Algorithmics

Spring Semester 2020

Prof. Dr. Matthias Krause

2020/05/25, 17:50

University of Mannheim

Prerequisites

Classification into the Overall Context of Business Informatics

- Process Management in Business and Society: Identifying problems to be solved for improving the overall system.
- **Formulating these problems in a formal manner as Computational problems**
- **Determine the Complexity of these problems**
  - Do we know efficient algorithms or do we have to handle computationally hard problems?
  - If the problem is hard, do we know efficient heuristics?
- **Make a decision concerning the solution algorithms**
- Solve the Problem by implementing the algorithms

What should you learn in this Course?

- Modelling informally specified problems as **formal computational problem**
  - Determine appropriate data structures for inputs and outputs (solutions)
  - Define the computational problem as input-output relation
- A list of basic computational problems, especially network optimization problems, which occur in many practical situations
- A selection of important efficient algorithms for some of these problems
- Techniques for showing that certain problems are hard in the sense that efficient algorithms do not exist for them

Prerequisites and Literature for Algorithmics

Prerequisites

- Programming
- Algorithms and Data Structures
- Probability Theory and Statistics
- Linear Algebra
- Calculus

Literature

- Introduction to Algorithms (Cormen Leiserson Rivest Stein) third edition, MIT Press 2009
- Handbook in Operations Research and Management Science, Vol. 7 ”Network Models”, edited by Ball, Magnanti, Monma, Nemhauser
- ...

Introduction

Computational Problems

Computational problems (for short: Problems) Π are relations $\Pi \subseteq X \times Y$, X set of valid inputs, Y a set of valid outputs, $(x, y) \in \Pi$ means: y is solution of x w.r.t. Π.

Examples:

- **Sorting:** Inputs are sequences $\vec{a} = (a_1, \cdots, a_n)$ of elements from an ordered set $(M, \leq)$, outputs are permutation $\pi \in \mathcal{S}_n$, π solution for $\vec{a}$ if $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.
- **Connectivity:** Input $G = (V, E)$ undirected Graph, outputs are 0 (false, G is not connected) or 1 (true, G is connected).

Inputs and Input length

Inputs $x \in X$ are associated with a parameter $|x| \in \mathbb{N}$, the input length. This yields a partition

$$X = \bigcup_{n \in \mathbb{N}} X_n,$$

$X_n = \{x \in X, |x| = n\}$ set of inputs of length n.

Examples:

- Inputs $x \in \mathbb{N}$, $|x| = \lfloor \log_2(x) \rfloor + 1$ bit length of x,
- Inputs $m \times n$ matrices M over $\{0, 1\}$, $|M| = m \cdot n$
- Inputs undirected graphs $G = (V, E)$, $|G| = |V|$ or $|G| = |V| + |E|$ oder $|G| = |E|$ (context dependent).

Algorithms

- We consider algorithms for sequential **computational devices**.
- Computational devices work clockwise over a given set of elementary operations including a STOP command.
- They can read and store data, execute elementary operations on stored data (ideally one operation per clock cycle), and can output data.
- Algorithms A are instructions for a device to execute a well defined sequence of computational steps in dependence of the stored input data x (one elementary operation per step).
- This sequence is called **computation** of A on x and can be finite or infinite.
- As the result of a finite computation, an output $A(x)$ will be produced.

Solving Problems with Algorithms

An algorithm A solves (or computes) a given problem $\Pi \subseteq X \times Y$, if

- A refers to a well defined rule how inputs $x \in X$ are stored (input data structure).
- A refers to a well defined rule how outputs $y \in Y$ are produced (output behaviour).
- For each input $x \in X$, the computation of A on x is finite and for the output $y = A(x) \in Y$ it holds $(x, y) \in \Pi$.

Cost Measures for Computations

Given an algorithm A, which refers to inputs $x \in X$.

- The time consumption $time_A(x)$ of the computation of A on x equals the sum of the time costs of the computational steps of the computation.
- Assignment of time costs to the computational steps depends from the context (milliseconds, processor clock cycles etc.).
- A usual approach is simplification: The execution of one elementary operation costs one time unit.
- The space consumption $space_A(x)$ equals the number of storage units used during the computation of A on x.

Time Behaviour of Algorithms

Let A be an algorithm processing inputs from $X = \bigcup_{n \in \mathbb{N}} X_n$, $X_n = \{x \in X, |x| = n\}$.

- **Worst Case Running Time** $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \max\{time_A(x), x \in X_n\}.$$

- **Best Case Running Time** $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \min\{time_A(x), x \in X_n\}.$$

- **Average Case Running Time** $time_A : \mathbb{N} \longrightarrow \mathbb{N}$,

$$time_A(n) = \mathbf{E}_{x \in_{P_n} X_n} time_A(x),$$

where P_n probability distribution X_n.

Designing and analyzing algorithms means

- **Design** an algorithm A for a given problem $\Pi \subseteq X \times Y$, $X = \bigcup_{n \in \mathbb{N}} X_n$, $X_n = \{x \in X, |x| = n\}$.
- **Proof of Correctness**: Show that for all $x \in X$ algorithm A stops on x, and that $A(x)$ is solution of x w.r.t. Π (i.e. $(x, A(x)) \in \Pi$).
- **Analysis of the Running Time**: Determine the **asymptotic growth order** of $time_A$, i.e., determine $time_A$ up to multiplicative constants (because this is **platform independent**).

Asymptotic Growth Order of Functions

Let $f, g, h : \mathbb{N} \longrightarrow \mathbb{R}^+$ be monotone increasing functions. We write

Definition 1

- $f(n) = O(g(n))$ (more exactly, $f \in O(g)$), if there is a constant $C \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that $f(n) \leq C \cdot g(n)$ for all $n \geq n_0$, Interpretation: f grows asymptotically not faster than g.
- $f(n) = \Omega(g(n))$, if there is a constant $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{N}$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$. Interpretation: f grows asymptotically not slower than g.
- $f(n) = o(g(n))$, if for all constants $c \in \mathbb{R}^+$ there is $n_0 \in \mathbb{N}$ such that $f(n) < c \cdot g(n)$ for all $n \geq n_0$. Interpretation: f grows asymptotically strictly slower than g.

Asymptotic Growth of Functions II

Definition 2

- $f(n) = \omega(g(n))$, if for all constants $C \in \mathbb{R}^+$ there is $n_0 \in \mathbb{N}$ such that $f(n) > C \cdot g(n)$ for all $n \geq n_0$. Interpretation: f grows asymptotically strictly faster than g.
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, i.e., f and g have the same asymptotic order of growth.

Observation: The asymptotic growth order notation of functions allows to neglect multiplicativ constants and additive low order terms, for example $5n^2 + 3n + 7 = \Theta(n^2)$.

Typical Growth Orders

- $\Theta(n)$ linear growth
- $\Theta(n^2)$ quadratic growth
- $\Theta(n^3)$ cubic growth
- $O(1)$ constant growth
- $\Theta(\log(n))$ logarithmic growth
- $n^{O(1)} = \bigcup_{k \in \mathbb{N}} O(n^k)$ polynomially bounded growth.
- $\exp(\Omega(n)) = 2^{\Omega(n)}$, exponential growth

Facts which one should know

- Higher degree polynomials grow strictly faster, $n^{k+1} = \omega(n^k)$.
- Sublinear is strictly faster than polylogarithmic, $n^c = \omega((\log(n))^k)$ for all $c > 0$.
- Weak exponential is strictly faster than polynomial, $2^{n^c} = \omega(n^k)$ for all $c > 0$ and $k \in \mathbb{N}$.
- Sequential Algorithms for nondegenerate problems have usually running time in $\Omega(n)$, as they have to read the complete input at least.

Efficiently solvable problems

Definition 3

A problem Π is considered to be efficiently solvable (w.r.t. to a given reasonable model of computation), if there is an polynomial time algorithm A for Π (i.e., $time_A = n^{O(1)}$)

Alonzo Church (1903-1995, US-mathematician and pioneer of computer science): *The set of efficiently solvable problems is for all reasonable models of computation the same.*

Definition 4

PTIME denotes the set of all problems having a polynomial time algorithm (in one reasonable model of computation, i.e. in all reasonable models of computation)

Exponential time is not efficient in practice

Consider exhaustive key search in $\{0,1\}^n$ w.r.t. to a cryptographic algorithm of key-length n (Advanced Encryption Standard (AES) has key-length $n = 128$).

Consider a special purpose TerraHertz processor P which tests $10^{12} \approx 2^{40}$ keys in a second.

- A year has $31.566 \cdot 10^3 \approx 2^{25}$ seconds.
- The expected lifetime of the earth is $4 \cdot 10^9 \approx 2^{32}$ years.
- Consequently, P can test $\approx 2^{97}$ keys in the expected lifetime of the earth.

Shortest Path Problems

1. **Single Pair Shortest Path**
   - **Input:** A directed edge-weighted graphs $G = (V, E, w)$, a pair (u, v) of nodes from V
   - **Output:**
     - ∞, if v is not reachable from u,
     - $-\infty$, if there is a walk from u to v containing a negative cycle,
     - a shortest path from u to v, otherwise
2. **Single Source Shortest Path**
   - **Input:** A directed edge-weighted graphs $G = (V, E, w)$, a source $s \in V$.
   - **Output:** The output of Single Pair Shortest Path for all pairs (s, v), $v \in V$
3. **All Pairs Shortest Path**
   - **Input:** A directed edge-weighted graphs $G = (V, E, w)$
   - **Output:** The output of Single Pair Shortest Path for all pairs (u, v), $u, v \in V$

Walks, Paths, Cycles

- **Inputs: Directed weighted graphs** $G = (V, E, w)$ with edge weight function $w : E \longrightarrow \mathbb{R}$.
- **Weight of edge sets** $E' \subseteq E$**:** $w(E') = \sum_{e \in E'} w(e)$.
- **Walks:** A sequence of consecutive edges $p = ((v_0, v_1), (v_1, v_2), \cdots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k))$ from E is called **walk** in G from v_0 to v_k of length k.
- **Paths:** The walk p is called **path** if $v_i \neq v_j$ for all $1 \leq i < j \leq k$.
- **Cycles:** the path p is called **cycle** if $v_0 = v_k$.
- **Shortest Paths:** A path p from a node u to a node v is called **shortest path from** u **to** v if for all paths p' from u to v it holds $w(p') \geq w(p)$.

Distances in Weighted Graphs

The **Distance** $\delta_G(u, v)$ **from node** u **to node** v is defined to be

- $\delta_G(u, v) = \infty$ if v is not reachable from u in G (i.e., there is no walk from u to v).
- $\delta_G(u, v) = -\infty$ if there is a walk from u to v containing a cycle K of negative weight (i.e., there are arbitrarily short walks from u to v obtained by going through K correspondingly often).
- $\delta_G(u, v) = w(p)$ if v is reachable from u in G and no walk from u to v contains a negative cycle, and if p denotes a shortest path from u to v.

Properties of Shortest Paths

Lemma 5

Let $G = (V, E, w)$ be a weighted graph, and let $s, u, v \in V$.

1. ***Length of Shortest Paths:*** *If $-\infty < \delta_G(u, v) < \infty$ then there is a shortest walk p from u to v which is a path with at most $|V| - 1$ edges.*
2. ***Monotonicity:*** *If p is a shortest path from u to v passing two nodes w and z in this order. Then the subpath of p from w to z is also a shortest path.*
3. ***Triangle Inequality:*** *If $(u, v) \in E$ then it holds that $\delta_G(s, v) \leq \delta_G(s, u) + w(u, v)$.*

Proof of Lemma 5

Proof of 1: As no negative cycles exist, each cycle in a shortest walk has to have nonnegative weight and can be cancelled.

Proof of 2: If there were a shorter path from w to z we could construct a shorter path from u to v.

Proof of 3: It holds that either

- u not reachable from s, i.e., $u.d = \infty$, or
- u is reachable from s, but there is no shortest path from s to v going through u, i.e., $\delta_G(s, v) < \delta_G(s, u) + w(u, v)$, or
- u is reachable from s and there is a shortest path from s to v going through u, i.e., $\delta_G(s, v) = \delta_G(s, u) + w(u, v)$.

Single Source Shortest Path: Data Structures

Many interesting algorithm for Single Source Shortest Path refer to the following data structure

- All the nodes v of input graph $G = (V, E, w)$ have components $v.d \in \mathbb{R}$ and $v.\pi \in V$.
- **Input:**
  - $s.d = 0$
  - $v.d = \infty$ for all $v \neq s \in V$,
  - $v.\pi = NIL$ for all $v \in V$
- **Output:** (if no negative cycles are reachable from s)
  - The subgraph $G_\pi = (V_\pi, E_\pi)$ forms a **Shortest Path Tree** with source s containing all nodes reachable from s, where
    - $V_\pi = \{v \in V, v.d < \infty\}$,
    - $E_\pi = \{(v.\pi, v); v \in V_\pi, v.\pi \neq NIL\}$
  - $v.d = \delta_G(s, v)$ for all $v \in V$,
  - $v.\pi$ denotes the predecessor of v along a shortest path from s to v, for all $v \in V_\pi, v.\pi \neq NIL$.

Basic Operations *Initialize* and *Relax*

Initialize(G, s)

1 **For all** $v \in V$
2 **do** $v.d \leftarrow \infty$
3 $v.\pi \leftarrow NIL$
4 $s.d \leftarrow 0$

Running time $O(|V|)$.

Relax(u, v, w)

1 **If** $v.d > u.d + w(u, v)$
2 // Relax edge (u, v)
3 **then** $v.d \leftarrow u.d + w(u, v)$
4 $v.\pi \leftarrow u$

Running time $O(1)$.

Main Observation

Apply *Initialize*(G, s), followed by a finite sequence of RELAX-operations, to $G = (V, E, w)$, a weighted graph with source $s \in V$.

Then one gets a subgraph $G_\pi = (V_\pi, E_\pi)$, where

- $V_\pi = \{v \in V, v.d < \infty\}$
- $E_\pi = \{(v.\pi, v), v.\pi \neq NIL\}$

Lemma 6

Suppose that no negative cycle is reachable from s. Then G_π is always a tree with root s, and for each $v \in V_\pi$ it holds that

$$v.d = \delta_{G_\pi}(s, v) \geq \delta_G(s, v),$$

i.e., v.d denotes the length of the path from s to v in the tree G_π.

Proof of Lemma 6,I

We assume $E_\pi \neq \emptyset$. Otherwise $V_\pi = \{s\}$, which makes the Lemma trivially true.

Remember that a directed graph is a tree with root s if and only if for each node v there is exactly one path from s to v.

We know that node s has indegree 0 and that all other nodes in G_π have indegree 1.

This implies that G_π is a disjoint union of one tree with root s and some cycles.

Consequently, we have to show that if there do not exist negative cycles in G, which are reachable from s, then G_π is acyclic.

We show that any cycle $K = (v_1, \cdots, v_k, v_1)$ in G_π defines a cycle in G with negative weight, which is reachable from s.

Proof of Lemma 6, II

W.l.o.g. let (v_k, v_1) be the last edge in K, which is relaxed and consider the situation directly before relaxing (v_k, v_1).

- The edges $(v_1, v_2), \cdots, (v_{k-1}, v_k)$ are already relaxed which implies

$$v_k.d = v_{k-1}.d + w(v_{k-1}, v_k) = \cdots = v_1.d + \sum_{i=2}^{k} w(v_{i-1}, v_i).$$

- However, as (v_k, v_1) is going to be relaxed, $v_1.d > v_k.d + w(v_k, v_1)$, which implies

$$v_1.d > v_1.d + \sum_{i=2}^{k} w(v_{i-1}, v_i) + w(v_k, v_1) = v_1.d + w(K).$$

Consequently $w(K) < 0$. $\square$

The Bellman-Ford Algorithm

BellmanFord(G, w, s)

1 *Initialize*(G, s)
2 **For** $i \leftarrow 1$ **to** $|V| - 1$
3 **do for all** $(u, v) \in E$
4 **do** *Relax*(u, v, w)
5 **For all** $(u, v) \in E$
6 **do if** $v.d > u.d + w(u, v)$
7 **then return false , STOP**
8 **return true**

Running time $O(|V||E|)$

Correctness of the Bellman-Ford Algorithm, I

Theorem 7

BellmanFord(G, w, s) *outputs* ***true*** *if and only if no negative cycle is reachable from s. In this case, the output* G_π *is a shortest path tree with root s which contains all nodes v which are reachable from s.*

Proof: If no negative cycle is reachable from s, for each node v, which is reachable from s, there is a shortest path from s to v with at most $|V| - 1$ edges.

Let $v \in V$ be reachable from s and $P = (e_1, \cdots, e_k)$ be a shortest path from s to v, $k \leq |V| - 1$, where for all i, $1 \leq i \leq k$, $e_i = (v_{i-1}, v_i)$ and $v_0 = s$ and $v_k = v$.

We show by induction over i that for all i, $1 \leq i \leq k$, after round it holds $(v_j).d = \delta_G(s, v_j)$ for all $j = 1, \cdots, i$.

Consequently, $v.d = \delta_G(s, v)$ after round k.

Correctness of the Bellman-Ford Algorithm, II

Case i=1: In round 1, the edge e_1 is relaxed. As $(v_0).d = 0$, $(v_1).d$ gets value $w(e_1)$ which equals $\delta_G(v_1)$, as p is a shortest path.

Case i>1: By the induction hypothesis, $(v_j).d = \delta_G(v_j)$ for all $j = 1, \cdots, i-1$. In round i, edge e_i is relaxed. As $(v_{i-1}).d = \delta_G(v_{i-1})$, $(v_i).d$ gets value $\delta_G(v_{i-1}) + w(e_i)$ which equals $\delta_G(v_i)$, as p is a shortest path.

Consequently, $v.d = \delta_G(s, v)$ after round k. Thus, G_π is a shortest path tree in G with root s.

As $v.d = \delta_G(s, v)$ for all $v \in V$, we obtain by the Triangel Inequality from Lemma 5 that $v.d \leq u.d + w(u, v)$ for all edges $(u, v) \in E$.

Correctness of the Bellman-Ford Algorithm, III

We still have to show that the Bellman-Ford Algorithm returns **false** if and only if G contains a negative cycle reachable from s.

We showed already one direction, if G does not contain a negative cycle reachable from s, then G_π is the shortest path tree with root s, and, due to the Triangle Equation (see Lemma 5), the output is **true**.

It remains to show that output **true** implies that G does not contain a negative cycle reachable from s.

This follows from the following lemma.

Lemma 8

Suppose there is a function $f : V \longrightarrow \mathbb{R} \cup \{\infty\}$ fulfilling $f(v) \leq f(u) + w(u, v)$ for all edges $(u, v) \in E$, where $f(v) \neq \infty$ for all nodes v reachable from s. Then all cycles $K = (v_1, \cdots, v_k, v_1)$ which are reachable from s have positive weight.

Correctness of the Bellman-Ford Algorithm, IV

Proof of Lemma 8: Let $K = (v_1, \cdots, v_k, v_1)$ be a cycle reachable from s. By assumption:

$$f(v_i) \leq f(v_{i-1}) + w(v_{i-1}, v_i)$$

for all $i = 2, \cdots, k$, and

$$f(v_1) \leq f(v_k) + w(v_k, v_1).$$

Summing up these k inequalities yields

$$\sum_{i=2}^{k} f(v_i) + f(v_1) \leq \sum_{i=1}^{k-1} f(v_i) + f(v_k) + \sum_{i=2}^{k-1} w(v_{i-1}, v_i) + w(v_k, v_1).$$

Consequently,

$$\sum_{i=1}^{k} f(v_i) \leq \sum_{i=1}^{k} f(v_i) + w(K)$$

which implies $w(K) \geq 0$. $\square$

Optimization problems

Input instances for many optimization problems can be formulated as instances $I = (n, c, R_1, \cdots, R_p, goal)$ for an ***M*-optimization problem**, $M \subseteq \mathbb{R}$, consisting of

- A set of n variables $x_1, \cdots, x_n$ taking values of M
- A target function $c = c(x_1, \cdots, x_n) : M^n \longrightarrow \mathbb{R}$.
- A number of restriction $R_1, \cdots, R_p : M^n \longrightarrow \{false, true\}$.
- A **goal** (**min** or **max**).

Most frequent cases $M = \mathbb{R}$ or $M = \mathbb{Z}$ or $M = \{0, 1\}$.

Set of valid solutions of *I*:

$$X(I) = \{x \in M^n, R_1(x) = \cdots = R_p(x) = true\}.$$

Solving I means finding $x^* \in X(I)$ with

$$c(x^*) = \textbf{goal}\{c(x), x \in X(I)\}.$$

Example: A Political Problem

Politician Jack tries to win his district consisting of 100,000 urban, 200,000 suburban, and 50,000 rural registered voters.

Jacks primary issues are

1. building more roads $(-2, 5, 3)$
2. more gun control $(8, 2, -5)$
3. more farm subsidies $(0, 0, 10)$
4. more gasoline tax $(10, 0, -2)$

The numbers show the effect of investing \$1000 for advertisement for each of these issues (in thousand voters).

Determine the minimal amount of money necessary for gaining 50,000 urban and 100,000 suburban and 25,000 rural votes.

Formal Specification

- Variables are x_1, x_2, x_3, x_4 corresponding to the investments in advertisement for the four issues.
- The target cost function is $c(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3 + x_4$.
- Three restrictions corresponding to the gain of urban, suburban, and rural votes, i.e.
  - R_1: $-2x_1 + 8x_2 + 10x_4 \geq 50$
  - R_2: $5x_1 + 2x_2 \geq 100$
  - R_3: $3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$
- The additional restrictions $x_j \geq 0$ for $j = 1, 2, 3, 4$.

Linear Optimization

Definition 9

- A restriction is called to be a **linear restriction** if it has the form $L(x) = b$ or $L(x) \leq b$, or $L(x) \geq b$, where $L : \mathbb{R}^n \longrightarrow \mathbb{R}$ denotes a linear function $L(x_1, \cdots, x_n) = \sum_{j=1}^{n} a_j x_j$ for real coefficients $a_1, \cdots, a_n$.
- An Opt-Instance $I = (n, c, R_1, \cdots, R_p, goal)$ is called to be **Linear Programming Instance (LP-instance)** if the target function $c(x) = \sum_{j=1}^{n} c_j x_j - z$ is affine and all restrictions are linear.

Many important practical problems correspond to LP-instances (see our example), and many important optimization problems can be formulated as LP-problems (e.g. Shortest Path).

Formulating Shortest Path as LP-problem

... given a weighted graph $G = (V, E, w)$ and $s \in V$. We suppose that all $v \in V$ are reachable from s. (Otherwise apply $BFS(G, s)$ for computing all nodes reachable from s in time $O(|V| + |E|)$).

Formulation as LP-instance $I(G)$ over real variables $\{v.d, v \in V\}$:

- Target function $c(d) = \sum_{v \in V} v.d$.
- Restrictions R_e for all $e = (u, v) \in E$:

$$v.d \leq u.d + w(u, v).$$

- Additional restriction $s.d = 0$.
- Goal is **maximize**

Proof of Correctness, I

We know from the Bellman-Ford algorithm that

- If G contains a negative cycle then each assignment d to the variables falsifies at least one restriction (see Lemma 8).
- $v.d \leftarrow \delta_G(s, v)$ defines a valid solution if G does not contain a negative cycle.

The correctness of our LP-formulation follows from

Lemma 10

Let G does not contain a negative cycle. Then for all valid solutions $d = (v.d)_{v \in V}$ it holds $v.d \leq \delta_G(s, v)$ for all $v \in V$ (i.e., $v.d \leftarrow \delta_G(s, v)$ defines the optimal solution).

Proof of Correctness, II

Proof of Lemma 10:

Show by induction on i that the claim is true for all $v \in V_i$, $i = 0, \cdots, |V| - 1$, where V_i denotes the set of all $v \in V$ for which the minimal number of edges of a shortest path from s to v is i.

The claim is trivially true for $i = 0$ (it holds $V_0 = \{s\}$).

Suppose that it is true for $j = 0, \cdots, i - 1$ and fix some $v \in V_i$.

Fix further $u \in V_{i-1}$ being the predecessor of v on a shortest path from s to v. Then

$$v.d \leq u.d + w(u, v) \leq \delta_G(s, u) + w(u, v) = \delta_G(s, v). \square$$

Solving Linear Programs

Convex Sets

Definition 11

For $x, y \in \mathbb{R}^n$ let $Line(x, y) = \{x + \lambda(y - x), \lambda \in [0, 1]\}$ denote the **line** connecting x and y.

Definition 12

A subset $X \subseteq \mathbb{R}^n$ is called **convex** if for all $x, y \in X$ it holds that $Line(x, y) \subseteq X$.

Definition 13

Let $X \subseteq \mathbb{R}^n$ be **convex**. A point $x \in X$ is called **inner point** if there are points $y, z \in X$ which are distinct from x and for which $x \in Line(y, z)$. The point $x \in X$ is called **extremal point** if it is not inner point.

Definition 14

For points $x, y \in \mathbb{R}^n$ let $|x - y| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$ denote the Euclidian distance of x, y.

For $x \in \mathbb{R}^n$ and $\epsilon > 0$ let $Ball(x, \epsilon) = \{y, |x - y| \leq \epsilon\}$ denote the ϵ-environment of x.

Optimal points

Definition 15

Let $c : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a function and $X \subseteq \mathbb{R}^n$.

- A point $x \in X$ is called a **global minimum** (resp. **global maximum**) of X w.r.t. c if $c(x) \leq c(y)$ (resp. $c(x) \geq c(y)$) for all $y \in X$.
- It is called a **local minimum** (resp. **local maximum**) of X w.r.t. c if there is some $\epsilon > 0$ such that x is global minimum (resp. global maximum) of $X \cap Ball(x, \epsilon)$ w.r.t. c.

Theorem 16

Let $X \subseteq \mathbb{R}^n$ be a convex set and let $c : \mathbb{R}^n \longrightarrow \mathbb{R}$ be a linear target function.

(1) *Each local minimum (resp. local maximum) of X w.r.t. c is a global minimum (resp. local maximum) of X w.r.t. c.*

(2) *If X has a maximum (resp. minimum) w.r.t c then it is taken in an extremal point of X.*

Proof of Theorem 16, Part (1)

Suppose that $x \in X$ is a local maximum w.r.t. c and fix some $\epsilon > 0$ such that x is global maximum of $X \cap \mathit{Ball}(x, \epsilon)$ w.r.t. c.

Assume that there is some $y \in X$ with $c(y) > c(x)$ and fix some $\delta > 0$ such that $z = x + \delta(y - x) = (1 - \delta)x + \delta y \in \mathit{Ball}(x, \epsilon)$.

As $\mathit{Line}(x, y) \subseteq X$ it holds $z \in X \cap \mathit{Ball}(x, \epsilon)$, but

$$c(z) = c((1 - \delta)x + \delta y) = (1 - \delta)c(x) + \delta c(y) > c(x),$$

contradiction to x being the maximum in $X \cap \mathit{Ball}(x, \epsilon)$. □

Theorem 17

Let $I = (n, c, R_1, \cdots, R_p, goal)$ be an LP-instance. Then $X(I) \subseteq \mathbb{R}^n$ ***is convex****.*

Proof: This follows from the easy provable fact that if a linear restriction R_i is true for $x \neq y \in \mathbb{R}^n$ then it is also true for $(1 - \lambda)x + \lambda y$ for all λ, $0 \leq \lambda \leq 1$. □

Characterizing Extremal Points of Linear Programs

Let $I = (n, c, R_1, \cdots, R_p, goal)$ be an LP-instance, where all restrictions R_i, $1 \leq i \leq p$, have the form $L_i(x) \leq b_i$ or $L_i(x) \geq b_i$ or $L_i(x) = b_i$, where $L_i(x_1, \cdots, x_n) = \sum_{j=1}^{n} a_{i,j} x_j$.

Definition 18

A subset of restrictions $\{R_i, i \in S\}$, where $S \subseteq \{1, \cdots, p\}$, is called to be **linearly independent**, if the set of corresponding coefficient vectors $\{(a_{i,1}, \cdots, a_{i,n}), i \in S\} \subseteq \mathbb{R}^n$ is linearly independent, i.e., the rank of the matrix $A[S]$, formed by the rows $\{(a_{i,1}, \cdots, a_{i,n}), i \in S\}$, equals $|S|$.

Theorem 19

A point $x \in X(I)$ is an extremal point in $X(I)$ if and only if x satisfies a subset of n linearly independent restrictions with equality.

Proof of Theorem 19, I

Proof: Let w.l.o.g. $x^* \in X(I)$ fulfill exactly relations $R_1, \cdots, R_k$ with equality, denote by X^* the affine subspace of all points x fulfilling $R_1, \cdots, R_k$ with equality, and let $r \leq n$ denote the rank of the set of coefficient vectors $\{(a_{i,1}, \cdots, a_{i,n}), 1 \leq i \leq k\}$.

We show first that x^* is not extremal if $r < n$.

As $r < n$, the dimension of X^* is $n - r > 0$, which implies that X^* contains a line L^* with x^* as inner point.

