Modelling and Analysis of Networks

Peter Bachhiesl
FH-TECHNIKUM Kärnten
Studiengang Telematik/Netzwerktechnik
Primoschgasse 8, 9020 Klagenfurt
p.bachhiesl@fh-kaernten.ac.at
+43 (463) 90500 3112

March 16, 2010

Contents

General remarks

Part I

Graphs and networks

Chapter 1

Graphs and networks - basic definitions

1.1 Examples of application of graphs and networks

bla, bla, ...

1.2 Formalism of graphs and networks

In order to deal with graphs and networks we have to formalize them properly. For graphs (and networks) two classes can be distinguished: directed and undirected ones.

Definition 1.2.1 (directed graph)
A directed graph is a quadruple $G = (V, E, \alpha, \omega)$ with the following properties:

1. V is a non-empty set of nodes.

2. E is a non-emty set of arcs.

3. $V \cap E = 0$

4. $\alpha : E \to V$ and $\omega : E \to V$ whereby $\alpha(e)$ and $\omega(e)$ are the tail and the head of the arc $e \in E$, respectively. We say that the arc $e \in E$ is directed from its tail $\alpha(e)$ to its head $\omega(e)$ and write $e = \langle \alpha(e), \omega(e) \rangle$.

5. For a graph G, $V(G)$ and $E(G)$ are the set of nodes and arcs, respectively.

Definition 1.2.2 (special arcs)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then we define:

- An arc $e \in E$ is a self loop, if $\alpha(e) = \omega(e)$. We denote G as loopless, if it does not contain any self-loop.

- Two arcs e and f are called multi-arcs or parallel arcs, if $\alpha(e) = \alpha(f)$ and $\omega(e) = \omega(f)$. e and f are denoted as inverse arcs or anti-parallel arcs, if $\alpha(e) = \omega(f)$ and $\omega(e) = \alpha(f)$.

- A graph is called simple, if it does not contain any self-loop and any pair of parallel arcs. Thus each arc $e \in E$ is uniquely identified by the pair $(\alpha(e), \omega(e))$. Thus we shortly say, that $E \subseteq V \times V$.

Definition 1.2.3 (adjacency and degrees)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then we define:

- An arc $e \in E$ and a node $v \in V$ are called to be incident, if v is the tail or the head of e.
- Two arcs $f, e \in E$ are called to be incident, if there exists a node $v \in V$ which is incident to both arcs.
- Two nodes $u, v \in V$ are called to be neighbours or to be adjacent, if there exists an arc $e \in E$ which is incident to both nodes.
- For a node $v \in V$ we define:
  - $\delta_G^+(v) := \{e \in E : \alpha(e) = v\}$ (set of arcs starting in v)
  - $\delta_G^-(v) := \{e \in E : \omega(e) = v\}$ (set of arcs discharging into v)
  - $N_G^+(v) := \left\{\omega(e) : e \in \delta_G^+(v)\right\}$ (set of successors of v)
  - $N_G^-(v) := \left\{\alpha(e) : e \in \delta_G^-(v)\right\}$ (set of predecessors of v)
  - $g_G^+(v) := \left|\delta_G^+(v)\right|$ (out-degree of v)
  - $g_G^-(v) := \left|\delta_G^-(v)\right|$ (in-degree of v)
  - $g_G(v) := g_G^+(v) + g_G^-(v)$ (degree of v)
- The quantity $\Delta(G) := \max\{g_G(v) : v \in V\}$ is denotes as the maximum-degree of the graph G.

Definition 1.2.4 (subgraph)
A graph $G' = (V', E', \alpha', \omega')$ is called a subgraph of $G = (V, E, \alpha, \omega)$ and we write $G' \subseteq G$, if the following two conditions hold:

1. $V' \subseteq V$ and $E' \subseteq E$.
2. The restrictions of α and ω to E' are valid; i.e. $\alpha|_{E'} = \alpha'$ and $\omega|_{E'} = \omega'$.

Vice versa we may call G the supergraph of G'. Replacing the $\subseteq$-relations by $\subset$-relations we get strict subgaphs.

Definition 1.2.5 (induced subgraph)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then we define:

- For the node set $V' \subseteq V$ the induced subgraph $G[V']$ is defined by: $V(G[V']) := V'$ and $E(G[V']) := \{e \in E : \alpha(e) \in V' \wedge \omega(e) \in V'\}$.
- For the arc set $E' \subseteq E$ the induced subgraph $G[E']$ is defined as $G[E'] := (V, E', \alpha|_{E'}, \omega|_{E'})$.
- For $v \in V$ and $e \in E$ we shortly write $G - v$ for $G[V \setminus \{v\}]$ and $G - e$ for $G[E \setminus \{e\}]$, respectively.

Definition 1.2.6 (undirected graph)
An undirected graph is a triple $G = (V, E, \gamma)$ with a non-empty set V of nodes, a non-empty set E of edges, $V \cap E = 0$, and a mapping $\gamma : E \to \{X \subseteq V : 1 \leq |X| \leq 2\}$, whereby $\gamma(e) = \{u, v\}$ declares the end nodes of e; in future we will also write $e = (u, v) = (v, u)$. We call G a complete graph, if $(u, v) \in E$ for all $(u, v) \in V \times V$, $u \neq v$.

The definition of incidence, adjacency, degree, sub- and supergraphs etc. are analogously to the directed case. If $|\gamma(e)| = 1$ then e is called a self-loop. Parallel edges e and f are obvously defined by equal mappings $\gamma(e) = \gamma(f)$. Moreover it is obvious, that:

$$\delta(v) := \{e \in E : v \in \gamma(e)\} \quad \text{(set of edges incident to } v\text{)}$$

$$N_G(v) := \{w \in V : \gamma(e) = \{v, w\} \text{ for some } e \in E\} \quad \text{(set of neighbours of } v\text{)}$$

$$g_G(v) := |\{e \in \delta(v) : |\gamma(e)| = 2\}| + 2\,|\{e \in \delta(v) : |\gamma(e)| = 1\}| \quad \text{(degree of node } v\text{)}$$

The (maximum) degree $\Delta(G)$ of G as well as the induction of subgraphs is declared in an analogous way as above .

Definition 1.2.7 (inverse graph, symmetric hull)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. For an arc $e = \langle \alpha(e), \omega(e) \rangle$ the arc

$$e^{-1} := \left\langle \alpha\left(e^{-1}\right) := \omega(e), \omega\left(e^{-1}\right) := \alpha(e) \right\rangle$$

is called the inverse arc of e. Thus, with $E^{-1} := \left\{e^{-1} : e \in E\right\}$ the graphs $G^{-1} := \left(V, E^{-1}, \alpha, \omega\right)$ and $G^{sym} := \left(V, E + E^{-1}, \alpha, \omega\right)$ are called the inverse graph and the symmetric hull of G, respectively. Deleting all paralell arcs and self loops in G^{sym} yields a simple symmetric hull. We call G symmetric if $G = G^{sym}$.

Definition 1.2.8 (assigned graph, orientation)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then the undirected graph $H = (V, F, \gamma)$ with

$$F := E \text{ wherby } \gamma(e) := \{\alpha(e), \omega(e)\} \text{ for all } e \in E$$

is called the undirected graph assigned to G. Vice versa we call G an orientation of H.

Remark 1.2.1
The simple symmetric hull of a directed graph may be seen equivalent to the assigned undirected graph.

1.3 Data-structures for graphs and networks

Let us consider a directed graph $G = (V, E, \alpha, \omega)$ and an undirected one $H = (V, F, \gamma)$. Furthermore let $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ respectively $F = \{f_1, \ldots, f_m\}$. The simplest way to store a graph is to use the following matrix:

$$A(G) \in Mat(n \times n) \text{ with } a_{ij} := |\{e \in E : \alpha(e) = v_i \wedge \omega(e) = v_j\}|$$

respectively

$$A(H) \in Mat(n \times n) \text{ with } a_{ij} := |\{f \in F : \gamma(f) = \{v_i, v_j\}\}|$$

We call $A(G)$ and $A(H)$ the adjacency matrix of the according graph. Obviously, in the undirected case, $A(H)$ is a symmetric matrix which allows to omit the lower triangular part if we have to store $A(H)$.

Up to now we have exclusively considered graphs; there exists no exact definition of the term network. Througout this lecture we mean by a network an extension of a graph $G = (V, E)$ by one ore more additional mappings $c : E \to \mathbb{R}$ or $\xi : V \to \mathbb{R}$. These additional mappings are useful in order to define costs or capacities on arcs or edges; often nodes are weighted by profit or return on investements. Thus we define for further notation that $e \in E : c(e) =: c\langle u, v\rangle$ for $e = \langle u, v\rangle$ or $c(e) =: c(u, v)$ for $e = (u, v)$.

A second option for storing graphs is the incidence matrix. We assume that G and H do not contain any self-loop. Then

$$I(G) \in Mat(n \times m) \text{ with } i_{kl} := \begin{cases} 1 & \text{if } \alpha(e_l) = v_k \\ -1 & \text{if } \omega(e_l) = v_k \\ 0 & \text{otherwise} \end{cases}$$

respectively

$$I(H) \in Mat(n \times m) \text{ with } i_{kl} := \begin{cases} 1 & \text{if } v_k \in \gamma(f_l) \\ 0 & \text{otherwise} \end{cases}$$

The third, and most common, alternative for dealing with graphs are incidence lists. $Adj(G)$ is a structured array $Mat(1 \times n)$, whereby for a node $v \in V$ the token $Adj(v)$ contains a list of pointers each of which pointing to a node $w \in N^+(v)$ or $w \in N(v)$, respectively. If v contains parallel arcs or edges to w, then the according pointer will be multiple listed.

Laboratory 1.3.1
We consider a bidirectional ad-hoc network $G = (V, E; d)$ in the two dimensional space $\mathbb{R}^2$. The mappings $P : V \to \mathbb{R}^2$ and $d : V \times V \to \mathbb{R}$ measure the position of each node and the euclidean distance between two nodes, respectively. We define that two nodes are connected by an edge e (are able to communicate) if they are sufficiently close; i.e. $e = (u, v) \in E :\Leftrightarrow d(u, v) \leq \Delta_{tol}$.

Remark 1.3.1
In the directory `[Course]\Routines\Tools` you find some useful tools for the construction and visualization of graphs.

