Combinatorial Optimization 1

Lecture notes, University of Technology Graz

Based on a lecture by Prof. Eranda Dragoti-Çela

Extended during a lecture by Prof. Bettina Klinz

Lukas Prokop

March 21, 2018

Contents

Course organization

This lecture took place on 2nd of Oct 2014.

Practicals every 2 weeks 2 hours. Homepage at opt.math.tu-graz.ac.at.

Partial exams:

- 28th of Nov 2014
- 30th of January 2014

Contents:

- graph theory
- linear optimization
- algorithms for linear optimization
- spanning trees and tree views
- integer programming
- shortest path problem
- network flows
- matching problems
- matroids

The lecture and therefore these lecture notes are heavily inspired by the book "Kombinatorische Optimierung".

Introduction

A generic combinatorial optimization problem

Synonym for "Instance". Input, given.

Synonym for "Task". Output.

An instance has a finite base set $E = \{e_1, \dots, e_n\}$. The set of valid solutions is a subset $F \subseteq 2^E = \mathcal{P}(E)$. One valid solution is $F \in F$.

$$c : F \to \mathcal{R}$$

$$F \mapsto c(F)$$

A task is some $F^* \in F$ with $c(F^*) = \min_{F \in F} c(F)$ (minimization problem). Some $F^* \in F$ with $c(F^*) = \max_{F \in F} c(F)$ is a maximization problem.

Possible common cost models

$$w : E \to \mathcal{R}$$

$$e \mapsto w(e)$$

Where w is called weight. Two common cost functions:

$$c(F) := \sum_{e \in F} w(e) \qquad \text{(sum problem)} \tag{1}$$

$$c(F) := \max_{e \in F} w(e) \qquad \text{(bottleneck problem)} \tag{2}$$

Problem 1: Drill machine problem

A drill machine must drill holes onto a board. Drill heads move vertically. Boards move horizontally. Movements happen with constant speed. Movements can happen simultaneous. Drilling is not considered as movement. With these assumptions the time necessary to drill all holes is direct proportional to the path taken by the drill head and board.

$$\text{production time} \propto \text{path(drill head, board)}$$

The time taken to move from hole 1 to hole 2 is

$$\max\{|x_1 - x_2|, |y_1 - y_2|\}$$

The total costs to drill all holes are:

$$\sum_{i=1}^{n-1} \max\{|x_i - x_{i+1}|, |y_i - y_{i+1}|\}$$

where the relative coordinates of n holes is denoted as (x_i, y_i) with $1 \leq i \leq n$ and the holes are drilled in order $1, 2, \ldots, n$.

Is π a permutation of $\{1, 2, \ldots, n\}$ $(\pi \in S_n)$. If holes are drilled in order π, then

$$\text{production time} = \sum_{i=1}^{n-1} \max\{|x_{\pi(i)} - x_{\pi(i+1)}|, |y_{\pi(i)} - y_{\pi(i+1)}|\}$$

The drill machine problem as generic problem (sum problem):

$$c = \text{production time}$$
$$F = S_n$$
$$c : S_n \to \mathcal{R}$$
$$\pi \to \sum_{i=1}^{n-1} \max\{|x_{\pi(i)} - x_{\pi(i+1)}|, |y_{\pi(i)} - y_{\pi(i+1)}|\}$$
$$E = \{l_\infty(P_i, P_j) : 1 \leq i, j \leq n, i \neq j\}$$

Problem 2: Scheduling problem

Given. We have m workers and n tasks. We assume that every task must be completed by some worker. Not necessarily every task can by done by every worker. Let $S_i \subseteq \{1, 2, \ldots, m\}$ be the set of workers, that can complete job i; with $1 \leq i \leq n$. All workers of S_i complete task i with the same speed. Let t_i be the required time to complete job i. Every task can be completed by several workers. Every worker can work on several tasks, but *not* simultaneously.

Find. Define a work schedule to minimize the total time to complete all tasks (bottleneck problem).

