- \geq 0 \\ & \bar{x}_3 \geq 0 \end{aligned}$$

Minimizing problem in canonical form

$$\begin{aligned} \min\ & x_1 - 2x_2^+ + 2x_2^- + 3\bar{x}_3 \\ & x_1 + 2x_2^+ - 2x_2^- - 4\bar{x}_3 \geq 12 \\ & x_1 - x_2^+ + x_2^- - \bar{x}_3 \geq 2 \\ & -x_1 + x_2^+ - x_2^- + \bar{x}_3 \geq -2 \\ & -x_1 - 2x_2^+ + 2x_2^- + \bar{x}_3 \geq -14 \\ & x_1 \geq 0 \\ & x_2^+ \geq 0 \\ & x_2^- \geq 0 \\ & \bar{x}_3 \geq 0 \end{aligned}$$

Minimizing problem in standard form

$$\begin{aligned} \min\ & x_1 - 2x_2^+ + 2x_2^- + 3\bar{x}_3 \\ & x_1 + 2x_2^+ - 2x_2^- - 4\bar{x}_3 - x_1^s = 12 \\ & x_1 - x_2^+ + x_2^- - \bar{x}_3 = 2 \\ & x_1 + 2x_2^+ - 2x_2^- - \bar{x}_3 + x_2^s = 14 \\ & x_1 \geq 0 \\ & x_2^+ \geq 0 \\ & x_2^- \geq 0 \\ & \bar{x}_3 \geq 0 \\ & x_1^s \geq 0 \\ & x_2^s \geq 0 \end{aligned}$$

Maximizing problem in inequality form

$$\begin{aligned} \max\ & -x_1 + 2x_2 + 3x_3 \\ & -x_1 - 2x_2 - 4x_3 \leq -12 \\ & -x_1 + x_2 - x_3 \leq -2 \\ & x_1 - x_2 + x_3 \leq 2 \\ & x_1 + 2x_2 + x_3 \leq 14 \\ & -x_1 \leq 0 \\ & x_3 \leq 0 \end{aligned}$$

Figure 3.1: Example for several transformations

3.2 Geometric aspects

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$. We call $\mathbf{Ax} \leq \mathbf{b}$ a *system of linear inequalities* and $\mathbf{Ax} = \mathbf{b}$ a *system of linear equations* for the variables $\mathbf{x} \in \mathbb{R}^n$.

A set of linear inequalities $\mathbf{a}^i\mathbf{x} \leq b_i$, $i \in I$, is said to be *linearly independent* if the vectors of coefficients $\mathbf{a}^i, i \in I$, are linearly independent (analogous for $\geq$ and $=$).

Let $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{b} \in \mathbb{R}^m$ and $B \subseteq \{1, \ldots, m\}$. Define **submatrix, subvector**:

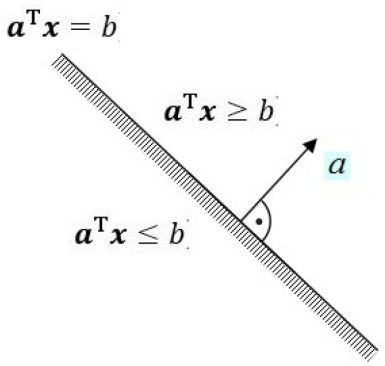
$$\mathbf{A}_B = (\mathbf{a}^i : i \in B) \text{ and } \mathbf{b}_B = (b_i : i \in B)$$

Example:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 5 & 3 \\ 6 & 4 \end{pmatrix} = \begin{pmatrix} \mathbf{a^1} \\ \mathbf{a^2} \\ \mathbf{a^3} \end{pmatrix}, B = \{1, 3\} \text{ then } \mathbf{A}_B = \begin{pmatrix} 1 & 2 \\ 6 & 4 \end{pmatrix} = \begin{pmatrix} \mathbf{a^1} \\ \mathbf{a^3} \end{pmatrix}$$

i.e. extract lines 1 and 3 from $\mathbf{A}$ to form $\mathbf{A}_B$

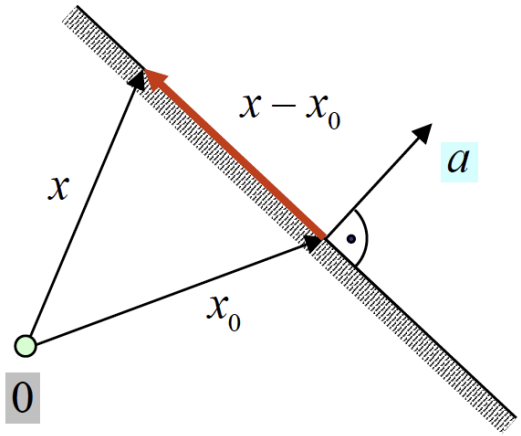
Definition 1. *Let $a \in \mathbb{R}^n$ and $n \in \mathbb{R}$.*
***Halfspace**: $H = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}^T\boldsymbol{x} \leq b\}$ (linear: $b = 0$, affine: b arbitrary)*
***Hyperplane**: $H' = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}^T\boldsymbol{x} = b\}$ (linear: $b = 0$, affine: b arbitrary)*
*H' is **defining hyperplane** of H.*



Remark: The hyperplanes in $\mathbb{R}^n$ are the (n-1)-dimensional affine subspaces ("flats") of $\mathbb{R}^n$.

Vector $\mathbf{a}$ is **normal vector** of hyperplane $H' = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{a}^T\mathbf{x} = b\}$.

Proof: Let $\mathbf{x}^0 \in H'$. For any $\mathbf{x} \in H'$:
$\mathbf{a}^T(\mathbf{x} - \mathbf{x}^0) = \mathbf{a}^T\mathbf{x} - \mathbf{a}^T\mathbf{x}^0 = b - b = 0$.



Definition 2. *A **polyhedron** $P \in \mathbb{R}^n$ is an intersection of a finite number of halfspaces, i.e.*

$$P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\} \quad \text{with } \boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{b} \in \mathbb{R}^m$$

$$\textit{Defining hyperplanes of } P: \quad \{\boldsymbol{x} \in \mathbb{R}^n : \mathbf{a^i x} = b_i\}, i = 1, \ldots, m$$

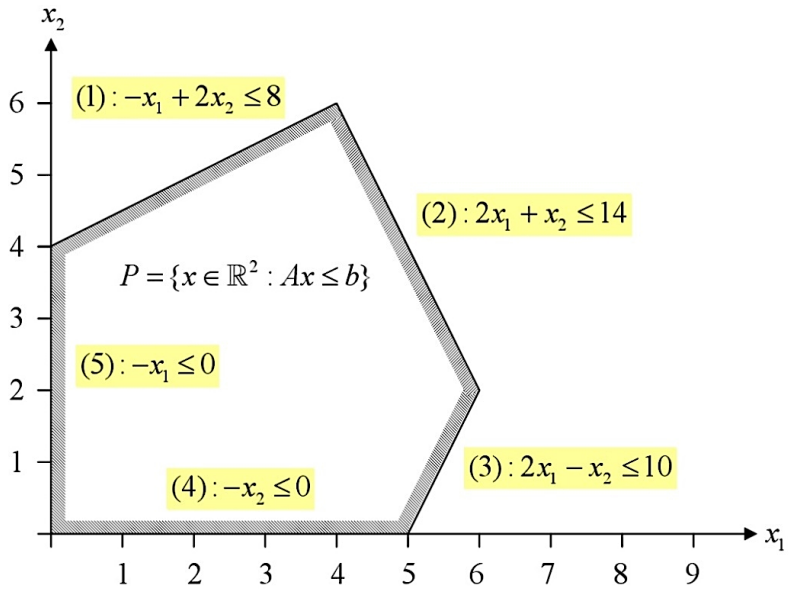


Figure 3.2: Example: Polyhedron in $\mathbb{R}^2$ with five defining hyperplanes

Note: Solution space of a system of linear equalities is also a polyhedron:

$$\mathbf{Ax} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{Ax} \leq \mathbf{b}, \mathbf{Ax} \geq \mathbf{b} \quad \Leftrightarrow \quad \begin{pmatrix} \mathbf{A} \\ \mathbf{-A} \end{pmatrix} \mathbf{x} \leq \begin{pmatrix} \mathbf{b} \\ \mathbf{-b} \end{pmatrix}$$

Definition 3. *A **polytope** in $\mathbb{R}^n$ is a bounded polyhedron (simply it can be put into a box), i.e. a polyhedron $P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ for which there exist $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$ such that:*

$$P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}\}$$

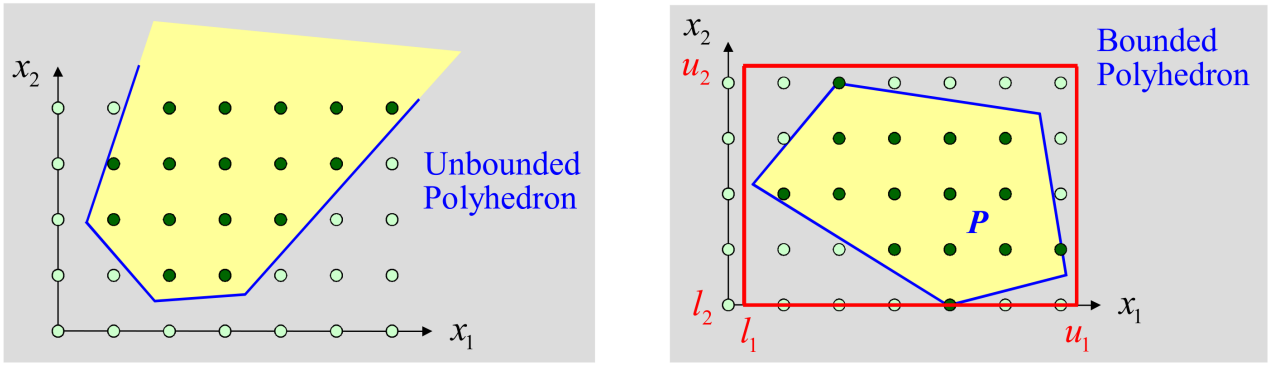


Figure 3.3: Example: Polytope

Proposition 1. *A **polyhedron** is a **convex** set.*

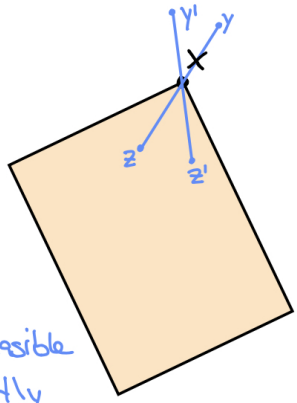
Proof. *A linear function is convex. Therefore, a halfspace of the form $\{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{a}^T\boldsymbol{x} \leq b\}$ is a convex set. Finally, the intersection of a set of halfspaces is convex.*

Definition 4. *A point $\boldsymbol{x} \in \mathbb{R}^n$ is a **vertex** of a polyhedron $P \in \mathbb{R}^n$ if:*

i) $\boldsymbol{x} \in P$

ii) $\boldsymbol{x}$ *is not a strict convex combination of two distinct points* $\boldsymbol{x}^1, \boldsymbol{x}^2 \in P$
(i.e. there exist no $\mathbf{x}^1, \boldsymbol{x}^2 \in P, \boldsymbol{x}^1 \neq \boldsymbol{x}^2$, *such that* $\boldsymbol{x} = \lambda\boldsymbol{x}^1 + (1-\lambda)\boldsymbol{x}^2$ *for some* $0 < \lambda < 1$.*)*

x is a vertex of P if these is no
$y, z \in P$ such that $\exists\, 0 < \lambda < 1$ and
$x = \lambda y + (1-\lambda) z$



We can try in several ways but these is no way to find such two points that are both feasible and such that x is strictly between the two points in the convex sense.

Figure 3.4: Visualization of a vertex

Proposition 2. *$\boldsymbol{x} \in \mathbb{R}^n$ is a **vertex** of a polyhedron $P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ if and only if:*

i) $\boldsymbol{x} \in P$

*ii) $\boldsymbol{x}$ lies on **linearly independent defining hyperplanes** of P*

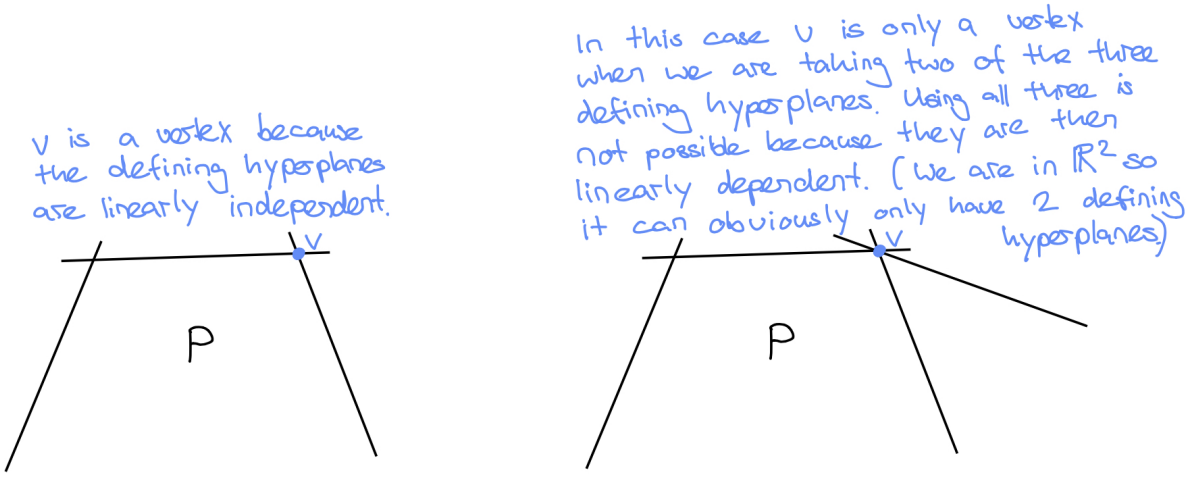


Figure 3.5: Example for linearly independent defining hyperplanes

Corollary 1. *$\boldsymbol{x} \in \mathbb{R}^n$ is a **vertex** of a polyhedron $P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ if:*

i) $\boldsymbol{x} \in P$

ii) there exists selection $B \subseteq 1, \ldots, m$ of $|B| = n$ linearly independent rows of $\boldsymbol{A}$, such that $\boldsymbol{A}_B\boldsymbol{x} = \boldsymbol{b}_B$, or equivalently $\boldsymbol{x} = \boldsymbol{A}_B^{-1}\boldsymbol{b}_B$

Note that $\mathbf{A}_B \subseteq \mathbb{R}^{n \times n}$ has full row (and column) rank and hence, it is non-singular. Consequently, the inverse $\mathbf{A}_B^{-1}$ exists, and the system of equations $\mathbf{A}_B\mathbf{x} = \mathbf{b}_B$ possesses exactly one solution given by $\mathbf{x} = A_B^{-1}\mathbf{b}_B$.

Corollary 1 is an immediate consequence of Proposition 2:The condition that $\mathbf{x}$ is lying on n linearly independent, defining hyperplanes is equivalent to the condition that $\mathbf{x}$ is a solution of the equation system $\mathbf{A}_B\mathbf{x} = \mathbf{b}_B$ where B consists of the indices of the n hyperplanes. The matrix $\mathbf{A}_B$ is non-singular as these hyperplanes are linearly independent, hence $\mathbf{x} = \mathbf{A}_B^{-1}\mathbf{b}_B$ is the only solution of this system. Geometrically, this corresponds to the fact that in $\mathbb{R}^2, n = 2$ linearly independent lines intersect in exactly one point, and analogously in $\mathbb{R}^3, n = 3$ linearly independent planes intersect in exactly one point, and in $\mathbb{R}^n, n$ linearly independent planes intersect in exactly one point.

Corollary 1 describes the vertices of a polyhedron in an "algebraic" way. In general, the following terminology will be used:

Definition 5. *Let $P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ with $\boldsymbol{A} \in \mathbb{R}^{m \times n}$*

*A selection $B \subseteq \{1, .., m\}$ of $|B| = n$ linearly independent rows of $\boldsymbol{A}$ is called a **basic selection**, and the corresponding matrix $\boldsymbol{A}_B$ is called a **basis** of $\boldsymbol{A}$.*

*The vector $\boldsymbol{x} = \boldsymbol{A}_B^{-1}\boldsymbol{b}_B$ is called the **basic solution** associated to basis $\boldsymbol{A}_B$.*

*$B, \boldsymbol{A}_B$, and $\boldsymbol{x} = \boldsymbol{A}_B^{-1}\boldsymbol{b}_B$ are called **feasible** if $\boldsymbol{x} \in P$.*

Theorem 6. *A LP with solution space P always has an optimal solution that is a vertex, as far as*

i) P has any vertices ("P is pointed.")

ii) the optimum is finite

Remember that in the section about convex programming, we stated the following: when maximizing a concave function over a convex solution space, a locally optimal solution is globally optimal. Further, when maximizing a convex function over a convex solution space, all local optimal solutions are located on the boundary of the solution

space. A linear function is both convex and concave, and both mentioned properties apply. Hence, when looking for a global optimum a LP, we can restrict ourselves to the boundary of the solution space and stop the search as soon as a local optimum has been found. Theorem 6 states even more: when looking for a globally optimal solution it is enough to check all vertices of the solution space. The number of vertices of a polyhedron is finite, thus Linear Programming can be interpreted as a discrete optimization problem with a finite solution space (consisting of all the vertices).

Geometric Solution of Linear Programs in $\mathbb{R}^2$ and $\mathbb{R}^3$

In $\mathbb{R}^2$ (and $\mathbb{R}^3$), Linear Programs can be solved "by hand" using elementary geometric instruments (with corresponding approximate accuracy). The idea is that, in $\mathbb{R}^2$, the level sets $L_\gamma = \{\mathbf{x} \in \mathbb{R}^2 : \mathbf{c}^T\mathbf{x} = \gamma\}$ of the objective function $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$ are represented by parallel lines, and the objective value γ grows when shifting these lines in the direction of the normal vector $\mathbf{c}$.

Solving a LP in maximization form means to shift the level line L_γ in a parallel way in direction of the vector $\mathbf{c}$ as far as possible such that it touches the solution space P at the last possible point, i.e. $L_\gamma \cap P \neq \emptyset$ and $L_{\gamma'} \cap P = \emptyset$ for all $\gamma' > \gamma$. This method is illustrated with an example in Figure 3.6.

In $\mathbb{R}^3$, the method corresponds to a parallel shift of level planes.

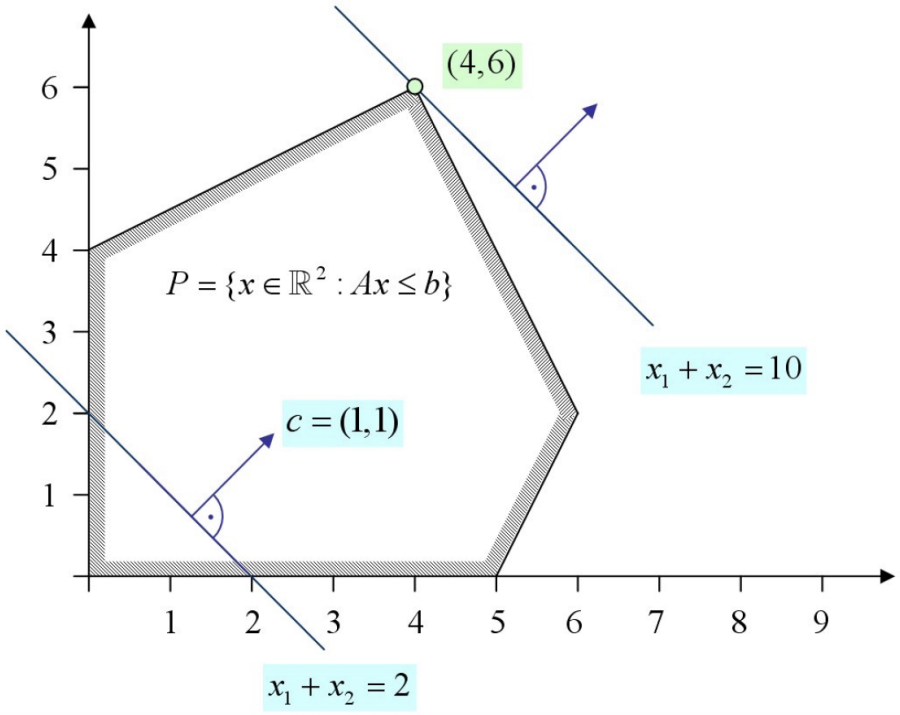


Figure 3.6: Geometric solution of LP's in $\mathbb{R}^2$

3.3 Simplex Algorithm

Consider a linear program of the form

$$\max\{\mathbf{c}^T\mathbf{x} : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$$

where we assume that $\mathbf{A} \in \mathbb{R}^{m\times n}$ has full column rank. In this case the polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ has at least one vertex, according to the above discussion.
The Simplex Algorithm starts at a vertex $\mathbf{v} \in P$, and performs a test whether there exists an edge starting in $\mathbf{v}$, such that going along that edge results in an increase of the objective function value. If there is no such edge, then the current vertex $\mathbf{v}$ is an optimal solution. Otherwise one can - starting at $\mathbf{v}$ - follow such an edge, and in this way, obtain higher objective function values. Either we end up in a neighbouring vertex $\mathbf{v}$', and start over again. Or it is possible to follow this edge forever without leaving the polyhedron P, i.e. the problem is unbounded.

For a formal presentation of the Simplex Algorithm we have to consider the following mathematical questions: How can vertices and edges be described algebraically? How does the objective function value change when moving along an edge? How far can we move along an edge?

3.3.1 Simplex Algorithm in Inequality Form

Given a Linear Program $\prod : \max\{\mathbf{c}^T\mathbf{x} : \mathbf{x} \in P\}$ with $P = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}, \mathbf{A} \in \mathbb{R}^{m\times n}$,
$\operatorname{rank}(\mathbf{A}) = n$, feasible selection B.