Laboratory 1.3.2

1. We initialize our network with a position mapping P_0. P_k are a randomized position mappings for $k \geq 0$ - let k identify the k-th discrete time-state of the network.
2. Simulate (and visualize) the situation of incidence of the network for several successive time-states of G.
3. Due to security we require that each node of the network (e.g. persons of an emergency service) has to be able to communicate with at least two other nodes. Thus, if a node does not fulfil this requirement in the k-th time states, correct P_k accordingly.

1.4 Complexity of algorithms

In order to measure the effort of algorithms and data structures on graphs we define the term complexity. Let $M := \{f \; ; \; f : \mathbb{N} \to \mathbb{R}\}$ be an set of real-valued functions. Then we define the following complexity measures for a function $f \in M$:

$$f \in O(g) :\Leftrightarrow \exists g \in M, c \in R, n_0 \in \mathbb{N} : f(n) \leq cg(n) \;\; \forall n \geq n_0 \tag{1.1}$$

$$f \in \Omega(g) :\Leftrightarrow \exists g \in M, c \in R, n_0 \in \mathbb{N} : f(n) \geq cg(n) \;\; \forall n \geq n_0 \tag{1.2}$$

$$f \in \Theta(g) :\Leftrightarrow f \in O(g) \text{ and } f \in \Omega(g) \tag{1.3}$$

We call $O(g)$, $\Omega(g)$ and $\Theta(g)$ the most rate of growth, the least rate of growth and the rate of growth, respectively. By (most, least) rate of growth we just mean growth of time consumption - we call the amount of time consumption as complexity. We assume that space complexity will be similar to time complexity, if not mentioned separately. If g is polynomial in n, we call $f \in O(g)$ to be of polynomial complexity.

Example 1.4.1
We consider the undirected graph $G = (V, E)$. We assume that G is simple - it does not contain parallel edges or self-loops. Thus if $m := |E|$ and $n := |V|$ we observe (e.g. by considering the according adjacency matrix) that the inequality

$$m \leq \frac{n(n-1)}{2} \leq n^2$$

holds. Thus we may say that $m \in O(n^2)$; the number of edges is of quadratic complexity with respect to the number of nodes. This estimate of complexity is also valid for simple directed graphs. Generally we denote a graph as sparse, if $m \ll n^2$; e.g. $m \in O(n)$; otherwise we call it a dens one.

Example 1.4.2
Let us examine the complexity of the three options for storing the graphs $G = (V, E)$. We assume that $m := |E|$ and $n := |V|$.

option	storage	$\langle v, w \rangle \in E$ or $(v, w) \in E$
$A(G)$	$O(n^2)$	$O(1)$
$I(G)$	$O(nm)$	$O(m)$
$Adj(G)$	$O(n+m)$	$O(g^+(v))$ or $O(g(v))$

Thus we see, that the adjacency list $Adj(G)$ is favourable in case of storing sparse graphs. Nevertheless, for finding existing arcs or edges we are intended to use adjacency matrices.

Considering a problem which has to be solved, three major classes of complexity are distinguished in the homonymous theory. In order to get a better association to the following terms, we will motivate them with the following examples:

Example 1.4.3 (MST - minimum spanning tree problem)
We consider the undirected graph $G = (V, E)$. The mapping $c : E \to \mathbb{R}_+$ measures the costs for connecting the nodes $u, v \in V$ by $c(u, v)$. Let $E' \subseteq E$ be a subset of edges such that two arbitrarly chosen nodes $u, v \in V$ are connected in the induced subgraph $G[E']$. Roughly spoken, we mean by connected the possiblility to walk from u along some of the edges in E' to v. The costs for this subgraph are given by

$$c(G[E']) = c(E') = \sum_{e \in E'} c(e)$$

The MST problem is to find a subset E' with minimal costs.

Example 1.4.4 (TSP - traveling salesman problem)
We consider the undirected network $G = (V, E)$. G is a complete graph and the mapping $c : E \to \mathbb{R}_+$ measures the distance between two cities $u, v \in V$ by $c(u, v)$. A cyclic permutation $(1 = \pi_1, \pi_2, \dots, \pi_n)$ of the set $V = \{v_1, \dots v_n\}$ defines a tour

$$1 = \pi_1 \to \pi_2 \to \dots \to \pi_n \to \pi_{n+1} = 1$$

through the graph on which each city is visited exactly one times; without loss of generality we start at city 1. The total distance of the tour is given by

$$c(\pi) = \sum_{i=1}^{n} c(\pi_i, \pi_{i+1})$$

We call a thour optimal if its total distance is minimal among all other possible tours. The TSP problem is the problem to find this optimal tour. There are existing $(n-1)!$ permutations; since the total costs of a tour are the same as that of the tour in reverse direction, we would have to investigate $(n-1)!/2$ tours.

Originally the three problem classes are defined on the basis of decesion problems which can be answered by 'yes' or 'no'. Notice, that each optimization problem can be formulated as an equivalent decision problem. The problem of finding a x within a set F of feasible solutions in order to minimize a cost function $f(x)$ is equivalent to the problem to decide proovable-correctly whether a $x \in F$ exists such that $f(x) \le M$ for an arbitrary chosen bound $M \in \mathbb{R}$:

$$\min_{x \in M} f(x) \quad \Leftrightarrow \quad \exists \text{ or } \nexists\ x \in F : f(x) \le M \ \text{ for all } M \in \mathbb{R}$$

With respect to such a decision problem we define:

1. The class P (polynomial) contains all decision problems for which a decision process yields the provable correct decision with polynomial complexity. Such problems are also called 'easy'.

2. The class NP (non-deterministic polynomial) encloses all decision problems for which a guessed 'yes'-descision can be verified with polynomial complexity. Notice that this is much more weaker than a proovable certificate which allways yields the correct decision. Obviously the inclusion $P \subseteq NP$ holds.

3. Within the class NP we consider the the set of NP-complete problems. A problem is denoted as NP-complete if it is in NP and if the polynomial solvability or this problem would imply that all problems in NP are polynomially solvable. In other words: each problem in NP can be transformed to the considered NP-complete problem with polynomial effort. Thus if we could find a polynomial algorithm for a NP-complete problem we would prove that $NP = P$. This represents a still open problem.

Example 1.4.5 (MST - an easy problem)
The MST problem is easy; we formulate it as a decision problem: does or does not a subset $E' \subseteq E$ exist, such that $c(E') \le M$ for an arbitrary bound $M \in \mathbb{R}_+$? We will see in the later sections, that a polynomial algorithm which solves the MST problem to the optimal can be found - thus we are also in the position to decide proovable-correctly on the basis of this algorithm with polynomial complexity.

Example 1.4.6 (TSP is in NP)
Also the TSP problem is an optimization problem. Thus, in analogy to the foregoing example, we transform it to a decision problem: does a feasible permutation π exist such that $c(\pi) \leq M$ for an arbitrary bound $M \in \mathbb{R}_+$? If a solution π is presented we can verify a 'yes'- answer easily by checking if π is feasible permutation, calculating $c(\pi)$ and comparing it with M. But on the other side we do not know any polynomial algorithm for solving the TSP problem to optimal and therefore for cheking a negative answer; namly: no feasible permutation exists such that $c(\pi) \leq M$ for an arbitrary bound $M \in \mathbb{R}_+$. The TSP problem is also NP-complete. Each problem of the class NP can be transformed to the TSP problem.

Chapter 2

Basic problems and algorithms

2.1 Pathes and cycles

Definition 2.1.1 (Path)
Let $G = (V, E, \alpha, \omega)$ and $G = (V, E, \gamma)$ be a directed- and an undirected graph, respectively.

- In the directed case, a path P is a finite series $(v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$ whereby $v_i \in V$ and $e_i \in E$ such that $\alpha(e_i) = v_{i-1}$ and $\omega(e_i) = v_i$ for $i = 1, \ldots, k$. If we write a series exclusively consiting of nodes, i.e. $(v_0, v_1, v_2, \ldots, v_k)$, then we imply that one unique identified arc from v_i to v_{i+1} exists; this is feasible in the case of not existing parallel arcs.

- Analogous, we define $P = (v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$ with $v_i \in V$, $e_i \in E$ and $\gamma(e_i) = \{v_{i-1}, v_i\}$ for $i = 1, \ldots, k$ as a path in the undirected graph.

- In both cases we call $\alpha(P) := v_0$ and $\omega(P) := v_k$ the start- and the end-node of P, respectively. Thus we call P also a v_0, v_k-path in G.

- If $\alpha(P) = \omega(P)$ and $k \geq 1$ then we call P a cycle in G.

- With $V(P)$ and $E(P)$ we mean the set of nodes and arcs or edges which are within the path P, respectively.

- We call P a simple path if $e_i \neq e_j$ for $i \neq j$.

- We call P an elementary path if $v_i \neq v_j$ for $i \neq j$, with exception of $v_i = \alpha(P)$ and $v_j = \omega(P)$.

- We call P degenerated if $k = 0$.

- The length of a path P is defined by the number of arcs or edges which are within P.

- A Hamiltonian path is a path P from $s \in V$ to $t \in V$ such that P contains each node in V exactly one times. For $s = t$ we call P as a Hamiltonian cycle. For example: in problem 1.4.4, page 7, we exemplarily search for a Hamiltonian cycle with minimum costs. In general, finding a Hamiltonian path or cycle, respectively to decide wheter a graph does contain one of them are problems of the complexity class NP-complete ($\rightarrow$[5], theorem 3.35 on page 50).

Lemma 2.1.1
Let $G = (V, E, \alpha, \omega)$ be a directed graph with $g^+(v) \geq 1$ for all $v \in V$. Then G contains an elementary cycle. If G is a simple graph and if $g^+(v) \geq m > 0$ for all $v \in V$ then G contains a elementary cycle of length at least $m + 1$.