We define t_{ij} as the length of time interval in which worker j completes job i such that $t_i = \sum_{j \in S_i} t_{i,j} \ \forall 1 \leq i \leq n$. The work time of worker j is defined as $\sum_{i:j \in S_i} t_{i,j}$ and time to complete is defined $\max_{1 \leq j \leq m, i:j \in S_i} \sum t_{ij} \to \min$. The completeness time is called "makespan".

Partial enumeration

Given. $n \in N, n \geq 3$, $\{p_1, p_2, \ldots, p_n\}$ are points on a plane, d is the distance function

Find. a permutation $\pi^* \in S_n$ with $c(\pi^*) = \sum_{i=1}^{n-1} d_\infty(p_{\pi^*}(i), p_{\pi^*}(i+1))$ minimized

```
1. Let $\pi(i) = i, \pi^*(i) = i, \ \forall 1 \leq i \leq n, i = n - 1$
2. Let $k = \min(\{\pi(i) + 1, \ldots, n + 1\} \setminus \{\pi(1), \pi(2), \ldots, \pi(i - 1)\})$
3. if $k \leq n$ then
     (a) Let $\pi(i) = k$
     (b) if $i = n$ and $c(\pi) < c(\pi^*)$ then $\pi^* = \pi$
     (c) if $i < n$ then set $\pi(i + 1) = 0$ and $i = i + 1$.
   if $k = n + 1$ then $i = i - 1$
   if $i \geq 1$ then goto 2
```

Working principle

In every step the algorithm finds the (lexicograpical) next possible value for $\pi(i)$ without duplicates of $\pi(1), \pi(2), \dots, \pi(i-1)$. If this is impossible, then reduce i by 1 (backtracking approach). Otherwise we set $\pi(i)$ to our new value k. If $i = n$ then a new permutation is given and costs are evaluated and compared, otherwise the algorithm tries all possible values $\pi(i+1) \dots \pi(n)$ and starts with $\pi(i+1) = 0$ with i being incremented successively.

Hence the algorithm generates all permutations in lexicographical order.

Algorithm efficiency

The actual costs can only be computed relative to n. We define costs in terms of steps. One step is defined as one arithmetic operation, assignment, comparison, logical statements, goto jump or value lookup ("elementary step", ES). We look at the algorithm in terms of costs:

1. $2n + 1$ ES
2. $O(n)$ with helper vectors auxiliary$(j) = 1$ if $j < i$.
3. Constant number of ES unless $i = n$, then $2n + 1$ additionally. In any case not more than $\leq O(n)$ ES.

How often are steps 2 and 3 of the algorithm executed? $O(n^2)$ if new permutation is not created (without backtracking) and $O(n)$ with backtracking. In total $O(1)$ every time $O(n^2)$ (without backtracking) or $O(n)$ every time $O(n)$ each $O(n^2)$ (with backtracking).

So the algorithm has computational complexity of $O(n^2 \cdot n!)$ ES.

Analysis of algorithms

This lecture took place on 6th of Oct 2014.

A finite, deterministic algorithm is a sequence of valid inputs and instructions which consists of elementary steps such that the computational task gets completed for every possible input. For every possible input the algorithm computes a finite, deterministic output.

Given. A sequence of numbers. If rational, then binary encodable:

$$e \in \mathbb{Z} \rightarrow_{\text{encodes}} \log|a| + 2$$

Logarithms are always considered with base 2 here.

Inputsize. We denote the size of the input for x with size(x). For some rational input x the size(x) is the number of 0 and 1 in the binary representation.

1. Let A be an algorithm which accepts inputs $x \in X$. Let $f : \mathbb{N} \rightarrow \mathbb{R}_+$. If there is some constant $\alpha > 0$, such that $A\ \forall x \in X$ terminates the computation after at maximum $\alpha f(\text{size}(x))$ elementary steps, we say "A has a *time complexity* of $O(f)$".

