Optimisation HS19

Pascal Baumann
pascal.baumann@stud.hslu.ch

1. Juli 2024

For errors or improvement raise an issue or make a pull request on the github repository.

Contents

1 Introduction

The field of Optimisation can be separated into two distinct fields:

- Quantitative Analysis and Optimisation:
  Based on quantifiable information and knowledge, such as numerical, measurable data, mathematical models and algorithms
- Qualitative Analysis and Optimisation:
  Based on non-quantifiable information and knowledge, such as informal facts, verbal description of processes and procedures, unstructured information, experience, and implicit knowledge

In this summary only the former will be discussed, although the latter is often encountered in a business consulting setting (often up to 80%).

We differentiate further between *Continuous Optimisation*, where there are infinitely many solutions represented by continuous variables, and *Discrete Optimisation*, with only finitely many solutions represented by integer variables. The main body of theory for continuos optimisation is *Local Optimisation*, whose methodology is usually based on first and second order derivatives. For discrete optimisation there exists a trivial, finite solution algorithm, the Enumeration of all solutions, though this is often impracticable for practical purposes, where the goal is to find good, or preferably optimal, solutions within reasonable time.

1.1 Linearity

Because discrete optimisation is concerned with a finite amount of solutions, which can be represented by a list of variables. This list is a finite set in n-dimensional space, whose contour is linear (due to the finiteness of the set). The convex hull of a finite set of points is a convex polyhedron, and every discrete optimisation problem can be modelled as a linear problem.

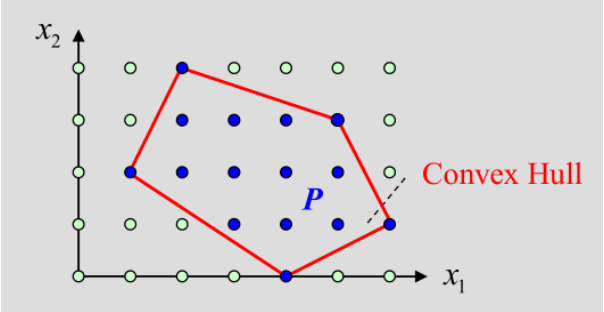


Figure 1.1

1.2 Decision Problems

Discrete Optimisation can help **decision makers** with support through (i) quantitative models and (ii) qualitative approaches. It's possible that there are multiple **alternatives** with associated **consequences**, which may be of a **deterministic** or **stochastic** nature. The focus of this course lies in the **evaluation** of these alternatives with regard to their consequences.

The evaluation of consequences is done on two levels: Satisfaction and Optimisation. Satisfaction means that the consequence has to fulfil certain **constraints** in order to have a **feasible** alternative. While Optimisation means that the consequence has to reach the best possible value, that is it must be **optimal** among all feasible alternatives.

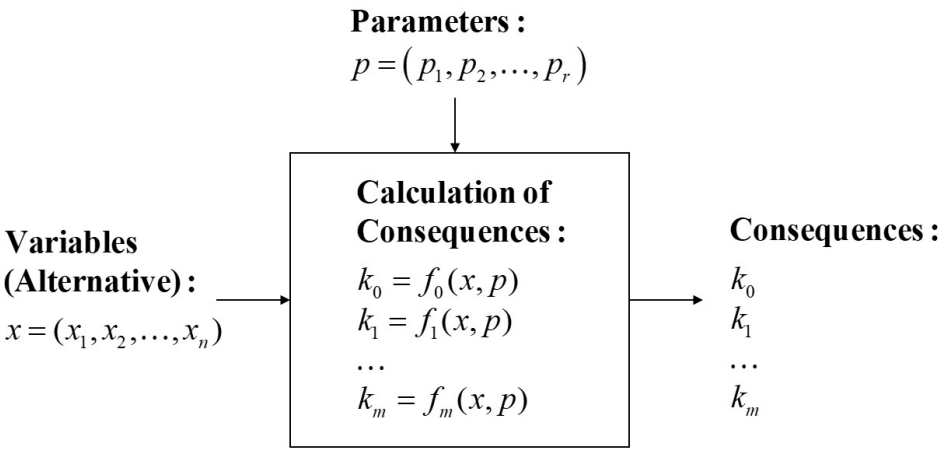


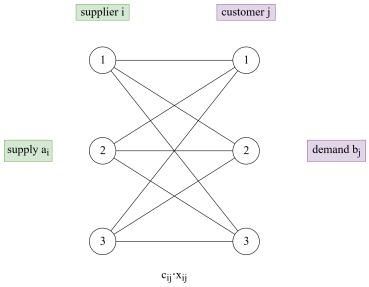
Figure 2.1: A descriptive model

2 Mathematical Models

2.1 Descriptive Models

Are also called Evaluation Models and answer the question "What If?" by calculating the resulting consequences of a given alternative.

2.1.1 Vehicle Dispatching in a Car Rental Company



Consequences:

k_0	Total distance of all transfers [km]
k_i^{Out}	Number of vehicles transferred out of location i, $i \in I$
k_j^{In}	Number of vehicles transferred into location j, $j \in J$

Model:

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

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

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

2.2 Optimisation Models

Are also called Prescriptive Models and answer the question "What's best" through calculating an **optimal** alternative in the set of all **feasible** alternatives. This set of feasible solutions is called the **solution space**, which is defined by **constraints** and is expressed in form of inequalities and equations. The one selected consequence to be optimised is called the **objective function**.

With these models the user does not have to specify alternatives, as they are specified by the constraints defining the solution space. The model itself does not calculate the solution however, **Optimisation algorithms** are needed for this.

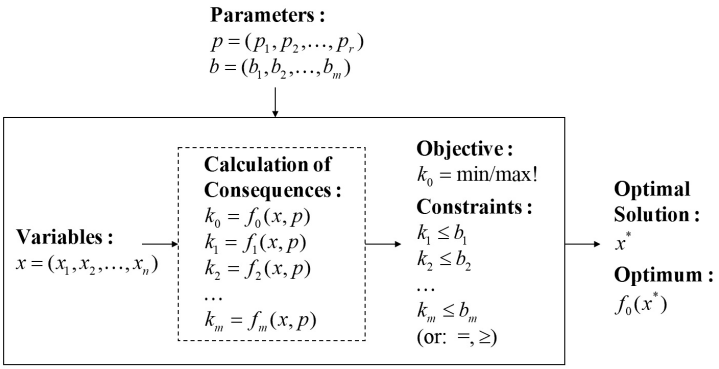


Figure 2.2

2.2.1 Formulation of an Optimisation Model

Sets:

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

Parameters:

a_i Number of available vehicles at location i, $i \in I$

b_j Number of requested vehicles at location j, $j \in J$

c_{ij} Distance [km] from location i to location j, $i \in I, j \in J$

Variables:

x_{ij} Number of available vehicles at location i, $i \in I$

Constraints:

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

$$\sum_{i \in I} x_{ij} \ge b_i \qquad j \in J$$

$$x_{ij} \ge 0 \qquad i \in I, j \in J$$

Objective Function:

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

Optimisation models involve **mathematical definition of solution space**, and have the information for the construction of feasible alternatives integrated. However, they do not have mechanisms for finding optimal solutions, these require the use of optimisation algorithms. Depending on the model type there are various different algorithms. The field studying optimisation models and algorithms is called **Operations Research**.

2.2.2 Notational Conventions

$\mathbb{R}, \mathbb{Q}, \mathbb{Z}$	set of real, rational and integer numbers
$\mathbb{R}_0, \mathbb{Q}_0, \mathbb{Z}_0$	corresponding sets constrained to nonnegative numbers
$\mathbb{R}^n$	set of n-dimensional vectors with components in $\mathbb{R}$

vectors are regarded as **column** vectors

S^n set of n-dimensional vectors with components in S
$\{0,1\}^3$ is the set of binary vectors with length 3

vectors are written in bold face, component in normal face with subscripts

multiple vectors are written using superscripts

$$\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^i = \begin{pmatrix} x_1^i \\ x_2^i \\ \vdots \end{pmatrix}$$

$\mathbb{R}^{m \times n}$ set of $m \times n$-dimensional matrices (m rows, n columns) with elements in $\mathbb{R}$

$S^{m \times n}$ set of $m \times n$-dimensional matrices with elements in S

Matrices are written in bold face and capitalised, elements in normal face with subscripts

Special notation:
i-th row of matrix $\mathbf{A}$ is denoted $\mathbf{a}^i$
j-th column of matrix $\mathbf{A}$ is denoted $\mathbf{A}_j$

$S' \subseteq S$ S' is a subset or equal to set S

2.2.3 Problem Formulation

General Optimisation Problem:	$\prod : \max\{f(\mathbf{x}) : \mathbf{x} \in S\}$
Decision variables:	$\mathbf{x} = (x_1, x_2, \ldots, x_n)^{\mathrm{T}} \in \mathbb{R}^n$
(Feasible) solution:	$\mathbf{x} \in S$
Solution space:	$s \subseteq \mathbb{R}^n$
Objective function:	$f : S \to \mathbb{R}$
Optimal solution:	$\mathbf{x}^* \in S$ such that $f(\mathbf{x}^*) \geq f(\mathbf{x}) \forall \mathbf{x} \in S$
Optimum:	$f(\mathbf{x}^*)$

2.2.4 Conditions for Existence of Optimum

Feasibility:	$S \neq \emptyset$
Boundedness:	feasible, $\exists \omega : f(\mathbf{x}) \leq \omega \forall \mathbf{x} \in S$
Closedness:	feasible, bounded, optimum exists

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

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

Example for Unclosedness $\max\{x_1 : x_1 \leq 1, \mathbf{x} \in \mathbb{R}\}$

2.2.5 Definition of Solution Space

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

ii) **Non-Functional Constraints**:
$S = \{\mathbf{x} \in G_1 \times \ldots \times G_n : \text{"}\mathbf{x} \text{ has certain properties"}\}$

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

2.3 Definitions and Concepts

2.3.1 Problems 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 $n, \mathbf{a}, b, \mathbf{c}$

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

2.3.2 Neighbourhoods and Local Optima

To define the concept of a local optimum a notion of a local neighbourhood is needed. The idea: A solution $\mathbf{x} \in S$ is locally optimal, if it is the best solution within a certain neighbourhood $N(\mathbf{x})$. A neighbourhood is a set of neighbour solutions: $N(\mathbf{x}) \subseteq S (\mathbf{x} \in N(\mathbf{x})$ is assumed). In general the Euclidean (or ε-) neighbourhood is used: open sphere around $\mathbf{x}$ with radius ε. However, often other neighbourhoods are used in **discrete optimisation**.

General Neighbourhood: A Set-Valued Function Neighbourhood $N : S \to P(S)$ where $S \subseteq \mathbb{R}^n$, $P(S)$ Powerset of S, $\mathbf{x} \mapsto N(\mathbf{x})$ (Powerset $P(S)$ of a set S is the set containing all subsets of S)

Neigbour Solutions: $N(\mathbf{x}) \subseteq S$ where $x \in N(\mathbf{x})$ assumed

Euclidean Neighbourhood, ε-Neighbourhood $N_\varepsilon(\mathbf{x}) = \{\mathbf{y} \in \mathbb{R}^n : \|\mathbf{y} - \mathbf{x}\| < \varepsilon\}$ where $\varepsilon > 0$, Euclidean Norm: $\|\mathbf{x}\| = \sqrt{\mathbf{x}^2} = \sqrt{x_1^2 + \cdots + x_n^2}$. It's a fundamental neighbourhood concept in analysis, and describes the geometric or topological proximity between points. Local optimality (with respect to N_ε) therefore refers to $\mathbf{x}$ being the best solution in the neighbourhood $N_\varepsilon(\mathbf{x})$.