Proof 2.1.1
Since $g^+(v) \geq 1$ for all $v \in V$ there exist elementary pathes in G; e.g. $(\alpha(e), e, \omega(e))$ is such a path of length one. Let $P = (v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$ be the longest elementary path in G; notice the length is restricted by $n := |V|$. Thus, since $g^+(v_k) \geq 1$ there must exist an edge e with $\alpha(e) = v_k$ and moreover $\omega(e) \in P$. The latter statements holds since, if not, the path $(v_0, \ldots . e_k, v_k, e, \omega(e))$ would be an elementary path of length grater then P - this case is excluded by assumption. Summarizing, if $\omega(e) = v_{i-1}$ then $(v_{i-1}, e_i, v_i, \ldots, e_k, v_k, e, v_{i-1})$ forms an elementary cycle. If $g^+(v_k) \geq m > 0$ then there are exist at least m arcs with end-nodes in P (for the same reason as before). Since G is simple these end-nodes have to be pairwise different. Let us take the arc $e \in \delta^+(v_k)$ with $\omega(e) = v_{i-1}$ such that i is minimal. Then obviously $k - i \geq m - 1$ holds. Thus the elementary cycle $(v_{i-1}, e_i, v_i, \ldots, e_k, v_k, e, v_{i-1})$ has length $\geq m+1$.
□

In analogy one may show the following two statements:

Lemma 2.1.2
Let $G = (V, E, \alpha, \omega)$ be a directed graph with $g^-(v) \geq 1$ for all $v \in V$. Then G contains an elementary cycle. If G is a simple graph and if $g^-(v) \geq m > 0$ for all $v \in V$ then G contains a elementary cycle of length at least $m + 1$.

Lemma 2.1.3
Let $G = (V, E, \gamma)$ be an undirected graph with $g(v) \geq 2$ for all $v \in V$. Then G contains an elementary cycle. If G is a simple graph and if $g(v) \geq m \geq 2$ for all $v \in V$ then G contains a elementary cycle of length at least $m + 1$.

Definition 2.1.2 (topological sorting)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then a topological sorting is a bijective mapping $\sigma : V \to \{1, 2, \ldots, n\}$, whereby $|V| =: n$ such that $\sigma(\alpha(e)) < \sigma(\omega(e))$ for all arcs $e \in E$.

Theorem 2.1.1
A directed graph $G = (V, E, \alpha, \omega)$ does not contain any cycle if and only if a topological sorting exists for G.

Proof 2.1.2
"$\Rightarrow$" Let G be without any cycle. We show this direction by induction. For the subgraph $G[\{v\}]$ induced by any $v \in V$ this direction is trivialy fulfilled. Let us now assume that $G[\{v_1, v_2, \ldots, v_k\}]$ is the subgraph subgraph induced by the node set $\{v_1, v_2, \ldots, v_k\}$. Since $G[\{v_1, v_2, \ldots, v_k\}]$ does not contain any cycle, there must exist a $v_i \in \{v_1, v_2, \ldots, v_k\}$ with $g^-(v_i) = 0$ - see lemma 2.1.2. Thus, if σ' is a topological sorting for the graph $G[\{v_1, v_2, \ldots, v_k\} \setminus \{v_i\}]$ then also $\sigma : \{v_1, v_2, \ldots, v_k\} \to \{1, 2, \ldots, k\}$ with $\sigma(v_i) = 1$ and $\sigma(v_j) = \sigma'(v_j) + 1$ for $j \neq i$ is also a topological sorting for $G[\{v_1, v_2, \ldots, v_k\}]$.
"$\Leftarrow$" We show this direction by contradiction. Let us assume that $(v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k = v_0)$

is a cycle. Due to the existence of a topological sorting the relation

$$\sigma(v_0) < \sigma(v_1) < \sigma(v_{k-1}) < \sigma(v_k) = \sigma(v_0)$$

must hold, which is obviously not possible. $\square$

This proof allows the construction of an algorithm for the topological sorting of a graph; or shows that this is not possible.

$$\text{Step } 1: G_1 := G, v_1 \in V(G_1) : g^-(v_1) = 0 \to \sigma(v_1) = 1$$

$$\text{Step } 2: G_2 := G_1 - v_1, v_2 \in V(G_2) : g^-(v_2) = 0 \to \sigma(v_2) = 2$$

$$\vdots$$

$$\text{Step } k+1: G_{k+1} := G_k - v_k, v_{k+1} \in V(G_{k+1}) : g^-(v_{k+1}) = 0 \to \sigma(v_{k+1}) = k+1$$

Using an adjaceny list for storing the graph G the complexity for finding a node with in-degree zero is $O(n+m)$ for $n := |V|$ and $m := |E|$. Thus the proposed algorithm will be of complexity $O(n(n+m))$. A more efficient implementation is:

```
Algorithm TopSort

  Specification: Topological sorting of directed graph.
  Input: Adjacency $Adj(G)$ list of $G = (V, E, \alpha, \omega)$.
  Output: Topological sorting $\sigma$.

 1. Initialize:

    (a) Compute $g^-(v)$ for all $v \in V$.
    (b) Let $L = \{v \in V : g^-(v) = 0\}$.

 2. For $i = 1, \ldots, n$ do:

    (a) If $L \neq 0$ then do:
          i. Take an $v \in L$ and set $\sigma(v) = i$.
         ii. Update $L = L - \{v\}$.
        iii. For all $e \in \delta^+(v)$ do:
               $g^-(\omega(e)) := g^-(\omega(e)) - 1$.
               If $g^-(\omega(e)) = 0$ then add $\omega(e)$ to $L$.
    (b) Otherwise terminate algorithm, since $G$ does not has any topological
        sorting (and does therfore contain a cycle).

End of Algorithm TopSort
```

Theorem 2.1.2
The algorithm TopSort computes a topological sorting of the graph G with a worst case complexity of $O(n+m)$ or shows that such a sorting does not exist.

Proof 2.1.3
The correctness follows from the proof of theorem 2.1.1. The step 1.a is $O(n+m)$ under application of $Adj(G)$. Step 2 has to be performed n times whereby decreasing the in-degrees and writing to L is performed under constant time. □

Laboratory 2.1.1
Consider the tasks of a project (production process, bussiness process etc.). If the beginning of several tasks requiere the finishing of one ore more forgoing taks, the project obviously shuold not contain any cycle and the tasks have to be ordered in a meaningful (i.e. topological sorted) chain of activities. Create your own project or use a project generator from the internet (e.g. http://tracer.lcc.uma.es/problems/psp/generator.html) and check the feasibility of the designed project; propose a meaningful chain of activities.

2.2 Connectivity of graphs

Definition 2.2.1 (reachability)
We consider a graph $G = (V, E)$. In both cases, for directed and undirected graphs the node $t \in V$ is called reachable from the node $s \in V$ if there exists a s,t-path in G. With $R(s)$ we define the set of all nodes which are reachable from s.

```
Algorithm Reachable

  Specification:  Computes the set $R(s)$ for a $s \in V$.
  Input:  Adjacency $Adj(G)$ list of $G = (V, E)$, node $s \in V$.
  Output:  $R(s)$.

  1. Initialize:
      (a) Mark $v$ with $m(v) = false$ for all $v \in V \backslash \{s\}$.
      (b) Mark $s$ with $m(s) = true$.
      (c) Initialize the queue $Q = [s]$
  2. While not is empty $Q$ do:
      (a) $v = Q(1)$.
      (b) $Q := Q(2 : end)$.
      (c) For all $\{w \in Adj(v) : m(w) = false\}$ do:
           i. Update $m(w) = true$.
          ii. Update $Q := [Q, w]$.
  3. $R(s) := \{v \in V : m(v) = true\}$.

End of Algorithm Reachable
```

Theorem 2.2.1
The algorithm `Reachable` computes the set $R(s)$ with complexity $O(n+m)$ for $n := |V|$ and $m := |E|$.

Proof 2.2.1
Obviously each node is at most one times in Q and each arc will be considered at most one times; in the undirected case each edge will be considered at most two times. This proofs the stated complexity. The correctness may be shown by induction. Since a node w will only be marked with $m(w) = true$ if there exists an arc e with $\alpha(e) = v$ and $\omega(e) = w$ (edge e with $\gamma(e) = \{v, w\}$) and v has allready been marked with *true*, it follows by induction with respect to the number of marking steps, that only the nodes in $R(s)$ will be marked with *true*. Thus it remains to show that all nodes in $R(s)$ will be correctly marked. Let $w \in R(s)$ and k be the length of the shortest s, w-path. For $k = 0$ and therefore $w = s$ the algorithm marks correctly. Now let $k \geq 1$ and $P = (s = v_0, e_1, v_1, \ldots, e_{k-1}, v_{k-1} = v, e_k, v_k = w)$ be the according path of length k. Thus the length of the shortest s, v-path is at most $k-1$. Due to assumption v will have been marked and added to Q correctly. So if v will be removed from Q in step 2.b then also the node w will be marked and added to Q correctly. $\square$

Remark 2.2.1
We will meet this procedure later and will identify it as breadth first search (BFS).

Definition 2.2.2
Let $G(V, E, \alpha, \omega)$ be a directed graph. Then the nodes u and v are called strong-connected if $u \in R(v)$ and $v \in R(u)$. A maximum set of nodes in V which are strong-connected among each other are called a strong connectivity component within G. We call a graph strong connected, if it contains exaktly one strong connectivity component..

Definition 2.2.3
A directed graph $G(V, E, \alpha, \omega)$ is called weak connected, if its simple symmetric hull G^{sym} is strong connected. A weak connectrivity component of a graph G is a maximum sized weak connected subgraph of G.

Remark 2.2.2
Per definition each node is (strong or weak) connected to itself.

Remark 2.2.3
For undirected graphs $G = (V, E, \gamma)$ we do not have to distinguish between strong or week connectivity. We can see an undirected graph as a directed one, with according pairwise inverse arcs - thus this directed graph is its own simple symmetric hull. If this hull is strong connected we say that G is connected; otherwise G is not connected and consits of connectivity components.

Remark 2.2.4
We can alternatively formulate a criterion for the weak connectivity of a directed graph G: if the assigned undirected graph is connected, then G is weak connected.

The following algorithm computes the connectivity components in undirected graphs with linear complexity.

```
Algorithm NetwConn

  Specification: Computes connected components of an undirected graph.
  Input: Adjacency $Adj(G)$ list of $G = (V, E)$.
  Output: mapping $C : V \to \{1, 2, \ldots, m\}$, whereby $G$ contains exaktly
          $m$ connected components;$C(v)$ is the number (index) of
          the component containing the node $v$.

  1. Initialize:
     (a) Assign component index $C(v) := 0$ for all $v \in V$.
     (b) Initialize component counter $i := 0$.
  2. While the set $V' := \{v \in V : C(v) = 0\} \neq 0$ do
     (a) Select an arbitrary node $v \in V'$.
     (b) Compute $R(v) =$ Reachable($Adj(G)$,$v$).
     (c) Update $i := i + 1$.
     (d) Assign $C(u) := i$ for all $u \in R(v)$.

End of Algorithm NetwConn
```

Remark 2.2.5
Checking a directed graph G for weak connectivity or to find the weak connectivity components is easy: consider the according assigned undirected graph G' and perform algorithm `NetwConn` on G'.