2. An algorithm A with rational inputs has a *polynomial* runtime (or "is polynomial") iff $\exists k \in \mathbb{N}_0$
   (a) A has a time complexity of $O(n^k)$
   (b) all intermediate values of the computation can be stored with $O(n^k)$ bits

3. An algorithm A with arbitrary input is called *strongly polynomial* if $\exists k \in \mathbb{N}_0$ such that A
   (a) requires for every of n numbers of the input a runtime of $O(n^k)$
   (b) is polynomial for every rational input

4. An algorithm A which is polynomial, but not strongly polynomial, is called *weakly polynomial.*

5. Let A be an algorithm which computes for every input $x \in X$ output $f(x) \in Y$. We state that A computes the function $f : X \to Y$. Is a function computable by a polynomial algorithm, we call it a *polynomial computable function*.

The runtime of a polynomial algorithm is a function of the input. The runtime of a strongly polynomial algorithm is a function of the *number* of input elements.

Remark. Let A have runtime complexity $O(n^2)$. This means not all instances of input length n require $\theta(n^2)$ elementary steps. $O(n^2)$ is an upper bound (worst-case time complexity).

Spanning trees and arborescences

$$G = (V, E) \qquad e \in E \text{ is } e = \{x, y\} \text{ with } x, y \in V$$

where $V(G)$ is the set of vertices of G and $E(G)$ is the set of edges of G. One of the earliest problems in combinatorial optimization is the computation of minimum spanning trees.

Minimum spanning tree problem (MST)

Given. Undirected graph $G, c : E(G) \to (R)$

Find. Find a spanning tree T with minimum weight $c(T) = \sum_{e \in T} c(e)$ in G or determine "G is not connected".

Maximum weight forest problem (MWF)

Given. Undirected graph $G, c : E(G) \to (R)$

Find. A spanning forest F (cyclefree subgraph with vertex set $V(G)$) with maximum weight

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

Equivalence of problems

Two problems are called *equivalent* if P is reducible to Q and Q is reducible to P. P is reducible to Q, if there are two linear computable functions f and g such that

1. for every instance I of P, $f(I)$ is an instance of Q
2. for every solution L of Q, $g(L)$ is a solution of P

$$(P[I]) \xrightarrow{f} (Q[f(I)], L) \xrightarrow{g} g(L)$$

MST and MWF are equivalent

Theorem 1. The MWF problem and MST problem are equivalent.

Reduce MWF to MST

MWF is reducible to MST. Let (G, c) be an instance of MWF. Remove all $e \in E(G)$ with $c(e) < 0$. Let $c'(e) = -c(e)$ for all remaining edges of $E(G)$. Insert a minimum set of edges F with arbitrary weights, such that the resulting graph is connected. Denote this graph with G'.

The computationally most intense task is insertion of the minimum set of edges. Determination of connected components is possible in linear time (eg. with DFS).

Consider instance (G', c') of the MST problem. Let T' be an optimal solution of MST with (G', c').

Remove F of T'. Let T be the resulting subgraph. Show that T is a spanning forest with maximum weight in (G, c). ($F \subseteq E(T')$ results from the definition of F as *minimum* …).

T must be spanning forest because T' is a spanning tree.

$$c(T) = -(c'(T') - c'(F)) = -c'(T') + c'(F)$$

$c'(F)$ is the constant available in all spanning tree of G'. If T' minimizes $c'(T')$ such that T of $c(T)$ is maximized.

Reduce MST to MWF

Let (G, i) be an instance of the MST problem. Let

$$c'(G) = \mathcal{K} - c(e) \text{ with } \mathcal{K} = \max_{e \in E(G)} c(e) + 1 \Rightarrow c'(e) > 0 \; \forall \, e \in E(G)$$

Consider (G, c') as instance of MWF (linear runtime). Let F be a maximum weight spanning forest in (G', c'). Case distinction:

F is not a tree: G is not connected

F is a spanning tree: F is the optimal solution of MST because

$$c'(F) = \sum_{e \in E(F)} (\mathcal{K} - c(e)) = (|V(G)|-1)\mathcal{K} - \sum_{e+E(F)} c(e) = (|V(G)|-1)\mathcal{K} - c(F)$$