2.3.3 Local Search Metaheuristics

The principle of Local Search Methods:

i. Given:
A feasible solution: $\mathbf{x} \in S$
A neighbourhood function: $\mathbf{x} \mapsto N(\mathbf{x})$

ii. Repeat:

   (a) Search for a better solution $\mathbf{x}'$ in the neighbourhood $N(\mathbf{x})$

   (b) If no better solution is found: Stop $\to$ The current solution $\mathbf{x}$ is locally optimal with respect to N.

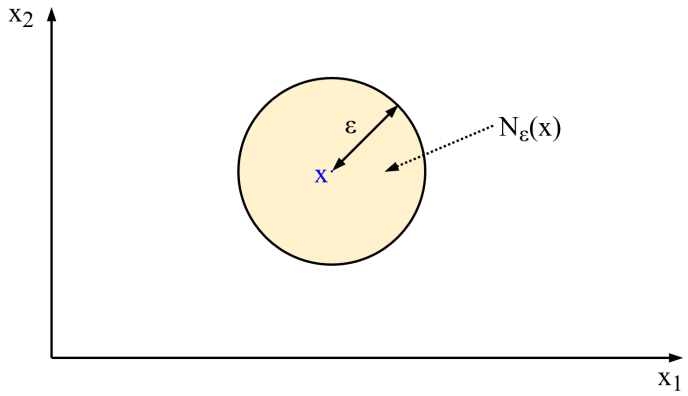


Figure 2.3: Visualisation of the Euclidean Neighbourhood

(c) Otherwise: Take $\mathbf{x}'$ as the new current solution, $\mathbf{x} := \mathbf{x}'$

This search process repeatedly moves from current solution $\mathbf{x}$ to a better neighbourhood solution $\mathbf{x}' \in N(\mathbf{x})$, until local optimum is reached, this forms a trajectory in solution space.

2.3.4 Global and Local Optima

Globally optimal solution $\mathbf{x}^*$: $f(x^*) \geq f(\mathbf{x}) \forall \mathbf{x} \in S$
Locally optimal solution $\mathbf{x}^*$: $f(x^*) \geq f(\mathbf{x}) \forall \mathbf{x} \in N(\mathbf{x}) \cap S$
The intersection of neighbourhood with S means, that only neighbours that are solutions are considered. Local optimality is always with respect to some neighbourhood $N_\varepsilon(\mathbf{x})$.

2.3.5 Topological Notions

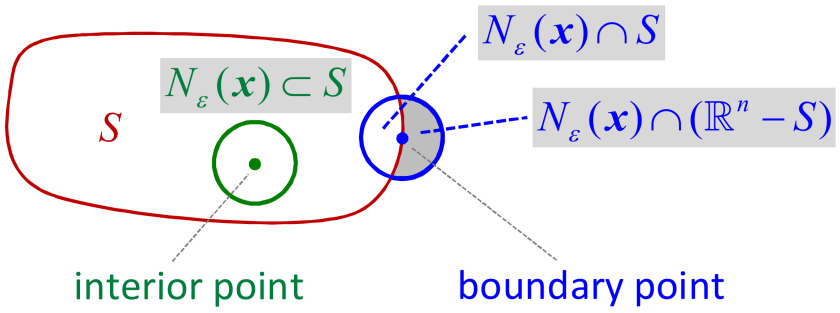


Figure 2.4: Illustration of interior and boundary point

S closed	all boundary points of S are in S
S open	all points of S are interior points
S bounded	$S \subseteq \{\mathbf{x} \in \mathbb{R}^n : \mathbf{a} \leq \mathbf{x} \leq \mathbf{b}\}$

Figure 2.5: Illustrated difference between open and closed sets

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.*

This is a fundamental theorem in Analysis and Continuous Optimisation.

Level Sets Given the graph of a function $f : S \mapsto \mathbb{R} : H = \{(\mathbf{x}, f(\mathbf{x})) : \mathbf{x} \in S\}$, a 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.

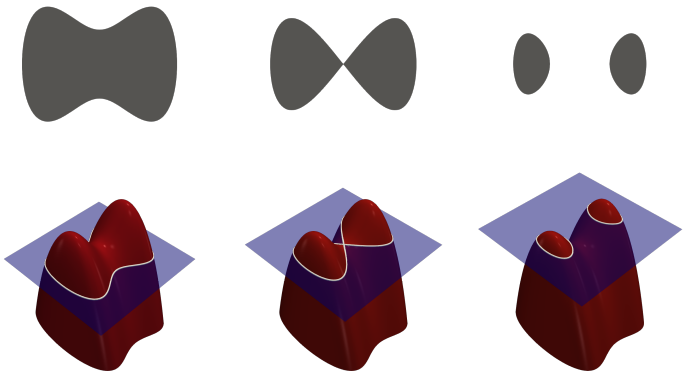


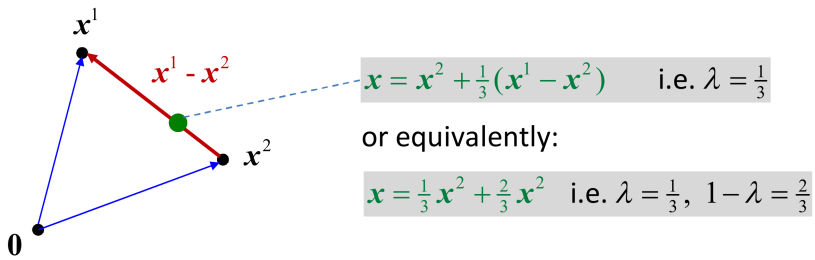
Figure 2.6: Illustration of level sets

2.4 Convex Optimisation

Convex optimisation 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.

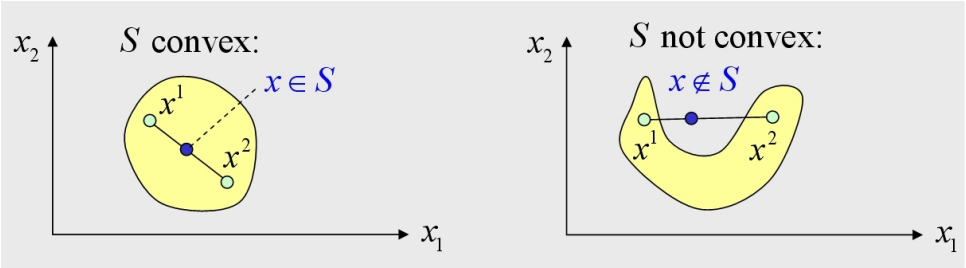


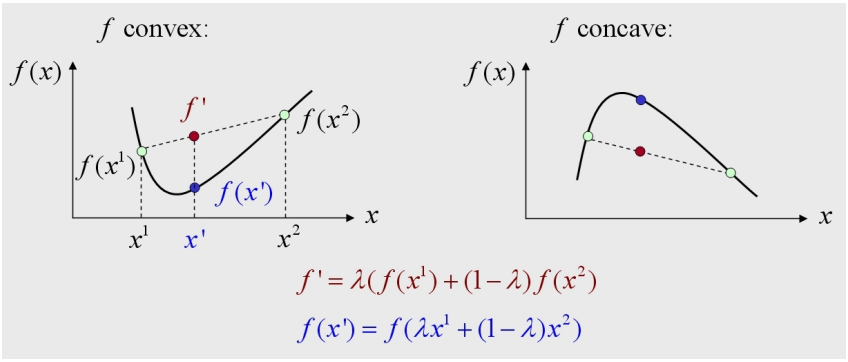
Figure 2.7: Illustration of convex and non-convex sets

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

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

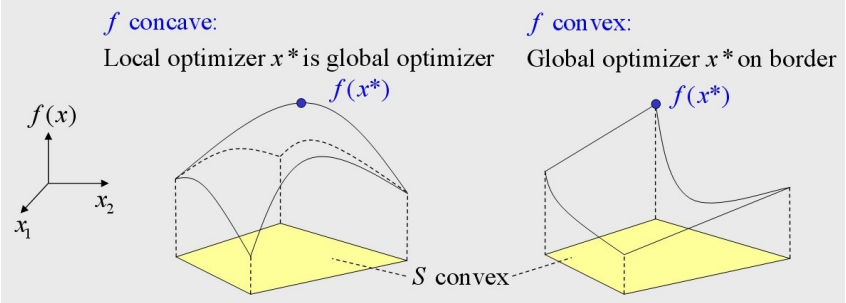
Concave function $f : S \to \mathbb{R}$, where S convex:

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



In convex optimisation a linear function can be convex and concave, both $\min\{f(\mathbf{x} : \mathbf{x} \in S)\}$ with f convex and S convex, and $\max\{f(\mathbf{x} : \mathbf{x} \in S)\}$ with f concave and S convex are **convex optimisation problems**.

Theorem 2 *In a convex optimisation problem, local optimums is a global optimum.*



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 convex.
Proposition: $S = \{\mathbf{x} \in \mathbb{R}^n : g_i(\mathbf{x}) \leq b_i, i \in I\}$ is convex (intersection of convex sub-level sets).

3 Linear Programming

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

$$f(\mathbf{x}) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j = \mathbf{a}^T\mathbf{x} \quad (\mathbf{a} \in \mathbb{R}^n)$$

Linear inequality $\sum_{j=1}^{n} a_{ij}x_j \leq b_i, i \in I$ or $\mathbf{a}^i\mathbf{x} \leq b_j, i \in I$ or $\mathbf{Ax} \leq \mathbf{b}$. This is analogous for linear inequalities of type "$\geq$" and linear equations "=". The general definition of linear functions f is as follows:

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

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

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

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

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

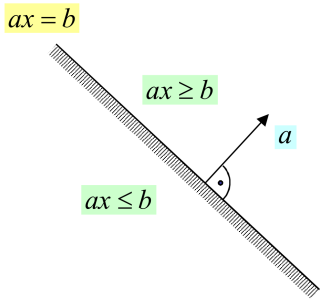
3.1.5 Transformations

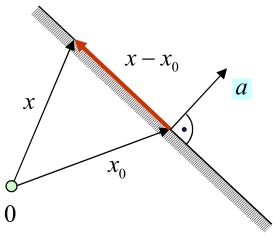
- **Inequalities to Inequalities**
  $\mathbf{a}^i\mathbf{x} \leq b_i \quad \leftrightarrow \quad -\mathbf{a}^i\mathbf{x} \geq -b_i$
- **Equalities to inequalities**
  $\mathbf{a}^i\mathbf{x} = b_i \quad \rightarrow \quad \mathbf{a}^i\mathbf{x} \leq b_i, \mathbf{a}^i\mathbf{x} \geq b_i$
- **Inequalities 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 $\quad \rightarrow \quad x_j := x_j^+ - x_j^- \qquad x_j^+, x_j^- \geq 0$

3.2 Geometric Aspects

The system of linear inequalities $\mathbf{Ax} \leq \mathbf{b}$ is called linearly indepent, if it's row-vectors $\mathbf{a}^i, i \in I$ are linearly independent.

