Contents

Lecture 1 (2011-10-10)**:**

Basic Graph definitions

Definition 1.1:
An *ungerichteter Graph (undirected graph)* (or short *graph*) is a pair $G = (V, E)$ consisting of a set of vertices V and a set of edges E, where each edge $e \in E$ is a two-element *unordered* subset of V.

Instead of $e \in E$ we also write $\{i, j\} \in E$ or $ij \in E$ with $i, j \in V$.

Definition 1.2:
A *gerichteter Graph (directed graph)* or short *digraph* is a pair $D = (V, A)$ consisting of a set of vertices V and a set of *Pfeil (arcs)* A, where each arc $a \in A$ is a two-element *ordered* subset of V.

Insted of $a \in A$, we also write $(i, j) \in A$ or $ij \in A$ with $i, j \in V$.

Definition 1.3:
An *Adjazenzmatrix (adjacency matrix)* or *Knoten-Knoten Inzidenzmatrix (node-node incidence matrix)* of a graph $G = (V, E)$ with $|V| = n$ nodes is a $n \times n$ matrix A containing only zeros and ones with

$$A_{ij} = \begin{cases} 1 & \text{if } \{i, j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

Definition 1.4:
An *Kante (edge)* $e = u, v$ is an unordered pair of vertices. The vertices u and v are called *Endknoten (end nodes)* of e.

Definition 1.5:
Two vertices v and w are *benachbart (Nachbarknoten) (neighbours)* in $G = (V, E)$ if an edge $e \in E$ exists with end nodes v and w.

Definition 1.6:
Two edges $e, f \in E$ with $e \neq f$ are *inzident (incident)* in $G = (V, E)$ if there exists a vertex $v \in V$ which is an end node of e as well as f.

Definition 1.7:
An edge $e \in E$ is called a *Schlinge (loop)* if both of its end nodes are identical.

Definition 1.8:
If both of the end notes of two edges $e, f \in E$ with $e \neq f$ are equal, e and f are *parallel (parallel)*. In this case the graph is called a *Multigraph (multigraph)*.

Definition 1.9:
A graph is called *einfach (simple)* if it has no loops ore parallel edges. If not stated differently, we will consider simple graphs.

Definition 1.10:
The *Menge der anliegenden Kanten (set of adjacent edges)* of a vertex $v \in V$ in a graph $G = (V, E)$ is denoted as $\delta_G(v)$ or $\delta(v)$. Where

$$\delta(v) = \{\{v, w\} \in E : \ u = v \text{ or } w = v\}.$$

Definition 1.11:
The *grad (degree)* of a vertex v (denoted $\deg(v)$) is the number of edges incident to v. Loops are counted twice.

A node v with $\deg(v) = 0$ is called *isoliert (isolated)*.

Definition 1.12:
A graph $G = (V, E)$ is called *k-regulär (k-regular)* if $\deg(v) = k$ for all vertices $v \in V$.

Definition 1.13:
For graphs $G = (V, E)$ and $H = (W, F)$, H is called a *Untergraph / Teilgraph (subgraph)* of G if $W \subseteq V$ and $F \subseteq G$

Definition 1.14:
The *induzierte Graph (induced Graph)* $G[W] = (W, E(W))$ of $G = (V, E)$ has the node set $W \subseteq V$ and all edges induced by W:

$$E(W) := \{\{v, w\} \in E : \ v \in W \text{ and } w \in W\}$$

Definition 1.15:
A *vollstäniger Graph (complete graph)* is a simple graph $G = (V, E)$ which satisfies

$$v \in V, w \in v, v \neq w \Rightarrow \{v, w\} \in E.$$

The complete graph on n vertices is denoted as W_n.

Definition 1.16:
A Cllique Q is a subset of V for which $G[Q]$ is complete.

Definition 1.17:
A *Kette / Kantenzug (walk)* in a graph is a finite sequence

$$W = (v_0, e_1, v_1, e_2, v_2, \ldots, c_k, v_k) \qquad k \geq 0,$$

that starts and ends with a node, with nodes and edges alterating each other so that every edge e_i has end nodes v_{i-1} and v_i.

Definition 1.18:
A *Weg (way)* is a chain in which all vertices are distinct.

Definition 1.19:
A *Pfad (path)* is a way in which all edges are distinct.

Definition 1.20:
A way is *geschlossen (closed)* if $v_0 = v_k$ and $k \geq 1$.

Definition 1.21:
A *Kreis (circle)* is a closed way so that all inner nodes are distinct.

Definition 1.22:
A *gewichteter Graph (weighted graph)* is a graph G with real numbers assigned to each edge (arc).

$$\text{weight matrix } (c_{ij}): \quad c_{ij} := \begin{cases} \text{weight of arc } [i,j] & \text{if } [i,j] \in E \\ \infty & \text{otherwise} \end{cases}$$

Definition 1.23:
A walk in a graph containing all edges exactly once is called an *Eulerweg (Eulerian path)*.

Definition 1.24:
A closed Eulerian path (i.e. start and end node is identical) is called an *Eulerkreis (Eulerian circuit)*.

Theorem 1.25:
A graph hat has an Eulerian circuit must have all vertices of even degree.

Proof. To have a closed walk, all vertices have to be entered and left equal number of times.

If all edges have to be different, an even number of edges incident to a vertex must be visited.

If all edges must be visited, the degree of every vertex must be even. □

Theorem 1.26:
A connected graph in which every vertex has even degree has an Eulerian circuit.

Proof. Recursive Algorithm Euler[v_1]:

```
1 if v_1 has no incident edges, then
2   return (v_1)
3 else
4   starting from v_1 create a circuit never visiting the
        same edges twice, until v_1 is reached again.
5   Let [v_1, v_2, ..., v_n, v_1] be this cicuit
6   Delete v_1, v_2, ..., v_n, v_1 from the graph
7   return (Euler(v_1), ..., Euler(v_n), v_1)
```

□

Lecture 2 (2011-10-13)**:**

Graph basics

Definition 2.1 (connected)**:**
A graph is called *zusammenhängend (connected)* if there exists a [s,t]-Path between all pairs of vertices $s, t \in V$.

Definition 2.2 (forrest, tree, spanning, forest problem, minimum spanning tree)**:** A *Wald (forest)* is a graph that does not contain a cycle (Kreis). A connected forest is called a *Baum (tree)*. A tree in a graph (as subgraph) is called *aufspannend (spanning)*, if it contains all vertices.

Given a graph $G = (V, E)$ with edge weights $c_e \in \mathbb{R}$ for all $e \in E$, the task to find a forest $W \subset E$ such that

$$c(W) := \sum_{e \in W}$$

is maximal, is called the *Problem des maximalen Waldes (Maximum Forest Problem)*. The task to find a tree $T \subset E$ which spans G and which weight $c(T)$ is minimal, is called the *minimaler Spannbaum (Minimum Spanning Tree (MST) problem)*.

Lemma 2.3:
A tree $G = (V, E)$ with at least 2 vertices has at least 2 vertices of degree 1.

Proof. Let v be arbitrary. Since G is connected, $\deg(v) \geq 1$. Assume $\deg(v) = 1$. So $\delta(v) = \{vw\}$. If $\deg(w) = 1$, we found two vertices with degree 1. If $\deg(w) > 1$, there exist a neighbour of w, different from v which we call u. Now, again u has degree 1 or higher. If we repeat this procedure we either find a vertex of degree 1 or find again *new* vertices. Hence, after at most $n - 1$ vertices we end up at a vertex of degree 1. Now, if $\deg(v) \geq 2$, we do the same and find a vertex of degree 1, say w. Then repeat the above, staring from w to find a second vertex of degree 1. □

Corollary 2.4:
A tree $G = (V, E)$ with maximum degree Δ has at least Δ vertices of degree 1.

Lemma 2.5: (a) For every graph $G = (V, E)$ it holds that $2|E| = \sum_{u \in V} \deg(u)$

(b) for every tree $G = (v, E)$ it holds that $|E| = |V| - 1$.

Proof. (a) trivial

(b) Proof by induction. Clearly, if $|V| = 1$ or $|V| = 2$ it holds. Assumption: true for $n \geq 2$. Let G be a tree with $n + 1$ vertices. By Lemma 2.3, there exists a vertex $v \in G$ with $\deg(v) = 1$. $G - v = G[V \setminus \{v\}]$ is a tree again with n vertices and thus $|E(G - v)| = |V(G - v)| - 1$. Since G differs by one vertex and one edge from $G - v$, the claim holds got G as well.

□

Lemma 2.6:
If $G = (V, E)$ with $|V| \geq 2$ has $|E| < |V| - 1$, G is not connected.

MST and Shortest Path Algorithms

$\min_{x\in X} = -\max_{x\in X} -c(x)$ ~~maximal forest~~
X spanning trees
$\min_{x\in X} +(n-1)D = -\max_{x\in X} -c(x)(n-1)D = \max_{x\in X} \sum \underbrace{D - C_{ij}x_{ij}}_{\geq 0 if D \geq \max_{ij\in E} c_{ij}}$

Theorem 2.7:
Kruskal's Algorithm returns the optimal solution.

Proof. Let T be Kruskal's tree and assume there exists a tree T' with $c(T') < c(T)$. Then there exist an edge $e' \in T' \setminus T$. Then $T \cup \{e'\}$ contains a cycle $\{e_1, e_2, \ldots, e_k, e'\}$. Let $c_f = max_{i=1,\ldots k} c_{l_i}$. At the moment Kruskal chooses edge f, edge e' cannot be added yet and therefore $c(e') \geq c(f)$. Now exchange e' by f in T'. Hence the number of differences beetween T' and T is reduced by one, $C(T'_{new}) \leq c(T') < c(T)$. Repeating the procedure results in $c(T) \leq \ldots < c(T)$, a contradiction. □

Lecture 3 (2011-10-17)**:**

Definition 2.7(+1):
The *Laufzeit (running time of algorithms)* of an algorithm is measured by the number of operations needed in worst case of a function of the input size. We use the $O(\cdot)$ notation (Big-O-notation) ot focus on the most important factor of the running time, ignoring constants and smaller factors.

Example 2.7(+2):
If the running time is $3n \cdot \log n + 26n$, the algorithm runs in $O(n \cdot \log n)$. If the running time is $3n \cdot \log n + 25n^2$, the algorirthm runs in $O(n^2)$.

For graph Problems, the running is expressed in the number of vertices $n = |V|$ and the number of edges $m = |E|$. Sometimes m is approximated by n^2.

Example 2.7(+3) (Kruskal's Algorithm)**:**
First, the edged are sorted according to nondecreasing weights. This can be done in $O(m \cdot \log m)$. Next, we repeatedly select an edge or reject its selection until $n-1$ edges are selected. Since the last selected edge might be after m steps, this routine is performed at most $O(m)$ times.

Checking whether the end nodes of $\{u, v\}$ are already in the same tree can be done in constant time, if we label the vertices of the trees selected so far: $r(u) = \#trees\, containing\, u$. If $r(u) \neq r(v)$, the trees are connected by $\{u, v\}$ to a new tree.

Without going into details, the resetting of labels in one of the old trees, can be done $O(\log n)$ on average. Since this update has to be done at most $n-1$ times, it takes $O(n \cdot \log n)$.

Overall, Kruskal runs in

$$O(n \log m + m + n \cdot \log n) = O(m \cdot \log m) = O(m \cdot \log n^2) = O(m \cdot \log n)$$

Definition 2.7(+4) (Shortest paths in acyclic digraphs)**:**
A directed graph (digraph) $D = (V, A)$ is called *azyklisch (acyclic)* if it does not contain any *(directed cycles)*, i.e. a *Kette (chain)* $(v_0, a_1, v_1, a_2, v_2, ...a_k, v_k)$, $k \geq 0$, with $a_i(v_{i-1}, v_i) \in A$ and $v_k = v_0$. In particular, D does not contain *entgegengesetzt (antiparallel)* arcs: if $(u, v) \in A$, $(v, u) \notin A$. With $\delta_D^+(v)$ we denote the arcs leaving vertex v:

$$\delta_D^+(v) = \{(u, w) \in A : u = v\}$$

similarly:

$$\delta_D^-(v) = \{(u, w) \in A : w = v\}$$

are the arcs entering v.

The *Ausgangsgrad (outdegree)* of v is $\deg_D^+(v) = |\delta^+(v)|$ (assuming simple digraph)

The *Eingangsgrad (indegree)* of v is $\deg_D^-(v) = |\delta^-(v)|$

Definition 3.1:
The *(shortest path)* problem in a acyclic digraph is, given an acyclic digraph $D = (V, A)$, a length function $C : A \to \mathbb{R}$ and two vertices $s, t \in V$, find a $[s, t]$-path of minimal length.

Question 3.1.1:
Does there exist a $[s, t]$-path at all?

Theorem 3.2:
A digraph $D = (V, A)$ is acyclic, if and only if there exists a permutaion $\sigma : V \to \{1, .., n\}$ of the vertices such that $\deg^-_{D[v_1,...,v_n]}(v_i) = 0$ for all $i = 1, ..., n$ with $v_i = \sigma^{-1}(i)$.

Proof. By induction:

For digraph with $|V| = 1$, the statement is true. Assume the statement is true for all digraphs with $|V| \leq n$ and consider $D = (V, A)$ acyclic with $n + 1$ vertices. If there does not exist a vertex with $\deg_D^-(v) = 0$, a directed cycle can be detected by following incoming arcs backwards until a vertex is repeated, a contradiction regarding the acyclic property of D.

Hence, let v be a vertex with $\deg_D^-(v) = 0$. Set $v_1 = v$. The digraph $D - v_1$ has n vertices and is acyclic, and thus has a permutation $(v_2, ..., v_{n+1})$ with

$$\deg^-_{D[v_i,...,v_{n+1}]}(v_i) = 0 \qquad \forall i = 2, ..., n + 1$$

Now, $(v_1, ..., v_{n+1})$ is a permutation fulfilling the condition.

In reverse, if there exists a permutation $(v1, ..., v_{n+1})$, $\deg_D^-(v_1) = 0$ and there cannot exist a directed cycle containing v_1. By induction, neither cycles containing v_i, $i = 2, ..., n + 1$ exist. □

Theorem 3.3:
A $[s, t]$-path exists in a acyclic Digraph $D = (V, A)$ if and only if in all permutations $\sigma : V \to \{1, ..., n\}$ with $\deg^-_{D[v_i,...,v_n]}(v_i) = 0$ for all $i = 1, ..., n$, it holds that $\sigma(s) < \sigma(t)$.

Proof. Assume there exists a permutation σ with $\sigma(s) > \sigma(t)$. Since outgoing arcs only go to higher ordered vertices, there does not exist a path from s to t in D.

In reverse, if there does not exist a path from s to t, we order all vertices with paths to t first, followed by t and s afterwards. □

Question 3.3.1:
How do we find the shortest $[s, t]$-path if it exists?

To simplify notation, let $V = \{1, ..., n\}, s = 1, t = n$ and $(i, j) \in A \Rightarrow i < j$. Let $D(i)$ be the distance from i to n and $NEXT(i)$ be the next vertex on the shortest path from i to n.

Bellman's Algorithm

1 $D(i) = \{\infty : i < n and NEXT(i) = NIL, 0 : i = n\}$
2 `FOR` $i = n - 1$ `DOWNTO 1 DO`
3 $D(i) = \min_{j=j+1,...,n}\{D(j) + c(i, j)\}$ `with` $c(i, j) = \infty$ `if` $(i, j) \notin A$
4 $NEXT(i) = \text{Argmin}_{j=i+1,...,n}\{D(j) + c(i, j)\}$

Theorem 3.4:
Bellman's Algorithm is correct and runs in $O(m + n)$ time.

Proof. Every path from 1 to n passes through vertices of increasing ID. Assume there exists a path $(a_1, ..., a_k)$ with $\sum_{i=1}^{k} c(a_i) < D(1)$. Let $a_1 = (1, j_1)$. Since $D(1) \leq c(a_1) + D(j_1)$, it should hold that

$$\sum_{i=2}^{2} c(a_i) < D(j_1)$$

But $D(j_1) \leq c(a_2) + D(j_2)$ with $a_2 = (j_1, j_2)$, etc.
In the end, $c(a_k) < D(j_{k-1})$ but $D(j_{k-1}) \leq c(a_k) + D(n) = c(a_k)$, contradiction. □

Lecture 4 (2011-10-20)**:**

Theorem 3.5:
Bellman's Algorithm is correct and runs in $O(m + n) = O(n)$.

Proof. of runtime:

$$D(i) = \min_{(i,j)\in A} D(j) + D(i, j)$$

⇒ Every arc is considered once, and thus overall $O(m)$ computations are needed. Initialization costs $O(n)$. □

Bemerkung 3.5(+1):
The running time does not contain the time to find the permutation.

Observation 1: We not only found the shortest path from 1 to n, but also from i to n, $i = 2, ..., n$.
Observation 2: We can use a similar procedure for the shortest path from 1 to i, $i = 2, ..., n$. (with $PREV(i)$ for previous instead of $NEXT(i)$).

Question 3.5.1:
Can we find a shortest path from 1 to i in a digraph that is not acyclic, i.e. it contains cycles?

Theorem 4.1:
The Moore-Bellman-Algorithm returns the shortest paths from 1 to $i = 1, ..., n$ provided D does not contain negative-weighted directed cycles.

Proof. We call an arc $(i, j) \in A$ an *upgoing* arc (Aufwärtsbogen) if $i < j$ and a *downgoing* arc (Abwärtsbogen) if $i > j$.
A shortest path from 1 to i contains at most $n - 1$ arcs. If an upgoing arc is followed by a downgoing arc (or vice versa), we have a *change of direction* (Richtungswechsel). With at most $n - 1$ arcs, at most $n - 2$ changes of direction are possible.
Let $D(i, m)$ be the value of $D(i)$ at the end of the m-th iteration. We will show (and this is enough):

$$D(i, m) = min\{c(W) : W \text{ is the directed } [1, i]\text{-path with at most } m \text{ changes of directions}\}$$

We prove it by induction on m.

- For $m = 0$, the algorithm is equivalent to Bellman's algorithm for acyclic grpahs. Thus, $D(i, 0)$ is the length of the shortest path without any changed of direction.
- Now, let us assume, that the statement is true for $m \geq 0$ and the subroutine is executed for the $m+1$-st time. The set of $[1, i]$-paths with at most $m+1$ changes of direction consists of
  (a) $[1, i]$-paths with $\leq m$ changes of direction
  (b) $[1, i]$-paths with exactly $m + 1$ changes of direction

  $\Rightarrow D(i, m)$
- Since every path starts with an upgoing arc $(1, k)$, the last arc after $m + 1$ changes is either a downgoing arc if $m + 1$ is odd or an upgoing arc if $m + 1$ is even. We restrict ourselves to $m + 1$ odd ($m + 1$ even is similar).

  To compute the minimum length path in (b) we use an additional induction on $i = n, n - 1, ..., j + 1$. Since every path ending at n ends with an upgoing arc, there do not exist such $[1, n]$-paths. Hence, $D(n, m + 1) = D(n, m)$.

Now assume that $D(k, m+1)$ is correctly computed for $i \leq k \leq n$. The shortest path from 1 to $i-1$ with exactly $m+1$ changes ends with a downgoing arc $(j, i-1)$, $j > i-1$.

