

**Lecture 2** (2011-10-13):**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  $c(W) := \sum_{e \in W} c_e$  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  $v : u$ . 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, starting from  $w$  to find a second vertex of degree 1.  $\square$

**Corollary 2.4:**

A tree  $G = (V, E)$  with maximum degree  $\Delta$  has at least  $\Delta$  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 for  $G$  as well.  $\square$

**Lemma 2.6:**

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

## Algorithm MST

$\min_{x \in X} = -\max_{x \in X} - c(x)$  maximal forest

$X$  spanning trees

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

### 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  $\{e_1, e_2, \dots, e_k, e'\}$ . Let  $c_f = \max_{i=1, \dots, k} c_{e_i}$ . 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 between  $T'$  and  $T$  is reduced by one,  $c(T'_{\text{new}}) \leq c(T') < c(T)$ . Repeating the procedure results in  $c(T) \leq \dots < c(T)$ , a contradiction.  $\square$

**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) to 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 algorithm 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 edges 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:  $r(u) = \# \text{trees containing } u$ . 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)*  $(v_0, a_1, v_1, a_2, v_2, \dots, a_k, v_k)$ ,  $k \geq 0$ , with  $a_i(v_{i-1}, v_i) \in A$  and  $v_k = v_0$ . In particular,  $D$  does not contain *(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 *(outdegree)* of  $v$  is  $\deg_D^+(v) = |\delta_D^+(v)|$  (assuming simple digraph)

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

**Definition 3.1:**

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

**Question 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 permutation  $\sigma : V \rightarrow \{1, \dots, n\}$  of the vertices such that  $\deg_{D[v_1, \dots, v_n]}^-(v_i) = 0$  for all  $i = 1, \dots, n$  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  $(v_2, \dots, v_{n+1})$  with

$$\deg_{D[v_1, \dots, v_{n+1}]}^-(v_i) = 0 \quad \forall i = 2, \dots, n + 1$$

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

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

**Theorem 3.3:**

A  $[s, t]$ -path exists in a acyclic Digraph  $D = (V, A)$  if and only if in all permutations  $\sigma : V \rightarrow \{1, \dots, n\}$  with  $\deg_{D[v_1, \dots, v_n]}^-(v_i) = 0$  for all  $i = 1, \dots, n$ , it holds that  $\sigma(s) < \sigma(t)$ .

*Proof.* Assume there exists a permutation  $\sigma$  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.  $\square$

### Question 2:

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

To simplify notation, let  $V = \{1, \dots, n\}$ ,  $s = 1$ ,  $t = n$  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   $D(i) = \{\infty : i < n \text{ and } NEXT(i) = NIL, 0 : i = n\}$ 
2  FOR  $i = n - 1$  DOWNTO 1 DO
3       $D(i) = \min_{j=i+1, \dots, n} \{D(j) + c(i, j)\}$  with  $c(i, j) = \infty$  if  $(i, j) \notin A$ 
4       $NEXT(i) = \text{Argmin}_{j=i+1, \dots, n} \{D(j) + c(i, j)\}$ 

```

### 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  $(a_1, \dots, a_k)$  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

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

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.  $\square$

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

### Theorem 3.5:

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

*Proof.* of runtime:

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

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

### 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  $n$ ,  $i = 2, \dots, n$ .

Observation 2: We can use a similar procedure for the shortest path from 1 to  $i$ ,  $i = 2, \dots, n$ . (with  $PREV(i)$  for previous instead of  $NEXT(i)$ ).

### Question 3:

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  $i = 1, \dots, n$  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 graphs. 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  $i = n, n - 1, \dots, j + 1$ . 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 avev 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 algorithm 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  $i, i = 1 \dots n$ , provided all weights  $\geq 0$ .

*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 shortest  $[1, i]$ -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  $k = 0, \dots, n$  we show that  $W^k$  is the matrix of shortest path lengths with vertices  $1, \dots, k$  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  $i \in \{1, \dots, n\}$

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  $\{1, \dots, k\}$  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  $\{1, \dots, k\}$  (voluntary). If the shortest path from  $i$  to  $j$  using  $\{1, \dots, k + 1\}$  does not contain  $k + 1$ , it only contains  $\{1, \dots, k\}$  (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  $[k + 1, j]$ -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  $\{1, \dots, k + 1\}$ , 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  $\{1, \dots, k\}$ ) and algorithm would have stopped earlier.  $\square$

**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.* Follows from ?? □

**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  $C_1, \dots, C_d$  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  $V_i = \{v \in V : \exists (s, v)\text{-path with } c(P) \leq i - 1\}$

$$\begin{aligned} v_1 &= \{s\} \\ v_2 &= \{5, 3, 4\} \\ v_3 &= \{5, 2, 3, 4\} \\ v_4 &= v_3 \cup \{6\} \end{aligned}$$

(for the example graph on the board)

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

Procedure:

---