Remark 2.2.6
Finding the strong connectivity components of directed graphs is more complicated. We will discuss this topic in the section 2.5, page 20.

Laboratory 2.2.1
Let us reconsider our fictive ad-hoc network of laboratory 1.3.1, page 5. We study two new cases of security regulations:

1. The number of connected components is restricted at each time step.
2. The network has to be connected in a redundant mode: an undirected graph $G = (V, E)$ is called two edge- and two node connected, if $G - \{e\}$ and $G - \{v\}$ remains connected for all $e \in E$ and for all $v \in V$, respectively.

Thus check both cases for all time states of the network and correct P_k accordingly in case of violance.

Definition 2.2.4 (Cut)
For a graph $G = (V, E)$ the partition $A, B \subset V$, $A \cap B = 0$, $A \cup B = V$ is called a cut of G - we write (A, B). For $S \subset V$ the edge sets

$$\delta^+(S) := \{e \in E : \alpha(e) \in S \text{ and } \omega(e) \notin S\}$$

$$\delta^{-}(S) := \{e \in E : \alpha(e) \notin S \text{ and } \omega(e) \in S\}$$

respectively

$$\delta(S) := \{e \in E : \gamma(e) = \{u, v\} \text{ with } u \in S \text{ and } v \notin S\}$$

are called the cut arcs or edges with respect to the cut $(S, V \backslash S)$.

Theorem 2.2.2
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then G is strong connected if and only if $\delta^{+}(S) \neq 0$ for all $S \in p(V)$ with $S \neq 0$ and $S \neq V$.

Proof 2.2.2
"$\Rightarrow$" .Obviously for an $S \subset V$, $S \neq 0$, there exist two nodes $v \in S$ and $w \in V \backslash S$. Since $w \in R(v)$ it follows that at least one edge has to be in $\delta^{+}(S)$.
"$\Leftarrow$". We show this case by contradiction. Assuming that G is not strong connected there exists at least one pair of edges v, w such that $w \notin R(v)$. So define $S := R(v)$. Then $(S, V \backslash S)$ is a cut with $\delta^{+}(S) = 0$ which is a contradiction to the precondition. $\square$

In analogy the following theorems can be shown:

Theorem 2.2.3
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then G is weak connected if and only if $\delta^{+}(S) \neq 0$ or $\delta^{-}(S) \neq 0$ for all $S \in p(V)$ with $S \neq 0$ and $S \neq V$.

Theorem 2.2.4
Let $G = (V, E, \gamma)$ be an undirected graph. Then G is connected if and only if $\delta(S) \neq 0$ for all $S \in p(V)$ with $S \neq 0$ and $S \neq V$.

2.3 Euler pathes and Euler cycles

Definition 2.3.1 (Euler path, cycle and graph)
Let $G = (V, E, \alpha, \omega)$ and $G = (V, E, \gamma)$ be a directed- and an undirected graph, respectively. Then a path P in G is called an Euler path if P runs through each edge of E exactly one times. If P is a cycle we call it an Euler cycle. If G does contain an Euler cycle we call it an Euler graph.

Let us consider the situation in directed graphs at first.

Theorem 2.3.1 (Euler I)
Let $G = (V, E, \alpha, \omega)$ be a directed and weak connected graph. Then G is an Euler graph if and only if $g^{+}(v) = g^{-}(v)$ for all $v \in V$.

Proof 2.3.1
"$\Rightarrow$". Let $E \neq \{\}$ and C be an Euler cycle. Since G is weak connected, each $v \in V$ meets $g(v) > 0$; therfore v must be a node in the cycle C. Obviously, if we run through C we leave v as often as we visit it - the in- and outdegree of v is equal in C. Due to definition C contains all edges of G; thus this direction is shwon.
"$\Leftarrow$". We give a constructive proof for this direction - for a schematic view see figure 2.1. Due

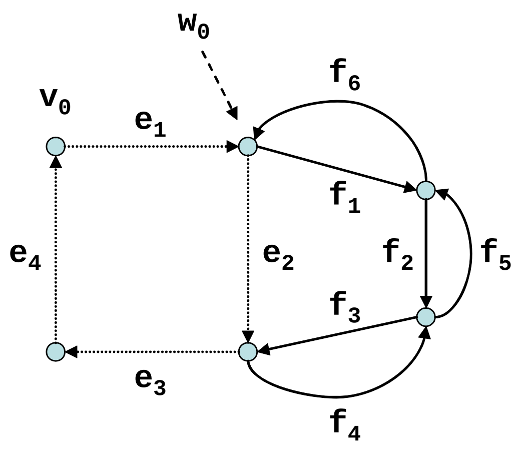


Figure 2.1: Schematic view for the constructive proof of theorem 2.3.1.

to weak connectivity and $g^+(v) = g^-(v)$ for each $v \in V$ there exists a least one arc leaving an arbitrary $v_0 \in V$. Initialize the degenerated path $C = (v_0)$ and iterate:

$$v_0 \in V : e_1 \in \delta^+(v_0) \text{ with } e_1 \text{ is unmarked} \rightarrow \text{mark } e_1,\ v_1 = \omega(e_1) \text{ and } C = C + (e_1, v_1)$$

$$v_1 \in V : e_2 \in \delta^+(v_1) \text{ with } e_2 \text{ is unmarked} \rightarrow \text{mark } e_2,\ v_2 = \omega(e_2) \text{ and } C = C + (e_2, v_2)$$

$$\vdots$$

$$v_{k-1} \in V : e_k \in \delta^+(v_{k-1}) \text{ with } e_k \text{ is unmarked} \rightarrow \text{mark } e_k,\ v_k = \omega(e_k) \text{ and } C = C + (e_k, v_k)$$

until in a node v_k no more unmarked outgoing arcs exist. This exactly happens if $v_0 = v_k$. Otherwise if $v_k \neq v_0$ then we have found that more arcs would end to v_k as leaving v_k which is a contradiction to the initial situation $g^+(v) = g^-(v)$ for all $v \in V$. If $E(C) = E$ then C is an Euler cycle and we have been done. Otherwise we construct the graph G' with $E(G') = E - E(C)$. Notice that $g^+(v) = g^-(v)$ remains valid in G'. Thus since G has been weakly connected there must exist a node $w_0 = v_i \in V(C)$ and an (unmarked) edge $f_1 \in E(G')$ with $\alpha(f) = w_0$. So we initialize a new cycle $C' = (w_0)$ and iterate onece again

$$w_0 : f_1 \in \delta^+(w_0) \text{ with } f_1 \text{ is unmarked} \rightarrow \text{mark } f_1,\ w_1 = \omega(f_1) \text{ and } C' = C' + (f_1, w_1)$$

$$\vdots$$

$$v_{l-1} : f_l \in \delta^+(w_{l-1}) \text{ with } f_l \text{ is unmarked} \rightarrow \text{mark } f_l,\ w_l = \omega(f_l) \text{ and } C = C + (f_l, w_l)$$

until $w_0 = w_l$. Since the two cycles C and C' have the node $w_0 = v_i$ in common we unite the two cycles as follows:

$$C + C' := (v_0, e_1, v_1, \ldots, e_i, v_i = w_0, f_1, w_1, \ldots, f_l, w_l = v_i, e_{i+1}, v_{i+1}, \ldots, e_k, v_k) \tag{2.1}$$

We say for this kind of union that we implement the cycle C' into the cycle C at position v_i. If $E(C + C') = E$ we have been done; otherwise we repeat the last step. $\square$

Remark 2.3.1
Since each directed Euler graph is also strong connected, the theorem Euler I yields criterions for checking the strong connectivity.

Based on the constructive proof of theorem Euler I we are able to give polynomial algorithms for computing Euler cycles, if they exist. In order to mark edges we introduce a pointer $cur(v)$ which points to the first unmarked edge in the adjacency list $Adj(v)$.

```
Algorithm Explore

  Specification:  Constructs a cycle starting at node $v \in V$.
  Input:  Adjacency list $Adj(G)$ of a weak connected directed graph.
          Pointers $cur(v)$ for all $v \in V$.
  Output:  A cycle $C$ and updated pointer positions $cur$.

  1. Set $e = cur(v)$ and $u = \omega(e)$.
  2. Initialize $C = (v, e, u)$ and update $cur(v) = cur(v) + 1$.
  3. While $u \neq v$ do
     (a) Set $e = cur(u)$ and $w = \omega(e)$.
     (b) Update $C = C + (e, w)$ and $cur(u) = cur(u) + 1$.
     (c) Update $u = w$.

End of Algorithm Explore
```

```
Algorithm Euler

  Specification:  Constructs an Euler cycle.
  Input:  Adjacency list $Adj(G)$ of a weak connected directed graph.
  Output:  Euler cycle $C$.

  1. For all $v \in V$ do
     (a) Initialize $cur(v) = 1$.
  2. Initialize $C = (v_0)$ for an arbitraryly chosen $v_0 \in V$.
  3. Initialize $v = v_0$ and $run = true$.
  4. While $run = true$ do
     (a) While $cur(v) \leq g^+(v)$ do
         i. $(C', cur) =$ Explore($v, Adj(G), cur$).
         ii. Implement the cycle $C'$ into the cycle $C$ at position $v$ according to
             (2.1).
     (b) Update $v = v'$ whereby $v'$ is the node which follows $v$ in $C$.
```

```
    (c) If $v_0 = v$ then $run = false$.
End of Algorithm Euler
```

Theorem 2.3.2
The algorithm `Euler` computes an Euler cycle for an Euler graph with complexity $O(m+n)$, wherby $m = |E|$ and $n = |V|$.