$D(j, m+1)$ is already computed correctly. If $PREV(j) > j$, no change of direction is required in j and $D(i-1, m+1) = D(j, m+1) + c(j, i-1)$ If $PREV(j) < j$, the last avec of the $[1, j]$-Path is upgoing, and thus $D(i-1, m+1) = D(j, m) + c(j, i)$. The last change of direction at j is thus, in worst case, the $(m+1)$-st change. Hence, $D(i-1, m+1)$ fulfills the statement.

□

Remark 1:
In fact, the algorithm finds the minimum length of a chain (kette) with at most $n-2$ changes of direction. In case of negative weighted cycles these might be in a chain several times.

In case no negative weighted cycles exist, the min. length chains are indeed paths. Hence, the algoithm only works correctly if *all* cycles are non-negative weighted.

Remark 2:
If a further executing of the subroutine ($m = n-1$) results in at least one change of a value $D(i)$, then the digraph contains negative weighted cycles.

Remark 3:
A more efficient implementation is given by E'sopo-Pape-Variant.

Dijkstra's Algorithm for non-negative weights

Theorem 4.2:
Dijkstra returns the shortest paths from 1 to i, $i = 1...n$, provided all weights ≥ 0.

Proof. Each step, one vetex is moved from T to S. At the end of a step, $D(j)$ is the shortest path from 1 to j via vertices in S.
If $S = V(T = \emptyset)$, $D(i)$ is thus the shortiest $[1, i]$-path □

Lecture 5 (2011-10-24)**:**

Shortest paths between all pairs of vertices

Solution 1: Apply Moore-Bellman or Dijkstra to all vertices i as starting vertex

Solution 2: Apply Floyd's Algorithm

Notation:

w_{ij} is the length of the shortest $[i, j]$-path, $i \neq j$

w_{ii} is the length of the shortest directed cycle containing i

p_{ij} is the predecessor of j on the shortest $[i, j]$-path (cycle)

$W = (w_{ij})$ is the *??? (shortest path length matrix)*

Theorem 5.1:
The Floyd Algorithm works correctly if and only if $D = (V, A)$ does not contain any negative weighted cycles.

D contains a negative weighted cycle if and only if one of the diagonal elements $w_{ii} < 0$.

Proof. Let W^k be the matrix W after iteration k, with W^0 being the initial matrix. By induction on $k = 0, ..., n$ we show that W^k is the matrix of shortest path lengths with vertices $1, ..., k$ as *possible* internal vertices, provided D does not contain a negative cycle on these vertices.

If D has a negative cycle, then $w_{ii}^k < 0$ for an $i \in \{1, ..., n\}$

For $k = 0$, the statement clearly true.

Assume, it is correct for $k \geq 0$, and we have executed the $(k+1)$st iteration.

It holds that $w_{ij}^{k+1} = \min\{w_{ij}^k, w_{i,k+1}^k + w_{k+1,j}^k\}$. Note that, provided no negative cycle exists, $w_{i,k+1}^{k+1}$ does not have any vertex $k+1$ as internal vertex, and thus $w_{i,k+1}^{k+1} = w_{i,k+1}^k$ (similarly, $w_{k+1,j}^{k+1} = w_{k+1,j}^k$).

$w_{i,k+1}^k$ is the minimal length of a $[i, k+1]$-path with $\{1, ..., k\}$ as allowed internal vertices. Similarly, $w_{k+1,j}^k$.

Thus, $w_{i,k+1}^k + w_{k+1,j}^k$ is the minimal length of an $[i, j]$-path (not necessarily simple) containing $k+1$ (mandatory) and $\{1, ..., k\}$ (voluntary). If the shortest path from i to j using $\{1, ..., k+1\}$ does not contain $k+1$, it only contains $\{1, ..., k\}$ (voluntary) and, hence, w_{ij}^k is the right value.

What remains to show is that the connection of the $[i, k+1]$-path with the $[l+1, j]$-path is indeed a simple path.

Let K be this chain. After removal of cycles, the chain K contains (of course) a simple $[i, j]$-path $\bar{K}$. Since such cycles may only contain vertices from $\{1, ..., k+1\}$, one cycle must contain $k+1$. If this cycle is not negatively weighted, then path $\bar{K}$ is shorter and $w_{ij} < w_{i,k+1}^k + w_{k+1,j}^k$.

If this cycle is negatively weighted, $w_{k+1,k+1}^k < 0$ (the cycle only contains internal vertices from $\{1, ..., k\}$) and algorithm would have stopped earlier. □

Min-Max-Theorems for combinatorial Optimization Problems

From "Optimierung A": Duality of linear programs

$$\max_{\text{s. t.}, Ax \leq b, x \geq 0} c^T x = \min_{\text{s.t.}, A^T y \geq c, y \geq 0} b^T y$$

For several combinatorial problems $\min\{c(x) : x \in X\}$

We can define a second set Y and a function $b(y)$ with $\max\{b(y) : y \in Y\} = \min\{c(x) : x \in X\}$ where Y and b(y) have a graph theoretical interpretation.

Existence of such a "Dual" Problem indicates often that the problem can be solved "efficiently". For the shortest path problem several max-min-theorems exist.

Definition 5.2:
An *(s,t)-Schnitt ((s,t)-cut)* in a digraph $D = (V, A)$ with $s, t \in V$ is a subset $B \subset A$ of the arcs with the property that every (s,t)-path contains at least one arc of B.

Stated otherwise, for every cut B, there exists a vertex set $W \subset V$ such that

- $s \in W,\ t \in V \setminus W$
- $\delta^+(w) = \{(i,j) \in A : i \in W, j \in V \setminus W\} \subseteq B$

Theorem 5.3:
Let $D = (V, A)$ be a digraph, $c(a) = 1\, \forall a \in A, s, t \in V, s \neq t$. Then the minimum length of a $[s,t]$-path equals the maximum number of arc-disjoint (s,t)-cuts.

Proof. Folows from 5.4 □

Theorem 5.4:
Let $D = (V, A)$ be a digraph, $c(a) \in \mathbb{Z}_+\, \forall a \in A \wedge s, t \in V \wedge s \neq t$. Then the min length of an $[s,t]$-path equals the maximum number d of (not necessarily different) (s,t)-cuts $C_1, ..., C_d$ such that every arc $a \in A$ is contained in at most $c(a)$ cuts.

Proof. We define (s,t)-cuts $C_i = \delta^+(v_i)$ with

$$
\begin{aligned}
v_i &= \{v \in V : \exists (s,v)\text{-path with } c(P) \leq i-1\} \\
v_1 &= \{s\} \\
v_2 &= \{5,3,4\} \\
v_3 &= \{5,2,3,4\} \\
v_4 &= v_3 \cup \{6\}
\end{aligned}
$$

(for the example graph on the board)

The shortest $[s,t]$-path P consists of arcs $a_1, ...a_k$ with arc a_j contained in (s,t)-cuts C_i, $i \in \{\sum_{l=1}^{j-1} c(a_l) + 1, ..., \sum_{l=1}^{j} a(a_l)\}$: exactly $c(a)$ cuts. □

Lecture 6 (2011-10-26)**:**

Knapsack problem

Definition 6.1:
The *Knapsack Problem (Knapsack problem)* is defined by a set of items $N = \{1, ..., n\}$ weights $a_i \in \mathbb{N}$, value $c_i \in \mathbb{N}$, and a bound $b \in \mathbb{N}$. We search for a subset $S \subset \mathbb{N}$ such that

$$a(S) = \sum_{i \in S} a_i \leq b \text{ and } c(S) = \sum_{i \in S} c_i \text{ maximum}$$

Approach 1:
Greedy algorithm

Idea: Items with small weight but high value are the most atrractive ones.

Procedure:

```
1 Sort the items such that c_1/a_1 ≤ c_2/a_2 ≤ ... ≤ c_n/a_n .
2
3 Set S = ∅.
4 For i = 1 to n do
5         if (a(s) + a_i ≤ b) then
6                 S = S ∪ {i}
7         endif
8 endfor
9 return S and c(S)
```

Theorem 6.2:
The greedy algorithm does *not* guarantee an optimal solution.

Proof. Let $b = 10$, $n = 6$

i	2	3	4	5	6
a_i	9	2	2	2	2
c_i	19	4	4	4	4

Greedy: $S = \{1\}$, $c(s) = 20$

Optimal: $S = \{2, 3, 4, 5, 6, \}$, $c(S) = 20$ □

Approach 2: Integer Linear Programming

The set of solutions X of a combinatorial optimization problem can (almost always) be written as the intersection of integer points in $\mathbb{N}_0^n$ and a polyhedron $\{x \in \mathbb{R}^n : Ax \leq b\}$

Let $x \in \{0, 1\}^n$ be a vector representing all solutions of the knapsack problem:

$$x_i = \begin{cases} 1 & \text{if} i \in S \\ 0 & \text{otherwise} \end{cases}$$

$X = \{0, 1\} \cap \{x \in \mathbb{R}^n : \sum_{i=0}^{n} a_i x_i \leq b\}$

Knapsack: $\max \sum i = 0^n c_i x_i$

The *Lineare Relaxierung (linear relaxation)* of an ILP is the linear program optained by relaxing the integrality of the variables:

$\max \sum_{i=1}^{n} c_i x_i$

s. t. $\sum_{i=1}^{n} a_i x_i \leq b, 0 \leq x_i \leq 1 \qquad \forall i \in \{1, ..., n\}$

Theorem 6.3:
An optimal solution $\tilde{x}$ of the linear relaxation of the knapsack problem is:

There exists a $k \in \{1, ..., n\}$ such that

$$\tilde{x}_i = \begin{cases} 1 & \text{if } i \leq k \\ 0 & \text{if } i > k+1 \\ (b - \sum_{i=1}^{k} a_j)/a_{k+1} & \text{if } i = k+1 \end{cases}$$

where $c_1/a_1 \geq c_2/a_2 \geq ... \geq c_n/a_n$.

Proof. Let x^* be an optimal solution with $c^T c^* > c^T \tilde{x}$. If $x_i^* < 1$ for $i \leq k$, there must exist a $j \geq k+1$ with $x_j^* > \tilde{x}_j$.
We define $\bar{x}$ with $\varepsilon \leq x_j^* - \tilde{x}_j$ as

$$\bar{x}_l = \begin{cases} x_k^* & \text{for } l \notin \{i, j\} \\ x_l^* - \varepsilon & \text{for } l = j \\ x_l^* + \frac{a_j}{a_l} \cdot \varepsilon & \text{for } l = i \end{cases}$$

Then $\bar{x}$ is feasible and

$$c^T \bar{x} = \sum_{l=1}^{n} c_l \bar{x}_l = \sum_{l=1}^{n} c_l x_l^* + \underbrace{c_i \cdot \frac{a_j}{a_i} \varepsilon - c_j \varepsilon}_{\geq 0} \geq c^T x^* > c^T \tilde{x}$$

Repetition yields $c^T \bar{x} > c^T \bar{x}$, a contradiction. □

Note:

If $\tilde{x}$ is integer valued, then the solution is also optimal for the knapsack problem. In this case, also the greedy algorithm is optimal.

Approach 3: Dynamic Programming

A dynamic program algorithm to solve a problem first solves similar, but smaller subproblems in order to use their solution to solve the original problem.

The problem should conform to the *optimality principle* of Bellman: Given an optimal solution for the original problem, a partial solution restricted to a subproblem is also optimal for the subproblem.

Let $f_k(b)$ be the optimal solution value of the knapsack problem with total weight equal to b and items from $\{1, ..., k\}$.

Theorem 6.4:
$f_{k+1}(b) = max\{f_k(b), f_k(b - a_{k+1} + c_{k+1}\}$.

Proof. An optimal solution of $f_{k+1}(b)$ either contains item $k+1$ or not. If $k+1$ is not contained, the problem is identical to $f_k(b)$. If $k+1$ is contained, other items in the solution should have total weight $b - a_{k+1}$.
Hence, $f_k(b - a_{k+1})$ is an optimal solution for the remaining items $+c_{k+1}$ for the item $k+1$. □

Corollary 6.5:
The knapsack problem can be solved in $O(nb)$ with value $max_{d=0,...,b} f_n(d)$.

Lecture 7 (2011-10-31)**:**

Matchings, Stable Sets, Vertex Covers and Edge Covers

Definition 7.1:
Let $G = (V, E)$ be an undirected graph. A *Paarung (matching)* is a subset $M \subset E$ such that $e \cap e' = \emptyset$ for all $e, e' \in M$ with $e \neq e'$.

A matching M is called a *Perfektes Matching (perfect matching)* if all vertices are incident to some edge in M.

Example 7.1(+1):
An airline has to allocate two pilots to each of the (round)trips on a single day. Only certain pairs of pilots can work together, due to experience, qualification, location, etc. By defining a graph with vertex set the pilots and edges if two pilots can work together, a daily corresponds to a matching since a pilot can work at two trips at the same time.

Definition 7.2:
A matching $M \subseteq E$ is called *maximal* if no further edges can be added.

A *maximum* matching is a matching M with maximal cardinality, i.e. no other matching M' exists with $|M'| > |M|$.

$\nu(G) =$ *Paarungszahl (matching number)*, size of a maximum matching.

Further related graph parameters

$\alpha(G) =$ *Stabile-Mange-Zahl (Stable set number / independent set number)*, size of a maximum stable set in G: a subset $S \subseteq V$ such that

$$|\{v, w\} \cap S| \leq 1 \quad \text{for all} \quad \{v, w\} \in E$$

$\rho(G) =$ *Kantenüberdeckungszahl (Edge cover number)*, minimum size of an edge cover of G: a subset $F \subseteq E$ such that

$$\forall v \in V : \exists e \in F : e = \{v, w\}.$$

$\tau(G) =$ Vertex *Knotenüberdeckungszahl (cover number)*, minium size of a vertex cover of G: a subset of $W \subseteq V$ such that

$$e \cap W \neq \emptyset \quad \text{for all} \quad e \in E :$$

.

Lemma 7.3:
$\rho(G) = \infty$ if and only if G contains isolated vertices.

Proof. If G contains isolated vertices, such vertices cannot be covered by any edge, hence $\rho(G) = \infty$. If G does not contain isolated vertices, then E is an edge cover itself, thus $\rho(G) \leq |E|$. □

Lemma 7.4:
$S \subseteq V$ is a stable set if and only if $V \setminus S$ is a vertex cover.

Proof. Exercise sheet. □

Lemma 7.5:
$\alpha(G) \leq \rho(G)$.

Proof. If $\rho(G) = \infty$, then $\alpha(G) \leq \rho(G)$ follows automatically. If $\rho(G) < \infty$, then every vertex has degree at least one. Let F be an edge cover in G. Since the vertices of a stable set are not adjacent, there must exist an edge $e \in F$ for all $v \in S$ such that $v \in e_v$ and $e_v \neq e_w$ for all $v, w \in S, v \neq w$. Hence, $|F| \geq |S|$ and it follows $\alpha(G) \leq \rho(G)$. □

Lemma 7.6:
$\nu(G) \leq \tau(G)$

Proof. To cover all edges of a matching M by vertices, we need at last $|M|$ vertices. Hence, $\nu(G) \leq \tau(G)$. □

Theorem 7.7 (Gallai's Theorem)**:**
For every graph $G = (V, E)$ without isolated vertices, it holds that

$$\alpha(G) + \tau(G) = |V| = \nu(G) + \rho(G)$$

Proof. $\alpha(G) + \tau(G) = |V|$ follows directly from Lemma 7.4.

To show $\nu(G) + \rho(G) = |V|$, consider a maximum matching M ($|M| = \nu(G)$). M covers all $2|M|$ vertices in M. For every vertex not covered by M, add an incident edge to M. The resulting edge cover has

$$|M| + (|V| - 2|M|) = |V| - |M| = |V| - \nu(G)$$

edges. Hence, $\rho(G) \leq |V| - \nu(G)$.

Now, let F be an edge cover with $|F| = \rho(G)$. Remove for every vertex $v \in V$ $deg_F(v) - 1$ edges incident to v. The resulting edge set is a matching with at least

$$\nu(G) \geq |F| - \sum_{v \in V} (deg_F(v) - 1) = |F| - 2|F| + |V| = |V| - |F| = |V| - \rho(G)$$

□

For all graph parameters, there exists a weighted version:

$$\nu(G, w), \tau(G, w), \alpha(G, w), \rho(G, w)$$

Question 7.7.1:
How do we find a maximum (weighted) matching?

Definition 7.8:
Let M be a matching in G. A path $P = (v_0 e_1 v_1 ... e_r v_r)$ in G is called *M-alternating (M-alternierend)*, if M contains either all edges e_i with i even or all edges e_i with i odd.

A M-alternating path P is called *M-augmenting (M-augmentierender)* path if v_0 and v_r are not matched in M, i.e. $v_0, v_r \notin \bigcup_{e \in M} e$.

Lemma 7.9:
If P is M-augmenting, then r odd and

$$M' = M \setminus \{e_2, e_4, ..., e_{r-1}\} \cup \{e_1, e_3, ..., e_r\}$$

is a matching with $|M'| = |M| + 1$.

Proof. Trivial. □

Lemma 7.10:
(Berge)
Let $G = (V, E)$ be a graph and M a matching in G. Then either M is a maximum matching or there exists a M-augmenting path.

Lecture 8 (2011-11-04)**:**

Matchings in bipartite graphs

Theorem 8.1:
(Berge) Let $G = (V, E)$ be a graph, M a matching in G. Then either M is a maximum matching or there exists an M-augmenting path.

Proof. If M is a maximum matching, no M-augmenting path can exist, since M' would be a larger matching.
Let $\bar{M}$ a matching with $|\bar{M}| > |M|$. Consider the *Komponente (components)* of $G' = (V, M \cup \bar{M})$. Since the the degree of vertices in (V, M) and $(V, \bar{M})$ is at most one, the degree in G' is at most two. Thus, each component of G' is either a path (possibly of length zero) or a cycle. Since $|\bar{M}| > |M|$, at least one component of G' has to have more edges from $\bar{M}$ than from M. Such a component cannot be a cycle and thus is a path, better an M-augmenting path since the end nodes are not matched in M. □

Definition 8.2:
A graph $G = (V, E)$ is called *bipartit (bipartite)* if and only if $V = U \cup W (U \cap W = \emptyset)$ such that $\{v, w\} \in E \Rightarrow v \in U, w \in W$ (or vice versa). The set U and W are called the color classes of V.

Example 8.2(+1):
n workers, m Jobs. Not every worker is qualified for every job. How many jobs can be processed simultaneously? U = set of worker. W = set of jobs, $\{u_i, w_j\} \in E \Leftrightarrow$ worker i is qualified for job j.