1. Calculate the inverse $\bar{\mathbf{A}} = \mathbf{A}_B^{-1}$ and the feasible basic solution $\mathbf{v} = \bar{\mathbf{A}}\mathbf{b}_B$
2. Calculate the reduced costs $\mathbf{u}^T = \mathbf{c}^T\bar{\mathbf{A}}$
3. If $\mathbf{u} \geq \mathbf{0}$ then stop. The vertex $\mathbf{v}$ is optimal
4. Otherwise it is $\mathbf{u} \ngeq \mathbf{0}$. Choose $j \in B$ with $u_j < 0$.
   Define direction $\mathbf{d} = -\bar{\mathbf{A}}_j$
5. Determine $\mathbf{A}\mathbf{d}$
6. If $\mathbf{A}\mathbf{d} \leq \mathbf{0}$ then stop. Linear Program is unbounded along the direction $\mathbf{d}$
7. Otherwise it is $\mathbf{A}\mathbf{d} \nleq \mathbf{0}$. Determine λ^*:

$$\lambda^* = \min\left\{\frac{b_i - \mathbf{a}^i\mathbf{v}}{\mathbf{a}^i\mathbf{d}} : i \in \{1,\ldots,m\}, \mathbf{a}^i\mathbf{d} > 0\right\}$$

8. Let the minimum be attained for index k. Perform basis change:
   $B' = B - \{j\} \cup \{k\}$. Set $B := B'$ and go to the begin.

3.3.2 Example for Simplex Algorithm

We are supposed to solve the following LP $\prod : \max\{\mathbf{c}^T\mathbf{x} : \mathbf{x} \in P\}$ with $P = \{\mathbf{x} \in \mathbb{R}^2 : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$:

$$\begin{aligned} \max\, & 5x_1 + 8x_2 \\ & -3x_1 + 2x_2 \leq 2 \\ & -2x_1 + 5x_2 \leq 16 \\ & 6x_1 + 5x_2 \leq 72 \\ & -x_1 \leq 0 \\ & -x_2 \leq 0 \end{aligned}$$

The input data in matrix notation is given by

$$\mathbf{A} = \begin{pmatrix} -3 & 2 \\ -2 & 5 \\ 6 & 5 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 2 \\ 16 \\ 72 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{c} = (5,8)^T$$

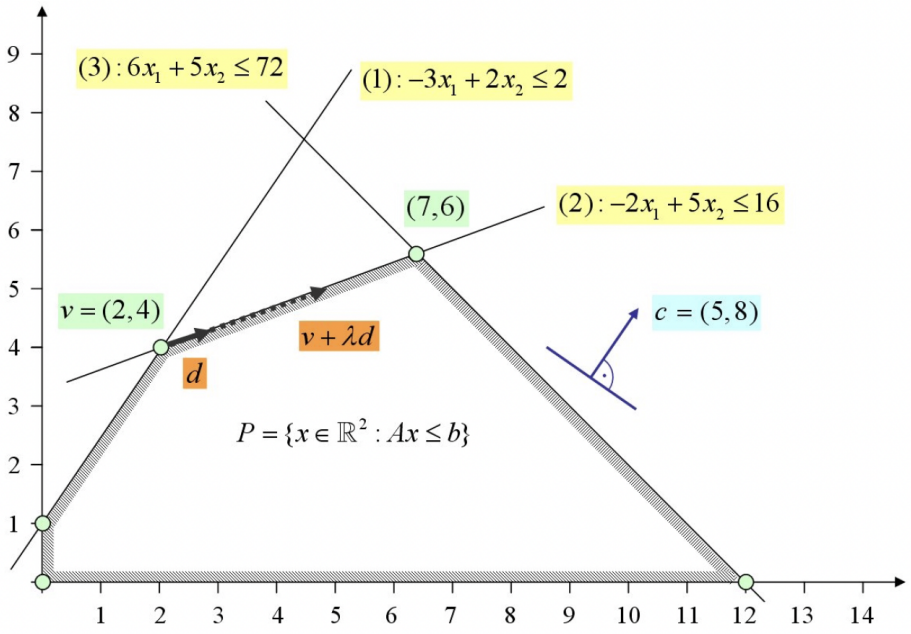


Figure 3.7: Example for Simplex Algorithm

Iteration 1:

1. Basis A_B and corresponding right hand side are given by $A_B = \begin{pmatrix} -3 & 2 \\ -2 & 5 \end{pmatrix}$, $b_B = \begin{pmatrix} 2 \\ 16 \end{pmatrix}$
   Calculation of the basis inverse $\bar{A} = A_B^{-1}$ and the corresponding feasible basic solution $v = \bar{A} b_B$ yields
   $\bar{A} = A_B^{-1} = \begin{pmatrix} -\frac{5}{11} & \frac{2}{11} \\ -\frac{2}{11} & \frac{3}{11} \end{pmatrix}$, $v = \bar{A} b_B = \begin{pmatrix} -\frac{5}{11} & \frac{2}{11} \\ -\frac{2}{11} & \frac{3}{11} \end{pmatrix} \begin{pmatrix} 2 \\ 16 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$
2. The vector $u^T = c^T \bar{A}$ of the "reduced costs" is calculated as
   $u^T = c^T \bar{A} = (5, 8) \begin{pmatrix} -\frac{5}{11} & \frac{2}{11} \\ -\frac{2}{11} & \frac{3}{11} \end{pmatrix} = \left(-\frac{41}{11}, \frac{34}{11} \right)$
3. If $u^T > 0^T$, then stop. In our case, this is not true.
4. We have $u^T \not\geq 0^T$. Choose $j \in B$ such that $u_j < 0$. The first component of u^T is the only possible choice in this case, hence $j = 1$. Define the corresponding direction $d = -\bar{A}_1 = \begin{pmatrix} \frac{5}{11} \\ \frac{2}{11} \end{pmatrix}$
5. Determine λ^* defined as the greatest number $\lambda \in \mathbb{R}_0$ such that $A(v + \lambda d) \leq b$, i.e.
   $A(v + \lambda d) = Av + \lambda Ad = \begin{pmatrix} -3 & 2 \\ -2 & 5 \\ 6 & 5 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \end{pmatrix} + \lambda \begin{pmatrix} -3 & 2 \\ -2 & 5 \\ 6 & 5 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 5/11 \\ 2/11 \end{pmatrix} = \begin{pmatrix} 2 \\ 16 \\ 32 \\ -2 \\ -4 \end{pmatrix} + \lambda \begin{pmatrix} -1 \\ 0 \\ 40/11 \\ -5/11 \\ -2/11 \end{pmatrix} \leq \begin{pmatrix} 2 \\ 16 \\ 72 \\ 0 \\ 0 \end{pmatrix} = b$
6. If $\lambda^* = \infty$, then stop. This is the case if and only if $Ad \leq 0$. As we see this is not true in our case
7. Thus, we get a finik value for $\lambda^* \geq 0$. This value can be determined in the following way:
   $\lambda^* = \min \left\{ \frac{b_i - a^i v}{a^i d} : i \in \{1, \ldots, m\}, a^i d > 0 \right\} = \min \left\{ \frac{b_3 - a^3 v}{a^3 d} \right\} = \min \left\{ \frac{72 - 32}{\frac{40}{11}} \right\} = 11$
   The minimum is attained for $k = 3$. Notice that $3 \notin B$.
8. Form the new feasible basic selection B' as follows:
   $B' = B - \{j\} \cup \{k\} = \{1, 2\} - \{1\} \cup \{3\} = \{2, 3\}$
   The corresponding basic solution v' is given by $v' = v + \lambda^* d = \begin{pmatrix} 2 \\ 4 \end{pmatrix} + 11 \begin{pmatrix} 5/11 \\ 2/11 \end{pmatrix} = \begin{pmatrix} 7 \\ 6 \end{pmatrix}$
   Set $B := B' = \{2, 3\}$, and go to step 1

Figure 3.8: Example for Simplex Algorithm: Iteration 1

Iteration 2:

1. Basis A_B and corresponding right hand side b_B are given by
   $A_B = \begin{pmatrix} -2 & 5 \\ 6 & 5 \end{pmatrix}$, $b_B = \begin{pmatrix} 16 \\ 72 \end{pmatrix}$
   Calculation of the inverse $\bar{A} = A_B^{-1}$ and the corresponding feasible basic solution $v = \bar{A} b_B$ yields
   $\bar{A} = A_B^{-1} = \begin{pmatrix} -\frac{1}{8} & \frac{1}{8} \\ \frac{3}{20} & \frac{1}{20} \end{pmatrix}$, $v = \bar{A} b_B = \begin{pmatrix} -\frac{1}{8} & \frac{1}{8} \\ \frac{3}{20} & \frac{1}{20} \end{pmatrix} \begin{pmatrix} 16 \\ 72 \end{pmatrix} = \begin{pmatrix} 7 \\ 6 \end{pmatrix}$
2. The vector $u^T = c^T \bar{A}$ of the "reduced costs" is calculated as
   $u^T = c^T \bar{A} = (5, 8) \begin{pmatrix} -\frac{1}{8} & \frac{1}{8} \\ \frac{3}{20} & \frac{1}{20} \end{pmatrix} = \left(\frac{23}{40}, \frac{41}{40} \right)$
3. If $u^T \geq 0^T$, then stop. This is true in our case. Hence, we have found the following optimal solution v with corresponding optimal objective value $f(v)$:
   $v = \begin{pmatrix} 7 \\ 6 \end{pmatrix}$, $f(v) = c^T v = (5, 8) \begin{pmatrix} 7 \\ 6 \end{pmatrix} = 83$

Figure 3.9: Example for Simplex Algorithm: Iteration 2

Simplex Example

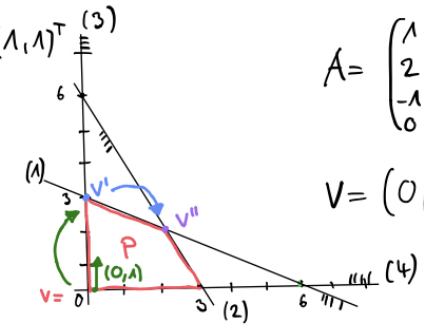
$x_1 + x_2 \to \max \quad c(1,1)^T$

$x_1 + 2x_2 \le 6 \quad (1)$

$2x_1 + x_2 \le 6 \quad (2)$

$-x_1 \le 0 \quad (3)$

$-x_2 \le 0 \quad (4)$



$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}, \quad b = \begin{pmatrix} 6 \\ 6 \\ 0 \\ 0 \end{pmatrix}$$

$V = (0,0)^T \quad B = \{3,4\}$

Iteration 1:

$$A_B = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \quad A_B^{-1} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} = A_B$$

This is just a check to verify

$$v = A_B^{-1} b_B = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

"reduced costs"

$$u^T = c^T \bar{A} = c^T A_B^{-1} = (1,1) \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} = (-1, -1)$$

choose second component

-1 ($j = 3$), -1 ($j = 4$)

$$d = -\bar{A}_j = -\bar{A}_4 = -\begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad d = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$Av' \le b = A(v + \lambda d) \le b = \underbrace{\begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}}_{A} \left[\underbrace{\begin{pmatrix} 0 \\ 0 \end{pmatrix}}_{v} + \lambda \underbrace{\begin{pmatrix} 0 \\ 1 \end{pmatrix}}_{d} \right] \le \begin{pmatrix} 6 \\ 6 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ \lambda \end{pmatrix} \le \begin{pmatrix} 6 \\ 6 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 0 \\ -1 \end{pmatrix} \lambda \le \begin{pmatrix} 6 \\ 6 \\ 0 \\ 0 \end{pmatrix}$$

Ad

we have $\lambda \le \frac{6}{2}$ and $\lambda \le 6$

$\lambda^* = \min\{3, 6\} = 3$

$B' = B - \{j\} \cup \{k\} = \{3,4\} - \{4\} \cup \{1\} = \{1,3\}$

Iteration 2:

$$A_B = \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 6 \\ 0 \end{pmatrix} \quad \bar{A} = A_B^{-1} = \begin{pmatrix} 0 & -1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \quad v = \bar{A} b_B = \begin{pmatrix} 0 & -1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 6 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$$

$$u^T = c^T \bar{A} = c^T A_B^{-1} = (1,1) \begin{pmatrix} 0 & -1 \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} = (0.5, -0.5)$$

1 ($j = 1$), -0.5 ($j = 3$)

$$d = -\bar{A}_3 = -\begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$Av + \lambda Ad = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \left[\begin{pmatrix} 0 \\ 3 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right] \le \begin{pmatrix} 6 \\ 6 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 0 \\ -3 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 2 \\ -1 \\ 0 \end{pmatrix} \le \begin{pmatrix} 6 \\ 6 \\ 0 \\ 0 \end{pmatrix}$$

we have $\lambda \le 3$ and $\lambda \le 2$

$\lambda^* = \min\{3, 2\} = 2$

The minimum is attained for $k = 2$

$B' = B - \{j\} \cup \{k\} = \{1,3\} - \{3\} \cup \{2\} = \{1,2\}$

Iteration 3:

$$A_B = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 6 \\ 6 \end{pmatrix} \quad \bar{A} = A_B^{-1} = \begin{pmatrix} \frac{1}{3} & -\frac{2}{3} \\ -\frac{2}{3} & \frac{1}{3} \end{pmatrix} \quad v = \bar{A} b_B = \begin{pmatrix} -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \end{pmatrix} \begin{pmatrix} 6 \\ 6 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

$$u^T = c^T \bar{A} = c^T A_B^{-1} = (1,1) \begin{pmatrix} -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \end{pmatrix} = \left(\frac{1}{3}, \frac{1}{3}\right)$$

$u^T \ge 0^T$ is true. We can stop.

$$v = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \quad f(v) = c^T v = (1,1) \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \underline{\underline{4}}$$

Chapter 4

Integer Linear Programming

Integer Linear Programming:

$$\max\{c^T x : x \in P \cap \mathbb{Z}^n\} \text{ with } P = \{x \in \mathbb{R}^n : Ax \leq b\}$$

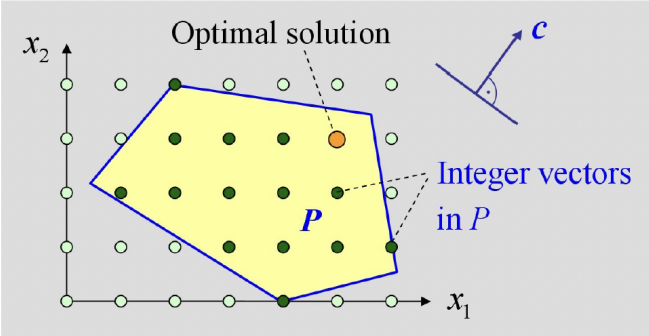


Figure 4.1: Integer Linear Programming

Binary Linear Programming:

$$\max\{c^T x : x \in P \cap \{0,1\}^n\} \text{ with } P = \{x \in \mathbb{R}^n : Ax \leq b\}$$

Mixed Integer Linear Programming:

$$\max\{c^T x : x \in P \cap \mathbb{Z}_K^n\} \text{ with } P = \{x \in \mathbb{R}^n : Ax \leq b\}$$

where $K \subseteq \{1, \ldots, n\}$ and $\mathbb{Z}_K^n = \{x \in \mathbb{R}^n : x_j \in \mathbb{Z} \text{ for } j \in K\}$

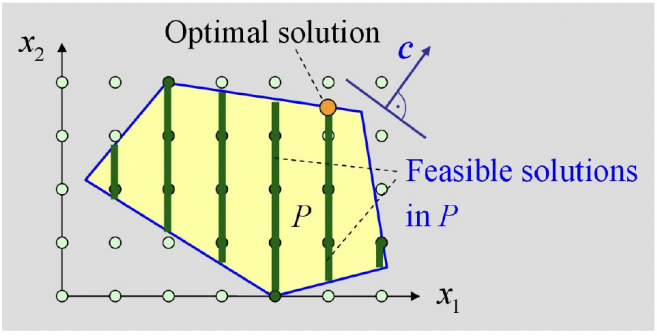


Figure 4.2: Mixed Integer Programming

4.1 Branch-and-Bound Method

4.1.1 General example

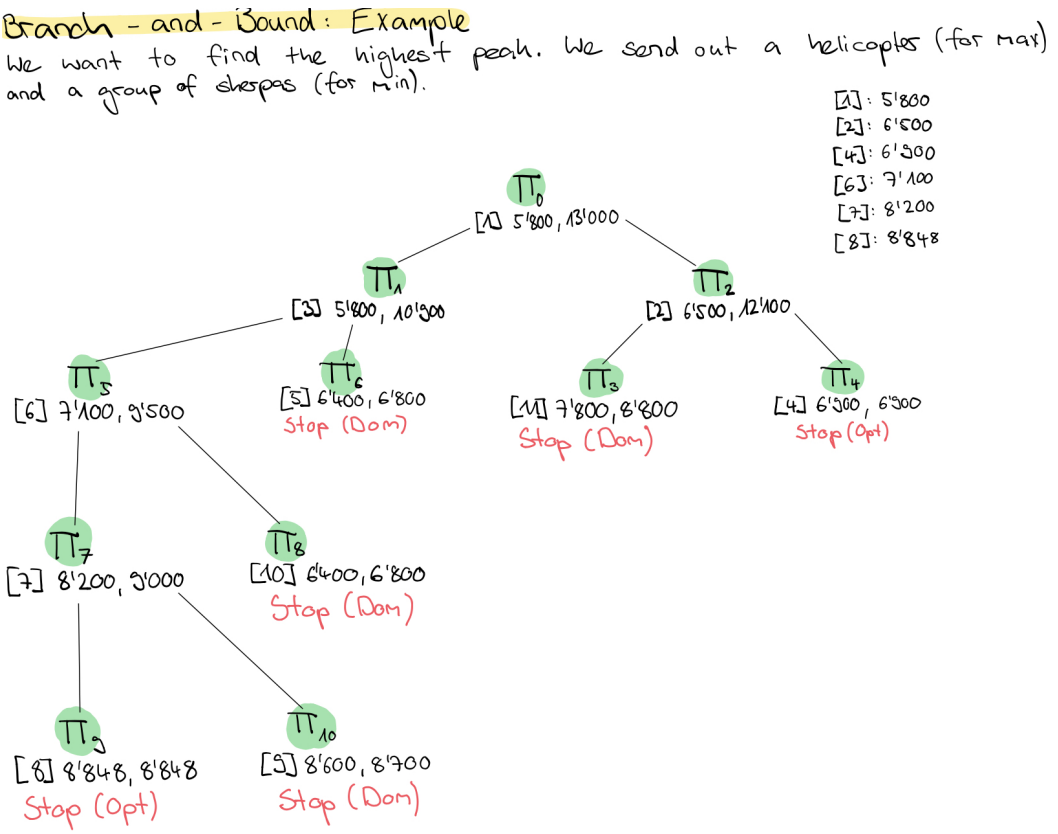


Figure 4.3: Example general branch-and-bound method

4.1.2 Branch-and-Bound Method for ILP

- Calculation of upper bound ("helicopters"): **LP-Relaxation**
- Branching:
  - If LP solution is all integer: STOP. Current node optimally solved.
  - Otherwise: Choose some fractional variable and **round up/down**

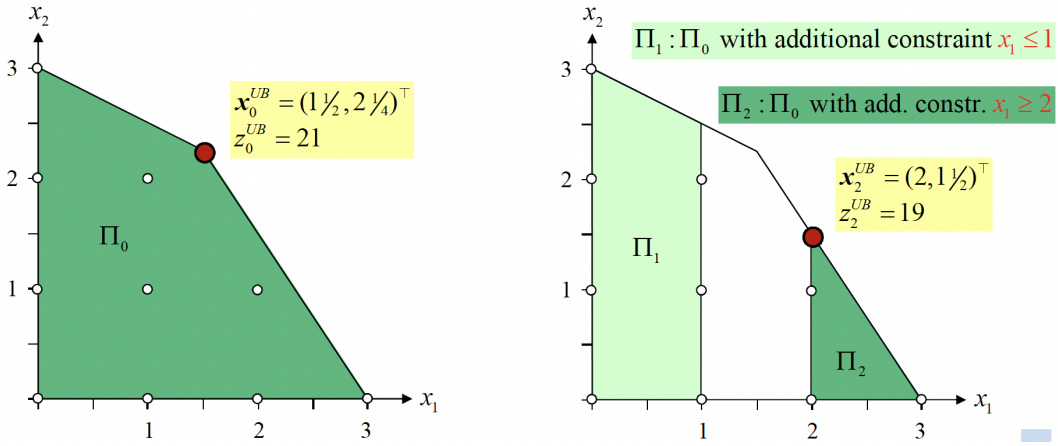


Figure 4.4: Branch-and-bound method for ILP

Algorithm for Branch-and-Bound Method for ILP

Given: ILP $\Pi : \max\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in P \cap \mathbb{Z}^n\}$ with $P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$.

1. Initialization: $\hat{r} := 0, P^0 := P, S^0 := P \cap \mathbb{Z}^n$,
   $\Pi^0 := \max\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in S^0\}, R := \{0\}, z^{LB} := -\infty$
2. Termination: If $R = \emptyset$ then stop.
3. Node selection: Choose a node $r \in R$ and set $R := R - \{r\}$.
4. Bound calculation and heuristic solution:
   (i) Calculate upper bound z_r^{UB} for Π^r by solving the LP relaxation
   $\Pi_{LP}^r : \max\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in P^r\}$: optimum z^{LP} and optimal solution $\boldsymbol{x}^{LP}$.
   - If Π_{LP}^r infeasible then $z_r^{UB} := \infty$, and go to Step 6.
   - Otherwise $z_r^{UB} := z^{LP}$.