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





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

Definition: A **polyhedron** $P \in \mathbb{R}^n$ is the intersection of a finite number of halfspaces

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

The defining hyperplanes of P are:

$$\{\mathbf{x} \in \mathbb{R}^n : \mathbf{a}^i\mathbf{x} = b_i\}, i = 1, \ldots, m$$

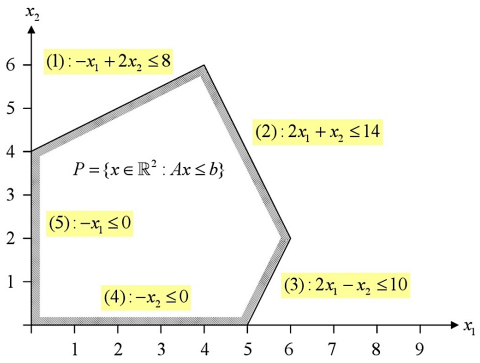
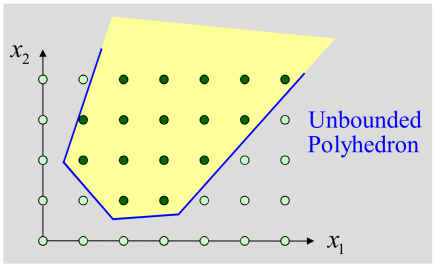
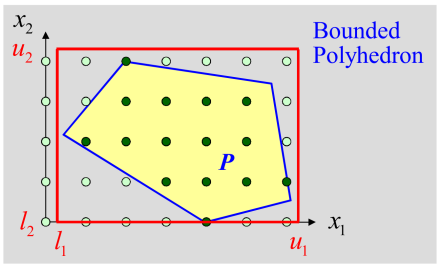


Figure 3.1: Polyhedron in $\mathbb{R}^2$

Definition: A **polytope** in $\mathbb{R}^n$ is a bounded polyhedron, that is a polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ for which there exist $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n$ such that:

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





A polyhedron is a convex set. A point $\mathbf{x} \in \mathbb{R}^n$ is a **vertex** of a polyhedron $P \in \mathbb{R}^n$ if:

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

ii) $\mathbf{x}$ is not a strict convex combination of two distinct points $\mathbf{x}^1, \mathbf{x}^2 \in P$

Proposition: $\mathbf{x} \in \mathbb{R}^n$ is a **vertex** of a polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ if and only if (iff):

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

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

Corollary: $\mathbf{x} \in \mathbb{R}^n$ is a **vertex** of a polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ if and only if (iff):

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

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

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

A selection $B \subseteq \{1, .., m\}$ of $|B| = n$ linearly independent rows of $\mathbf{A}$ is called a **basic selection**, and the corresponding matrix $\mathbf{A}_B$ is called a basis of $\mathbf{A}$. The vector $\mathbf{x} = \mathbf{A}_B^{-1}\mathbf{b}_B$ is called the **basic solution** associated to basis $\mathbf{A}_B$. $B, \mathbf{A}_B$, and $\mathbf{x} = \mathbf{A}_B^{-1}\mathbf{b}_B$ are called **feasible** if $\mathbf{x} \in P$.

3.3 Simplex Algorithm

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

i) Calculate the inverse $\bar{\mathbf{A}} = \mathbf{A}_B^{-1}$ and the feasible basic solution $\mathbf{v} = \bar{\mathbf{A}}\mathbf{b}_B$

ii) Calculate the reduced costs $\mathbf{u}^T = \mathbf{c}^T\bar{\mathbf{A}}$

iii) If $\mathbf{u} \geq \mathbf{0}$ then stop. The vertex $\mathbf{v}$ is optimal

iv) Otherwise it is $\mathbf{u} \ngeq \mathbf{0}$. Choose $j \in B$ with u_j. Define direction $\mathbf{d} = -\bar{\mathbf{A}}_j$

v) Determine $\mathbf{Ad}$

vi) If $\mathbf{Ad} \leq \mathbf{0}$ then stop, because the Linear Program is unbounded along the direction $\mathbf{d}$

vii) Otherwise $\mathbf{Ad} \leq n\mathbf{0}$ holds true, determine λ^*:

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

viii) If a minimum is attained for index k perform a basis change:
$B' = B - j \cup k$. Set $B := B'$ and start anew.

An example calculation can be found in section A.1 in the appendix.

4 Integer Linear Programming

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

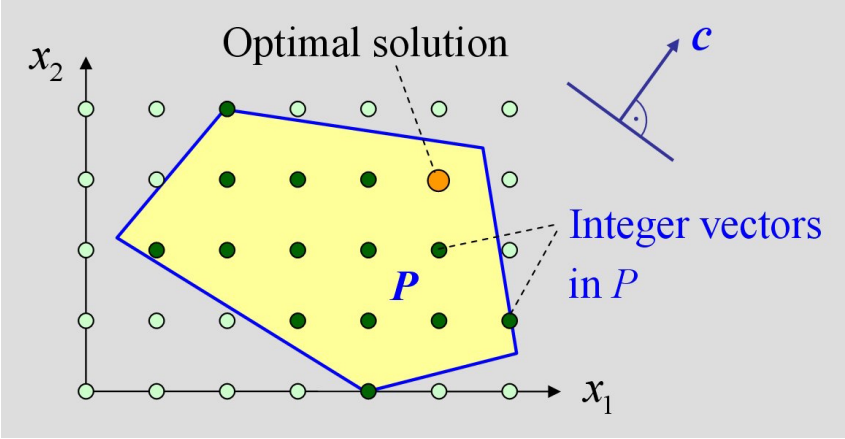


Figure 4.1: Feasible integer solutions inside convex hull P

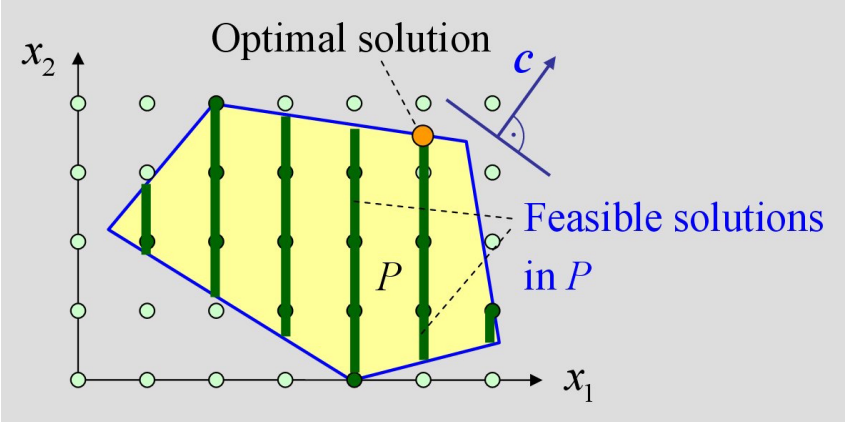
Binary Linear Programming

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

Mixed Integer Linear Programming

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

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



Often integer linear programming is used when the linear program includes binary variables, that is yes or no decisions, and it belongs to the computationally hardest class of problems NP-complete, where only exponential algorithms are known.

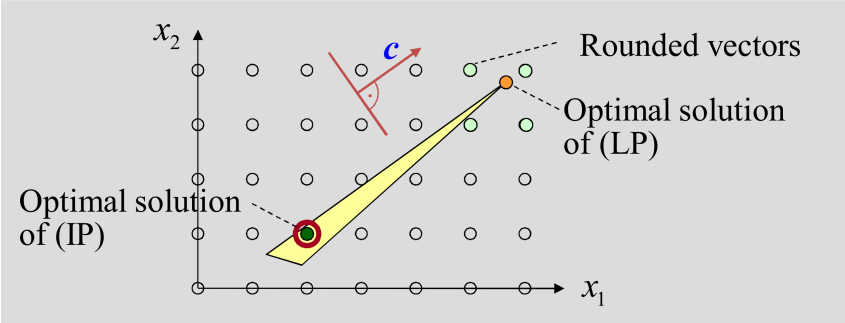
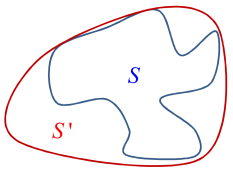


Figure 4.2: Explanation on why rounding down from LP solutions does not work in ILP

4.1 Relaxations

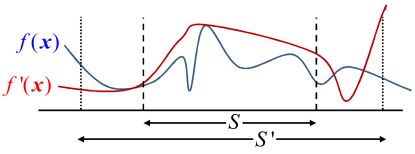
A relaxation of an optimisation problem can be achieved through

- **enlarging the solution space** $S \to S'$ with $S \subseteq S'$ by removing constraints



$$\max\{f(\boldsymbol{x}) : \boldsymbol{x} \in S'\} \geq \max\{f(\boldsymbol{x}) : \boldsymbol{x} \in S\}$$

- **increasing the objective function** $f(\mathbf{x}) \to f'(\mathbf{x})$ with $f'(\mathbf{x}) \geq f(\mathbf{x})$ for $\mathbf{x} \in S$



$$\max\{f'(\boldsymbol{x}) : \boldsymbol{x} \in S'\} \geq \max\{f(\boldsymbol{x}) : \boldsymbol{x} \in S\}$$

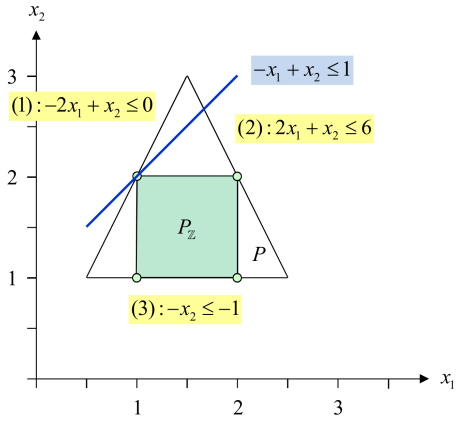
Theorem 3 *Given an optimisation problem* $\prod : \max\{f(\boldsymbol{x}) : \boldsymbol{x} \in S\}$. $\prod' : \max\{f'(\boldsymbol{x}) : \boldsymbol{x} \in S'\}$ *is a* ***relaxation*** *of* $\prod$ *if* $S \subseteq S'$ *and* $f'(\boldsymbol{x}) \geq f(\boldsymbol{x})$ *for* $\boldsymbol{x} \in S$.

4.2 Cutting Plane Method

Theorem 4 *Let* $P \subseteq \mathbb{R}^n$ *be a polyhedro. An inequality* $\boldsymbol{\alpha^T x \leq \beta}$ $(\alpha \in \mathbb{R}^n, \beta \in \mathbb{R})$ *is a valid inequality for* P *if* $\boldsymbol{\alpha^T x \leq \beta}$ *is valid for all* $\boldsymbol{x} \in P$.