$\nu(G) \leq \tau(G)$ vertex cover

Theorem 8.3:
(König's Matching Theorem, 1931) For every bipartite graph $G = (V, E) : \nu(G) = \tau(G)$

Proof. $\nu(G) \leq \tau(G)$ holds by Lemma 7.6. We therefore only show $\nu(G) \geq \tau(G)$. We may assume that G has at least one edge (otherwise $\nu(G) = \tau(G) = 0$). We will show that G has a vertex v that is matched in every maximum matching. Let $\{v, w\} = e$ be an arbitrary edge in G and assume that M and N are two maximal matchings with u not matched in M and v not matched in N. Define P as the component of $(V, M \cup N)$ containing u. Since u is only matched in N, $deg_P(u) = 1$ and thus u is an end node of the path P. Since M is maximum, P is *not* an M-augmenting path. Hece, the length of P is even. Consequently, P does not contain v (otherwise P ends at v which contradict the bipartiteness of $G : P \cup \{u, v\}$ would be an odd cycle). The path $P \cup \{u, w\}$ is thus odd, starts with vertex v not matched in N and ends with another vertex not matched in N. Hence $P \cup e$ is N-augmenting path; contracdiction since N is a maximum matching.
So, either u or v must be contained in all maximum matchings, let's say v. Now, consider $G' := G - v$. It holds that $\nu(G) = \nu(G) - 1$. By induction on $n = |V|$ we may assume that G' has a vertex cover W with $|W| = \nu(G')$. Then $W \cup \{v\}$ is a vertex cover of G of size $\nu(G') + 1 = \nu(G)$. It follows $\tau(G) \leq \nu(G)$. □

Corollary 8.4:
(König's Edge Cover Theorem) Every bipartite graph has $\alpha(G) = \rho(G)$.

Proof. Foolows from Thm 7.7(Gallai's Thm) and 8.3 □

Matching augmenting algorithm for bipartite graphs

Input:bipartite graph $G = (V, E)$ and a matching M

Output: matching M' with $|M'| > |M|$ (if it exists)

Description: Let U, W be the color classes of G. Orientate every edge $e = \{u, w\}(u \in U, w \in W)$ as follows:
if $e \in M$, then orientate from w to u
if $e \notin M$, then orientate from u to w
Let D be the digraph constructed this way. Consider

$$U' := U \setminus \bigcup_{e \in M} e \quad \text{and} \quad W' := W \setminus \bigcup_{e \in M} e$$

An M-augmenting path exists if and only if a directed path in D exists starting at a vertex U' and ending at a vertex W'. Augment M with this path return M'.

Theorem 8.5:
A maximum matching can be found by applying at most $\frac{1}{2}|V|$ times the above algorithm.

Proof. Thm 8.2 says that either a maximum matching or an M-augmenting path exists. This path can be found by the digraph as it has to start with an unmatched vertex and alternates between matched and not matched edges. Thes is guaranteed by the direction of the edges in the digrpah.. If we start with $M = \emptyset$, the size increases by one in every iteration. A max matching can at most $\frac{1}{2}|V|$ edges. □

Lecture 9 (2011-11-07)**:**

Corollary 9.1 (Frobenius theorem)**:**
A bipartite graph $G = (V, E)$ has a perfect matching, if and only if all the vertex covers contain at least $\frac{1}{2}|V|$ nodes.

Proof. Let $\tau(G) \geq \frac{1}{2}|V|$ hold. We derive $\nu(G) \geq \frac{1}{2}|V|$ from König's theorem, but $\nu(G) \leq \frac{1}{2}|V|$ holds as well. If $\tau(G) < \frac{1}{2}|V|$ holds, $\nu(G) < \frac{1}{2}|V|$ holds as well and therefore no perfect matching exists. □

Corollary 9.2:
Every regular($\deg(v) = \deg(w)\ \forall v, w \in V$) bipartite graph, with positive degree, has a perfect matching.

Proof. exercise □

Theorem 9.3 (Hall's theorem (Hochzeitssatz))**:**
Let $G = (U \cup W, E)$ be a bipartite graph with classes U, W. For each subset $X \subseteq U$, $N(X)$ describes the set of nodes in W, which are adjacent to a node in X (neighborhood)

$$\nu(G) = |U| \Leftrightarrow |\nu(X)| \geq |X|\ \forall X \subseteq U$$

Proof. necessity. If $|N(X)| < |X|$ holds, then a matching M can contain at most $|N(X)|$ edges with ending nodes in X. Therefore noes remain in X which cannot be covered by a matching.

Sufficient: Assume $\nu(G) < |U|$, but $|N(X)| \geq |X|\ \forall X \subseteq U$. Then there exists a vertex cover Z for G with $|Z| < |U|$.

Define $X := U \setminus Z$. Then it follows that $N(X) \subseteq W \cap Z$ and therefore

$$|X| - |N(X)| \geq \underbrace{|U| - |U \cap Z|}_{=|X|} - \underbrace{|W \cap Z|}_{\geq |N(X)|} = |U| - |Z| > 0$$

produces a contradiction □

Maximum Weighted Matching on bipartite graphs

Let $G = (V, E)$ be a graph and let $\omega : E \to \mathbb{R}$ be a weight function For any subset M of E define the weight $\omega(M)$ of M by

$$\omega(M) = \sum_{e \in M} \omega(e)$$

Definition 9.3(+1):
We call a matching extreme if it has maximum weight among all matchings of *Kardnialität $|M|$ (cardinality $|M|$)*

$$l(e) = \begin{cases} w(e), & \text{if } e \in M \\ -w(e), & \text{if } e \notin M \end{cases}$$

Lemma 9.4:
Let $P\{e_1, e_2, ..., e_r\}$ be an M-augmenting path of minimal length. If M is extreme, then $M' = M \setminus \underbrace{\{e_2, ..., e_{n-1}\}}_{\in M} \cup \underbrace{\{e_1, e_2, ..., e_r\}}_{\notin M}$ is again extreme

Proof. Let N be any extreme matching of size $|M|+1$. As $|N| > |M|$, $M \cup N$ has a component $Q = \{f_1, f_2, ..., f_q\}$ that is an M-augmenting path. As P is a shortest M-augmenting path, we know $l(P) \leq l(Q)$. Moreover, as $M^* = N \setminus \{f_1, f_3, ..., f_q\} \cup \{f_2, ..., f_{q-1}\}$ is a matching of size $|M|$ and as M is extreme, we know that $\omega(M^*) \leq \omega(M)$ Hence $\omega(N) = \omega(M^*) - l(Q) \leq \omega(M) - l(P) = \omega(M')$. Hence M' is extreme. □

Hungarian method for maximum weighted matching

"find iteratively extreme matchings $M_1, M_2, ..., M_j$ with $|M_k| = K$. Then the matching among $M_1, ..., M_j$ of maximum weight is a maximum weight matching."

Define D as in the maximum cardinality matching algorithm.

set

$$U' = U \setminus (\cup_{l \in M} e) \qquad W' = W \setminus (\cup_{l \in M} e)$$

extend D with nodes s and t and arcs (s, u) $\forall u \in U'$ with length 0 and (w, t) $\forall w \in W'$ with length 0.

Now we find a shortest path form s to t, to get an extreme matching M' from extreme matching M ($|M'| > |M|$).

Theorem 9.5:
Let M be an extreme matching. Then D does not contain a directed circuit of negative length.

Proof. Suppose C is a directed circuit in D with length $l(C) < 0$. We may assume $C = (u_0, w_1, u_1, ..., w_t, u_t)$ with $u_0 = u_t$, $u_1, ..., u_t \in U$ and $w_1, ..., w_t \in W$. Then $w_1u_1, ..., w_tu_t$ belong to M and the edges $u_0w_1, ..., u_{t-1}w_t$ do not belong to M. Then $M'' = M \setminus \{w_1u_1, ..., w_tu_t\} \cup \{u_0w_1, ..., u_{t+1}w_t\}$ matching of cardinality $|M''| = |M|$ with weight: $\omega(M'') = \omega(M) - l(C) > \omega(M)$. This contradicts to M being extreme. □

Corollary 9.6:
The maximum weighted matching in bipartite graphs can be found by using a shortest path algorithm $\frac{1}{2}|V|$-times.

Lecture 10 (2011-11-10)**:**

Matchings in non-bipartite graphs

Definition 10.1:
A component of a graph is called *ungerade (odd)*, if its number of vertices is odd. We define $o(G) :=$ number of odd components of G. $G - U := G[V \setminus U]$ denotes the subgraph induced by $V \setminus U$.

Theorem 10.2 (Tutte-Berge-Formula)**:**
For a graph $G = (V, E)$ it holds that $D(G) = \min_{U \subset V} \left\{ \frac{1}{2}(|V| + |U| - o(G - U)) \right\}$

Proof. We first prove $\leq$. For $U \subseteq V$ it holds that

$$\begin{aligned}\nu(F) &\leq |U| + \nu(G - U) \leq |U| + \frac{1}{2}(|V \setminus U| - o(G - U)) \\ &= \frac{1}{2}(|U \setminus V| + 2|U| - o(G - U)) \\ &= \frac{1}{2}(|V| + |U| - o(G - U))\end{aligned}$$

Now, we prove $\geq$. We apply inductions on $|V|$. For $|V| = 0$, the statement is trivial. Furterh, we may assume w.l.o.g (o.B.d.A) that G is connected, otherwise the result follows by induction on the components of G.

Case 1:
There exists a vertex v that is matched in all maximum matches (like in the bipartite case). Thus $\nu(G - V) = \nu(G) - 1$ and by induction there exists a subset $U' \subseteq V - v$ with $\nu(G - V) = \frac{1}{2}(|V \setminus \{v\}| + |U'| - o(G - v - U'))$ Set $U := U' \cup \{V\}$. Then

$$\begin{aligned}\nu(G) = \nu(G - v) - 1 &= \frac{1}{2}(|V \setminus \{v\}| + |U'| - o(G - U' - v)) - 1 \\ &= \frac{1}{2}(|V| - 1 + |U| - 1 - o(G - U) + 2) \\ &\leq \min_{T \subseteq V} \frac{1}{2}(|V| + |T| - O(G - T))\end{aligned}$$

Case 2:
There does (not) exist a vertex matched in all maximum matchings. So for all $v \in V$, there exists a maximum matching without v. Then, in particular $\nu(G) < \frac{1}{2}|V|$. We will show that there exists in this case a matching of size $\frac{1}{2}(|V| - 1)$. By this, we have proven the theorem (set $U = \emptyset$).

If there does not exist a matching of size $\frac{1}{2}(|V| - 1)$, then for every matching M, there exist two vertices u and v such that both are not matched. Among all maximum matchings, select a triple (M, u, v) for which $\text{Dist}(u, v)$ is minimum there, $\text{Dist}(u, v)$ is the minimum number of edges or a path between u and v. That is, for any other maximum matching N and pair of unmatched vertices s, t, $\text{Dist}(s, t) \geq \text{Dist}(u, v)$.

If $\text{Dist}(u, v) = 1$, then u and v are adjacent and we can extend M with $\{u, v\}$, a contradiction. Thus, $\text{Dist}(u, v) \geq 2$ and hence there exists a vertex t on the shortest path from u to v. Not that t is matched in M, otherwise $\text{Dist}(u, t) < \text{Dist}(u, v)$; a contradition. For t there exist other maximum matchings not covering t. Choose N such that $|M \cap N|$ is maximal.

Also, u and v are covered by N, since otherwise N would have a pair (t, u) (or (t, v)) of unmatched vertices with $\text{Dist}(u, t) < \text{Dist}(u, v)$ (or $\text{Dist}(t, v) < \text{Dist}(u, v)$, respectively).

Since $|M| = |N|$, there must be a second vertex $x \neq t$ which is not covered by N, but covered by M. Let $e = \{x, y\} \in M$. Then y is also covered by N, otherwise N could be extended by $\{x, y\}$, contradiction to maximality of $|N|$. Let $f = \{y, z\} \in N$. Now replace N by $N \setminus \{f\} \cup \{e\}$. The new matching has one more edge in common with M, contradiction.

Hence, a maximum matching cannot mis two or more vertices. Thus $\nu(G) = \frac{1}{2}(|V| - 1)$

□

Corollary 10.3 (Tutte's 1-Factor Theorem)**:**
A graph $G(V, E)$ has a perfect matching if and only if $G - U$ contains at most $|U|$ odd components for all $U \subseteq V$.

Corollary 10.4:
Let $G = (V, E)$ be a graph without isolated vertices. Then

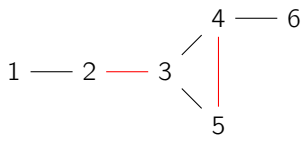
$$\rho(G) = \max_{U \subseteq V} \frac{|U| + o(G[U])}{2}$$

Proof. Homework □

Edmond's Matching Algorithm (blossom shrinking algorithm)

Again we are looking for M-augmenting paths. In bipartite graphs we just have to find a shortest path in the orientation of G by M.

Example 10.4(+1):



und noch ein Graph

Example 10.4(+2):
Weitere 2 graphen

We define for sets X and Y:

$$X/Y := \begin{cases} X & \text{if } X \cap Y = \emptyset \\ (X \setminus Y) \cup \{Y\} & \text{if } X \cap Y \neq \emptyset \end{cases}$$

Thus, if $G = (V, E)$ is a graph and $C \subseteq V$, then V/C is the set of ver- tices where all vertices in C are replaced by a singe vertex C. For an edge $e \in E$, $e/C = e$ if e and C are disjoint, whereas $e/C = \{u, C\}$ if $e \in \{u, v\}$ with $u \notin C$, $v \in C$, and $e/C = \{C, C\}$ if $e = \{u, v\}$ with $u, v \in C$.

The last type is unimporteant fo mathings and can be ignored. Further, for $F \subseteq E$, we have $F/C := \{f/C : f \in F\}$ and thus $G/C := (V/C, E/C)$ is again a graph which results from *Schrumpfen (shrinking)* of C.

Lecture 11 (2011-11-21)**:**

Let G be a graph, M a matching in G and W the set of unmatched vertices. A M-augmenting path is a M-alternating W-W chain of positive length where all vertices are distinct.

We call a M-alternating chain $P = \{v_0, v_0, ..., v_t\}$ a *M-Blüte (M-blossom)* if $v_0, ..., v_{t-1}$ are dstinct, v_0 is not matched in M and $v_t = v_0$

Theorem 11.1:
Let C be a M-blossom in G. M is a maximum matching if and only if M/C is a maximum matching in G/C.

Proof. Let $C = \{v_0, v_1, ..., v_t\}$, $G' = G/C$, $M' = M/C$. First, let P be a M-augmenting path in G. W.l.o.g we may assume that P does not start in v_0 (therwise turnaround P). If P does not visit any vertex of C, then P is a M-augmenting path in G' as well. If P visits a vertex of C, then we can decompose P in Q and R with Q ending at the first vertex of C. Replace this vertex in Q with vertex C in G'. Then Q is a M- augmenting path in G'. Now, on the reverse, let P' be an M'-augmenting path in G'. If P' does not visit vertex C, then P' is M-augmenting path in G. If vertex C is visited by P', we may assume that C is the end vertex (since M' does not cover C). We thus can replace vertex C with a suitable vertex $v_i \in C$ such that the new path Q ends at v_2 in G. If i is odd, then we extend Q with $v_{i+1}, v_{i+2}, ..., v_t = v_0$. If i is even, then we extend Q with $v_{i-1}, v_{i-2}, ..., v_0$. The resulting path is a M-augmenting path in G □

Edmonds' Algorithm

Input: Mathing M with set $W \subseteq V$ of unmatched vertices Output: Mathing N with $|N| = |M| + 1$ or a certificate that M is a maximum matching. Description:

```
1  Case 1: No M-alternating W-W chain exists:
2          Then M is an maximum matching.
3          STOP.
4  Case 2: A M-alternating W-W chain exists:
5          Let P = (v_0, v_1, ..., v_t) be a shortest M-alternating
6          W-W chain
7          Case 2A: P is a path.
8                   Then P is M-augmenting and N := MΔP.
9          Case 2B: P is not a path.
10                  Choose i < j with v_i = v_j and j as small as
                        possible
11                  Replace M by MΔ{v_0, ..., v_i}
12                  Then C := {v_i, v_{i+1}, ..., v_j} is a M-blossom.
13                  Apply the Algorithm recursively
14                  until G' := G/C and M' := M/C.
15
16                  * If a M'-augmenting path P in G'
```

```
17                  is returned, transform P' to a
18                  M-autmenting path in G
19                  (by \todo{Thm 11.1})
20              * If M' is a maximum matching in G',
21                then M is a maximum matching in G
22                (\todo{Thm 11.1})
```

Theorem 11.2:
Edmonds' Algorithm is correct and needs at most $\frac{1}{2}|V|$ repetitions to find a maximum matching.

Proof. Follows directly from and . □

thm 7.11

thm 11.1

How do we find a M-alternating W-W chain?

Definition 11.3:
A *M-alternierender Wald (M-alternating forest)* (V, F) is a forest with $M \subseteq F$, each component of (V, F) contains *either* a vertex for W *or* exists of a single edge in M, and each path in (V, F) starting with a vertex in W is M-alternating.

Let

$$even(F) = \{v \in V : F \text{ contains a } W\text{-}v \text{ path of even length}\}$$
$$odd(F) = \{v \in V : F \text{ contains a } W\text{-}v \text{ path of odd length}\}$$
$$free(F) = \{v \in V : F \text{ does not contain a } W\text{-}v \text{ path}\}$$

insert Figure ELISA 1

Note that each vertex $u \in odd(F)$ is incident to a unique edge in $F \setminus M$ and a unique edge in M.

Lemma 11.4:
If no edge $e \in E$ exists that connects $even(F)$ with $even(F) \cup free(F)$, then M is a maximum matching.

Proof. If no such edge exists, then even(F) is a stable set in $G - odd(F)$. In fact every vertex $u \in even(F)$ is an odd component in $G - odd(F)$. Let $U := odd(F)$.

$$o(G - U) \geq |even(F)| = |W| + |odd(F)| = |V| - 2|M| + |U|$$
$$\Leftrightarrow 2|M| \geq |V| + |U| - o(G - U).$$

By Tutte-Berge-Formula, M is a maximum matching. □

Construction of an M-alternating forest

Intialization: $F := M$. Choose for every vertex $v \in V$ an edge $e_v = vu$ with $u \in W$ (if possible).

Iterate: Find $v \in even(F) \cup free(F)$ for which $e_v = vu$ exists.
Case 1 : $v \in free(F)$: Add uv to F. Let $vw \in M$. For all $wx \in E$, set $e_x = \{x, w\}$

Case 2: $v \in even(F)$: Find $W - u$ respectively $W - v$ paths P and Q in F
2a: If P and Q are disjoint, then $F \cup \{uv\} \cup Q$ is a M-augmenting path
2b: If P and Q are not disjoint, then $P \cup Q \cup \{uv\}$ contains a M-blossom C. For all edges cx with $c \in C$ and $x \notin C$, set $e_x = C_x$. Replace G by G/C.

insert Figure ELISA 2

$$F = M \cup \{\}$$
$$W = \{1, 16, 18\}$$

v	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$\{v, u\} = e_V$		1								18														

Lecture 12 (2011-11-24)**:**

Weighted Matchings in general graphs

Instead of a maximum weighted matching, we search for a minimum weighted perfect matching. Without loss of generallity we may assume that G contains at least one perfect matching: construct a new graph H by copying G, add $n = |V(G)|$ new vertices, each one adjacent to one vertex of G, an add a clique on the new vertices. All new edges have weight zero. To turn a maximum weight perfec matching into a minimum weight perfect matching, we define new weights

insert figure NIKLAS 1

$$w'(e) := W - w(e) \quad \text{with} \quad W = \max_{e \in E} w(e)$$

Lemma 12.1:
A maximum weight matching in (G, w) corresponds to a minimum weight perfect matching in (H, w').