Moreover, for all i, $k + 1 \leq i \leq p$, x^* has a positive distance to the hyperplane $H_i = \{x, L_i(x) = b_i\}$.

Consequently, there exist some $\epsilon > 0$ such that all points in $Ball(x^*, \epsilon)$ satisfy all relation R_i, $k + 1 \leq i \leq p$.

This implies that $L^* \cap Ball(x^*, \epsilon)$ defines a line in $X(I)$ containing x^* as an inner point, x^* is not extremal.

Proof of Theorem 19, II

Now suppose that $r = n$, i.e. $X^* = \{x^*\}$.

For deriving a contradiction suppose that x^* is not extremal and fix some points $y, z \in X(I)$ such that x^* is an inner point of the line between x and y, i.e., there is some λ, $0 < \lambda < 1$, with $x^* = \lambda \cdot y + (1 - \lambda) \cdot z$.

As $y \notin X^*$, there is some i, $1 \leq i \leq k$, such that y does not satisfy R_i with equality. This implies that R_i has the form $L_i(x) \leq b_i$ or $L_i(x) \geq b_i$, as in the case $L_i(x) = b_i$ the point y would not satisfy R_i.

We assume R_i has the form $L_i(x) \leq b_i$, which implies that $L_i(y) < b_i$.

But this implies that $L_i(z) > b_i$, which is a contradiction to $z \in X(I)$.

This is because in the case $L_i(z) \leq b_i$ we had

$$L_i(x^*) = \lambda \cdot L_i(y) + (1 - \lambda) \cdot L_i(z) < \lambda \cdot b_i + (1 - \lambda) \cdot L_i(z) \leq \lambda \cdot b + (1 - \lambda) \cdot b_i = b_i,$$

which contradicts to the assumption that $L_i(x) = b_i$. $\square$

Comments

- Theorem 19 defines an **Exhaustive Search Algorithm** for Linear Programming.

  (1) For all subsets $S \subseteq \{1, \cdots, p\}$, $|S| = n$ do
  (2) test if the set of linear restrictions $\{R_i, i \in S\}$ is linearly independent
  (3) If yes, compute x^S satisfying all relations in $\{R_i, i \in S\}$ with equality and test if $x^* \in X(I)$
  (4) If yes, put x^* to a set *Extr*, which is initially empty
  (5) Output a point $x^* \in Extr$ with optimal c-value.

- If an optimum exists, the exhaustive search algorithm finds an optimal point in $X(I)$.
- **Running Time:** $O(\binom{p}{n})$ iterations, where the rank determination in step 3, and solving a system of linear equations in step 4 are the most expensive ones (cost $O(n^3)$). This yields overall running time $O(\binom{p}{n} \cdot n^3)$.
- **Caution:** This implies **exponential running time!** (Note, e.g., that $\binom{2n}{n} > 2^n$.)
- **Important Question:** Can we do better?

LP-Instances in Normal Form

An LP-instance is called to be of **normal form** if it is defined as

- Maximize $\sum_{j=1}^{n} c_j x_j - z$ under
- $\sum_{j=1}^{n} a_{i,j} x_j \leq b_i$ for all $i = 1, \cdots, m$, and
- $x_j \geq 0$ for $j = 1, \cdots, n$.

Consequently, an n-dimensional LP-instance in normal form corresponds to a tupel

$$I = (n, m, c, z, A, b)$$

where n, m are naturals, $c = (c_1, \cdots, c_n) \in \mathbb{R}^n$, $z \in \mathbb{R}$, $A = (a_{i,j})_{i,j=1}^{m,n} \in \mathbb{R}^{m \times n}$ and $b = (b_1, \cdots, b_m) \in \mathbb{R}^m$.

Transforming general LP-Instances into Normal Form

- Make **min** to **max** by multiplying the target function by -1,
- Replace equalities $\sum_{j=1}^{n} a_{i,j}x_j = b_i$ by two inequalities

$$\sum_{j=1}^{n} a_{i,j}x_j \leq b_i,\ \sum_{j=1}^{n}(-a_{i,j})x_j \leq -b_i,$$

  or replace the equality and put

$$x_n = \frac{b_i}{a_{i,n}} - \sum_{j=1}^{n-1} \frac{a_{i,j}}{a_{i,n}} x_j$$

  into the other restrictions.
- Replace inequalities $\sum_{j=1}^{n} a_{i,j}x_j \geq b_j$ by $\sum_{j=1}^{n}(-a_{i,j})x_j \leq -b_j$
- Replace unrestricted variables x_j by

$$x_j^+ - x_j^-,\ x_j^+ \geq 0,\ x_j^- \geq 0.$$

Slacking Extensions

Definition 20

For each point $d = (d_1, \cdots, d_n) \in \mathbb{R}^n$ we denote by

$$\left(d_1, \cdots, d_n, b_1 - \sum_{j=1}^{n} a_{1,j} d_j, \cdots, b_m - \sum_{j=1}^{n} a_{m,j} d_j \right) \in \mathbb{R}^{n+m}$$

the **slacking extension** of d.

Lemma 21

(i) *$d \in \mathbb{R}^n$ fulfils n linear independent restrictions in I with equality if the **slacking extension** of d is zero at n linearly independent positions.*

(ii) *$d \in X(I)$ iff the **slacking extension** of d has only nonnegative components,*

(iii) *d is an **extremal point** in $X(I)$ iff $d \geq 0$ and the **slacking extension** of d is zero at n linearly independent positions.* □

Slacking Normal Form Instances

Given an normal form LP-instance $I = (m, n, c, z, A, b)$ over $x = (x_1, \cdots, x_n)$, the equivalent **Slacking Normal Form Instance** $SNF(I)$ is defined as follows

Definition 22

- Maximize $\sum_{j=1}^{n} c_j x_j - z$ under
- $x_1 \geq 0, \cdots, x_n \geq 0, x_{n+1} \geq 0, \cdots, x_{n+m} \geq 0$, where for all $i = 1, \cdots, m$
- the **slacking variables** x_{n+i} are defined by $x_{n+i} = b_i - \sum_{j=1}^{n} a_{i,j} x_j$.

Notes

- The slacking variable $x_{n+i} = b_i - \sum_{j=1}^{n} a_{i,j} x_j$ measures the **slacking distance** of $\sum_{j=1}^{n} a_{i,j} x_j$ from the bound b_i.
- The point $d \in \mathbb{R}^n$ is a valid solution of I iff the slacking extension of d is a valid solution of $SNF(I)$.

Basic Points

Let $d = (d_1, \cdots, d_{n+m}) \in \mathbb{R}^{n+m}$ denote the slacking extension of a point $(d_1, \cdots, d_n)$.

Definition 23

- Point d is called a **basic point**, if there is a set $N \subset \{x_1, \cdots, x_{n+m}\}$ of n linearly independent variables such that $d_k = 0$ for all $x_k \in N$. In this case, N is called the set of **non-basic variables** of the basic point d and $B = \{x_1, \cdots, x_{n+m}\} \setminus N$ is called the set of **basic variables** of d.
- A basic point d is called **admissible basic point** if $d_k \geq 0$ for all k, $1 \leq k \leq n + m$.
- The basic point $(\vec{0}, b) = (0, 0, \cdots, 0, b_1, \cdots, b_m)$ is called the **canonical basic point**. It is admissible iff $b \geq \vec{0}$.

Note: The admissible basic points are exactly the slacking extensions of the extremal points of the polyhedron defined by $X(I) = \{x;\ A \circ x \leq b,\ x \geq 0\}$.

The Simplex Tableaux corresponding to $SNF(I, N)$

Let $I = (n, m, c, z, A, b)$ a normal form LP-instance, $N = \{x_1, \cdots, x_n\}$ and $B = \{x_{n+1}, \cdots, x_{n+m}\}$.

We collect all information describing the slacking normal form instance $SNF(I, N)$ in a data structure called the **Simplex Tableaux** $T(I, N)$:

T(I,N,B):=

		x_1	$\cdots$	x_n
	z	c_1	$\cdots$	c_n
x_{n+1}	b_1	a_{11}	$\cdots$	a_{1n}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_{n+m}	b_m	a_{m1}	$\cdots$	a_{mn}

The Simplex Method

for non-negative restriction vectors $b \geq \vec{0}$

First Steps

First we extract from $T(I, N)$ the information

- if $(\vec{0}, b)$ is optimal, or,
- if the problem is unbounded, i.e., $\max = \infty$, or,
- which neighbor admissible basic point of $(\vec{0}, b)$ improves the target function.

Directions starting from $(\vec{0}, b)$

Note: The point $(\vec{0}, b)$ is left by n lines $D_1, \cdots, D_n$, called **directions**, where for all q, $1 \leq q \leq n$, direction D_q is defined by increasing component q, i.e.,

$$D_q = \{d_q(\lambda),\ \lambda \geq 0\},$$

where

$$d_q(\lambda) = (0 \cdots, 0, \lambda, 0 \cdots, 0, b_1 - \lambda \cdot a_{1,q}, \cdots, b_m - \lambda \cdot a_{m,q}).$$

where λ stands at position q.

- A direction D_q is called **bounded** if there is some bound $\lambda_0 \geq 0$ such that $d_q(\lambda)$ is admissible if and only if $0 \leq \lambda \leq \lambda_0$.
- A direction D_q is called **improving** if the target function strictly increases on D_q with increasing λ.

Improving Directions and Optimality of $(\vec{0}, b)$

Lemma 24

Suppose that $(\vec{0}, b)$ is an admissible basic point.

(i) *A direction D_j, $1 \leq j \leq n$, is improving if and only if $c_j > 0$.*

(ii) *If $c_j \leq 0$ for all j, $1 \leq j \leq n$, then $(\vec{0}, b)$ is optimal.*

Proof of (i): Observe that the target function c behaves at direction D_j as

$$c(d_j(\lambda)) = c_j \cdot \lambda.$$

Consequently, if c is (strictly) increasing on D_j then $c_j > 0$.

Proof of (ii): Note that all admissible points $x \in X(I)$ in the environment of $\vec{0}$ have to have only nonnegative components. Consequently, if $c_j \leq 0$ for all j, $1 \leq j \leq n$, then $\vec{0}$ is a local optimum as $c(x) \leq c(\vec{0}) = 0$ for all $x \in X(I)$ in the environment of $\vec{0}$. By Theorems 16 and 17, $\vec{0}$ is optimal. □

Bounded and Unbounded Directions

Lemma 25

Suppose that $(\vec{0}, b)$ *is an admissible basic point.*

(i) *A direction* D_j, $1 \leq j \leq n$, *is bounded if and only if there is some* i, $1 \leq i \leq m$, *such that* $a_{i,j} > 0$.

(ii) *In this case* $d_j(\lambda) \in X(I)$ *if and only if* $0 \leq \lambda \leq \min\{\frac{b_i}{a_{i,j}}; 1 \leq i \leq m, a_{i,j} > 0\}$.

Proof: Note that $d_j(\lambda) \in X(I)$ if $b_i - a_{i,j} \cdot \lambda \geq 0$, i.e., $\lambda \leq \frac{b_i}{a_{i,j}}$, for all i, $1 \leq i \leq m$, fulfilling $a_{i,j} > 0$. □

Lemma 26

If there is some j, $1 \leq j \leq n$, *such that* $c_j > 0$ *and* $a_{i,j} \leq 0$ *for all* i, $1 \leq i \leq m$, *then* D_j *is an* ***improving unbounded direction****, i.e.,* $opt(I) = \lim_{\lambda \to \infty} c_j \cdot \lambda = \infty$. □

Pivot Positions

Definition 27

Suppose that $(\vec{0}, b)$ is admissible.

A pair of indicees (p, q), $1 \leq p \leq m$, $1 \leq q \leq n$, is called a **Pivot Position**

(1) $c_q > 0$ and $a_{p,q} > 0$,
(2) $\frac{b_p}{a_{p,q}} = \min\{\frac{b_i}{a_{i,q}}, 1 \leq i \leq m, a_{i,q} > 0\}$.

Lemma 28

Suppose that $(\vec{0}, b)$ is admissible but not optimal, and that all improving directions starting at $(\vec{0}, b)$ are bounded. Then there is a Pivot position (p, q), and for the point $y = d_q(\frac{b_p}{a_{p,q}})$ it holds that y is an admissible basic point, and that $c(y) > c((\vec{0}, b))$.

Proof of Lemma 28

Note first that, due to definition 27

$$c(y) = c_q \cdot \frac{b_p}{a_{p,q}} > 0.$$

Note further that

$$y_{n+p} = b_p - a_{p,q} \cdot \frac{b_p}{a_{p,q}} = 0,$$

i.e. $y = d_q(\frac{b_p}{a_{p,q}})$ is zero at positions $\{1, \cdots, n\} \setminus \{q\}$ and at position $n + p$.

It can be easily shown that the set of variables

$$(\{x_1, \cdots, x_n\} \setminus \{x_q\}) \cup \{x_{n+p}\}$$

is linearly independent if and only if $a_{p,q} \neq 0$ (which is true as (p, q) is a Pivot position). $\square$

Scheme of the Simplex Method

Input: $T = (I, N, B)$, where $I = (n, m, c, z, A, b)$ and $N = \{x_1, \cdots, x_j\}$ and $B = \{x_{n+1}, \cdots, x_{n+m}\}$.

0 $x \leftarrow (\vec{0}, b)$

1 Check if x is optimal and stop if yes.

2 Check if there is an unbounded improving direction starting at x and stop if yes.

3 Fix a pivot position (p, q), set $y = d_q(\frac{b_p}{a_{p,q}})$ and compute the simplex tableaux $T(I, \tilde{N}, \tilde{B})$ corresponding to $\tilde{N} = (\{x_1, \cdots, x_n\} \setminus \{x_q\}) \cup \{x_{n+p}\}$ and $\tilde{B} = (\{x_{n+1}, \cdots, x_{n+m}\} \setminus \{x_{n+p}\}) \cup \{x_q\}$.

4 $x \leftarrow y$

5 **goto 1**

The Simplex Transformation (1)

Let (p, q) a pivot position and $y =$

$$\left(0, \cdots, 0, \frac{b_p}{a_{p,q}}, 0, \cdots, 0, b_1 - \frac{a_{1,q}b_p}{a_{p,q}}, \cdots, b_{p-1} - \frac{a_{p-1,q}b_p}{a_{p,q}}, 0, b_{p+1} - \frac{a_{p+1,q}b_p}{a_{p,q}}, \cdots, b_m - \frac{a_{m,q}b_p}{a_{p,q}}\right)$$

the corresponding admissible neighbor basic point of $(\vec{0}, b)$.

Computing the simplex tableaux $T(I, \tilde{N}, \tilde{B})$ with $\tilde{N} = \{x_1, \cdots, x_{q-1}, x_{n+p}, x_{q+1}, \cdots, x_n\}$ means to compute coefficients $\tilde{c}_j, \tilde{z}, \tilde{b}_i, \tilde{a}_{i,j}$ such that

- $c(x) = \sum_{j=1}^{q-1} \tilde{c}_j x_j + \tilde{c}_q x_{n+p} + \sum_{j=q+1}^{n} \tilde{c}_j x_j - \tilde{z}$
- $x_{n+i} = \tilde{b}_i - \sum_{j=1}^{q-1} \tilde{a}_{i,j} x_j - \tilde{a}_{i,q} x_{n+p} - \sum_{j=q+1}^{n} \tilde{a}_{i,j} x_j$, for $i \neq p$, and
- $x_q = \tilde{b}_p - \sum_{j=1}^{q-1} \tilde{a}_{p,j} x_j - \tilde{a}_{p,q} x_{n+p} - \sum_{j=q+1}^{n} \tilde{a}_{p,j} x_j$.

The Simplex Transformation (2)

It holds

$$x_{n+p} = b_p - \sum_{j=1}^{q-1} a_{p,j} x_j - a_{p,q} x_q - \sum_{j=q+1}^{n} a_{p,j} x_j.$$

This implies

$$x_q = \frac{b_p}{a_{p,q}} - \sum_{j=1}^{q-1} \frac{a_{p,j}}{a_{p,q}} x_j - \frac{1}{a_{p,q}} x_{n+p} - \sum_{j=q+1}^{n} \frac{a_{p,j}}{a_{p,q}} x_j.$$

Hence

- $\tilde{b}_p = \frac{b_p}{a_{p,q}}$,
- $\tilde{a}_{p,j} = \frac{a_{p,j}}{a_{p,q}}$ for $j \neq q$, and
- $\tilde{a}_{p,q} = \frac{1}{a_{p,q}}$

The Simplex Transformation (3)

For $i \neq p$ it holds

$$x_{n+i} = b_i - \sum_{j=1}^{q-1} a_{i,j} x_j - a_{i,q} x_q - \sum_{j=q+1}^{n} a_{i,j} x_j.$$

This implies

$$x_{n+i} = b_i - \sum_{j=1}^{q-1} a_{i,j} x_j - a_{i,q} \left(\frac{b_p}{a_{p,q}} - \sum_{j=1}^{q-1} \frac{a_{p,j}}{a_{p,q}} x_j - \frac{1}{a_{p,q}} x_{n+p} - \sum_{j=q+1}^{n} \frac{a_{p,j}}{a_{p,q}} x_j \right) - \sum_{j=q+1}^{n} a_{i,j} x_j.$$

This can be transformed in the desired form

$$x_{n+i} = \left(b_i - \frac{a_{i,q} \cdot b_p}{a_{p,q}} \right) - \sum_{j=1}^{q-1} \left(a_{i,j} - \frac{a_{i,q} \cdot a_{p,j}}{a_{p,q}} \right) x_j - \left(-\frac{a_{i,q}}{a_{p,q}} \right) x_{n+p} - \sum_{j=q+1}^{n} \left(a_{i,j} - \frac{a_{i,q} \cdot a_{p,j}}{a_{p,q}} \right) x_j.$$

The Simplex Transformation (4)

Consequently, for $i \neq p$,

- $\tilde{b}_i = b_i - \frac{a_{i,q} \cdot b_p}{a_{p,q}}$,
- $\tilde{a}_{i,j} = a_{i,j} - \frac{a_{i,q} \cdot a_{p,j}}{a_{p,q}}$ for $j \neq q$, and
- $\tilde{a}_{i,q} = -\frac{a_{i,q}}{a_{p,q}}$.

The Simplex Transformation (5)

It holds

$$c(x) = \sum_{j=1}^{q-1} c_j x_j + c_q x_q + \sum_{j=q+1}^{n} c_j x_j - z.$$

This implies

$$c(x) = \sum_{j=1}^{q-1} c_j x_j + c_q \left(\frac{b_p}{a_{p,q}} - \sum_{j=1}^{q-1} \frac{a_{p,j}}{a_{p,q}} x_j - \frac{1}{a_{p,q}} x_{n+p} - \sum_{j=q+1}^{n} \frac{a_{p,j}}{a_{p,q}} x_j \right) + \sum_{j=q+1}^{n} c_j x_j - z.$$

This can be transformed into the desired form

$$c(x) = \sum_{j=1}^{q-1} \left(c_j - \frac{c_q \cdot a_{p,j}}{a_{p,q}} \right) x_j + \left(-\frac{c_q}{a_{p,q}} \right) x_{n+p} + \sum_{j=q+1}^{n} \left(c_j - \frac{c_q \cdot a_{p,j}}{a_{p,q}} \right) x_j - \left(z - \frac{c_q \cdot b_p}{a_{p,q}} \right).$$

The Simplex Transformation (5)

This implies

- $\tilde{c_j} = c_j - \frac{c_q \cdot a_{p,j}}{a_{p,q}}$ for $j \neq q$,
- $\tilde{c_q} = -\frac{c_q}{a_{p,q}}$,
- $\tilde{z} = z - \frac{c_q \cdot b_p}{a_{p,q}}$.

Result

We obtain the simplex tableaux

$$T(I,\tilde{N},\tilde{B}) := \begin{array}{|c|c|ccc|}\hline & & \tilde{x_1} & \cdots & \tilde{x_n} \\ \hline & \tilde{z} & \tilde{c_1} & \cdots & \tilde{c_n} \\ \hline \tilde{x_{n+1}} & \tilde{b_1} & \tilde{a_{11}} & \cdots & \tilde{a_{1n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tilde{x_{n+m}} & \tilde{b_m} & \tilde{a_{m1}} & \cdots & \tilde{a_{mn}} \\ \hline \end{array}$$

which corresponds to the slacking normal form $SNF(I,\tilde{N},\tilde{B})$

- **Maximize** $\sum_{j=1}^{n} c_j x_j - z = \sum_{j=1}^{n} \tilde{c_j}\tilde{x_j} - \tilde{z}$
- under $\tilde{x_{n+i}} = \tilde{b_i} - \sum_{j=1}^{n} \tilde{a_{i,j}}\tilde{x_j}$ for all $i = 1, \cdots, m$,
- and $\tilde{x_k} \geq 0$ for all $k = 1, \cdots, n+m$.

Simplex Tableaux

Definition 29

- An (m,n)-Simplex Tableaux $T = (t_{i,j})_{0 \leq i \leq m, 0 \leq j \leq n}$ is a real $(m+1) \times (n+1)$ matrix.
- Rows and columnes are labelled by variables from $\{x_1, \cdots, x_{n+m}\}$ such that each of these variables occurs exactly once as a label.
- $N(T)$ denotes the set of variables occuring as columne labels and is called the set of non basic variables w.r.t. T.
- $B(T)$ denotes the set of variables occuring as row labels and is called the set of basic variables w.r.t. T.
- $x(T) \in \mathbb{R}^{n+m}$ denotes the basic point corresponding to T and is defined as $x(T)_k = 0$ if $x_k \in N(T)$ and $x(T)_k = t_{i_k,0}$, where i_k, $1 \leq i_k \leq m$, is the index of the row labelled by x_k, when $x_k \in B(T)$.

.

The Simplex Tableaux $T = T(I, N)$

The Simplex Tableaux $T = T(I, N)$, $I = (m, n, A, b, c, z)$ corresponding to our LP-instance is defined as follows:

- $t_{0,0} = z$,
- $t_{0,j} = c_j$ for $1 \leq j \leq n$,
- $t_{i,0} = b_i$ for $1 \leq i \leq m$,
- $t_{i,j} = a_{i,j}$ for $1 \leq i \leq m$, $1 \leq j \leq n$,
- $N(T) = N = \{x_1, \cdots, x_n\}$,
- $B(T) = \{x_{n+1}, \cdots, x_{n+m}\}$.

Note that $x(T) = (\vec{0}, b)$.

The Simplex Tableaux Transformation

Given an (m, n)-Simplex Tableaux T, the Transformation $\tilde{T} = Pivot_{p,q}(T)$, $1 \leq p \leq m$, $1 \leq q \leq n$ is possible if $t_{p,q} \neq 0$ and is defined as

1. $\tilde{t}_{p,q} = \frac{1}{t_{p,q}}$,
2. $\tilde{t}_{p,j} = \frac{t_{p,j}}{t_{p,q}}$, for $j \neq q$,
3. $\tilde{t}_{i,q} = -\frac{t_{i,q}}{t_{p,q}}$, for $i \neq p$,
4. $\tilde{t}_{i,j} = t_{i,j} - \frac{t_{i,q} \cdot t_{p,j}}{t_{p,q}}$, for $i \neq p$ and $j \neq q$,
5. exchange the labels of row p and columne q.

Note that if $T = T(I, N)$ then $\tilde{T} = T(I, \tilde{N})$ and that $x(\tilde{T}) = y$.

Known definitions for Simplex Tableaus (1)

Consider an (m, n)-Simplex Tableaux T.

Definition 30

- T is called **admissible** if $t_{i,0} \geq 0$ for all $i = 1, \cdots, m$.
- T is called **optimal** if T is admissable and $t_{0,j} \leq 0$ for all $j = 1, \cdots, n$.
- T is called **unbounded** if there is some q, $1 \leq q \leq n$, such that $t_{0,q} > 0$ and $t_{i,q} \leq 0$ for all i, $1 \leq i \leq m$.
- Suppose that T is admissible then (p, q), $1 \leq p \leq m$, $1 \leq q \leq n$, is called a **Pivot Position** of T if

  (1) $t_{0,q} > 0$ and $t_{p,q} > 0$,

  (2) $\frac{t_{p,0}}{t_{p,q}} = \min\{\frac{t_{i,0}}{t_{i,q}}, 1 \leq i \leq m, t_{i,q} > 0\}$.

The Main Program of the Simplex Method

SimplexSearch(T) (* T admissible Simplex Tableaux *)

1 **Repeat if** T not optimal
2 **then if** T not unbounded
3 **then choose** Pivot Position (p, q)
4 $T \leftarrow Pivot_{p,q}(T)$
5 **until** T optimal or unbounded
6 **Output** T

Correctness

Theorem 31

Suppose that SimplexSearch(T(I,N)) ***terminates*** *and let $\bar{T}$ be the output tableaux. Then the following holds*

- *$\bar{T}$ is unbounded if and only if c is unbounded on $X(I)$.*
- *If $\bar{T}$ is not unbounded then the admissible basic point $x(\bar{T})$ is the optimum, i.e.,*

$$c(x(\bar{T})) = -\bar{t}_{0,0} = \max\{c(x), x \in X(I)\}. \square$$

Two Problems still to be solved

(1) How we can ensure that *SimplexSearch(m,n,T)* terminates?

(2) How we can solve the problem if $(\vec{0}, b)$ is not admissible, i.e., if there is some i, $1 \leq i \leq m$, for which $b_i < 0$?

(3) How we can detect if the LP-instance $I = (m, n, A, b, c)$ is **infeasible**, i.e., $X(I) = \emptyset$?

Note: Infeasibility implies that $(\vec{0}, b)$ is not admissible, otherwise $X(I) \neq \emptyset$.

Degenerate Pivot steps

Lemma 32

Degenerate Pivot steps: *If $T' = Pivot_{p,q}(T)$ and $t_{p,0} = 0$ then $x(T) = x(T')$.*

Proof: This is true as for all position $(i, 0)$ in the leftmost column of T' it holds

$$t'_{i,0} = t_{i,0} - \frac{t_{i,q} \cdot t_{p,0}}{t_{p,q}} = t_{i,0}$$

for $i \neq p$ and $t'_{p,0} = \frac{t_{p,0}}{t_{p,q}} = 0 = t_{p,0}$. $\square$

Comments: Degenerate Pivot can occur in admissible basic points with more than n zeros. They have to be performed if for all pivot positions (p, q) in T it holds $t_{0,p} = 0$.

Making a degenerate Pivot step means staying in the same basic points but changing to another set of n non-basic variables (which hopefully makes an improving direction visible).

Consequences

- If *SimplexSearch(T)* performs only non degenerate Pivot steps then it always terminates.
- If it performs also degenerate Pivot steps then it may not terminate, if the heuristic of choosing the next Pivot position is badly designed.
- By an appropriate control structure, it should be ensured that *SimplexSearch(T)* never visits tableaux which are defined w.r.t. the same set of nonbasic variables.

Detecting Infeasibility, Idea

How we can detect that $X(I) \neq \emptyset$ for a normal form instance $I = (n, m, c, A, b)$ with $b_p = \min\{b_i, i = 1, \cdots, m\} < 0$.

For all $x_0 \geq 0$ we consider the set

$$X(I, x_0) = \{x \in \mathbb{R}^n; \sum_{j=1}^{n} a_{1,j}x_j \leq b_1 + x_0, \cdots, \sum_{j=1}^{n} a_{m,j}x_j \leq b_m + x_0, x \geq \vec{0}\}.$$

Observations

- $X(I, 0) = X(I)$ and $X(I, x_0) \subseteq X(I, x_0')$ for $x_0 \leq x_0'$.
- $X(I, -b_p) \neq \emptyset$ as $b_i + (-b_p) \geq 0$ for all $i = 1, \cdots, m$, and, thus $\vec{0} \in X(I, -b_p)$.

Our Approach: We compute $x_0^{\min} = \min\{x_0; x_0 \geq 0, X(I, x_0) \neq \emptyset\}$.

By Definition: $X(I) \neq \emptyset$ if and only if $x_0^{\min} = 0$.

Computing $x_0^{\min}$ and a starting tableaux for I

We compute $x_0^{\min}$ by solving the following LP-instance I_{aux}:

- **Maximize** $-x_0$ subject to
- $\sum_{j=1}^{n} a_{i,j}x_j - x_0 \leq b_i$ for all i, $1 \leq i \leq m$,
- $x_j \geq 0$ for all j, $0 \leq j \leq n$.