Proof 2.3.2
The correctness of the algorithm follows from the proof of theorem 2.3.1. The complexity n follows from the initialization step 1; for complexity m we observe: if an edge has been explored and added to C it will not be considered once again since the according pointer *cur* has been updated. □

Theorem 2.3.3 (Euler II)
Let $G = (V, E, \alpha, \omega)$ be a directed and weak connected graph. Then G contains an Euler path, which is no Euler cycle, if and only if there exists two node $s, t \in V$, $s \neq t$, such that

$$g^+(s) = g^-(s) + 1$$

$$g^+(t) = g^-(t) - 1$$

$$g^+(v) = g^-(v) \quad \text{for all } v \in V \setminus \{s, t\}$$

Then the nodes s and t are the start- and the end node of the Euler path, respectively.

Proof 2.3.3
"$\Rightarrow$". Count the node degrees of $v \in V$ in the Euler path.
"$\Leftarrow$". Add a new arc $\langle t, s\rangle$ to the graph G, then the updated graph G' is a Euler graph containing the Euler cycle C, since $g^+(v) = g^-(v)$ for all nodes v in $V(G')$. Thus $C - \langle t, s\rangle$ has to be an Euler path in the original graph G. □

Let us now consider the undirected case.

Theorem 2.3.4 (Euler for undirected graphs)
An undirected and connected graph is an Euler graph if and only if all nodes have an even degree. The graph has an Euler path, which is no Euler cycle, if and only if exactly two nodes have an odd degree.

Proof 2.3.4
[5], p. 49. □

The algorithm `Euler` still remains (excepting small adaptions) the same for the undirected case.

Laboratory 2.3.1
Use the tool `GRand.m` in the directory `[Course]\Routines\Tools` in order to construct an undirected or directed Euler graph. Compute the according Euler cycle and visualize it. For reference you find the Routine Laboratory in the directory `[Course]\Routines\Euler`.

2.4 Trees in graphs - basic definitions

Definition 2.4.1 (Forest and tree)
An undirected graph $G = (V, E, \gamma)$ is called a forest if it does not contain any cycle. If G is connected, we call it a tree.

Definition 2.4.2 (Spanning tree and spanning forest)
Let $G = (V, E, \gamma)$ be an undirected graph. A subgraph $H = (V, E', \gamma)$ of G with $E' \subseteq E$ is a spanning tree if H is a tree. A subgraph $G' = (V, E', \gamma)$ of G with $E' \subseteq E$ is a spanning forest if each connectivity component of H is a spanning tree of a connectivity component of G.

Theorem 2.4.1 (Characterization of trees)
Let $G = (V, E, \gamma)$ be an undirected graph. Then the following statements are equivalent:

1. G is a tree.
2. G does not contain any cycle; but every graph $G' = (V, E', \gamma)$ with $E \subset E'$ does contain at least one elementary cycle.
3. For each pair $u, v \in V$ there exists exactly one elementary path P in G with $\alpha(P) = u$ and $\omega(P) = v$.
4. G is connected and $G - \{e\}$ is not connected for all $e \in E$.
5. G is connected and $|E| = |V| - 1$.
6. G does not contain any elementary cycle and $|E| = |V| - 1$.

Proof 2.4.1
See [5], pp. 100 ff. $\square$

Theorem 2.4.2 (Cayley)
Let $G = (V, E, \gamma)$ be a complete undirected graph with $|V| = n$. Then there exist n^{n-2} different spanning trees on G.

Proof 2.4.2
See [5], pp. 101 ff. $\square$

The definitions for the undirected case can be matched to the directed one in the following sens: We call a directed graph $G = (V, E, \alpha, \omega)$ a tree if the assigned undirected one ($\rightarrow$ Definition 1.2.8, page 4) is a tree. Thus directed trees are weakly connected graphs with $|E| = |V| - 1$. In this sens we can specialize:

Definition 2.4.3 (Rooted tree)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. The node s is called a root node of G, if $V = R(s)$; i.e. all other nodes can be reached from s. We call a subgraph $T \subseteq G$ a rooted tree if it is a tree and if it contains a root. All nodes v in a rooted tree which meet $g^{+}(v) = 0$ are called leaves of the rooted tree. A rooted spanning tree is a rooted tree $T \subseteq G$ which contains all nodes in V.

Definition 2.4.4 (Ancestor and descendant)
Let $G = (V, E, \alpha, \omega)$ be a rooted tree with root $s \in V$. Then for all $v \in V$ there exists an a unique path from s to v. Each node $u \in V$ on this path is called an ancestor of v, and v itself is a decendant of each u.

Theorem 2.4.3 (Characterization of rooted trees)
Let $G = (V, E, \alpha, \omega)$ be a directed graph. Then the following statements are equivalent:

1. G is a rooted tree with root $s \in V$.
2. G is a tree and $g^{-}(s) = 0$ and $g^{-}(v) = 1$ for all $v \in V - \{s\}$.
3. $g^{-}(s) = 0$ and $g^{-}(v) \leq 1$ for all $v \in V - \{s\}$ and $R(s) = V$.

Proof 2.4.3
See [5], pp. 136 ff. $\square$

Definition 2.4.5 (Rooted trees in undirected graphs)
Let $G = (V, E, \gamma)$ be an undirected graph. The node $s \in V$ is called a root node of the graph G if $V = R(s)$. A rooted tree with root node s is a tree $G = (V, E, \gamma)$ and $s \in V$ is explicitely specified as a root node. The definition of ancestors and descendants is the same as in the directed case.

2.5 Searching in graphs

We initialy consider a directed network $G = (V, E, \alpha, \omega)$. In order to get information about the structure or the contents of the network, we have to traverse it. What is the most efficient way to do this job?

In order to answer this question we introduce three colors: white, gray and black. We say that a node $v \in V$ is a white node if it has not been discovered; v is a gray node if it has been discovered; v is a black node if all nodes, which are reachable from v via an arc, have been discovered. Thus we can assign the following time markers to each node $v \in V$: $d(v)$ measures the time at which v became gray; thus let us denote d as discovery time. $f(v)$ measures the time at which v became black; thus let us denote f as finishing time. Let $I(v) = [d(v), f(v)]$ be the according interval of times. Furthermore, if node v has been discovered from node w via an arc, we say that w is the predecessor node of v - we write $w = p(v)$. With this definitions we formulate the following algorithms:

```
Algorithm DFSMain

  Specification:  Performs depth first search on directed graphs.
  Input:  Graph $G = (V, E)$.
  Output:  $d$, $f$, $p$

  1. Initialize:

      (a) color($v$) = white $\forall v \in V$
      (b) $p(v) =$ NaN $\forall v \in V$
```

```
    (c) time = 0
    (d) $E_p = \{\}$

 2. For all $u \in V$ do

    (a) If color($u$) = white, then do
        i. DFS($u$)

End of Algorithm DFSMain
```

```
Algorithm DFS

  Specification: Performs depth first search for predfined root node.
  Input: node $u$, Graph $G = (V, E)$, time, $E_p$
  Output: $d$, $f$, time, $p$, $E_p$

 1. color($u$) = gray

 2. $d\,(u)$ = time

 3. time = time+1

 4. For all $v \in Adj\,(u)$ do

    (a) If color($v$) = white, then do
          i. $p\,(v) = u$
         ii. $E_p = E_p \cup \{\langle u, v\rangle\}$
        iii. DFS($v$)

 5. color($u$) = black

 6. $f\,(u)$ = time

 7. time = time+1

End of Algorithm DFS
```

The algorithms `DFSMain` and `DFS` are known as depth first search in graphs. The runtime complexity of `DFSMain` and `DFS` is $O\,(|V|)$ and $O\left(\sum_{u \in V} g^+\,(u)\right) = O\,(|E|)$, respectively. Summing up we get an overall complexity of $O\,(|V| + |E|)$. Based on depth first search we can consider the following statement:

Theorem 2.5.1
Let $G_p := \left(V, E_p, \alpha|_{E_p}, \omega|_{E_p}\right)$ be the predecessor graph of the depth first search on a directed graph $G = (V, E; \alpha, \omega)$. Each weak connectivity component of G_p is a rooted tree.

Proof 2.5.1
See [5], pp. 146. □

Thus G_p represents a predecessor forest with rooted predecessor trees. Thus we denote them as depth first search forest and respectively depth first search trees, respectively.

Theorem 2.5.2
Let $u, v \in V$ be two nodes with $d(u) < d(v)$. Then exclusively one of the following statements holds:

1. $I(u) \cap I(v) = 0$ (disjoint intervals)
2. $I(v) \subset I(u)$ and the node v is a descendant of the node u in the depth first search forest G_p.

Proof 2.5.2
See [5], pp. 148 ff. $\square$

Theorem 2.5.3
All nodes of a strong connectivity component of G are in the same rooted depth first search tree.

Laboratory 2.5.1
In the directory `[Course]\Routines\DFS` you find the network `G01.ist` (basic *.ist file $\rightarrow$ section 6.1, page 80).

1. Compute the discovery- and the finishing time for each node using depth first search.
2. Visualize the depth first search forest.
3. Realize a non-recursive implementation of `DFS`.

For reference you will find the file `Laboratory01.m` in the directory `[Course]\Routines\DFS` as a demo implementation in `MATLAB`.

Laboratory 2.5.2
In the directory `[Course]\Routines\DFS` you find the network `G02.ist` (basic `*.ist` file $\rightarrow$ section 6.1, page 80). Find the node v of the network which can send his information to a maximum number of other nodes. For reference you will find the file `Laboratory02.m` in the directory `[Course]\Routines\DFS` as a demo implementation in `MATLAB`.

Laboratory 2.5.3
In the directory `[Course]\Routines\DFS` you find the network `G03.ist` (basic `*.ist` file $\rightarrow$ section 6.1, page 80). The root node $v = 1$ supplies all other nodes whereby each node u has an demand of $\pi(u)$. Find the minimum capacities of the arcs in the network such that the all nodes will recieve their neccessary amount of goods. For reference you will find the file `Laboratory03.m` in the directory `[Course]\Routines\DFS` as a demo implementation in `MATLAB`.

With respect to the depth first search one may classify the edges $e \in E$ as follows:

Definition 2.5.1
Let $G = (V, E, \alpha, \omega)$ be a directed graph and $G_p = \left(V, E_p, \alpha|_{E_p}, \omega|_{E_p}\right)$ be the according depth first search forest. Then an arc $e \in E$ is a

1. tree arc, if $e \in E_p$.

2. back arc, if $e \notin E_p$ and $\alpha(e)$ is a descendant of $\omega(e)$ in G_p. Self loops will always be considered as back arcs.