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

4.2.1 Gomory-Chvatal Cut



- Choose arbitraty split $p, 1-p$, in this case $p = \frac{3}{4}$
- Combine inequalitites: $\frac{3}{4} \cdot (1) + \frac{1}{4} \cdot (2)$
- $(-\frac{3}{4} \cdot 2x_1 + \frac{3}{4} \cdot x_2) + (\frac{1}{4} \cdot 2x_1 + \frac{1}{4} \cdot x_2) \leq \frac{3}{4} \cdot 0 + \frac{1}{4} \cdot 6 \Leftrightarrow -x_1 + x_2 \leq 1$
- Round down right side
- Resulting new inequality (G-C-Cut): $-x_1 + x_2 \leq 1$

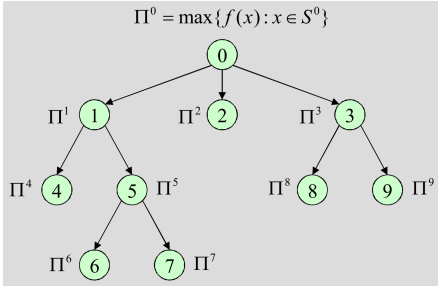
Matrix Notation

- $\mathbf{A} = \begin{pmatrix} -2 & 1 \\ 2 & 1 \\ 0 & -1 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} 0 \\ 6 \\ -1 \end{pmatrix}$
- $\mathbf{u} = (\frac{3}{4}, \frac{1}{4}, 0) \geq \mathbf{0}$
- $\boldsymbol{\alpha} := \mathbf{u^T A} = \mathbf{(-1, 1)^T}$
- $\beta := \mathbf{u}^T\mathbf{b} = 1\frac{1}{2}$
- G-C-Cut: $\alpha^T \mathbf{x} \leq \lfloor \beta \rfloor$

4.3 Branch-and-Bound Method

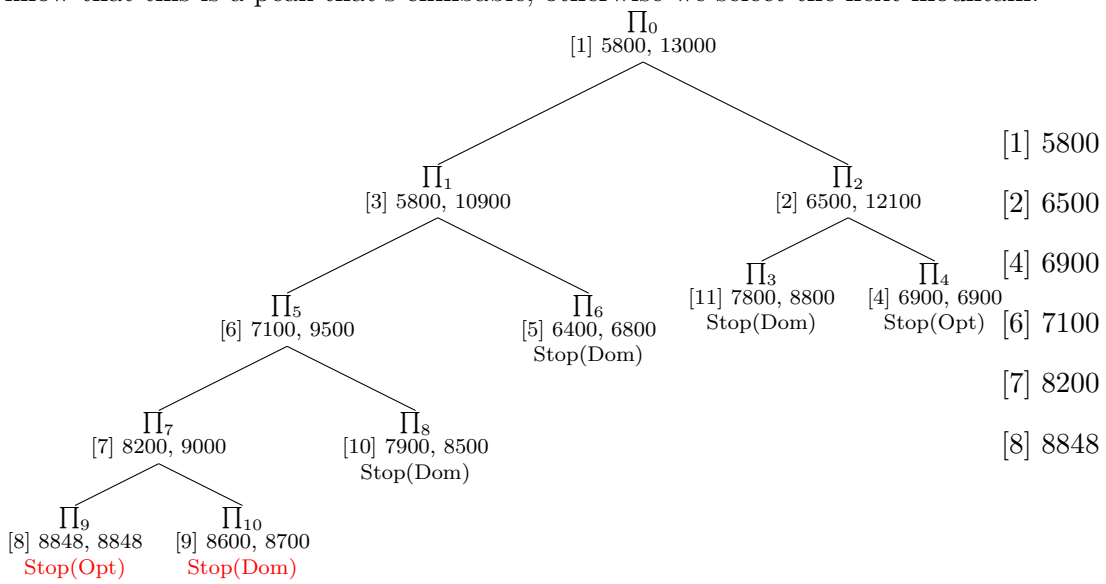
There are two main approaches to solve an Integer Linear Program or a Mixed Integer Linear Program, which are typically combined, **Branch-and-Bound** and **Cutting Planes**. Branch-and-Bound is a general solution method, independent from Integer Linear Program, and the general procedure is as follows:

1. Split the solution space into smaller subspaces or branches iteratively
2. For each sub-problem or branch:
   - 2.i Calculate an upper bound
   - 2.ii Calculate a feasible solution
3. Use this information to prune certain sub-problems or branches



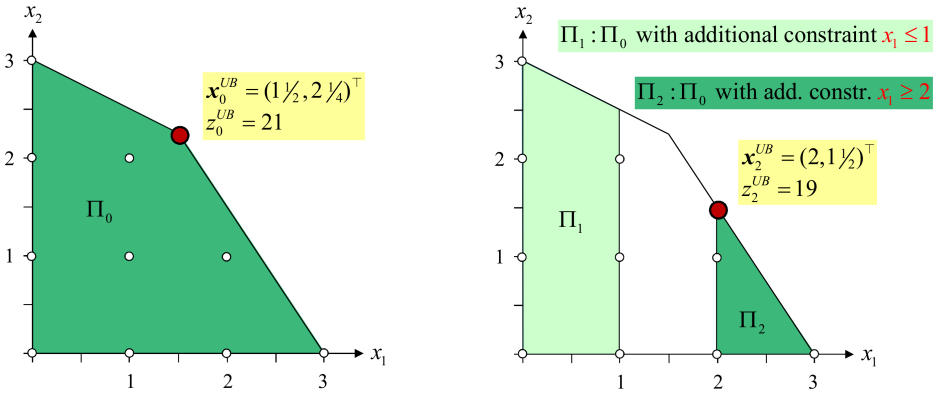
4.3.1 Example

We want to find the highest peak in the area accessible by foot. We send Sherpas to climb the mountains and a helicopter to land on said mountain. If they both arrive at the same height, we know that this is a peak that's climbable, otherwise we select the next mountain.



4.4 Branch-and-Bound for Integer Linear Programming

- Calculation of an upper bound through LP Relaxation
- Branching
  - If LP solution is all integer, stop as the current note is optimal
  - Else choose some fractional variable and round up or down



- Calculation of a feasible solution, that is a lower bound
  i. Simplistic: Do nothing and hope that LP solution is integer
  ii. In practice: Use various heuristic functions

4.4.1 Branch-and-Bound Method for Integer Linear Programming

Given an integer linear program $\prod . \max\{\mathbf{c}^T\mathbf{x} : \mathbf{x} \in P \cap \mathbb{Z}^n\}$ with $P = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{Ax} \leq \mathbf{b}\}$

1. Initialisation: $\hat{r} := 0, P^0 := P, S^0 := P \cap \mathbb{Z}^n, \prod^0 := \max\{\mathbf{c}^T\mathbf{x} : \mathbf{x} \in S^0\}$, $R := \{0\}, z^{LB} := -\infty$
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 $\prod^r$ by solving the Linear Program relaxation $\prod_{LP}^r$: $\max\{\mathbf{c}^T\mathbf{x} : \mathbf{x} \in P^r\}$: Optimum z^{LP} and the optimal solution $\mathbf{x}^{LP}$
      - If $\prod_{LP}^r$ infeasible then declare $z_r^{UB} := \infty$ and go to step 6
      - Otherwise $z_r^{UB} = z^{LP}$
   ii. Otherwise $z_r^{LB} := -\infty$
5. Global Improvement: If $z_r^{LB} > z^{LB}$ then set $z^{LB} := z_r^{LB}$ and $\mathbf{x}^{LB} := \mathbf{x}_r^{LB}$
6. Pruning: Go to step 2 in case of the appearance of one of the following cases
   - $z_r^{UB} = \infty$ (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: $P^{\hat{r}+1} := \{\mathbf{x} \in P^{\hat{r}} : x_j \leq \lfloor x_j^{LP} \rfloor\}, P^{\hat{r}+2} := \{\mathbf{x} \in P^{\hat{r}} : x_j \geq \lfloor x_j^{LP} \rfloor + 1\}$
   Set $S^q := P^q \cap \mathbb{Z}^n$ for $q \in R^r := \{\hat{r}+1, \hat{r}+2\} \rightarrow$ Consider $\prod^q : \max\{\mathbf{c}^T\mathbf{x} : \mathbf{x} \in S^q\}, q \in R^r$.
   Set $R := R \cup R^r, \hat{r} := \hat{r} + 2$.

4.5 Knapsack Problem

The Branch-and-Bound method for Integer Linear Programming can be illustrated with the *Knapsack Problem*. A finite number of objects $j \in J$ are given. Every object $j \in J$ has a certain volume $a_j \geq 0$ and a certain benefit $c_j \geq 0$. The task is to fill the knapsack with capacity b by a selection of the given objects such that the total volume of the selected objects doesn't exceed the capacity of the knapsack and at the same time the total benefit becomes maximal.

Sets:

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

Parameters:

a_j Weight of item $j \in J$

b Capacity of the rucksack

c_j Value of item $j \in J$

Variables:

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

Objective function and constraints:

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

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

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

4.5.1 LP Relaxation

i. For every item $j \in J$ determine the utility coefficient $\frac{c_j}{a_j}$

ii. Sort all items in a decreasing order with regard to the utility coefficient, such that the new order $J = \{1, \ldots, n\}$ becomes

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

iii. Determine $k = \max\{k' : \sum_{j=1}^{k'} a_j \leq b\}$

iv. The optimal solution is given by

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

4.5.2 Heuristic Search

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

4.5.3 Definition of the Sub-Problems

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

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

- Sub-problem $\prod^r$ given by

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

Strategy for choosing a node: Depth-First-Search strategy - next node to process is the one arising from *rounding up* the fractional variable

5 Continuous Optimisation

5.1 Gradient Descent

Gradient descent is an algorithm to find a local minimum of an (unconstrained multidimensional) function f. The idea behind GD is that the function increases in the direction of the gradient, so by going in the opposite direction the function values decrease until a minimum is reached.

Algorithm

- *Initialisation*: Choose starting point x^0
- *Iteration step* $x^i \to x^{i+1}$: Determine local gradient $\nabla f(x^i)$ and move by some amount β in opposite direction $-\nabla$ to get a new point x^{i+1}
  Repeat until gradient value is zero or goes below a target threshold

5.1.1 Step Size

There are different methods to find the optimal step size β to balance speed of convergence and risk of overshooting.

Rule 1: Successive Halving of the Step Size

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

Set $b := 1$ and check whether

$$f(x^i - \beta \nabla f(x^i)) < f(x^i) \tag{1}$$

If (1) is not satisfied, set $\beta := \frac{\beta}{2}$ and check again. Iterate until (1) is satisfied.