   (ii) Search a feasible solution $\boldsymbol{x}_r^{LB}$ of Π^r :
   - If $\boldsymbol{x}^{LP} \in \mathbb{Z}^n$ then $\boldsymbol{x}_r^{LB} := \boldsymbol{x}^{LP}$ is optimal for Π^r, set $z_r^{LB} := \boldsymbol{c}^T\boldsymbol{x}^{LP}$.
   - Otherwise $z_r^{LB} := -\infty$.
5. Global improvement: If $z_r^{LB} > z^{LB}$ then set $z^{LB} := z_r^{LB}$ and $\boldsymbol{x}^{LB} := \boldsymbol{x}_r^{LB}$.
6. Pruning: Go to Step 2 if one of the following cases appears:
   - $z_r^{UB} = \infty$ (Infeasibility)
   - $z_r^{LB} = z_r^{UB}$ (Optimality)
   - $z_r^{UB} \leq z^{LB}$ (Dominance)
7. Branching: Part the set S^r in $S^{\hat{r}+1}$ and $S^{\hat{r}+2}$. Find j with $x_j^{LP} \notin \mathbb{Z}$. Set:
   $P^{\hat{r}+1} := \{\boldsymbol{x} \in P^{\hat{r}} : x_j \leq \lfloor x_j^{LP} \rfloor\}, \qquad P^{\hat{r}+2} := \{\boldsymbol{x} \in P^{\hat{r}} : x_j \geq \lfloor x_j^{LP} \rfloor + 1\}$
   Set $S^q := P^q \cap \mathbb{Z}^n$ for $q \in R^r := \{\hat{r}+1, \hat{r}+2\}$
   $\longrightarrow$ Consider $\Pi^q : \max\{\boldsymbol{c}^T\boldsymbol{x} : \boldsymbol{x} \in S^q\}, q \in R^r$. Set $R := R \cup R^r$, $\hat{r} := \hat{r} + 2$.

Figure 4.5: Branch-and-bound method for ILP - Algorithm

4.2 Knapsack Problem

Sets:

J Set of items, $J = \{1, 2, \ldots, n\}$

Parameters:

a_j Weight of item j, $j \in J$

b Capacity of the rucksack

c_j Value of item j, $j \in J$

Variables:

x_j Binary indicator with value 1 if item j put into the rucksack $j \in J$

Objective functions and constraints:

$$\max \sum_{j \in J} c_j x_j$$

$$\sum_{j \in J} a_j x_j \leq b$$

$$x_j \in \{0, 1\} \qquad j \in J$$

4.2.1 1. LP relaxation for the Knapsack Problem

$$\begin{aligned} \max \sum_{j \in J} c_j x_j & \\ \sum_{j \in J} a_j x_j & \leq b \\ 0 \leq x_j \leq 1 & \qquad j \in J \end{aligned}$$

For the Knapsack Problem, the LP relaxation can be solved easily:

- For every item $j \in J$, determine the utility coefficient $\frac{c_j}{a_j}$
- Sort all items decreasingly with regard to the utility coefficient, i.e. for the new order $J = \{1, \ldots, n\}$ we have

$$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \ldots \geq \frac{c_n}{a_n}$$

- Determine $k = \max\{k' : \sum_{j=1}^{k'} a_j \leq b\}$
- The optimal solution is given by

$$x_j^{LP} = \begin{cases} 1 & \text{for } j = 1, \ldots, k \\ (b - \sum_{j'=1}^{k} a_{j'})/a_j & \text{for } j = k+1 \\ 0 & \text{for } j = k+2, \ldots, n \end{cases}$$

Figure 4.6: LP relaxation for the Knapsack Problem

4.2.2 2. Heuristic search for feasible solutions

$$x_j^{LB} = \begin{cases} 1 & \text{for } j = 1, \ldots, k \\ 0 & \text{for } j = k+1, \ldots, n \end{cases}$$

Figure 4.7: Heuristic search for feasible solutions

4.2.3 3. Definition of the sub-problems

- Sub-problems defined by sets $J_r^0, J_r^1 \subseteq J$:

$$x_j = \begin{cases} 0 & \text{for } j \in J_r^0 \\ 1 & \text{for } j \in J_r^1 \\ \text{not fixed} & \text{for } j \in J_r := J - J_r^0 - J_r^1 \end{cases}$$

- Sub-Problem Π^r given by:

$$\Pi^r : \max \left\{ \sum_{j \in J_r} c_j x_j : \sum_{j \in J_r} a_j x_j \leq b - \sum_{j \in J_r^1} a_j x_j \right\}$$

Figure 4.8: Definition of the sub-problems

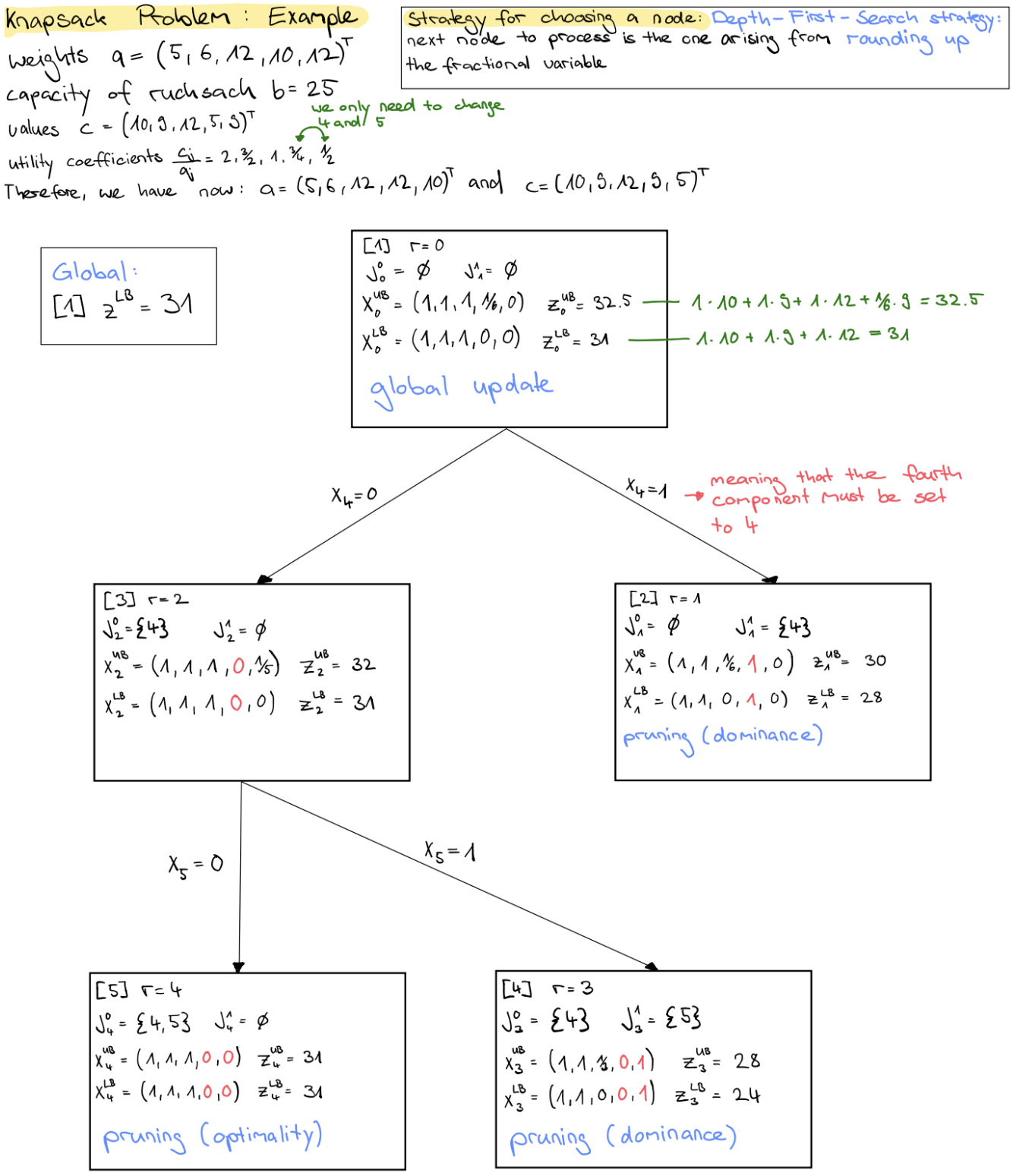


Figure 4.9: Example of a Knapsack Problem

4.3 Cutting Plane Method

Definition 6. ***Convex Hull***

The convex hull of $S \subseteq \mathbb{R}^n$ *is defines as*
$conv(S) = \{x \in \mathbb{R}^n : x = \sum_{i=1}^{k} \lambda_i x^i \text{ with } x^i \in S, \lambda_i \geq 0, i = 1, \ldots, k, \sum_{i=1}^{k} \lambda_i = 1\}$

Geometrically: $conv(\{x^1, x^2\})$: line segment joining x^1 and x^2.

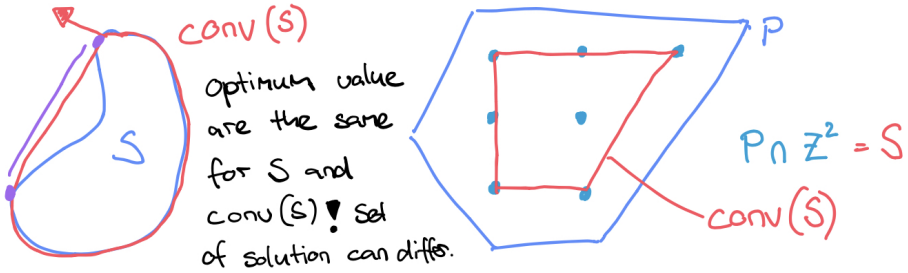


Figure 4.10: Example Convex Hull

Theorem 7. ***Optimization of Linear Objective over Convex Hull***

Let $S \subseteq \mathbb{R}^n$. *Then* $\max\{c^T x : x \in S\} = \max\{c^T x : x \in conv(S)\}$

Theorem 8. ***Integer Hull***

(i) If $P \subseteq \mathbb{R}^n$ *is a rational polyhedron then* $P_{\mathbb{Z}}$ *is a rational polyhedron.*
(ii) All vertices of $P_{\mathbb{Z}}$ *are integer*

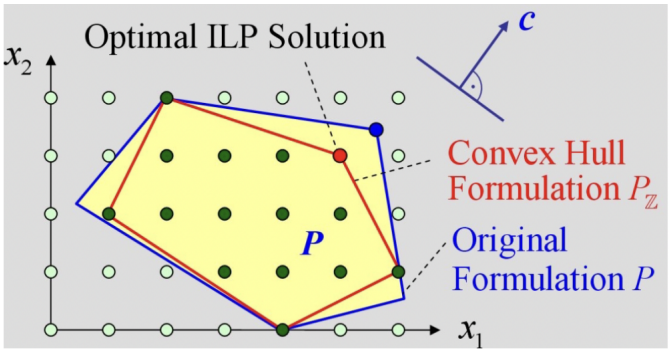


Figure 4.11: Optimal ILP Solution

4.3.1 Cutting Plane Method

Definition 7. ***Cutting Plane (Cut)***

A cutting plane for a polyhedron P is a valid inequality for $P_{\mathbb{Z}}$.

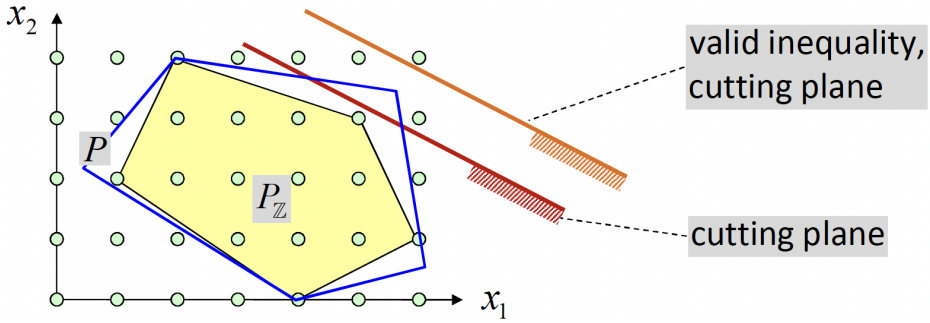
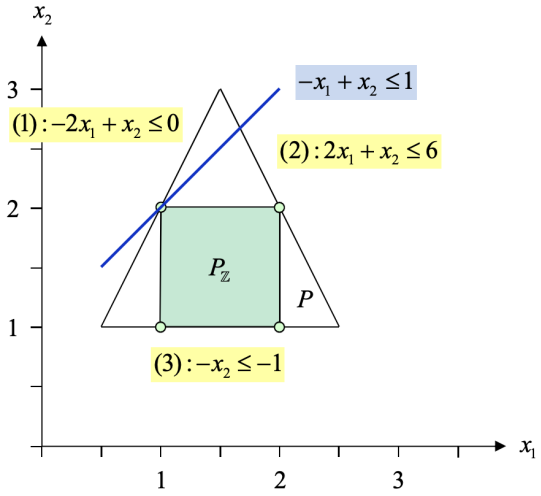


Figure 4.12: Cutting Plane

4.3.2 Gomory-Chvatal Cuts



- Combine inequalities: $\frac{3}{4} \cdot (1) + \frac{1}{4} \cdot (2)$
- $(-\frac{3}{4} \cdot 2x_1 + \frac{3}{4} x_2) +$
  $(\frac{1}{4} \cdot 2x_1 + \frac{1}{4} x_2) \le \frac{3}{4} \cdot 0 + \frac{1}{4} \cdot 6$
  $\Leftrightarrow -x_1 + x_2 \le 1\frac{1}{2}$
- Hence, right side can be rounded down.
- Resulting new inequality (G-C-Cut):
  $-x_1 + x_2 \le 1$

Matrix notation:

- $\boldsymbol{A} = \begin{pmatrix} -2 & 1 \\ 2 & 1 \\ 0 & -1 \end{pmatrix}$ $\boldsymbol{b} = \begin{pmatrix} 0 \\ 6 \\ -1 \end{pmatrix}$
- $\boldsymbol{u} = (\frac{3}{4}, \frac{1}{4}, 0)^\top \ge \boldsymbol{0}$
- $\boldsymbol{\alpha} := \boldsymbol{u}^\top \boldsymbol{A} = (-1, 1)^\top$
- $\beta := \boldsymbol{u}^\top \boldsymbol{b} = 1\frac{1}{2}$

- G-C-Cut: $\boldsymbol{\alpha}^\top \boldsymbol{x} \le \lfloor \beta \rfloor$

Notation:
$\lfloor \beta \rfloor$ "floor":
largest integer $\le \beta$

Figure 4.13: Gomory-Chvatal Cuts

Chapter 5

Unconstrained continuous optimization

5.1 Gradient Descent

Idea: The function increases in direction of the gradient, so by going in the opposite direction we will get decreasing function values.

Funktion $f(x)$	**Ableitung** $f'(x)$	**Bemerkung, Regel**
x	1	
x^2	$2x$	
x^3	$3x^2$	
x^n	$n \cdot x^{n-1}$	$n \in \mathbb{R}$
$\frac{1}{x}$	$-\frac{1}{x^2}$	
$\sqrt{x}$	$\frac{1}{2\sqrt{x}}$	
$\sin x$	$\cos x$	
$\cos x$	$-\sin x$	
$\tan x$	$1 + \tan^2 x = \frac{1}{\cos^2 x}$	
e^x	e^x	e ist die Eulersche Zahl
a^x	$\ln a \cdot a^x$	
$\ln x$	$\frac{1}{x}$	
c	0	Konstantenregel
$g(x) + h(x)$	$g'(x) + h'(x)$	Summenregel
$k \cdot g(x)$	$k \cdot g'(x)$	Faktorregel
$g(x) \cdot h(x)$	$g(x) \cdot h'(x) + g'(x) \cdot h(x)$	Produktregel
$(g(x))^n$	$n \cdot (g(x))^{n-1} \cdot g'(x)$	Potenzregel
$\frac{g(x)}{h(x)}$	$\frac{h(x) \cdot g'(x) - g(x) \cdot h'(x)}{(h(x))^2}$	Quotientenregel

Figure 5.1: Ableitungsregeln

Hessian matrix

$$\begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} & \cdots \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} & \cdots \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Task 1 $\quad f(x,y) := (x-2)^4 + (x-2y)^2$

a) $\frac{\partial f}{\partial x}(x,y) = 4(x-2)^3 + 2(x-2y)$

$\frac{\partial f}{\partial y}(x,y) = -4(x-2y)$

$\frac{\partial^2 f}{\partial x^2}(x,y) = 12(x-2)^2 + 2$

$\frac{\partial^2 f}{\partial x \partial y}(x,y) = -4$

$\frac{\partial^2 f}{\partial y^2}(x,y) = 8$

$\frac{\partial^2 f}{\partial y \partial x}(x,y) = -4$

Are the same. According to Schwarz's theorem it does not matter whether we first differentiate wrt. x and then wrt. y or vice versa.

b) $\nabla f(0,0) = \begin{bmatrix} \frac{\partial f}{\partial x}(0,0) \\ \frac{\partial f}{\partial y}(0,0) \end{bmatrix} = \begin{bmatrix} -32 \\ 0 \end{bmatrix}$

$H_f(0,0) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2}(0,0) & \frac{\partial^2 f}{\partial x \partial y}(0,0) \\ \frac{\partial^2 f}{\partial y^2}(0,0) & \frac{\partial^2 f}{\partial y \partial x}(0,0) \end{bmatrix} = \begin{bmatrix} 50 & -4 \\ -4 & 8 \end{bmatrix}$

Figure 5.2: Example Gradient calculations

5.1.1 Algorithm:

- *Initialization:* Choose the starting point x^0.
- *Iteration step* $x^i \Rightarrow x^{i+1}$:
  Determine local gradient $\nabla f(x^i)$ and move by some amount β in opposite direction $\Rightarrow$ new point x^{i+1}.

Repeat until gradient is (approximately) zero/no more (significant) decreases of the function values are observed.

5.1.2 Step size

What is the optimal **step size** β, i.e. how far should we go in direction of $-\nabla f(x^i)$ in one step?

$$x^{i+1} = x^i - \beta \nabla f(x^i)$$

Rule 1: Successive halving of the step size
Set $\beta := 1$ and check whether

$$f(x^i - \beta \nabla f(x^i)) < f(x^i)$$

If (1) is **not satisfied**, set $\beta := \beta/2$ and check again. Do this until condition (1) is satisfied.

If (1) is **satisfied**, repeat doubling $\beta := 2 * \beta$ while function values $f(x^i - \beta \nabla f(x^i))$ are decreasing.

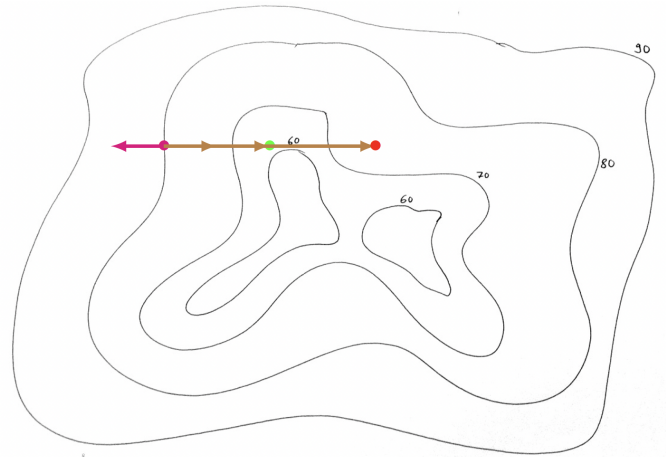


Figure 5.3: Visualization of Rule 1

Task 2 $f(x,y) := (x-2)^4 + (x-2y)^2$

Start from point $x^0 = (0,0)$

a) Gradient descent with successive halving of the step size

At point $(x_0, y_0) = (0,0)$ we have $f(0,0) = 16$ — $(0-2)^4 + (0-2\cdot 0)^2 = (-2)^4 = 16$

$\nabla f(0,0) = \begin{bmatrix} -32 \\ 0 \end{bmatrix}$

β	$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \beta \cdot \nabla f(x_0, y_0) = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \beta \begin{pmatrix} -32 \\ 0 \end{pmatrix}$	$f(x_1, y_1)$
1	$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - 1 \begin{pmatrix} -32 \\ 0 \end{pmatrix} = \begin{pmatrix} 32 \\ 0 \end{pmatrix}$	811'024
1/2	$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 8 \\ \end{pmatrix} - \frac{1}{2} \begin{pmatrix} -32 \\ 0 \end{pmatrix} = \begin{pmatrix} 16 \\ 0 \end{pmatrix}$	38'672
1/4	$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 8 \\ \end{pmatrix} - \frac{1}{4} \begin{pmatrix} -32 \\ 0 \end{pmatrix} = \begin{pmatrix} 8 \\ 0 \end{pmatrix}$	1'360
1/8	$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \frac{1}{8} \begin{pmatrix} -32 \\ 0 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \end{pmatrix}$	32
1/16	$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \frac{1}{16} \begin{pmatrix} -32 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$	$4 < 16$ done!

The next iteration point therefore is $x^1 = (x_1, y_1) = (2,0)$ (when only using successive halving)

Figure 5.4: Example of Rule 1

Rule 2: Successive halving of the step size with subsequent parabola fitting
Improvement of β after applying Rule 1: Approximate f near x^i in direction of $-\nabla f(x^i)$ by a quadratic parabola. Consider choosing x^{i+1} according to the minimum of the parabola.