```

1  Sort the items such that  $\frac{c_1}{a_1} \leq \frac{c_2}{a_2} \leq \dots \leq \frac{c_n}{a_n}$ .
2
3  Set  $S = \emptyset$ .
4  For  $i = 1$  to  $n$  do
5      if  $(a(s) + a_i \leq b)$  then
6           $S = S \cup \{i\}$ 
7      endif
8
9  endfor
10 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:

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

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

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

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

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

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

**Theorem 6.3:**

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

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

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

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

*Proof.* Let  $x^*$  be an optimal solution with  $c^T x^* > c^T \tilde{x}$ . 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

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

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.  $\square$

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  $\{1, \dots, k\}$ .

**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  $k+1$ .  $\square$

**Corollary 6.5:**

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

**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  $e \cap e' = \emptyset \forall e, e' \in M$  with  $e \neq e'$ .

A matching  $M$  is called *perfect* 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  $|M'| > |M|$ .

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

## Further related graph parameters

$\alpha(G)$  = Stable set number / independent set number (Stabile-Menge-Zahl), size of a maximum stable set in  $G$ : a subset  $S \subseteq V$  such that  $\forall \{v, w\} \in E \mid |\{v, w\} \cap S| \leq 1$

$\rho(G)$  = Edge cover number (Kantenüberdeckungszahl), minimum size of an edge cover of  $G$ : a subset  $F \subseteq E$  such that  $\forall v \in V : \exists e \in F : v \in e$ .

$\tau(G)$  = Vertex cover number (Knotenüberdeckungszahl), minimum size of a vertex cover of  $G$ : a subset of  $W \subseteq V$  such that  $\forall e \in E : e \cap W \neq \emptyset$ .

### 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  $\rho(G) \leq |E|$ .  $\square$

### Lemma 7.4:

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

*Proof.* Exercise sheet.  $\square$

### 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  $\alpha(G) \leq \rho(G)$ .  $\square$

**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,  $\nu(G) \leq \tau(G)$ .  $\square$

[B

**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

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

$\square$

For all graph parameters, there exists a weighted version:  $\nu(G, w), \tau(G, w), \alpha(G, w), \rho(G, w)$ .

**Question 4:**

How do we find a maximum (weighted) matching?

**Definition 7.8:**

Let  $M$  be a matching in  $G$ . A path  $P = (v_0 e_1 v_1 \dots e_r v_r)$  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

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

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

*Proof.* Trivial.  $\square$

**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 and  $M$  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 components (Komponente) of  $G' = (V, M \cup \bar{M})$ . Since 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  $M$ .  $\square$

**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  $j$ .

$$\nu(G) \leq \tau(G) \text{ vertex cover}$$

**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. Hence, the length of  $P$  is even. Consequently,  $P$  does not contain  $v$  (otherwise  $P$  ends at  $v$  which contradicts the bipartiteness of

$G : P \cup \{u, v\}$  would be an odd cycle). The path  $P \cup \{v, v\}$  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; contradiction 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  $\tau(G) \leq \nu(G)$ .  $\square$

**Corollary 8.4:**

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

*Proof.* Follows from Thm 7.7 (Gallai's Thm) and 8.3  $\square$

### 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  $e = \{v, w\}$  ( $u \in U, w \in W$ ) 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  $U' := U \setminus \left( \bigcup_{e \in M} \right)$ ,  $W' := W \setminus \left( \bigcup_{e \in M} \right)$ . 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. This is guaranteed by the direction of the edges in the digraph. 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.  $\square$

## Acronyms

**$(s, t)$ -cut**  $(s, t)$ -Schnitt. 7

**acyclic** azyklisch. 3

**antiparallel** . 3

**bipartite** bipartit. 13

**chain** Kette. 3

**connected** zusammenhängend. 1

**directed cycles** . 3

**forest** Wald. 1

**indegree** . 3

**Knapsack problem** Knapsack Problem. 8

**linear relaxation** Lineare Relaxierung. 9

**matching** Paarung. 11

**matching number** Paarungszahl. 11

**Maximum Forest Problem** Problem des maximalen Waldes. 1

**Minimum Spanning Tree (MST) problem** minimaler Spannbaum. 1

**outdegree** . 3

**running time of algorithms** Laufzeit. 2

**shortest path** . 3

**shortest path length matrix** ????. 6

**spanning** aufspannend. 1

**tree** Baum. 1

---