Theorem 33

(0) $X(I_{aux}) \neq \emptyset$ *(as* $(-b_p, 0, \cdots, 0) \in X(I_{aux})$*).*

(1) *If* $opt(I_{aux}) < 0$ *then* $X(I) = \emptyset$ *(as* $x_0^{\min} = -opt(I_{aux})$*).*

(2) *Let* $opt(I_{aux}) = 0$ *and* $(0, \bar{x}_1, \cdots, \bar{x}_n)$ *be* I_{aux}*-optimal. Then* $\bar{x} = (\bar{x}_1, \cdots, \bar{x}_n)$ *belongs to* $X(I)$ *and defines an admissible basic point of* I*.*

Proof of Theorem 33

Proof of (2): Suppose that $opt(I_{aux}) = x_0^{\min} = 0$ and fix an optimal I_{aux}-solution $(0, \bar{x}_1, \cdots, \bar{x}_n)$ which is an extremal point in $X(I_{aux})$. Note that the slacking normal form extension of this point is

$$\left(0, \bar{x}_1, \cdots, \bar{x}_n, b_1 - \sum_{j=1}^{n} a_{1,j}\bar{x}_j, \cdots, b_m - \sum_{j=1}^{n} a_{m,j}\bar{x}_j\right)$$

As this I_{aux}-basic point is admissible, all components are non negative. Moreover, besides $\bar{x}_0 = 0$, it contains n 0-components. This implies that $\bar{x}$ is an extremal point in $X(I)$ as

$$\left(\bar{x}_1, \cdots, \bar{x}_n, b_1 - \sum_{j=1}^{n} a_{1,j}\bar{x}_j, \cdots, b_m - \sum_{j=1}^{n} a_{m,j}\bar{x}_j\right)$$

is an admissible basis point w.r.t. I. □

The Tableaux $T^{aux}(I)$, $I = (m, n, c, z, A, b)$, $b \not\geq \vec{0}$

$$T^{aux}(I) := \begin{array}{|c|c|c|c|c|c|} \hline & & x_0 & x_1 & \cdots & x_n \\ \hline & 0 & -1 & 0 & \cdots & 0 \\ \hline & z & 0 & c_1 & \cdots & c_n \\ \hline x_{n+1} & b_1 & -1 & a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n+m} & b_m & -1 & a_{m1} & \cdots & a_{mn} \\ \hline \end{array}$$

Note 1: Row 2 corresponds to the original target function, which, for efficiency, we include into the transformation.

Note 2: Again, $T^{aux}(I)$ is not admissible, as $b \not\geq \vec{0}$.

Finding an admissible neighbor basic point w.r.t. $T^{aux}(I)$

We have seen that

$$\tilde{d} = (-b_p, 0, \cdots, 0) \in X(I^{aux}),$$

where $b_p = \min\{b_i, 1 \leq i \leq m\}$.

Note that the slacking extension of $\tilde{d}$ is

$$(-b_p, 0, \cdots, 0, b_1 - (-1)(-b_p), \cdots, b_m - (-1)(-b_p))$$

$$= (-b_p, 0, \cdots, 0, b_1 - b_p, \cdots, b_m - b_p) \geq 0$$

is an admissible basic point w.r.t. I^{aux}, as it has another zero at position $n + 1 + p$.

This implies that $T = Pivot_{p,0}(T^{aux})$ is an admissible tableaux, i.e., $SimplexSearch(T)$ yields $opt(I^{aux})$ and, if $opt(I^{aux}) = 0$, an admissible starting tableaux for I.

Initialize(I), $I = (m, n, c, z, A, b)$ in Normal Form

1 **if** $b \geq 0$ **then output** $T(I)$
2 **else fix** p, $1 \leq p \leq m$, s.t. $b_p = \min\{b_i, 1 \leq i \leq m\}$
3 $T' = T^{aux}(I)$
4 $T' \leftarrow Pivot_{p,0}(T')$
5 $T' \leftarrow$*SimplexSearch(T')* (* w.r.t. target function $-x_0$ *)
6 **if** $t'_{0,0} \neq 0$
7 **then output infeasible**
8 **else if** $x_0 \in B(T')$ (* as label of row i *)
9 **then** $T' \leftarrow Pivot_{i,j}(T')$ (* for some $t'_{i,j} \neq 0$ *)
11 **else fix** j s.t. x_0 is label of column j
12 $T \leftarrow$ **Delete** columne j and row 0 of T'
13 **output** T

Correctness *Initialize*(I)

One has to show that solving the LP-problem corresponding to T is equivalent to solving I. This is obvious if $b \geq 0$.

If not, after executing line 8, T' is optimal w.r.t. to the secondary basic function, and $x(T')$ defines $(0, \bar{x}_1, \cdots, \bar{x}_n)$ as in Theorem 33.

If $x_0 \in B(T')$, i.e. x_0 occurs as label of a row i, $1 \leq i \leq m$, then $t'_{i,-1} = 0$.

Consequently, the Pivot step in line 9 is degenerate and does not change $x(T')$.

It can be derived straightforwardly that T obtained after 11 is defined by the admissible basic point defined by $(\bar{x}_1, \cdots, \bar{x}_n)$.

Simplex(I), $I = (m, n, A, b, c)$ in Normal Form

```
1  if Initialize(I) =infeasible
2    then output infeasible
3    else T ← Initialize(I)
4         T ← SimplexSearch(T)
5         if T is unbounded
6           then output unbounded
7           else output opt(I) = −t_{0,0}, taken at x_opt
```

where x_{opt} is obtained by the values assigned by $x(T)$ to the variables $x_1, \cdots, x_n$.

Comments

- The Worst Case Running Time of the Simplex Method is exponential in n and m, but the average running time is polynomial.
- There are polynomial time LP-algorithms (e.g., Khachians Ellipsoid Method, Kamarka's Method).
- In practice, the Simplex Method performs well, much better as the Ellipsoid Method, or Kamarka's Algorithm.
- There are very efficient LP-solvers for practice, which use a large number of additional nontrivial techniques.

The Dual Linear Programm

Let $I = (n, m, c, 0, A, b)$ an normal form LP-instance called **primal**,

- **Maximize** $c(x) = \sum_{j=1}^{n} c_j x_j$ subject to
- $\sum_{j=1}^{n} a_{i,j} x_j \leq b_i$ for $i = 1, \cdots, m$
- $x_j \geq 0$ for $j = 1, \cdots, n$.

The corresponding **dual** LP is defined by

- **Minimize** $b(y) = \sum_{i=1}^{m} b_i y_i$ subject to
- $\sum_{i=1}^{m} a_{i,j} y_i \geq c_j$ for $j = 1, \cdots, n$ and
- $y_i \geq 0$ for $i = 1, \cdots, m$.

Theorem 34

Given a primal LP-instance $I = (n, m, A, b, c)$ in normal form, let I_{dual} denote the corresponding dual LP-instance. There are three possibilities:

- *$X(I)$ is empty, then I_{dual} is unbounded.*
- *$X(I_{dual})$ is empty, then I is unbounded.*
- *Both problems are feasible and not unbounded. Then*

$$opt(I) = opt(I_{dual}),$$

which implies that $c(x) \leq b(y)$ for all $x \in X(I)$ and $y \in X(I_{dual})$.

Proof: The proof will follow the following calculations.

Proof of Theorem 34, Weak Duality

Lemma 35

Let $x \in \mathbb{R}^n$ be a valid solution for the primal LP and $y \in \mathbb{R}^m$ be a valid solution for the dual LP. Then $c(x) \leq b(y)$.

Proof: It holds

$$c(x) = \sum_{j=1}^{n} c_j x_j \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{i,j} y_i \right) x_j = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{i,j} x_j \right) y_i \leq \sum_{i=1}^{m} b_i y_i = b(y). \quad \square$$

Corollary 36

Let x be a valid solution for the primal LP and y be a valid solution for the dual LP and let $c(x) = b(y)$.

*Then x is **optimal** for the primal LP and y for the dual LP.* $\square$

Proof of Theorem 34, LP-Duality (1)

Suppose that $opt(I)$ exists and let T denote the last tableaux T produced by the Simplex algorithm. Let

T:=

		x_{j_1}	$\cdots$	x_{j_n}
	z'	c'_{j_1}	$\cdots$	c'_{j_n}
x_{i_1}	b'_{i_1}	a'_{11}	$\cdots$	a'_{1n}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_{i_m}	b'_{i_m}	a'_{m1}	$\cdots$	a'_{mn}

Note that $c'_{j_r} \leq 0$ for all r, $1 \leq r \leq n$, and $b'_{i_s} \geq 0$ for all s, $1 \leq s \leq m$.

Proof of Theorem 34, LP-Duality (2)

Let $\bar{x} \in \mathbb{R}^n$ denote the optimal solution of the primal program. We know that for all $j = 1, \cdots, n$, of the primal program is defined by

$$\bar{x}_j = \begin{cases} b'_j, \text{ if } j \in \{i_1, \cdots, i_m\} \\ 0, \text{ if } j \in \{j_1, \cdots, j_n\} \end{cases}$$

We define a dual point $\bar{y} \in \mathbb{R}^m$. For all $i = 1, \cdots, m$ let

$$\bar{y}_i = \begin{cases} -c'_{n+i}, \text{ if } x_{n+i} \in \{j_1, \cdots, j_n\} \\ 0, \text{ if } x_{n+i} \in \{i_1, \cdots, i_m\} \end{cases}$$

Lemma 37

It holds that $\bar{y}$ is valid for the dual LP, and that $c(\bar{x}) = b(\bar{y}) = -z'$, i.e. $\bar{x}$ and $\bar{y}$ are optimal for the primal and dual LP, resp.

The Proof of Theorem 34 (3)

For $k = 1, \cdots, n+m$ let

$$c'_k = \begin{cases} c'_k, \text{ if } x_k \in N \\ 0, \text{ if } x_k \in B \end{cases}$$

Note that $\bar{y}_i = -c'_{n+i}$ for all $i = 1, \cdots, m$.

For all $x = (x_1, \cdots, x_n) \in \mathbb{R}^n$ it holds

$$c(x) = \sum_{j=1}^{n} c_j x_j = \sum_{k=1}^{n+m} c'_k x_k - z' = \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} c'_{n+i} x_{n+i} - z'$$

$$= \sum_{j=1}^{n} c'_j x_j + \sum_{i=1}^{m} (-\bar{y}_i) \left(b_i - \sum_{j=1}^{n} a_{i,j} x_j \right) - z', \text{ i.e.,}$$

The Proof of Theorem 34 (4)

$$\sum_{j=1}^{n} c_j x_j = \sum_{j=1}^{n} c'_j x_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{i,j} \bar{y}_i \right) x_j - z'$$

$$= \left(-z' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{i,j} \bar{y}_i \right) x_j.$$

As this equality holds for all $x = (x_1, \cdots, x_n) \in \mathbb{R}^n$, it holds

$$-z' - \sum_{i=1}^{m} b_i \bar{y}_i = 0, \text{ and}$$

$$c_j = c'_j + \sum_{i=1}^{m} a_{i,j} \bar{y}_i$$

for all j, $1 \leq j \leq n$.

The Proof of Theorem 34 (5)

The first equality implies that $b(\bar{y}) = c(\bar{x}) = -z'$.

The second equality implies that

$$c_j \leq \sum_{i=1}^{m} a_{i,j}\bar{y}_i$$

as $c'_j \leq 0$ for all j, $1 \leq j \leq n$.

Consequently, $\bar{y}$ is feasible for the dual LP. $\square$

The Maximum Flow Problem

Flow Networks

Definition 38

A flow network $G = (V, E, c)$ refers to a directed graph $G = (V, E)$ and is defined as follows

- There is a distinguished nonempty set $S^* \subseteq V$ called the set of **sources**.
- There is another nonempty distinguished set $T^* \subseteq V$, $S^* \cap T^* = \emptyset$, called the set of **sinks**.
- There is a capacity function $c : E \longrightarrow \mathbb{R}$, where $c(e) > 0$ for all $e \in E$.

Note: Flow networks can be used to model systems of transportation routes (streets, pipelines etc.) for goods.

Note: For $u, v \in V$ let a capacity value $c(u, v) = 0$ be equivalent to $(u, v) \notin E$.

Flows on Flow Networks

Definition 39

A mapping $f : E \longrightarrow \mathbb{R}$ is called a **flow** for $G = (V, E, c)$ if

- for all $e \in E$ it holds $0 \leq f(e) \leq c(e)$, and
- for all $v \in V \setminus (S^* \cup T^*)$ it holds

$$\sum_{u,(u,v)\in E} f(u,v) = \sum_{w,(v,w)\in E} f(v,w).$$

The value

$$|f| = \sum_{s\in S^*} \left(\sum_{w,(s,w)\in E} f(s,w) - \sum_{u,(u,s)\in E} f(u,s) \right)$$

is called **the size** of the flow f.

The Maximum Flow Problem

Definition 40

The **Maximum Flow Problem** is defined as:

- **Input:** $G = (V, E, c)$ flow network with $S^*, T^* \subseteq V$.
- **Output:** A flow on G with maximal value.

The maximum flow problem has many practical applications (transportation problems, pipelining problems etc.). We solve it for **restricted flow networks:**

Definition 41

A flow network $G = (V, E, c)$ is called **restricted** if it contains only one source $s \in V$ and one sink $t \in V$, where $indeg_G(s) = outdeg_G(t) = 0$.

Moreover, G does not contain pairs (u, v), (v, u) of **antiparallel edges**, i.e., $(u, v) \in E$ implies $(v, u) \notin E$.

Efficient Simulation of General Flow Networks by Restricted Ones (1)

Lemma 42

Let $G = (V, E, c)$ be a general flow network with set of sources S^* and set of sinks T^*. Then there is a restricted network $G' = (V', E', c')$ with $|V'| \leq |V| + |E|/2 + 2$, such that each flow on G corresponds to a flow on G' with the same value, and vice versa.

Construction Idea:

- We add a new source s and edges (s, s^*) for all $s^* \in S^*$, where

$$c'(s, s^*) = \sum_{v \in V} c(s^*, v),$$

  which upper bounds the maximal possible flow coming from source s^* in G.

Efficient Simulation of General Flow Networks by Restricted Ones (2)

- We add a new sink t and edges (t^*, t), where

$$c'(t^*, t) = \sum_{v \in V} c(v, t^*),$$

  which upper bounds the maximal possible flow reaching sink t^* in G.

- For all pairs $(v, w), (w, v)$ in E we add a new vertex y and replace one of these edges (say (w, v)) by two new edges $(w, y), (y, v)$ with $c'(w, v_p) = c'(v_p, v) = c(w, v)$.

Claim: Each flow f corresponds to a flow f' on G' (and vice versa) with $|f| = |f'|$, where the flow f' is defined as follows:

Efficient Simulation of General Flow Networks by Restricted Ones (3)

- For all edges $e \in E$ which are also contained in E' it holds $f'(e) = f(e)$.
- For all edges $(w, v) \in E$ which are replaced in E' by two new edges $(w, y), (y, v)$ it holds $f'(w, y) = f'(y, v) = f(w, v)$.
- For all edges $(s, s^*) \in E'$ it holds

$$f'(s, s^*) = \sum_{v \in V, (s^*, v) \in E} f(s^*, v) - \sum_{u \in V, (u, s^*) \in E} f(u, s^*).$$

- For all edges $(t^*, t) \in E'$ it holds

$$f'(t^*, t) = \sum_{v \in V} f(v, t^*) - \sum_{w \in V} f(t^*, w).$$

It can be easily checked that f' is a flow on G' and $|f| = |f'|$. $\square$

MaxFlow Instances as LP-instances

Maximum flow instances $G = (V, E, c)$ can be formulated as LP-instances:

- **Variables:** $f(u, v)$, where $(u, v) \in E$
- **Maximize** $\sum_{w,(s,w)\in E} f(s, w)$ **subject to**

$$0 \leq f(u, v) \leq c(u, v)$$

for all $(u, v) \in E$, and

$$\sum_{u,(u,v)\in E} f(u, v) - \sum_{w,(v,w)\in E} f(v, w) = 0$$

for all $v \in V \setminus \{s, t\}$.

Do there exist MaxFlow-Algorithms better than Simplex ?

Towards the Ford-Fulkerson Method: Residual Networks

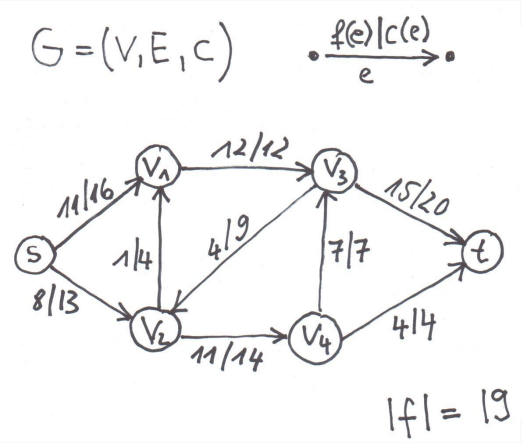
Definition 43

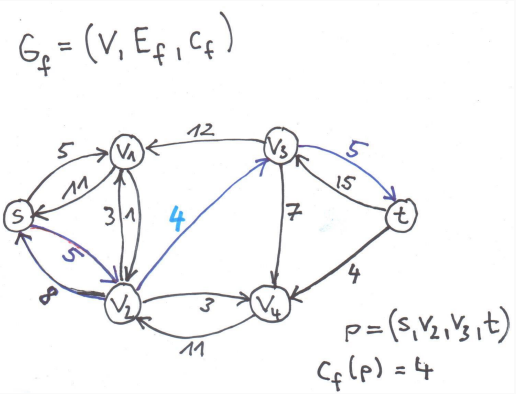
Let $G = (V, E, c)$ be a flow network, and $f : E \longrightarrow \mathbb{R}$ a flow for G. Then the residual flow network $G_f = (V, E_f, c_f)$ is defined as follows.

- For all $(u, v) \in E$ let $c_f(u, v) = c(u, v) - f(u, v)$ and $c_f(v, u) = f(u, v)$.
- For all $(u', v') \in V \times V$ for which $(u', v') \notin E$ and $(v', u') \notin E$ let $c_f(u', v') = 0$.
- Let $E_f = \{(u, v) \in V \times V; c_f(u, v) \neq 0\}$.

Note 1: E_f consists of **forward edges** (u, v), for which $(u, v) \in E$ and $f(u, v) < c(u, v)$, and **backward edges** (v, u), for which $(u, v) \in E$ and $f(u, v) > 0$.

Example


G = (V, E, c)
f(e)|c(e)
e
s
V1
V2
V3
V4
t
11|16
12|12
15|20
1|4
4|9
7|7
8|13
11|14
4|4
|f| = 19


Gf = (V, Ef, cf)
s
V1
V2
V3
V4
t
5
12
5
11
15
3
1
4
7
5
8
3
4
11
p = (s, V2, V3, t)
cf(p) = 4

Basics on Residual Networks

The capacities on G_f give the information how f can be changed for getting a better flow.

The capacity $c_f(u, v)$ of a forward edge $(u, v) \in E_f$ indicates the maximal value by which the flow through $(u, v) \in E$ on G can be increased.

The capacity $c_f(v, u)$ of a backward edge $(v, u) \in E_f$ indicates the maximal value by which the flow through the corresponding edge $(u, v) \in E$ can be decreased.

Definition 44

Each directed path p from s to t in G_f is called an **augmenting path** with respect to G and f. and we denote by

$$c_f(p) = \min\{c_f(e), e \in p\}.$$

the minimal capacity of p.

Special Flows on G_f: Augmenting Paths

Definition 45

Given an augmenting path p on G_f we define a flow $g = g(f, p)$ on G. For all edges $(v, w) \in E$ let

(1) $g(v, w) = f(v, w)$, if $(v, w) \notin p$ and $(w, v) \notin p$,

(2) $g(v, w) = f(v, w) + c_f(p)$, if $(v, w) \in p$, and

(3) $g(v, w) = f(v, w) - c_f(p)$, if $(w, v) \in p$.

Lemma 46

The mapping $g = g(f, f_p)$ defines a flow on G and it holds $|g| = |f| + c_f(p)$.

The Proof of Lemma 46

First we have to show that $g(v, w) \geq 0$ for all $(v, w) \in E$.

The only critical situation occurs if item (3) of Definition 45 is applied.

Let us fix some $(v, w) \in E$ for which $(w, v) \in p$. Then, as $c_f(w, v) \geq c_f(p)$.

$$g(v, w) = f(v, w) - c_f(p) \geq$$

$$f(v, w) - c_f(w, v) = f(v, w) - f(v, w) = 0.$$

We have still to show that g satisfies the balance rule for all nodes $v \in V \setminus \{s, t\}$. There can occur three cases:

(a) g changes two ingoing edges or two outgoing edges of v,

(b) g changes one ingoing and one outgoing edge of v,

(c) The g-values of all ingoing and all outgoing edges of v equal the f-values.

The Proof of Lemma 46, (II)

In case (a) exactly one edge corresponds to a forward and exactly one edge to a backward edge in G_f, i.e., one g value is defined according rule (2) and one according to rule (3).

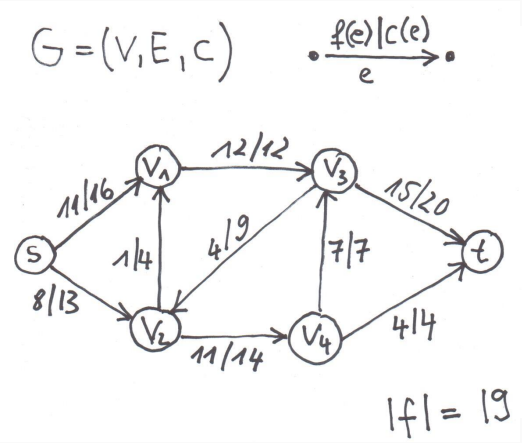
In case (b) either both edges correspond to forward edges and the g values are defined according to rule (2), or both edges correspond to backward edges and the g values are defined according to rule (3).

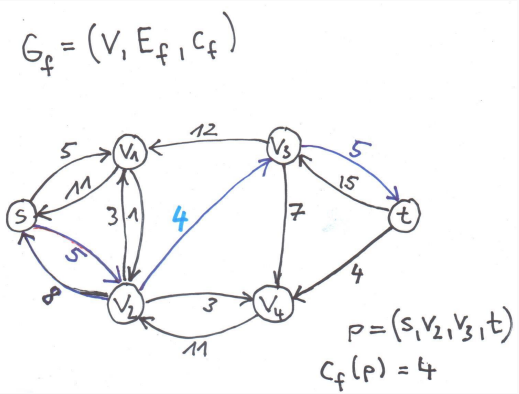
In all cases, the balance rule is preserved in node v.

Finally observe that the first edge along p starting in s has to be a forward edge, as $indeg_G(s) = 0$.

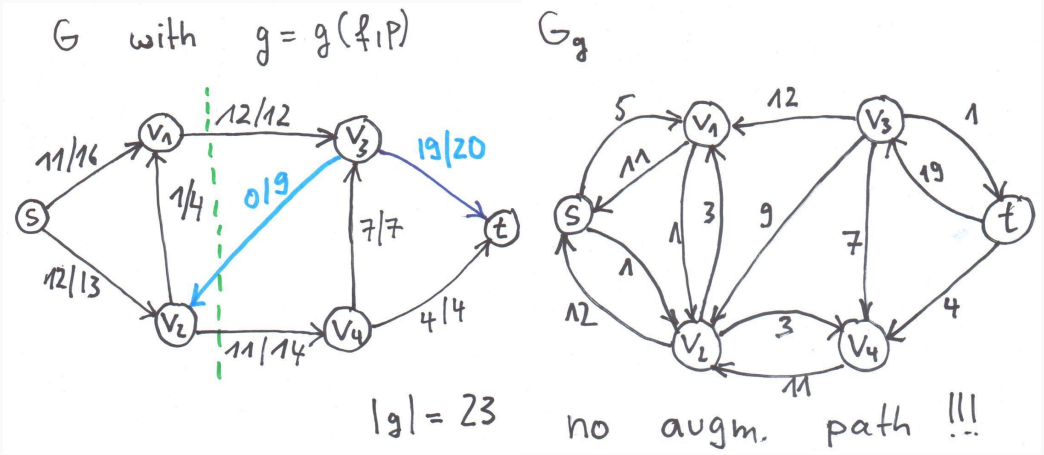
This implies that $|g| = |f| + c_f(p)$. $\square$

Example


G = (V, E, c)
f(e)|c(e)
e
s
V1
V2
V3
V4
t
11/16
12/12
15/20
1/4
4/9
7/7
8/13
11/14
4/4
|f| = 19


Gf = (V, Ef, cf)
s
V1
V2
V3
V4
t
5
12
5
11
15
3
1
4
7
5
8
3
4
11
p = (s, V2, V3, t)
cf(p) = 4

Example


G with g = g(f|P)
s
v1
v2
v3
v4
t
11/16
12/13
1/4
12/12
0/9
19/20
7/7
4/4
11/14
|g| = 23
G_g
5
12
1
11
19
3
1
g
7
1
12
3
4
11
no augm. path !!!

Cuts in G

Definition 47

A partition $V = S \cup T$ is called a cut of G iff $s \in S$ and $t \in T$.

- Let
$$c(S,T) = \sum_{e=(u,v)\in E, u\in S, v\in T} c(e)$$
denote the capacity of the cut (S,T).
- Let f be a flow on G, then
$$f(S,T) = \sum_{e=(u,v)\in E, u\in S, v\in T} f(e) - \sum_{e=(v,u)\in E, v\in T, u\in S} f(e)$$
denotes the flow through the cut (S,T).

Lemma 48

Let f be a flow on G and (S, T) be a cut. Then $|f| = f(S, T)$.

Proof: It holds

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s).$$

Moreover, for all $u \in S \setminus \{s\}$ it holds $\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0$. Consequently,

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S \setminus \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right).$$

Proof of Lemma 48

This implies

$$|f| = \sum_{u \in S} \sum_{v \in V} f(u,v) - \sum_{v \in V} \sum_{u \in S} f(v,u)$$

$$= \sum_{u \in S} \sum_{v \in T} f(u,v) + \sum_{u \in S} \sum_{v \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) - \sum_{v \in S} \sum_{u \in S} f(v,u).$$

Now observe that the second and fourth term are equal. We obtain

$$|f| = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) = f(S,T). \square$$

A Consequence: MinCut bounds MaxFlow

As $|f| = f(S,T) \leq c(S,T)$ applies, we obtain the following lemma.

Lemma 49

For all flows f on $G = (V,E,c)$ and cuts (S,T) of V it holds $|f| \leq c(S,T)$. $\square$

An even stronger result applies:

Theorem 50

Let $G = (V,E,c)$ be a flow network and f a flow on G. Conditions (1) to (3) are equivalent

(1) *The flow f is maximal.*

(2) *There is no path from s to t in G_f.*

(3) *There is a cut (S,T) such that $|f| = c(S,T)$.*

Proof: From (1) follows (2) by Lemma 46, from (3) follows (1) by the above Lemma.

The Proof of Theorem 50 (1)

It remains to show that from (2) follows (3). Suppose that there is no path from s to t in G_f. We construct a cut (S, T) such that $|f| = c(S, T)$.

Let S be the set of all $v \in V$ which are reachable from s in G_f, and $T = V \setminus S$. By definition, $s \in S$ and $t \in T$, i.e., (S, T) is a cut. Remember that

$$|f| = \sum_{e=(u,v) \in E, u \in S, v \in T} f(e) - \sum_{e=(v,u) \in E, v \in T, u \in S} f(e).$$

Consider first an arbitrary edge $e = (u, v) \in E$ for which $u \in S, v \in T$. It must hold that $c_f(e) = 0$, otherwise v would be reachable from s in G_f. This implies $f(e) = c(e)$.

The Proof of Theorem 50 (2)

Consider now an arbitrary edge $e = (v, u) \in E$ for which $v \in T, u \in S$. It must hold that $c_f(u, v) = 0$, otherwise v would be reachable from s in G_f. This implies that $f(v, u) = 0$. Consequently,

$$|f| = \sum_{e=(u,v)\in E, u\in S, v\in T} c(e) - \sum_{e=(v,u)\in E, v\in T, u\in S} 0 = c(S, T). \square$$

The Ford-Fulkerson Method

Input: A Flow Network $G = (V, E, s, t, c)$

Output: A maximal flow $f : E \longrightarrow \mathbb{R}$.

1 Let f be the constant-0 flow on G
2 **Repeat**
3 Fix a path p from s to t in G_f
4 $f \leftarrow g(f, f_p)$
5 **until** t is not reachable from s in G_f
6 **Output** f

If the program terminates, the correctness follows from Theorem 50.

Termination of Ford Fulkerson

Let $G = (V, E, c)$ be a flow network.

- If all capacities are **integral** then Ford-Fulkerson terminates after at most $\sum_{e \in E} c(e)$ iterations. (This is because $|f|$ increased by at least one in each iteration, and as $|f| \leq \sum_{e \in E} c(e)$).
- It is possible to construct flow networks with **irrational** capacities and a stupid way for choosing augmenting pathes such that Ford-Fulkerson never terminates!
- Networks with **rational** capacities can be transformed into equivalent networks with integral capacities (multiply all capacities by the least common multiple (lcm) of all denominators of capacities of G).
- If in each iteration a shortest augmenting path is chosen, then Ford-Fulkerson terminates after $O(|V| \cdot |E|)$ iterations. This corresponds to the **Edmonds-Karp Algorithm**.