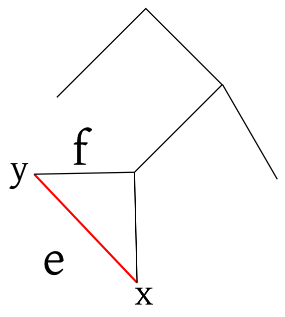

Figure 1: Sketch for proof construction $a \Rightarrow b$

Theorem 2. (Optimality conditions.) Let (G, i) be an instance of MST and T be a spanning tree in G. In this case the following statements are equivalent:

- T is optimal
- $\forall\, e = \{x, y\} \in E(G) \setminus E(T)$: no edge of the x-y-path in T has greater weight than e
- $\forall\, e \in E(T)$: If C is one of the connected components of $T \setminus \{e\}$, then e is an edge from $\delta(V(c))$ with minimum weight.
- $E(T) = \{e_1, e_2, \ldots, e_{n-1}\}$ can be ordered such that $\forall\, i \in \{1, 2, \ldots, n-1\}$ there is a set $X \subseteq V(G)$ such that $e_i \in \delta(X)$ with minimum weight and $e_j \neq \delta(X)\ \forall\, j \in \{1, 2, \ldots, i-1\}$.

Cut.

$$X \subset V(G)$$
$$\delta(X) = \{e \in E(G) : |e \cap X| = 1\}$$

Theorem 3. $a \Rightarrow b \Rightarrow c \Rightarrow d \Rightarrow a$.

$a \Rightarrow b$

$c(e) \geq c(f)$ for every f in x-y-path in T, because otherwise $T - e + f$ is a spanning tree with $c(T - e + f) = c(T) - c(e) + c(f) < c(T)$ which contradicts.

$b \Rightarrow c$

Show $\forall\, e \in E(T) : c(e) \leq c(f)\ \forall\, f \in \delta(V(c))$. Every edge $e \in T$ defines an cut $\delta(V(c)) = \delta(e)$. Hence we improve the tree.

$c \Rightarrow d$

Show there exists some weighted order and cuts with properties like in c. Select a random order $\{e_1, e_{i-1}, e_i, e_{n-1}\}$. $\forall\, e \in \{1, 2 \ldots, n-1\}$ consider cut $\delta(c_{e_i})$. It has the desired properties.

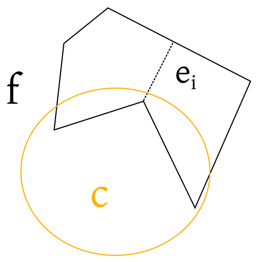

Figure 2: Sketch for $b \Rightarrow c$

$d \Rightarrow a$

Assumption $E(T) = \{e_1, \dots, e_{n-1}\}$ is satisfied.

Let T^* be an optimal spanning tree such that $i(T^*) = \min\{h \in \{1, 2, \dots, n-1\} : e_n \notin E(T^*)\}$ is maximum.

We show $i = +\infty$ and hence $T^* \equiv T \Rightarrow T$ is optimal. Assumption $i < +\infty$. Then $X \subset V(G)$ with $e_i \in \delta(v)$ with minimum weight with $e_j \notin \delta(X)\ \forall\, j < i$.

$$\exists f \neq e_i \text{ with } f \in T^* \cap \delta(X)$$

$$c(f) \geq c(e_i)$$

$$c(f) \leq c(e_i)$$

$$\Rightarrow c(f) = c(e_i)$$

$T^* - f + e_i$ is an optimal spanning tree and has $i(T^* - f + e_i) > i(T^*)$. This is a contradiction.

Kruskal's algorithm

This lecture took place on 7th of Oct 2014.

Algorithm 1 Kruskal's algorithm

Given. G is a connected undirected graph, $c : E(G) \to \mathbb{R}$
Find. minimum spanning tree

```
1: Sort edges $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$      $(m = |E(G)|)$
2: Set $T := (V(G), \phi)$
3: for $i$ from 1 to $m$ do
4:     if $T \cup \{e_i\}$ is cycle-free then
5:         $E(T) = E(T) \cup \{e_i\}$
6:     end if
7: end for
```

Theorem 4. Krukal's algorithm is correct.