Compute the parabola $P(t) = at^2 + bt + c$ such that

$$P(0) = f(x^i)$$
$$P(\beta) = f(x^i - \beta \nabla f(x^i))$$
$$P(2\beta) = f(x^i - 2\beta \nabla f(x^i))$$

Inserting values in $P(t) = at^2 + bt + c$ gives

$$P(0) = c = f(x^i)$$
$$P(\beta) = a\beta^2 + b\beta + c = f(x^i - \beta \nabla f(x^i))$$
$$P(2\beta) = 4a\beta^2 + 2b\beta + c = f(x^i - 2\beta \nabla f(x^i))$$

$$a = \frac{1}{2\beta^2}\left[P(0) - 2 \cdot P(\beta) + P(2\beta)\right]$$

$$b \quad \frac{1}{2\beta}\left[-3P(0) + 4 \cdot P(\beta) - P(2\beta)\right]$$

$$\beta^* = \frac{-b}{2a} = \frac{\beta}{2} \cdot \frac{3 \cdot P(0) - 4P(\beta) + P(2\beta)}{P(0) - 2P(\beta) + P(2\beta)}$$

Figure 5.5: Rule 2

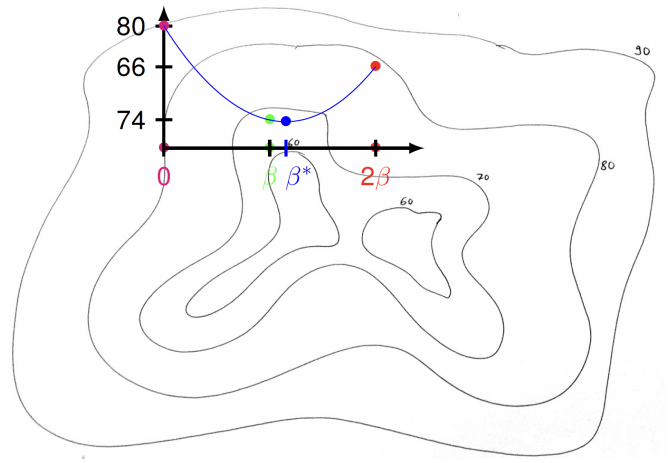


Figure 5.6: Visualization of Rule 2

b) Using the result from a), we first fit a parabola $P(t) = at^2 + bt + c$ through the following three sample points. (Recall from a) that after the successive halving phase we have $\beta = 1/16$

	t	$P(t) \stackrel{!}{=} f\left(\binom{0}{0} - t\binom{-32}{0}\right)$
$P(0)$	0	$P(0) \stackrel{!}{=} f\left(\binom{0}{0} - 0\binom{-32}{0}\right) = f\binom{0}{0} = 16$
$P(\beta)$	$1/16$	$P(1/16) \stackrel{!}{=} f\left(\binom{0}{0} - 1/16\binom{-32}{0}\right) = f\binom{2}{0} = 4$
$P(2\beta)$	$1/8$	$P(1/8) \stackrel{!}{=} f\left(\binom{0}{0} - 1/8\binom{-32}{0}\right) = f\binom{4}{0} = 32$

$$\Rightarrow \beta^* = \frac{-b}{2a} = \frac{\beta}{2} \cdot \frac{3 \cdot P(0) - 4 \cdot P(\beta) + P(2\beta)}{P(0) - 2 \cdot P(\beta) + P(2\beta)}$$

$$= \frac{1/16}{2} \cdot \frac{3 \cdot 16 - 4 \cdot 4 + 32}{16 - 2 \cdot 4 + 32} = \frac{1/16}{2} \cdot \frac{64}{40} = 0.05 = \beta^*$$

Therefore, we considers

$\binom{x_1}{y_1} = \binom{0}{0} - 1/20\binom{-32}{0} = \binom{8/5}{0} = \binom{1.6}{0}$ as our next iteration point. But first we check the resulting function value

For $\beta = 1/16$ we got $f(2,0) = 4$, for $\beta^* = 1/20$ we get $f(1.6, 0) = 2.5856\ldots$ which is better. The next iteration point therefore is $x^1 = (x_1, y_1) = (1.6, 0)$

Figure 5.7: Example of Rule 2

5.2 Newton's method

For a multidimensional function $f : \mathbb{R}^n \to \mathbb{R}$:

$$x^{i+1} = x^i - (H_f(x^i))^{-1} * \nabla f(x^i)$$

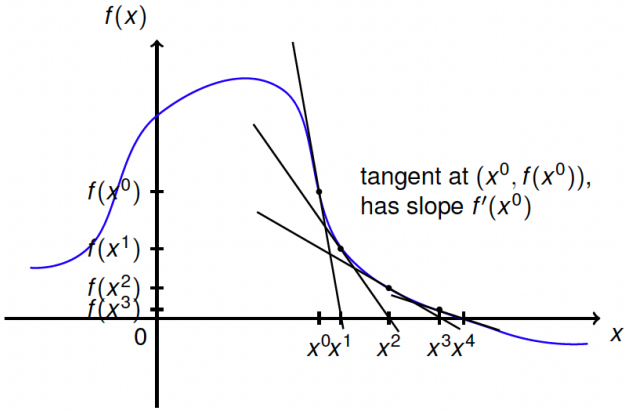


Figure 5.8: Visualization Newton's method

5.2.1 Algorithm

- *Initialization:* Choose starting point x^0.
- *Iteration step* $x^i \Rightarrow x^{i+1}$:
  $x^{i+1} = x^i - (H_f(x^i))^{-1} * \nabla f(x^i)$

Repeat iteration step until break condition is met.

5.2.2 Example

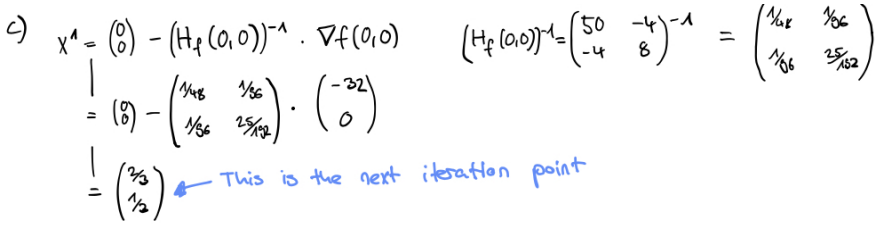


Figure 5.9: Example of Newton's method

5.3 Speed of convergence

Linear convergence (Gradient descent):
There is $c \in (0,1)$ and $i_0 \in \mathbb{N}$ such that for all $i \geq i_0$ we have

$$\|x^* - x^{i+1}\| \leq c\|x^* - x^i\|$$

Superlinear convergence (Broyden's method):
There is a sequence $\{c_i\}_{i\in\mathbb{N}}$ with $\lim_{n\to\infty} c_i = 0$ such that

$$\|x^* - x^{i+1}\| \leq c_i\|x^* - x^i\|$$

(Slightly stronger than linear convergence: Now the factor c itself tends towards 0.)

Quadratic convergence (Newton's method):
There is $c > 0$ and $i_0 \in \mathbb{N}$ such that for all $i \geq i_0$ we have

$$\|x^* - x^{i+1}\| \leq c\|x^* - x^i\|^2$$

(The precision (number of correct digits) roughly doubles in each step!)

5.4 Broyden's method

5.4.1 Algorithm

- *Initialization:* Start with x^0.
  Compute $\nabla f(x^0)$, the inverse of the Hessian matrix $H_f(x^0)$, and set $(A^0)^{-1} := (H_f(x^0))^{-1}$. Use this to compute
  $$x^1 = x^0 - (A^0)^{-1}\nabla f(x^0)$$
- *Iteration step:* To compute x^{i+1} from x^i, compute first $\nabla f(x^i), g^i = \nabla f(x^i) - \nabla f(x^{i-1})$ and $d^i = x^i - x^{i-1}$, then
  $$(A^i)^{-1} = (A^{i-1})^{-1} - \frac{((A^{i-1})^{-1}g^i - d^i)(d^i)^T(A^{i-1})^{-1}}{(d^i)^T(A^{i-1})^{-1}g^i}$$
  and set
  $$x^{i+1} = x^i - (A^i)^{-1}\nabla f(x^i)$$

After the first step, we never deal with the Hessian explicitly and this leads to a speed up from $O(n^3)$ to $O(n^2)$.

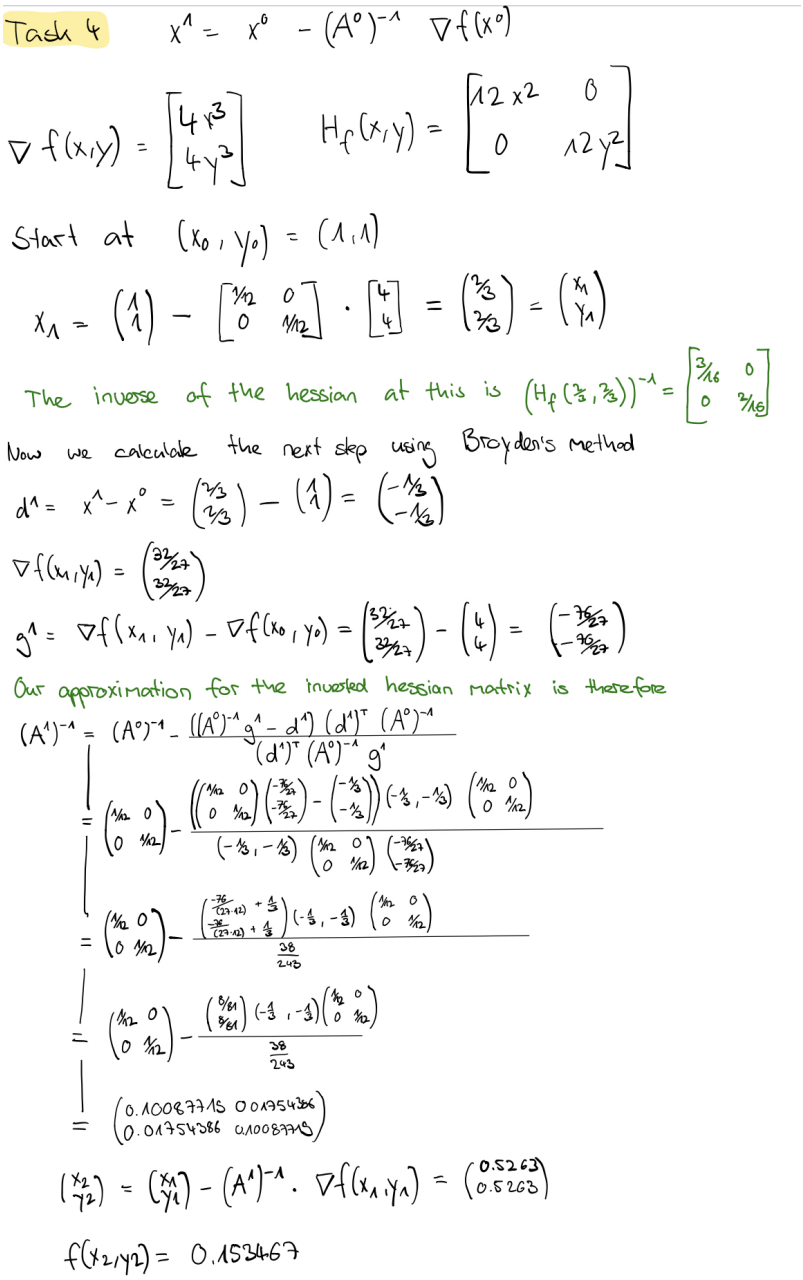


Figure 5.10: Example of Broyden's method

5.5 Aitken's acceleration method

i	x^i	Aitken y^i
0	4	-
1	2.666667	-
2	3.466667	3.166667
3	2.895238	3.133333
4	3.339683	3.145238
5	2.976046	3.139683
6	3.283738	3.142713

$$y^i = x^i - \frac{(x^i - x^{i-1})^2}{x^i - 2x^{i-1} + x^{i-2}}$$

Figure 5.11: Aitken's acceleration method

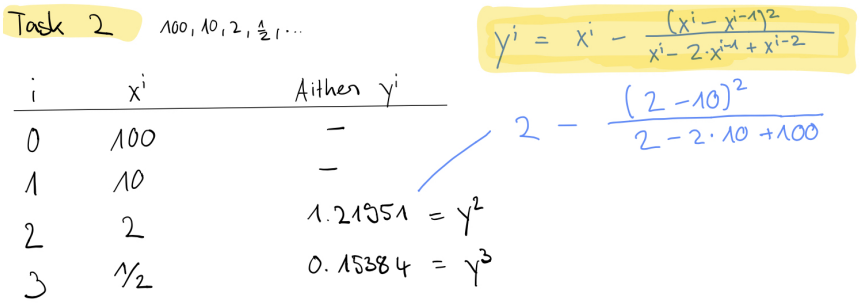


Figure 5.12: Example of Aitken's acceleration method

Chapter 6

Graph and network optimization

6.1 Graphs

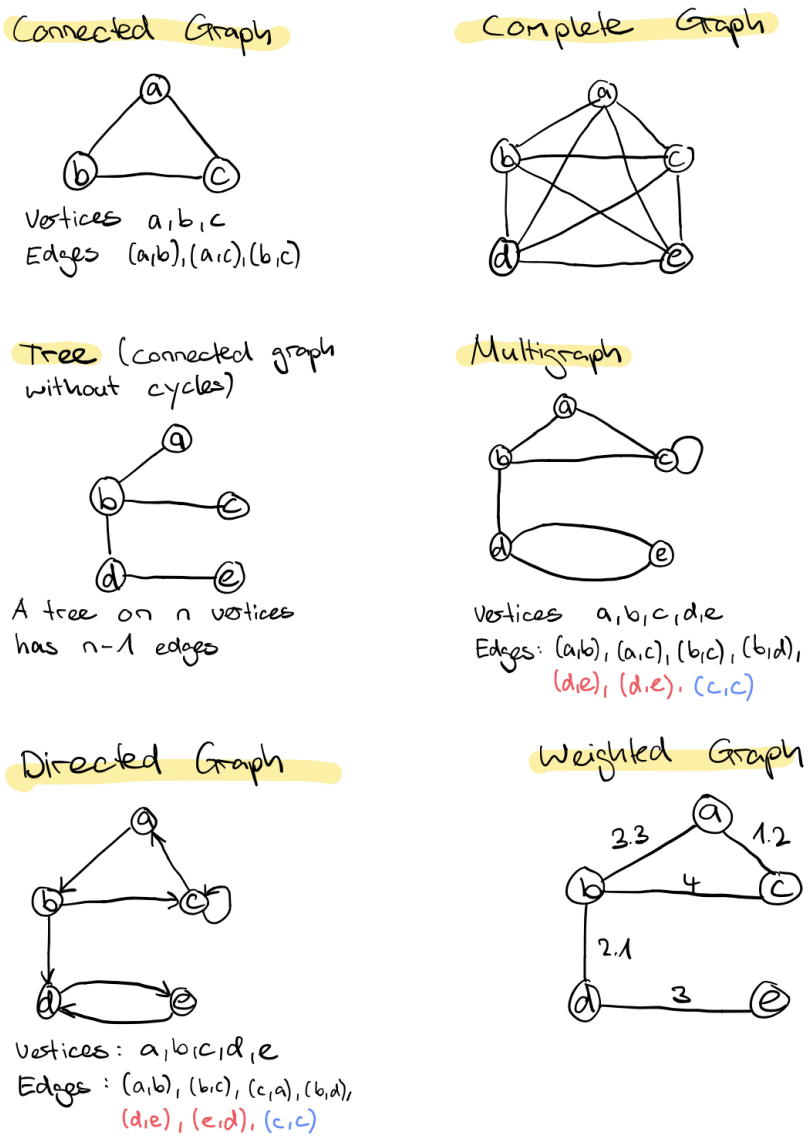


Figure 6.1: Graph basics

6.2 Graph Search

6.2.1 Depth-First Search (DFS)

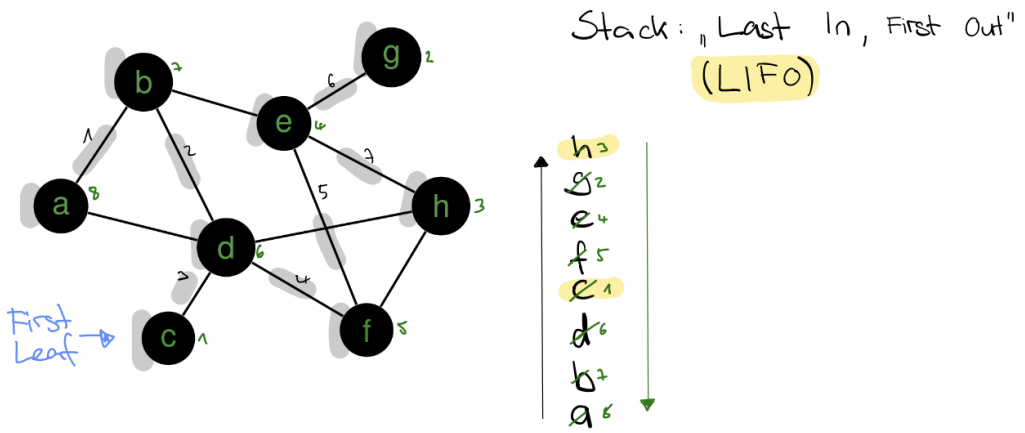


Figure 6.2: Example of DFS

6.2.2 Breadth-First Search (BFS)

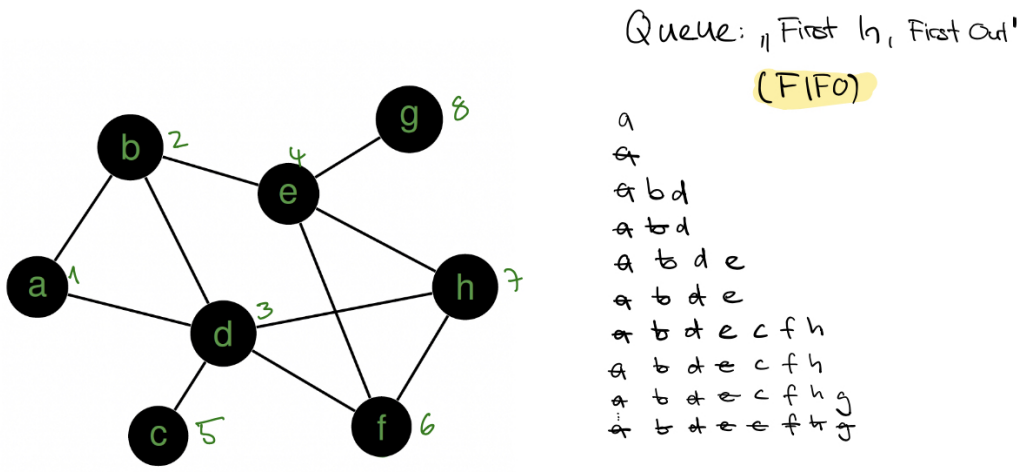


Figure 6.3: Example of BFS

6.3 Undirected graph algorithms

Spanning Trees
Given a graph G(V,E) with positive edge weights.
Find a set of edges that connects all vertices of G and has minimum total weight.

6.3.1 Optimistic approach / Kruskal's algorithm

Successively build the cheapest connection available that is not redundant.

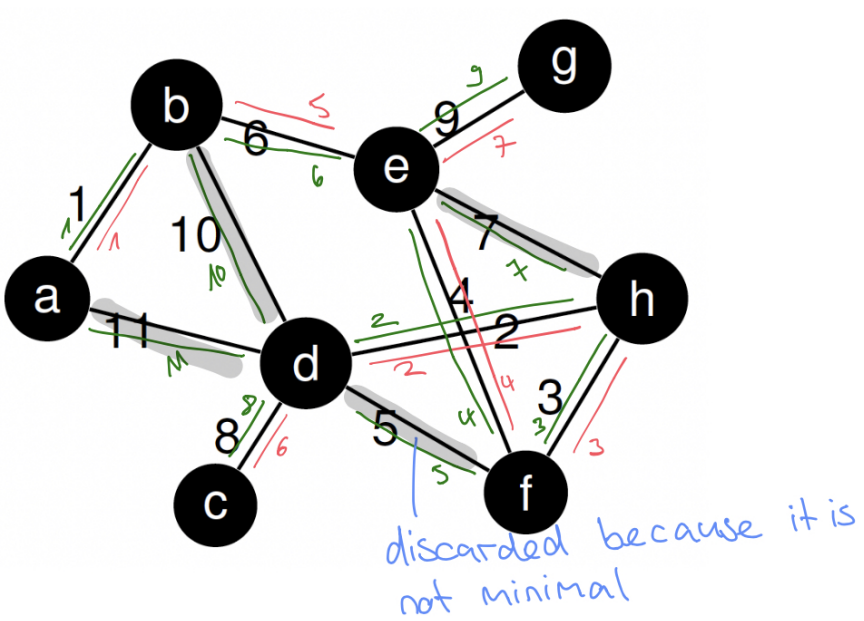


Figure 6.4: Example of optimistic approach

6.3.2 Pessimistic approach

Successively rule out the most expensive line that is not absolutely needed.

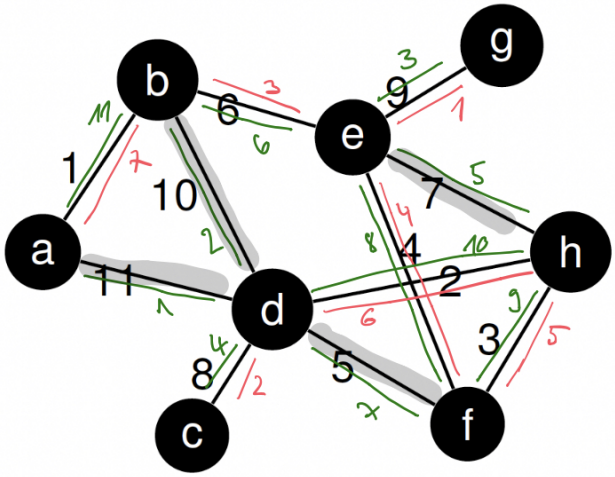


Figure 6.5: Example of pessimistic approach

6.3.3 Prim's algorithm

Iteratively connect a vertex from the neighbourhood that can be reached the cheapest. The tree we are building up is always connected!