Running time of the Edmonds-Karp Algorithm

Definition 51

Given a flow f on $G = (V, E, c)$ and nodes $u, v \in V$, we denote by $\delta_f(u, v)$ the distance of v from u in G_f, i.e. the length (=number of edges) of a shortest path from u to v in G_f.

Now suppose f is a flow on $G = (V, E, c)$ and flow f' is obtained from f by one iteration of Ford-Fulkerson, where the corresponding augmenting path is a shortest path from s to t.

Lemma 52

For all $v \in V$ it holds that $\delta_f(s, v) \leq \delta_{f'}(s, v)$.

Proof of Lemma 52 (1)

Suppose that Lemma 52 is not true. Define

$$W = \{w \in V, \delta_f(s, w) > \delta_{f'}(s, w)\}$$

and suppose correspondingly that $W \neq \emptyset$.

Choose $v \in W$ such that $\delta_{f'}(s, v) = \min\{\delta_{f'}(s, w), w \in W\}$.

Choose a shortest path from s to v in $G_{f'}$ and let u be the predecessor of v along p.

It holds $\delta_{f'}(s, v) = \delta_{f'}(s, u) + 1$, and, as $u \notin W$, $\delta_{f'}(s, u) \geq \delta_f(s, u)$.

It holds $(u, v) \notin E_f$, otherwise it would hold

$$\delta_f(s, v) \leq \delta_f(s, u) + 1 \leq \delta_{f'}(s, u) + 1 = \delta_{f'}(s, v),$$

which would contradict to $v \in W$.

Proof of Lemma 52 (2)

The fact that $(u, v) \notin E_f$ but $(u, v) \in E_{f'}$ implies that f' changes the flow on edge (v, u), i.e. (v, u) belongs to the augmenting path corresponding to the transition from f to f'.

As this path is a shortest path in G_f, we obtain.

$$\delta_f(s, v) = \delta_f(s, u) - 1 \leq \delta_{f'}(s, u) - 1 = \delta_{f'}(s, v) - 2,$$

which contradicts to $v \in W$. $\square$

Correctness Theorem

Definition 53

Given a flow f on $G = (V, E, c)$ and an augmenting path p in G_f. An edge $e = (u, v)$ from p is called **critical**, if it has minimal capacity in p, i.e., $c_f(e) = c_f(p)$.

Lemma 54

During the execution of the Edmonds-Karp Algorithm on a flow network $G = (V, E, c)$, each edge $e \in E$ can become critical at most $|V|/2$ times.

Proof: Consider an arbitrarily fixed iteration of the Edmonds-Karp algorithm, fix the corresponding flow f on G and an corresponding augmenting path p, and suppose that the edge $(u, v) \in E$ becomes critical. Then, after the execution of this iteration, (u, v) is removed from the residual network, but (v, u) remains.

Proof of Lemma 54 (2)

The edge (u, v) can only return to the residual network after a transition in which (v, u) belongs to the augmenting path. Let f' denote the flow in the corresponding iteration, and p' the corresponding augmenting path. As p and p' are shortest paths, we obtain by Lemma 52

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1$$
$$= \delta_f(s, u) + 1 + 1 = \delta_f(s, u) + 2.$$

Now consider the iteration when (u, v) becomes critical the next time and denote by $\tilde{f}$ the corresponding flow. By Lemma 52 we obtain

$$\delta_{\tilde{f}}(s, u) \geq \delta_{f'}(s, u) \geq \delta_f(s, u) + 2.$$

Proof of Lemma 54 (3)

Consequently, for each time (u, v) becomes critical the distance from s to u in the residual network increases by at least 2. As this distance is upperbounded by $|V| - 1$, (u, v) can become critical in at most $|V|/2$ iterations. □

Theorem 55

The Karp-Edmonds Algorithm stops after at most $|E| \cdot (|V|/2)$ iterations. Thus the running time is $O(|V||E|^2)$.

Proof: Using Breadth-First Search in G_f in each iteration for computing a shortest augmenting path from s to t yields running time $O(|E|)$ in each iteration. □

Matchings in Undirected Graphs

The Maximum Matching Problem

Input: An undirected graph $G = (U, E)$.

Output: A **maximal matching** in G, i.e., a **matching** $M \subseteq E$ in G with maximal number of edges.

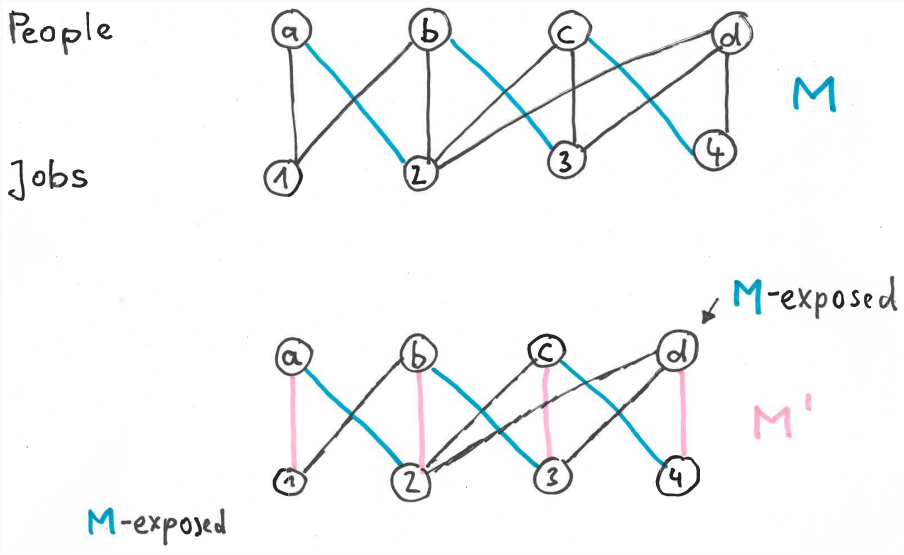
Remember: A graph $G = (V, E)$ is called **undirected** if for all edges $e = (u, v) \in E$ it holds that $(v, u) \in E$. Pairs of antiparallel edges $(u, v), (v, u)$ are identified with the undirected edge $\{u, v\}$.

Definition 56

Let $G = (V, E)$ be an undirected graph. A subset $M \subseteq E$ is called to be a **matching** in G if for all edges $e \neq e' \in M$ it holds that $e \cap e' = \emptyset$.

The Maximum Matching problem has many practical applications.

Example



Characterizing Maximum Matchings by Augmenting Paths

Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ be a matching.

Definition 57

- A node $v \in V$ is called M**-exposed** if no edge of M contains v.
- A simple path p in G is called M**-augmenting**, if its endpoints are M-exposed and, if p has more than one edge, it contains alternatingly edges which are not in M and edges which are in M.

Theorem 58

A matching $M \subseteq E$ is maximal if and only if there is no M-augmenting path.

Proof of the Matching Theorem 58 (I)

Proof: We prove first that the existence of an M-augmenting path p implies that M is not maximal.

Let $p = (w_0, v_1, w_1, \cdots, v_s, w_s, v_{s+1})$, $s \geq 0$, where

- w_0, v_{s+1} are M-exposed,
- the set of edges $M^1 = \{(w_0, v_1), (w_1, v_2), \cdots, (w_{s-1}, v_s), (w_s, v_{s+1})\}$ is disjoint with M and,
- if $s > 0$, then the set of edges $M^2 = \{(v_1, w_1), \cdots, (v_s, w_s)\}$ is a subset of M.

Note that $M' = (M \setminus M^2) \cup M^1$ is a matching with $|M'| = |M| + 1$.

Proof of the Matching Theorem 58 (II)

Now we show that if M is not maximal then there is an M-augmenting path.

Let N be another matching of G with $|N| > |M|$, let $E' = M \Delta N$ consist of all edges contained either in M or in N, and let $G' = (V, E')$.

Note that the connected components of G' are isolated points, or simple paths, or simple circuits, where in the paths and circuits the edges are alternatingly from N or from M.

Consequently, in all circuits, the number of N- and M-edges is the same.

As we have more N- than M-edges, there must be a simple path with one more N-edge. This path defines an M-augmenting path. $\square$

Maximal Matchings in Bipartite Graphs

Definition 59

An undirected graph $G = (U, E)$ is called **bipartite**, if there is a partition $U = V \cup W$ of the node set into two disjoint subsets such that for all edges $e = \{v, w\} \in E$ it holds that $v \in V$ and $w \in W$.

One practical application: The marriage problem

- Given a set V of females and W of males.
- For $v \in V$ and $w \in W$ let $\{v, w\} \in E$ if both v and w do in principle not object to marry each other.
- Problem: Compute a maximal number of possible couples.

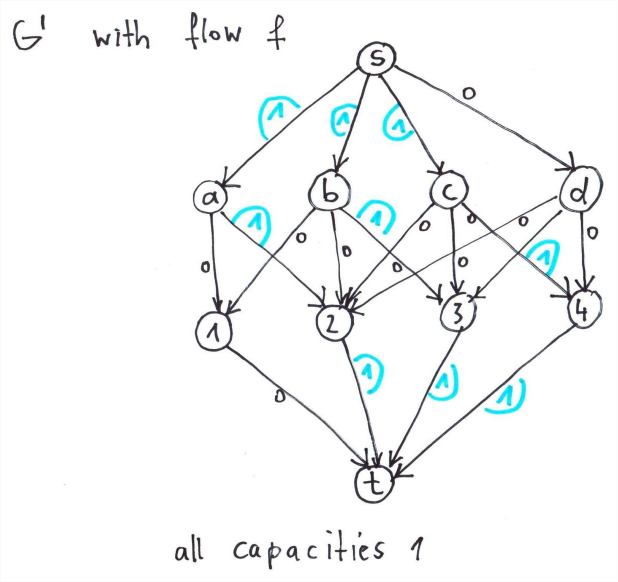
Solving the Maximal Matching Problem for Bipartite Graphs

1.) Transform the bipartite input graph $G = (V, W, E)$ into a flow network $G' = (V', E', c)$, where

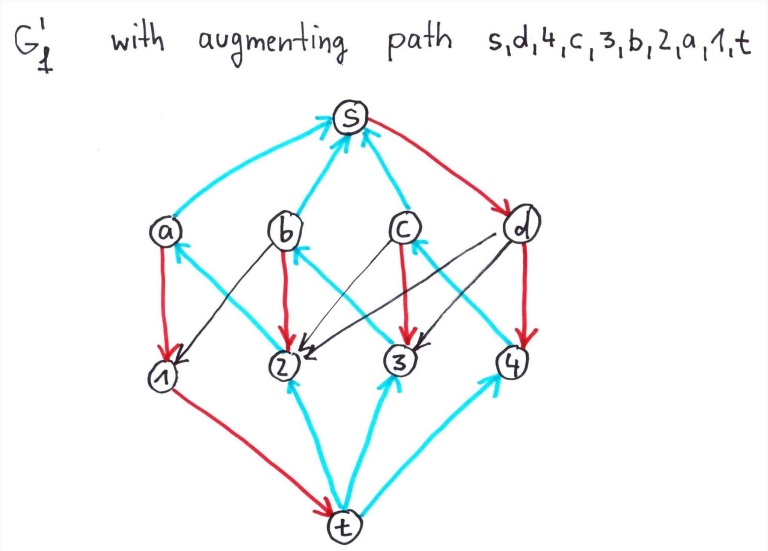
- $V' = V \cup W \cup \{s, t\}$
- $E' = E'_1 \cup E'_2 \cup E'_3$, where
- $E'_1 = \{(s, v), v \in V\}$
- $E'_2 = \{(v, w), v \in V, w \in W, \{v, w\} \in E\}$
- $E'_3 = \{(w, t), w \in W\}$,
- $c(e') = 1$ for all $e' \in E'$.

2.) Compute a maximum flow on G' with Ford-Fulkerson.

Example



Example



Proof of Correctness (1)

We show that

- Each flow computed by Ford-Fulkerson on G' is integral.
- Each integral flow f on G' defines a matching of G with $|f|$ edges, and vice versa.

Lemma 60

If all capacities of a flow network are integral then all flows computed during the application of the Ford-Fulkerson Method are integral, i.e., they assign only integers to edges.

Proof: We start with an integral flow (the constant-0 flow). Then, in each iteration, the flow through each edge remains either unchanged, or it will be increased or decreased by the capacity of the critical edge, which is always integral. □

Proof of Correctness (2)

Lemma 61

Each integral flow f on G' defines a matching of G with $|f|$ edges, and vice versa.

Proof: Let f be an integral flow on G'. As all capacities are 1, for each edge $e' \in E'$ it holds $f(e') \in \{0, 1\}$. Moreover, the indegree of s is 0. Consequently, there is a subset of nodes $\tilde{V} \subseteq V$, $|\tilde{V}| = |f|$, such that $f(s, v) = 1$ for all $v \in \tilde{V}$ and $f(s, v) = 0$ for all $v \notin \tilde{V}$.

Due to the flow maintaining property, for each $v \in \tilde{V}$ there is exactly one node $w(v) \in W$ such that $(v, w(v)) \in E$ and $f(v, w(v)) = 1$. As each node $w \in W$ is left by exactly one edge, it holds $w(v) \neq w(v')$ for all $v \neq v'$ in $\tilde{V}$.

Consequently, the edges $\{\{v, w(v)\}, v \in V'\}$ build a matching of size $|f|$ in G.

Proof of Correctness (3) and Summary

Now let $M = \{\{v_i, w_i\}, i = 1, \cdots, s\}$ be a matching of size s in G. Define a flow f on G' by

$$f(s, v_i) = f(v_i, w_i) = f(w_i, t) = 1$$

for all $i = 1, \cdots, s$, and $f(e') = 0$ for all other edges in E'.

It can be easily checked that f is a flow of size s on G'. $\square$

Summary: A maximal matching in a bipartite graph $G = (V, W, E)$ can be computed in time $O(\min\{|V|, |W|\} \cdot |E|)$ by applying the Ford-Fulkerson Method.

The Structure of residual networks, I

Let f be a non maximal integral flow on G' defining a (nonmaximal) matching $M_f = \{\{v_i, w_i\}, i = 1, \cdots, s\}$ in G.

The residual network G'_f contains edges of the following six types, all with capacity one:

- Forward edges of type (s, v) for some $v \in V$. This implies that $f(s, v) = 0$, i.e., v is M_f-exposed.
- Backward edges of type (v, s) for some $v \in V$. This implies that $f(s, v) = 1$, i.e., in v starts some M_f-edge.
- Forward edges of type (v, w) for some $v \in V$, $w \in W$. This implies that $f(v, w) = 0$, i.e., $(v, w) \notin M_f$.

The Structure of residual networks, II

- Backward edges of type (w, v) for some $v \in V$, $w \in W$. This implies that $f(v, w) = 1$, i.e., $(v, w) \in M_f$.
- Forward edges of type (w, t) for some $w \in W$. This implies that $f(w, t) = 0$, i.e., w is M_f-exposed.
- Backward edges of type (t, w) for some $w \in W$. This implies that $f(w, t) = 1$, i.e., w belongs to an M_f-edge.

The Structure of Augmenting Paths

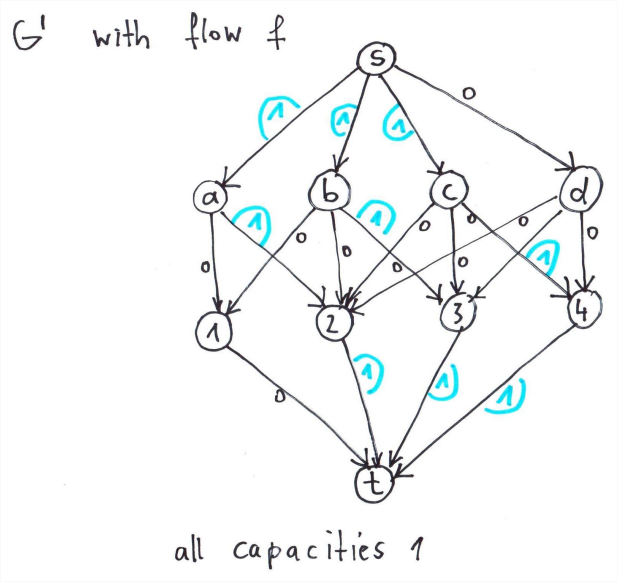
Note that augmenting paths p in G'_f

- start with a forward edge (s, v) with $v \in V$
- followed by a sequence of edges which are alternatingly forward and backward edges, where the first and the last edge are forward edges,
- finished by a forward edge (w, t) with $w \in W$.

As v, w are M_f-exposed, and as the forward and backward edges between v and w correspond to edges not in M_f and in M_f, resp., the subpath of p from v to w form an M_f-augmenting path p' in G.

The improved flow on G' corresponding to f and f_p corresponds to the matching in G improved via p'.

Example



NP-Completeness and -Hardness

Feasible and Unfeasible Problems

Efficiently solvable (feasible) Problems:

- Sorting
- Computation of connected components, minimal spanning trees,
- Arithmetic operations on integers, Primality Testing
- Solving Linear Programs $\cdots$

Appearently Unfeasible Problems:

- SAT: Satisfiability of Boolean formulas in conjunctive normal form (CNF-formulas)
- Traveling Salesman Problem
- Maximum Clique Problem,
- Solving Linear Integer Programs
- Computation of Discrete Logarithms and Integer Factorization

How to prove Non Feasibility?

How to prove that a given problem Π does not have an efficient algorithm?

1.) **Empirical** ... many smart people tried for many years to find an efficient algorithm for Π ... but did not succeed.
Not convincing from a scientific point of view

2.) **Absolute** ... someone found a mathematical proof that the problem does not have an efficient algorithm.
Ideal from a scientific point of view but this seems to be impossible

3.) **Relative** ... One can identify a complexity class *HARD* such that for any problem $\Pi \in$ *HARD* the following holds: The discovery of an efficient algorithm for Π has the drastic consequence that for a huge set of problems, which are assumed to be unfeasible, efficient algorithms can be constructed.

The theory of *NP*-Completeness allows to identify such a complexity class.

Unfeasible Graph Problems I: Maximum Clique

Maximum Clique

- **Input:** Undirected graph $G = (V, E)$
- **Output:** A maximum clique $V' \subseteq V$ of G, i.e., a clique with the maximal number of nodes.

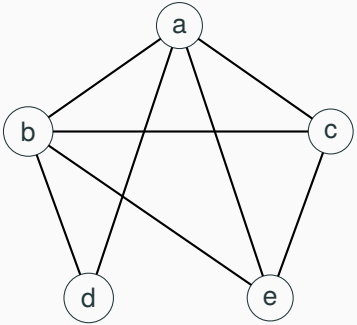
Definition 62

A node set $V' \subseteq V$ is called **clique** (or complete subgraph) of $G = (V, E)$, if for all $v \neq w \in V'$ it holds $\{v, w\} \in E$.

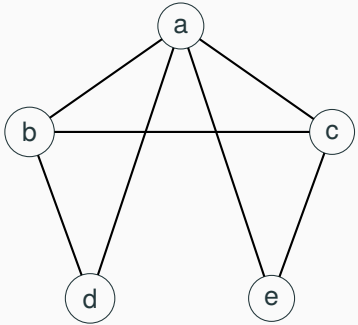
Decisional variant

- **Input:** (G, k), where $k \in \mathbb{N}$
- **Accept** (G, k) iff there is a clique of G with (at least) k nodes.

Example: Cliques



4-Clique
$\{a, b, c, e\}$



3-Cliques, e.g. $\{a, c, e\}$
no 4-Clique

Unfeasible Graph Problems II: Hamiltonian Circuit Problem (HC)

Sir William Rowan Hamilton (1805-1865), irish mathematician and physicist, founder of the Hamiltonian Mechanics, formulated the Hamiltonian Circuit Problem (HC) for dodecahedral graphs.

- **Input:** Undirected graph $G = (V, E)$
- **Accept** if G has a Hamiltonian circuit.

Definition 63

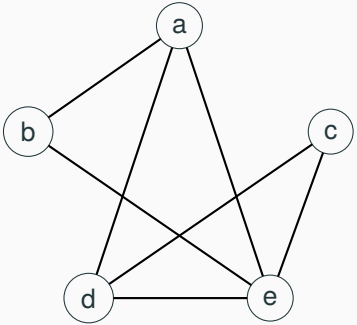
A Hamilton Circuit is a simple circuit in G with $|V|$ edges which visits all nodes $v \in V$.

Observation: Consider the **Euler Circuit Problem**

- **Input:** $G = (V, E)$
- **Accept** if G has an Euler circuit, i.e. a circuit in G which contains all edges.

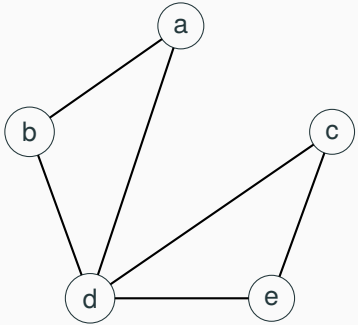
Note: The **Euler Circuit Problem** has an efficient algorithm.

Example: Hamiltonian Circuit



Hamiltonian Circuit

$a \to b \to e \to c \to d \to a$



No Hamiltonian Circuit

Unfeasible Graph Problems III: The Traveling Salesman Problem (TSP)

- **Input:** Weighted Graph $G = (V, E, d)$, $V = \{v_1, \cdots, v_n\}$, encoded as distance matrix $D = (d(v_i, v_j))_{i,j=1}^{n}$.
- **Output:** A shortest round trip

$$v_{\pi(1)} \to v_{\pi(2)} \to \cdots \to v_{\pi(n)} \to v_{\pi(1)}$$

  in G, encoded as permutation $\pi \in \mathcal{S}_n$.
- **Cost of a round trip:**

$$c(D, \pi) = D_{\pi(1),\pi(2)} + \cdots + D_{\pi(n-1),\pi(n)} + D_{\pi(n),\pi(1)}.$$

TSP Example

$$D = \begin{pmatrix} 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 2 & 1 \\ 2 & 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

$$c\left(D, \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 5 & 3 & 4 \end{pmatrix}\right) = D_{1,2} + D_{2,5} + D_{5,3} + D_{3,4} + D_{4,1} = 1 + 1 + 1 + 1 + 1 = 5.$$

$$c\left(D, \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 2 & 3 & 5 \end{pmatrix}\right) = D_{1,4} + D_{4,2} + D_{2,3} + D_{3,5} + D_{5,1} = 1 + 2 + 2 + 1 + 1 = 7.$$

Unfeasible Problems: The Knapsack Problem (KP)

Input

- n objects to be packed into a knapsack, with **weights** $w = (w_1, \cdots, w_n) \in (\mathbb{R}^+)^n$ and **benefit values** $c = (c_1, \cdots, c_n) \in (\mathbb{R}^+)^n$,
- a **weight bound** W.

Output

A subset $I \subseteq \{1, \cdots, n\}$ of objects for which $w(I) = \sum_{i \in I} w_i \leq W$ and

$$c(I) = \sum_{i \in I} c_i = \max\{c(I'); I' \subseteq \{1, \cdots, n\}, w(I') \leq W\}.$$

Example: $n = 10$, $w = (8, 4, 3, 7, 1, 6, 4, 1, 2, 3)$, $c = (7, 2, 2, 7, 2, 4, 3, 1, 3, 2)$, $W = 20$

$I = \{4, 5, 6, 7, 9\}$, $w = (8, 4, 3, 7, 1, 6, 4, 1, 2, 3)$, $c = (7, 2, 2, 7, 2, 4, 3, 1, 3, 2)$,

$w(I) = 20$, $c(I) = 19$.

Is benefit 19 the maximum?

Unfeasible Problems: Partition

Input: A set A of n natural numbers $A = \{a_1, \cdots, a_n\}$.

Output: *Yes* if and only if there is a subset $I \subseteq \{1, \cdots, n\}$ fulfilling

$$\sum_{i \in I} a_i = \sum_{i \notin I} a_i.$$

Example: $n = 11$, $A = \{15, 28, 30, 17, 15, 19, 22, 34, 27, 13, 24\}$

$$\Sigma = 244$$

Solution: $A = \{15, 28, 30, 17, 15, 19, 22, 34, 27, 13, 24\}$

$$\Sigma = \Sigma = 122$$

The Knapsack Problem (KP)

Decisional Variant

- **Input:** (n, w, c, W) as above and a value bound C.
- **Accept** (n, w, c, W, C) iff there is a subset $I \subseteq \{1, \cdots, n\}$ of objects such that $w(I) \leq W$ and $c(I) \geq C$.

Formulation as Integer Linear Program

- **Maximize** $c(x) = \sum_{i=1}^{n} c_i x_i$ subject to
- $\sum_{i=1}^{n} w_i x_i \leq W$, and
- $0 \leq x_i \leq 1$ and $x_i \in \mathbb{Z}$, for all $i = 1, \cdots, n$.

An **Integer Linear Program** is a Linear Program with the additional restriction that all components of the solution have to be **integral**.

Unfeasible Number Theoretic Problems: Discrete Logarithm

Input: An n-bit prime $p \in \mathbb{N}$, an n-bit basis $g \in \mathbb{N}$ for $\mathbb{Z}_p^* = \{1, \cdots, p-1\}$, an n-bit number $y \in \mathbb{N}$.

Note that g basis implies that $\mathbb{Z}_p^* = \{1, g^2, g^3, \cdots, g^{p-2}\}$

Output: The discrete logarithm $dlog_g(y)$ in $\mathbb{Z}_m$, i.e., an n-bit number $x \in \mathbb{N}$ such that

$$g^x \equiv y \mod m.$$

Example

Input: $m = 11, g = 2, y = 9$

$\mathbb{Z}_{11}^* = \{2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8, 2^4 = 5, 2^5 = 10, 2^6 = 9, 2^7 = 7, 2^8 = 3, 2^9 = 6\}$

Output: $dlog_2(9) = 6$ in $\mathbb{Z}_{11}^*$.

Unfeasible Number Theoretic Problems: Factorization

- **Input:** An n-bit number m
- **Output:** A **proper divisor** d of m (if existent), i.e., a natural number d, $2 \leq d \leq m-1$, such that $m = d \cdot \lambda$ for some integer λ.
- **Decisional Variant:** Additional input $b \in \mathbb{N}$, decide if there is a proper divisor d of m fulfilling $d \leq b$.

Remark: The practical unfeasibility of the Discrete Logarithm Problem and the Factorization problem (for input length $n \geq 2048$) is the basic security assumption for many practical cryptographic systems.

The Satisfiability Problem (SAT), Preliminaries (1)

- **Boolean variables** are variables x_i which can take the values 1 (*true*) and 0 (*false*).
- **Boolean formulas** over a set $x_1, \cdots, x_n$ of Boolean variables are recursively defined as follows:
  - Boolean variable x_i and the constant $0, 1$ are Boolean formulas.
  - If F, G are Boolean formulas then also $\neg(F)$, $F \vee G$ and $F \wedge G$.
- **Special Boolean formulas**
  - **Constants** $0, 1$.
  - **Literals** x_i, $\neg x_i$.
  - **Clauses** $L_1 \vee L_2 \vee \cdots \vee L_s$, L_k literals, i.e., $\neg x_1 \vee x_2 \vee \neg x_4$
  - **Conjunctive Normal Form (CNF) Formulas** $C = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, C_j clauses. $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$
  - **Monomials** $L_1 \wedge L_2 \wedge \cdots \wedge L_s$, L_k literals, i.e., $x_1 \wedge \neg x_3 \wedge \neg x_4$.
  - **Disjunctive Normal Form (DNF) Formulas** $D = M_1 \vee M_2 \vee \cdots \vee M_m$, M_j monomials, $(x_1 \wedge x_2) \vee (\neg x_2 \wedge \neg x_3)$

The Satisfiability Problem (SAT), Preliminaries (2)

- **Boolean formulas** $F = F(x_1, \cdots, x_n)$ assign to each $\{0,1\}$-assignment $b = (b_1, \cdots, b_n) \in \{0,1\}^n$ to $x_1, \cdots, x_n$ a function value $F(b) \in \{0,1\}$ via
  - $1(b) = 1$, $0(b) = 0$,
  - $x_i(b) = b_i$,
  - $\neg(F)(b) = 1 - F(b)$,
  - $(F \vee G)(b) = \max\{F(b), G(b)\}$, and
  - $(F \wedge G)(b) = \min\{F(b), G(b)\}$.
- $b = (b_1, \cdots, b_n) \in \{0,1\}^n$ is called a **satisfying assignment** for a formula $F = F(x_1, \cdots, x_n)$ if $F(b) = 1$.
- **Example:** $(0,0,1)$ satisfies $x_1 \vee x_2 \vee x_3$ but not $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$.

The Satisfiability Problem (SAT)

- **Input:** A CNF-formula $C = C_1 \wedge \cdots \wedge C_m$ over $\{x_1, \cdots, x_n\}$.
- **Decide** if C is **satisfiable**, i.e.,

  if there is a **satisfying assignment** $b = (b_1, \cdots, b_n) \in \{0,1\}^n$ which satisfies all clauses C_j, $j = 1, \cdots, m$, of C.