From the previous theorem we can derive: $\forall\, e = \{x, y\} \notin E(T)$ we can say $c(f) \leq c(e)\ \forall\, f$ edges from the x-y-path in T.

$$(x, y) = e \notin E(T) \Rightarrow e_i \text{ closes cycle with } E(T) \cap \{e_1, \dots, e_{i-1}\}$$

$$\Rightarrow E(x - y - \text{path} \in T) \subseteq \{e_1, \dots, e_{i-1}\}$$

$$\Rightarrow\ \forall\, f \in E(x - y - \text{path}) : c(f) \leq c(e_i)$$

Trivial implementation. $O(m \cdot n)$ because there are m iterations and per iteration one check whether the current edge with the given T ($\leq n$ edges) creates a cycle ($O(n)$ with DFS).

Definition 5. A digraph G is called *branching*, if it is cycle-free and every $v \in V(G)$: indegree$(v) \leq 1$.

Notation. indegree$(v) = \deg^-(v)$.

Definition 6. A connected branching is called *arborescence*. The vertex r with $\deg^-(r) = 0$ is called *root*. An arborescence is the directed-graph equivalent of a rooted tree.

Notation.

$$\delta(\{v\}) = \delta(v)$$

$$\delta^+(v) = \{e = (v, y) \in E(G)\}$$

$$\delta^-(v) = \{e = (x, v) \in E(G)\}$$

Theorem 7. Let G be a digraph with n vertices. The following 7 statements are equivalent:

1. G is an arborescence with root r.
2. G is a branching with $n - 1$ edges and $\deg^-(r) = 0$.
3. G has $n - 1$ edges and every vertices is reachable from r.
4. Every vertex is reachable from r and removal of one edge destroys this property.
5. G satisfies $\delta^+(X) \neq 0\ \forall\, X \subset V(G)$ with $r \in X$. The removal of one arbitrary edge destroys this property.
6. $\delta^-(r) = 0$ and $\forall\, v \in V(G) \setminus \{r\}\ \exists$ one distinct directed $r - v$-path in G
7. $\delta^-(r) = 0$ and $|\delta^-(v)| = 1\ \forall\, v \in V(G) \setminus \{r\}$ and G is cycle-free.

A proof for Theorem 7 is not provided. It will be provided in the practicals.

Theorem 8. Kruskal's algorithm can be implemented with time complexity $O(m \log n)$.

Proof. The implementation keeps a branching B with

- $V(B) = V(G)$
- connected components of B vertex-correspond with the connected components of T

□

How can we create such a branching?

1. At the beginning (after initialization): $B = (V(G), 0)$.

Be aware that checking whether $\{v, w\}$ creates a cycle with T, consider that w must be in the same connected component in T (or B). We can check this in $O(\log n)$.

In branching:

- Let r_v (r_w) be the root of $v(w)$ contained solutions of B.

$$r_v = r_w$$

  The computational effort for checking is equivalent to the effort for the determination of r_v and r_w which is proportional to the sum of the lengths of r_v-v-path or r_w-w-path in B.

Hypothesis 9. In B it holds that $h(r) \leq \log n$ for every root r where $h(r)$ is the maximum length of an r-v-path in B.

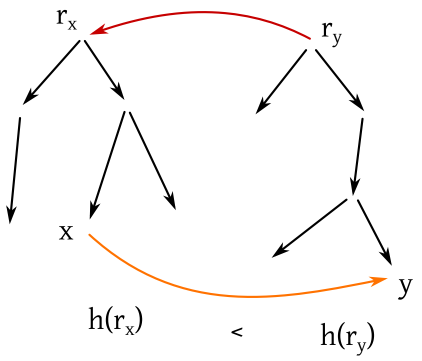

Figure 3: Branching insertion operation

Proof. Induction over number of edges in B.

Base. For zero edges this is trivial to prove.
Step. We will add a new edge $\{x, y\}$ and beforehand $h(r) \leq \log n$ is satisfied. We have to show that $h(r) \leq \log n$ is satisfied after the insertion of $\{x, y\}$.