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

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 $P(t) = a \cdot t^2 + b \cdot t + c$ such that

$$\begin{aligned} P(0) &= f(x^i) \\ P(\beta) &= f\left(x^i - \beta \nabla f(x^i)\right) \\ P(2\beta) &= f\left(x^i - 2\beta \nabla f(x^i)\right) \end{aligned}$$

$P(t)$ attains its minimum in the interval $[0, 2\beta]$ at $\beta^* = -\frac{b}{2a}$. Choose either β or β^* as the final value of the step size β, whichever gives the lower function value for x^{i+1}.

$$\begin{aligned} a &= \frac{1}{2\beta^2}\left[f(x^i) - 2f\left(x^i - \beta \nabla f(x^i)\right) + f\left(x^i - 2\beta \nabla f(x^i)\right)\right] \\ b &= \frac{1}{2\beta}\left[-3f(x^i) + 4f\left(x^i - \beta \nabla f(x^i)\right) - f\left(x^i - 2\beta \nabla f(x^i)\right)\right] \\ c &= f(x^i) \\ \beta^* &= \frac{-b}{2a} = \frac{\beta}{2} \cdot \frac{3f(x^i) - 4f\left(x^i - \beta \nabla f(x^i)\right) + f\left(x^i - 2\beta \nabla f(x^i)\right)}{f(x^i) - 2f\left(x^i - \beta \nabla f(x^i)\right) + f\left(x^i - 2\beta \nabla f(x^i)\right)} \end{aligned}$$

5.2 Newton's Method

Newton's method finds the zero of a function through iterative approximation. The method works through laying the tangent t through point $(x^i, f(x^i)$ which has the slope $f'(x_i)$

$$t(x) = f'(x^i)(x - x^i) + f(x^i)$$

Finding the zero of the tangent yields

$$\begin{aligned} t(x) &= 0 \\ f'(x^i)(x - x^i) + f(x^i) &= 0 \\ x - x^i &= -\frac{f(x^i)}{f'(x^i)} \\ x &= x^i - \frac{f(x^i)}{f'(x^i)} \end{aligned}$$

Therefore the new iteration point is $x^{i+1} = x^i - \frac{f(x^i)}{f'(x^i)}$.

Newton's method applied to the function $f : \mathbb{R} \to \mathbb{R}$ approximates the zeros of the function. Applied to the derivative f'

$$x^{i+1} = x^i - \frac{f'(x^i)}{f''(x^i)}$$

approximates the zeros of the derivative, that is the stationary points of the function f.

For a multi-dimensional function $f : \mathbb{R}^n \to \mathbb{R}$ the Newton method is

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

$$\nabla f(x^i) = \begin{pmatrix} \frac{\partial f(x^i)}{\partial x_1} \\ \frac{\partial f(x^i)}{\partial x_2} \vdots \\ \frac{\partial f(x^i)}{\partial x_n} \end{pmatrix}$$

$$H_f(x^i) = \begin{pmatrix} \frac{\partial^2 f(x^i)}{\partial x_1 \partial x_1} & \frac{\partial^2 f(x^i)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x^i)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x^i)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x^i)}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f(x^i)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x^i)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x^i)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x^i)}{\partial x_n \partial x_n} \end{pmatrix}$$

5.3 Speed of Convergence

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

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

Gradient Descent achieves linear convergence

Superlinear convergence There is a sequence $\{c_i\} i \in \mathbb{N}$ with $\lim_{n\to\infty} c_i = 0$ such that

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

Newton's Method achieves superlinear convergence

Quadratic convergence There is $c > 0$ and $i_0 \in \mathbb{N}$ such that $\forall i \geq i_0$ we have

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

Broyden's Method achieves quadratic convergence

5.4 Approximating Partial Derivatives

Computing the partial derivatives of f exactly may be impossible or computationally too expensive. In these cases working with approximations may be preferably.

Approximation for First Order Partial Derivative

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{1}{2\epsilon} \left(f(x_1, x_2, \ldots, x_i + \epsilon, \ldots, x_n) - f(x_1, x_2, \ldots, x_i - \epsilon, \ldots, x_n)\right)$$

Approximation for Second Order Partial Derivatives

$$\frac{\partial^2 f(x)}{\partial x_i^2} \approx \frac{1}{\epsilon^2}(f(x_1, x_2, \ldots, x_i + \epsilon, \ldots, x_n) - 2f(x_1, \ldots, x_n) - f(x_1, x_2, \ldots, x_i - \epsilon, \ldots, x_n))$$

$$\begin{aligned} \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{4\epsilon^2}(&f(x_1, \ldots, x_i + \epsilon, \ldots, x_j + \epsilon, \ldots, x_n) \\ &-f(x_1, \ldots, x_i + \epsilon, \ldots, x_j - \epsilon, \ldots, x_n) \\ &-f(x_1, \ldots, x_i - \epsilon, \ldots, x_j + \epsilon, \ldots, x_n) \\ &+f(x_1, \ldots, x_i - \epsilon, \ldots, x_j - \epsilon, \ldots, x_n)) \end{aligned}$$

5.5 Broyden's Method

Computing and inverting the Hessian matrix $H_f(x^i)$ exactly in the approximate version of Newton's method

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

is computationally expensive.

The idea in quasi-Newton methods is to approximate the inverse of the Hessian $H_f(x^i)$ by some matrix $(A^i)^{-1}$ that can be computed more efficiently. The matrix $H_f(x^i)$ is the multidimensional derivative of the gradient at the point x^i . Hence it makes sense to require that the approximation A^i satisfies

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

Broyden's method exploits the fact that this requirement is satisfied exactly for

$$A^i = A^{i-1} + \frac{\left(\left(\nabla f(x^i) - \nabla f(x^{i-1})\right) - A^{i-1}(x^i - x^{i-1})\right)(x^i - x^{i-1})^T}{\|x^i - x^{i-1}\|^2}$$

Define g^i and d^i in order to abbreviate

$$\begin{aligned} A^i &= A^{i-1} + \frac{(\overbrace{(\nabla f(x^i) - \nabla f(x^{i-1})}^{:=g^i} - A^{i-1})(\overbrace{x^i - x^{i-1}}^{:=d^i}))(\overbrace{x^i - x^{i-1}}^{:=d^i})^T}{\|\underbrace{x^i - x^{i-1}}_{:=d^i}\|^2} \\ &= A^{i-1} + \frac{(g^i - A^{i-1}d^i){d^i}^T}{\|d^i\|^2} \end{aligned}$$

The key insight of Broyden's method is that we do not need to invert A^i explicitly in each step

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

Instead $(A^i)^{-1}$ can be computed by updating $(A^{i-1})^{-1}$ according to the **Sherman-Morrison formula**.

> **Theorem 5** *Let A be a regular matrix and u, v two vectors. If $u^T A^{-1} \neq -1$ then $A + vu^T$ is regular as well and the following holds true:*
>
> $$\left(A + vu^T\right)^{-1} = A^{-1} - \frac{A^{-1}vu^TA^{-1}}{1 + u^TA^{-1}v}$$

Using this formula, $(A^i)^{-1}$ can be computed directly from $(A^{i-1})^{-1}$

$$\begin{aligned} A^i = \left(A^{i-1} + \frac{(g^i - A^{i-1}d^i){d^i}^T}{\|d^i\|^2}\right)^{-1} &= (A + vu^T)^{-1} \\ &\overset{\text{SM}}{=} A^{-1} - \frac{A^{-1}1vu^TA^{-1}}{1 + u^TA^{-1}v} \\ &= (A^{i-1})^{-1} - \frac{\left((A^{i-1})^{-1}g^i - d^i\right)(d^i)^T(A^{i-1})^{-1}}{(d^i)^T(A^{i-1})^{-1}g^i} \end{aligned}$$

5.5.1 Algorithm

1. **Initialisation**: Start with x^0. Compute $\nabla f(x^0)$, the inverse of the Hessian Matrix $H_f(x^0)$ and set $(A^0)^{-1} = \left(H_f(x^0)\right)^{-1}$. Use this to compute

$$x^1 = x^0 - (A^0)^{-1}\nabla f(x^0)$$

2. **Iteration step**: To compute x^{i+1} from x^i, compute $\nabla f(x^i)$, $g^i = \nabla f(x^i) - \nabla f(x^{i-1})$ and $d^i = x^i - x^{i-1}$, then

$$(A^i)^{-1} = (A^{i-1})^{-1} - \frac{\left((A^{i-1})^{-1}g^i - d^i\right)(d^i)^T(A^{i-1})^{-1}}{(d^i)^T(A^{i-1})^{-1}g^i}$$

and set

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

After the first step, A^i or $H_f(x^i)$ is never dealt with explicitly, only $(A^i)^{-1}$ is maintained and updated. This saves significant computer cost, especially for an n-dimensional problem, as inverting A^i or $H_f(x^i)$ explicitly would take $O(n^3)$, while computing the inverse with Sherman-Morrison updates and multiplying by the gradient $\nabla f(x^i)$ can all be done in $O(n^2)$.

5.6 Aitken's Acceleration Method

Aitken's Δ^2 method is not a new method for finding local extrema, but can be used to improve the convergence speed of other existing methods that would converge slowly otherwise.

Let $\{x^i\}_{i\in\mathbb{N}}$ be a scalar sequence that converges linearly towards its limit x^*. Construct a new sequence $\{y^i\}_{i\in\mathbb{N}}$ from $\{x^i\}_{i\in\mathbb{N}}$ as follows

$$\begin{aligned} y^i &= x^i - \frac{(x^i - x^{i-1})^2}{x^i - 2x^{i-1} + x^{i-2}} \\ &= x^i - \frac{(\Delta x^i)^2}{\Delta^2 x^i} \end{aligned}$$

where $\Delta^2 x^i = \Delta(\Delta x^i) = \Delta(x^i - x^{i-1})\Delta x^i - \Delta x^{i-1}$. Then $\{x^i\}_{i\in\mathbb{N},i\geq 2}$ converges quadratically towards the same limit x^*.

6 Graph and Network Optimisation

Theorem 6 *A **graph** $G = (V, E)$ consists of finite V of vertices and a set $E \subseteq V \times V$ of edges. An edge $e \in E$ is of the form $e = (u, v)$ with $u, v \in V$.*

*A **weight function** on $G = (V, E)$ is a function assigning a real number to each edge $e \in E$*

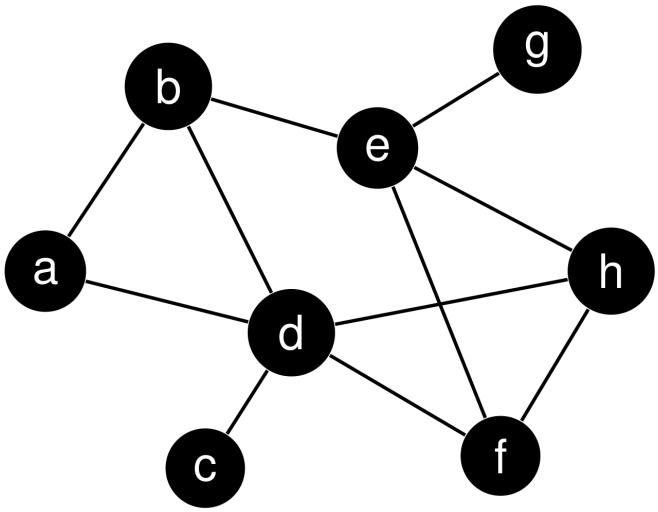
Theorem 7 *The **adjacency matrix** of a graph G with $|N| = n$ is the $n \times n$-matrix with entries*

$$a_{ij} = \begin{cases} 1 & \text{if } (u_i, u_j) \in E \\ 0 & \text{if } (u_i, u_j) \notin E \end{cases}$$

For an undirected graph, the adjacency matrix is symmetric. For a weighted graph, the adjacency matrix specifies the edge weights.

The **adjacency list** representation of a graph consists of a doubly linked list for each vertex, specifying all of its neighbours.