  Note: An assignment $b = (b_1, \cdots, b_n) \in \{0,1\}^n$ satisfies $C = C_1 \wedge \cdots \wedge C_m$ if and only if in each clauses C_j of C at least one literal is satisfied by b.
- **Examples:**
  - $(x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$ is in SAT, satisfied by $(1,1,1)$.
  - $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ is not in SAT

Efficiently Verifiable Proofs

Definition 64

We say that a given decision problem Π has **Efficiently Verifiable Proofs** if there is a proof scheme $(Proof_\Pi, V_\Pi)$ for Π, which is defined as follows

- **Proofs:** $Proof_\Pi$ assigns to each input x for Π a set $Proof_\Pi(x)$ of **possible proofs**.
- **Efficient Verification:** $V_\Pi = V_\Pi(x, y)$ denotes a **decision algorithm** of running time polynomially bounded in $|x|$, which decides for each input x for Π and each possible proof $y \in Proof_\Pi(x)$ if y is a proof for the claim that $\Pi(x) = 1$.
- **Correctness:** It holds that $\Pi(x) = 1$ if and only if there is some $y \in Proof_\pi(x)$ such that $V_\Pi(x, y) = 1$.

A Corresponding Game

Let a decision problem Π have efficiently verifiable proofs $(Proof_\Pi, V_\Pi)$. This corresponds to the following communication game between (potentially ingenious) Alice and Bob. Bob is a normal boy who can only solve problems which are computable in polynomial time.

- **Alice** claims: For this input x it holds $\Pi(x) = 1$!
- **Bob:** I do not believe you, give me a proof !
- **Alice** chooses some possible proof $y \in Proof_\Pi(x)$
- **Bob** verifies - by computing $V_\Pi(x, y)$ - if y is a correct proof for the claim $\Pi(x) = 1$ (and wins if $V_\Pi(x, y) = 0$, otherwise Alice wins) .

Note: The efficiency of V_Π implies that the **bit length of proofs** y **from** $Proof_\Pi(x)$ has to be **polynomially bounded in** $|x|$.

Efficiently Verifiable Proofs for $\Pi = SAT$

- **Input** a CNF-formula $C = C_1 \wedge \cdots \wedge C_m$ over $\{x_1, \cdots, x_n\}$,
- **Proofs** $Proof_\Pi(C)$ contains all possible assignment $b \in \{0,1\}^n$ for the variables $\{x_1, \cdots, x_n\}$.
- **Verification** $V(C, b) = 1$ if and only if $C_j(b) = 1$ for all $j = 1, \cdots, m$.

Corresponds to a game

- **Alice:** This CNF-formula $C = C_1 \wedge \cdots \wedge C_m$ is satisfiable !!
- **Bob:** I do not believe you, give me a proof !
- **Alice** chooses some assignment $b \in \{0,1\}^n$ for the variables in C.
- **Bob** verifies - by computing $C_j(b)$ for all $j = 1, \cdots, m$ - if b is a correct proof for the claim that C is satisfiable (and wins if this is not the case).

Further Examples for Efficiently Verifiable Proofs

- $\Pi = \mathit{Clique}$:
  - **Input** $G = (V, E)$ and a bound $k \leq |V|$.
  - **Proofs** $\mathit{Proof}_\Pi(G, k)$ contains all $V' \subseteq V$, $|V'| = k$.
  - **Verification:** $V_\Pi(G, V') = 1$ if V' is Clique in G.
- $\Pi = \mathit{TSP}$:
  - **Input** A distance matrix $D = (d_{i,j})_{i,j=1}^{n}$ over n cities, a bound d.
  - **Proofs** the set of all round trips $\pi \in \mathcal{S}_n$
  - **Verification:** Accepts π as a proof for $\mathit{TSP}(D, d) = 1$ if the length $c(D, \pi)$ does not exceed d.

Examples for Efficiently Verifiable Proofs (3)

$\Pi = KP$:

- **Input** a KP-instance (n, w, c, W, C) corresponding to n objects.
- **Proofs** a subset of objects $I \subseteq \{1, \cdots, n\}$
- **Verification** Accept I as a correct proof if $w(I) \leq W$ and $c(I) \geq C$.

The Complexity Classes P and NP

Definition 65

- *PTIME* denotes the set of **all problems** which can be computed in polynomial worst-case time.
- P denotes the set of **all decision problems** computable in polynomial worst-case time.
- NP denotes the set of **all decision problems having efficiently verifiable proofs**.

- **Observation 1:** $P \subseteq NP$
- **Observation 2:** *SAT* and the decision variants of *Clique*, *HC*, *TSP*, *KP*, *DiscreteLog*, *Factorization* belong all to NP.
- **A Fundamental Problem:** $P = NP$? (Millions of Dollars for a solution, e.g., from the New York Times.)

Basic Properties of Efficiently Verifiable Proofs

- **Important:** Efficiently Verifiable Proofs $(Proof_\Pi, V_\Pi)$ for a decision problem Π **do not yield** an efficient algorithms for Π.

  For that reason we would need an efficient algorithm for finding correct proofs y for $\Pi(x) = 1$ in $Proof_\Pi(x)$. This is problematic as the size of $Proof_\Pi(x)$ is exponentially in $|x|$.

- **However:** Efficiently Verifiable Proofs $(Proof_\Pi, V_\Pi)$ for Π yield an **Exhaustive Search Algorithm** for Π:
  - **Given** an input x for Π
  - **For all** proof candidates $y \in Proof_\Pi(x)$
  - **do** If $V_\Pi(x, y) = 1$ **then output** $\Pi(x) = 1$, **stop**
  - **output** $\Pi(x) = 0$

  Running time exponentially in $|x|$.

Background PNP-Problem

The problem if $P \neq NP$ or $P = NP$ concern a fundamental question in artificial intelligence: Is it really more complicated to find a mathematical proof for a given theorem than to check if a given proof for this theorem is correct?

$P \neq NP$ implies that finding a proof is more complicated than verifiying correctness.

$P = NP$ implies that the creative process of finding a proof (or making arts or making some other creative things) can be in principle efficiently simulated by a computer.

Polynomial Reductions

We identify decision problems $\Pi : \{0,1\}^* \longrightarrow \{0,1\}$ with languages $L = L_\Pi \subseteq \{0,1\}^*$ via $x \in L \iff \Pi(x) = 1$.

Definition 66

Let $L, L' \subseteq \{0,1\}^*$ be languages. A function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ is called **polynomial reduction from L to L'** iff

- $f \in PTIME$, and
- for all $x \in \{0,1\}^*$ it holds that $x \in L$ if and only if $f(x) \in L'$.

We say that L is polynomially reducible to L' (abbreviation $L \leq_{pol} L'$) if there is a polynomial reduction from L to L'.

Informal Explanation

Suppose that we know an efficient algorithm A' for a decision problem Π', corresponding to a language $L' \subseteq \{0,1\}^*$.

Let Π be another decision problem, corresponding to a language $L \subseteq \{0,1\}^*$, for which we are looking for a good algorithm.

Suppose we know another efficient algorithm A_f, computing a polynomial reduction from L to L', i.e., A_f transforms inputs x for L into inputs x' for L' such that $\Pi(x) = \Pi'(f(x))$.

Then

- Run A_f on input x for computing $f(x)$
- Run A' on $f(x)$ for computing $\Pi(x) = \Pi'(f(x))$.

is an efficient algorithm A for Π. Consequently,

Lemma 67

If $L' \in P$ and $L \leq_{pol} L'$ then also $L \in P$. $\square$.

One Example: $HC \leq_{pol} TSP$

Let $G = (V, E)$ be an undirected graph, let $V = \{v_1, \cdots, v_n\}$.

We construct a distance matrix $D = (d_{i,j})_{i,j=1}^{n}$ such that

$$G \in HC \iff (D, n) \in TSP.$$

i.e., there is a round trip of length $\leq n$ w.r.t. D if and only if G has a Hamiltonian circuit. Let

$$d_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 2, & \text{if } (i,j) \notin E \end{cases}$$

Note that each each Hamiltonian circuit in G induces a round trip of length n in D.

On the other hand, if G does not contain a HC then each roundtrip in D contains one transition $i \to j$ for which $(i,j) \notin E$, i.e., the length of the round trip is at least $n - 1 + 2 = n + 1$.

NP-Completeness

Definition 68

- A language $L \in NP$ is called NP-complete, if for all $L' \in NP$ it holds $L' \leq_{pol} L$, i.e., all languages in NP are polynomially reducible to L.
- NPC denotes the class of NP-complete problems.

Theorem 69

If $P \neq NP$ then $P \cap NPC = \emptyset$, i.e., NP-complete problems do not have polynomial time algorithms.

Proof: Let $L \in P \cap NPC$. Then $L' \leq_{pol} L$ for all $L' \in NP$. But this implies that $L' \in P$ for all $L' \in NP$, as P is closed w.r.t. polynomial reductions. This implies $NP = P$. $\square$

Important Question: Do NP-complete problems exist?

Cook's Theorem

Theorem 70

*(Cook 1971) SAT is NP-complete (Stephen A. Cook (*1939), Uni Toronto).*

The Proof: Let $L \in NP$ be arbitrarily fixed. We construct for each input $x \in \{0,1\}^*$ a CNF-formula C^x of polynomial size such that $x \in L \iff C^x \in SAT$.

The only thing we know about L is that there are polynomially bounded functions $p = p(n)$ and $t = t(n)$ and a verification algorithm V_L such that

- For all $x \in \{0,1\}^*$ and proofs $y \in \{0,1\}^{p(|x|)}$ V_L stops at input (x,y) after exactly $t(|x|)$ steps with output $V_L(x,y) \in \{0,1\}$.
- For all $x \in \{0,1\}^*$ it holds that $x \in L$ if and only if there is a proof $y \in \{0,1\}^{p(|x|)}$ such $V_L(x,y) = 1$.

We assume that proofs $y \in Proof_L(x)$ are encoded as $\{0,1\}^{p(|x|)}$-strings and that V_L is equipped with a clock ensuring the same running time for inputs of equal length.

Proof of Cook's Theorem: Turing Machines

For the proof we use the **Extended Hypothesis of Church**[1], which implies that each polynomial time algorithm can be executed on a polynomial time bounded **One-tape Turing machine** (for short 1-TM)[2], a very simple type of a formal computational device:

- A 1-TM M has **one linear tape** consisting of a (potentially unbounded) number of register cells, where in each register cell characters from $\{0, 1, \#\}$ can be stored.
- At the tape a **head** is operating which is connected with a CPU.
- M is working clockwise on the basis of a TM-program, called **state transition function**. In each clock cycle, the machine is in a certain inner state q, and the head is at and reads a character b from a certain register cell. Depending on q and b, the head writes a new letter into this cell, moves to the left or to the right neighbor cell, and changes the inner state.

[1] Alonzo Church (1903-1995), US-mathematician, logician and philosopher.
[2] Alan Turing (1912-1954), English Pioneer of Modern Computer Science

Proof of Cook's Theorem: Formal TM Definition

Definition 71

A 1-TM is a triple $M = (Q, q_0, \delta)$, where Q is a finite set of states including the initial state q_0, and

$$\delta : Q \times \{0, 1, \#\} \longrightarrow \{0, 1, \#\} \times MOVE \times Q$$

is the state transition function, where $MOVE = \{L, R, N\}$ consists of the commands L (move the head to the left), R, (move the head to the left), and N (do not move).

- δ**-Instances** $(q, b; b', m, q')$ correspond to **TM-commands** of type: **If** M is in state q and reads b **then** write b', **move** the head according to m and **change** into state q'.
- At the beginning of an computation, M is in initial state q_0 and the head is at some predefinded initial position.
- Instances $(q, b; b, N, q)$ for $q \in Q$ and $b \in \{0, 1, \#\}$ are called **stop instances**.
- States $q \in Q$ for which $(q, b; b, N, q) \in \delta$ for all $b \in \{0, 1, \#\}$ are called **stop states**.

1-TM Example: Incrementing a Binary Number

- We define a 1-TM $M = (\{q_0, q_1, q_2\}, q_0, \delta)$ which increments a given binary number $x = (x_{n-1}, \cdots, x_0)$ by 1, where

$$\delta = \{(q_0, 1; 0, L, q_0), (q_0, 0; 1, R, q_1), (q_0, \#; 1, R, q_1), (q_1, 0; 0, R, q_1),$$
$$(q_1, 1; 1, R, q_1), (q_1, \#; \#, L, q_2), (q_2, 0; 0, N, q_2), (q_2, 1; 1, N, q_2), (q_2, \#; \#, N, q_2)\}.$$

- At the initial configuration, the inscription of the tape is

$$\cdots \#\#\# x_{n-1} x_{n-1} \cdots x_1 x_0 \#\#\# \cdots$$

  and the head is on the cell with x_0 and M is in state q_0.
- Then, under q_0, while reading characters 1 the M-head overwrites these characters 1 by 0 and goes left until it finds the rightmost character 0 or the left delimiter $\#$, overwrites this character by 1 and changes to q_1.
- Under q_1, while reading characters 0, the M-head goes right until it finds the right delimiter $\#$, there it moves right, changes to state q_2 and stops as no further command is applicable.

1-TM Configurations and Computations

Definition 72

- An M**-configuration** $K = (q, w, i)$ of a 1-TM $M = (Q, q_0, \delta)$ describes the overall state of M at a given time. It consists of the current state q, the current tape inscription $w \in \{0, 1, \#\}^*$ and the current head position $i \in \mathbb{Z}$.
- An δ-instance $I = (q, b; b', m, q') \in \delta$ is **applicable** to an M-configuration $K = (\tilde{q}, w, i)$ if $q = \tilde{q}$ and $w_i = b$.
- Note that for each M-configuration K **exactly one** δ-instance I is applicable to K. $I(K)$ denotes the **successor configuration** of K obtained by executing command I on K (denotation $K \xrightarrow{M} I(K)$).
- K is called **stop configuration** if the instance which is applicable to K is a stop instance.
- The sequence $K \xrightarrow[M]{} K_1 \xrightarrow[M]{} K_2 \xrightarrow[M]{} \cdots$ is called **computation** of M on K, which is called **halting** if it leads to a stop configuration.

Example Computation M on K

We consider the 1-TM $M = (\{q_0, q_1, q_2\}, q_0, \delta)$ incrementing a given binary number. Let the start configuration K on input number 1010111 be

$$\#\#101011q_01\#\#$$

Here, head position and current state are encoded by inserting into the current tape inscription the state as right neighbor of the register cell at which the head is. The corresponding computation of M on K is

$$\#\#101011q_01\#\# \xrightarrow{M} \#\#10101q_010\#\# \xrightarrow{M} \#\#1010q_0100\#\#$$
$$\xrightarrow{M} \#\#101q_00000\#\# \xrightarrow{M} \#\#1011q_1000\#\# \xrightarrow{M} \#\#10110q_100\#\#$$
$$\xrightarrow{M} \#\#101100q_10\#\# \xrightarrow{M} \#\#1011000q_1\#\# \xrightarrow{M} \#\#101100q_20\#\#$$

which implies $time_M(K) = 8$.

Back to the Proof of Cook's Theorem

To Do: We have to define a polynomial reduction that assigns to each $x \in \{0,1\}^*$ a CNF-Formula C_x of polynomial size in $|x|$ such that

$$x \in L \iff C_x \in SAT.$$

We know that there are polynomially bounded functions $P = P(n)$ and $T = T(n)$ and a 1-TM $M = (Q, q_1, \delta)$ with the following properties:

- Let $Q = \{q_1, \cdots, q_{s-1}, q_s\}$, where q_{s-1} (output $0 =$ *reject*) or q_s (output $1 =$ *accept*) are the only stopping states of M.
- For all $n \in \mathbb{N}$, $x \in \{0,1\}^n$ and possible proofs $y \in \{0,1\}^{P(n)}$ on the starting configuration $K_0(x,y) = \cdots \# q_1 x_1 \cdots x_n \# y_1 \cdots y_{P(n)} \# \cdots$, M stops after $T(n)$ steps.
- For all $x \in \{0,1\}^*$ it holds that $x \in L$ if and only there is a proof $y \in \{0,1\}^{P(n)}$ and a sequence of M-configurations $K_1, \cdots, K_{T(n)}$ such that

  $$K_0(x,y) \xrightarrow{M} K_1 \xrightarrow{M} \cdots \xrightarrow{M} K_{T(n)}$$

  and $K_{T(n)}$ is a stop configuration with the accepting state q_s.

Encoding Configurations

- We fix $n \in \mathbb{N}$ and $x \in \{0,1\}^n$ and set $P = P(n)$ and $T = t(n)$. Note that $T \geq P + n$.
- For constructing C_x we encode sequences of M-configurations $K_0, \cdots, K_T$ as $\{0,1\}$-assignments of a set $H \cup R \cup Q \cup Y$ of Boolean variables, where the variables in the set $H = \bigcup_{t=0}^{T} H^t$ encode the tape inscriptions in the steps $t = 0, \cdots, T$, the variables in $R = \bigcup_{t=0}^{T} R^t$ encode the head positions in the steps $t = 0, \cdots, T$, the variables in $Q = \bigcup_{t=0}^{T} Q^t$ encode the inner states in the steps $t = 0, \cdots, T$, and the variables $Y = \{Y_1, \cdots, Y_P\}$ encode the proofs.
- As for all proofs $y \in \{0,1\}^P$ M runs T steps on $K_0(x,y)$, the leftmost head position is $-T+1$ and the rightmost head position is $T+1$, i.e., $R^t = \{R^t_{-T+1}, \cdots, R^t_{T+1}\}$, where $R^t_j = 1$ means that in step $t \in \{0, \cdots, T\}$ the M-head is on position $j \in \{-T+1, \cdots, T+1\}$.
- Further $H^t = \bigcup_{j=-T+1}^{T+1} \{H^t_{j,0}, H^t_{j,1}, H^t_{j,\#}\}$, where $H^t_{j,b} = 1$ means that in step $t \in \{0, \cdots, T\}$ the character stored at tape position j is $b \in \{0,1,\#\}$.
- $Q^t = \{Q^t_1, \cdots, Q^t_s\}$, where $Q^t_r = 1$ means that in step $t \in \{0, \cdots, T\}$ the machine M is in state q_r, $r \in \{1, \ldots, s\}$.

The Structure of $C_x = C_x(H, R, Q, Y)$

C_x has to be constructed in such a way that an assignment to $H \cup R \cup Q \cup Y$ satisfies C_x if and only if it encodes an accepting computation of M on $K_0(x, y)$ for some proof $y \in \{0,1\}^P$. C_x is defined over $O(T + P)$ variables and has the following structure.

$$C_x = \bigwedge_{t=0}^{T} C_x^{1,t} \wedge C_x^2 \wedge \bigwedge_{t=1}^{T} C_x^{3,t} \wedge Q_s^T, \text{ where}$$

- for all $t = 0, \cdots, T$ an assignment to $H^t \cup R^t \cup Q^t$ satisfies $C_x^1(H^t, R^t, Q^t)$ if and only if it is an M-Conf assignment in the sense that it correctly encodes an M-configuration K,
- an M-Conf assignment K to $H^0 \cup R^0 \cup Q^0$ and an assignment y to Y satisfy $C_x^2(H^0, R^0, Q^0, Y)$ if and only if $K = K_0(x, y)$.
- for all $t = 1, \cdots, T$, M-Conf assignments K to $H^{t-1} \cup H^t \cup R^{t-1}$ and K' to $R^t \cup Q^{t-1} \cup Q^t$ satisfy $C_x^{3,t}(H^{t-1}, H^t, R^{t-1}, R^t, Q^{t-1}, Q^t)$ if and only if $K \xrightarrow{M} K'$.

The Construction of $C_x^{1,t}$

Note that an assignment to $H^t \cup R^t \cup Q^t$ is an M-conf assignment if and only if it satisfies **exactly one R^t variable**, and **exactly one Q^t variable**, and, for all $j = -T+1, \cdots, T+1$, **exactly one H_j^t variable**.

We use the fact that the special CNF-formula

$$F(z_1, \cdots, z_m) = \left(\bigvee_{i=1}^{m} z_i \right) \wedge \bigwedge_{1 \le i \ne j \le m} (\neg z_i \vee \neg z_j),$$

with $1 + (m-1)m = O(m^2)$ clauses is satisfied if and only if exactly one z-variable is satisfied.

Consequently, an assignment to $H^t \cup R^t \cup Q^t$ satisfies

$$C_x^{1,t} = F(R^t_{-T+1}, \cdots, R^t_{T+1}) \wedge F(Q^t_1, \cdots, Q^t_s) \wedge \bigwedge_{j=-T+1}^{T+1} F\left(H^t_{j,\#}, H^t_{j,0}, H^t_{j,1}\right)$$

if and only if it is an M-conf assignment. Note that $C_x^{1,t}$ has $O(T^2)$ clauses.

The Construction of C_x^2

We have to construct $C_x^2(H^0, R^0, Q^0, Y)$ in such a way that an M-conf assignment K to $H^0 \cup R^0 \cup Q^0$ and an assignment y to Y satisfies $C_x^2(H^0, R^0, Q^0, Y)$ if K coincides at each tape position with $K_0(x, y)$.

This implies

$$C_2^x = \bigwedge_{j=-T+1}^{0} H_{j,\#}^0 \wedge \bigwedge_{j=1}^{n} H_{j,x_j}^0 \wedge H_{n+1,\#}^0 \wedge \bigwedge_{j=1}^{P} \left((\neg Y_j \to H_{n+1+j,0}^0) \wedge (Y_j \to H_{n+1+j,1}^0) \right) \wedge \bigwedge_{j=n+P+2}^{T+1} H_{j,\#}^0.$$

Note that the logical implication $x \to y$ is false if and only if $x = 1$ and $y = 0$, which implies

$$x \to y \equiv \neg x \vee y.$$

Note further that C_x^2 has $O(T)$ clauses.

The Construction $C_x^{3,t}$, $t = 1, \cdots, T$

We have to construct $C_x^{3,t}(H^{t-1}, H^t, R^{t-1}, R^t, Q^{t-1}, Q^t)$ in such a way that M-conf assignments K to $H^{t-1} \cup R^{t-1} \cup Q^{t-1}$ and K' to $H^t \cup R^t \cup Q^t$ satisfy $C_x^{3,t}$ if and only if $K \xrightarrow{M} K'$.

We assume that $\delta = \{(q_r, b; b_{r,b}, m_{r,b}, q_{next(r,b)}); r = 1, \cdots, s, b \in \{0, 1, \#\}\}$. Then

$$C_x^{3,t} = \bigwedge_{j=-T+1}^{T+1} \bigwedge_{r=1}^{s} \bigwedge_{b \in \{0,1,\#\}} \left(\left(Q_r^{t-1} \wedge R_j^{t-1} \wedge H_{j,b}^{t-1} \right) \longrightarrow \left(H_{j,b_{q,b}}^t \wedge R_{m_{q,b}(j)}^t \wedge Q_{next(r,b)}^t \right) \right) \wedge$$

$$\wedge \bigwedge_{j=-T+1}^{T+1} \bigwedge_{b \in \{0,1,\#\}} \left(\left(\neg R_j^{t-1} \wedge H_{j,b}^{t-1} \right) \longrightarrow H_{j,b}^t \right).$$

Hereby, $m(j) = j - 1$ for $m = L$, $m(j) = j + 1$ for $m = R$, and $m(j) = j$ für $m = N$.

Note that $C_x^{3,t}$ has $O(T)$ clauses.

Comments to the Construction of $C_x^{3,t}$

Note that the effect of the formula

$$(z_1 \wedge z_2 \wedge z_3) \longrightarrow (z_4 \wedge z_5 \wedge z_6)$$

is that each satisfying assignment which satisfies z_1 and z_2 and z_3 has to satisfy also z_4 and z_5 and z_6.

Note further that

$$(z_1 \wedge z_2 \wedge z_3) \longrightarrow (z_4 \wedge z_5 \wedge z_6) \equiv$$
$$(\neg z_1 \vee \neg z_2 \vee \neg z_3 \vee z_4) \wedge (\neg z_1 \vee \neg z_2 \vee \neg z_3 \vee z_5) \wedge (\neg z_1 \vee \neg z_2 \vee \neg z_3 \vee z_6)\,.$$

Summary

We defined an CNF-formula C_x defined over $O(T)$ and with $O(T^3)$ clauses. We showed that C_x can be satisfied only by encodings of an accepting computation of M on input $x\#y$ for some proof $y \in \{0,1\}^P$. $\square$

The Reduction Method for Showing *NP*-Completeness

Theorem 73

We want to show that a given NP-Problem L is NP-complete.

For that it is sufficient to ***find an appropriate*** *NP****-complete problem*** *L' (for example, SAT), and to* ***construct a polynomial reduction*** *g* ***from*** *L'* ***to*** *L.*

Proof

Let $g : \{0,1\}^* \longrightarrow \{0,1\}^*$ denote a polynomial reduction from L' to L.

For showing that L is *NP*-complete it is sufficient to fix an arbitrary problem $\tilde{L} \in NP$ and construct a polynomial reduction from $\tilde{L}$ to L.

As L' is *NP*-complete there is a polynomial reduction $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ from $\tilde{L}$ to L', i.e., for all $x \in \{0,1\}^*$ it holds that

$$x \in \tilde{L} \Longleftrightarrow f(x) \in L' \Longleftrightarrow g(f(x)) \in L.$$

Consequently, $\tilde{L} \leq_{pol} L$, as $g \circ f$ defines a polynomial reduction from $\tilde{L}$ to L. $\square$

A Polynomial Reduction from SAT to $3SAT$, I

Theorem 74

$3SAT \in NPC$.

Proof: We construct a polynomial reduction from SAT to $3SAT$ which assigns for each $n \in \mathbb{N}$ and each CNF-formula $C = C_1 \wedge \cdots \wedge C_m$ over $X_n = \{x_1, \cdots, x_n\}$ the 3-CNF-Formula $D = D_1 \wedge \cdots \wedge D_m$ over $X \cup Y^1 \cup \cdots \cup Y^m$ with

$$C \in SAT \Longleftrightarrow D \in 3SAT. \tag{1}$$

Here, for all $j = 1, \cdots, m$, D^j is a $3CNF$-formula over the variables $X_n \cup Y^j$, where Y^j is a set of special helping variables which occur only in D_j.

If $C_j = L_1^j \vee \cdots \vee L_{s_j}^j$ and $s_j \leq 3$ then $C_j = D_j$. Otherwise $Y^j = \{y_1^j, \cdots, y_{s_j-3}^j\}$ and

$$D_j = (L_1^j \vee L_2^j \vee y_1^j) \wedge (\neg y_1^j \vee L_3^j \vee y_2^j) \wedge (\neg y_2^j \vee L_4^j \vee y_3^j) \wedge \cdots$$
$$\cdots \wedge (\neg y_{s_j-4}^j \vee L_{s_j-2}^j \vee y_{s_j-3}^j) \wedge (\neg y_{s_j-3}^j \vee L_{s_j-1}^j \vee L_{s_j}^j).$$

The Polynomial Reduction from SAT to $3SAT$, II

Lemma 75

For all $\{0,1\}$-assignments b of X_n it holds $C_j(b) = 1$ if and only if there is a $\{0,1\}$-assignment c of Y^j with $D_j(b,c) = 1$.

Proving Relation 1 with Lemma 75:

$C \in SAT \iff$ There is an assignment b to X_n such that $C_j(b) = 1$ for all $j = 1, \cdots, m$

$$\iff$$

There is an assignment b to X_n such that for all $j = 1, \cdots, m$ there is an assignment c_j to Y^j such that $D_j(b, c_j) = 1$ (this is true by Lemma 75).

$$\iff$$

$D \in 3SAT$ as $(b, c_1, \cdots, c_m)$ satisfies D. $\square$

The Proof of Lemma 75

Assume that $C_j = L^j_1 \vee \cdots \vee L^j_s$ and $s \geq 4$ (for $s \leq 3$, the proof of Lemma 75 is obvious).

The **Proof Lemma 75** consists of two observations

Observation 1: Fix an assignment b to X_n such that $L^j_k(b) = 0$ for all $k = 1, \cdots, s_j$. Then

$$D_j(b, \cdot) = (0 \vee 0 \vee y^j_1)(\neg y^j_1 \vee 0 \vee y^j_2) \wedge \cdots \wedge (\neg y^j_{s_j-4} \vee 0 \vee y^j_{s_j-3}) \wedge (\neg y^j_{s_j-3} \vee 0 \vee 0)$$

$$= y^j_1 \wedge (\neg y^j_1 \vee y^j_2) \wedge \cdots \wedge (\neg y^j_{s_j-4} \vee y^j_{s_j-3}) \wedge \neg y^j_{s_j-3}.$$

This formula **can not be satisfied**, as the first clause forces that $y^j_1 \leftarrow 1$, which, by the second clause, forces that $y^j_2 \leftarrow 1$, and so on, where the penultimate clause forces that $y^j_{s_j-3} \leftarrow 1$, but the last clause forces $y^j_{s_j-3} \leftarrow 0$.

Further in the Proof of Lemma 75

Observation 2: Fix assignment b to X_n such that $L_k^j(b) = 1$ for **exactly one** $k = 1, \cdots, s_j$.