Proof. Let M be a perfect matching in H.

$$w'(M) = n \cdot W - \sum_{e \in M \cap E(G)} w(e)$$

Hence

$$\min_{M \in E(H)} w'(M) = n \cdot W - \max_{M \in E(G)} w(M)$$

insert NIKLAS 3

In contrast to the maximum cardinality matching problem, we also might have to "unshrink" a blossom: Expand □

Definition 12.2:
A collection Ω of odd subsets of V is called *verschachtelt (nested)*, if for all $U, W\Omega$ either $U \cap W = \emptyset$ or $U \subseteq W$ or $W \subseteq U$.

Example 12.2(+1):

$$\Omega = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{1, 2, 3\}, \{4, 5, 6\}, \{4, ..., 7\}\}$$

insert NIKLAS 4

if

$$\Omega = \{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{1,2,3\},\{4,5,6\}\}$$

then

$$\Omega^{max} = \{\{1,2,3\},\{4,5,6\},\{7\}\}$$

We will assume that $\{v\} \in \Omega$ for all $v \in V$. As a consequence Ω covers V. Further, there exist inclusionwise maximal elements in Ω. Let Ω^{max} denote the subsets of Ω that are inclusionwise maximal. Therefor Ω^{max} is a partition of V. The algorithm for finding a minimum weight perfect matching is "primal-dual", i.e., during the algorithm both a matching and a dual object, a function $\Pi : \Omega \to \mathbb{Q}$ is carried on. We consider functions $\Pi : \Omega \to \mathbb{Q}$ satisfying following conditions:

$$\begin{cases} \Pi(U) \geq 0 & \text{if } U \in \Omega \text{ with } |U| \geq \varepsilon \\ \sum\limits_{U \in \Omega : e \in \delta(U)} \Pi(U) \leq w(e) & \text{for all } e \in E \end{cases} \tag{1}$$

Lemma 12.3:
Let N be a perfect matching. Then $w(N) \geq \sum\limits_{U \in \Omega} \Pi(U)$

Proof.

$$w(N) = \sum_{e \in N} w(e) \overset{1}{\geq} \sum_{e \in N} \sum_{U \in \Omega : e \in \delta(U)} \Pi(U) = \sum_{u \in \Omega} \Pi(U) \cdot |N \cap \delta(U)| \overset{*}{\geq} \sum_{U \in \Omega} \Pi(U) \tag{2}$$

insert NIKLAS 5

(*): since all elements of Ω are of odd size, there always must be at least one vertex of U matched with a vertex of $V \setminus U$ in every perfect m.

So if we have a perfect matching N and a function Π fullfilling 2 with equality, we are done. Thus, a given $\Pi : \Omega \to \mathbb{Q}$, we define

$$w_\Pi(e) := w(e) - \sum_{U \in \Omega : e \in \delta(U)} \Pi(U) \overset{1}{\geq} 0$$

This in the end $w_\Pi(e) = 0$ for all $e \in N$. Further, leth $G \setminus \Omega$ be the graph with all $U \in \Omega^{max}$ shrunken (with U as shrunken vertex). Thus $G \setminus \Omega$ has vertex set Ω^{max} and $U, W \in \Omega^{max}$ are adjacent if and only if there exist $u \in U$, $w \in W$ with $\{u, w\} \in E$. Finally, we only consider collections Ω such that for all $U \in \Omega$ with $|U| \geq 3$, the graph H_U contains a Hamilton cycle C_U with edges e having $w_\Pi(e) = 0$. Hence H_U is the graph obtaind by shrinking all inclusionwise minimal subsets in $G[U]$ □

Edmonds' Minimum Weighted Perfect Matching Algorithm

```
1 Initialize:
2         M := ∅,
3         F := ∅,
```

$\Omega := \{\{v\}, v \in V\}$,
$\Pi(\{v\}) := 0$
As long as M is not perfect in $G \setminus \Omega$ iterate as follows
a) Select α maximal such that
$\Pi(U) - \alpha \geq 0 \forall U \in \Omega, |U| \geq \varepsilon, U \in odd(F)$,
$\Pi(U) + \alpha \geq 0 \forall U \in \Omega, |U| \geq \varepsilon, U \in even(F)$ and
$$\sum_{U \in \Omega \cap odd(F): e \in \delta(U)} (\Pi(u) - \alpha) + \sum_{U \in \Omega \cap even(F): e \in \delta(U)} (\Pi(u) - \alpha) \leq w(e) \forall e \in E$$

Reset $\Pi(U) := \Pi(U) - \alpha$ for $U \in odd(F)$ and
$\Pi(U) := \Pi(U) + \alpha$ for $U \in even(F)$
The new Π fullfills 1 and in addition either of the following holds:
i) $\exists e \in G \setminus \Omega$ with $w_\Pi(e) = 0$ such that e intesects with $even(F)$ but not with $odd(F)$
ii) $\exists U \in odd(F)$ with $|U| \geq \varepsilon$ and $\Pi(U) = 0$
b) If (i) holds, and only 1 end vertex of e belongs to $even(F)$ and the other one is in $free(F)$ then extend F with e GROW
If (i) holds, and both end vertices of e belong to $even(F)$ and $F \cup \{e\}$ contains a cycle U, add U to Ω with $\Pi(U) := 0$, replace F by $F \setminus U$ and M by $M \setminus U$ SHRINK
If (i) holds, and both ... $even(F)$ and $F \cup \{e\}$ contains a M-augmenting path, augment M an replace $F := M$. AUGMENT
c) If (ii) holds, remove U from Ω, replace F by $F \cup P \cup N$ and M by $M \cup N$, where P is the path of even length in C_U connection the two vertices incident to the edges in F adjacent to U and N the matching C_U which covers all vertices in U not covered by M EXPAND
END

Lecture 13 (2011-11-25)**:**

The Traveling Salesman Problem

Definition 13.1:
A (directed) cycle (path) with $|V|$ (resp. $|V| - 1$) edges is called a (directed) *Hamiltonkreis (Hamilton cycle)* (*Hamiltonpfad (Hamilton path)*).

Occasionally, a Hamilton cycle will be called a *Tour (tour)*.

The *Problem des Handelsreisenden (traveling salesman problem (TSP))* is to find, given distances c_{ij} for all i, j, a minimum length Hamilton cycle in a complete graph.

If $c_{ij} = c_{ji} \forall i, j$, the problem is *symmetrisch (symmetric)*, otherwise it is *asymetrisch (assymmetric)*.

If c_{ij} represent euclidean distances in $\mathbb{R}^2$, the TSP is euclidean.

Lemma 13.2:
If $c_{ij} < \infty$ fulfills the triangle inequality $c_{ij} + c_{jk} \geq c_{ik}$, the vertices can be placed in $\mathbb{R}^2$ such that the TSP is euclidean. □

Let $C(S, k) =$ minimum length of a path from vertex 1 to vertex k which visits all vertices in S exactly once (and no other vertices). Hence, the TSP is solved as soon as we have computed $C(\{2, ..., n\}, 1)$.

Lemma 13.3:

$$C(S, k) = min_{j \in S} \{C(S \setminus \{j\}, j) + c_{jk}\}$$

Proof. The optimal path from 1 to k via S has as last-but-one vertex a vertex $j^* \in S$. The path from 1 to j^* via $S \setminus \{j^*\}$ should be optimal (i.e. has value $C(S \setminus \{j^*\}, j^*)$), otherwise we can improve it. Vertex j^* can be determined by taking the minimum among all $j \in S$. □

Listing 1: Held-Karp Algorithm for TSP

```
1  Initialize C(∅, k) = c_1k ∀k ∈ {2, ..., n}
2  FOR 2 ≤ l ≤ n − 1 DO
3          FOR ALL S ⊆ {2, ..., n} with |S| = l DO
4                  FOR k ∈ {2, ..., n} \ S DO
5                          C(S, k) = min_{j∈S} {C(S \ {j}, j) + c_jk}
6                  ENDDO
7          ENDDO
8  ENDDO
9  C* = min_{j∈{2,...,n}} {C({2, ..., n} \ {j}, j) + c_j1}
10 RETURN C*
```

Theorem 13.4:
The Held-Karp algorithm is correct.
In total $(n-1)2^{n-2} - (n-2)$ values have to be computed.

Proof. Follows directly from Lemma 13.2 and the slides. □

For comparison: Complete enumeration searches through $(n-1)!$ tours. $999! \approx 10^{2500}$. Stirling formula tells us:

$$\lim_{n \to \infty} \frac{n! \cdot e^n}{\sqrt{2\pi n} \cdot n^n} = 1 \Rightarrow 2^n << n!$$

The pratical running time of the algorithm can be reduced significantly by a good solution value (upper bound): all values $C(S, k) >$ upper bound can be ignored for further computations.

SpanningTree heuristic

A TSP tour is a set of $|V|$ edges ~~building a cycle~~ connecting all vertices. Instead, I might search for a relaxation, the minimum spanning tree plus one edge. Stated otherwise, the MST is a lower bound on the TSP tour.

Listing 2: The MST heuristic is therefore

```
1 Find a MST T in G
2 Duplicate all edges in T: T_2
3 Determine an euler tour in (V,T_2)
4 Replace vertices visited twice by the shortest to the
    next not yet visited vertex.
```

Lecture 14 (2011-12-01)**:**

Flows in Networks

Menger's Theorem

Definition 14.1:
Let $D = (V, A)$ be a digraph and S, T subsets of V. A path is called a S-T-path, if the start vertex is in S and the end vertex is in T.

If $S = \{s\}$ and $T = \{t\}$ we also refer to the path as S-T-path instead of $\{s\}-\{t\}$-path.

Definition 14.2:
Two S-T-paths P_1 and P_2 are called *knotendisjunkt (vertex disjoint)* if P_1 and P_2 have no common vertices.

Two S-T-paths are called internally vertex disjoint if they have no common vertices, except for the start and end vertices.

Two S-T-paths are called *bogendisjunkt (arc disjoint)* if they have no arcs in common.

Question: How many vertex/arc disjoint paths exist between S and T resp. s and t?

Definition 14.3:
A set $C \subset V$ *(separates)* S from T if every S-T-path intersects with C (C can intersect $S \cup T$). C is called a *($(S$-$T)$-separator)*.

Theorem 14.4:
(Menger's Theorem, directed vertex disjoint version)
Let $D = (V, A)$ be a digraph and $S, T \subset V$. Then, the maximum number of pairwise vertex disjoint S-T-paths equals the minimum size of a S-T-separator.

Proof. Clearly, the number of vertex disjoint paths cannot exceed the size of a S-T-separator C (i.e. for $v \in C$, at most one path exists). We will show $\geq$ by

induction on $|A|$. For $|A| = 0$, the statement ist trivial. Now, let k be the minimum size of a S-T-separator. Select $a = (u, v) \in A$ arbitrarely. If every S-T-separator in $D \setminus a$ has size $\geq k$, then by incduction k vertex-disjoint paths exist in $D \setminus a$, and thus in D as well.
Thus, we can assume w.l.o.g that $D \setminus a$ has a S-T-separator C with $|C| \leq k - 1$. Then $C \cup \{u\}$ and $C \cup \{v\}$ are S-T-separators in D of size k.
Now, every S-$(C \cup \{u\})$-separator B of $D \setminus a$ has size at least k, since B also separates S from T in D: every S-T-path in D intersects $C \cup \{u\}$ and thus P contains S-$(C \cup \{u\})$-subpath in $D \setminus a$. Therefore, the subpath and thus P itself intersects B. By induction $D \setminus a$ has k vertex disjoint S-$(C \cup \{u\})$-paths. Similarly there exist k vertex disjoint $(C \cup \{v\})$-T-paths in $D \setminus a$. Since both path sets use all vertices in C, we can connect these $k - 1$ S-C-paths with the C-T-paths. One more path in D can be established by connecting the S-u-path with the v-T-path via arc $a = (u, v)$. □

Definition 14.5:
A set $U \subset V$ is called a *((s-t)-vertex cut)* if $s, t \notin u$ and every S-T-path intersects U.

Corollary 14.6:
Let $D = (V, A)$ be a digraph and s, t two non-adjacent vertices. then the maximum number of *(internally vertex disjoint)* S-T-paths equals the minimum size of a S-T-vertex cut.

Proof. Let $D' = D \setminus s - t$. $S = N_D^+(s)$, $T = N_D^-(t)$. Apply □

Definition 14.7:
A arc set $C \subset A$ define a *(s-t)-Schnitt ((s-t)-cut)* if a subset $U \subset V, s \in U, t \notin U$ with $\delta^+(U) \subset C$.

Corollary 14.8:
(Menger's Theorem, directed arc version) let $D = (V, A)$ be a digraph with $s, t \in V$. Then the maximum number of arc disjoint S-T-paths equals the minimum size of a S-T-cut.

Proof. Übungsblatt □

Question: How to find arc disjoint S-T-paths? *Answer:*Given D a digrpah and a path P, let us define D/P as the digraph in which all arcs of P are reversed.

```
1 Initialize D_0 := D,     k = 0;
2 WHILE D_k contains a S-T-path P_k DO
3         D_{k+1} := D_k/P_k
4         k := k + 1
5 ENDWHILE
6 The reversed arcs in D − k are k − 1 arc disjoint S-T-
    paths.
```

Note: A S-T-path P_k in D_k can be found with e.g. Dijkstra.

Lecture 15 (2011-12-05)**:**

Flows in Networks

Consider the following problem:

Let $D = (V, A)$ be a digraph with *arc capacities* $c(a) \geq 0$ for all $a \in A$. How many paths can be established from s to t (for $s, t \in V$ and duplicates allowed) without having more than $c(a)$ paths via arc $a \in A$.

If $c(a) = 1$ for all $a \in A$, we search for arc-disjoint paths and Menger's Theorem provides the answer.

Instead of specifying the paths explicitly, we can define values on the arcs stating the number of paths across that arc.

Definition 15.1:
A function $x : A \to \mathbb{R}$ (or vector $x \in \mathbb{R}^{|A|}$) is a *zulässiger (s,t)-Fluss (feasible (s,t)-flow)* if the following conditions are satisfied:

$$0 \leq x_a \leq c_a \; \forall a \in A \tag{3}$$

$$\sum_{a \in \delta^+(v)} x_a = \sum_{a \in \delta^-(v)} x_a \tag{4}$$

(3) representing the *Kapazitätsbedingungen (capacity constraints)*, and (4) representing the *Flusserhaltungsbedingungen (flow conservation constraints)*.

The *Wert des (s,t)-Flusses ((s,t)-flow-value)* x is

$$val(x) := \sum_{a \in \delta^+(s)} x_a - \sum_{a \in \delta^-(s)} x_a$$

The vertex s is the *Quelle (source/origin)* and t is the *Senke/Ziel (target/sink/destination)*.

Question what is the maximum value of a (s,t)-flow?

A (s,t)-cut $S \subseteq A$ interrupts every path from s to t. The capacity of a cut S is $\sum_{a \in S} c_a =: c(S)$

Lemma 15.2:
Let $W \subseteq V$.

(a) If $s \in W$, $t \notin W$, then $val(x) = \sum_{a \in \delta^+(w)} x_a - \sum_{a \in \delta^-(w)} x_a$ for every feasible (s,t) flow x.

(b) The maximum value of a (s,t)-flow is at most the minimum capacity of a (s,t)-cut.

Proof. (a) By flow conservation constraint (z), it follows that

$$\begin{aligned} val(x) &= \sum_{a\in\delta^+(s)} x_a - \sum_{a\in\delta^-(s)} x_a + \sum_{v\in W\setminus\{s\}} \left(\sum_{a\in\delta^+(v)} x_a - \sum_{a\in\delta^-(v)} x_a \right) \\ &= \sum_{v\in W} \left(\sum_{a\in\delta^+(v)} x_a - \sum_{a\in\delta^-(v)} x_a \right) \\ &= \sum_{a\in\delta^+(W)} x_a - \sum_{a\in\delta^-(W)} x_a \end{aligned}$$

(b) Let $\delta^+(W)$ be an arbitrary (s,t)-cut and x a feasible (s,t)-flow. By (3) and part (a) it holds that

$$val(x) = \sum_{a\in\delta^+(W)} x_a - \sum_{a\in\delta^-(W)} x_a \leq \sum_{a\in\delta^+(W)} c_a = c(\delta^+(W))$$

□

Theorem 15.3 (Max-Flow-Min-Cut-Theorem)**:**
Let $D = (V, A)$ be a digraph, $s, t \in V$ and $c : A \to \mathbb{R}_+$ a capacity function. The maximum value of a (s,t)-flow equals the minimum capacity of a (s,t)-cut:

$$\max_{x\ (s,t)\text{-flow}} val(x) = \min_{(s,t)\text{-cut } S} c(S)$$

Proof.

- If c is integer, every arc $a \in A$ is replaced by c_a parallel arcs of capacity one. Now the result follows from Menger's Theorem for arc-disjoint paths.
- If c is rational, then there exists a N such that $N \cdot c_a$ is an integer for all $a \in A$. By this scaling, both the max flow and min cut are scaled with N as well. Now, the results holds by the result of the integer case.
- If c is real valued, the result follows from continuity and compactness of $\mathbb{Q}$

□

Corollary 15.4:
If c is integer, there exists an *integer* maximum (s,t)-flow. For $c : A \to \mathbb{Q}$, there exists a combinatorial algorithm to determine a max (s,t) flow: Ford-Fulkerson Algorithm

Definition 15.5:
Let $D = (V, A)$ be a digraph with arc capacities $c(a) \in \mathbb{Q}\ \forall a \in A, s, t \in V, s \neq t$ and x a feasible (s,t)-flow in D. In an *undirected* $[s,v]$-path P we call an arc (i,j) a *Vorwärtsbogen (forward arc)* if it is directed from s to v on the path, otherwise (i,j) is a *Rückwärtsbogen (backward-arc)*.

Path P is called an *augmenting $[s,v]$-path with respect to (s,t)-flow x* if $x_{ij} < c_{ij}$ for all forward arcs and $x_{ij} > 0$ for all backward arcs (i,j).

Theorem 15.6:
A (s,t)-flow x is maximum if and only if no augmenting $[s,t]$-path with respect to x exists.

Proof. If P defined an augmenting $[s,t]$-path with respect to x, then let

$$\varepsilon := \min \begin{cases} c_{ij} - x_{ij} & \text{if } (i,j) \in P \text{ forward arc} \\ x_{ij} & \text{if } (i,j) \in P \text{ backward arc} \end{cases}$$

Now define

$$\tilde{x}_{ij} := \min \begin{cases} x_{ij} + \varepsilon & \text{if } (i,j) \in P \text{ forward arc} \\ x_{ij} - \varepsilon & \text{if } (i,j) \in P \text{ backward arc} \\ x_{ij} & \text{if } (i,j) \in A \end{cases}$$

Clearly $\tilde{x}_{ij}$ defines a feasible (s,t)-flow. Moreover $val(\tilde{x}) = val(x) + \varepsilon$, hence x was not maximum.
Now , assume x does not have an augmenting $[s,v]$-path. Then, let

$$W := \{v \in V : \text{ there exists an augmenting } [s,t]\text{-path with respect to } x\}$$

Hence, $s \in W$ and $t \notin W$ by assumption. To be more precise,

$$\begin{aligned} x_a &= c_a \quad \text{for all } a \in \delta^+(W) \text{ and} \\ x_a &= 0 \quad \text{for all } a \in \delta^-(W) \end{aligned}$$

It follows $val(x) = x(\delta^+(W)) - X(\delta^-(W)) = c(delta^+(W))$. By lemma 15.2(b), x is maximum. □

Lecture 16 (2011-12-08)**:**

Minimum Cost Flows

Definition 16.1:
Let $D = (V,A)$ be a digraph with arc capacities $c(a) \geq 0$ for all $a \in A$ and cost coefficients $w(a)$ for all $a \in A$. Let $s,t \in V$.
Consider all (s,t)-flows of value f. The *Minimaler Kosten Netzwerkflussproblem (Minimum Cost Flow (MCF) Problem)* consists of finding a (s,t)-flow x with $val(x) = f$ and cost $\sum_{a \in A} w(a)x_a$ minimum among all (s,t)-flows of value f.