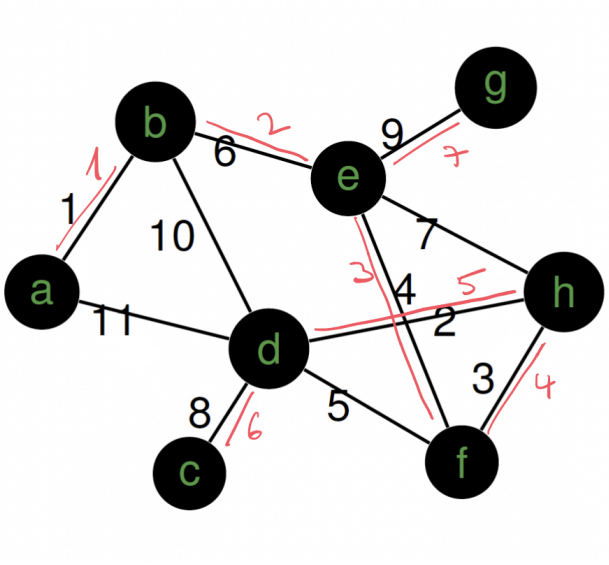


Figure 6.6: Example of Prim's algorithm

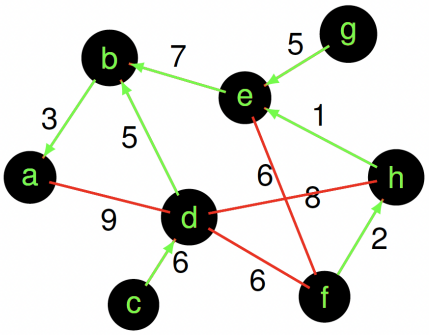
Shortest path:
Given a weighted graph G=(V,E), two vertices $u, v \in V$. Weights are positive, interpreted as distances

Find...

- a shortest path in G connecting u to v
- shortest paths form start vertex v_0 to all other vertices
- shortest paths between all pairs of vertices

The efficient solution for the first two variants is **Dijkstra's algorithm.**
The efficient solution for the third variant (all-pairs shortest paths) is the **Floyd-Warshall algorithm**.

6.3.4 Dijkstra's algorithm



i	e_i	V_i	E_i	$l(v_i)$
0		$\{a\}$	$\{\}$	$l(a)=0$
1	(a,b)	$\{a,b\}$	$\{(a,b)\}$	$l(b)=l(a)+3=3$
2	(b,d)	$\{a,b,d\}$	$\{(a,b),(b,d)\}$	$l(d)=l(b)+5=8$
3	(b,e)	$\{a,b,d,e\}$	$\{(a,b),(b,d),(b,e)\}$	$l(e)=l(b)+7=10$
4	(e,h)	$\{a,b,d,e,h\}$	$\{(a,b),(b,d),(b,e),(e,h)\}$	$l(h)=l(e)+1=11$
5	(h,f)	$\{a,b,d,e,h,f\}$	$\{(a,b),(b,d),(b,e),(e,h),(h,f)\}$	$l(f)=l(h)+2=13$
6	(d,c)	...	...	$l(c)=l(d)+6=14$
7	(e,g)	...	...	$l(g)=l(e)+5=15$

Figure 6.7: Dijkstra's algorithm

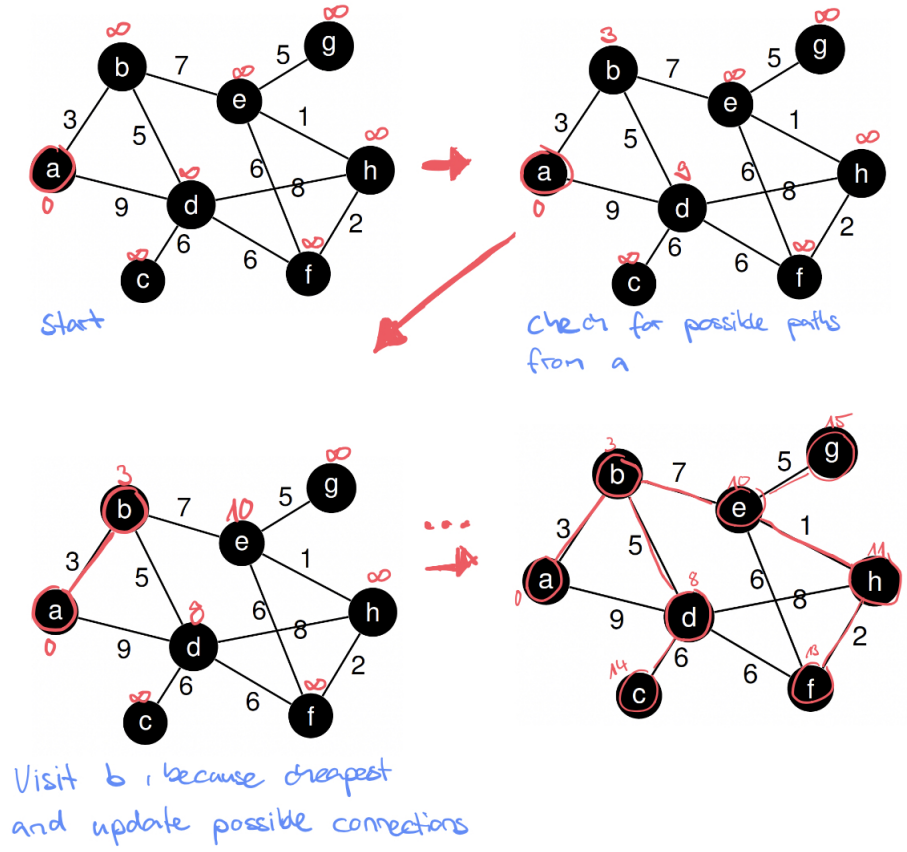


Figure 6.8: Example of Dijkstra's algorithm

6.3.5 Floyd-Warshall algorithm

$\min(m_{ij}, m_{ik} + m_{kj})$

k=1:

	A	B	C	D	E	F
A	0	200	580	–	250	1200
B	200	0	500	820	450	1'400
C	580	500	0	230	150	1100
D	-	820	230	0	380	–
E	250	450	150	380	0	1'450
F	1200	1'400	1100	–	1'450	0

k=2:

	A	B	C	D	E	F
A	–	200	580	1'020	250	1200
B	200	0	500	820	450	1'400
C	580	500	0	230	150	1100
D	1'020	820	230	0	380	2'220
E	250	450	150	380	0	1'450
F	1200	1'400	1100	2220	1'450	0

k=3:

	A	B	C	D	E	F
A	0	200	580	810	250	1200
B	200	0	500	730	450	1'400
C	580	500	0	230	150	1100
D	810	730	230	0	380	1'330
E	250	450	150	380	0	1'250
F	1200	1'400	1100	1'330	1'250	0

k=4:

	A	B	C	D	E	F
A	0	200	580	810	250	1200
B	200	0	500	820	450	1'400
C	580	500	0	230	150	1100
D	810	820	230	0	380	1'330
E	250	450	150	380	0	1'250
F	1200	1'400	1100	1'330	1'250	0

k=5:

	A	B	C	D	E	F
A	0	200	580	630	250	1200
B	200	0	500	820	450	1'400
C	580	500	0	230	150	1100
D	630	820	230	0	380	1'330
E	250	450	150	380	0	1'250
F	1200	1'400	1100	1'330	1'250	0

k=6:

	A	B	C	D	E	F
A	0	200	580	630	250	1200
B	200	0	500	820	450	1'400
C	580	500	0	230	150	1100
D	630	820	230	0	380	1'330
E	250	450	150	380	0	1'250
F	1200	1'400	1100	1'330	1'250	0

Figure 6.9: Example of Floyd-Warshall algorithm

6.4 Network flows

Definition 8. ***Maximum network flow***
An st-network (G, ω, s, t) is a weighted graph G(V,E) with weight function ω and two distinguished vertices $s, t \in V$, where s is the ***source*** *and t is the target/****sink.***

Question: *What is the maximal flow we can route through G(V,E) from s to t, respecting the capacity limits given by ω?*

6.4.1 Maximum flow: Greedily finding augmenting paths

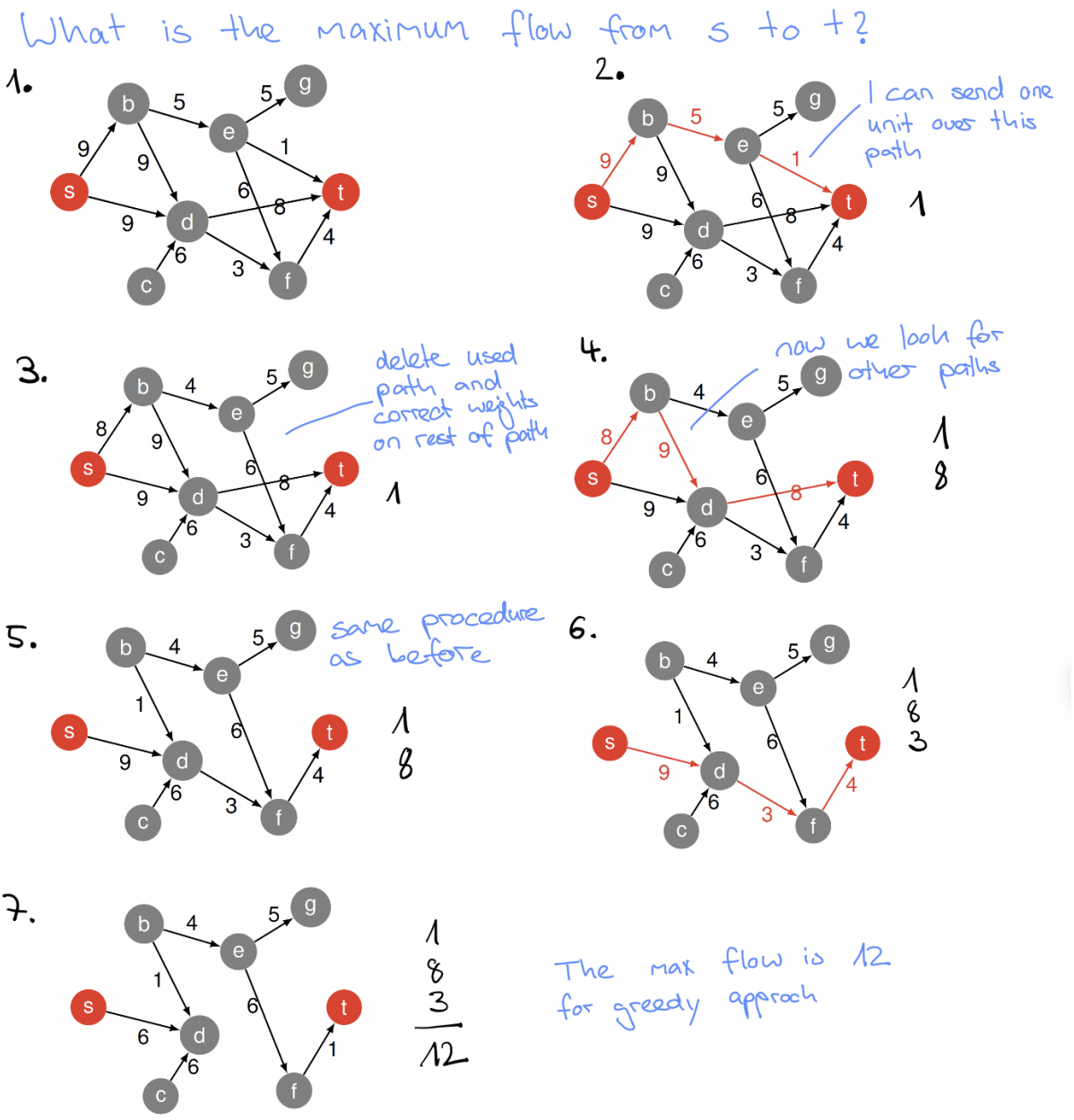


Figure 6.10: Example of greedily finding augmenting paths

Not optimal algorithm! The amount of flows/max flow we obtain depends on the order we process the graph. Therefore, it is not an optimal algorithm.

6.4.2 Ford-Fulkerson algorithm

Idea: Insert backward edges that can be used to (partially) undo previous paths!

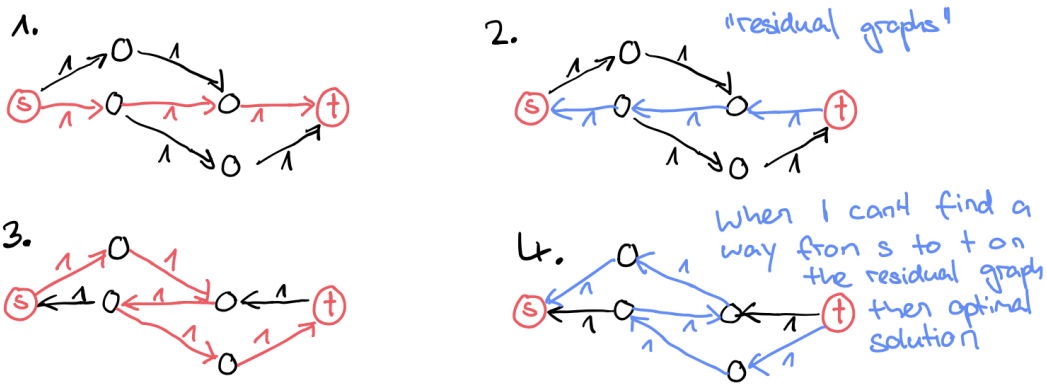


Figure 6.11: Idea of the Ford-Fulkerson algorithm

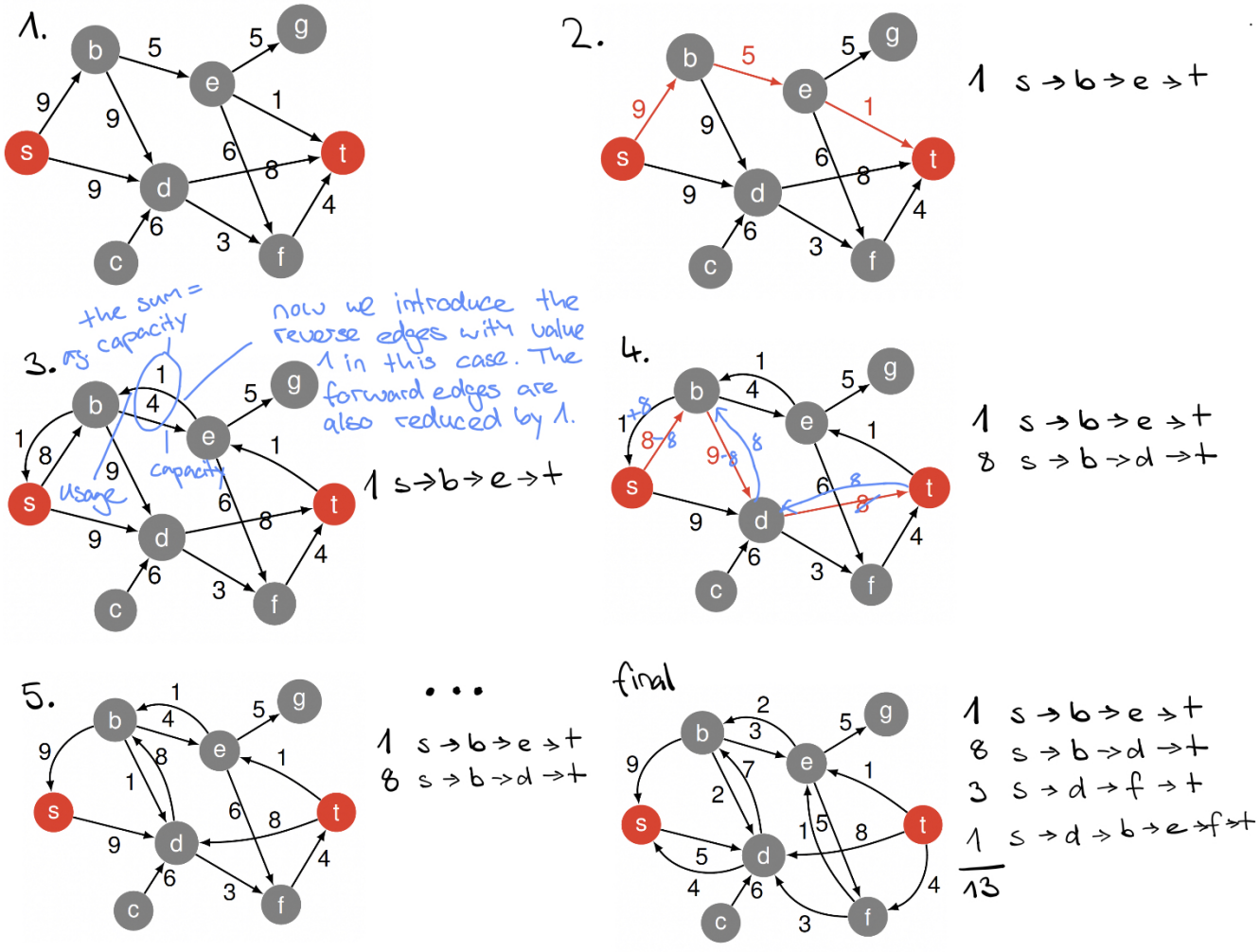


Figure 6.12: Example of Ford-Fulkerson algorithm

Ford-Fulkerson algorithm: Inefficient behaviour
If the augmenting paths are chosen arbitrarily, the algorithm might take many iterations to find the maximum flow:

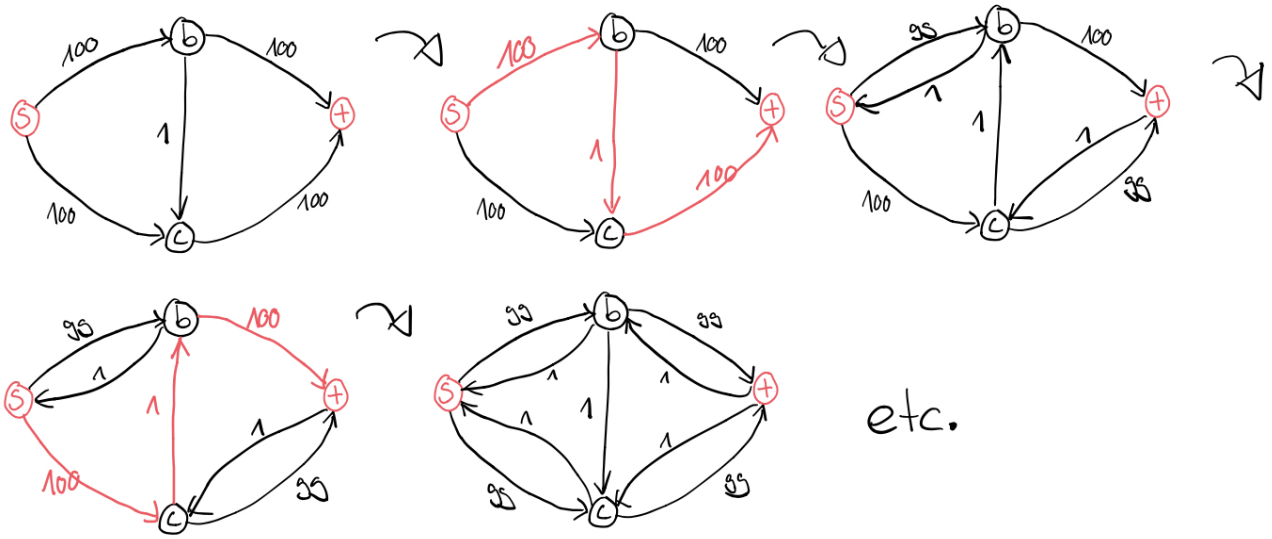


Figure 6.13: Example of inefficient behaviour of Ford-Fulkerson algorithm

6.4.3 Edmonds-Karp algorithm

In each iteration, use a shortest augmenting path, i.e. a path from s to t with the **fewest number of edges**. Can be found with BFS in time $O(|E|)$. Such a algorithm is also called topological sort algorithm. Using this extension we would avoid the situation shown above in the example.

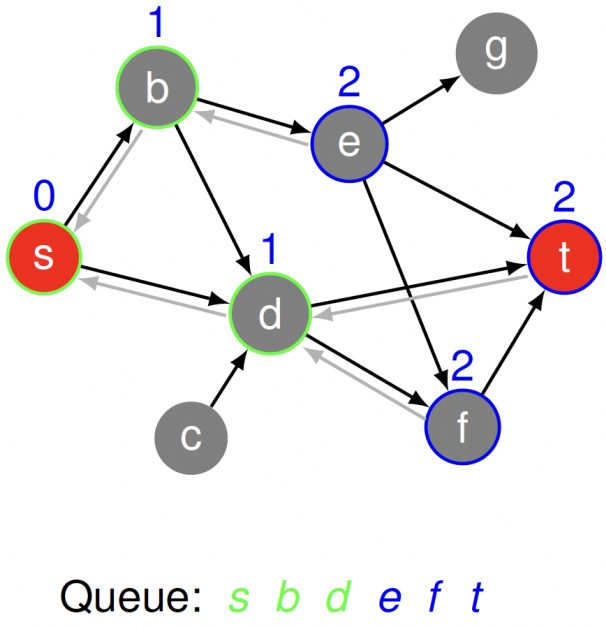


Figure 6.14: Example of finding augmenting path with fewest number of edges

In this case the shortest augmenting path is: $s \to d \to t$

6.4.4 Distribution problems reduced to Max-Flow

Insert source s and connect it with capacity ∞ edges to first layer.

Insert sink t and connect it with capacity ∞ edges to last layer.