Then $D_j(b, \cdot)$ is obtained from $y_1^j \wedge (\neg y_1^j \vee y_2^j) \wedge \cdots \wedge (\neg y_{s_j-4}^j \vee y_{s_j-3}^j) \wedge \neg y_{s_j-3}^j$ by deleting the clause which contained L_k^j.

We show that for all $k = 1, \cdots, s_j$ the corresponding formula $D_j(b, \cdot)$ is satisfiable.

- **Case** $k = 1, 2$: Then $D_j(b, \cdot) = (\neg y_1^j \vee y_2^j) \wedge \cdots \wedge (\neg y_{s_j-4}^j \vee y_{s_j-3}^j) \wedge \neg y_{s_j-3}^j$
  This is satisfied by $y_1^j = y_2^j = \cdots = y_{s_j-4}^j = y_{s_j-3}^j \leftarrow 0$.
- **Case** $k = s - 1, s$: Then $D_j(b, \cdot) = y_1^j \wedge (\neg y_1^j \vee y_2^j) \wedge \cdots \wedge (\neg y_{s_j-4}^j \vee y_{s_j-3}^j)$
  This is satisfied by $y_1^j = y_2^j = \cdots = y_{s_j-4}^j = y_{s_j-3}^j \leftarrow 1$.
- **Case** $k \geq 5$, $3 \leq k \leq s - 2$: L_k^j is in clause $(\neg y_{k-2}^j \vee L_k^j \vee y_{k-1}^j)$ and $D_j(b, \cdot) =$
  $y_1^j \wedge (\neg y_1^j \vee y_2^j) \wedge \cdots \wedge (\neg y_{k-3}^j \vee y_{k-2}^j) \wedge (\neg y_{k-1}^j \vee y_k^j) \wedge \cdots \wedge (\neg y_{s_j-4}^j \vee y_{s_j-3}^j) \wedge \neg y_{s_j-3}^j$.
  This is satisfied by $y_1^j = y_2^j = \cdots = y_{k-2}^j \leftarrow 1$ and $y_{k-1}^j = \cdots = y_{s_j-3}^j \leftarrow 0$.

Concluding the Proof of Lemma 75

Observation 3: Fix an assignment b to X_n such that $L_k^j(b) = 1$ for **more than one** literal L_k^j, $k \in \{1, \cdots, s_j\}$.

Then $D_j(b, \cdot)$ is obtained by deleting one clause or more clauses from one of the satisfiable formulas in the three cases of Observation 2.

Note that any CNF-formula C which is obtained by deleting clauses from a satisfiable formula C' is satisfiable too, as any satisfying assignment for C' satisfies C.

Consequently, $D_j(b, \cdot)$ is satisfiable. $\square$

An Illustrating Examples

- The clause $C_j = x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4$ defines
$$D_j = (x_1 \vee \neg x_2 \vee y_1^j) \wedge (\neg y_1^j \vee \neg x_3 \vee x_4).$$
- Consider assignment $b = (0, 1, 1, 0)$ which does not satisfy C_j, i.e., $C_j(0, 1, 1, 0) = x_1(0) \vee \neg x_2(1) \vee \neg x_3(1) \vee x_4(0) = 0$.
- Then
$$D_j(0, 1, 1, 0, y_1^j) = (0 \vee 0 \vee y_1^j) \wedge (\neg y_1^j \vee 0 \vee 0) = y_1^j \wedge \neg y_1^j,$$
which **can not be satisfied**.
- However,
$$D_j(0, 0, 1, 0, y_1^j) = (0 \vee 1 \vee y_1^j) \wedge (\neg y_1^j \vee 0 \vee 0) = \neg y_1^j,$$
can be satisfied with $y_1^j \leftarrow 0$,
- and
$$D_j(0, 1, 1, 1, y_1^j) = (0 \vee 0 \vee y_1^j) \wedge (\neg y_1^j \vee 0 \vee 1) = y_1^j,$$
can be satisfied with $y_1^j \leftarrow 1$.

A Further Example

- **The clause** $C_j = x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5$ defines

$$D_j = (x_1 \vee \neg x_2 \vee y^j_1) \wedge (\neg y^j_1 \vee \neg x_3 \vee y^j_2) \wedge (\neg y^j_2 \vee x_4 \vee \neg x_5).$$

- $D_j(0,1,1,0,1,y^j_1,y^j_2) = (0 \vee 0 \vee y^j_1) \wedge (\neg y^j_1 \vee 0 \vee y^j_2) \wedge (\neg y^2_1 \vee 0 \vee 0) = y^j_1 \wedge (\neg y^j_1 \vee y^j_2) \wedge \neg y^j_2$, **can not be satisfied**.
- $D_j(0,0,1,0,1,y^j_1,y^j_2) = (0 \vee 1 \vee y^j_1) \wedge (\neg y^j_1 \vee 0 \vee y^j_2) \wedge (\neg y^2_1 \vee 0 \vee 0) = (\neg y^j_1 \vee y^j_2) \wedge \neg y^j_2$, **can be satisfied** by $(y^j_1, y^j_2) \leftarrow (0,0)$.
- $D_j(0,1,0,0,1,y^j_1,y^j_2) = (0 \vee 0 \vee y^j_1) \wedge (\neg y^j_1 \vee 1 \vee y^j_2) \wedge (\neg y^2_1 \vee 0 \vee 0) = y^j_1 \wedge \neg y^j_2$, **can be satisfied** by $(y^j_1, y^j_2) \leftarrow (1,0)$.
- $D_j(0,1,1,0,0,y^j_1,y^j_2) = (0 \vee 0 \vee y^j_1) \wedge (\neg y^j_1 \vee 0 \vee y^j_2) \wedge (\neg y^2_1 \vee 0 \vee 1) = y^j_1 \wedge (\neg y^j_1 \vee y^j_2)$, **can be satisfied** by $(y^j_1, y^j_2) \leftarrow (1,1)$.

Clique is *NP*-complete, I

Theorem 76

Clique is NP-complete.

Proof: We define a **polynomial reduction from** 3*SAT* **to** *Clique* by constructing for each 3*CNF*-formula $C = C_1 \wedge \cdots \wedge C_m$ over $X = \{x_1, \cdots, x_n\}$ a graph $G = (V, E)$ with $|V| = 3m$ nodes, such that $C \in SAT \iff G$ **has a clique of size** m, i.e., $(G, m) \in Clique$.

As $L \equiv L \vee L$ for each literal L we can assume w.l.o.g. that each clause in C has exactly 3 literals, i.e.,

$$C = (L_1^1 \vee L_2^1 \vee L_3^1) \wedge (L_1^2 \vee L_2^2 \vee L_3^2) \wedge \cdots \wedge (L_1^m \vee L_2^m \vee L_3^m).$$

The Set V **of Nodes of** G consists of m groups $V^1 \cup V^2 \cup \cdots \cup V^m$ of three nodes each, i.e., $V^j = \{v_1^j, v_2^j, v_3^j\}$ corresponding to the literals $\{L_1^j, L_2^j, L_3^j\}$ of clause C_j.

The Set E **of Edges of** G**:** Let $\{v_a^j, v_b^k\} \in E$ if and only v_a^j and v_b^k belong to **different groups** V^j and V^k, i.e., $j \neq k$, and the corresponding literals L_a^j and L_b^k **do not contradict**, i.e., $L_a^j \neq \neg L_b^k$.

Clique is *NP*-complete, II

We show that $C \to (G, m)$ defines a polynomial reduction from 3*SAT* to *Clique*.

- Assume first that G contains an m-clique $V' = \{v^{j_1}_{a_1}, \cdots, v^{j_m}_{a_m}\}$.
- As there are no edges inside the groups V^i, all nodes in V' have to come from different subgroups, i.e.,
$$V' = \{v^1_{a_1}, v^2_{a_1}, \cdots, v^m_{a_m}\},$$
and the corresponding set $\{L^1_{a_1}, L^2_{a_2}, \cdots, L^m_{a_m}\}$ does not contain a pair $\{x_i, \neg x_i\}$ of contradicting literals.
- Consequently, there is an $\{0,1\}$-assignment b of X_n such that $L^1_{a_1}(b) = L^2_{a_2}(b) = \cdots L^m_{a_m}(b) = 1$, which implies that $C(b) = 1$ as b satisfies at least one literal per clause, i.e., $C \in 3SAT$.
- Now assume that $C \in 3SAT$, let b a satisfying assignment for C, and fix for each clause C_j a literal $L^j_{a_j}$ with $L^j_{a_j}(b) = 1$.
- Consequently, $\{L^1_{a_1}, L^2_{a_2}, \cdots, L^m_{a_m}\}$ does not contain a pair $\{x_i, \neg x_i\}$ of contradicting literals, which implies that $\{v^1_{a_1}, v^2_{a_1}, \cdots, v^m_{a_m}\}$ is an m-clique in G. $\square$

An Example

Consider the 3*CNF*-formula

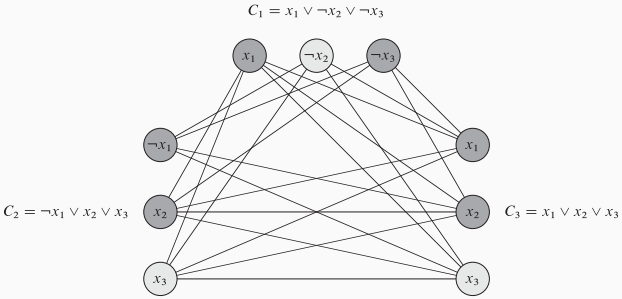
$$C = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

defining the graph G over the node groups V^1, V^2, and V^3:

$$\begin{pmatrix} v_1^1 \\ v_2^1 \\ v_3^1 \end{pmatrix} \quad \begin{pmatrix} v_1^2 \\ v_2^2 \\ v_3^2 \end{pmatrix} \quad \begin{pmatrix} v_1^3 \\ v_2^3 \\ v_3^3 \end{pmatrix}$$

Consider the assignment $b = (0, 0, 1)$ which satisfies the blue literals and forms the blue 3-clique in G.

Example


$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_1
$C_2 = \neg x_1 \vee x_2 \vee x_3$
x_2
x_2
$C_3 = x_1 \vee x_2 \vee x_3$
x_3
x_3

The NP-Completeness of the Special Knapsack Problem

Definition 77

The special knapsack problem KP^* consists of input instances $(n, w_1, \cdots, w_n, W)$, where $w_i, W \in \mathbb{N}$, for which there is a subset $I \subseteq \{1, \cdots, n\}$ fulfilling $\sum_{i \in I} w_i = W$.

Lemma 78

$KP^* \leq_{pol} KP$.

Proof: We assign to each KP^*-instance $(n, w_1, \cdots, w_n, W)$ the KP-instance $(n, w_1, \cdots, w_n, W, c_1, \cdots, c_n, C)$ with $c_i \leftarrow w_i$, and $C \leftarrow W$.

It can be easily checked that $(n, w_1, \cdots, w_n, W) \in KP^* \Leftrightarrow$
$(w_1, \cdots, w_n, W, w_1, \cdots, w_n, W) \in KP$. $\square$

Theorem 79

$3SAT \leq_{pol} KP^*$, *i.e., the special knapsack problem and the knapsack problem are NP-complete.*

The Polynomial Reduction $3SAT$ to KP^*

Proof of Theorem 79: We assign to each $3CNF$-formula $C = C_1 \wedge \cdots \wedge C_m$ over $X = \{x_1, \cdots, x_n\}$ the following decimal numbers with $n + m$ digits

- Numbers $a^i = (\underset{1}{0}, \cdots, 0, \underset{i}{1}, 0 \cdots, 0, \underset{n}{a^i_{n+1}}, \cdots, a^i_{n+m})$ for $i = 1, \cdots, n$,
  with $a_{n+j} \in \{0, 1\}$ and $a_{n+j} = 1$ if and only if **literal** x_i **belongs to clause** C_j.
- Numbers $b^i = (\underset{1}{0}, \cdots, 0, \underset{i}{1}, 0 \cdots, 0, \underset{n}{b^i_{n+1}}, \cdots, b^i_{n+m})$ for $i = 1, \cdots, n$,
  with $b_{n+j} \in \{0, 1\}$ and $b_{n+j} = 1$ if and only **literal** $\neg x_i$ **belongs to clause** C_j.
- Numbers $c^j = (\underset{1}{0}, \cdots, 0, \underset{n}{0}, 0, \cdots, 0, \underset{n+j}{1}, 0, \cdots, 0)$ for $j = 1, \cdots, m$
- Numbers $d^j = (\underset{1}{0}, \cdots, 0, \underset{n}{0}, 0, \cdots, 0, \underset{n+j}{2}, 0, \cdots, 0)$ for $j = 1, \cdots, m$
- Number $W = (\underset{1}{1}, \cdots, 1, \underset{n}{4}, \cdots, 4)$.

Example

Consider the 3*CNF*-formula

$$C = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

with $n = m = 3$ defining

$$a^1 = 100101 \qquad a^2 = 010110 \qquad a^3 = 001100$$

$$b^1 = 100010 \qquad b^2 = 010001 \qquad b^3 = 001011$$

$$c^1 = 000100 \qquad c^2 = 000010 \qquad c^3 = 000001$$

$$d^1 = 000200 \qquad d^2 = 000020 \qquad d^3 = 000002$$

Comments

Obviously, the KP^*-instance $(2(n+m), a^1, \cdots, a^n, b^1, \cdots, b^n, c^1, \cdots, c^m, d^1, \cdots, d^m, W)$ can be constructed from C in **polynomial time** in n and m.

Note that for all $i = 1, \cdots, n$ the **numbers a^i and b^i encode the formula** C in the sense that they **record the occurences of the literals x_i and $\neg x_i$ in** C.

Lemma 80

The assignment $C \to (2(n+m), a^1, \cdots, a^n, b^1, \cdots, b^n, c^1, \cdots, c^m, d^1, \cdots, d^m, W)$ defines a polynomial reduction from 3SAT to KP^*.

Proof of Lemma 80: Let us try to construct a collection of a^i, b^i, c^i, d^i- numbers which sum up to W.

For matching the n leading ones of W, for all $i = 1, \cdots, n$ we have to insert **either a^i or b^i** into the collection.

The Proof of Lemma 80

This is equivalent to choose some assignment $\beta \in \{0,1\}^n$ and to include a^i if $\beta_i = 1$ and to include a^i if $\beta_i = 0$.

Let us denote by $W(\beta)$ the corresponding partial sum

$$W(\beta) = \sum_{i,\beta_i=1} a^i + \sum_{i,\beta_i=0} b^i = (1, \cdots, 1, W(\beta)_{n+1}, \cdots, W(\beta)_{n+m}).$$

Note that for all $j = 1, \cdots, m$ the entry $W(\beta)_{n+j}$ belongs to $\{0,1,2,3\}$ and equals **the number of literals in clause C_j which are satisfied by β**.

Consequently, β satisfies C if and only if $W(\beta)_{n+j} \in \{1,2,3\}$ for all $j = 1, \cdots, m$.

Lemma 81

There are subsets $J, K \subseteq \{1, \cdots, m\}$ such that $W(\beta) + \sum_{j\in J} c^j + \sum_{k\in K} d^k = W$ if and only if $W(\beta)_{n+j} \in \{1,2,3\}$ for all $j = 1, \cdots, m$.

Note that Lemma 81 concludes the proof of Lemma 80.

Example

Consider the 3*CNF*-formula

$$C = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

with $n = m = 3$ defining

$$a^1 = 100101 \qquad a^2 = 010110 \qquad a^3 = 001100$$

$$b^1 = 100010 \qquad b^2 = 010001 \qquad b^3 = 001011$$

$$c^1 = 000100 \qquad c^2 = 000010 \qquad c^3 = 000001$$

$$d^1 = 000200 \qquad d^2 = 000020 \qquad d^3 = 000002$$

$W(1,1,0) = a^1 + a^2 + b^3 = 111222$, i.e., $W(1,1,0) + c^1 + c^2 + c^3 = 111444$.

$W(1,0,1) = a^1 + b^2 + a^3 = 111202$, i.e., $C(1,0,1) = 0$.

The Proof of Lemma 81

For all $j = 1, \cdots, m$ **do**

- If $W(\beta)_{n+j} = 1$ (i.e., β satisfies exactly one literal in clause C_j) then $(W(\beta) + c^j + d^j)_j = 4$, i.e., include j into J and K.
- If $W(\beta)_{n+j} = 2$ (i.e., β satisfies exactly two literals in clause C_j) then $(W(\beta) + d^j)_j = 4$, i.e., include j into K.
- If $W(\beta)_{n+j} = 3$ (i.e., β satisfies exactly three literals in clause C_j) then $(W(\beta) + c^j)_j = 4$, i.e., include j into J.

If there is some j such that $W(\beta)_{n+j} = 0$ (which is equivalent to β **does not satisfy** C_j and, thus, β **does not satisfy** C) then there is **no possibility to bring this component to** 4.

We have shown that $C \in 3SAT$ if and only if $(2(n+m), a^1, \cdots, a^n, b^1, \cdots, b^n, c^1, \cdots, c^m, d^1, \cdots, d^m, W) \in KP^*$. $\square$

The *NP*-Completeness of *Partition*

Remember that the problem **Partition** is to decide if a sequence of natural numbers can be partitioned into two subsets which have the same sum.

Theorem 82

It holds $KP^* \leq$ Partition, i.e., Partition is NP-complete.

Proof: We assign to each KP^*-instance $(n, w_1, \cdots, w_n, W)$ the *Partition*-instance $(w_1, \cdots, w_n, S - W + 1, W + 1)$ for *Partition*, where $S = \sum_{i=1}^{n} w_i$, and show that this defines a polynomial reduction.

Assume first that $(n, w_1, \cdots, w_n, W) \in KP^*$, i.e., there is some $I \subseteq \{1, \cdots, n\}$ such that $\sum_{i \in I} w_i = W$.

Then

$$\sum_{i \in I} w_i + S - W + 1 = S + 1 = S - W + W + 1 = \sum_{i \notin I} w_i + W + 1,$$

which implies that $(w_1, \cdots, w_n, S - W + 1, W + 1) \in$ *Partition*.

The Proof of Theorem 82

Now assume that $(w_1, \cdots, w_n, S - W + 1, W + 1) \in \mathit{Partition}$, i.e., there is some subset $J \subseteq \{1, \cdots, n + 2\}$ which defines a partition.

As $S - W + 1 + W + 1 = S + 2 > S + 1$, it holds that either $S - W + 1 \in J$ or $W + 1 \in J$.

We assume w.l.o.g. that $S - W + 1 \in J$, i.e., there is some subset $I \subseteq \{1, \cdots, n\}$ such that

$$\sum_{i \in I} w_i + S - W + 1 = S + 1,$$

which implies $\sum_{i \in I} w_i = W$ and, thus, $(n, w_1, \cdots, w_n, W) \in KP^*$. $\square$

The *NP*-Completeness of *TSP* and *HC*

Definition 83

The **Directed Hamiltonian Circuit Problem**, for short *DHC*, denotes the problem to decide if a given **directed** graph $G = (V, E)$ contains a DHC, i.e., a directed circuit with n edges which visits each node $v \in V$ exactly once.

- The *NP*-Completeness of *TSP* and *HC* follows from the reductions

$$3SAT \leq_{pol} DHC \leq_{pol} HC \leq_{pol} TSP.$$

- We have already shown the polynomial reduction $HC \leq_{pol} TSP$.
- The reduction $DHC \leq_{pol} HC$ will be shown later.

We show now

Theorem 84

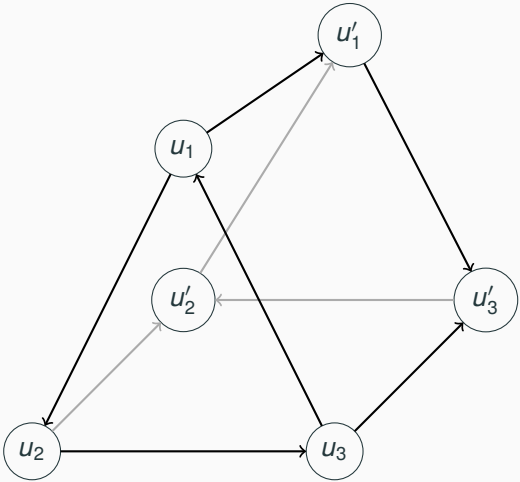
$3SAT \leq_{pol} DHC.$

The *NP* Polynomial Reduction from 3*SAT* to *DHC*

We assign to each 3*CNF*-formula $C = C_1 \wedge \cdots \wedge C_m$ over $X_n = \{x_1, \cdots, x_n\}$ a directed graph $G = (V, E)$, which has the following components:

- n **control nodes** $v_1, \cdots, v_n$,
- m **component graphs** $G_1, \cdots, G_m$, where each component graph G_j has six nodes, three **entry nodes** u^j_1, u^j_2, u^j_3 and three **exit nodes** u'^j_1, u'^j_2, u'^j_3.
  G_j consists of the **entry triangle** $\{(u^j_1, u^j_2), (u^j_2, u^j_3), (u^j_3, u^j_1)\}$ and the reversely oriented **exit triangle** $\{(u'^j_1, u'^j_3), (u'^j_3, u'^j_2), (u'^j_2, u'^j_1)\}$, connected by the **bridging edges** $\{(u^j_1, u'^j_1), (u^j_2, u'^j_2), (u^j_3, u'^j_3)\}$
- For each i, $1 \leq i \leq n$, there is an **directed x_i-path and an directed $\neg x_i$-path** , both starting in control node v_i, **visiting all component graphs G_j for which clause C_j contains the literal x_i, respectively** $\neg x_i$, and finishing at control node v_{i+1}, for $i < n$ and v_1 for $i = n$.
- Consequently, G has $n + 6m$ nodes, and the set of edges contains the $9m$ internal edges of the component graphs and the edges of the x_i- and the$\neg x_i$-paths.

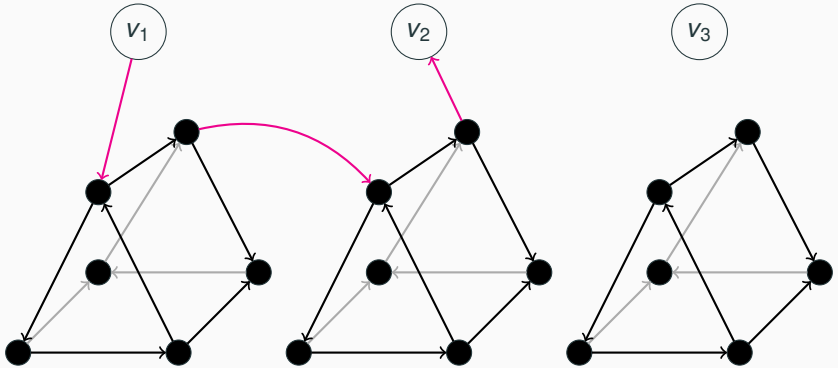
Reduction from 3-SAT to Hamiltonian Circuit



How the x_i-Paths visit Components

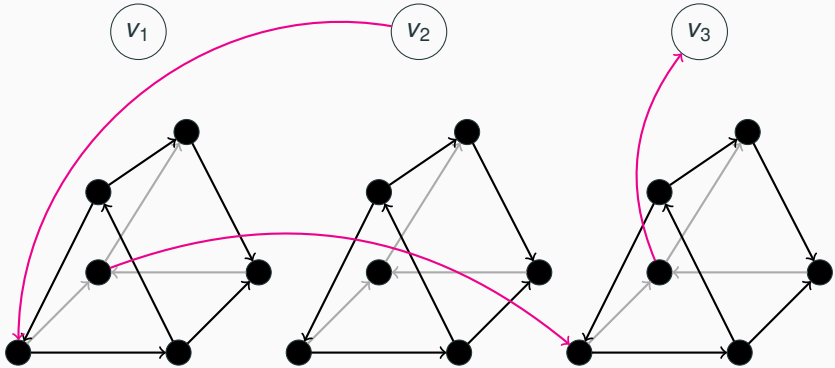
- Let $C_{j_1}, \cdots, C_{j_s}$ denote the clauses containing x_i, where $j_1 < j_2 < \cdots < j_s$.
- Then the x_i-path connects the components $G_{j_1}, \cdots, G_{j_s}$
- Choose values $k_1, \cdots, k_s$ from $\{1, 2, 3\}$ such that the literals $L^{j_1}_{k_1}, \cdots, L^{j_s}_{k_s}$ equal x_i.
- If $i < n$ then the x_i-path contains the edges $v_i \to u^{j_1}_{k_1}$, $u'^{j_1}_{k_1} \to u^{j_2}_{k_2}, \cdots, u'^{j_{s-1}}_{k_{s-1}} \to u^{j_s}_{k_s}$, $u^{j_s}_{k_s} \to v_{i+1}$.
- If $i = n$ then the x_i-path contains the edges $v_n \to u^{j_1}_{k_1}$, $u'^{j_1}_{k_1} \to u^{j_2}_{k_2}, \cdots, u'^{j_{s-1}}_{k_{s-1}} \to u^{j_s}_{k_s}$, $u^{j_s}_{k_s} \to v_1$.
- The $\neg x_i$-path connects v_i with the components containing $\neg x_i$ and $v_{i+1 \mod n}$ in the same way.

Reduction from 3-SAT to Hamiltonian Circuit, x_1-path



$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$$

Reduction from 3-SAT to Hamiltonian Circuit, $\bar{x}_2$-path



$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$$

Properties of Possible DHCs in G

We have to show that $C \in 3SAT$ if and only if $G \in DHC$.

First note that there are the following nine possibilities that a **simple directed circuit in G goes through component G_j**, where **the entry node** is u_1^j.

(1) $\cdots \to u_1^j \to u_1'^j \to \cdots$
(2) $\cdots \to u_1^j \to u_1'^j \to u_3'^j \to \cdots$
(3) $\cdots \to u_1^j \to u_1'^j \to u_3'^j \to u_2'^j \cdots$
(4) $\cdots \to u_1^j \to u_2^j \to u_2'^j \to \cdots$
(5) $\cdots \to u_1^j \to u_2^j \to u_2'^j \to u_1'^j \to \cdots$
(6) $\cdots \to u_1^j \to u_2^j \to u_2'^j \to u_1'^j \to u_3'^j \to \cdots$
(7) $\cdots \to u_1^j \to u_2^j \to u_3^j \to u_3'^j \to \cdots$
(8) $\cdots \to u_1^j \to u_2^j \to u_3^j \to u_3'^j \to u_2'^j \to \cdots$
(9) $\cdots \to u_1^j \to u_2^j \to u_3^j \to u_3'^j \to u_2'^j \to u_1'^j \to \cdots$

Corresponding statements are true if the entry nodes are u_2^j or u_3^j.

The Key Lemma 85

The key for the construction lies in the following property of the component graphs G_j.

Lemma 85

Suppose that a DHC in G enters component G_j via u^j_1. Then the following three out of nine possibilities are possible.

(1) $\cdots \to u^j_1 \to u'^j_1 \to \cdots$,

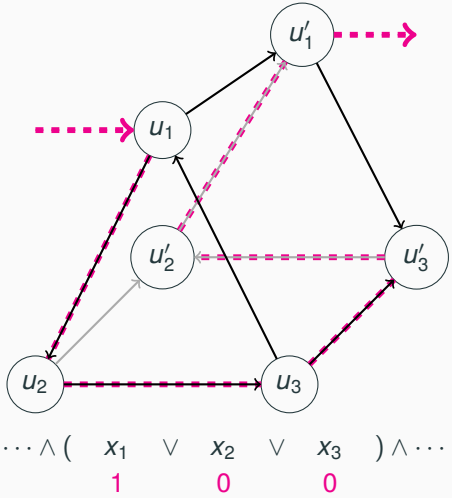
(5) $\cdots \to u^j_1 \to u^j_2 \to u'^j_2 \to u'^j_1 \to \cdots$,

(9) $\cdots \to u^j_1 \to u^j_2 \to u^j_3 \to u'^j_3 \to u'^j_2 \to u'^j_1 \to \cdots$.

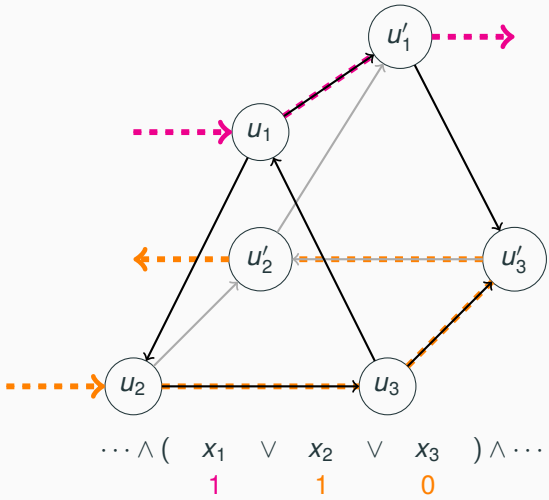
A corresponding result holds if a DHC enters G_j via u^j_2 resp. u^j_3.