Case distinction:

- $h(r_x) = h(r_y)$. Insert (r_x, r_y) or (r_y, r_x) In the figure we have to ensure $h(r_x) \leq \log r_x \Leftrightarrow m_x \leq 2^{h(r_x)}$.

$$\hat{h}(r_x) = h(r_x) + 1$$
$$n_{x,y} = n_x + n_y \geq 2^{h(r_x)} + 2^{h(r_y)}$$
$$= 2 \cdot 2^{h(r_x)} = 2^{h(r_x)+1}$$
$$= 2^{\hat{h}(r_x)}$$

- $h(r_x) < h(r_y)$. Insert (r_y, r_x).

$$\hat{h}(r_y) = h(r_y)$$

$$n_{xy} \geq n_y \geq 2^{h(r_y)} = 2^{\hat{h}(r_y)}$$

□

Prim's algorithm

Algorithm 2 Prim's algorithm

Given. G is a connected undirected graph, $c : E(G) \to \mathbb{R}$

Find. minimum spanning tree

```
1: Set $T = (\{v\}, 0)$ for an arbitrary $v \in V(G)$
2: while $V(T) \neq V(G)$ do
3:     Select one edge $e \in \delta_G(V(T))$ with minimum weight
4:     $T := T + e$
5: end while
```

Theorem 10. Prim's algorithm is correct and can be implemented with time complexity of $O(n^2)$. Correctness follows from theorem 2.2.d ($a \Rightarrow b \Rightarrow c \Rightarrow d \Rightarrow a$): Spanning tree is optimal $\Leftrightarrow$ order of edges $e_1, \ldots, e_{n-1}$ such that $\forall\, i \in \{1, 2, \ldots, n-1\}\ \exists x_i \subset V(G)$ with $e_i \in \delta(X_i)$ is the minimum edge in $\delta(X_i)$ and $e_j \notin \delta(X_i)$ is the cheapest edge of $\delta(X_i)$ and $e_j \notin \delta(X_i)\ \forall\, 1 \leq j \leq i-1$. This is satisfied by construction.

The desired order (or cuts) will be created by the algorithm.

Time complexity. The number of iterations is the number of edges in the tree which is $n-1$. We have to show that every iteration is completed in $O(n)$ time.

Maintain a list of candidate edges: $\forall\, w \notin V(T)$ is the candidate edge $K(w)$ the minimum edge between w and $V(T)$. In the general case we add the minimum edge $K(w)\ \forall\, w \notin V(T)$ $(T = T + k(w))$.

Definition 11. $|V(G) \setminus V(T)| = O(n)$ can be computed in $O(n)$ time.

Update of candidate edges: If v_k was added to T in the last iteration then compare $c(v_k, w), c(k(w))$ and if $c(v_k, w) < c(k(w))$ then $k(w) = (v_k, w)$. This requires $O(n)$ time.

Theorem 12. Is Prim's algorithm implemented with Fibonacci-Heaps we can solve the MST problem in $O(m + n \log n)$ time.

$$O(n^2) \qquad O(m + n \log n) \qquad m = \theta(n^2) \qquad G \text{ is dense}$$

A proof for Theorem 12 is not provided.

Number of spanning trees

This lecture took place on 9th of Oct 2014.

Theorem 13. (Arthur Cayley) The complete graph K_n has n^{n-2} spanning trees.

Proof. (J. Pitman, Coalescent random forests, Journal of Combinatorial Theory A 85, 1999, 165–193) Double-counting approach counting the number of labelled rooted trees (LRT). In labelled trees every edge has a label. Two LRTs are equivalent if and only if their tree structure is the same and labels are equivalent.

1. Let $\tau(n)$ be the number of spanning trees in K_n. Every tree can have n root candidates and $(n-1)!$ labels. In conclusion we can create $n \cdot (n-1)! \cdot \tau(n)$ LRTs with n vertices.