The MCF Problem can be formulated as linear program (later more) and special network-simplex algorithms exist to solve the problem in polynomial time. Alternatively several combinatorial algorithms exist.

Definition 16.2:
Let x be a feasible (s,t)-flow in D and let C be a (not necessarily directed) cycle in D. The cycle C can be orientated in two possible ways (clockwise or

counterclockwise). Given an orientation of C, *forward arcs* on C are directed along the orientation, *backwards arcs* opposite.

A cycle C is called an *augmentierender Kreis (augmenting cycle)* w.r.t. x if there exists an orientation of X such that

$$x_a < c_a \text{ for all forward arcs } a \in C$$
$$x_a > 0 \text{ for all backward arcs } a \in C$$

Definition 16.3:
A *cost* of an augmenting cycle C is defined as

$$\sum_{a \in C:forward} w_a - \sum_{a \in c:backward} w_a$$

Theorem 16.4:
A feasible (s,t)-flow x in D with value f has minimum cost if and only if no augmenting cycle C w.r.t. x with negative costs exists.

Proof. only $\Rightarrow$, $\Leftarrow$ later.

Let $\sum w(a)x_a$ be minimum. Assume that there exists an augmenting cycle with negative cost. We define

$$\varepsilon = \min(c_{ij} - x_{ij} if (i,j) \in C forward; x_{ij} if (i,j) \in C backward)$$

and

$$\tilde{x}_{ij} = (x_{ij} + \varepsilon if (i,j) \in C forward, x_{ij} - \varepsilon if (i,j) \in C backward, x_{ij} if (i,j) \in A \setminus C)$$

Now, $\tilde{x}$ is a feasible (s,t)-flow with $val(\tilde{x}) = val(x) = f$ and cost

$$\sum_{a \in A} w(a)\tilde{x}_a = \sum_{a \in A} w(a)x_a + \varepsilon \left(\sum_{a \in C:foward} w(a) - \sum_{a \in C:backward} w(a) \right) < \sum_{a \in A} w(a)x_a$$

To proe the backward direction, we first describe the algorithmic procedure. □

Definition 16.5:
Given a (s,t)-flow x with $val(x) = f$, we define the *augmentierendes Netzwerk (augmenting network)* w.r.t. x as follows: $N = (V, \overline{A}, \overline{C}, \overline{w})$ with

$$\overline{A} = A_1 \cup A_2$$
$$A_1 = \{ij \in A : x_{ij} < c_{ij}\}$$
$$A_2 = \{ji : ij \in A, x_{ij} > 0\}$$

For $a \in A$, we denote by $a_1 \in A_1$ and / or $a_2 \in A_2$ the corresponding arcs in A_1 and A_2. If a_1 (or a_2) does not exist, we evaluate $x(a_1)$ (or $x(a_2)$) with zero. Further,

$$\overline{c}(\overline{a}) = \begin{cases} c(a) - x(a) & \text{if } \overline{a} = a_1 \\ x(a) & \text{if } \overline{a} = a_2 \end{cases} \qquad \forall a \in A$$

$$\overline{w}(\overline{a}) = \begin{cases} w(a) & \text{if } \overline{a} = a_1 \\ -w(a) & \text{if } \overline{a} = a_2 \end{cases} \qquad \forall a \in A$$

Lemma 16.6:
Every augmenting cycle in D corresponds with exactly one *directed* cycle in N. The cost of both cycles are identical.

Proof. Exercise sheet. □

Theorem 16.7:
The flow x is cost minimal among all (s,t)-flows in D of value f if and only if no directed cycle in N has negative cost.

Proof. $\Rightarrow$: analogue to Theorem 16.4.
$\Leftarrow$: Assume x is not cost minimal. Hence, there exists a $\tilde{x}$ with value f and $w^T\tilde{x} < w^T x$. Now define for $\overline{a} \in \overline{A}$ (w.r.t. x)

$$\overline{x}(\overline{a}) = (max\{0, \tilde{x}(a) - x(a)\} if \overline{a} = a_1 \in A_1, max\{0, x(a) - \tilde{x}(a)\} if \overline{a} = a_2 \in A_2)$$

Thus it holds that $\overline{x}(a_1) - \overline{x}(a_2) = \tilde{x}(a) - x(a)$. Further, it holds that

$$\overline{x}(a_1)\overline{w}(a_1) + \overline{x}(a_2)\overline{w}(a_2) = w(a)(\tilde{x}(a) - x(a))$$

and thus

$$\sum_{\overline{a} \in \overline{A}} \overline{w}(\overline{a})\overline{x}(\overline{a}) = \sum_{a \in A} w(a)(\tilde{x}(a) - x(a)) = w^T\tilde{x} - w^T x < 0$$

Moreover, flow conservation holds for all $v \in V$:

$$\begin{aligned}
&\sum_{a \in \delta_N^+(v)} \overline{x}(a) - \sum_{a \in \delta_N^-(v)} \overline{x}(a) \\
=& \sum_{a \in \delta_D^+(v)} \overline{x}(a_1) - \overline{x}(a_2) - \sum_{a \in \delta_D^-(v)} \overline{x}(a_1) - \overline{x}(a_2) \\
=& \sum_{a \in \delta^+(v)} \tilde{x}(a) - x(a) - \sum_{a \in \delta^-(v)} \tilde{x}(a) - x(a) \\
=& \left(\sum_{a \in \delta^+(v)} \tilde{x}(a) - \sum_{a \in \delta^-(v)} \tilde{x}(a) \right) - \left(\sum_{a \in \delta^+(v)} x(a) - \sum_{a \in \delta^-(v)} x(a) \right) \\
=& (0 - 0 if v \in V \setminus \{s,t\}, f - f = 0 if v = s, -f + f = 0 if v = t)
\end{aligned}$$

Thus $\overline{x}$ is a feasible flow of value 0 and $\overline{w}^T\overline{x} < 0$. Since $\overline{x} \neq 0$ and flow conservation holds for all $v \in V$, $\overline{x}$ is a *Zirkulation (circulation)* and can be decomposed into flows x_i defined on cycles C_i $(i = 1, ..., k < |\overline{A}|)$. With $\sum_{i=1}^{k} x_i(\overline{a}) = \overline{x}(\overline{a}) \forall \overline{a} \in \overline{A}$. For at least one cycle C_j it must hold that $\overline{w}^T x_j < 0$, otherwise $\overline{w}^T\overline{x} \geq 0$. Then the vector x_j defines an augmenting cycle with negative cost. Contradiction. □

Lecture 17 (2012-12-12)**:**

Complexity Theory (Komplexitätstheorie)

A *Problem (problem)* is a general question, where several parameters are left open. A *Lösung (solution)* consists of answers for those parameters. A problem is defined by a description of all its parameters and which properties require an answer. A *Probleminstanz (problem instance)* is a specific input where all known parameters are explicitly given.

The question "What is the shortest travelling salesman tour in a graph?" is a *problem*, whereas the question "What is the shortest travelling salesman tour in bier127.tsp?" is a *problem instance* of the earlier problem. The known parameters are the number of cities and the distance matrix. The parameters left open are the follow-up city for every city.

In general, we denote a problem as Π, whereas an instace of problem Π is denoted by $I \in \Pi$.

An *Algorithmus (algorithm)* *solves* a problem Π if for every problem instance $I \in \Pi$, the algorithm finds a solution. The aim of designing algorithms is to develop procedures to fnd a solution for any problem instance that is as *"efficient"* as possible, in particular efficient regarding *time* and *memory* requirements.

There exist two types of problems: problems that can be answered by "yes" or "no" and those that ask to find an object with certain properties. "Does there exist a solution to the TSP with the value at most K?" can be answered by "yes" or "no". Of course, the answer "yes" should be verifiable with a tour of length $\leq K$; an answer "no" should also be guaranteed.

For yes/no problems, we do not have to distinguish between *solution* and *optimal solution*.

The *Zeitkomplexität (time complexity)* (respectively the *Speicherkomplexität (memory complexity)*) of an algorithm depends in general on the *"size"* of the problem instance, for example the *amount* of input data.

The *encoding* of a problem instance is of critical importance. Integers are *binary encoded*. The binary encoding of nonnegative integer n requires $\lceil \log_2(n+1) \rceil$ bits. One more bit is required for the sign of an integer. The *Codierungslänge (coding length)* $< n >$ is the number of bits required to encode n.

$$< n >:= \lceil \log_2(n+1) \rceil + 1$$

A rational number $r = \frac{p}{q}$ has coding length $< r >:=< p > + < q >$. The coding length of a vector $x \in \mathbb{Q}^n$ is $< x >:= \sum_{i=1}^{n} < x_i > \leq n \cdot \max < x_i >$ and of a matrix $A \in \mathbb{Q}^{n \times n}$ is $< A >:= \sum_{i=1}^{m} \sum_{j=1}^{n} < a_{ij} > \leq m \cdot n \cdot \max < a_{ij} >$.

Data structures to encode graphs and digraphs will be discussed in the next lecture.

The total number of bits required to describe a problem instance is called the *coding length* or *Eingabelänge (input length)* $< I >$ of I.

Example 17.0(+1):
The Knapsack problem has input length

$$< I >=< c > + < a > + < b >= 2 \cdot n \cdot \max < c_i, a_i > + < b >$$

To execute an algorithm and to compute its running time and memory requirement depending on the input length of a problem instance, we need a so-called *computing model*: a *Turing-machine* or *RAM-machine* are two examples (see exercise sheet).

The algorithm first reads the data of a problem instance and uses for this $< I >$ bits. Further bits are required to compute the solution. The *Speicherbedarf (memory requirement)* of algorithm A to solve I is the number of bits that are used at least once during the execution of A.

Example 17.0(+2):
Knapsac dynamic programming

$$f(k, b) = \max\{f(k-1, b), f(k-1, b-a_k) + c_k\}$$

. Only $2 \cdot b$ numbers hare to be kept for the computations.

The *Laufzeit (running time)* of A to solve I is the number of elementary operations which A requires until the end of the procedure. *elementare Operationen (Elementary operations)* are

- reading, writing, and deleting
- adding, subtracting, multiplying, dividing and comparing

of rational (integer) numbers. Here, we estimate each operation with respect to the maximum numbers involved.

The function $f_A : \mathbb{N} \to \mathbb{N}$ defined by

$$f_A(n) := \max_{I \in \Pi \text{ with } <I> \leq n} \{\text{running time of } A \text{ to solve } I\}$$

is called the *Laufzeitfunktion (running time function)* of A.

The function $S_A : \mathbb{N} \to \mathbb{N}$ defined by

$$f_A(n) := \max_{I \in \Pi \text{ with } <I> \leq n} \{\text{memory requirement of } A \text{ to solve } I\}$$

is called the *Speicherbedarfsfunktion (memory function)* of A.

Example 17.0(+3):

Examples:	Knapsack DP	TSP DP
Inputlength	$(2n+1) < b >$	$< n > + n^2 max < c_{ij}$
Runningtime	$\mathcal{O}(n \cdot b)$	$2^{n-2}(n-1) - (n-2)$

The algorithm A has *polynomielle Laufzeit (polynomial running time)* (short: A is a polynomial algorithm if there exists a polynomial $p : \mathbb{N} \Leftarrow \mathbb{N}$ with $f_A(n) \leq p(n) \; \forall \; n \in \mathbb{N}$. If p is a polynomial of degree k, we call f_A of order n^k and write $f_A = O(n^k)$. Algorithm A has polynomial memory requirements if there exists a polynomial $q : \mathbb{N} \Leftarrow \mathbb{N}$ such that $s_A(n) \leq q(n) \; \forall \; n \in \mathbb{N}$.

Example Given a sequence of number $a_i \in \mathbb{N}, \ldots, a_k \in \mathbb{N}$, determine the largest value Algorithm:

```
1 max:=0
2 FOR i=1 TO k DO
3         IF a_i > max THEN max:=a_i
```

Inputlength: $k \cdot max < a_j >\leq n$

Running time: $f_A(n) = \frac{n}{max<a_i>} \leq n$ polynomial $p(n) = 5n + 1$

Memory: $s_A(n) = 2max < a_i >\leq n$

Lecture 18 (2012-12-15)**:**

Complexity Theory – P, NP, NP-completeness

Dynamic Programming Algorithms solve knapsack in pseudopolynomial time (b depends on input, not on size of input, but also not in the exponent) and TSP in exponential time.

Question 18.0.1:
Do there exist polynomial time algorithms for knapsack and/or TSP?

- Dijkstra for shortes path is polynomial time
- Algorithm for min-Cost-flow is polynomial time

What is the difference between those problems?

Definition 18.1:
A *Entscheidungsproblem (decision problem)* is a problem with exactly two possible answers: *yes* and *no*.

We restrict ourselves to decision problems with solution algorithms that have *finite* running time.

Definition 18.2:
The class of all decision problems for which there exists a polynomial time algorithm is denoted by $\mathcal{P}$.

The decision problem "does G have a cycle?" belongs to $\mathcal{P}$ (a polytime algorithm might be: determine all connnected components, and check for each connected component whether the number of edges is at least the number of vertices).

For the decision problem "does G have a Hamilton cycle?" it is until today unclear whether it belons to $\mathcal{P}$.

To distinguish between these problems (without knowing wethere they belong to $\mathcal{P}$) we define a second class of problems: $\mathcal{NP}$ - *nichtdeterministisch polynomiell (nondeterministic polynomial)*

Informally speaking, adecision problem Π belongs to $\mathcal{NP}$ if a *Zertifikat(Lösung) (certificate(solution))* of a "yes" answer for the problem instance $I \in \Pi$ can be verified in polynomial-time. here a certificate means a proof of answering yes.

Example 18.2(+1):
The problem "Hamilton cycle in G" belons to $\mathcal{NP}$ because its certificate is a sequence of n vertices such that $v_i v_{i+1} \in E (i = 1, \ldots, n-1)$ and $v_n v_1 \in E$. Verification implies checking the existence of all edges, which can be done in polytime.

Definition 18.3:
A decision problem Π belongs to the class $\mathcal{NP}$ if

(a) for every problem instance $I \in \Pi$ with answer "yes" an object Q (certificate) exists, which allows the verification of the "yes" answer.

(b) there exists an algorithm to verify on the basis of Q the answer "yes" in time polynomial in $< I >$ (thus only in the encoding length of I, not Q)

Hence, both "does G have a cycle" and "does G have a hamilton cycle" belong to $\mathcal{NP}$ since Q in these cases is a sequence of vertices, constituting a (Hamilton) cycle.

Example 18.3(+1):
"Is $n \in \mathbb{Z}$ the product of two integers $\neq 1$" (Is n not prime) is in $\mathcal{NP}$ since given $a, b \neq 1$ the correctness of an "yes" answer on the basis fo a, b can be verified by a single multiplication.

What about "Does G *not* have a Hamilton cycle?" or equivalently, what about the "no" answer of "Does G have a Hamilton cycle?". this much more difficult to verify. $\Rightarrow \mathcal{NP}$ is not "symmetric".

If a "no" answer can be verified in polynomial time with an object Q, the problem belongs to the class co-$\mathcal{NP}$.

Stating the above differently

$$\Pi_1 := \{G : \ G \text{ is a graph and has a perfect matching}\}$$

belongs to $\mathcal{NP}$ since

$$\Pi_1' := \{(G, M) : \ G \text{ is a graph and } M \text{ is a perfect matching in } G\}$$

belongs to $\mathcal{P}$.

Π_1 also belongs to co-$\mathcal{NP}$ since the problem

$$\Pi_1'' := \{(G, W) : \ G \text{ is a graph and } W \text{ is a subset of the vertex set of } G \text{ such that } G\text{-}W \text{ has more than } |W| \text{ odd components}\}$$

belongs to $\mathcal{P}$.

A *undeterministischer Algorithmus (non demterministic algorithm)* solves a problem by guessing a solution (object) and then verifies (in polynomial time) the correctness of the solution.

Lemma 18.4:
$\mathcal{P} \subseteq \mathcal{NP}$

Proof. For $\Pi \in \mathcal{P}$ we have a polynomial time algorithm to compute the solution. Hence, we can also verify this solution with the algorithm. □

Lemma 18.5:
$\mathcal{P} \subseteq$ co-$\mathcal{NP}$

Corollary 18.6:
$\mathcal{P} \subseteq \mathcal{NP} \cap$ co-$\mathcal{NP}$.

Wether $\mathcal{P} = \mathcal{NP}$ is one of the millenium problems ($ 1M) If $\mathcal{NP} \neq$ co-$\mathcal{NP}$, then $\mathcal{P} \neq$ co-$\mathcal{NP}$

Definition 18.7:
Let Π and $\tilde{\Pi}$ be two decision problems. A *Polynomielle Reduktion (polynomial transformation)* of Π to $\tilde{\Pi}$ is a polynomial algorithm which constructs from a problem instance $I \in \Pi$ a problem instance $\tilde{I} \in \tilde{\Pi}$ such that the answer of I is "yes" if and only if the answer of $\tilde{I}$ is "yes".

Note if $\tilde{\Pi}$ is solvable in polynomial time, then also Π:

TSP of size n $\xrightarrow{poly}$ Shortest Path of size $poly(n)$ $\xrightarrow{poly}$ Dijkstra of size $poly(poly(n)) = poly(n)$ $\xrightarrow{yes/no}$ TSP.

Definition 18.8:
A decision problem Π is called *NP-vollstandig (NP-complete)* if

- $\Pi \in \mathcal{NP}$
- every other problem in $\mathcal{NP}$ can be polynomially transformed to Π

Note, if $\mathcal{NP}$-complete problem Π is polynomial time solveable, then all problems in $\mathcal{NP}$ can be solved in polynomial time

in such a case, $\mathcal{P} = \mathcal{NP}$

the most difficult problems in $\mathcal{NP}$-complete problems (if they exist?)

Lecture 19 (2011-12-19)**:**

Complexity classes:

$\mathcal{P}$	polynomial-time solveable decision problems
$\mathcal{NP}$	nondeterministic polytime solveable dicision problems
$\mathcal{NP}$-complete	every other problem in $\mathcal{NP}$ can be polynomially transformed (reduced) to it and it is in $\mathcal{NP}$ itslef.

Remark 4:
If Π can be polynomiall tranformed to Π' and $\Pi' \in \mathcal{P}$, then also $\Pi \in \mathcal{P}$.

Corollary 19.1:
If $\Pi \in \mathcal{NP}$, Π' is $\mathcal{NP}$-complete, and there exists a polynomial time transformation form Π' to Π, then Π is $\mathcal{NP}$-complete as well.