3. forward arc, if $e \notin E_p$ and $\omega(e)$ is a descendant of $\alpha(e)$ in G_p.

4. cross arc, if none of the criterions can be applied.

Obvioulsly one may easily check this conditions by application of theorem 2.5.2 based the results for d and f after depth first search. Moreover it is easy to augment the algorithm `DFS` in order to classify the arcs of a graph during the run of depth first search. Considering an arc $e \in E$ with $\alpha(e) = u$ and $\omega(e) = v$ we analyze:

1. If v is a white node, then e will be a tree arc, due to construction of the algorithm.

2. If v is gray, then e is a back arc. One may observe that the gray nodes form a path of descendants at each time of the algorithm `DFS`. Thus if we link from a gray node u to an allready gray node v it must be a back arc.

3. If v is black, then e is a forward arc if $d(u) < d(v)$ and a cross arc if $d(u) > d(v)$. This can be shown as follows: since u is gray and v is black the relation $f(v) < f(u)$ must hold. If $d(u) < d(v)$ then also $I(v) \subset I(u)$ holds; thus due to theorem 2.5.2 it follows that v is a descendant of u in the depth first search forest; thus e is a forward edge. Otherwise for $d(u) > d(v)$ it is clear that v can not be a descendant of u in the depth first search forest. Moreover it follows from theorem 2.5.2 that $I(v) \cap I(u) = 0$ such that u is also not a descendant of v. Summing up these two observations e must be a cross edge.

Laboratory 2.5.4
In the directory `[Course]\Routines\DFS` you find the network `G02.ist` (basic `*.ist` file $\rightarrow$ section 6.1, page 80). Perform depth first search on the network and classify the arcs. For reference you will find the file `Laboratory04.m` in the directory `[Course]\Routines\DFS` as a demo implementation in `MATLAB`.

We remark two applications of depth first search. One may test the network with respect to cycles using the following theorem:

Theorem 2.5.4
Depth first search yields back edges if and only if there are existing cycles in the according graph.

Proof 2.5.3
See [5], pp. 151 ff. □

Another application is to compute the strong connectivity components of a network. We give the following algorithm:

```
Algorithm StrgComp

  Specification:  Finds strong connectivity components in directed graphs.
  Input:  Graph $G = (V, E)$.
  Output:  Strong connection components.
```

```
1. Call [$d, f, p$] = DFSMain($G$)
2. Compute the inverse graph $G^{-1}$.
3. Call [$d^{-1}, f^{-1}, p^{-1}$] = DFSMain($G^{-1}$); run through nodes $u \in V$ in step 2 of
   DFSMain in decreasing order of their values $f(u)$
4. Each depth first search tree according to $p^{-1}$ represents a strong
   connectivity component in $G$.

End of Algorithm StrgComp
```

Theorem 2.5.5
The algorithm StrgComp correctly finds the strong connectivity components of a directed graph $G = (V, E, \alpha, \omega)$ with complexity $O(|V| + |E|)$.

Proof 2.5.4
See [5], pp. 153 ff. □

Laboratory 2.5.5
In the directory `[Course]\Routines\DFS` you find the network `G02.ist` (basic `*.ist` file → section 6.1, page 80). Find the strong connectivity components of the network. For reference you will find the file `Laboratory05.m` in the directory `[Course]\Routines\DFS` as a demo implementation in `MATLAB`.

Up to now we have exclusively considered directed graphs; however depth first search is also applicable to undirected ones $G = (V, E, \gamma)$.

- The algorithms `DFSMain` and `DFS` remain the same.
- Thus, in analogy to theorem 2.5.1, page 21, $G_p = (V, E_p)$ still represents a forest on G; each connectivity component $G' = \left(V', E'_p\right) \subseteq G_p$ is a rooted tree, whereby its root is specified by the unique node $v \in V'$ with $p(v) = 0$.
- Also the theorem 2.5.2, page 22, remains valid. This allows us to specify all ancestors and descendants within the depth first search trees.
- The classification of edges $e \in E$ is defined in analogy to the directed case; but we do not distinguish between back- and forward edges. We denote them all as back edges.
- In Analogy to theorem 2.5.3, page 22, the edge sets of each rooted tree in G_p define the connectivity components of G. Thus we may use depth first search for determining these components.

The formal correctness of these statements are shown in [5], pp. 156 ff.

Beside depth first search there exists another technique in order to run through graphs $G = (V, E, \alpha, \omega)$. Therfore we mesure the distance $d(s, v)$ of the node $v \in V$ from the node $s \in V$

as the minimal number of arcs which are neccessary to reach v from s; if $v \notin R(s)$ we set $d(s,v) = \infty$. So we can define the level set

$$V_i := \{v \in V : d(s,v) = i\}$$

which is computable by the following recursion:

$$V_0 = \{s\}$$

$$V_{i+1} = \left\{v \in V - \bigcup_{j=0}^{i} V_j : v \in N_+(V_i)\right\} \quad \text{for } i = 1, 2, 3, \ldots$$

The correctness can be seen by induction: if we assume that all nodes $w \in V_i$ are correctly assigned, which means that they are reachable through at most i arcs, then each $v \in N_+(V_i)$ is reachable through at most $i+1$ arcs. Obviously this searching technique considers all neighbours of a node at any time - thus we call this technique breadth first search (with root $s \in V$). This motivates the following algorithm:

```
Algorithm BFS

  Specification:  performs breadth first search on graphs.
  Input:  Graph $G = (V,E)$, root $s \in V$
  Output:  $d(s,v)$ for all $v \in V$.

   1. Initialize
      (a) $d(v) = \infty$ for all $v \in V - \{s\}$.
      (b) $d(s) = 0$.
      (c) $Q = \{s\}$.
   2. While $Q \neq \{\}$ do
      (a) Find that $u \in Q$ with the shortest distance $d(u)$ among all elements in $Q$.
      (b) Update $Q = Q - \{u\}$.
      (c) For all $v \in Adj(u)$ with $d(v) = \infty$ do
           i. $d(v) = d(u) + 1$.
          ii. $p(v) = u$.
         iii. Update $Q = Q \cup \{v\}$.

End of Algorithm BFS
```

Obviously breadth first search does also work in the undirected case.

Laboratory 2.5.6
In the directory `[Course]\Routines\DFS` you find the network `ClgN1BoundI1-01.ist` (basic `*.ist` file → section 6.1, page 80). The network stores the land use classes of a district of Cologne/Germany, the existing fiberoptic infrastructure (see section `I` in the file), possible connection points to the infrastuctur (see section `J` in the file) as well as connection objects (all nodes with type grater than 0). Check if all connection objects can be connected to one of the connection points of the existing infrastucture; if it is possible determine the number of edges which are at least neccessary for doing that. For reference you will find the file `Laboratory07.m` in the directory `[Course]\Routines\DFS` as a demo implementation in `MATLAB`.

Theorem 2.5.6
The algorithm BFS computes the distances of $v \in V$ from $s \in V$ correctly - the computational complexity is $O(|V| + |E|)$.

Proof 2.5.5
See [5], pp. 159 ff. □

2.6 Optimal trees in graphs

2.6.1 Basic algorithms for trees

If the nodes of a graph $G = (V, E, \gamma)$ represent terminals and the edges are the possible links between these terminals, then a spanning tree seems to be the 'natural' approach for an efficient connection of the terminals. Efficiency may be explained that we will have to use only $|V| - 1$ edges to realize that connection. If we intruduce a weight (cost) function $c : E \to \mathbb{R}_+$ on the edges (leasing of transmission costs on $e \in E$) we can formulate the following problem:

Definition 2.6.1 (Minimum spanning tree and forest - MST and MSF)
Find a spanning tree $T \subseteq G$, respectively a spanning forest $F \subseteq G$ if G is not connected, such that the total costs

$$c(T) = \sum_{e \in E(T)} c(e) \quad \text{respectively} \quad c(F) = \sum_{e \in E(F)} c(e)$$

are minimal.

Obviously a spanning forest F on G represents a MSF if each tree in F represents a MST on a connectivity component of G. Thus searching a MSF on a connected graph means per definition to search a MST. Therefore we discuss the MSF problem furtheron.

Definition 2.6.2 (Error-free edge sets and secure edges)
Let $G = (V, E, \gamma)$ be an undirected graph with a weight function on E. Then we say that an edge set $F \subseteq E$ is error-free, if there exists a minimum spanning forest F^* such that $F \subseteq F^*$. We call an edge $e \in E$ secure for the error-free edge set F if $F \cup \{e\}$ is also error-free.

Theorem 2.6.1
Let $F \subset E$ be an error-free edge set and (A, B) be a cut such that $\delta(A) \cap F = 0$. The edge $e \in \delta(A)$ with the lowest costs among all edges in $\delta(A)$ is secure for F.

Proof 2.6.1
Let F^* be an arbitrary MSF and $F \subset F^*$ be an error-free edge set in G. Furthermore let $e \in \delta(A)$ such that $c(e) \leq c(f)$ for all $f \in \delta(A)$. Now let us assume that $e \notin F^*$ which means that e is not secure for F. For a schematic view we refer to figure 2.2. Obviously each edge which is

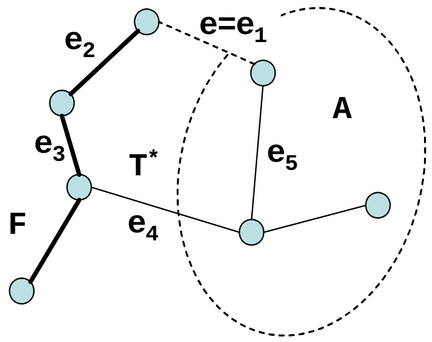


Figure 2.2: Schematic view of proof 2.6.1.

not in F^* connects two nodes which are spanned by an MST $T \subseteq F^*$. Thus $T + \{e\}$ contains an elementary cycle $C = \{v_0, e_1, v_1, \ldots, e_k, v_k = v_0\}$ wherby without loss of generality $e = e_1$. Then there must exist an edge $e_i \in C$ with $e_i \neq e$ and $e_i \in \delta(A)$. Since $c(e_i) \geq c(e)$ the modified tree $T' = (V(T), E(T) - \{e_i\} + \{e\})$ meets $c(T') \leq c(T)$. Thus if $c(T') < c(T)$ we get a contradiction to the optimality of F^* which means that e has to be secure. In the case $c(T') = c(T)$ we may simply replace e_i by e and will get another MSF - thus e is also secure. $\square$

Corollary 2.6.1
Let $F \subset E$ be an error-free edge set and U be a connectivity component of the subgraph (V, F). Then the edge $e \in \delta(U)$ with the lowest costs among all edges in $\delta(U)$ is secure for F.