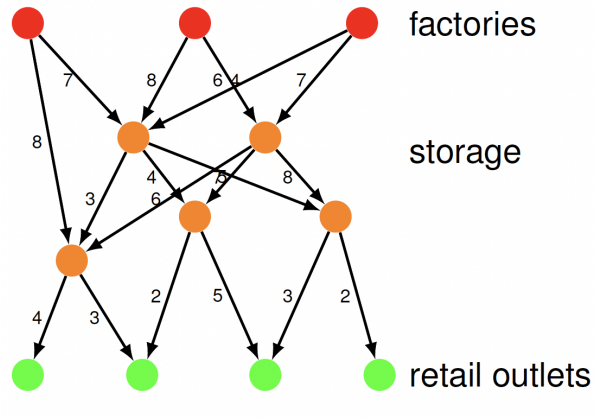


Figure 6.15: Distribution problems reduced to Max-Flow

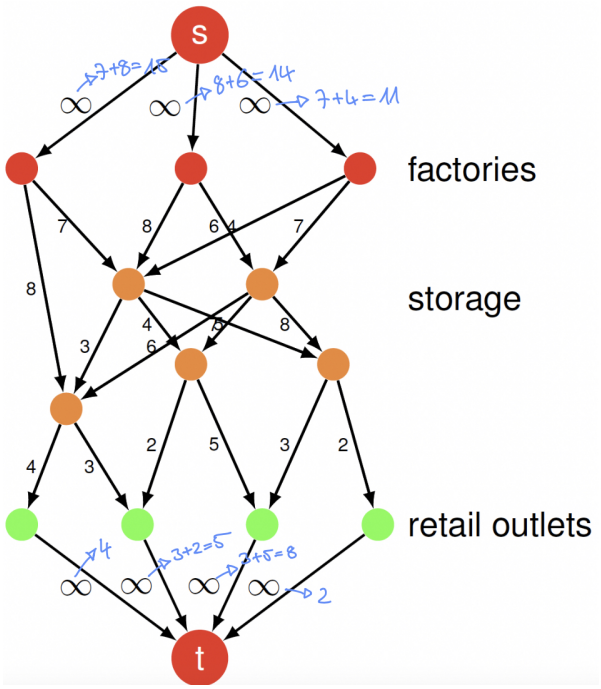


Figure 6.16: Distribution problems reduced to Max-Flow

6.4.5 Modeling vertices with restrictions

Maximum flow through vertices is restricted. Specifically, for s by 15, for t by 20, for b by 3, and for d by 9.

Idea: Represent each vertex restriction by an additional edge.

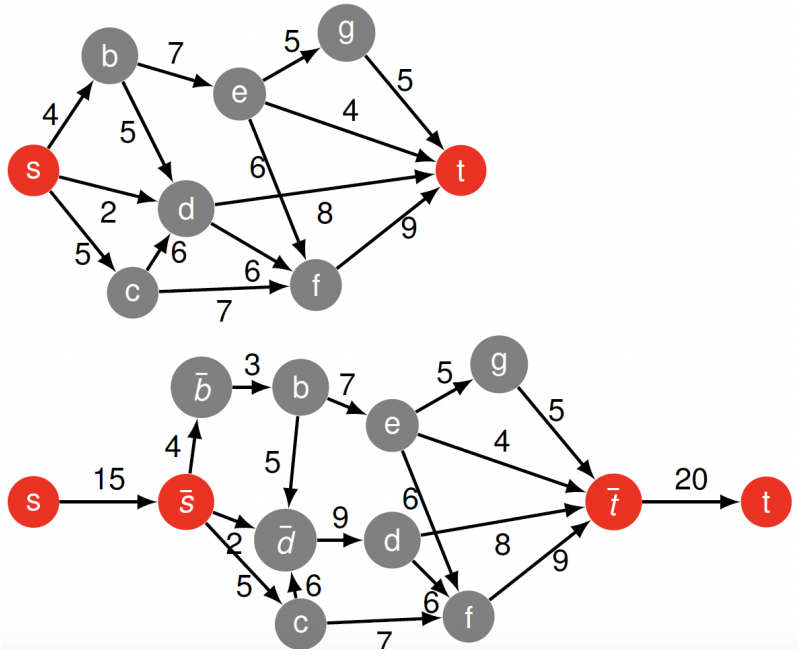


Figure 6.17: Modeling vertices with restrictions

6.4.6 Bipartite matching

Make edges directed. Add source and sink vertices s and t with corresponding edges. Set all edge weights to 1. Find integer (!) maximum flow with Ford-Fulkerson algorithm.

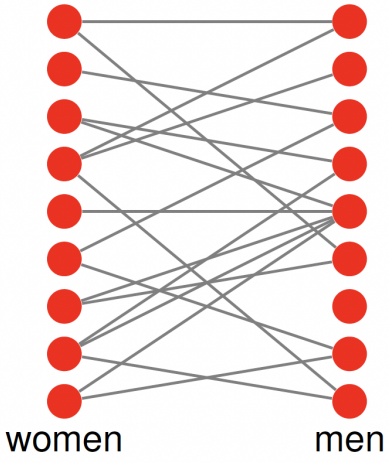


Figure 6.18: Bipartite matching

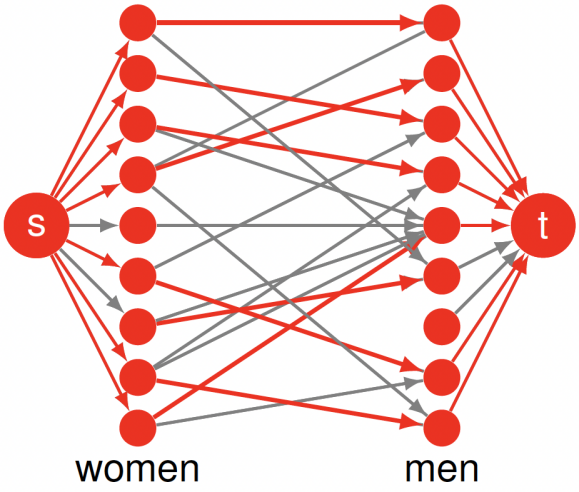


Figure 6.19: Bipartite matching

6.5 Max-Flow Min-Cut

Definition 9. *An st-**cut** in the graph $G(V, E)$ is a partition of the vertex set into two sets S and $T = V \setminus S$, with $s \in S$, $t \in T$. The **value** of such an st-cut is the total weight of all edges that go from S to T.*

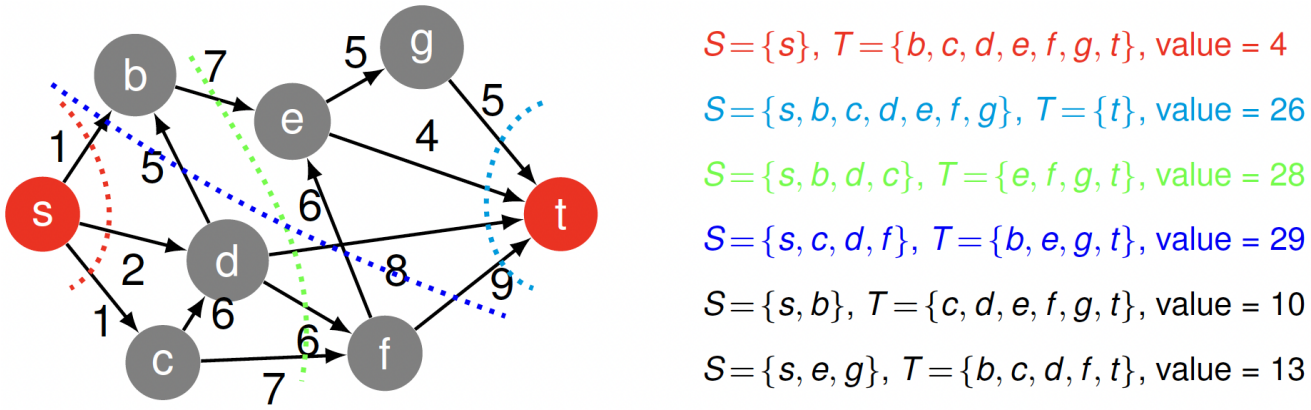


Figure 6.20: Max-flow min-cut theorem

Theorem 9. ***Max-flow min-cut:*** *The maximum st flow equals the minimum value of all (exponentially many!) possible st-cuts.*

6.6 Min-Cost Max-Flow

Often there are several possible maximum flows in a graph, sending different amounts of flow along different edges. The Ford-Fulkerson algorithm can be extended to take into account an additional **cost criterion**.

Each edge has not only a capacity ω, but also a **cost per flow unit** going through that edge. The min-cost max-flow algorithm finds a maximum flow with minimum cost.

First we just apply Ford-Fulkerson with additionally annotating the costs. **Then we look for cycles with negative cost.** Send flow along these cycles, respecting the capacities. This does not change the total flow from s to t, but decreases the total cost.

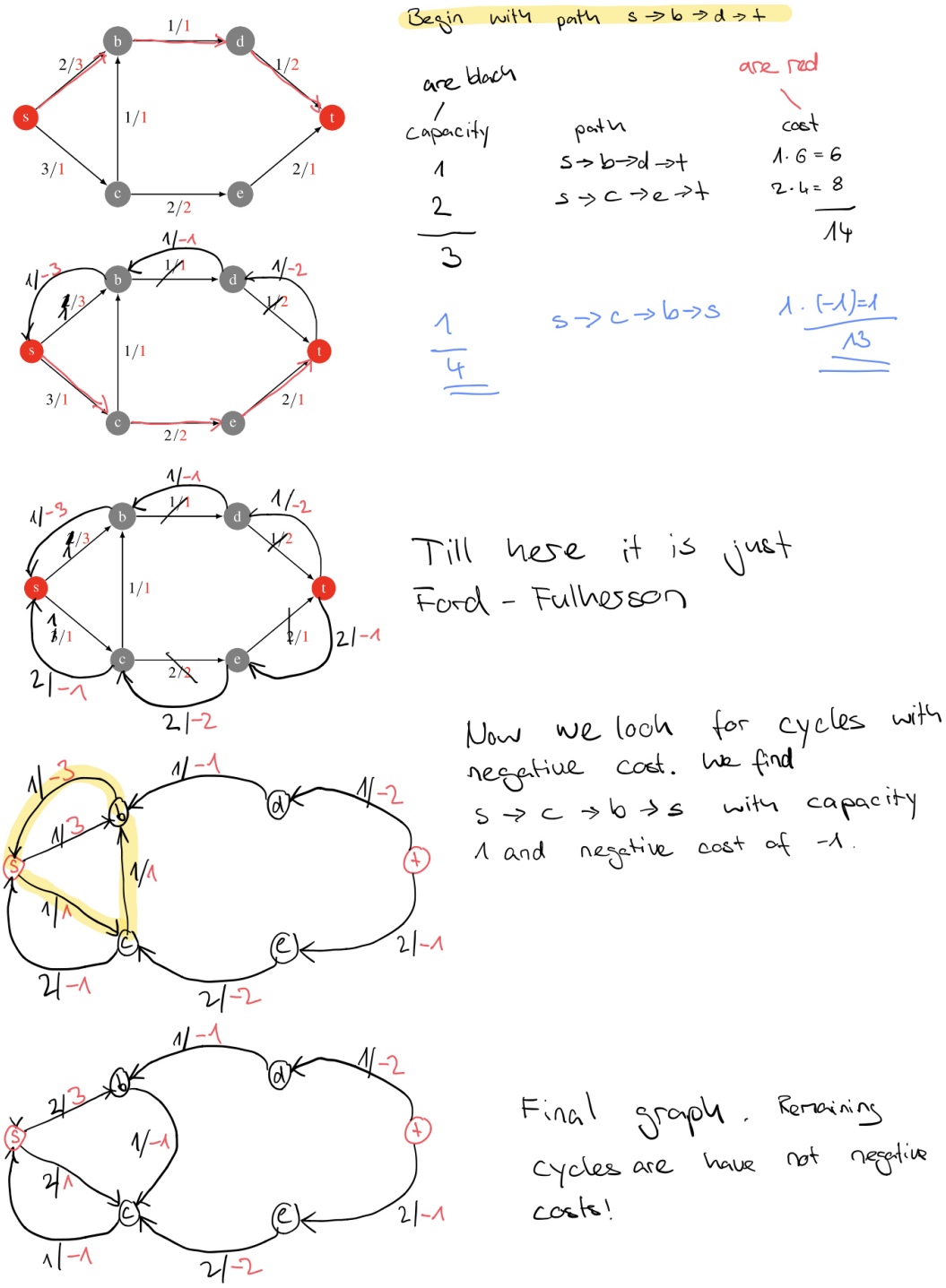


Figure 6.21: Exmple of Min-Cost Max-Flow

Chapter 7

Heuristic optimization

7.1 Example: Pathfinding

Find shortest path from current position to given goal.

Dijkstra:
Always finds optimal path, but is computationally expensive.

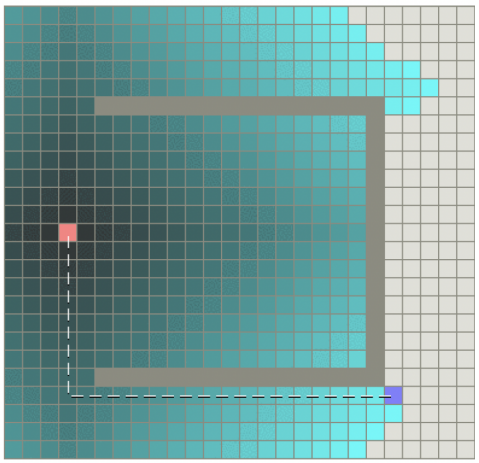


Figure 7.1: Example Path finding with Dijkstra

Best-first heuristic:
Runs into dead end, but eventually finds a detour. Less computationally expensive. But it is still fast.

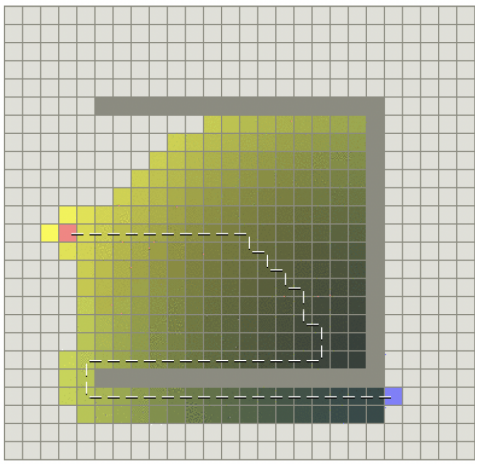


Figure 7.2: Example Path finding with Best-first heuristic

A^***:**
Combines **Dijkstra** (shortest distance to start) and **best-first heuristic** (shortest distance to goal): Explores vertex with **smallest weighted sum** in each step.

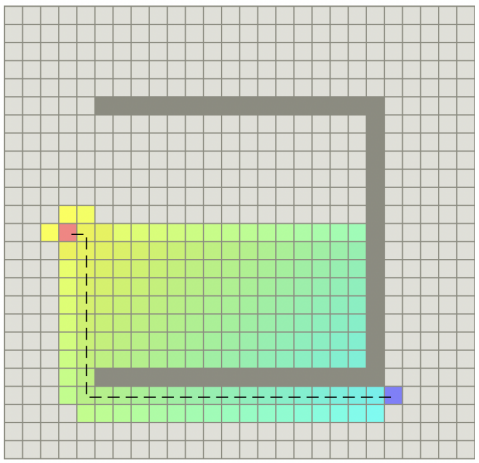


Figure 7.3: Example Path finding with A^*

Definition 10. *A **heuristic** is a problem-solving strategy that is based on rules of thumb or common sense, possibly also using expert knowledge. Heuristics are often used when no efficient (exact) algorithms are known, or when applying such algorithms would take too long.*

***Advantage**: Hopefully good solution in a reasonable amount time.*

***Disadvantage**: Usually no reliable optimality guarantees.*

Definition 11. ***Problems and complexity classes***
P: solvable in polynomial time
NP: solvable in polynomial time with a non-deterministic machine
NP-hard: at least as hard as the hardest problems in NP
NP-complete: in NP and NP-hard

The following problems are known to be in P, i.e. a polynomial-time algorithm is known for them:

- minimum spanning tree in a graph
- shortest paths in a graph
- maximum flow in a network
- sorting of items in alphabetically order
- searching e.g. a text for instances of a word

7.2 TSP - Traveling Salesman Problem

Given: A weighted graph G(V,E) on n vertices (cities).
Find: A tour through the n cities that has minimum length.

n cities → n! possibilities
5 cities → 120 possibilities
10 cities → 3'628'800 possibilities
15 cities → 1'307'674'368'000 possibilities
49 cities → 608'281'864'034'267'560'872'252'163'321'295'376'887'552'831'379'210'240'000'000'000 possibilities

It is considered to be one of the hardest problems.

7.2.1 TSP - Nearest Neighbor Heuristic

Nearest Neighbor Heuristic: Always go to closest city next.
Start in a. Does it find an optimal tour? No!

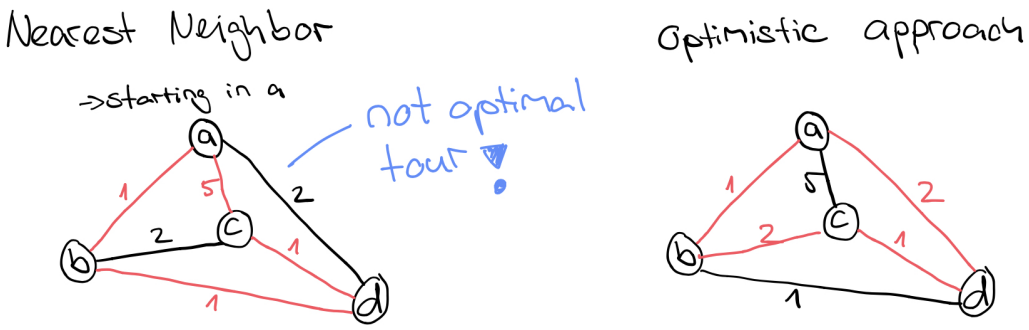


Figure 7.4: Example of Nearest Neighbor Heuristic

7.2.2 TSP - MST-based Heuristic

- Construct a minimal spanning tree.
- Duplicate each edge and convert them into directed edges with opposite directions.
- Starting from a node s, traverse the cycle by using every edge exactly once, such that you visit every node one time:
  - Go to unvisited nodes first.
  - If a node has only visited neighbors, take a shortcut to the next unvisited node, following the unused edges.
- Do so, until all edges are used and you are back at s.

Definition 12. *A **metaheuristic** is a general heuristic problem-solving strategy that can be employed in optimization problems.*

Can be employed both in discrete and continuous problems.

Can be used both when looking for local and for global optima.

Can sometimes be combined with more rigorous local methods (e.g. intensification strategy) to search for global optima in continious optimization problems (hybrid methods).

Definition 13. *Given a discrete or continuous optimization problem. No efficient algorithm is known and enumerating all possibilities is not feasible.*

→ ***Trajectory-based metaheuristics:*** *Start with a (random) „solution“ (e.g. an initial tour in the TSP) and improve it continuously by exploring its „neighborhood“ (improve the tour).*

We obtain a trajectory through the solution space, similar to e.g. gradient descent.

Often not obvious for a given problem:
*- How to define and explore the **solution space**?*
*- How to define a **good neighborhood**?*

7.3 Hill climbing

Idea: Search for improvements in neighborhood and improve current solution successively.

7.3.1 Hill climbing algorithm

1. **Initialization**:
   Start with random solution. Evaluate initial solution.
2. **Iteratively**:
   Explore neighborhood and evaluate new solution candidates. Continue with best solution found, if this improves the target function. (Otherwise we're in a local optimum.)
3. Return solution.

Variant: Always take best solution from neighborhood, even if this decreases target function. Pros and cons?

- Might escape from local optima.
- Not monotone increasing.
- Might go back and forth, i.e. run into a cycle.

Remarks:

- Implementation is trivial in principle, but...
- Definition of neighborhood, or equivalently definition of allowed modification steps?
- In particular: Size of neighborhood? Too small: might get stuck quickly without exploring solution space; too large: computation time!
- Computational effort to determine and evaluate neighborhood?
- Possibly helpful: Evaluating solutions by updates, or by some approximate computation.
- Hill climbing is (like e.g. gradient descent) a **local search** algorithm. Always a good idea: Multiple runs from different (random) starting points!

7.3.2 Example: Move-Set for TSP (also k-opt neighborhood)

Neighborhood defined through modification/move steps. Therefore, we delete two connections and create two one. The neighborhood consists of all possible combinations that are valid. If the new path is smaller, it is accepted. The next option would be a 3-change. This is often considered as a perfect trade-off to find a good solutions.

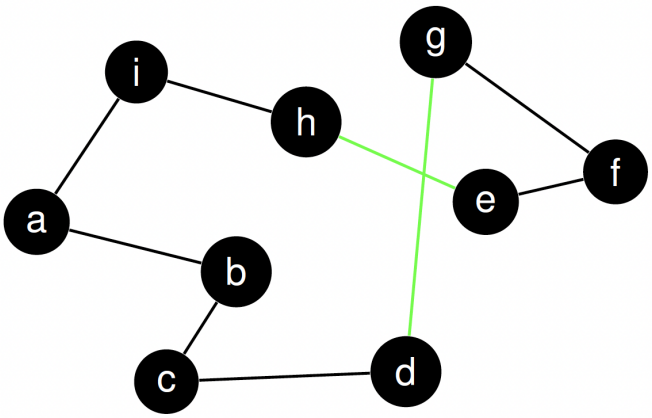


Figure 7.5: Example: 2-change

7.3.3 Stochastic hill climbing algorithm

1. **Initialization**:
   Start with random solution. Evaluate initial solution.
2. **Iteratively**:
   Choose random solution in neighborhood (i.e. perform a random modification). Evaluate new solution candidate, and accept it, if it is better. Continue until termination criterion is reached.
3. Return solution.

7.3.4 Hill climbing: Terminating conditions

Terminate

- if no significant improvement occurs after a fixed number of steps without improvement.
- after a fixed total number of iterations or computing time.
- when a prescribed target value for the target function is reached.

Further remarks:

- Neighborhoods need to be large enough to explore solution space.
- **What if we get invalid „solutions“?**
  2-change in TSP always produces valid solutions.

7.3.5 Continuous hill climbing

Setup as previously in continuous optimization: Find $\vec{x}$ minimizing $f(\vec{x})$ for a given target function $f : \mathbb{R}^n \to \mathbb{R}$.

1. **Initialization** $i = 0$:
   Initial solution x_0
2. **Iteration** $i \to i + 1$:
   Sample random solution y_i near x_i.

$$x_{i+1} = \begin{cases} y_i, & \text{if } f(y_i) \leq f(x_i) \\ x_i, & \text{if } f(y_i) > f(x_i) \end{cases}$$

3. Return solution

Idea is similar to gradient descent, but since we do not rely on derivatives, the idea also works in discrete settings.

What would be a good heuristic for the traveling salesman problem?