Proof. Since Π' is $\mathcal{NP}$-complete, every problem in $\mathcal{NP}$ can be polynomially transformed to Π'. A second poly. transformed from Π' to Π yields a polynomial transformation from every problem in $\mathcal{NP}$ to Π. In addition, if $\Pi \in \mathcal{NP}$, both conditions are fulfilled □

Definition 19.2:
A *Boolescher Ausdruck (boolean expression)* is a logical concatenation of true/-false variables $x_1, x_2, ..., x_n$. Given two boolean expressions v and w, also

$$\underset{\text{or}}{v \vee w},$$
$$\underset{\text{and}}{v \wedge w} \text{ and}$$
$$\underset{\text{not}}{\neg v}$$

are boolean expressions. The *Erfüllbarkeitsproblem (satisfiability problem (SAT))* requestes an assignment of true/false (1/0) to the variables such that the boolean expression $f(x_1, ..., x_n)$ equals true.

Example 19.2(+1):

$$\begin{aligned} x_2 = T, x_3 = F, x_5 = F & \\ ((x_2 \wedge x_3) \vee \neg(x_3 \vee x_5)) \wedge x_2 &= ((T \wedge F) \vee \neg(F \vee F)) \wedge T \\ &= (F \vee T) \wedge T \\ &= T \wedge T = T \end{aligned}$$

The following rules hold:

$$\begin{aligned} 0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0, 1 \wedge 1 = 1, \\ 0 \vee 0 = 0, 0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1, \\ \neg 0 = 1, \neg 1 = 0, (0) = 0, (1) = 1 \end{aligned}$$

Theorem 19.3 (Cook's Theorem, 1971)**:**
The Satisfiability Problem SAT is $\mathcal{NP}$-complete.

Proof. See Schrÿver's lecture notes, pages 105-107
SAT $\rightarrow$ 3-SAT $\rightarrow$ EXACT COVER $\rightarrow$ DIRECTED HAMILTONIAN CYCLE $\rightarrow$ UNDIRECTED HAMILTONIAN CYCLE $\rightarrow$ TSP $\square$

Definition 19.4:
Let $K \in \mathbb{Z}_+$. Let B_1 be the set of boolean expressions involving one variable: $x_1, \neg x_1, x_2, \neg x_2, ..., x_n, \neg x_n$ the *Literale (literals)*.

Let B_2 be the set of boolean expressionsof the form $(w_1 \vee w_2 \vee ... \vee v_k)$ with $w_i \in B_1$ and $1 \leq k \leq K$, the *Klauseln (clauses)*

Let B_3 be the set of boolean expressions of the form $(w_1 \wedge w_2 \wedge ... \wedge v_k)$ with $w_i \in B_2$ and $1 \leq k \leq K$. A boolean expression $f(x_1, x_2, ..., x_n) \in B_3$ is *erfüllbar (satisfiable)* if an assignment $x_i := \alpha_i \in \{0, 1\}$ exists such that $f(\alpha_1, \alpha_2, ..., \alpha_n) = 1$.

The *K-SAT Problem (K-SAT Problem)* is defined as: Given $f \in B_3$, is f satisfiable? (Every clause of f has at most K literals)

Corollary 19.5:
The 3-SAT Problem is $\mathcal{NP}$-complete.

Proof. Since SAT $\in \mathcal{NP}$, also 3-SAT$\in \mathcal{NP}$. We give a polynomial time transformation from SAT to 3-SAT.

Define a variable Y_g for every subexpression g of f (f instance of SAT), i.e. every part of f that is a boolean boolean expression again. Now f is satisfiable if and only if the following system is satisfiable:

$$\begin{aligned} Y_g &= Y_{g'} \vee Y_{g''} \text{ if } g = g' \vee g'' \\ Y_g &= Y_{g'} \vee Y_{g''} \text{ if } g = g' \vee g'' \\ Y_g &= \neg Y_{g'} \text{ if } g = \neg g' Y_f = 1 \end{aligned}$$

The equality $Y_g = Y_{g'} \vee Y_{g''}$ can also be written as

$$Y_g \vee \neg Y_{g'} = 1, \neg Y_g \vee Y_{g'} \vee Y_{g''} = 1 \text{ or } (Y_g \vee \neg Y_{g'}) \wedge (Y_g \vee \neg Y_{g''}) \wedge (\neg Y_g \vee Y_{g'} \vee Y_{g''}) = 1$$

. Similarly $Y_g = Y_{g'} \wedge Y_{g''}$ can be written as

$$(\neg Y_g \vee Y_{g'}) \wedge (\neg Y_g \vee Y_{g''}) \wedge (Y_g \vee \neg Y_{g'} \vee Y_{g''}) = 1$$

The expression $Y_G = \neg Y_{g'}$ equals

$$(Y_g \vee Y_{g'}) \wedge (\neg Y_g \vee \neg Y_{g'}) = 1$$

All expressions above has at most $K = 3$ literals and thus belong to B_2. They all have to be satisfied simultaneously. Thus, f is satisfied if and only if this 3-SAT instance is satisfied.

The transformation is polynomial since only $O(n^2)$ subexpressions exist, where n is the number of literals in f. □

Definition 19.6:
The *exakte Überdeckung (Exact Cover)* problem is defined as follows: given a collection $\mathcal{C}$ of subsets of a finite set X, does there exist a subcollection of $\mathcal{C}$ that partitions X, that means every element of X is in exactly one subset of the subcollection.

Theorem 19.7:
EXACT COVER is $\mathcal{NP}$-complete.

Proof. EXACT COVER $\in \mathcal{NP}$ since a solution $(Y_1, ..., Y_l)$ with $Y_i \in \mathcal{C}$ can be verified easily.

We reduce 3-SAT to EXACT COVER in polynomial time. Let $f = w_1 \wedge ... \wedge w_k$ with $w_i \in B_2$ ($K = 3$) and $x_1, ..., x_m$ the veriables of a 3-SAT instance.

Define a bipartite graph G two color classes: the $\{w_1, ..., w_k\}$ and the variables $\{x_1, ..., x_m\}$. An edge $\{w_i, x_j\}$ exists if and only if literal x_j or $\neg x_j$ is part of clause w_i. Hence, every verte w_i has degree ≤ 3. Let X be the set of all vertices *and* edges.

Further, let $\mathcal{C} = \mathcal{C}' \cup \mathcal{C}''$ with $\mathcal{C}'$ being the collection of all sets of the form $\{w_i\} \cup E'$ with E' being a nonempty subset of the edges adjacent to w_i (for every w_i at most 7 subsets E' exist). □

Lecture 20 (2011-12-22)**:**

Proof. □ der Beweis

Theorem 20.1:
Directed Hamiltonian Cycle (DHC) is $\mathcal{NP}$-complete

Proof. Verifying whether a sequence of vertices includes a DHC can be done in polynomial time. Hence, the problem is a member of $\mathcal{NP}$.

Next, we reduce EXACT COVER to DHC. Let $\mathcal{C} = \{C_1, ..., C_m\}$ the collection of subsets of a ground set $\mathcal{X} = \{x_1, ..., x_k\}$.

We construct a digraph $D = (V, A)$ with $V = \{r_0, ..., r_m\} \cup \{s_0, ..., s_k\} \cup T$ (T is to be defined).

Graphen von Valentina einfügen

For $i = 1, ..., m$, let $C_i = \{x_1, ..., x_{i_t}\}$. Construct the following digraph:

The orange vertices are either between r_{i-1} and r_i in a Hamilton Cycle or between s_{i_l-1} and s_{i_l} for $l = 1, ..., t$. Finally add the arcs $(r_m, s_0$ and (s_k, r_0). Hence, the overall graph looks like:

Graphen von Valentina einfügen

Thus, s_k has only one outgoing arc (s_k, r_0), s_0 has only one incoming arc (r_m, s_0) and from s_i to s_j we have to use 3 vertices from T corresponding to a subset C_j with $x_{i+1} \in \mathcal{C}_j$. In such cases (r_{j-1}, r_j) hasve to be used on the path from r_0 to r_m. Since only one subset of T can be used from s_i to s_{i+1}, all other sets C_k with $x_{i+1} \in C_k$ have to be visited on the path from r_{k-1} to r_k implying that all $x \in C_k$ are visited this way.

Now, on the one hand, if there exists a subcollection $\{C_{j1}, ..., C_{jk}\}$ which partitions $\mathcal{X}$, then the path between S_{i-1}, and s_i choose $(r_{j-1}\Delta r_j)$ as part of the Hamilton cycle, and for all $C_j \notin B$ follow the path via the orange vertices from r_{j-1} to r_j. Then, all vertices are visited exactly once, hence the digraph is Hamiltonian.

On the other and, if there exists a Hamilton cycle D, B can be constructed by selecting C_j if (r_{j-1}, rj) is part of the cycle and otherwise not. In summary there exists an EXACT COVER if and only if D is Hamiltonian. And the reduction is polynomial.

□

Theorem 20.2:
UNDIRECTED HC is $\mathcal{NP}$-complete.

Proof. We reduce DHC to UHD. Let D be a digraph. Replace every vertex v by three vertices v', v', and v'''. Construct edges $\{v', v''\}$ and $\{v'', v'''\}$. Further, replace (v, w) by $\{v', v'''\}$.

G is Hamiltonian if and only if D is Hamiltonian can be easily verified. □

TSP

INPUT: A complete graph $G = (V, E)$, a length function l on E and a rational r.
FIND: Does there exists a Hamiltonian cycle with length $\leq r$?

Theorem 20.3:
TSP is $\mathcal{NP}$-complete.

Proof. Let $G = (V, E)$ be an instance of UHC. Define G' as complete graph on vertex set V. Let $l(e) := 0$ for all $e \in E$ and $l(e) := 1$ for edges $e \notin E$. Then G is Hamiltonian, if and only if G' has a Hamilton cycle of length ≤ 0. □

Lecture 21 (2012-01-09)**:**

HAPPY NEW YEAR!!

Definition 20.4:
A problem Π for which it is unknown to be in $\mathcal{NP}$can be reduced polynomially to Π is called *NP-schwer (NP-hard)*. The optimization versions of $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard: most times it is unknown whether an optimal solution can be verified in polynomial time. Special cases of $\mathcal{NP}$-complete problems are sometimes solveable in polynomial time. For example, Undirected Hamilton Cycle is polynomail solveable if G is complete, if $\Delta(G) \leq 2$, or $\delta(G) \leq \frac{n}{2}$ ($n \geq 3$, see Graphentheorie I)

What do the shortest path, the maximum flow problem, the minimum cut proble, the min cost flow problem, the matching problem and Min Spanning Tree have in common, whereas TSP, Knapsack, and Independent Set (all $\mathcal{NP}$-complete) do not have?

Question 20.4.1:
Do they have a common property? Does there exist a unifying solution algorithm for these problems?

Answer: Jein. Yes and No. Most of the above (but matching only in bipartite graphs, Mininmal Spanning Tree not at all) can be solved as Linear Programming (LP) problems

Linear Programming crash course (Optimierung A)

- Vector $x \in \mathbb{R}^n$ of *variables* (unknowns)
- Contraints (Nebenbedingungen) are described by a matrix $A \in \mathbb{R}^{m \times m}$ and a vector $b \in \mathbb{R}^m$: $Ax \leq b$ (m linear constraints)
- Objective (Zielfunktion): $c^T x$ with $c \in R^n$ objective coefficients vector

- LP (in standard form):

$$(LP): \begin{cases} \max_{\text{subject to s.t.}} & c^T x \\ & Ax \leq b \\ & x \geq 0 \end{cases}$$

- $P = \{x \in \mathbb{R}^n : \ Ax \leq b, x \geq 0\}$ is the *feasible region* and defines a *polyhedron*(Polyeder)
- Every non-vertex (nicht-Ecke) $x \in P$ can be written as convex combination of vertices $x^1, \ldots, x^k \in P$:
  $x = \sum_{i=1}^{k} \lambda^i x^i, \sum_{i=1}^{k} \lambda^i = 1, \qquad \lambda^i \geq 0, \ i = 1, \ldots, k$
- for every objective $c^T x$, there exists at least one vertex i with $c^T x^i \geq c^T x \, \forall \, x \in P$.
- The LP can be solved by a sophisticated sequence of vertices of P:
  Phase 1: find a vertex of $P.x^{(a)} = 0$ (for example)
  Phase 2: As long as there exists a neighboring vertex x of $x^{(i)}$ with $c^T x > c^T x^{(i)}$ define $x^{(i+1)} := x$ and set $i := i + 1$.
  This method is known as the *Simplex Algorithm*(Dantzig, 1947)
- The Simplex Algorithm requires (in theory) exponentially iterations to find an optimal solution: not a polynomial time algorithm.
- *Innere Punktmethoden (Interior Point Methods)* follow a path through the interior of the polyhedron and either end up at the unique optimal solution(vertex) or at a convex combination of optimal vertices(i.e. at the optimal face of the polytope).
- There exist interior point methods that run in polynomial time: Khachiyan(1979) developed the first "ellipsoid" method, Karmarkar's projective algorithm followed 1984. The most effective methods are nowadays algorithm based on so-called *barrier* functions, incorporating the boundary of the polyhedron by increasing the function value to infinity.
- Thus LP $\in \mathcal{P}$
- In practice, a variant of the Simplex algorithm often outperforms interior point methods.

Many combinatorial optimization problems (both $\in \mathcal{P}$ and $\mathcal{NP}$-complete) can be modelled as *integer* linear programs (ILP): $max\{c^T x : Ax \leq b, x \in \mathbb{Z}_+^n\}$

Example 21.0(+1):
Matching problem in $G = (V, E)$ Let A be the $V \times E$ incidence matrix: every row is indexed by a vertex of G, every column by an edge of G, and for $v \in V$ and $e \in E$:

$$A_{v,e} := \begin{cases} 1 & \text{if } v \in e \\ 0 & \text{otherwise} \end{cases}$$

Then the maximum matching problem equals $max\{\mathbf{1}^x : Ax \leq \mathbf{1}, x \in \mathbb{Z}_+^n\}$ or stated otherwise

$$\begin{aligned} max & \sum_{e \in E} 1 \cdot x_e \\ s.t. & \sum_{v \in e} x_e \leq 1 \qquad v \in V \\ & x_e \geq 0 \qquad e \in E \\ & x_e \in \mathbb{Z} \qquad e \in E \end{aligned}$$

In general $max\{c^T x : Ax \leq b, x \in \mathbb{Z}_+^n\} \leq max\{c^T x : Ax \leq b, x \geq 0\}$. For $G = K_3$ the matching problem has ILP value 1, whereas LP$= \frac{3}{2}$.

Theorem 21.1:
(Karp,1972) Integer Linear Programming (ILP) is $\mathcal{NP}$-complete.

Question 21.1.1:
How should A resp. b look like to have equality in 1?

Lecture 22 (2012-01-12)**:**

Question 22.0.2:
How should A resp. b look like to have

$$\max\{c^T x \mid Ax \leq b, x \in \mathbb{Z}^n\} = \max\{c^T x \mid Ax \leq b, x \geq 0\}$$

Answer (partly): A should be a *total unimodular (totally unimodular)* matrix.

Definition 22.1:
A matrix A is totally unimodular (TUM) if every $k \times k$ submatrix of A has determinant equal to -1, 0 or 1. In particular, every entry of A has to be -1,0 or 1.

Theorem 22.2:
Let A be a TU $m \times n$ matrix and let $b \in \mathbb{Z}^m$. Then every vertex of the polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ is integer.

Proof. Let z be a vertex of P and define A_z as the submatrix of A consisting of the rows a_i of A with $a_i z = b_i$. From linear programming theory, it follows that z is a vertex of P if and only if $rang(A_z) = n$.

Consequently, A_z contains a non-singular $n \times n$ submatrix A'. Let b' be the corresponding subvector of b. By definition, $A'z = b'$ and accordingly, $z = (A')^{-1}b'$.

From TU of A, it follows that $|\det(A')| = 1$ and all entries of $(A')^{-1}$ are integer (Cramer's rule). Since $b \in \mathbb{Z}^n$, it follows $z \in \mathbb{Z}^n$. □

Definition 22.3:
A polyhedron P is *ganzzahlig (integer)* if for all $c \in \mathbb{R}^n$ with $\max\{c^T x \mid x \in P\}$ finite the maximum is attained by an integer vector x.

Stated otherwise, if $P = \{x \mid Ax \leq b\}$ with A a matrix of rang n, then P is integer if and only if all vertices of P are integer.

Corollary 22.4:
If A is TU and $b \in \mathbb{Z}^m$, then $P = \{x \mid Ax \leq b\}$ is integer.

Theorem 22.5 (Hoffman-Kruskal, 1956)**:**
Let A be an integer $m \times n$ matrix. Then A is TU if and only if for all $b \in \mathbb{Z}^m$, $P = \{x \mid Ax \leq b, x \geq 0\}$ is integer.

Proof. To prove the forward direction, we define
$\overline{A} = \begin{pmatrix} A \\ -\mathrm{I} \end{pmatrix}$ with I the $n \times n$ identity matrix and $\overline{b} = \begin{pmatrix} b \\ 0 \end{pmatrix}$. Then $P = \{x \mid \overline{A}x \leq \overline{b}\}$.

Every square submatrix of $\overline{A}$ consists of a part of the identity matrix and a part of A. To compute the determinant, we perform a Laplacian expansion along the rows of the identity part.

Hence, the determinant equals either zero or plus/minus the determinant of a smaller submatrix. Thus $\overline{A}$ is TU if and only if A is TU. By Corollary 22.4, P is integer.

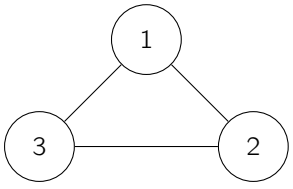
For the reverse, assume that A is not TU. Then, there exists a $r \times r$ submatrix B with $\det(B) \notin \{-1, 0, 1\}$ and $r \leq \min\{n, m\}$. W.l.o.g. let $B = (a_{ij})_{1 \leq i,j \leq r}$.

Since B is non-singular, B^{-1} exists. By integrality of A, $\det(B)$ is integer and $\det(B) \cdot \det(B^{-1}) = 1$, $\det(B^{-1}) \notin \mathbb{Z}$. Thus B^{-1} has at least one column j with a non-integer entry: $B^{-1}e_j \notin \mathbb{Z}^r$.

We construct $b' \in \mathbb{Z}^r$ as follows. Find $y \in \mathbb{Z}^r$ such that $B^{-1}e_j + y \geq 0$ and define $b' = (b'_1, \ldots, b'_r) = B(B^{-1}e_j + y) = e_j + By \in \mathbb{Z}^r$.

But the unique solution to $Bz = b'$ is $z = B^{-1}e_j + y \notin \mathbb{Z}^r$.