Proof 2.6.2
Follows from theorem 2.6.1, since $\delta(U) \cap F = 0$. $\square$

Thus we can prove the correctness of the following alogirthm:

```
Algorithm MSFKruskal

  Specification:  computes a MSF for an undirected graph.
  Input:  graph $G = (V, E)$, weight function $c$.
  Output:  edge set $E_F \subseteq E$ of MSF.

  1. Sort all edges such that $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$ holds.

  2. Initialize $E_F = \{\}$.

  3. For $i = 1, \ldots, m$ do

      (a) If $(V, E_F + \{e_i\})$ does not contain any cycle, then
```

```
        i. Update $E_F = E_F + \{e_i\}$.

 End of Algorithm MSFKruskal
```

Laboratory 2.6.1
In the directory `[Course]\Routines\Trees` you find the network `G01.ist` (basic `*.ist` file $\rightarrow$ section 6.1, page 80). Compute the costs of the MST in the Euclidean sens, i.e. $c(e) = \|u - v\|_2$ for $\gamma(e) = \{u, v\}$ as well as for the edge weights provided in the file. For reference you will find the file `Laboratory01.m` in the directory `[Course]\Routines\Trees` as a demo implementation in `MATLAB`.

Theorem 2.6.2
The algorithm `MSFKruskal` computes an MSF. If the graph is connected the algorithm computes a MST.

Proof 2.6.3
We prove that the algorithm constructs a MST for a connected graph. Then, if G is not connected, we can apply the same argumentation to each connectivity component of G and can therfore conclude the construction of a MSF.
Let $G_1, \ldots, G_p$ be the connectivity components of the subgraph (V, E_F). If $p = 1$ we are finished and (V, E_F) is a spanning tree (notice: the graph is connected and does not contain any cycle due to the if-statement 3.a). If $p > 1$ then there must exist an edge $e \in \delta(V(G_i))$ since G is a connected graph. Obviously $e \notin E_F$ and for $\gamma(e) = \{u, v\}$ the conditions $u \in V(G_i)$ and $v \in V(G_j)$ for $i \neq j$ must hold. Thus this edge will pass the the if-condition in step 3.a and will be added to E_F (notice: adding an edge linking two cycle-free components will never construct a cycle).
Thus in order to show the optimality of the constructed solution it remains to prove that this edge e was secure for E_F. This is easy since $e \in \delta(V(G_i))$ and e is the edge with minimal costs among all edges in $\delta(V(G_i))$ - this is due to the initial sorting: every other edge with potential lower costs than e would have been tested earlier. Therefore corollary 2.6.1 tells us that e was secure. □

Obviously the complexity of the algorithm `MSFKruskal` is codetermined by the step 3.a. We can implement the test if an edge e_i does construct an elementary cycle in (V, E_F) in a simple way. Therefore we notice that e_i, with $\gamma(e_i) = \{u, v\}$, does not construct a cycle if and only if it connects two different connectivity components V_u and V_v of (V, E_F). Thus we have to implement a data structure which contains the nodes in the according connectivity components of (V, E_F) during the run of the algorithm. Thus we define the function $K : V \rightarrow \mathbb{N}$, whereby $K(v)$ means the index of the connectivity component to which the node v currently belongs.

```
 Algorithm MSFKruskal

   Specification:  computes a MSF for an undirected graph.
   Input:  graph $G = (V, E)$, weight function $c$.
   Output:  edge set $E_F \subseteq E$ of MSF.

  1. Sort all edges such that $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$ holds.
 2. Initialize $E_F = \{\}$.

 3. For all $v_i \in V$ do

     (a) $K(v) = i$.

 4. Initialize $k = 0$.

 5. While $\max_{v \in V} K(v) > 1$ do

     (a) Update $k = k + 1$.
     (b) Let $\gamma(e_k) = \{u, v\}$.
     (c) If $K(u) \neq K(v)$ then do
           i. Update $E_F = E_F + \{e_k\}$.
          ii. For all $\{w \in V : K(w) = \max\{K(u), K(v)\}\}$ do
              ($\alpha$)   $K(w) = \min\{K(u), K(v)\}$.

End of Algorithm MSFKruskal
```

Theorem 2.6.3
The algorithm MSFKruskal can be implemented with complexity $O(|E| + |V| \log |V|)$.

Proof 2.6.4
See [5], pp. 106 ff. □

Another useful and quiet simple method for computing a MST for an undirected, simple and connected graph is the algorithm of Prim:

```
Algorithm MSTPrim

  Specification:  computes a MST for an undirected, simple, connected graph.
  Input:  graph $G = (V, E)$, weight function $c$.
  Output:  edge set $E_T \subseteq E$ of MST.

 1. Choose an arbitrary $s \in V$.

 2. Initialize $E_T = \{\}$ and $S = \{s\}$.

 3. While $S \neq V$ do

     (a) Choose an edge $(u, v) \in \delta(S)$ with minimal costs among all edges in $\delta(S)$.
     (b) Update $E_T = E_T \cup \{(u, v)\}$ and $S = S \cup \{u, v\}$.

End of Algorithm MSTPrim
```

Theorem 2.6.4
The algorithm MSTPrim computes a MST.

Proof 2.6.5
The correctness of this algorithm follows immediately from crollary 2.6.1 by induction. □

The complexity of the algorithm `MSTPrim` is obviously determined by step 3.a, namely the determination of the minimum weighted cut edge with respect to $S \subset V$. One way to implement this step in an efficient way is the following: for $v \in V \backslash S$ let $e(v)$ be the minimum weighted edge among all edges vom v to any node in S and $d(v)$ be the according weight of $e(v)$. At the initialization step 1 we set $e(v) = NaN$ and $d(v) = \infty$ for all $v \in V$. Thus at step 3.a we have only to select the node u with the lowest marker $d(u) \neq \infty$. Notice such a node must be existent due to the assumed conectivity of the graph. After adding u to S we investigate all neighbours $v \in N(u) \cap V \backslash S$: if $c(u, v) < d(v)$ then we update $e(v) = (u, v)$ and $d(v) = c(u, v)$.

```
Algorithm MSTPrim

  Specification: computes a MST for an undirected, simple, connected graph.
  Input: graph $G = (V, E)$, weight function $c$.
  Output: edge set $E_T \subseteq E$ of MST.

  1. Choose an arbitrary $s \in V$.
  2. Initialize $e(v) = NaN$ and $d(v) = \infty$ for all $v \in V$.
  3. Initialize $d(s) = 0$.
  4. Initialize $S = \{\}$.
  5. While $S \neq V$ do
      (a) Extract $u \in V \backslash S$ with the minimum value $d(u)$ among all nodes in $V \backslash S$.
      (b) Update $S = S \cup \{u\}$.
      (c) For all $v \in Adj(u) \cap V \backslash S$ do
          i. If $c(u, v) < d(v)$ then $d(v) = c(u, v)$ and $e(v) = (u, v)$.
  6. $E_T = \{e(v) : v \in V \backslash \{s\}\}$.

End of Algorithm MSTPrim
```

The steps 5 and 5.a require $|V|^2$ iterations at most, the step require 5.c $|E|$ at most; thus the complexity of this version of `MSTPrim` is $O\left(|V|^2 + |E|\right) = O\left(|V|^2\right)$, since $O(|E|) = O\left(|V|^2\right)$.

Laboratory 2.6.2
In the directory `[Course]\Routines\Tools` you find the the routine `GRand.m`. Use this routine in order to construct a representative population of sparse and dense connected graphs with $10 \leq |V| \leq 200$. Compare the routines `MSFKruskal` and `MSTPrim` with respect to their real run time complexity in finding according MST. Motivate your answers statistically.

2.6.2 A MST heuristc for the metric Traveling Salesman Problem (TSP)

Based on the algorithms above we can give a heuristic approach for the TSP on the complete graph $G = (V, E)$ with weight function $c : E \to \mathbb{R}_+$ ($\to$ problem 1.4.4, page 7). We further assume that c is metric, which means that c meets the triangle inequaltity

$$c(u,v) \leq c(u,w) + c(w,v) \quad \text{for all } u, v, w \in V.$$

It can be shown ([5], pp.133 resp. theorem 3.35 on page 50) that even this metric case of the TSP is in the complexity class NP. Thus we are searching for an polynomial algorithm which constructs an approximation of the minimum weighted Hamiltonian cycle on G.
Therfore let C^* be the optimal Hamiltonian cycle with weight $c(C^*) = OPT$. We know that $|E(C^*)| = n$ for $|V| = n$; this implies

$$\max_{e \in E(C^*)} c(e) \geq \frac{OPT}{n}$$

Let e be the according edge in C^* with maximum weight. Furthermore $C^* - \{e\}$ represents a spanning tree on G. Thus also

$$c(T) \leq OPT - c(e)$$

holds if T is a MST on G. Summarizing it follows that for a MST T on G the condition

$$c(T) \leq OPT - c(e) \leq OPT - \frac{OPT}{n} \leq \left(1 - \frac{1}{n}\right) OPT$$

holds. Based on this observation we may construct a Hamiltonian cycle as follows (see figure 2.3 for a schematic view): let T' be the graph resulting from doubeling all edges in T. Then T' is an

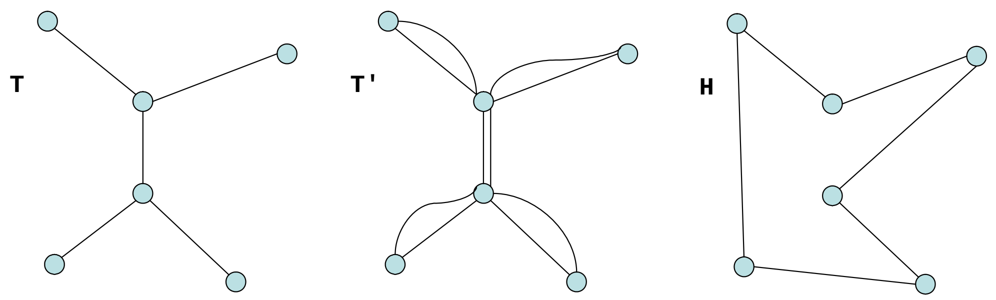


Figure 2.3: Schematic view showing the strategy of the MST heuristic for the TSP.