Proof: Note that in all but in possibility (9), the *DHC* has to enter G_j at least twice as not all nodes in G_j have been visited. We prove Lemma 85 by showing for possibilities (2,3,4) and (6,7,8) that it is not possible for the *DHC* to catch all nodes in G_j during later visits.

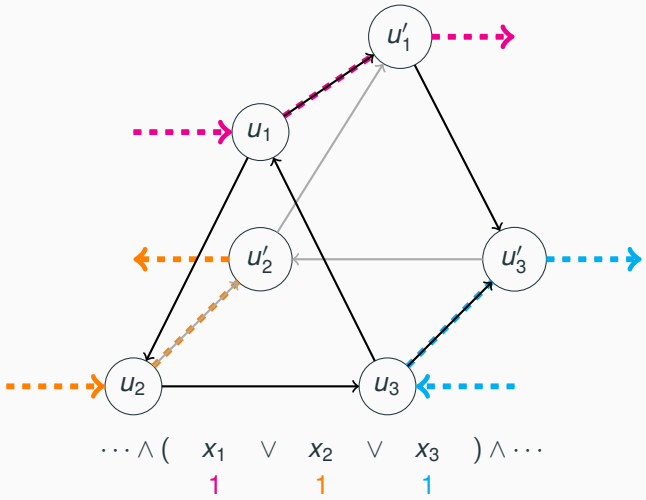
Reduction from 3-SAT to Hamiltonian Circuit, One Visit



Reduction from 3-SAT to Hamiltonian Circuit, Two Visits



Reduction from 3-SAT to Hamiltonian Circuit, Three Visits



The Proof of Lemma 85

Look, for instance at possibility (8) $\cdots \to u^j_1 \to u^j_2 \to u^j_3 \to u'^j_3 \to u'^j_2 \to \cdots$. Here, no *DHC* can reach the remaining node U^j_1, as the only nodes, which have edges to u'^j_1, namely u'^j_2 and u^j_1, have been already visited.

The proofs of the remaining cases are left to the reader. □ Lemma 85 implies

Lemma 86

- *Each possible DHC entering G_j via u^j_k has to leave it via u'^j_k, $k = 1, 2, 3$.*
- *Each possible DHC* ***has to follow*** *β for some assignment $\beta = (\beta_1, \cdots, \beta_n) \in \{0,1\}^n$ in the sense that it has to follow either the x_1-path ($\beta_1 = 1$), or the $\neg x_1$-path ($\beta_1 = 0$) between v_1 and v_2, and either the x_2-path ($\beta_2 = 1$) or the $\neg x_2$-path ($\beta_2 = 0$) between v_2 and v_3, $\cdots$, and either the x_n-path ($\beta_n = 1$) or the $\neg x_n$-path ($\beta_n = 0$) between v_n and v_1.*
- *Each possible DHC can visit G_j either* ***once*** *(taking possibility (9)), or* ***twice*** *(taking possibility (1) and (5)), or* ***three times*** *(taking possibility (1) each time). □*

Consequences from Lemma 86, Completing the Proof of the Theorem

- A possible *DHC* which follows $\beta = (\beta_1, \cdots, \beta_n) \in \{0,1\}^n$ visits a component $G_j \iff$ it contains an L-path for some literal L occuring in $C_j \iff \beta$ satisfies C_j.
- Consequently, if $C(\beta) = 0$ then no possible *DHC* which follows $\beta = (\beta_1, \cdots, \beta_n) \in \{0,1\}^n$ can stay a *DHC* in G, as the components G_j for which $C_j(\beta) = 0$ will not be visited.
- **Consequently, if** $C \notin SAT$ **then** $G \notin DHC$.
- If $C(\beta) = 1$ then all components G_j are visited. We transform a possible *DHC* which follows $\beta = (\beta_1, \cdots, \beta_n) \in \{0,1\}^n$ into a complete *DHC* by choosing, for all $j = 1, \cdots, m$,
  - possibility (9), if one literal in C_j is satisfied by β,
  - possibility (1) and (5), if two different literals in C_j are satisfied by β, or
  - three times possibility (1), if three different literals in C_j are satisfied by β.
- **Consequently, if** $C \in SAT$ **then** $G \in DHC$. $\square$

The *NP*-Completeness of *HC*

Theorem 87

It holds DHC $\leq_{pol}$ HC, i.e., HC is NP-complete.

Proof: We define a polynomial reduction from *DHC* to *HC* by assigning to each **directed graph** $G = (V, E)$ an **undirected graph** $G' = (V', E')$ as follows

$$V' = \bigcup_{v \in V} \{v_l, v_m, v_r\}$$

$$E' = \bigcup_{v \in V} \{\{v_l, v_m\}, \{v_m, v_r\}\} \cup \bigcup_{(v,w) \in E} \{\{v_r, w_l\}\}.$$

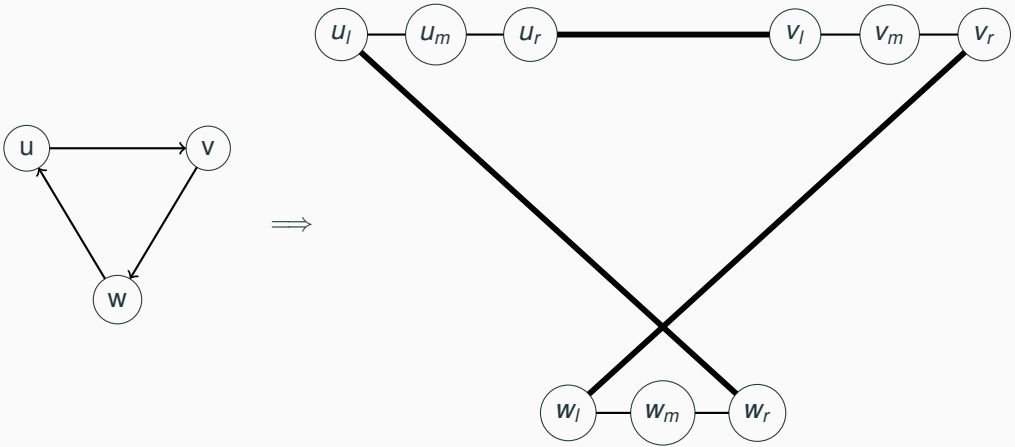
- Suppose that G contains a *DHC* $u^1 \to u^2 \to \cdots \to u^{|V|} \to u^1$.
- Then

  $$u_r^1 - u_l^2 - u_m^2 - u_r^2 - \cdots - u_l^{|V|} - u_m^{|V|} - u_r^{|V|} - u_l^1 - u_m^1 - u_r^1$$

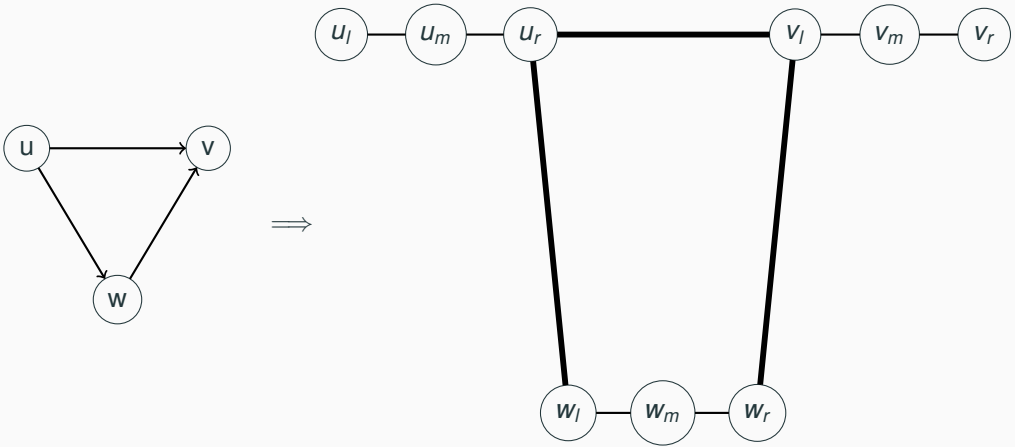
  defines a *HC* in G', i.e., $G \in DHC \Longrightarrow G' \in HC$.

Reduction from DHC to HC

Reduction from DHC to HC



Reduction from DHC to HC



Completing the Proof of Theorem 87

(1) Suppose now that G' contains a HC.

(2) As every node in an HC is part of two consecutive HC-edges, and as $u_l{-}u_m{-}u_r$ are the only edges in E' containing a node of type u_m, the HC has to contain the pair $u_l{-}u_m{-}u_r$ of consecutive edges for all $u \in V$.

(3) Obviously, the HC has to contain edges of type $\{v_r, w_l\}$ as only edges of this type connect the isolated components $u_l{-}u_m{-}u_r$.

(4) Assume that such an edge $\{v_r, w_l\}$ will be followed by an edge of type $\{w_l, z_r\}$, i.e., the HC contains a part $v_r - w_l - z_r$. Then the edge $w_l{-}w_m$ can not be in the HC which contradicts to item (2).

(5) This implies that the HC has the form

$$u_r^1 - u_l^2{-}u_m^2{-}u_r^2 - \cdots - u_l^{|V|}{-}u_m^{|V|}{-}u_r^{|V|} - u_l^1{-}u_m^1{-}u_r^1$$

which defines the DHC $u^1 \to u^2 \to \cdots \to u^{|V|} \to u^1$ in G.

(6) Consequently, $G' \in HC \Longrightarrow G \in DHC$. $\square$

Hardness of General Problems

The Goal

- **Assume $P \neq NP$:** If a **decision problem** L is **NP-complete** then $L \notin P$.
- A corresponding concept for **general computational problems** Π**:**

$$\Pi \text{ is called } \textbf{NP-hard} \text{ if } \Pi \in PTIME \text{ implies } P = NP.$$

How to show NP-hardness?

- We showed NP-completeness by **constructing polynomial reductions**, a concept applicable **only to decision problems**.
- Here we define **Turing-reducibility ($\Pi \leq_T \Pi'$)**, a reducibility concept applicable to **general problems** Π, Π'.
- Informally, a **Turing reduction from** Π **to** Π' is a polynomial time algorithm for Π, which has **access to a subroutine for** Π', which on arbitrary inputs x returns immediately a solution y with $(x, y) \in \Pi'$.
- More formally, $\cdots$

Oracle Algorithms and Turing-Reducibility

Definition 88

- A Π'**-Oracle** is a **computational device** with possibly **supernatural computational power**. It solves the **computational problem** Π' with **maximal efficiency**, i.e. for **all inputs** x the Π'-Oracle returns a **solution** y with $(x, y) \in \Pi'$ in time $|x| + |y|$.
- A Π'**-Oracle Algorithm** for a given computational problem Π' is an algorithm which **solves** Π and has **access to an** Π'**-oracle**.
- We define Π to be **Turing-reducible to** Π' ($\Pi \leq_T \Pi'$), if there is a **polynomial time** Π'**-oracle algorithm** for Π.

- The concept of oracle algorithms corresponds to the well-known concept in programming that a given **computer program calls a subprogram**.
- The concept of **Turing-Reducibility** allows to determine the **complexity** of a given problem Π **relative to another problem** Π' in the sense of **If** Π **is hard then also** Π' **is hard** or **If** Π' **is easy then also** Π **is easy**.

Oracle-Algorithms, Example *SAT.ASSIGNMENT*

Lemma 89

If $\Pi' \in PTIME$ ***and*** $\Pi \leq_T \Pi'$ ***then*** $\Pi \in PTIME$, *or, equivalently,* ***if*** $\Pi \notin PTIME$ ***and*** $\Pi \leq_T \Pi'$ ***then*** $\Pi' \notin PTIME$.

Proof: Suppose Π' has a $t(n)$-time bounded algorithm A' and Π has a $p(n)$-time bounded Π'-oracle algorithm, where t and p are polynomials in n.

We get a $p(t(n))$ time bounded algorithm for Π by simulating each oracle call of A to the Π'-oracle with input x by applying A' to x. $\square$

Example Problem

SAT.ASSIGNMENT

Input: A CNF-formula $C = C_1 \wedge \cdots \wedge C_m$ over $X_n = \{x_1, \cdots, x_n\}$

Output: **If** $C \in SAT$ **then output** $b \in \{0,1\}^n$ **with** $C(b) = 1$ **else output** $C \notin SAT$.

Example

$$C = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee) \wedge (x_1 \vee \neg x_3)$$

$SAT.ASSIGNMENT(C)$ outputs $b = (1, 0, 1)$

$$C = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee) \wedge (x_1 \vee \neg x_3)$$

An *SAT*-Oracle Algorithm for *SAT.ASSIGNMENT*

Lemma 90

SAT.ASSIGNMENT $\leq_T$ *SAT.*

Proof: How to **efficiently compute satisfying assignments** on the basis of an **efficient satisfiability test**?

Here, a ***SAT*-oracle algorithm** for *SAT.ASSIGNMENT* (input C):

1 **If** $SAT(C) = 0$ **then output** $C \notin SAT$**, stop.**
2 **else for** $i \leftarrow n$ **downto** 1
3 **do if** $SAT(C|_{x_i=1}) = 1$ **then** $b_i \leftarrow 1$, $C \leftarrow C|_{x_i=1}$ **else** $b_i \leftarrow 0$, $C \leftarrow C|_{x_i=0}$
4 **output** b

The **correctness** follows from $C(b_1, \cdots, b_n) = C|_{x_n=b_n}(b_1, \cdots, b_{n-1})$.

Note that $C|_{x_i=1}$ can be computed from C by deleting all clauses containing literal x_i and by deleting all occurences of literal $\neg x_i$ in the clauses containing $\neg x_i$.

Example

$$C = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee) \wedge (x_1 \vee \neg x_3)$$

$$C|_{x_1=0} = (0 \vee x_2) \wedge (1 \vee x_2 \vee x_3) \wedge (0 \vee \neg x_2 \vee x_3) \wedge (1 \vee \neg x_2 \vee \neg x_3 \vee) \wedge (0 \vee \neg x_3)$$

$$\implies C|_{x_1=0} = (x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_3) \text{ not satisfiable}$$

$$C|_{x_1=1} = (1 \vee x_2) \wedge (0 \vee x_2 \vee x_3) \wedge (1 \vee \neg x_2 \vee x_3) \wedge (0 \vee \neg x_2 \vee \neg x_3 \vee) \wedge (1 \vee \neg x_3)$$

$$\implies C|_{x_1=1} = (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee) \text{ satisfiable with } x_2 \leftarrow 0, x_3 \leftarrow 1$$

yields satisfying assignment $b = (1, 0, 1)$ for C.

Basic Cases of Turing Reducibility

Lemma 91

Polynomial Reductions are special cases of Turing reductions, i.e., from $L \leq_{pol} L'$ it follows that $L \leq_T L'$.

Proof: Let A_f be the polynomial time algorithm computing the polynomial reduction from L to L'.

Here, the **L'-oracle algorithm** for L:

- Given input x, **compute** $y = A_f(x)$.
- **Accept** $x \iff L'(y) = 1$. □

Lemma 92

For all optimization problems Π, the decisional variant L_Π is Turing reducible to Π. □

Proof: One call to the Π-oracle with input x yields the cost (resp. benefit) of an optimal solution to x, which immediately allows to decide if x respects a given cost (resp. benefit) bound. □

Self Reducibility

Self Reducibility means that the **reverse case is also true**, i.e., the optimization variant is Turing-reducible to the (easier looking) decisional variant. Here one example:

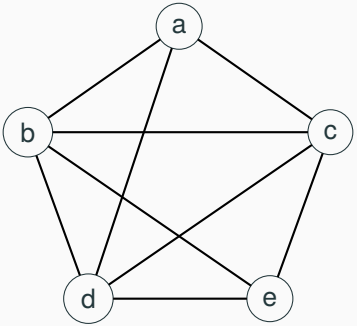
Lemma 93

The optimization problem MaxClique (of computing a clique of maximal size in a given input graph $G = (V, E)$) is Turing-reducible to its decision variant Clique (of deciding if G contains a clique of size at least k).

Proof: We describe a **polynomial time** *Clique*-**oracle algorithm for** *MaxClique*, applied to input $G = (V, E)$:

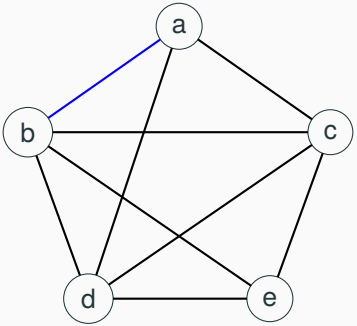
1 **Compute** $k^* \leftarrow \max\{k, \mathit{Clique}(G, k) = 1\}$ ($\leq |V|$ oracle calls).
2 **For all** $e \in E$
3 **do** $G' \leftarrow (V, E \setminus \{e\})$
4 **If** $\mathit{Clique}(G', k^*) = 1$ **then** $G \leftarrow G'$.

Example: Cliques 1



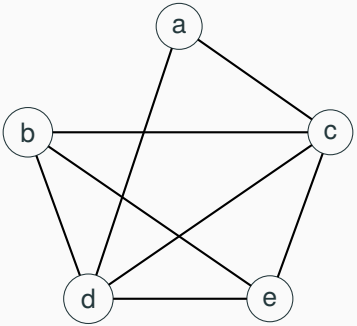
$k^* = 4$, due to $\{b, c, d, e\}$

Example: Cliques 1



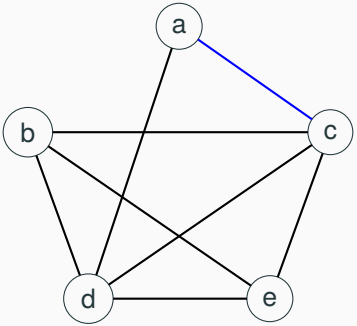
G' $Clique(G', 4) = 1$, due to $\{b, c, d, e\}$

Example: Cliques 1



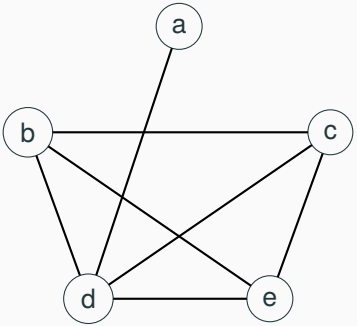
new G

Example: Cliques 1



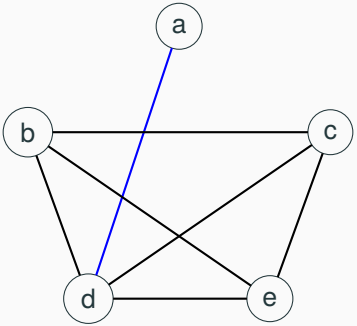
G', $Clique(G', 4) = 1$ due to $\{b, c, d, e\}$

Example: Cliques 1



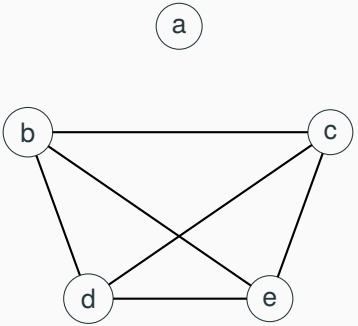
new G

Example: Cliques 1



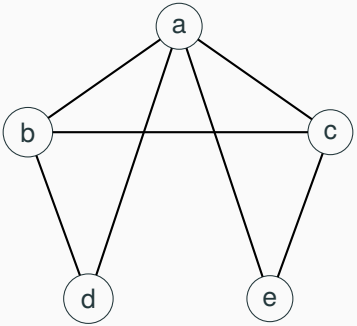
G', $Clique(G', 4) = 1$ due to $\{b, c, d, e\}$

Example: Cliques 1



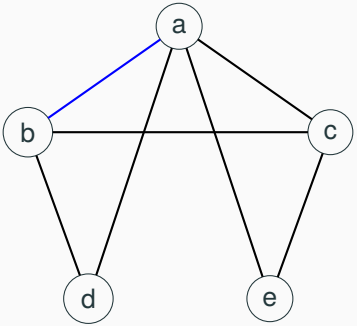
new G is 4-clique $\{b, c, d, e\}$

Example: Cliques 2



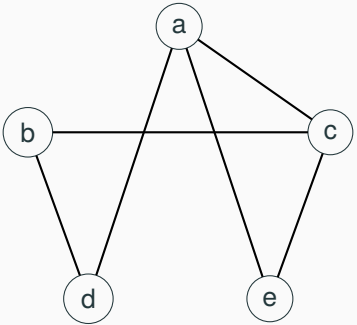
$k^* = 3$, due to, e.g., $\{a, b, c\}$

Example: Cliques 2



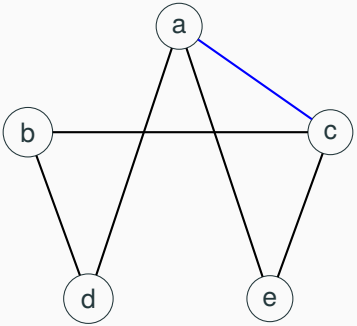
G', $Clique(G', 3) = 1$ due to $\{a, c, e\}$

Example: Cliques 2



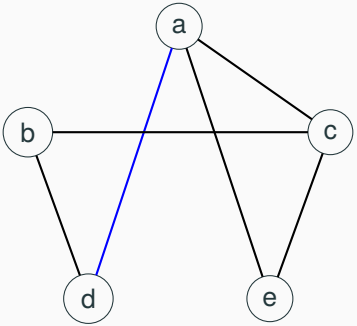
new G

Example: Cliques 2



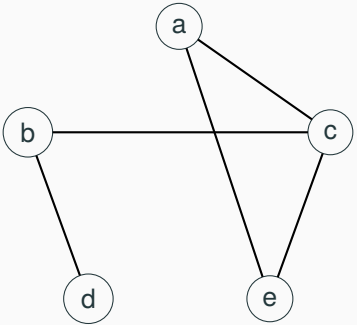
G', $Clique(G', 3) = 0$

Example: Cliques 2



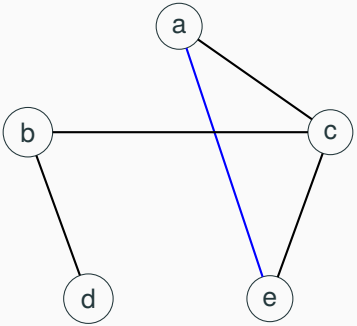
G', $Clique(G', 3) = 1$, due to $\{a, c, e\}$

Example: Cliques 2



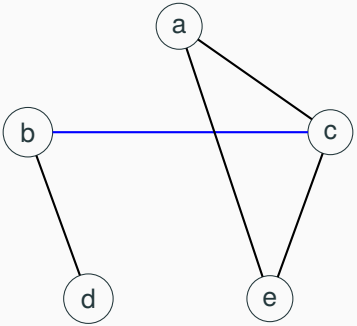
new G

Example: Cliques 2



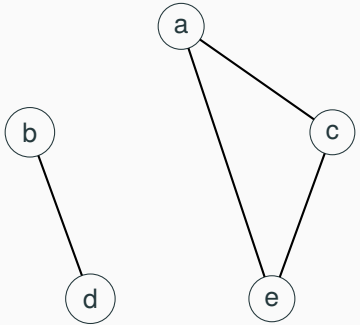
G', $Clique(G', 3) = 0$

Example: Cliques 2



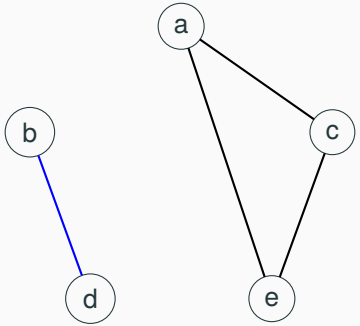
G', $Clique(G', 3) = 1$, due to $\{a, c, e\}$

Example: Cliques 2



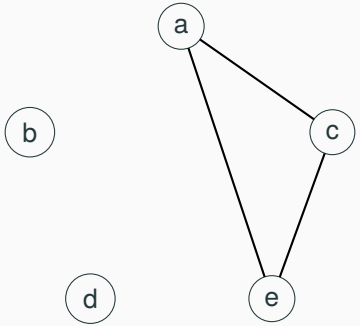
new G

Example: Cliques 2



G', $Clique(G', 3) = 1$, due to $\{a, c, e\}$

Example: Cliques 2



new G is 3-clique $\{a, c, e\}$

The Proof of Correctness

Claim: The output graph consists of a clique V^* of maximal size k^* and $n - k^*$ isolated nodes.

- As we remove only edges for which the removal does not lower the clique size, the output graph contains a clique V^* of G of maximal size k^*.
- We have to show that the output graph does not contain any edge which is outside of the clique over V^*.
- Let us assume that the opposite is true, i.e., that the output graph contains such an edge, say e^*.
- However, when calling the *Clique*-oracle with input $G \setminus \{e^*\}, k^*$ the answer is 1 as $G \setminus \{e^*\}$ contains the clique V^*. This implies that e^* is removed, contradiction. $\square$

Remark: Similar self reductions from the optimization variant to the decision variant can be constructed for other optimization problems like *Knapsack*, *TSP* and many others.

NP-hard, *NP*-easy and *NP*-equivalent Problems

Definition 94

A computational problem Π is called

- ***NP*-hard**, if there is an *NPC*-problem L with $L \leq_T \Pi$.
- ***NP*-easy**, if there is an *NP*-problem L with $\Pi \leq_T L$.
- ***NP*-equivalent**, if Π is *NP*-easy and *NP*-hard.

Some properties:

- Let Π ***NP*-hard**. **Then** $\Pi \in PTIME \Longrightarrow P = NP$.
- Let Π ***NP*-easy**. **Then** $P = NP \Longrightarrow \Pi \in PTIME$**.**
- **All** *NP* **problems are** *NP*-**easy, all** *NPC*-**problems are** *NP*-**equivalent.**
- Optimization problems, for which the decisional variant is *NP*-complete, are *NP*-hard.
- Optimization problems, which can be Turing-reduced to its decisional variant are *NP*-easy ($MaxClique, MinTSP, MaxKP, \cdots$).

Algorithms for NP-hard Optimization Problems

Approaches to solving *NP*-hard problems

Situation: In many practical situations there occur *NP*-hard optimization problems. Nevertheless, they have to be solved.

Approaches

- Algorithms which run efficiently on certain inputs (e.g. pseudopolynomial algorithms)
- Approximation algorithms computing good but not optimal solutions
- Improved Exhaustive search heuristics (e.g. Branch and Bound)
- Alternative algorithmic approaches (e.g. Simulated Annealing, Genetic Algorithms, DNA-computing, Randomized algorithms, $\cdots$)
- Combinations of different approaches

DynamicKP, a Dynamic Algorithm for *MaxKP*, (1)

W present a **dynamic algorithm** for the knapsack problem *MaxKP*, which is efficient if the benefit values are bounded.

Theorem 95

There is an algorithm for the NP-hard problem MaxKP which needs time $O(n \cdot c_{max}(I))$ for each integral input instance $I = (n, w_1, \cdots, w_n, c_1, \cdots, c_n, W)$, where $c_{max}(I) = \max\{c_i, i = 1, \cdots, n\}$.

Proof: We assume that $0 < w_i \leq W$ for all $i = 1, \cdots, n$, otherwise the corresponding objects could be removed.

Remember that the solution to I is a set of objects $S_{opt} \subseteq \{1, \cdots, n\}$ with $w(S_{opt}) \leq W$ and

$$c(S_{opt}) = c_{opt} = \max\{c(S), S \subseteq \{1, \cdots, n\}, w(S) \leq W\}.$$

Here is the idea how the algorithm computes c_{opt} and S_{opt}.

- For all $i = 1, \cdots, n$ let $\mathcal{C}_i = \{c(S); S \subseteq \{1, \cdots, i\}, w(S) \leq W\}$ contain **all benefit values** of subsets **of the first i objects**, which respect the weight bound.

DynamicKP, a Dynamic Algorithm for *MaxKP*, (2)

- **Note** that $c_{opt} = \max\{c; c \in \mathcal{C}_n\}$.
- **Moreover** $|\mathcal{C}_i| \leq i \cdot c_{\max}$ for all $i = 1, \cdots, n$.
- We compute for all $i = 1, \cdots, n$ the set $\mathcal{C}_i$ and for each $c \in \mathcal{C}_i$ a **minimal weight value** $w_{c,i}$ and a corresponding set $S_{c,i} \subseteq \{1, \cdots, i\}$ with
  - $c(S_{c,i}) = c$
  - $w_{c,i} = w(S_{c,i}) \leq W$
  - $w_{c,i} = \min\{w(S); S \subseteq \{1, \cdots, i\}, c(S) = c\}$.
- **Note** that $S_{opt} = S_{c_{opt},n}$.
- **Let** $T_i(I) = \{(c, w_{c,i}, S_{c,i}); c \in \mathcal{C}_i\}$.
- **Note** that $T_1(I) = \{(0, \emptyset, 0), (c_1, \{1\}, w_1)\}$
- **Thus**, for computing S_{opt} it is sufficient to define an **algorithm which computes** $T_{i+1}(I)$ **from** $T_i(I)$ for $i = 2, \cdots, n$.