6.1 Depth-First Search



1. Start at a and put it on the stack
2. If there is an unmarked neighbour, go there and put it on the stack
3. If there is no unmarked neighbour, backtrack and go to step 2.

Stack: a, b, d, ~~e~~, f, e, ~~g~~, h

6.2 Breadth-First Search

1. Start at a and put it in queue
2. Output first vertex from queue. Mark all neighbours and put them in queue. Do so until queue is empty.

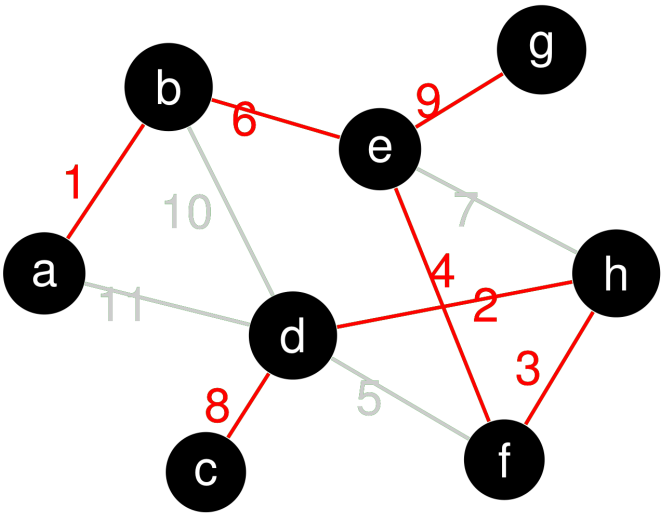
Queue: a, b, d, e, c, f, h, g

6.3 Spanning Tree

Given a graph $G(V, E)$ with positive edge weights find a set of edges that connects all vertices of G with minimum weight.

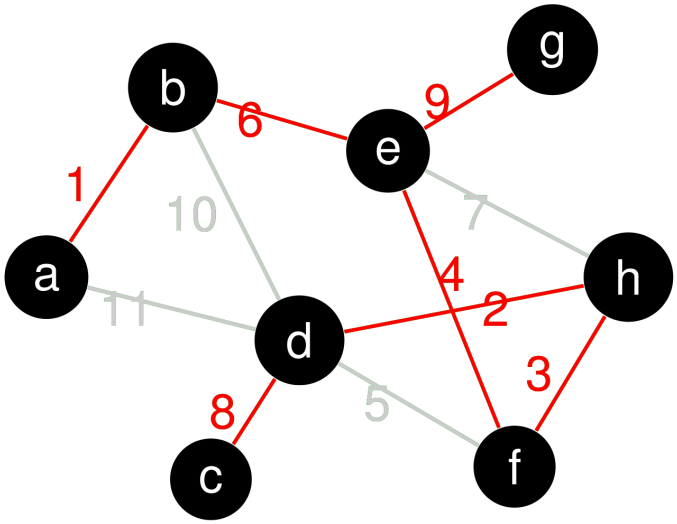
6.3.1 Optimistic Approach

Successively build the cheapest connection available that is not redundant.



6.3.2 Pessimistic Approach

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



6.3.3 Prim's Algorithm

Choose an arbitrary start vertex v_0 and set $M = \{v_0\}$. Iteratively add the cheapest to reach vertex in $V \setminus M$ to the current set M. Select the corresponding edge. Continue until $M = V$.

The edges selected in this way form a minimum-weight spanning tree.

6.3.4 Kruskal's Algorithm

Set $M = \{\}$. Iteratively add the cheapest edge in $E \setminus M$ that connects two vertices that are not yet connected by edges in M. Continue until all vertices are connected by edges in M.

The edges in M form a minimum-weight spanning tree. Kruskal's Algorithm is equivalent to the optimistic approach.

Prim's and Kruskal's Algorithms are both greedy algorithms.

6.4 Shortest Paths

Given a graph $G(V, E)$, two vertices $u, v \in V$ with positive weights, interpreted as distances.

Dijkstra's Algorithm can be used to find a shortest path in G connecting u to v or a shortest path from a start vertex v_0 to *all* other vertices. And the **Floyd-Warshall Algorithm** can be used as an efficient solution to find the shortest path between *all pairs* of vertices.

6.4.1 Dijkstra's Algorithm

1. **Initialisation**: Set $V_0 = \{v_0\}$, $E_0 = \{\}$ and $I(v_0) = 0$
2. Do the following $n - 1$ times:
   $V_i = \{v_0, \ldots, v_i\}$ and $E_i = \{e_0, \ldots, e_i\}$ are the sets of vertices and edges already visited.
   For each edge $e = (u, v)$ with $u \in V_i$ and $v \in V \setminus V_i$ compute $l(u) + \text{weight}(e)$. Choose the edge minimising this as $e_{i+1} = (u_{i+1}, v_{i+1})$.
   Set $V_{i+1} = V_i \cup \{v_{i+1}\}, E_{i+1} = E_i \cup \{e_{i+1}\}$ and $l(v_{i+1}) = l(u_{u+1}) + \text{weight}(e_{i+1})$.

6.4.2 Floy-Warshall Algorithm

Let $V = \{v_1, \ldots, v_n\}$ and let $\text{minPath}(i, j, k)$ denote the length of a shortest path from v_i to v_j that only uses vertices from $\{v_1, \ldots, v_k\}$

Notes : In the end there should be a $\text{minPath}(i, j, n)$ for all vertices v_i and v_j. $\text{minPath}(i, j, 0)$ is the weight of the edge (v_i, v_j) or ∞ if the edge does not exist. Suppose for all i, j the $\text{minPath}(i, j, k)$ has been computed, then

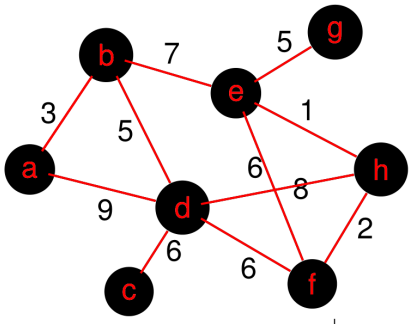
$$\text{minPath}(i, j, k) = \min \begin{cases} \text{minPath}(i, j, k) \\ \text{minPath}(i, k+1, k) + \text{minPath}(k+1, j, k) \end{cases}$$

The minimum is attained for the second term if it pays off to go via v_{k+1}.

Given $G = (V, E)$ with weight function $\omega : E \to \mathbb{R}$. Find the shortest distances between all pairs of vertices.

1. **Initialisation**:
$$\text{minPath}(i, j, 0) = \begin{cases} \omega(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ \infty & \text{else} \end{cases}$$
2. For $k = 1, 2, \ldots, n$ compute:
   $\text{minPath}(i, j, k)$ for all i and j according to iteration formula

The values $\text{minPath}(i, j, n)$ are the shortest distances from v_i to v_j.

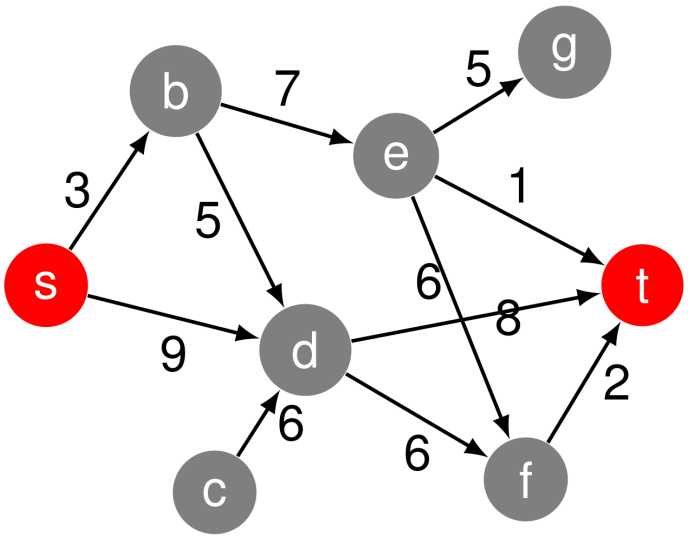


$minPath(i, j, 8)$

	a	b	c	d	e	f	g	h
a	-	3	14	8	10	13	15	11
b	3	-	11	5	7	10	12	8
c	14	11	-	6	15	12	20	14
d	8	5	6	-	9	6	14	8
e	10	7	15	9	-	3	5	1
f	13	10	12	6	3	-	8	2
g	15	12	20	14	5	8	-	6
h	11	8	14	8	1	2	6	-

6.5 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** or **sink**. What is the maximal flow which can be routed through $G(V, E)$ from s to t, respecting the capacity limits given by ω?



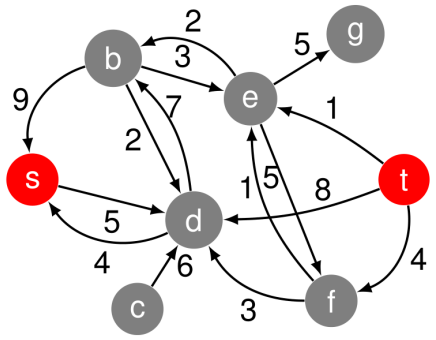
Theorem 8 *A graph $G(V, E)$ is **bipartite** if its vertex set can be split in two parts A and B such that all edges have one vertex in A and one in B.*

6.6 Ford-Fulkerson Algorithm

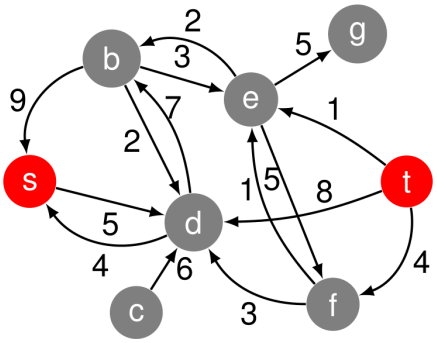
Given $G(V, E)$ with positive capacities $\omega : E \to \mathbb{R}$, source s and sink t, find the maximum flow $F_{s \to t}$ from s to t through G.

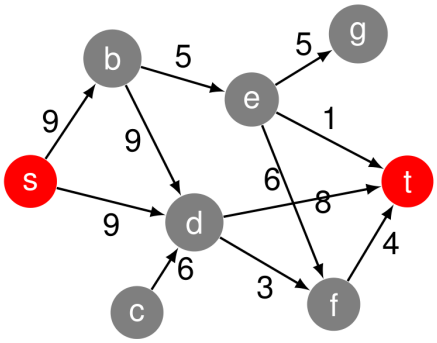
1. **Initialisation**: Set the total flow $F_{s \to t} =$, residual graph $G_R = G$ and edge flows $f(e) = 0 \ \forall e \in E$

2. While there is a path P from s to t in G_R
   - Determine smallest capacity g along the path P
   - Reduce capacities along P by g
   - Increase capacities of backwards edges by g
   - Delete edges with capacity 0 in G_R
   - Increase $F_{s\to t}$ by g
   - Increase flows $f(e)$ along P by g
3. Output $F_{s\to t}$, $f(e)$, $e \in E$



$1 \quad s \to b \to e \to t$
$8 \quad s \to b \to d \to t$
$3 \quad s \to d \to f \to t$
$1 \quad s \to d \to b \to e \to f \to t$
13



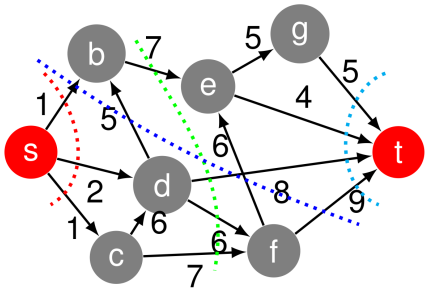


The Ford-Fulkerson algorithm has inefficient behaviour, that if the augmenting paths are chosen arbitrarily, the algorithm might take many iterations to find the maximum flow.