2. Insert edges successively such that adding $n-1$ vertices creates a LRT. For one edge we have …
$$2 \cdot \frac{n(n-1)}{2} = n(n-1) \text{ possibilities}$$
We added k edges ($k < n-1$). Followingly the graph has $n-k$ connected components with $n_1, n_2, \ldots, n_{m-k}$ vertices each. Every connected component is a LRT. If the $k+1$-th edge to be added starts at component 1, then this edge must have the root as source. This edge can have every other vertex as destination. There are $n - n_1$ possible such edges. In total we start with an arbitrary component and get:
$$(n-n_1) + (n-n_2) + \ldots + (n-n_{n-k}) = n(n-k) - n$$
This is the number of possilities for edge $k+1$. In total for all $n-1$ edges to add:
$$\prod_{k=0}^{n-2} (n \cdot (n-k) - n) = \prod_{k=0}^{n-2} n(\underbrace{n-k-1}_{1 \le t \le n-1}) = n^{n-1}(n-1)$$
It holds:
$$n(n-1)!\delta(n) = n^{n-1}(n-1)! \Rightarrow \delta(n) = n^{n-2}$$

□

Minimum Weight Arborescence Problem (MWA)

Given. Digraph $G = (V, E), c : E(b) \to \mathbb{R}$

Find. Spanning arborescence with minimum weight or claim $\nexists$ spanning arborescence in G

Minimum Weighted Rooted Arborescence Problem (MWRA)

Given. Digraph $G = (V, E), r \in V(G), c : E(G) \to \mathbb{R}$

Find. Spanning arborescence with root r and with minimum weight in G or claim $\nexists$ spanning tree with root r in G

Maximum Weighted Branching Problem (MWB)

Given. Digraph $G = (V, E), c : E(G) \to \mathbb{R}$

Find. Branching B with maximum weight

Equivalence of MWA, MWRA and MWB

Hypothesis 14. The three problems MWA, MWRA and MWB are equivalent.

Partially the proof is given in the practicals.

Proof. We assume without loss of generality that $c(e) \geq 0 \; \forall e \in E(G)$ because negative edges cannot occur in a maximum branching.

$$\deg^-(v) \leq 1 \; \forall v \in V(G)$$

Greedy approach: $\forall v \in V(G)$ select one $e_v \in \operatorname{argmax}\{c(e) : e = (x, v) \in E(G)\}$ let $B := \{e_v : v \in V(G)\}$.

If B_0 is cycle-free: B_0 is branching with maximum weight. Otherwise cycles have to be avoided / destroyed.

Theorem 15. Let B_0 be a subgraph of G with maximum weight and $\deg^-_{B_0}(v) \leq 1 \; \forall v \in V(G)$. Then $\exists$ an optimal branching $B \in G$ with properties $\forall$ cycle $C \in B_0 : |E(C) \setminus E(B)| = 1$.

Proof. Assumption. Such an optimal branching B does not exist.

Let B be a maximum branching in G with maximum number of common edges with B_0. Let C be a cycle in B_0 with $|E(C) \setminus E(B)| \geq 2$.

$$E(C) \setminus E(B) = \{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\}$$

in order they appear inside the cycle.

Hypothesis 16.

$$\forall 1 \leq i \leq k \exists b_i - b_{i-1} - \text{path in } B (b_0 \equiv b_k)$$

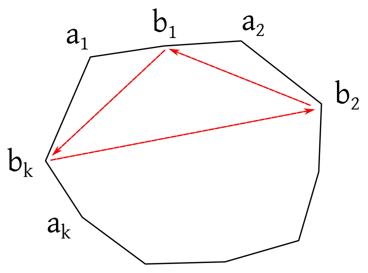

Figure 4: Red cycle

The existence of a red cycle in B shows a contradiction. □

Consider a fixed $i \in \{1, 2, \ldots, k\}$. Let B'_i be a subgraph of G with $V(B'_i) = V(G)$ and $E(B'_i) = \{(x, y) \in B : y \neq b_i\} \cup \{(a_i, b_i)\}$. If so, then $c(B'_i) \geq c(B)$ and thus B'_i would be an optimal branch with one common edge (a, b) more in B_0. This is a contradiction.

B'_1 is no branching. So there is a cycle in B'_i, which contains (a_i, b_i). So a b_i-a_i-path in B'_i is exists in B. □