A good heuristic for the Traveling Salesman Problem (TSP) is one that generates high-quality solutions quickly and efficiently. Some common heuristics used for TSP include:

- **Nearest Neighbor:** This heuristic starts at a random city, and at each step, it chooses the nearest unvisited city to visit next. This heuristic is simple and fast, but it may not always produce the best solutions.
- **2-Opt:** This heuristic involves swapping two edges in the solution to reduce the total distance. The optimization process continues until no further improvements can be made.
- **Christofides Algorithm:** This heuristic solves the TSP by first finding a minimum spanning tree of the cities, and then finding a minimum weight perfect matching between the nodes not yet covered by the tree.
- **Genetic Algorithm:** This heuristic uses principles from evolution and genetics to find a solution. It starts with a population of randomly generated solutions and evolves this population over multiple generations using selection, crossover, and mutation operations.
- **Simulated Annealing:** This heuristic is based on the idea of slowly cooling a system to find the global minimum. In the TSP, it starts with a random solution and iteratively changes the solution, accepting the change if it reduces the cost, and sometimes accepting it even if it increases the cost, with a probability that decreases as the temperature decreases.

These are some of the most commonly used heuristics for TSP, and their performance will depend on the specific problem instance. To get the best results, it's often a good idea to combine multiple heuristics or use a metaheuristic that incorporates several different heuristics.

7.4 Tabu search

Idea: Similar to hill climbing, but with some memory. Try to avoid steps that go back to previously visited solutions, or that undo the effect of previous steps. (These steps are „tabu“.)

The goal is to promote diversity of the solutions explored, in particular to reduce cyclic behaviour and to escape from local optima.

1. **Initialization**:
   Start with random solution. Evaluate initial solution.
2. **Iteratively**:
   Explore neighborhood and evaluate new solution candidates. Only consider steps that are not tabu.
   Proceed with step 2 with the best solution found and update tabu list until termination criterion is reached.
3. Return solution.

7.4.1 Example: CVRP - Capacitated Vehicle Routing Problem

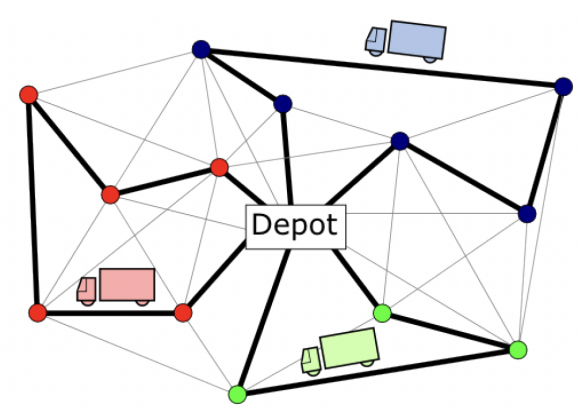


Figure 7.6: Example of CVRP - Capacitated Vehicle Routing Problem

Given: Graph $G = (V, E)$. One vertex is the depot with m identical trucks of capacity N. All other vertices represent customers with demand n_i and service time t_i. All edges are assigned travel costs k_{ij} and travel times t_{ij}.

Find: The least expensive set of delivery routes s.t.:

- each customer gets his delivery, i.e. is on one of the routes,
- each route begins and ends at the depot,
- the total demand on each route does not exceed N, and
- for each route, the total time (travel times t_{ij} plus service times t_i) does not exceed some global limit T.

Neighborhoods, defined by possible moves:

- Reassign customer to a different route.
- Insert it randomly or at best possible position.
- Exchange two customers between different routes.
- Exchange two customer and reoptimize involved route(s).
- Chains of exchanges, e.g. of length 3
- Optimizing the routes individually (cf. TSP)

Tabu lists: Customer 1 was reassigned from route 1 to route 2: Possible entries in the tabu list to block the opposite move:

Variant 1: tabu (customer 1, route 2 → route 1)
Variant 2: tabu (customer 1, route 2 → any other route)
Variant 3: tabu (customer 1) even within route 2

CVRP - Minimizing number of trucks

Goal: Minimizing **number of trucks** (instead of cost).

Problem: *Target function only takes a few discrete values.*
Use replacement function that rewards desirable properties, e.g. good utilisation of the truck's capacities!

7.4.2 Tabu list

- Simplest variant: Only store last solution as a tabu (to avoid going back and forth).
- Probably better: Keep e.g. a tabu list corresponding to last k moves.
- How restrictive should the tabus be? (Forbid specific configurations, whole classes of moves, certain values for certain variables, ...?)
- Duration of tabus (short- vs. mid- vs. longterm)?
- Size and organization of tabu list(s)?
- Are the tabus enforced strictly or do we allow exceptions („aspiration")? When and why?

7.4.3 Exploring the solution space

The goal of the tabus is to help exploring the solution space, i.e. to foster **diversity** of the considered solutions.

Further considerations:

- Again: Size of neighborhoods, computational requirements? Approximate evaluation or approximation by updates possible?
- Restart in parts of the solution space that have not yet been explored.
- Exploiting structure: Consider common parts of solutions encountered and rule them out (⇒ diversity, early in the process) or focus on them (⇒ intensification, later in the process)
- etc... No exact science, many tradeoffs!

7.4.4 Invalid solutions

Should we always rule out steps that violate constraints (e.g. capacity or time constraints in CVRP)?

Good valid solutions often are very close to violating the constraints. Considering invalid solution might help with finding these! → Allow invalid solutions but **penalize** them.

Strategic oscillation: Dynamically adjust size of penalty to reach limits of feasiblity. If we encounter many invalid solutions, increase penalty, if we encounter few invalid solutions, maybe decrease penalty to encourage diversity.

Variants:

- Fixed penalty for invalid solutions
- Variable penalty, adapting to degree of violation.

7.4.5 Randomized tabu search (Stochastic version)

1. **Initialization**:
   Start with random solution. Evaluate initial solution.
2. **Iteratively**:
   Choose a random solution from the neighborhood. Evaluate the new solution candidate, if its not tabu. If a better solution is found, continue with this solution, update tabu list and proceed with step 2. Continue until termination criterion is reached.
3. Return solution.

Appropriate tabus, when optimizing a continous function?
Disable contrary moves, i.e. moves in the opposite direction.

Remarks:

- Keep track, if the tabus take effect.
- Set unchanging elements of solution candidates tabu, to explore solution space.
- Adjust the neighborhood dynamically.
- Choose multiple solution candidates from the neighborhood in each iteration. How?
- Start tabu search with different starting solutions and compare the best found solutions.

7.5 Simulated annealing

Idea: Similar to hill climbing, but also allow non-improving moves to escape from local optima.

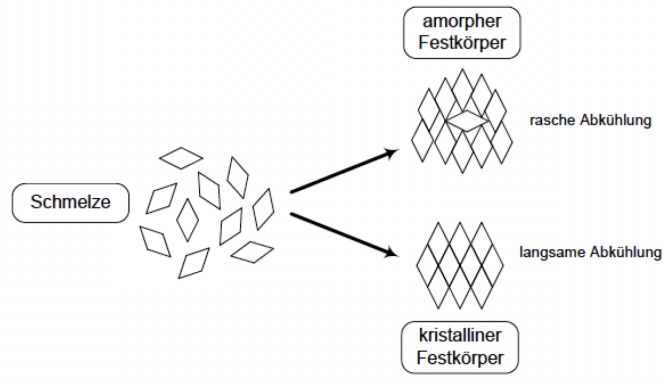


Figure 7.7: Idea of Simulated annealing

Idea from solid physics when **cooling down crystalline structures**:
The ideal **state of minimal energy** is a perfect crystal structure, but when cooling happens too quickly, the crystalline structures cannot form properly – the system gets stuck in suboptimal states.

To get a perfect crystal structure it is necessary to escape from local optima. The probability of a state (crystal structure) x with energy $E(x)$ at temperature T is proportional to $e^{-\frac{E(x)}{T}}$. Defined by these probabilities it is possible to get to inferior states. These probabilities increase with temperature.

Do hill climbing with a „**temperature**“ T and decrease T over time according to some cooling schedule.

7.5.1 Simulated annealing algorithm

1. **Initialization**:
   Start with random solution. Evaluate initial solution.

2. **Iteratively**:
   Choose random solution in neighborhood and evaluate it. If it is better, accept it. If it is worse, accept it only with some probability. Continue until termination criterion is reached.

3. Return solution.

Idea 1: Accept with probability e.g. 0.1

Idea 2: Probability decreases over time

Idea 3: Probability decreases over time and also depends on quality of solution.

1. **Initialization**:
   Start with x_0

2. **Iteratively**: $x_i \to x_{i+1}$
   Sample a random solution y_i in the neighborhood and accept it to be x_{i+1} with probability $\min\{(1, e^{\frac{f(x_i)-f(y_i)}{T_i}})\}$

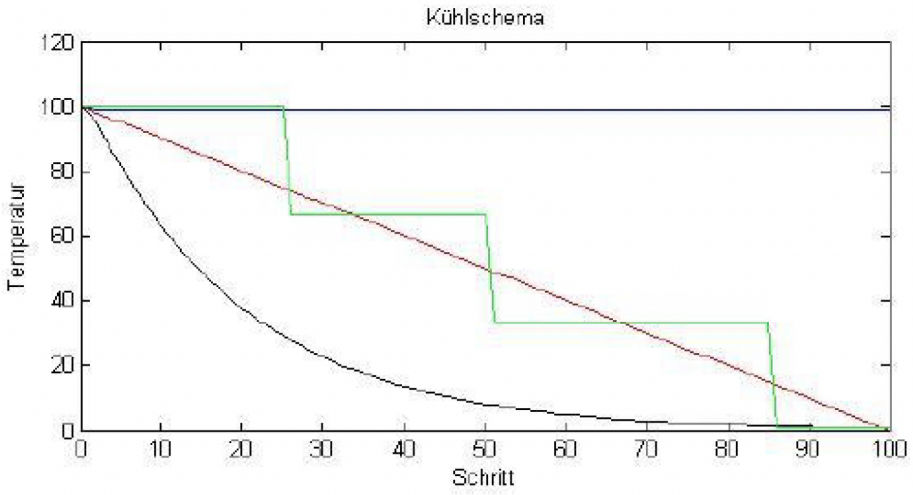


Figure 7.8: Different cooling schedules for T_i are possible:

Definition 14. ***Population-based methods***

Key idea*: An evolving* ***population*** *of (partial) solutions, whose members evolve and adapt individually to the problem and are searching for the optimum. Problem-specific information can be* ***exchanged*** *between the members of the population, and can also be* ***passed on*** *to descendants.*

Can be employed both in discrete and continuous problems.

Definition 15. ***Biological principles as an inspiration***

Evolving population consisting of individuals

- *Finite lifespan of individuals (generations)*
- *Adaptation by natural selection, survival of the fittest*
- *Inheritance of traits, information is passed on to descendants*
- *Recombination, information is exchanged between parents*
- *Diversification, e.g. by mutations*
- *Population acquires and cultivates shared knowledge („culture“, „collective memory“)*
- *Local development of populations (different cultures in different regions)*

These principles are the inspiration for evolutionary algorithms, in which ***populations of solution candidates*** *evolve over time.*

7.6 Definition evolutionary algorithms

Definition (Hertz and Kobler): **Population-based methods** are **iterative solution techniques** that handle a **population of individuals** and make them **evolve according to some rules** that have to be clearly specified. At each iteration, periods of **self-adaptation** alternate with periods of **cooperation**.

7.6.1 Classification criteria

Classification of evolutionary algorithms (Hertz and Kobler)

1. **Individuals:** Solution candidates or partial solutions
2. **Evolutionary process:** E.g. constant-size population without survivors from one generation to the next.
3. **Neighborhood structure:** Which individuals can exchange information?
4. **Information sources:** E.g. transmitting information from „parents“ to „children“. Also input from external sources?
5. **Restriction-violating solution candidates:** Discard, repair or penalize?
6. **Intensification strategy:** Exploiting neighborhood by subsequent adjustment of solution candidates.
7. **Diversification strategy:** Exploring the entire solution space, e.g. by mutations.

7.7 Biological inspiration

7.7.1 Molecular genetics

A gene is an encoding segment in a specific place (locus) on a chromosome.

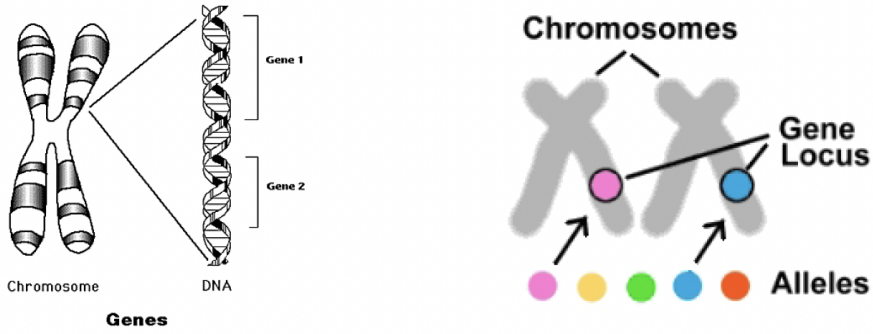


Figure 7.9: Molecular genetics

An allele is a possible value of a gene, encoded by a sequence of bases A, C, G and T and results in a characteristic trait. The genes encode the properties of an individuum.

7.7.2 Life cycle of a generation

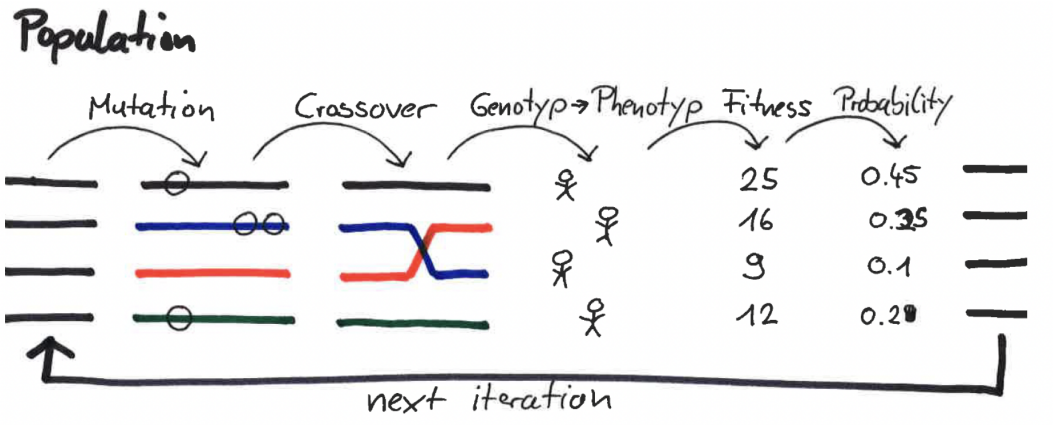


Figure 7.10: Life cycle of a generation

Example: Maximizing 2-dimensional function

Maximize $f(x, y) = \sin^2(4\pi x)\sin^2(4\pi y)e^{-(x+y)}$ on $[0, 1] \times [0, 1]$.

An individual has a chromosome with two genes x and y.

	x	y
Individual 1:	0.71	0.33
Individual 2:	0.44	0.21
⋮		
Individual n:	0.12	0.23

Fitness is measured by $f(x, y) = \sin^2(4\pi x)\sin^2(4\pi y)e^{-(x+y)}$.

Figure 7.11: Maximizing 2-dimensional function

In the first step we choose randomly values for x and y. Here, we can the function itself as the **fitness function**. Possible **mutations** could be random additions/subtractions or we just replace the value with a random number between 0 and 1, etc. A possible **crossover** would consist of 0.71 and 0.21. A possible **selection** would be in this case the one with the highest fitness function value (with a certain probability).

7.7.3 Encoding the optimization problem

The first and most important step is a suitable encoding: We have to come up with the genotype. What is a proper encoding of our real world problem. Often we come up with a vector with certain entries that describe our population members. From this we also have the induced fitness function. That is not the same as the fitness function itself but similar or it represents the same just in a different way. Remember the vehicle routing example from last week.

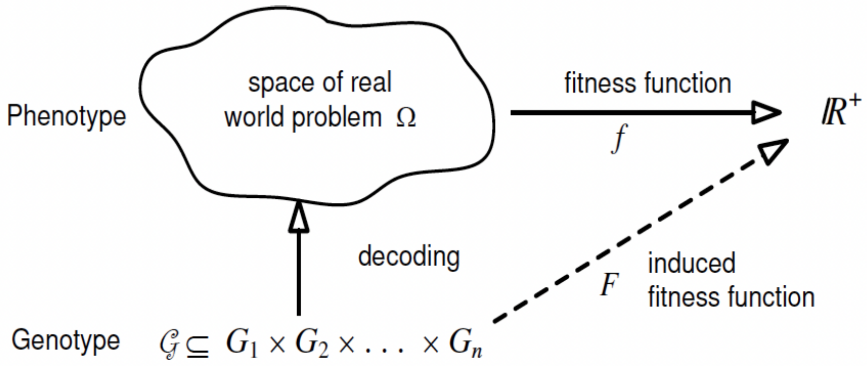


Figure 7.12: Encoding the optimization problem

Example: Knapsack problem

Maximize value of packed items without exceeding knapsack capacity!

What would be a possible **Encoding** and **Fitness function**?

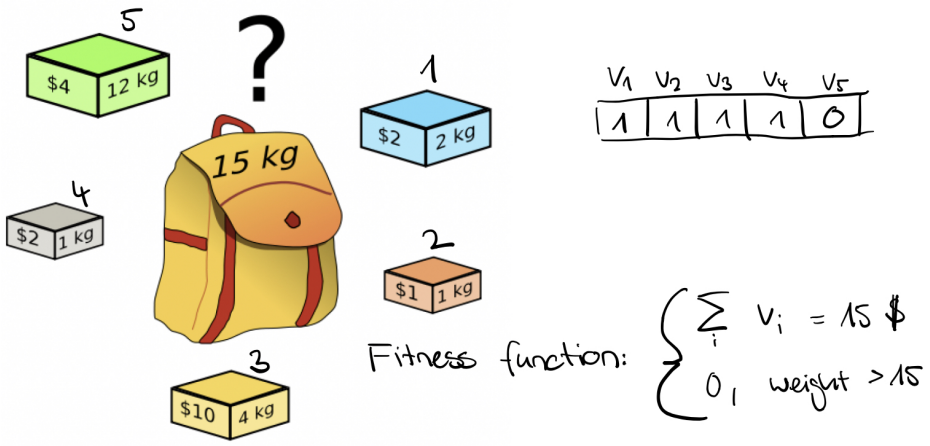


Figure 7.13: Encoding the Knapsack problem

7.8 Genetic algorithm

7.8.1 Terminology

Definition 16. *An **individual** is a possible solution candidate of the optimization problem.*

Often it is represented as a vector with n (binary, integer or real) entries.

Interpretation is problem-specific.

Definition 17. *The individual entries in (e.g.) the vector representation of an individual are its **genes**.*

They describe the genetic information and the properties of the individuals.

E.g., the individual (0; 0; 1; 0; 1) (binary encoding) consists of five genes.

Definition 18. *The concrete values of a gene can take in an individual are called **alleles**.*

In binary-encoded individuals, the only possible alleles are 0 and 1.

More generally, alleles can be integers, real numbers, ...

Definition 19. *The **population** is the set of all individuals in an optimization problem at a given time.*

All individuals of the population are represented in the same way.

Definition 20. *A **generation** is the population at a specific point in time.*

In each iteration of a genetic algorithm, a new generation is created.

The hope is that over time, the population members of each generation are better and better solutions to the optimization problem.

Definition 21. *The **genotype** is the encoded form of an individual.*

The choice of encoding is important for the behavior of the genetic algorithm!

Definition 22. *The **phenotype** is the decoded form of an individual. It does not depend on our choice of encoding.*

In other words, the phenotypes are the solution candidates for our optimization problem.

Definition 23. *The **fitness function** is our measure for the quality of a solution candidate in the optimization problem.*

It comes into play e.g. when selecting parents for creating the next generation.

7.8.2 Algorithm

1. **Initialization**: Random starting population.
2. **Iteratively**: Create next generation according to evoluationary principles:

   * Assign fitness to individuals
   * Natural selection and choosing parents for reproduction
   * Recombination process
   * Mutation process

   Repeat 2 until termination criterion is satisfied
3. Return best individual

7.9 Encoding of an optimization problem

Desirable: A small change in genotype corresponds to a small change in phenotype.

A very simple encoding is often given by a **standard binary encoding.** (Issues with Hamming distance $\rightarrow$ **Gray-Code**)

In the TSP example we use an **integer encoding**; more precisely, a permutation.

In continuous optimization problems, **real-valued encodings** are sometimes more useful.

7.9.1 Example TSP with $n = 12$ cities: Encoding

3	7	9	1	12	2	10	11	5	8	4	6

An individual, i.e. a tour through n (numbered) cities can be represented by a vector of n integers.

$\Rightarrow$ A valid tour is a **permutation** indicating in which order the cities are visited.

7.9.2 Example Gray code

Gray code is basically a modification of the standard binary code that the hamming distance is always equal to 1 between neighbors.

decimal	standard binary	Gray code
0	000	000
1	001 (Hamming distance = 1)	001 (Hamming distance = 1)
2	010 (Hamming distance = 2)	011 (Hamming distance = 1)
3	011 (Hamming distance = 1)	010 (Hamming distance = 1)
4	100 (Hamming distance = 3)	110 (Hamming distance = 1)
5	101 (Hamming distance = 1)	111 (Hamming distance = 1)
6	110 (Hamming distance = 2)	101 (Hamming distance = 1)
7	111 (Hamming distance = 1)	100 (Hamming distance = 1)

Standard binary code b= $b_n...b_1 \mapsto$ Gray code g= $g_n...g_1$:
g = b $\oplus$ (b $\gg$1)

Example: 110 $\oplus$ (110 $\gg$ 1) =110 $\oplus$ 011 = 101

Gray code g $\mapsto$ Standard binary code b:
$b_j = \oplus \sum_{k=j}^{n} g_k$

Figure 7.14: Gray code

Example: Standard binary code to Gray code

b	1	0	1	0	0	1	1	1
$b \gg 1$	0	1	0	1	0	0	1	1
$b \oplus (b \gg 1)$	1	1	1	1	0	1	0	0

Example: Gray code to standard binary code

$g = g_n...g_1$	1	1	1	1	0	1	0	0
	1	2	3	4	4	5	5	5
$b_j = \oplus \sum_{k=j}^{n} g_k$	1	0	1	0	0	1	1	1

Figure 7.15: Example of Gray code to standard binary code and vice versa

7.10 Evolution - Replacement schemes

Many options, many tradeoffs (diversification vs. fitness improvement). E.g.:

- **Generational replacement:** Replace all individuals of a generation
- **Strict elitism:** Keep only m best individuals for next generation
- **Weak elitism:** m best individuals are mutated to obtain the next generation
- **Delete**-n: Replace n random individuals
- **Delete**-n-**last:** Replace n worst individuals (and keep the others)
- **„Retirement home“:** Store a part of the population in a „retirement home“ for some time, give it another chance to procreate later on in the process

7.11 Selection - Choice of parents

Individuals which are not adapted to their environment will be chased and become extinct. Only well fitted individuals survive and produce offspring. Remember example white and brown rabbit!