Now extend z as $\overline{x} := \begin{pmatrix} z \\ 0 \end{pmatrix} \notin \mathbb{Z}^n$ and define for $j = r+1, \ldots, m$, $b_j \in \mathbb{Z}$ such that $a_j\overline{x} < b_j$, e.g. $b_j = \lceil a_j\overline{x} \rceil + 1$. Then $\overline{b} := (b'_1, \ldots, b'_r, b_{r+1}, \ldots, b_m) \in \mathbb{Z}^m$. Further $\overline{x} \geq 0$ and $A\overline{x}$, hence $\overline{x} \in \{x \in \mathbb{R}^n \mid Ax \leq \overline{b}\}$ with he first r rows satisfied with equality. Since $\det(B) \neq 0$, these rows are linearly independent. By LP theory it follows that $\overline{x}$ is a vertex of P, but $\overline{x}$ ist non-integer, a contradiction. □



	1	2	3	
1	1	1	0	$\in R_1$
2	1	0	1	$\in R_2$
3	0	1	1	$\in R_3$

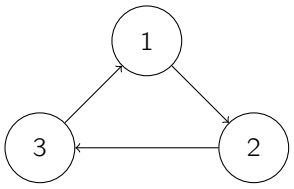
not TU by Hoffman.

A bipartite graph $G = (S \cup T, E)$ looks like

R_1	s_1	1		1
	s_2			
	$\vdots$		1	
R_2	t_1	1		1
	t_2			
	$\vdots$		1	

Definition 22.6:
Let $D = (V, A)$ be a digraph. The $V \times A$ incidence matrix M of D is defined by

$$M_{V,a} := \begin{cases} +1 & \text{if } a = (v, w) \text{ for some } w \in V \\ -1 & \text{if } a = (w, v) \text{ for some } w \in V \\ 0 & \text{otherwise} \end{cases}$$



$\Rightarrow$

	1	2	3	
1	1	0	-1	$\in R_1$
2	-1	1	0	$\in R_2$
3	0	-1	1	$\in R_3$

Lecture 23 (2012-01-19)**:**

Theorem 23.1:
The incidence matrix M of digraph D is totally unimodular.

Proof. Let B a square $t \times t$ submatrix of M. We prove $\det(B) \in \{-1, 0, 1\}$ by induction on t.

For $t = 1$, the result is trivial.

For $t > 1$, we distinguish three cases:

Case 1: B has a column with only zeros. Then $\det(B) = 0$.

Case 2: B has a column with exactly one non-zero. By Laplacian expansion, it follows that $\det(B) = \pm\det(B')$ where B' is the $t - 1 \times t - 1$ submatrix after deleting the row and column containing the non-zero.

By induction, $\det(B') \in \{-1, 0, 1\}$ and hence $\det(B)$ as well.

Case 3: Every column of B contains two non-zeros, exactly one 1 and one -1. If we take the linear combination of all rows with $\lambda_i = 1, i = 1, \ldots, t$, the result is a zero vector, hence the rows are linear depending and thus $\det(B) = 0$.

□

Corollary 23.2:
The shortest path problem, the max flow problem and the min cost flow problem have, written as LP, only integer vertices. (If the capacities of the arcs are integer).

Proof. Follows from Theorem 23.1 and Corollary 22.4. □

LP Duality

Theorem 23.3:
Let A be a $m \times n$ matrix, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$. Further, let $P = \{x \mid Ax \leq b\}$ and $Q = \{x \mid A^T y = c, y \geq 0\}$.

(i) *schwache Dualität (weak duality)*
If $x \in P$ and $y \in Q$, then $c^T x \leq b^T y$

(ii) *starke Dualität (strong duality)*
$\underbrace{\max\{c^T x \mid Ax \leq b\}}_{\text{primal LP}} = \underbrace{\min\{b^T y \mid A^T y = c, y \geq 0\}}_{\text{dual LP}}$
if both P and Q are non-empty

Proof. Optimierung A, exercise sheet. □

Total Dual Integrality (TDI)

Definition 23.4:
A rational system $Ax \leq b$ is *total dual integral (total dual integral)* if for every integer vector c such that $z_{LP} = \max\{c^T x \mid Ax \leq b\}$ exists (finite), the dual $\min\{b^T y \mid A^T y = c, y \geq 0\}$ has an integer optimal solution.

Example 23.4(+1):
Let $b_1, b_2 \in \mathbb{Z}$, $c_1, c_2 \in \mathbb{Z}$ and consider the primal problem

$$\begin{array}{lll} \max & c_1 x_1 + c_2 x_2 & \\ s.t. & x_1 + x_2 & \leq b_1 \\ & 2x_1 + x_2 & \leq b_2 \end{array}$$

The dual problem is

$$\begin{array}{lll} \min & b_1 y_1 + b_2 y_2 & \\ s.t. & y_1 + 2y_2 & = c_1 \\ & y_1 + y_2 & = c_2 \\ y_1 \geq 0, y_2 \geq 0 & & \end{array}$$

Hence, $y_1 = 2c_2 - c_1$, $y_2 = c_1 - c_2$.
If $c_2 > c_1$, the primal problem is unbounded ($x_1 \to -\infty$, $x_2 \to \infty$).
If $c_1 > 2c_2$, the primal problem is unbounded ($x_1 \to \infty$, $x_2 \to -\infty$).
If $c_1 \in [c_2, 2c_2]$, the primal problem is finite and the optimal dual solution is integer.
$\Rightarrow Ax \leq b$ is TDI.

Theorem 23.5:
If $Ax \leq b$ TDI and b integer, then $P = \{x \mid Ax \leq b\}$ is integer. (Primal integrality is for free)

Proof. If $Ax \leq b$ TDI and b integer, then the dual LP $\min\{b^T y \mid A^T y = c, y \geq 0\}$ has an integer optimal solution value (if it exists) since an optimal solution y^* is integer, b is integer, and thus $b^T y^*$ is integer.

LP duality $z_{LP} = \max\{x^T c \mid Ax \leq b\} = \min\{b^T y \mid A^T y = c, y \geq 0\}$ implies, that t_{LP} integer for all $c \in \mathbb{Z}^n$ for which the maximum exists.

Now, assume that nevertheless P is not integer. Then there exists a vertex $\tilde{x}$ of P and $1 \leq j \leq n$ with $\tilde{x}_j$ fractional. By polyhedral theory, there exists a $c \in \mathbb{Z}^n$ such that $\tilde{x}$ is the unique optimal solution of $\max\{x^T x \mid Ax \leq b\}$.

Since $\tilde{x}$ is the unique optimal solution there exists a big value $w \in \mathbb{N}$ such that $\tilde{x}$ is also optimal for objective $\overline{x} := x + \frac{1}{w} e_j$.

Now, we scale $\overline{c}$ with w: $\tilde{c} := w \cdot \overline{c} = w \cdot c + e_j \in \mathbb{Z}^n$. Scaling does not change the set of optimal solution vectors, and thus $\tilde{x}$ remains optimal.

It holds $\underbrace{\tilde{c}^T \tilde{x}}_{\in \mathbb{Z}} - \underbrace{w \cdot c^T \tilde{x}}_{\in \mathbb{Z}} = (w \cdot c^T \tilde{x} + e_j^T \tilde{x}) - w \cdot c^T \tilde{x} = e_j^T \tilde{x} = \tilde{x}_j \notin \mathbb{Z}$.

Hence, one of the two values must be fractional: a contradiction. □

Theorem 23.6:
Let P be a rational polyhedron. Then, there exists a rational TDI system $Ax \leq b$ with A integer and $P = \{x \mid Ax \leq b\}$. The vector B can be chosen integer, if P is integer.

Proof. See Cook, Cunningham, Pulleyblank, Schrÿver 1998, Theorem 6.30 □

Implication: for "almost all" combinatorial optimization problems, there exists a system $Ax \leq b$ TDI. Four step procedure:

1) Find an appropriate system $Ax \leq b$ with A, b integral

2) Prove $Ax \leq b$ TDI

3) Use Theorem 23.5 to show that $P = \{x \mid Ax \leq b\}$ is integral

4) Solve the LP $\max\{c^T x \mid Ax \leq b\}$.

The difficulty is step 1 (we often don't know) and the sze of A might imply the LP cannot be solve in polynomial time (move in GLO, WS 12/13).

Example 23.6(+1):
Matching can be solved by LP if G is bipartite. If Q is not bipartite $\begin{pmatrix} A \\ -I \end{pmatrix} \leq \begin{pmatrix} b \\ 0 \end{pmatrix}$ with A the node-edge-incidence matrix is not TDI. *But* there exists a $\tilde{A}$ and $\tilde{b}$ such that $\tilde{A}x \leq \tilde{b}$ TDI and $\max\{c^T x \mid \tilde{A}x \leq \tilde{b}\} = \nu(G, c)$.

Question 23.6.1:
What can we do if we don't know $\tilde{A}$, $\tilde{b}$?

Lecture 24 (2012-01-23)**:**

Approximation of Combinatorial Optimization Problems

Question 24.0.2:
What do we wo, if the problem is $\mathcal{NP}$-hard and branch and bound takes too long?

Approximate the optimal solution.

Minimum spanning Tree heuristic for TSP (Lecture 12)

(a) Determine a MST T_1 in the complete graph $G = (V, E)$ with $c(e) \geq 0$ for all $e \in E$

(b) Double the edges and choose a directed Euer tour T_2

(c) Follow the Euer tour along the orientation, replace vertices already visited by the shortcut to the next unvisited vertex.

Question 24.0.3:
How good is the solution in comparison to the optimal solution?

Definition 24.1:
Let Π be a combinatorial optimization problem, and let

$$\Pi' := \{P \in \Pi : \ P \text{ has a feasible solution}\}$$

be the set of feasible problem instances. For every problem instance $Pin\Pi'$, denote with $C_{OPT}(P)$ the optimal solution value of P.
Further, let A be an algorithm that generates a feasible solution for every problem instance $P \in \Pi'$. The evalue of this solution is denoted by $C_A(P)$. To avoid a time consuming case-by-case analysis, we assume $C_{OPT}(P) > 0$ for all $P \in \Pi'$, and set $C_{OPT}(P) := C_A(P) := 1$ for all $P \in \Pi \setminus \Pi'$.

(a) Let $\varepsilon > 0$. In case Π is a maximization problem, in addition $\varepsilon \leq 1$. If for all problem instances $P \in \Pi$

$$R_A(P) := \frac{|C_A(P) - C_{OPT}(P)|}{C_{OPT}(P)} \leq \varepsilon$$

then A is an *e-Approximationsalgorithmus (e-approximation Algorithm)* and and the value ε is the *(approximation guarantee)* (since $C_A(P) \geq (1 - \varepsilon)C_{OPT}(P)$).

If Π is a minimization problem, $1 + \varepsilon$ is the worst-case bound of A (since $C_A(P) \leq (1 + \varepsilon)C_{OPT}(P)$).

Question 24.1.1 (reformulated)**:**
Does the MST heuristic of TSP have a worst case bound $1 + \varepsilon < \infty$?

Definition 24.2:
The symmetric TSP is *metrisch (metric)*, if all distances fulfill the triangle inequality, i. e. for all triples i, j, k of vertices

$$c_{ik} \leq c_{ij} + c_{jk}$$

holds.

Theorem 24.3:
The MST heuristic for metric TSP has a worst-case bound of $1 + \varepsilon = 2$

Proof. If we remove an edge from a tour, we obtain a spanning tree. Since the dinstances in a metric TSP are non-negative, it follows that the MST solution value is less than the optimal tour $C_{OPT}(P)$. Thus,

$$c(T_2) = 2 \sum_{i,j \in T_1} c_{ij} \leq 2 \cdot C_{OPT}(P).$$

The tour constructed from T_2 is optained by replace paths between vertices by directed edges. Since the trangle inequality holds, the length of the new edges is at most the length of the path. Hence

$$C_{MST}(P) \leq C(T_2) \leq 2 \cdot C_{OPT}(P)$$

□

Question 24.3.1:
Can we do better?

Christofides heuristic for TSP:

(a) Determine a MST T_1

(b) Let W be the set of vertices in (V, T_1) with odd degree (note $|W|$ is even)

  (a) Determine a minimum weight perfect matching M in $G[W]$ ($G[W]$ is complete)

  (b) Define $T_2 := T_1 \cup M$ (in (V, T_2) all vertices have even degree). Choose an oriented Euler tour along T_2

  (c) See MST heuristic

Theorem 24.4:
The Christofides heuristic is a $\frac{1}{2}$-approximation for a metric TSP

Proof. We have to estimate the value of the minimum weight perfect matching. Let $\{i_1, i_2, \ldots, i_{2m}\} =: W$, where the numbering has ben taken according to the order in T_{OPT} (optimal tour), i.e.,

$$T_{OPT} = (i_1, \alpha_1, i_2, \alpha_2, ..., l_{2m-1}, \alpha_{2m-1}, i_{2m}, \alpha_{2m})$$

where α_i is a (possible empty) sequence of vertices. We consider two matchings

$$M_1 := \{i_1 i_2, i_3 i_4, ..., i_{2m-1} i_{2m}\}$$
$$M_2 := \{i_2 i_3, i_4 i_5, ..., i_{2m-2} i_{2m-1}, i_{2m} i_1\}$$

By the triangle inequality [in particular, $C(i_1, \alpha_1) + c(\alpha_1, i_2) \leq c(i_1, i_2)$] it holds

$$C_{OPT}(P) = C(T_{OPT}) \geq C(M_1) + C(M_2)$$

Since matching M is optimal, it further holds that

$$C_{OPT}(P) = C(T_{OPT}) \geq C(M_1) + C(M_2) \geq C(M) - 2$$

and thus $C(M) \leq \frac{1}{2} C_{OPT}(P)$. Similar to the MST heuristic

$$C_{Christofides}(P) \leq C(T_2) = C(T_1) + C(M) \leq \frac{3}{2} C_{OPT}(P)$$

The approximation ratio of Christofides heuristic is up to today the best known approximation reatio for metric TSPs.

□

For symmetric TSPs, the situation is different (worse):

Theorem 24.5:
If there exists a $\varepsilon > 0$ and a polynomial time algorithm H such that for every symmetric TSP instance P

$$C_{OPT}(P) \leq C_H(P) \leq (1 + \varepsilon) C_{OPT}(P)$$

then $\mathcal{P} = \mathcal{NP}$, i.e, the ε-approximation problem of symmetric TSP is $\mathcal{NP}$-complete.

Proof. UNDIRECTED HAMILTON CYCLE is $\mathcal{NP}$-complete. We show that if H is a polynomial ε-approximation algorithm for symmetric TSP, then there exists a polynomial time algorithm for UHC.
Let $G = (V, E)$ be a graph with $n = |V|$ and let $M := \varepsilon \cdot n + 2$. We define a symmetric TSP on n cities with the following distances

$$c_{ij} := \begin{cases} 1 & \text{if } ij \in E \\ M & \text{otherwise} \end{cases}$$

Now, it holds

$$G \text{ is Hamiltonian} \Leftrightarrow C_{OPT}(P) = n$$

If a tour T does not correspond to a Hamilton cycle, we have

$$c(T) \geq n - 1 + M = n - 1 + n \cdot \varepsilon + 2 > (1 + \varepsilon) \cdot n$$

Now, let T_H be the tour generated by H. If G is Hamiltonian,

$$C_{OPT}(P) = n \leq C(T_H) \leq (1 + \varepsilon) C_{OPT}(P) = (1 + \varepsilon) \cdot n$$

Hence, $C(T_H) = n$ and the Hamilton cycle is found in polynomial time. □

$\Rightarrow$ the symmetric TSP is not approximable, unless $\mathcal{P} = \mathcal{NP}$

Lecture 25 (2011-01-26)**:**

Definition 25.1: (a) An *Approximationsschema (approximation scheme)*(AS) for problem Π is an algorithm A taking two inputs a problem instance $P \in \Pi$ an a rational number $\varepsilon > 0$. Algorithm A generates for *every* input $P \in \Pi$ and *every* $\varepsilon > 0$ a solution with $R_A(P) \leq \varepsilon$.

(b) An AS A for Π is a *polynomielles Approximationsschema (polynomial (time) approx. scheme)*(PAS/PTAS) if A has running time polynomial in the input length of $P \in \Pi$

(c) An AS A for Π is a fully-polynomial (time) approximation scheme (FPAS/F-PTAS) if A has running time polynomial in the input length of $P \in \Pi$ and polynomial in $\frac{1}{\varepsilon}$

Note that a FPTAS is not polynomial in the total input: to store $\varepsilon = \frac{p}{q}$ (resp. $\frac{1}{\varepsilon}$) we need $< \varepsilon > \approx \log p + \log q$ bits.

Theorem 25.2:
Let Π be a combinatorial optimization problem. If there exists a FPAS for Π which is also polynomial in $< \varepsilon >$, then there exist a polynomial time algorithm to solve all problem instances $P \in \Pi$.

Proof. If A is a FPAS, then the coding length of both values $c_A(P)$ and $c_{opt}(P)$ are polynomial in the input length of $P \in \Pi$. If A is polynomial in $< \varepsilon >$, we can define a polynomial time algorithm B for Π as follows:
Let $\varepsilon := \frac{1}{2}$ and apply A to $P \in \Pi$ yielding the value $C_{A,\varepsilon}(P)$
In case of maximization problem, define $\rho := \frac{1}{2c_{A,\varepsilon}(P)+1}$
(The case of minimization: Übungsblatt)
Apply A to P with precision δ, yielding $c_{A,\delta}(P)$
The running time of B is polynomial in $< P >$, since $\varepsilon = \frac{1}{2}$ is a constant and thus $c_{A,\varepsilon}$ and δ have coding length polynomial in $< P >$ only. W.l.o.g. we assume that the objective is integer. Thus, $c_{A,\varepsilon}(P)$, $c_{A,\delta}(P)$ and $c_{opt}(P)$ are integer. We now show $c_{A,\delta}(P) = c_{opt}(P)$.
if Π is a maximization problem, it holds

$$c_{A,\varepsilon}(P) \geq \frac{1}{2} c_{opt}(P) \Rightarrow 2c_{A,\varepsilon}(P) + 1 > c_{opt}(P)$$

$$c_{A,\delta}(P) \geq (1-\delta)c_{opt}(P) = (1-\frac{1}{2c_{A,\varepsilon}(P)+1}c_{opt}(P) > (1-\frac{1}{c_{opt}(P))c_{opt}(P)} = c_{opt}(P)-1$$

combined with integrality of the objective, we have

$$c_{A,\delta}(P) = c_{opt}(P)$$

Similar for minimization problems. $\square$

Example 25.2(+1):
PAS for SUBSET SUM = Knapsack with $c_i = a_i$

$$max \sum_{i=1}^{n} a_i x_i$$
$$s.t. \sum_{i=1}^{n} a_i x_i \leq b = \frac{A}{2}$$
$$x_i \in \{0,1\}\ \forall i \in \{1,\ldots,n\}$$

max $\frac{A}{2} \Leftrightarrow$ PARTITION exists!
Let x^* be an optimal solution. A PAS for SUBSET SUM takes $a_i \in \mathbb{Z}_+$ and $0 < \varepsilon = \frac{p}{q} \leq 1$ The output is a feasible solution $\overline{x}$ such that

$$\sum_{i=1}^{n} a_i \overline{x}_i \geq (1-\varepsilon) \sum_{i=1}^{n} a_i x_i^*$$

(1) Set $k := \lceil \frac{p}{q} \rceil = \lceil \frac{1}{\varepsilon} \rceil$

(2) Partition the index set $N := \{1,\ldots,n\}$ in "large" (L) and "small" (S) indices:

$$L := \{j \in N : a_j \geq \frac{b}{k}\}$$
$$M := \{j \in N : a_j < \frac{b}{k}\}$$

(3) Solve the SUBSET SUM instance:

$$max \sum_{j \in L} a_j x_j$$
$$s.t. \sum_{j \in L} a_j x_j \leq b \qquad x_j \in \{0,1\}$$

to optimality (eg. by enumeration of all solutions). Let $\overline{x}_j, j \in L$, be an optimal solution of (SSP) and let

$$P := \{j \in L . \overline{x}_j = 1\}, \qquad b' := \sum_{j \in L} a_j \overline{x}_j$$

(4) In case for all $j \in S, b' + a_j > b$, no small element $j \in S$ can be added anymore. Set $x_j = 0\ \forall j \in S$. STOP

(5) Otherwise, find $a_j \in S$ such that $b = (b' + a_j) \geq 0$ and this difference is minimized. Set

$$P := P \cup \{j\} \qquad b' := b' + a_j$$
$$S := S \setminus \{j\} \qquad \overline{x}_j := 1$$

and go to 4.