DynamicKP, a Dynamic Algorithm for *MaxKP*, (2)

Input $T_i(I)$ for some $i = 1, \cdots, n-1$

Output $T_{i+1}(I)$

1 $T_{i+1}(I) \leftarrow T_i(I)$
2 **For all** $(c, w, S) \in T_i(I)$
3 **do if** $w + w_{i+1} \leq W$ and $T_{i+1}(I)$ does not contain already a triple $(c + c_{i+1}, \tilde{w}, \tilde{S})$ with $\tilde{w} \leq w + w_{i+1}$
4 **then put** $(c + c_{i+1}, w + w_{i+1}, S \cup \{i+1\})$ **to** $T_{i+1}(I)$
5 and **cancel** all other triples $(c + c_{i+1}, \cdot, \cdot)$ from $T_{i+1}(I)$.

Running time $O(|T_i(I)|) = O(n \cdot c_{max}(I))$, where $c_{max}(I) = \max\{c_i; i = 1, \cdots, n\}$.

This allows to compute $T_n(I)$ and the optimal solution S_{opt} in time $O(n^2 \cdot c_{max}(I))$. □

Example

$c = (11, 18, 7, 4)$, $w = (7, 9, 4, 2)$, $W = 16$

$T_1 : (0, 0, \emptyset)\ \ (11, 7, \{1\})$

$T_2 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})$

$T_3 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})$

$(7, 4, \{3\})\ \ (18, 11, \{1, 3\})\ \ (25, 13, \{2, 3\})\ \ (36, 23, \{1, 2, 3\})$

Example

$c = (11, 18, 7, 4)$, $w = (7, 9, 4, 2)$, $W = 16$

$T_1 : (0, 0, \emptyset)\ \ (11, 7, \{1\})$

$T_2 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})$

$T_3 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})$

$(7, 4, \{3\})\ \ (18, 11, \{1, 3\})\ \ (25, 13, \{2, 3\})\ \ (36, 23, \{1, 2, 3\})$

Example

$c = (11, 18, 7, 4)$, $w = (7, 9, 4, 2)$, $W = 16$

$T_1 : (0, 0, \emptyset)\ \ (11, 7, \{1\})$

$T_2 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})$

$T_3 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})\ \ (7, 4, \{3\})\ \ (25, 13, \{2, 3\})$

$T_4 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})\ \ (7, 4, \{3\})\ \ (25, 13, \{2, 3\})$

$(4, 2, \{4\})\ (15, 9, \{1, 4\})\ (22, 11, \{2, 4\})\ (36, 18, \{1, 2, 4\})\ (14, 6, \{3, 4\})\ (29, 15, \{2, 3, 4\})$

Example

$c = (11, 18, 7, 4)$, $w = (7, 9, 4, 2)$, $W = 16$

$T_1 : (0, 0, \emptyset)\ \ (11, 7, \{1\})$

$T_2 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})$

$T_3 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})\ \ (7, 4, \{3\})\ \ (25, 13, \{2, 3\})$

$T_4 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})\ \ (7, 4, \{3\})\ \ (25, 13, \{2, 3\})$

$(4, 2, \{4\})\ (15, 9, \{1, 4\})\ (22, 11, \{2, 4\})\ (36, 18, \{1, 2, 4\})\ (14, 6, \{3, 4\})\ (29, 15, \{2, 3, 4\})$

Example

$c = (11, 18, 7, 4)$, $w = (7, 9, 4, 2)$, $W = 16$

$T_1 : (0, 0, \emptyset)\ \ (11, 7, \{1\})$

$T_2 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})$

$T_3 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (29, 16, \{1, 2\})\ \ (7, 4, \{3\})\ \ (25, 13, \{2, 3\})$

$T_4 : (0, 0, \emptyset)\ \ (11, 7, \{1\})\ \ (18, 9, \{2\})\ \ (7, 4, \{3\})\ \ (25, 13, \{2, 3\})$

$(4, 2, \{4\})\ \ (15, 9, \{1, 4\})\ \ (22, 11, \{2, 4\})\ \ (14, 6, \{3, 4\})\ (29, 15, \{2, 3, 4\})$

$S_{opt} = \{2, 3, 4\},\ \ c_{opt} = 29$

Number Problems and Pseudopolynomial Algorithms

Definition 96

- A problem is called **number problem** if the components of input instances I are natural numbers.
- If I is an input instance of a number problem then let $Max(I)$ denote the maximal component of I.

Definition 97

Let Π denote a number problem. An algorithm A for Π is called **pseudopolynomial** if the running time is polynomially bounded in $|I|$ **and** $Max(I)$.

We have shown

Lemma 98

The NP-hard MaxKP has a pseudopolynomial algorithm. $\square$

Strong NP-hard problems

Definition 99

If Π is a number problem and $d : \mathbb{N} \to \mathbb{N}$ a bound function, then $\Pi_{\leq d}$ denotes the problem Π restricted to inputs I with $Max(I) \leq d(|I|)$.

Definition 100

A number problem Π is called **strong NP-hard** if there is a polynomial bound d such that $\Pi_{\leq d}$ is *NP*-hard.

Lemma 101

If a number problem Π has a pseudopolynomial algorithms then for all polynomial bounds d it holds that $\Pi_{\leq d} \in$ PTIME.

Consequently, if $P \neq NP$ then strong NP-hard number problems do not have pseudopolynomial algorithms. $\square$

The Strong *NP*-Hardness of *TSP*

Lemma 102

MinTSP is strong NP-hard.

Proof: Remember the polynomial reduction f from the *NP*-complete problem *HC* to *TSP*.

To each undirected input graph $G = (V, E)$ the reduction f assigns a distance matrix D_G with $Max(D_G) \leq 2$.

Consequently, $TSP_{\leq 2}$ is *NP*-complete, and consequently, $MinTSP_{\leq 2}$ is *NP*-hard. □

Definition 103

Each **optimization problem** Π can defined by the parameters $\Pi = (L_\Pi, c, goal)$, where

- $L_\Pi(x)$ defines for each input $x \in \{0,1\}^*$ the **set of admissible solutions** to x.
- $c : \{0,1\}^* \times \{0,1\}^* \longrightarrow \mathbb{R}$ is the **target function**, which defines for each pair (x, y), $y \in L_\Pi(x)$, the **target value** $c(x, y) \in \mathbb{R}$ of the adnissible solution y to x.
- The $goal \in \{\min, \max\}$ defines if Π is a **minimization- or a maximization problem**.

Π consists of all pairs $(x, opt_\Pi(x))$, where $x \in \{0,1\}^n$ is the input, and $opt_\Pi(x) \in L_\Pi(x)$ an optimal solution to x, i.e.,

$$c(x, opt_\Pi(x)) = goal\{c(x, \tilde{y}), \tilde{y} \in L_\Pi(x)\}.$$

Definition 104

- An algorithm A is called an **approximation algorithm** for $\Pi = (L_\Pi, c, goal)$ if it outputs for **each input** x **an admissible solution** to x, i.e. $A(x) \in L_\Pi(x)$.
- The **approximation ratio** $R_A(x) \geq 1$ of A on input x is defined to be

$$R_A(x) = \begin{cases} \frac{c(x,A(x))}{c(x,opt_\Pi(x))} & goal = \min \\ \frac{c(x,opt_\Pi(x))}{c(x,A(x))} & goal = \max \end{cases}$$

- The function $R_A : \mathbb{N} \longrightarrow \mathbb{R}$ defined by

$$R_A(n) = \max\{R_A(x), |x| = n\}$$

is called the **(worst case) approximation ratio** of A.

Note: $R_A = 1$ means that A computes an optimum for all inputs, i.e. A **solves** Π.

Properties of Approximation Algorithms

Important Fact

Many NP-complete optimization problems have **polynomial time approximation algorithms** which reach a **good approximation ratio**. In particular,

- **Good News:** There are *NP*-hard optimization problems with polynomial time approximation algorithms of **constant ratio** $R_a(n) \in O(1)$ (*VertexCover*).
- **Better News:** There are even *NP*-hard optimization problems with polynomial time approximation algorithms with constant ratio **arbitrarily close to one** (*MaxKP*).
- **Bad News:** There are other *NP*-hard optimization problems which do **not have polynomial time approximation algorithms with constant ratio** (*MinTSP*).

Determining the **minimal ratio** for which a given *NP*-hard optimization problem **can be approximated in polynomial time** is one of the **main research topics** in modern algorithmics.

Example Vertex Cover

Definition 105

A subset $V' \subseteq V$ is called a **vertex cover** of a **undirected graph** $G = (V, E)$ if **for all edges** $e = \{u, v\} \in E$ the set V' **contains** u **or** v **(or both)**.

The **problem** *VertexCover* is to compute for given undirected input graphs a **vertex cover of minimal size**.

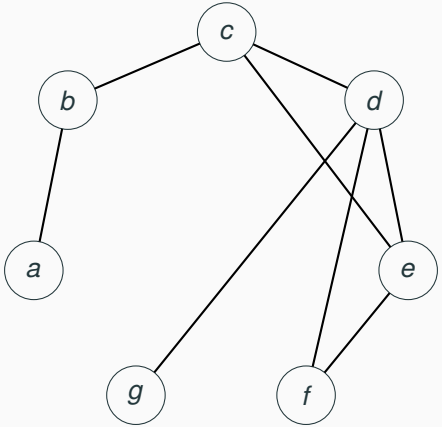
Theorem 106

*The **decisional variant of** VertexCover (decide if for given $r \in \mathbb{N}$ an undirected graph G has a vertex cover of size at most r) is **NP-complete**.*

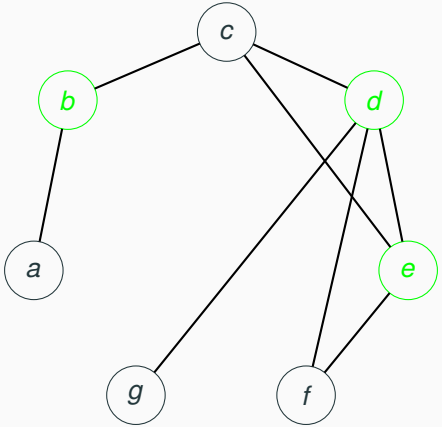
Proof: V' is a vertex cover in $G \iff$ for all nodes $v, v' \in V \setminus V'$ it holds $\{v, v'\} \notin E \iff V \setminus V'$ is a clique in the complementary graph $\bar{G}$.

Consequently, $(G, k) \to (\bar{G}, |V| - k)$ defines a **polynomial reduction from** *Clique* **to** *VertexCover*. $\square$

Example Graph Vertex Cover



Optimal Vertex Cover (3 Nodes)



An Approximation Algorithm for *VertexCover*

Definition 107

A matching $M \subseteq E$ of an undirected graph $G = (V, E)$ is called a **locally maximal matching**, if $M \not\subset M'$ for all matchings M' of G.

Lemma 108

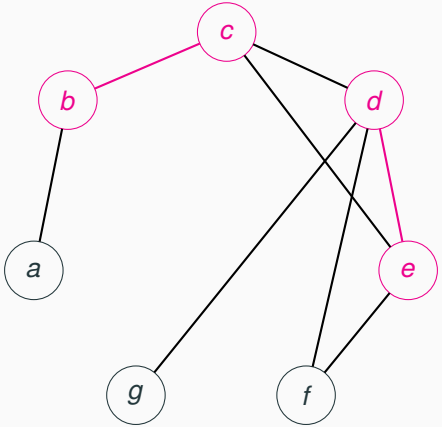
*Let $M \subseteq E$ be a **locally maximal matching** and $V' \subseteq V$ a **minimal vertex cover** in $G = (V, E)$ then $|M| \leq |V'| \leq 2|M|$.*

Proof: As the edges in a matching are pairwise disjoint, V' has to contain at least one node per M-edge, i.e., $|M| \leq |V'|$.

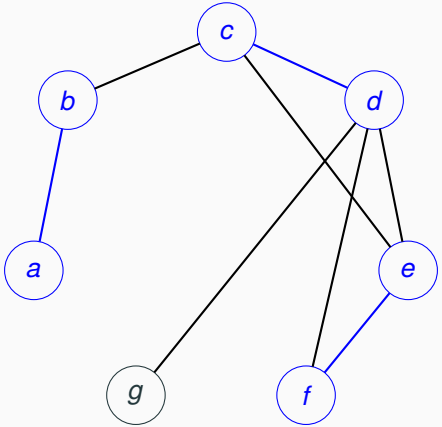
Moreover, as no $E \setminus M$-edge can be added to M without destroying the matching property, the set V_M of all nodes which are contained in an M-edge forms a vertex cover, i.e., $|V'| \leq |V_M| = 2|M|$. $\square$

Attention: A locally maximal matching is not necessarily a (globally) maximal matching.

A Locally Maximal Matching M and V_M



A (Globally) Maximal Matching M'



Efficient Computation of a Locally Maximal Matching

We proved

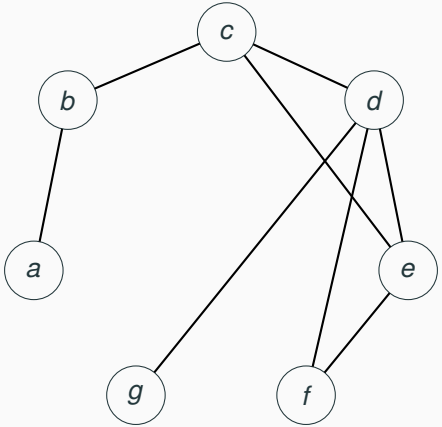
Lemma 109

Computing a locally maximal matching $M \subseteq E$ *in an undirected input graph* $G = (V, E)$ *and* ***outputting*** V_M *yields an* ***approximation algorithm for*** *VertexCover* ***with constant approximation ratio*** 2. □

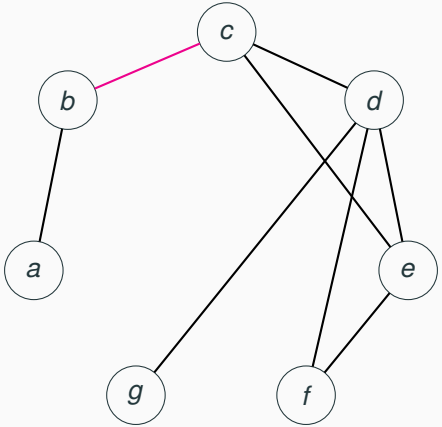
A simple **efficiently greedy algorithm** for computing a locally maximal matching $M \subseteq E$ in an undirected input graph $G = (V, E)$:

```
1 E' ← E
2 M ← ∅
3 while E' ≠ ∅ do
4     choose e = (u, v) ∈ E'
5         M ← M ∪ {e}
6         Delete all edges in E' containing u or v
```

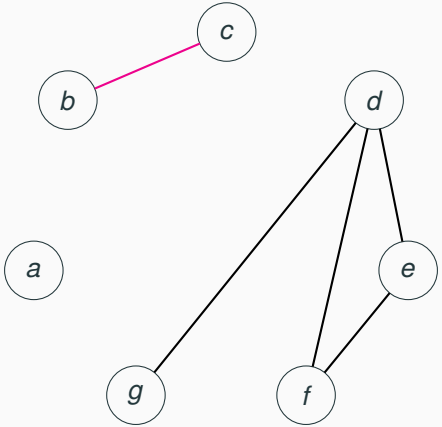
Example Graph Vertex Cover



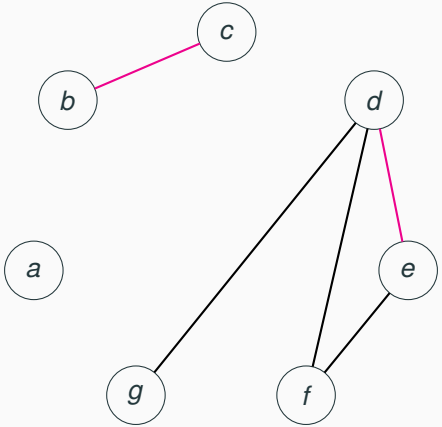
Computing a Locally Maximal Matching, 1



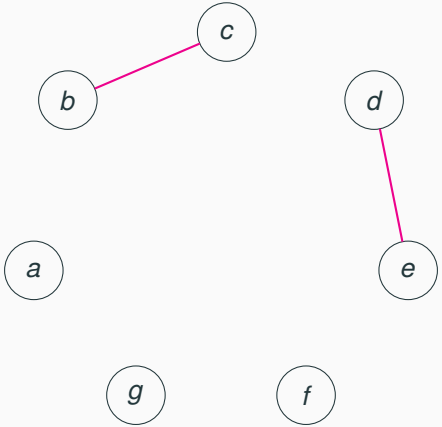
Computing a Locally Maximal Matching, 2



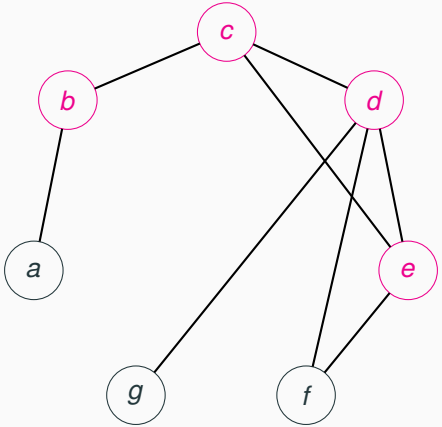
Computing a Locally Maximal Matching, 3



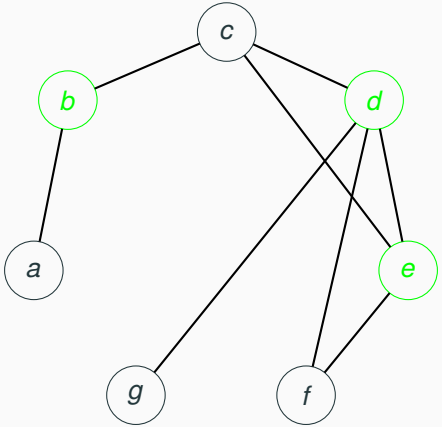
Computing a Locally Maximal Matching, 4



Approximate Vertex Cover (4 Nodes)



Optimal Vertex Cover (3 Nodes)



Fully Polynomial Approximation Schemes

Definition 110

A **Fully Polynomial Approximation Scheme (FPTAS)** for an optimization problem Π is an algorithm $A = A(x, \epsilon)$, such that, for all $\epsilon > 0$, $A(\cdot, \epsilon)$ **is an approximation algorithm for** Π **with ratio** $1 + \epsilon$.

Moreover, for all $\epsilon > 0$, the running time of $A(\cdot, \epsilon)$ is polynomial in $|x|$ and ϵ^{-1}.

Theorem 111

MaxKP has a FPTAS.

Proof: The corresponding algorithm is called **KP-FPTAS**. It is based on the pseudopolynomial algorithm **DynamicKP** for *MaxKP*.

Remember that **DynamicKP** computes an optimal subset $S_{opt}(I) \subseteq \{1, \cdots, n\}$ with respect to a KP-instance I with n objects in time $O(n^2 \cdot c_{\max}(I))$.

The FPTAS for *MaxKP*

Input: KP-instance $I = (n, c_1, \cdots, c_n, w_1, \cdots, w_n, W)$ and some $\epsilon > 0$.

KP-FPTAS(I,ϵ)

1 $t \leftarrow \left\lfloor \log_2 \left(\frac{\epsilon \cdot c_{\max}}{(1+\epsilon)n} \right) \right\rfloor$, where $c_{\max} \leftarrow \max\{c_1, \cdots, c_n\}$
2 **For** $i \leftarrow 1$ **to** n
3 **do** $c'_i \leftarrow \lfloor c_i \cdot 2^{-t} \rfloor$
4 **Output DynamicKP(**I'**)**, $I' = (n, c'_1, \cdots, c'_n, w_1, \cdots, w_n, W)$

Observation:

- **KP-FPTAS(I,ϵ)** outputs an **optimal solution** $S_{opt}(I') \subseteq \{1, \cdots, n\}$ **for** I'.
- $S_{opt}(I')$ is **admissible also for** I, as the weight values of I and I' are equal, but **not necessarily optimal** for I.

We estimate the **ratio and the running time** of **KP-FPTAS(I,ϵ)**.

Estimating the Ratio of KP-FPTAS(I,ϵ)

Lemma 112

It holds $\frac{c(S_{opt}(I))}{c(S_{opt}(I'))} \leq 1+\epsilon$.

Proof: As $S_{opt}(I)$ is optimal for I it holds $c(S_{opt}(I)) \geq c(S_{opt}(I'))$.

As $c'_i = \lfloor c_i \cdot 2^{-t} \rfloor$, it holds $c'_i \leq c_i \cdot 2^{-t}$, which implies $c_i \geq 2^t \cdot c'_i$, i.e.,

$$c(S_{opt}(I')) \geq 2^t \cdot c'(S_{opt}(I')) \geq 2^t \cdot c'(S_{opt}(I)) \quad \text{(as } S_{opt}(I') \text{ is optimal for } I')$$

$$= \sum_{i \in S_{opt}(I)} 2^t \lfloor c_i \cdot 2^{-t} \rfloor \geq \sum_{i \in S_{opt}(I)} 2^t (c_i \cdot 2^{-t} - 1) \geq c(S_{opt}(I)) - n \cdot 2^t.$$

Consequently,

$$c(S_{opt}(I)) - c(S_{opt}(I')) \leq n \cdot 2^t = n \cdot \left\lfloor \frac{\epsilon \cdot c_{\max}}{(1+\epsilon) \cdot n} \right\rfloor$$

$$\leq n \cdot \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)n} \leq \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)},$$

Completing the Proof of Lemma 112

$\cdots$ which implies $c(S_{opt}(I')) \geq c(S_{opt}(I)) - \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)}$.

We assume $w_i \leq W$ for all $i = 1, \cdots, n$, which implies $c(S_{opt}(I)) \geq c(\{i_{max}\}) = c_{\max}$.

Consequently,

$$c(S_{opt}(I')) \geq c_{\max} - \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)} = \frac{(1+\epsilon) \cdot c_{\max}}{(1+\epsilon)} - \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)} = \frac{c_{\max}}{(1+\epsilon)}.$$

We obtain

$$\frac{c(S_{opt}(I))}{c(S_{opt}(I'))} = \frac{c(S_{opt}(I')) + c(S_{opt}(I)) - c(S_{opt}(I'))}{c(S_{opt}(I'))}$$

$$\leq 1 + \frac{\epsilon \cdot \frac{c_{\max}}{1+\epsilon}}{\frac{c_{\max}}{1+\epsilon}} = 1 + \epsilon. \square$$

Estimating the Running Time of KP-FPTAS(I,ϵ)

Lemma 113

*The running time of **KP-FPTAS(I,ϵ)** is $O(n^3 \cdot \epsilon^{-1})$.*

Proof: The running time of **DynamicKP** on I' is $O(n^2 \cdot c'_{\max}(I'))$, where

$$c'_{\max}(I') = \left\lfloor c_{\max} \cdot 2^{-t} \right\rfloor = \left\lfloor c_{\max} \cdot 2^{-\left\lfloor \log_2\left(\frac{\epsilon \cdot c_{\max}}{(1+\epsilon)n}\right)\right\rfloor} \right\rfloor$$

$$\leq c_{\max} \cdot 2^{-\left\lfloor \log_2\left(\frac{\epsilon \cdot c_{\max}}{(1+\epsilon)n}\right)\right\rfloor} \leq c_{\max} \cdot 2^{-\left(\log_2\left(\frac{\epsilon \cdot c_{\max}}{(1+\epsilon)n}\right)-1\right)} = c_{\max} \cdot \frac{(1+\epsilon) \cdot n}{\epsilon \cdot c_{\max}} \cdot 2$$

$$= 2 \cdot n \cdot \frac{1+\epsilon}{\epsilon} \leq 4 \cdot n \cdot \epsilon^{-1}$$

if $\epsilon \leq 1$. $\square$

Example

Let the KP-instance I be defined over $n = 4$ objects with weight bound $W = 18$ and

$$c_1 = 13,200 \qquad w_1 = 7$$
$$c_2 = 21,600 \qquad w_2 = 9$$
$$c_3 = 8,400 \qquad w_3 = 4$$
$$c_4 = 4,800 \qquad w_4 = 2$$

Let $\epsilon = \frac{1}{2}$.

This implies $t = \left\lfloor \log_2 \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)\cdot n} \right\rfloor = \left\lfloor \log_2 \frac{21,600}{3 \cdot 4} \right\rfloor = \lfloor \log_2(1800) \rfloor = 10.$

Consequently,

$$c'_1 = \lfloor 13,200 \cdot 2^{-10} \rfloor = 12 \quad \text{and} \quad c'_2 = \lfloor 21,600 \cdot 2^{-10} \rfloor = 21$$

$$c'_3 = \lfloor 8,400 \cdot 2^{-10} \rfloor = 8 \quad \text{and} \quad c'_4 = \lfloor 4,800 \cdot 2^{-10} \rfloor = 4.$$

Example

Let the KP-instance I be defined over $n = 4$ objects with weight bound $W = 18$ and

$$c_1 = 13,200 \qquad w_1 = 7$$
$$c_2 = 21,600 \qquad w_2 = 9$$
$$c_3 = 8,400 \qquad w_3 = 4$$
$$c_4 = 4,800 \qquad w_4 = 2$$

Let $\epsilon = \frac{1}{9}$.

This implies $t = \left\lfloor \log_2 \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)\cdot n} \right\rfloor = \left\lfloor \log_2 \frac{21,600}{10 \cdot 4} \right\rfloor = \lfloor \log_2(540) \rfloor = 9.$

Consequently,

$$c'_1 = \left\lfloor 13,200 \cdot 2^{-9} \right\rfloor = 25 \quad \text{and} \quad c'_2 = \left\lfloor 21,600 \cdot 2^{-9} \right\rfloor = 42$$

$$c'_3 = \left\lfloor 8,400 \cdot 2^{-9} \right\rfloor = 16 \quad \text{and} \quad c'_4 = \left\lfloor 4,800 \cdot 2^{-9} \right\rfloor = 9.$$

Example

Let the KP-instance I be defined over $n = 4$ objects with weight bound $W = 18$ and

$$c_1 = 13,200 \qquad w_1 = 7$$
$$c_2 = 21,600 \qquad w_2 = 9$$
$$c_3 = 8,400 \qquad w_3 = 4$$
$$c_4 = 4,800 \qquad w_4 = 2$$

Let $\epsilon = \frac{1}{99}$.

This implies $t = \left\lfloor \log_2 \frac{\epsilon \cdot c_{\max}}{(1+\epsilon)\cdot n} \right\rfloor = \left\lfloor \log_2 \frac{21,600}{100 \cdot 4} \right\rfloor = \lfloor \log_2(54) \rfloor = 5.$

Consequently,

$$c'_1 = \left\lfloor 13,200 \cdot 2^{-5} \right\rfloor = 412 \quad \text{and} \quad c'_2 = \left\lfloor 21,600 \cdot 2^{-5} \right\rfloor = 675$$

$$c'_3 = \left\lfloor 8,400 \cdot 2^{-5} \right\rfloor = 262 \quad \text{and} \quad c'_4 = \left\lfloor 4,800 \cdot 2^{-5} \right\rfloor = 150.$$

The Nonapproximability of *MinTSP*, (1)

Theorem 114

Let $a \geq 1$ be a constant. If $P \neq NP$ then there is no polynomial time approximation algorithm for TSP of worst case approximation ratio a.

Proof: We consider the following polynomial reduction from *HC* to *TSP*:

For all $n \in \mathbb{N}$ and each undirected graph $G = (V, E)$ over $V = \{1, \cdots, n\}$ we denote by $D^G = (d^G_{i,j})^n_{i,j=1}$ a distance matrix for n cities, defined by

$$d^G_{i,j} = 1 \iff (i,j) \in E \quad \text{and} \quad d^G_{i,j} = n(a-1)+2 \iff (i,j) \notin E.$$

Obviously, D^G can be computed from G in polynomial time in n.

Observations

- $G \in HC \Longrightarrow D^G$ has a roundtrip of length n, namely this corresponding to the HC in G.
- $G \notin HC \Longrightarrow$ all roundtrips in D^G contain at least one non-edge-transition. Correspondingly, their length is at least $n - 1 + n(a-1) + 2 = an + 1$.

The Nonapproximability of *MinTSP*, (1)

Now assume that *MinTSP* has a polynomial time approximation algorithm A with worst case ratio $R_A(n) = a$.

$G \in HC \Longrightarrow A$ computes on D^G a roundtrip of length at most $a \cdot n$.

$G \notin HC \Longrightarrow A$ computes on D^G a roundtrip of length at least $a \cdot n + 1$.

Thus, A yields a polynomial time algorithm for computing HC

1. Given $G = (V, E)$, compute D^G
2. If A computes on D^G a roundtrip of length $\leq a \cdot n$ then output $G \in HC$, else output $G \notin HC$

Consequently, $P = NP$ as $HC \in NPC$. $\square$

End of Lecture