7.11.1 Selection pressure

Better individuals should have a higher chance of reproduction.

- **Better exploration** of search space when selection pressure is **low**.
- **Better exploitation** of good individuals when selection pressure is **high**. Be careful with dominant solution candidates $\Rightarrow$ crowding

Strategy: Low selection pressure early on, increasing selection pressure in later generations. (Compare with simulated annealing.)

Many options and tradeoffs between diversification vs. exploitation of good individuals:

- (Unbiased) Random selection
- Random selection with bias on better individuals: Roulette wheel selection
- Tournament selection
- ...

7.11.2 Tournament selection

Sample k individuals randomly from population.

Choose best individual with probability p.
Choose second-best individual with probability $(1-p)p$.
Choose third-best individual with probability $(1-p)^2p$.
$\vdots$
Choose (k-1)-best individual with probability $(1-p)^{(k-2)}p$.
Choose k-best individual with probability $(1-p)^{(k-1)}$.

$p = 1$ gives deterministic tournament selection.
What happens for $k = 1$ and for $k =$ size of population?
The larger k, the higher the selection pressure.

Figure 7.16: Tournament selection

Advantages:
Based on ranking and selection pressure can be adjusted.

7.12 Recombination

Crossing of individuals to exchange and pass advantageous properties.

7.12.1 one-point crossover

Figure 7.17: one-point crossover

7.12.2 two-point crossover

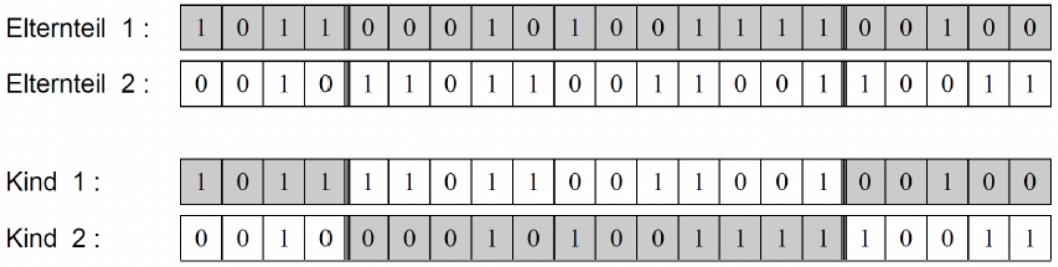


Figure 7.18: two-point crossover

7.12.3 uniform crossover

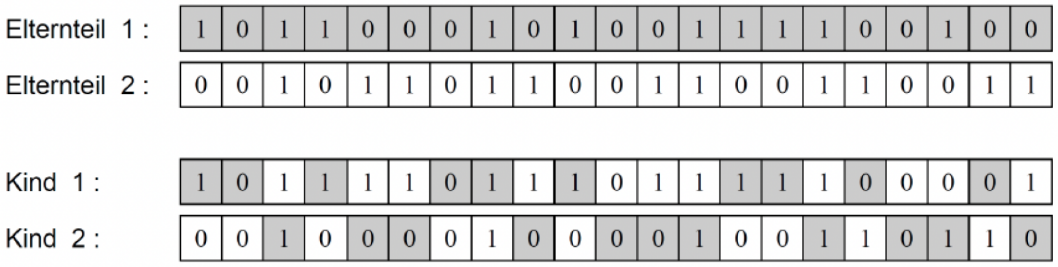


Figure 7.19: uniform crossover

7.12.4 PMX operator

PMX operator (2-point crossover with repair mechanism):

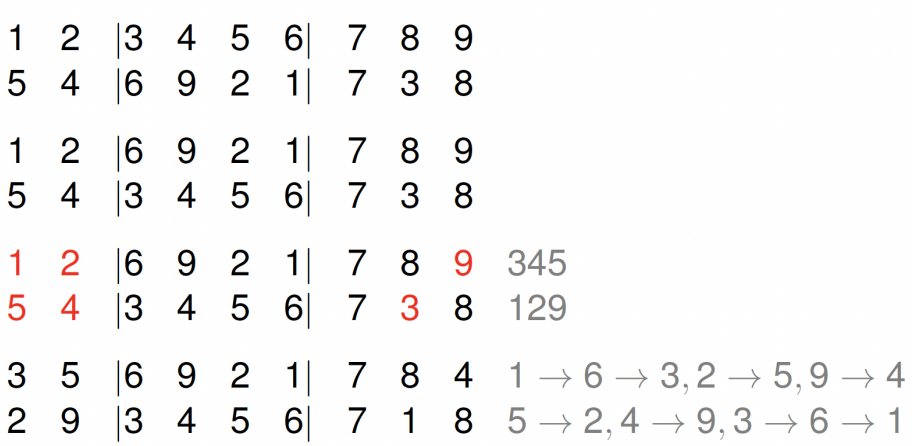


Figure 7.20: PMX operator

7.13 Mutation

Options:

- bit-flip
- positional mutation
- inversion
- ...

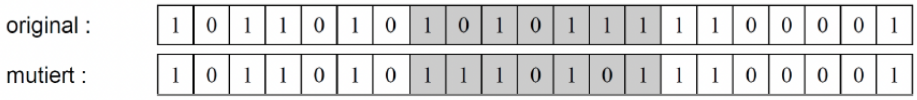


Figure 7.21: Mutation one flip

7.13.1 Mutation: Examples for TSP

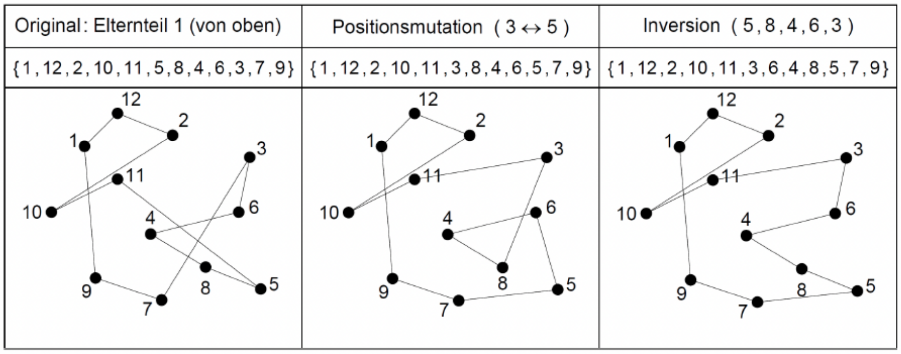


Figure 7.22: Mutation: Examples for TSP

Note: Inversion corresponds to 2-change (here with edges (11,5) and (3,7).

7.14 Combined scheme

Combination of different possibilities.

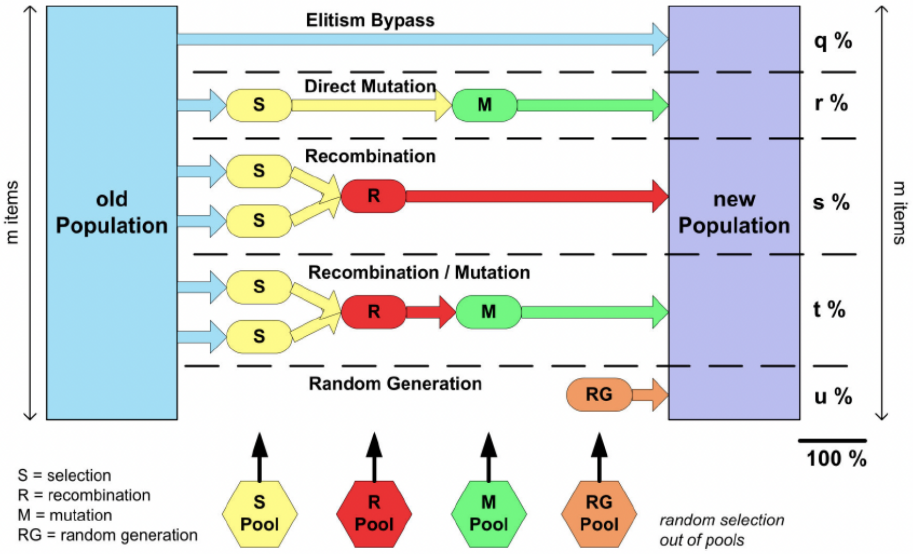


Figure 7.23: Combined scheme

7.14.1 Classification according to Hertz and Kobler

Genetic algorithms

1. **Individuals:** Solutions
2. **Evolutionary process:** Depends
3. **Neighborhood structure:** Unstructured
4. **Information sources:** Selection, choice of parents, recombination of genes
5. **Restriction-violating solution candidates:** Discard, penalize or repair
6. **Intensification strategy:** None
7. **Diversification strategy:** Mutations

Chapter 8

Exercise 13 and Examples

8.1 Comprehension questions on Hill Climbing

1. **What are the advantages and disadvantages of randomized hill climbing in comparison to the gradient method in the optimization of a continuous function?**

   Hill climbing is simple to implement, as there is no need to calculate the gradient or the step size and therefore it does not require much computation time to propose a new solution candidate. The disadvantage of hill climbing is that a random point is selected from the neighborhood and the acceptance probability of the proposed solution can become very small. In the gradient method a good direction is chosen automatically and then the step size is adapted. This means information about the shape of the function is used, so that the algorithm does not scrape around in the search space, but rather the function is minimized in a targeted way.

2. **What are the strengths and weaknesses of the gradient method and of randomized hill climbing when searching for global and local optima?**

   The gradient method is geared towards successively improving an existing approximate solution and will therefore move into the nearest local optimum. The gradient method is not designed for searching for global optima. With the hill climbing method, local and global optima can be searched for by the neighborhoods from which a random solution is proposed being defined to be correspondingly narrow or wide.

3. **What kind of problems occur when optimizing a function with the randomized hill climbing algorithm if you choose the newly proposed solution candidates from neighborhoods that are too small or too large?**

   If the proposed solution candidates are taken from too narrow a neighborhood, then hill climbing fails to escape from local optima. An additional problem is too slow convergence, since the step sizes are too short. If the neighborhoods chosen are too large, the convergence into local and global optima is difficult, because the local space around an optimum is not accurately searched enough, since new solution candidates are proposed from too distant neighborhoods instead.

4. **How can you ensure whether you have found the global optimum with your hill climbing algorithm?**

   One could run hill climbing from many different starting point in order to verify whether hill climbing converges to the same solution in all cases.

8.2 Vertex cover problem

You are given a graph $G = (V, E)$. In the vertex cover problem, we are looking for a smallest possible set of vertices $U \subseteq V$ so that every edge of the graph is incident to a vertex from U.

1. **Propose a heuristic for obtaining a good starting solution.**

Successively choose a vertex which contains the most edges not yet connected, and include it with the set U. Continue like this until all edges are covered.

2. **Define a *possible encoding* of the optimization problem for a genetic algorithm.**

   One possible encoding would be a binary vector, where each element stands for a vertex. The entry 1 means that the vertex is selected and the entry 0 means that the vertex is not selected.

   A *suitable fitness* function would be the number of zeros in the vector. The more zeros are included in the vector, the fewer vertices are used in order to cover all edges. However, this only makes sense for individuals which represent valid solution candidates.

   A *fitness function which allows invalid solution candidates* would need to have a corresponding penalty term: e.g. the number of zeros minus c times the number of edges not covered, for some constant $c > 0$.

3. **Specify possible modification steps, such as mutations and recombination. What kind of problems occur and how can you deal with them?**

   We can use the standard mutations and recombination procedures here, i.e. one-/two-point or uniform crossover and point or position mutations.

   However, most of these operations will lead to invalid solutions quite often. We could either (1) keep invalid solutions but penalize them in our fitness function (see b)), or (2) we could first repair and then intensify them. That is, we first add vertices until all edges are covered (e.g. by the heuristic suggested in a)), and then try to leave out superfluous vertices. (A third possibility for dealing with invalid solutions would be to simply throw all of them away and generate enough offspring in each generation to make sure there are enough valid solutions to continue. However, usually the other two approaches work better.)

   Note that point mutations will not be very useful in a genetic algorithm that only allows valid solutions, because a good solution will almost always turn into an invalid one or a worse one. Also the other operations will quite often produce worse or invalid solutions. As outlined above, these issues can be dealt with appropriate repair and intensification procedures.

8.3 Assignment problem

Consider the following assignment problem: You are given n workers, who have to perform n jobs. Each worker is to be assigned exactly one job, in such a way that the total cost is minimized. The costs for job i, performed by worker j, are given by the entries c_{ij} in a cost matrix.

1. **Define an encoding of the optimization problem suitable for application in a genetic algorithm.**

   A *less effective encoding* would be a binary $n \times n$-matrix, where a 1 in row i and column j means that job i is performed by worker j. The matrix could then be assembled row by row to form a vector.

   A *better type of encoding* would be a vector of length n, with each entry corresponding to one worker. Each job is also given a number between 1 and n, which is written into the vector at the position corresponding to the worker who will perform the job. A valid vector therefore contains each number exactly once, i.e. can be viewed as a permutation (as in the TSP example discussed in the lecture).

2. **Specify possible modification steps, such as mutations and recombination. What kind of problems occur and how can you deal with them?**

   A suitable mutation is the position mutation. It corresponds to two workers switching jobs and will never create invalid solutions.

   In the case of (e.g. two-point) crossover, invalid solutions will arise because typically the two children will no longer be permutations, i.e. they contain unassigned jobs and jobs assigned to two workers. A possible repair mechanism would be to assign jobs of the second kind to the worker which has lower cost for the job in question, freeing up the other worker. The unassigned jobs can then be assigned to the free workers, e.g.

in increasing order of cost or by some other heuristic. As an intensification strategy, one could then possibly try to save costs by swapping pairs of jobs.

3. **Specify possible neighborhoods which you could use e.g. in the hill climbing method.**

   A neighborhood could e.g. be defined as all solutions obtained by permuting 2, 3 or more jobs. Permuting two jobs corresponds to a position mutation. Permuting three jobs can be seen as two position mutations which coincide in one of the positions.

8.4 Knapsack problem

In the knapsack problem you have the task of packing valuable items without exceeding a specified total weight G. The weights of the n objects are given by w_i and the values by v_i, $i = 1, \ldots n$. Your goal is to maximize the total value of the packed items.

1. **Devise a heuristic strategy to solve the problem. Does your strategy always give an optimal solution? Why (not)?**

   A reasonable greedy heuristic is to sort the items according to the ratio of value to weight and successively pack the item with the largest possible value/weight ratio until no more items can be added. The procedure is not guaranteed to be optimal, because it could be the case that only very heavy and valuable items are left over, one of which could have been added if a very small item with slightly higher value/weight ratio (but lower actual value) had been left out.

2. **Propose an encoding of the optimization problem for application in a genetic algorithm.**

   A *suitable encoding* would be a binary vector, where a 1 entry means that the item is packed and a 0 entry means that the item is not packed. (In view of the above heuristic, it might be useful to sort the vector positions according to the value/weight ratio of the corresponding items.)

   A *suitable fitness function* for a genetic algorithm in which only valid solution are allowed is the total value of the packed items. Otherwise, a penalty term would have to be added, which penalizes the maximum weight being exceeded. For example: Fitness is equal to total value of the packed items minus a factor c times excess weight.

3. **What are possible operations for mutations and crossover in the algorithm?**

   A *suitable mutation* would be a position mutation at two points in the genome with different values. That is, the inclusion of one item is replaced by the inclusion of another item.

   In the case of a *recombination*, e.g. a one-point, two-point or uniform crossover, a portion of the items from one parent is replaced by a portion of the items from the other parent. Typically this will create one valid and one invalid solution as offspring. We could either use a repair and intensify approach similar to Task 2 (i.e. drop excess items and greedily add items if there is space left), or just accept these offspring as they are and penalize restriction-violating ones as outlined in b).

   If we allow invalid solution candidates, the weighting factor c of the penalty term should be dynamically adjusted. If there are too many invalid candidates in the population, c should be increased, if there are very few, we might lower it. In that spirit, we should start out with a moderate value of c and increase it towards the end of the algorithm in order to obtain good solutions which satisfy and exhaust the weight constraint.

8.5 Exact cover problem

In the exact cover problem, n subsets U_i of a base set M are given and you have to determine whether there is a choice of subsets U_i such that each element of M is contained in exactly one of the chosen sets. (I.e, the union of the chosen subsets should be M, and no element should occur in two or more of the chosen sets.) The optimization variant of this problem is as follows: Find a choice of subsets U_i such that each element of M is contained in one of the chosen sets, and the sum of the cardinalities of the selected subsets is as small as possible.

1. **Devise a heuristic strategy to find an approximate solution.**

   Successively select the subset U_i which has the smallest intersection of elements with the elements selected so far. Sets with greater cardinality should be preferred, because this means the sets of lower cardinality remain to fill in the gaps later in the process.

2. **Give an encoding of the optimization problem for application in a genetic algorithm, a suitable fitness function and possible operations for mutations and crossover.**

   A *suitable encoding* would be a binary vector, where each element stands for a subset U_i. The entry 1 means the corresponding subset is selected and the entry 0 means that the subset is not selected. With regard to the heuristic strategy one could represent the sets in the vector ranked according to their cardinality.

   A *suitable fitness function* in the case of only valid solution candidates would be the negative sum of the cardinalities of the selected sets.

   A point mutation then almost always leads to a solution that is either worse or invalid, since a subset with elements is removed or added. Similarly, a non-trivial position mutation (i.e. one between positions with values 0 and 1) frequently leads to invalid solutions. Also most non-trivial recombinations will likely lead to invalid solutions, since e.g. in the case of a one or two-point crossover, some sets with elements will be removed and other sets with other elements will be added. In order to obtain good and valid solutions again, one would have to first repair the invalid solutions and then intensify them. That is, first add sets until a valid solution emerges and then try to remove redundant sets.

   An additional intensification strategy to improve valid solution would be the following: Try to remove one set at a time and to replace it by a better one. One could also remove two or more sets and then use the heuristic from a) to fill up again until all elements are covered.

8.6 Example - N Queens

1. **What is a good heuristic for the N Queens problem**

   The n-queens problem is a classic problem in artificial intelligence and computer science, which involves finding a placement of n queens on a chessboard such that no two queens threaten each other. One possible heuristic for solving the n-queens problem is the "min-conflicts" heuristic.

   The min-conflicts heuristic works by starting with a randomly generated initial solution and iteratively modifying it to reduce the number of conflicts, or pairs of queens that attack each other. At each step, the heuristic selects the queen that is involved in the most conflicts and moves it to a new location that minimizes the number of conflicts. The process continues until a solution is found or a maximum number of iterations is reached.

   The min-conflicts heuristic is a simple and effective approach for solving the n-queens problem and has been shown to find solutions quickly for moderate-sized problems. It can be combined with other search techniques, such as hill climbing or simulated annealing, to further improve its performance.

2. **How would a suitable genetic algorithm look like**

   The n-queens problem can be encoded as a genetic algorithm by using a chromosome representation that encodes the position of each queen on the chessboard. Here is one common encoding method:

   (a) Encoding/Chromosome representation: Each chromosome represents a candidate solution to the n-queens problem and is a permutation of the integers from 0 to n-1. The i-th element of the chromosome represents the row number of the queen in the i-th column.

   The chromosome representation for the n-queens problem is a way to encode a potential solution to the problem as a string of values. Each chromosome represents a possible arrangement of n queens on an n x n chessboard, and the values in the chromosome correspond to the row positions of the queens in each column of the chessboard.

For example, if n=8, a chromosome could be represented as [4, 6, 2, 0, 5, 7, 1, 3], which means that the queen in the first column is placed in row 4, the queen in the second column is placed in row 6, and so on. The chromosome is a permutation of the integers from 0 to n-1, and each value in the chromosome corresponds to a row position for a queen in a column.
The chromosome representation provides a compact way to represent a candidate solution to the n-queens problem, and it can be easily manipulated by the genetic algorithm to explore different arrangements of queens on the chessboard. The chromosome representation is also easily transformed into a graphical representation of the solution, making it easy to visualize and understand.

(b) Initial population: A random initial population of chromosomes is generated, with each chromosome representing a potential solution to the n-queens problem.

(c) Fitness function: A fitness function is used to evaluate the quality of each chromosome. The fitness function assigns a lower score to chromosomes that have more queens attacking each other, and a higher score to chromosomes that have fewer conflicts.

(d) Selection: The chromosomes are selected for reproduction based on their fitness values, with the fittest chromosomes being more likely to be selected.

(e) Crossover: Two selected chromosomes are combined to create offspring by exchanging parts of their permutation representation. This generates new candidate solutions that inherit traits from both parents.

(f) Mutation: Small random changes are introduced into the offspring chromosomes to explore new solutions and avoid getting stuck in a local optimum.

(g) Repeat: The steps 4-6 are repeated until a satisfactory solution is found or a maximum number of generations is reached. The best chromosome from the final population is taken as the solution to the n-queens problem.

This encoding method is a simple and effective way to represent the n-queens problem as a genetic algorithm and allows for the exploration of different candidate solutions through genetic operations such as crossover and mutation.

Bibliography