Theorem 25.3:
The algorithm is a PAS for SUBSET SUM.
The time to solve (SSP') is $O(n^k)$ by complete enumeration of all subsets of L of at most k elements.

Proof. In case $S \subset P$ at the end of the algorithm, i.e., $\overline{x}_j = 1 \forall j \in S$, then $\overline{x}$ is optimal for (SSP): Assume there exists a better solution x' for (SSP). By optimality of step (3) for (SSP') we have $\sum_{j \in L} a_j \overline{x}_j \geq a_j x_j^*$. Then

$$\sum_{j \in L} a_j x_j^* + \sum_{j \in S} a_j x_j^* \overset{x_j^* \leq 1}{\leq} \sum_{j \in L} a_j x_j^* + \sum_{j \in S} a_j \overset{S \subset P}{\leq} \sum_{j \in L} a_j \overline{x}_j + \sum_{j \in S} a_j \overline{x}_j$$

Contradiciton.
Now, assume that for at least one $j_0 \in S, \overline{x}_{j_0} = 0$ and let b^* be the optimal solution. We derive

$$\sum_{j \in N} a_j \overline{x}_j = b' > b - a_{j_0} > b - \frac{b}{k} = b(1 - \frac{1}{k}) \geq b^*(1 - \frac{1}{k})$$

and hence $b' > (1 - \varepsilon) b^*$. So, the solution is an ε-approximation.
Runningtime:

- Step (2) requires $O(n)$ time.
- Step (4) and (5) also require $O(n)$ time.
- Step (3) requires $O(\binom{n}{k} k)$ time (if we enumerate all subsets of k elements, $k+1$ elements exceed the capacity for sure)

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{1}{k!} n \cdot (n-1)(n-2) \ldots (n-k+1) = O(\frac{n^k}{k})$$

Overall: $O(n) + O(n) + O(\binom{n}{k} k) = O(n^k)$ for fixed k, this polynomial in n but not polynomial in k.

□

Lecture 26 (2011-01-30)**:**

Dynamic Programming for Knapsack Revisited

$$\max \sum_{j=1}^{n} c_j x_j \qquad \text{s.t. } \sum_{j=1}^{n} a_j x_j \leq b, x_j \in \{0,1\} \; \forall j \in \{1, \ldots, n\}$$

Dynamic Programming:

$$f(r, d) = \max\{\sum_{j=1}^{r} c_j x_j \mid \sum_{j=1}^{m} a_j x_j = d, x_j \in \{0,1\}, j \in \{1, \ldots, r\}\}$$

Recursive formula:

$$f(r,d) = \max\{f(r-1,d), f(r-1,d-a_r)+c_r\}$$
$$f(r,0) = 0$$
$$f(1,d) = \begin{cases} 0 & \text{if } d = 0 \\ c_1 & \text{if } d = a_1 \\ -\infty & \text{otherwise} \end{cases}$$

Theorem 26.1:
The computational complexity to compute $\max_{x=0,\dots,b} f(n,d)$ is $\mathcal{O}(n \cdot b)$. The algorithm is polynomial in n and b, but not polynomial in the input size.
For the latter, the running time must be polynomial in $\langle b \rangle$.

Proof. Trivial. □

Definition 26.2:
An algorithm is *pseudopolynomiell (pseudo-polynomial)* if it is polynomial in the input values (instead of the coding length). A $\mathcal{NP}$-complete problem Π is *schwach NP-vollständig (weakly NP-complete)* if there exists a pseudo-polynomial algorithm. Otherwise, the problem is *stark NP-vollständig (stronly NP-complete)*, i.e., even if the size of the numbers in the input of the problem is bounded polynomially, the problem remains $\mathcal{NP}$-complete.

Corollary 26.3:
Knapsack is weakly $\mathcal{NP}$-complete.

Sort $\frac{c_1}{a_1} \geq \dots \geq \frac{c_n}{a_n}$.

Listing 3: Greedy algorithm

```
1 b' := b
2 FOR j := 1,...,n DO
3           IF b' ≥ a_j THEN
4                     x_j := 1
5                     b' := b' − a_j
6           ELSE
7                     x_j := 0
8 ENDFOR
```

It holds that $c_{\text{Greedy}}(P) > c_{\text{OPT}}(P) - \max_{i=1,\dots,n} c_i$.

Proof. The LP relaxation is an upper bound for c_{OPT} and takes the same items until $b' \leq a_j$ for the first time. In this case, it takes a fraction of the item. So the value up to here plus the next item is an upper bound for sure. In the worst case, this item has maximum c_j and no further items are added by the Greedy algorithm. □

FPTAS for Knapsack

Listing 4: FPTAS for Knapsack

```
1  INPUT: a_j, c_j ∈ ℤ_+ with c_1/a_1 ≥ ··· ≥ a_n/c_n, b ∈ ℤ_+, two parameters s, t
2  OUTPUT: Approximate solution
3
4  1) Estimate the optimal solution value:
5          Let k be the largest index such that Σ_{j=1}^{k} a_j ≤ b
6          c_est := Σ_{j=1}^{k} c_j
7          It holds: c_OPT ≤ c_est ≤ 2 · c_OPT
8          In case k = n, the optimal solution is
               x_j = 1 ∀j = 1,... n, STOP
9
10 2) Decompose the index set:
11         L := {j ∈ {1,... n} | c_j ≥ t} (large)
12         S := {j ∈ {1,... n} | c_j < t} (small)
13
14 3) Solve the special "Equality" Knapsack problems
15         SEKP_d: min Σ_{j∈L} a_j x_j, s.t. Σ_{j∈L} ⌊c_j/s⌋ x_j = d  x_j ∈ {0,1} ∀j ∈ L
16     for d = 0,..., ⌊c_est/s⌋
17         Idea: we try to fill the knapsack with large
               items with weight as small as possible and
               value close to the optimal solution.)
18
19 4) For d = 0,..., ⌊c_est/s⌋ apply:
20         a) Let x_j^d, j ∈ L, be an optimal solution of SEKP_d
21            If Σ_{j∈L} a_j x_j^d ≤ b, then apply Greedy to the
                  following KP (SKP_d):
22                max Σ_{j∈S} c_j x_j  s.t. Σ_{j∈S} a_j x_j ≤ b − Σ_{j∈L} a_j x_j^d = b_d,
                      x_j ∈ {0,1} ∀j ∈ S
23         b) Let x_j^d, j ∈ S be the greedy solution of SKP_d,
               then x_j^d, j ∈ {1,..., n} is a solution of KP.
24
25 5) Choose the best of the ⌈c_est/s⌉ + 1 solutions found.
```

Theorem 26.4:
Let c_A be the solution value of the described algorithm. Then

$$c_A \geq c_{\mathrm{OPT}} - \left(\frac{s}{t}c_{\mathrm{OPT}} + t\right) = (1 - \frac{s}{t})c_{\mathrm{OPT}} - t$$

Proof. Let $x_1^*, \ldots, x_n^*$ be an optimal solution of KP. Set $d := \sum_{j\in L} \lfloor \frac{c_j}{s} \rfloor x_j^*$. Then

$$d \leq \lfloor \frac{1}{s} \sum_{j\in L} c_j x_j^* \rfloor = \lfloor \frac{1}{s} c_{\mathrm{OPT}} \rfloor \leq \lfloor \frac{1}{s} c_{est} \rfloor$$

Hence, the value d has been considered in step 3). Let $\overline{x}_j := x_j^d, j \in L$, be the candidate solution. (From the optimal solution, it is clear that SEKP_d is not

infeasible, thus $\overline{x}$ exists). By definition of d, it further holds that $b_d \geq 0$ and thus the solution can be extended in step 4). It follows:

$$c_A \geq \sum_{j=1}^{n} c_j \overline{x}_j = c_{\mathrm{OPT}} - \left(\underbrace{\left(\sum_{j \in L} c_j x_j^* - \sum_{j \in L} c_j \overline{x}_j \right)}_{1} + \underbrace{\left(\sum_{j \in S} c_j x_j^* - \sum_{j \in S} c_j \overline{x}_j \right)}_{2} \right)$$

1 can be bounded by

$$\begin{aligned}
\sum_{j \in L} c_j x_j^* - \sum_{j \in L} c_j \overline{x}_j &\leq \underbrace{s \cdot \sum_{j \in L} \left\lfloor \frac{c_j}{s} \right\rfloor x_j^*}_{''d} + \sum \left(c_j - s \left\lfloor \frac{c_j}{s} \right\rfloor \right) x_j^* - \underbrace{s \cdot \sum \left\lfloor \frac{c_j}{s} \right\rfloor \overline{x}_j}_{''d} \\
&= \sum_{j \in L} \underbrace{\left(c_j - s \cdot \left\lfloor \frac{c_j}{s} \right\rfloor \right)}_{\leq s} x_j^* \\
&\leq s \cdot \sum_{j \in L} x_j^* + \overset{c_j \geq t}{\leq} \frac{s}{t} \sum_{j \in L} c_j x_j^* \\
&\leq \frac{s}{t} c_{\mathrm{OPT}}
\end{aligned}$$

2 can be indirectly bounded by

$$\begin{aligned}
c_{\mathrm{Greedy}}^{\mathrm{SKP}_d} = \sum_{j \in S} x_j \overline{x}_j > c_{\mathrm{OPT}}^{\mathrm{SKP}_d} - \max_{j \in S} x_j \geq c_{\mathrm{OPT}}^{\mathrm{SKP}_d} - t \\
\Rightarrow t > c_{\mathrm{OPT}}^{\mathrm{SKP}_d} - c_{\mathrm{Greedy}}^{\mathrm{SKP}_d} \geq \sum_{j \in S} c_j x_j^* - \sum_{j \in S} c_j \overline{x}_j
\end{aligned}$$

In summary, $c_A \geq c_{\mathrm{OPT}} - \left(\frac{s}{t} c_{\mathrm{OPT}} + t \right)$ □

Theorem 26.5:
Let $\varepsilon > 0$, set $s := \left(\frac{\varepsilon}{3} \right)^2 c_e s t$ and $t := \frac{\varepsilon}{3} c_e s t$. Then

(a) $c_a \geq (a - \varepsilon) c_{OPT}$

(b) The running time of A is $\mathcal{O}(n \log n) + \mathcal{O}(n \left(\frac{1}{\varepsilon} \right)^2)$, i.e. A is a FPTAS.

Proof. (a) $t = \frac{\varepsilon}{3} c_e s t \leq \frac{2\varepsilon}{3} c_{OPT}$. Thus

$$c_A \geq c_{OPT} - \left(\frac{\varepsilon}{3} c_{OPT\Delta} + \frac{2\varepsilon}{3} c_{OPT} \right) = (1 - \varepsilon) c_{OPT}$$

(b) To apply the greedy algorithm, the items have to be sorted, requiring $\mathcal{O}(n \log n)$= time. To compute s and $t, \mathcal{O} \log \varepsilon + n)$ operations are needed. The solution of (SEKP_d) requires $\mathcal{O}\left(n \lfloor \frac{c_e s t}{s} \rfloor \right) = \mathcal{O}\left(\frac{1}{\varepsilon} \right)$. In step 4 the greedy algorithm is applied $\mathcal{O}\left(\frac{1}{\varepsilon^2} \right)$ times, each time requiring $\mathcal{O}(n)$ operations, hence also $\mathcal{O}\left(\frac{n}{\varepsilon^2} \right)$.

□

Matroids and Independence Systems

Question 27.0.1:
The mininum spanning tree problem can be solved by the greedy algorithm: always take as next edge a minimum weight one that does nbot close a cycle. Are there other problems where the greedy algorithm solves the problem to optimality?

General Independdence Systems(Unabhängigkeitssystem)
Let E be a finite set. 2^E denotes the set of all subsets of E.

Definition 27.1:
A set $\mathcal{I} \subseteq 2^E$ is an *Unabhängigkeitssystem (independence system)*(IS) on E, if $\mathcal{I}$ satisfies the following conditions:

(I.1) $\emptyset \in \mathcal{I}$

(I.2) $F \subseteq G \in \mathcal{I} \Rightarrow F \in \mathcal{I}$

often, we call the pair $(E, \mathcal{I})$ an IS. The subsets of E contained in $\mathcal{I}$ are *unabhängige Mengen (independent sets)*; all other subsets of E are *abhängige Mengen (dependent sets)*. the inclusion-wise minimal dependent sets of E are the *(circuits)*(or*(cycles)*), i.e., $C \subseteq E$ is a circuit if C is dependent and $C \setminus \{i\}$ is independenbt for all $i \in C$.
The set of all circuits is called the *(circuit system)* and denoted by $\mathcal{C}$. Given $F \subseteq E$, every independent subset of F which is not contained in another independent subset of F is called a *Basis (basis)* of F, i.e.,

$$B \text{ basis of } F \Leftrightarrow (B, B' \in \mathcal{I}, B \subseteq B' \subseteq F \Rightarrow B = B')$$

The set of all bases of the ground set E is called the *Basissystem (basis sytem)*(regarding $\mathcal{I}$) and is denoted by $\mathcal{B}$.
For every set $F \subseteq E$ is the *Rang (rank)* of F defined by

$$r(F) := max\{|B| : B \text{ basis of } F\}$$

Example 27.1(+1):
Let $\mathcal{I}$ be the set of all forests of E.

- Circuits: cycles in $G = (V, E)$
- basis of $F = E$: spaning tree (if G is connected)
- basis vof $F \subset E$: spanning trees in the connected components of (V, E)
  $\rightarrow$ maximum weighted Spanning tree is a special case of Def(27.3)
  All bases are spanning trees and thus have #vertices-1 edges per component
  $\Rightarrow$ (27.4) is satisfied
- $r(E) = n - 1$ if G is connected
- $r(F) = n -$ #components of $(V(F), F)$

Lemma 27.2:
For an independent set $F \subseteq E$, it holds taht

(a) F is a basis F

(b) $r(F) = |F|$

for a circuit $F \subseteq E$ it holds that

(a) $F \setminus \{i\}$ is a basis of F for all $i \in F$

(b) $r(F) = |F| - 1$

In general it holds that

(a) $r(F) \leq |F|$ subcardinality

(b) $F \subseteq G \Rightarrow r(F) \leq r(G)$ monotonicity

(c) $F, F_i (i \in K)$ with $F_i \subseteq F$ and $|\{i \in K : e \in F_i\}| = k \; \forall e \in F$.
It holds

$$k \cdot r(f) \leq \sum_{i \in K} r(F_i) \qquad \text{strong subadditivity}$$

No proof

Definition 27.3:
Let $c : E \to \mathbb{R}$ be a weight function on E. For $F \subseteq E$ let $c(F) := \sum_{e \in F} c(e)$.
The optimization problem on the IS $\mathcal{I} \subseteq 2^E$ is

$$max\{c(I) : I \in \mathcal{I}\}$$

Example 27.3(+1):
$G = (V, E)$. A subset $S \subseteq V$ is stable(clique) if every pair in S is non-adjacent.
The set of sable sets(cliques) is an IS on V
The maximum weighted stable set problem(clique) can be described as an IS optimization problem.
Note those two problems are $\mathcal{NP}-$complete.

Definition 27.4:
A *(matroid)* $\mathcal{M}$ consits of a ground set F and an IS $\mathcal{I} \subseteq 2^E$ which satisfies the following *equivalent* conditions:

(I.3) $I, J \in \mathcal{I}, \; |I| = |J| - 1 \Rightarrow \exists j \in J \setminus I$ with $I \cup \{j\} \in \mathcal{I}$

(I.3') $I, J \in \mathcal{I}, \; |I| < |J| \Rightarrow \exists K \subseteq J \setminus I$ with $|I \cup K| = |J|$ such that $I \cup K \in \mathcal{I}$

(I.3") $F \subseteq E$ and B, B' bases of $F \Rightarrow |B| = |B'|$

matching is not matroid
A matroid defined on a graph is called a *(graphical matroid)*

Greedy algorithm for IS

```
1  1 Sort the weights in non-increasing order
2  2 Set I_g := ∅
3  3 FOR i=1 To n DO
4    IF c_i ≤ 0 THEN GOTO 4
5    IF I_g ∪ {i} is independent, then set I_g := I_g ∪ {i}
6    ENDFOR
7  4 Return I_g.
```

To analyze, let $F \subseteq E$ and define the *unter Rangfunktion (lower rank function)* of F

$$r_L(f) := min\{|B| : B \text{ basis of } F\}$$

Further, let

$$q := F \subseteq E, \overset{min}{r(f) > 0} \frac{r_L(F)}{f(F)} \text{ be the } \textit{Rangquotient (rank quotient)}$$

Theorem 27.5:
Let $\mathcal{I}$ be an IS on E. Let I_g be the greedy solution and let I_{OPT} an optimal solution of $max\{c(I : I \in \mathcal{I}\}$ Then

$$q \leq \frac{c(I_g}{c(I_{OPT}} \leq 1$$

and for every IS there exists weights $c(e) \in \{0, 1\}$ $\forall e \in E$ such that the left inequality is satisfied with equality.

No proof

Corollary 27.6:
An IS $\mathcal{I}$ on E is a matroid if and only if for all weight functions $C : E \to \mathbb{R}$ or $\{0, 1\}$ the greedy algorithm provides an optimal solution.

Acronyms

(s,t)-**cut** (s,t)-Schnitt. 13

(s,t)-**flow-value** Wert des (s,t)-Flusses. 32

M-**alternating forest** M-alternierender Wald. 25

M-**blossom** M-Blüte. 24

k-**regular** k-regulär. 4

(S-T**)-separator** . 30

(s-t**)-cut** $(s$-$t)$-Schnitt. 31

(s-t**)-vertex cut** . 31

acyclic azyklisch. 8

adjacency matrix Adjazenzmatrix. 3

algorithm Algorithmus. 37

antiparallel entgegengesetzt. 8

approximation guarantee . 52

approximation scheme Approximationsschema. 55

arc disjoint bogendisjunkt. 30

arcs Pfeil. 3

assymmetric assymetrisch. 29

augmenting cycle augmentierender Kreis. 35

augmenting network augmentierendes Netzwerk. 35

backward-arc Rückwärtsbogen. 33

basis Basis. 61

basis sytem Basissystem. 61

bipartite bipartit. 18

boolean expression Boolescher Ausdruck. 42

capacity constraints Kapazitätsbedingungen. 32

certificate(solution) Zertifikat(Lösung). 39

chain Kette. 8

circle Kreis. 4