Edmonds-Karp Algorithm In each iteration, use a ***shortest*** augmenting path, that is a path from s to t with the fewest number of edges. Can be found with BFS in time $O(|E|)$.

6.7 Max-Flow Min-Cut Theorem

Theorem 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. The maximum* st *flow equals the minimum value of all possible* st*-cuts.*



$S = \{s\}$, $T = \{b, c, d, e, f, g, t\}$, value = 4

$S = \{s, b, c, d, e, f, g\}$, $T = \{t\}$, value = 26

$S = \{s, b, d, c\}$, $T = \{e, f, g, t\}$, value = 28

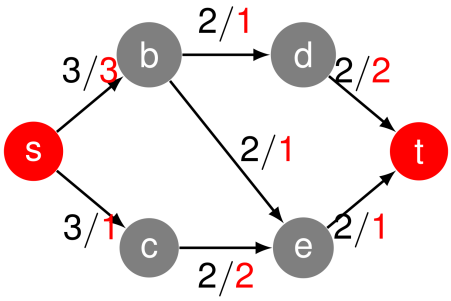
$S = \{s, c, d, f\}$, $T = \{b, e, g, t\}$, value = 29

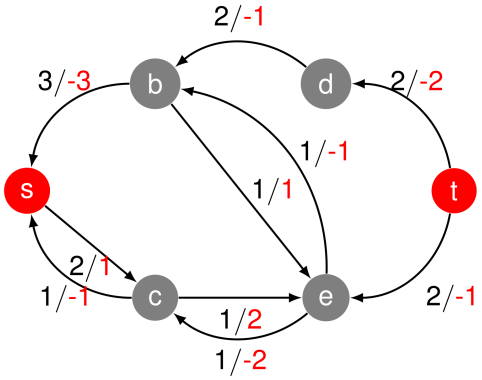
$S = \{s, b\}$, $T = \{c, d, e, f, g, t\}$, value = 10

$S = \{s, e, g\}$, $T = \{b, c, d, f, t\}$, value = 13

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**.

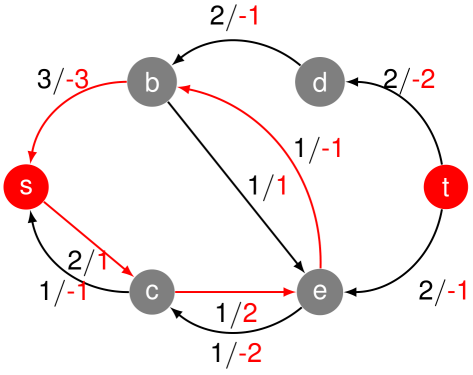
Each edge has not only a capacity ω, but also a **cost per flow unit** going through that edge. The min-cost max-flow algorithm finds a maximum flow with minimum cost.





capacity	path	cost
2x	$s \to b \to d \to t$	$2 * 6 = 12$
1x	$s \to b \to e \to t$	$1 * 5 = 5$
1x	$s \to c \to e \to t$	$1 * 4 = 4$
4		21

Until now the Ford-Fulkerson algorithm was used while annotating the costs. Now, 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.



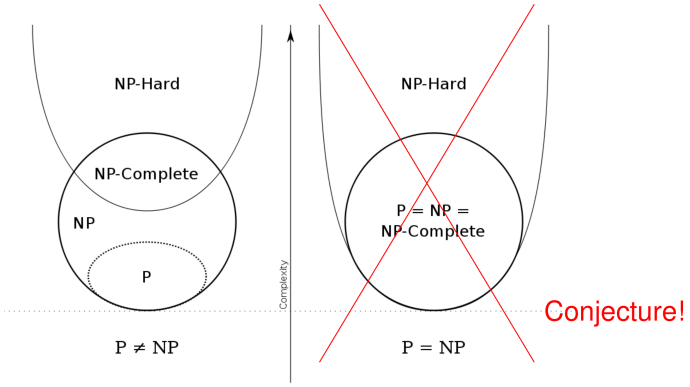
capacity	path	cost
2x	$s \to b \to d \to t$	$2 * 6 = 12$
1x	$s \to b \to e \to t$	$1 * 5 = 5$
1x	$s \to c \to e \to t$	$1 * 4 = 4$
4		21
1x	$s \to c \to e \to b \to s$	$1 * (-1) = -1$

7 Heuristic Optimisation

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.

This gives the advantage of possibly finding a good solution in a reasonable time, with the drawback that no reliable optimality guarantees can be given.



Many algorithmic problems are solvable in principle, but no efficient algorithm is known. The classification happens by complexity with regards to growth behaviour with more inputs.

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

A **metaheuristic** is a general heuristic problem-solving strategy that can be used in optimisation problems. They can be employed both in discrete and continuous problems and both when looking for local and for global optima.

Metaheuristics can sometimes be combined with more rigorous local methods to search for global optima in continuous optimisation problems (hybrid methods).

7.0.1 Trajectory-based Metaheuristics

Start with a (random) "solution" and improve it continuously by exploring its neighbourhood and improving the tour. Thus a trajectory through the solution space is obtained, similar to what is done in Gradient Descent.

Often it is not obvious how to **define and explore the solution space** or how to **define a good neighbourhood** for a given problem.

7.1 Hill Climbing Algorithm

1. **Initialisation**: Start with random solution. Evaluate initial solution
2. **Iteration**: Explore neighbourhood and evaluate new solution candidates. Continue with best solution found, if this improves the target function, otherwise stop as this is a local optimum
3. Return solution

The implementation of this algorithm is trivial but the size of the neighbourhood or allowed modification steps poses a difficulty. In particular the size of the neighbourhood can be problematic, if chosen too small the algorithm might get stuck quickly without exploring much of the solution space, while a too large size takes a lot of computation time.

As Hill Climbing is a ***local search*** algorithm it is usually a good idea to have multiple runs with different starting points and take the best solution out of all runs.

7.1.1 Stochastic Hill Climbing

1. **Initialization**: Start with random solution. Evaluate initial solution
2. **Iteration**: Choose random solution in neighbourhood, that is perform a random modification. Evaluate new solution candidate, and accept it, if it is better. Continue until termination criterion is reached.
3. Return solution

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.

7.1.2 Continuous Hill Climbing

Find $\vec{x}$ minimising $f(\vec{x})$ for a given target function $f : \mathbb{R} \to \mathbb{R}$

1. **Initialisation** $i = 0$: 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

This idea is similar to gradient descent, but since this does not rely on derivatives the idea also works in discrete settings.

7.2 Tabu Search

The idea behind Tabu search is similar to those in Hill climbing, but the algorithm has a **memory** and tries to avoid steps that go back to previously evaluated solutions or those that undo the effect of previous steps.

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

1. **Initialisation**: Start with random solution. Evaluate initial solution
2. **Iteration**: Explore neighbourhood and evaluate new solution candidates. Only consider steps that are not tabu. Proceed with this step, saving the best solution found and updating the tabu list until the termination criterion is reached.
3. Return solution

The simplest variant of Tabu search is to only store the last solution as a tabu, it would probably be better to keep the last k moves. The goal of the tabus is to help exploring the solution space, that is fostering diversity of the considered solutions.

Further considerations can be made in terms of neighbourhood size, computational requirements, and if evaluations should be approximated or if approximation by updates is possible. Restarting the algorithm should be done in areas not yet explored and exploit the structure of solutions encountered, to learn features ruling out duds or focussing on promising candidates.

It would be beneficial to not flat out disqualify invalid solutions, meaning solutions that violate constraints, but **penalise** them. **Strategic oscillation**: Dynamically adjust size of penalty to reach limits of feasiblity. If there are many invalid solutions encountered, the penalty should be increased, if few invalid solutions are encountered, the penalty should be decreased to encourage diversity.

Variants

- Fixed penalty for invalid solutions
- Variable penalty which is adapting to the degree of violation

7.2.1 Stochastic or Randomised Tabu Search

1. **Initialisation**: Start with random solution. Evaluate initial solution
2. **Iteration**: Choose a random solution from the neighbourhood. Evaluate the new solution candidate, if its not in the Tabu list. If a better solution is found, continue with this solution, update the Tabu list and repeat this step. Continue until termination criterion is reached.
3. Return solution

7.3 Simulated Annealing

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

1. **Initialisation**: Start with random solution. Evaluate initial solution
2. **Iteration**: Choose a random solution from the neighbourhood and evaluate it. If it is better, accept it. If it is worse, accept it only with some probability. Repeat until termination criterion is reached.
3. Return solution

And with a hard threshold T

1. **Initialisation**: Start with x_0
2. **Iteration**: $x_i \to x_{i+1}$. Sample random solution y_i in the neighbourhood and accept it to be x_{i+1} with probability $\min\left\{1, e^{\frac{f(x_i)+f(y_i)}{T_i}}\right\}$

- If $f(y_i) \leq f(x_i) \to e^{\frac{f(x_i)+f(y_i)}{T_i}} \geq 1$ and y_i is accepted
- If T is very large, every solution y_i will be accepted

$$\min\left\{1, e^{\frac{f(x_i)+f(y_i)}{T_i}}\right\} = \min\left\{1, e^{\frac{f(x_i)+f(y_i)}{\infty}}\right\} = \min\left\{1, e^{\pm 0}\right\} = 1$$

- If $T \to 0$, only solutions better than y_i will be accepted

$$\min\left\{1, e^{\frac{f(x_i)+f(y_i)}{T_i}}\right\} = \min\left\{1, e^{\frac{f(x_i)+f(y_i)}{+0}}\right\} = \min\left\{1, e^{\pm\infty}\right\} = 0 \text{ or } 1$$

7.4 Population Based Methods

The key idea behind the following methods is 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.

Biological principles serve as an inspiration:

- Finite lifespan of individuals and 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 by mutations
- Population acquires and cultivates shared knowledge
- Local development of populations

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

Theorem 10 *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****.*

- **Individuals**: Solution candidates or partial solutions
- **Evolutionary process**: constant-size population without survivors from one generation to the next
- **Neighbourhood structure**: Which individuals can exchange information
- **Information sources**: Information gets transmitted between generations
- **Restriction-violating solution candidates**: Discard, repair or penalize?
- **Intensification strategy**: Exploiting neighbourhood by subsequent adjustment of solution candidates
- **Diversification strategy**: Exploring the entire solution space through mutation of the solutions

7.5 Genetic Algorithm

7.5.1 Terminology

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 optimisation problem.

The **genotype** is the encoded form of an individual. While the **phenotype** is the decoded form of an individual. It does not depend on the choice of encoding. The **fitness function** is a measure for the quality of a solution candidate in the optimisation problem.

The individual entries in the vector representation of an individual are its **genes**. They describe the genetic information and the properties of the individuals. The concrete value a gene can take in an individual are called **alleles**.

The fitness function is the measure for the quality of a solution candidate in the optimisation problem.

7.5.2 Algorithm

1. **Initialisation** Random starting population
2. **Iteration** Create next generation according to evolutionary principles
   - Assign fitness to individuals
   - Natural selection and choosing predecessors for reproduction
   - Recombination process
   - Mutation process

   Repeat until termination criterion is satisfied
3. Return best individual

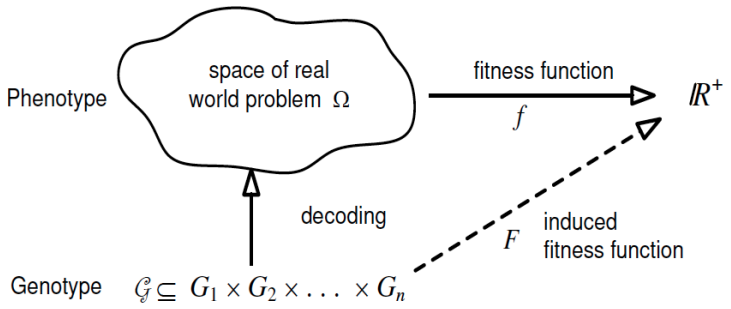


Figure 7.1: The first and most important step is to find a suitable encoding

7.5.3 Encoding of an Optimisation Problem

Desirable would be that a small change in genotype results in a small change in the phenotype. A very simple encoding is often given by a **standard binary encoding**, however as this has issues with the Hamming distance **Gray code** is preferable. In continuous optimisation problems **real-valued encodings** are sometimes more useful.

decimal	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 $b = b_n \ldots b_1 \mapsto$ Gray code $g = g_n \ldots g_1$

$$g = b \oplus (b \gg 1)$$

7.5.4 Replacement Schemes

- **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

7.5.5 Selection of Predecessors

Tournament Selection

Sample k individuals randomly from the population

Choose best individual with probability p

Choose second best individual with probability $(1-p)p$

Choose third best individual with probability $(1-p)^2 p$

$\vdots$

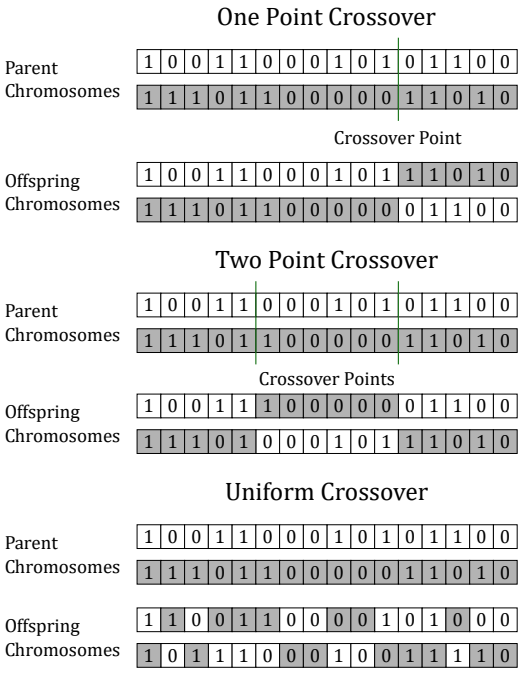
Choose $(k-1)$-best individual with probability $(1-p)^{(k-2)}p$

Choose k-best individual with probability $(1-p)^{(k-1)}$

The larger k, the higher the selection pressure. The advantage of this method is that it is based on ranking and that the selection pressure can be adjusted.

7.5.6 Recombination

With a **One-point Crossover** a point is chosen randomly and designated crossover point. The bits are swapped between both parents to generate two offspring. **Two Point Crossover** follows the same mechanism, but with more crossover points. This method can be generalised for an arbitrary k crossover points. With **Uniform Crossover** each bit is chosen from either parent with equal probability. Other mixing ratios are sometimes used, resulting in offspring which inherit more genetic information from one parent than the other.



7.5.7 Recombination for Permutation - PMX Operator

PMX Crossover is a genetic algorithm operator and is a two-point crossover with repair mechanism. For some problems it offers better performance than most other crossover techniques. Basically, parent 1 donates a swath of genetic material and the corresponding swath from the other parent is sprinkled about in the child. Once that is done, the remaining alleles are copied direct from parent 2.

1. Randomly select a swath of alleles from parent 1 and copy them directly to the child. Note the indexes of the segment.
2. Looking in the same segment positions in parent 2, select each value that hasn't already been copied to the child.
    A. For each of these values:
        i. Note the index of this value in Parent 2. Locate the value, V, from parent 1 in this same position.
        ii. Locate this same value in parent 2.
        iii. If the index of this value in Parent 2 is part of the original swath, go to step i. using this value.
        iv. If the position isn't part of the original swath, insert Step A's value into the child in this position.
3. Copy any remaining positions from parent 2 to the child.

7.5.8 PMX Operator Example

```
Parent 1: 8 4 7 3 6 2 5 1 9 0
Parent 2: 0 1 2 3 4 5 6 7 8 9
Child 1:  _ _ _ 3 6 2 5 1 _ _
```

1. We copy a random swath of consecutive alleles from Parent 1 to the Child.

```
Parent 1: 8 4 7 3 6 2 5 1 9 0
Parent 2: 0 1 2 3 4 5 6 7 8 9
Child 1:  _ _ _ 3 6 2 5 1 _ _
```

2. '4' is the first value in the swath of Parent 2 that isn't in the child. We identify 6 as the value in the same position in Parent 1. We locate the value 6 in Parent 2 and notice that it is still in the swath. So, we go back to step 'i' using 6 as the value.

```
Parent 1: 8 4 7 3 6 2 5 1 9 0
Parent 2: 0 1 2 3 4 5 6 7 8 9

Child 1:  _ _ _ 3 6 2 5 1 _ _
```

3. Repeating Step i: Once again, we see that 5 is in the same position in Parent 1, and we locate 5 in Parent 2. It also is in the swath, so we repeat step 'i' once more with '5' as our value.

```
Parent 1: 8 4 7 3 6 2 5 1 9 0
Parent 2: 0 1 2 3 4 5 6 7 8 9
Child 1:  _ _ 4 3 6 2 5 1 _ _
```

4. Repeating Step i: We see that 2 is in the same position in Parent 1, and we locate 2 in Parent 2 in the 3rd position. Finally, we have obtained a position in the Child for the value 4 from Step 2.

```
Parent 1: 8 4 7 3 6 2 5 1 9 0
Parent 2: 0 1 2 3 4 5 6 7 8 9
Child 1:  _ 7 4 3 6 2 5 1 _ _
```

5. '7' is the next value in the swath in Parent 2 that isn't already included in the Child. So, we check the same index in Parent 1 and see a '1' in that position. Next, we check for '1' in Parent 2 and find it in the 2nd position. Since the 2nd position is not part of the swath, we've found a home for the value '7'.

```
Parent 1: 8 4 7 3 6 2 5 1 9 0
Parent 2: 0 1 2 3 4 5 6 7 8 9
Child 1:  0 7 4 3 6 2 5 1 8 9
```

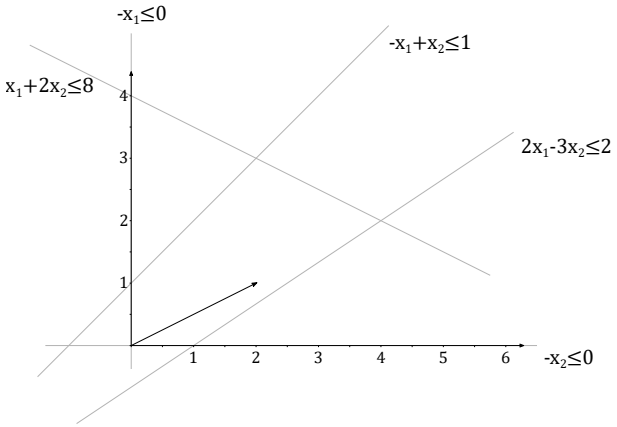
6. Now the easy part, we've taken care of all swath values, so everything else from Parent 2 drops down to the child.

If we wish to create a 2nd child with the same set of parents, simply swap the parents and start over.

A Examples

A.1 Simplex Algorithm

$$
\begin{aligned}
&\max(2x_1 + x_2)\\
-x_1 + x_2 &\leq 1\\
x_1 + 2x_2 &\leq 8\\
2x_1 - 3x_2 &\leq 2\\
-x_1 &\leq 0\\
-x_2 &\leq 0
\end{aligned}
$$



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

   1) $B = (1,2)$

   2) $A_B = \begin{pmatrix} -1 & 1 \\ 1 & 2 \end{pmatrix} \quad b_B = \begin{pmatrix} 1 \\ 8 \end{pmatrix}$

   3) ${A_B}^{-1} = \frac{1}{-1 \cdot 2 - 1 \cdot 1} \begin{pmatrix} 2 & -1 \\ -1 & -1 \end{pmatrix} = -\frac{1}{3} \begin{pmatrix} 2 & -1 \\ -1 & -1 \end{pmatrix} = \begin{matrix} \\ \\ -d \end{matrix} \begin{pmatrix} -\frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} \end{pmatrix} = \bar{A}$

   4) $u = c^T \bar{A} = (2,1) \cdot \begin{pmatrix} -\frac{2}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} \end{pmatrix} = (-1,1) \not\geq \vec{0}$

   5) $Av = \begin{pmatrix} 1 \\ 8 \\ -5 \\ -2 \\ -3 \end{pmatrix} \quad d = \begin{pmatrix} -\frac{2}{3} \\ \frac{1}{3} \end{pmatrix} Ad = \begin{pmatrix} -1 \\ 0 \\ \frac{7}{3} \\ -\frac{2}{3} \\ \frac{1}{3} \end{pmatrix} \quad \lambda = \begin{pmatrix} 0 \\ 0 \\ 3 \\ -3 \\ 9 \end{pmatrix} \rightarrow \lambda^* = 3$

   6) $v' = v + \lambda^* \cdot d = \begin{pmatrix} 2 \\ 3 \end{pmatrix} + 3 \cdot \begin{pmatrix} -\frac{2}{3} \\ \frac{1}{3} \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$

2. $v = (4,2)$

   1) $B = (2,3)$

   2) $A_B = \begin{pmatrix} 1 & 2 \\ 2 & -3 \end{pmatrix} \quad b_B = \begin{pmatrix} 8 \\ 2 \end{pmatrix}$

   3) ${A_B}^{-1} = \frac{1}{1 \cdot -3 - 2 \cdot 2} \begin{pmatrix} -3 & -2 \\ -2 & 1 \end{pmatrix} = \begin{pmatrix} \frac{3}{7} & \frac{2}{7} \\ \frac{2}{7} & -\frac{1}{7} \end{pmatrix}$

   4) $u = (2,1) \begin{pmatrix} \frac{3}{7} & \frac{2}{7} \\ \frac{2}{7} & -\frac{1}{7} \end{pmatrix} = (\frac{8}{7}, \frac{3}{7}) \geq \vec{0}$ Vertex v is optimal