Euler graph (see theorem 2.3.4, page 18) containing an Euler cycle

$$K = (v_0, e_1, v_1, \ldots, e_k, v_k = v_0)$$

with

$$c(K) = 2c(T) \leq \left(2 - \frac{2}{n}\right) OPT.$$

We start at v_0 and walk along K and accept the walk as Hamiltonian cycle H as long as we meet an already visited node - without loss of generality let v_i be such a node. Thus we have the situation:

$$H = (v_0, e_1, v_1, \ldots, e_{i-1}, v_{i-1})$$

Then we take a shortcut from v_{i-1} to the next not visited node in K - let this node be v_j and we get

$$H = (v_0, e_1, v_1, \ldots, e_{i-1}, v_{i-1}, f, v_j) \quad \text{with } f = (v_{i-1}, v_j)$$

Due to the triangle inequality the weight $c(f)$ is lower or equal to the weight of the v_{i-1}, v_j-path in K. Thus completing this strategy till we reach v_0 yields a Hamiltonian cycle H with

$$c(H) \leq c(K) \leq \left(2 - \frac{2}{n}\right) OPT.$$

Thus we may formulate the following algorithm:

```
Algorithm TSPMST

  Specification: computes an $(2 - \frac{2}{n})$-approximation for the metric TSP.
  Input: complete graph $G = (V, E)$, metric weight function $c$.
  Output: Hamiltonian cycle $H \subseteq E$.

  1. Compute a MST $T$ on $G$.
  2. For all $e \in E(T)$ do
     (a) Update $E(T) = E(T) + \{e\}$ (add an additional parallel edge to each $e$).
  3. Compute the Euler cycle $K = (v_0, e_1, v_1, \ldots, e_k, v_k = v_0)$ on $T$. (→ Algorithm
     Euler, section 2.3, page 15).
  4. Initialize $H = ()$.
  5. Initialize $i = 0$.
  6. While $i < k$ do
     (a) Update $H = H + (f, v_i)$ with $f = (\omega(H), v_i)$.
     (b) While $v_i \in V(H)$ and $i < k$ do
         i. Update $i = i + 1$.
  7. Update $H = H + (f, v_0)$ with $f = (\omega(H), v_0)$.

End of Algorithm TSPMST
```

Theorem 2.6.5
The algorithm TSPMST computes a $\left(2 - \frac{2}{n}\right)$-approximation for the metric TSP with complexity $O\left(n^2\right)$.

Proof 2.6.6
We get the complexity $O\left(m+n\log n\right) = O\left(n^2\right)$ for solving the MST using Kruskal since the graph is complete, $O\left(m'+n\right)$ for computing the Euler cycle wherby $m' = 2\left(n-1\right)$ and we have iterate in step 6 at most $\left|E\left(K\right)\right| = 2\left(n-1\right)$ times. Summing up this yields a complextiy of $O\left(n^2\right)$. $\square$

Laboratory 2.6.3
In the directory `[Course]\Routines\Tools` you find the the routine `GRand.m`. Use this routine in order to construct a complete graph with metric edge weights - use the option `'euclidean'`. Implement the algorithm `TSPMST` and compute a near optimal tour visiting all nodes. For reference you will find the file `Laboratory02.m` in the directory `[Course]\Routines\DFS` as a demo implementation in `MATLAB`.

2.6.3 Spanning trees with degree constraints

In this section we are interested in finding spanning trees with degree constraints for undirected graphs $G = (V, E, \gamma)$. The descision problem whether G contains a spanning tree T with $\Delta\left(T\right) \leq k$, $k \in \mathbb{N}$, or not is NP-complete. This can easily be seen - for $k = 2$ this problem is equivalent to find a Hamiltonian path in G which is known to be NP-complete.

An associated optimization problem is the minimum degree spanning tree problem: search a spanning tree T^* with a minimal degree $\Delta\left(T^*\right)$. Furtheron we will give an quality-proven approximation algorithm for this problem. This algorithm bases on a sufficient condition identifying trees which are 'near' the optimal solution. In order to prove this condition we need the following lemma:

Lemma 2.6.1
Let $G = (V, E, \gamma)$ be an undirected graph and $W \subset V$ be a subset of nodes. Let $V_1, \ldots, V_p$ be the connectivity components of the graph $G - W$. Then the inequaltiy

$$\Delta\left(T^*\right) \geq \left\lceil \frac{|W| + p - 1}{|W|} \right\rceil$$

holds.

Proof 2.6.7
Consider the following partition of the node set V to the sets $V_1, \ldots, V_p$ and the one-element sets $w \in W$ - this partition obviously contains $|W| + p$ node sets. Thus each spanning tree on G has at least $|W| + p - 1$ edges (compare theorem 2.4.1, page 19). None of these edges connects a node in V_i with one in V_j, $i \neq j$; otherwise V_i and V_j would not be connectivity components. Thus all of these $|W| + p - 1$ edges are incident to at least one node in W. Thus the avarage degree of the nodes in W is at least $\left(|W| + p - 1\right) / |W|$ - the degree is an integer and therefore we are allowed to round up. $\square$

With this lemma we are in the position to prove the following sufficient condition for a 'near-optimum' spanning tree T:

Theorem 2.6.6
Let T be a spanning tree for the undirected graph $G = (V, E, \gamma)$ with the maximum degree $\Delta(T) = k$. Let S_k and S_{k-1} be the set of nodes which have a degree k and $k-1$ in T; i.e. $g_T(v) = k$ for all $v \in S_k$ and $g_T(v) = k-1$ for all $v \in S_{k-1}$. For an arbitray $U_{k-1} \subseteq S_{k-1}$ we consider the subgrah $T - (S_k \cup U_{k-1})$: if no edge in G can be found which connects two connectivity components of this subgraph, then the inequality $k \leq \Delta(T^*) + 1$ holds.

Proof 2.6.8
First we observe that the connectivy components of $T - (S_k \cup U_{k-1})$ and $G - (S_k \cup U_{k-1})$ are identically: since $T \subseteq G$ the inclusion $T - (S_k \cup U_{k-1}) \subseteq G - (S_k \cup U_{k-1})$ holds; since there exists no edge in G connecting any two of the components of $T - (S_k \cup U_{k-1})$ also the inclusion $T - (S_k \cup U_{k-1}) \supseteq G - (S_k \cup U_{k-1})$ is true. Thus let us furtheron investigate the components of $T - (S_k \cup U_{k-1})$ - see figure ? for a schematic view. Obvisously the number of them has to be equal to the number of edges in the cut $\delta_T(S_k \cup U_{k-1})$, i.e.

$$|\delta_T(S_k \cup U_{k-1})| = \sum_{w \in S_k \cup U_{k-1}} g(w) - 2\eta$$

whereby η is the number of edges in T which connectes two nodes in $S_k \cup U_{k-1}$. Since T is a tree,

$$\eta \leq |S_k \cup U_{k-1}| - 1 = |S_k| + |U_{k-1}| - 1$$

holds, and thus

$$|\delta_T(S_k \cup U_{k-1})| \geq k\,|S_k| + (k-1)\,|U_{k-1}| - 2\,(|S_k| + |U_{k-1}| - 1) =: q$$

Thus we know that $T - (S_k \cup U_{k-1})$ has at least q connectivity componentes - this is also true for $G - (S_k \cup U_{k-1})$. Applying lemma 2.6.1 to $W := S_k \cup U_{k-1}$ yields the estimate

$$\begin{aligned}
\Delta(T^*) &\geq \left\lceil \frac{|W| + p - 1}{|W|} \right\rceil \geq \left\lceil \frac{|W| + q - 1}{|W|} \right\rceil = \\
&= \left\lceil \frac{|S_k| + |U_{k-1}| + k\,|S_k| + (k-1)\,|U_{k-1}| - 2\,(|S_k| + |U_{k-1}| - 1) - 1}{|S_k| + |U_{k-1}|} \right\rceil = \\
&= \left\lceil \frac{(k-1)\,(|S_k| + |U_{k-1}|)}{|S_k| + |U_{k-1}|} - \frac{|U_{k-1}| - 1}{|S_k| + |U_{k-1}|} \right\rceil = \left\lceil (k-1) - \frac{|U_{k-1}| - 1}{|S_k| + |U_{k-1}|} \right\rceil
\end{aligned}$$

Since

$$\frac{|U_{k-1}| - 1}{|S_k| + |U_{k-1}|} \leq 1$$

we see that $\Delta(T^*) \geq k - 1$. $\square$

Thus we are interested in the construction of an algorithm which starts with an arbitrary spanning tree T on G (e.g. a MST) and which modifies T iteratively as long as the conditions of theorem 2.6.6 are met. Such an algorithm is based on two principles: the improvement step and blocking.

Let us assume that $\Delta(T) = k$ and we consider an edge $e \in E$ with $e \notin E(T)$, $\gamma(e) = \{u, v\}$ whereby $g_T(u) \leq k-2$ and $g_T(v) \leq k-2$. The graph $T + \{e\}$ contain an elementary cycle C (compare theorem 2.4.1, page 19). If there exists a node $w \in V(C)$ with $g_T(w) = k$ one may update $T' := T + \{e\} - \{f\}$ whereby f is an edge in C and incident to w. Thus for the updated

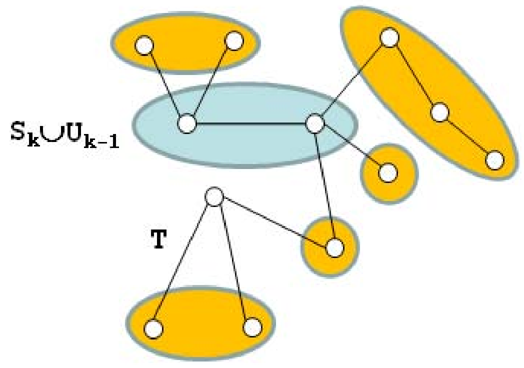


Figure 2.4: Schematic view for the proof 2.6.8 in case of $k = 4$.

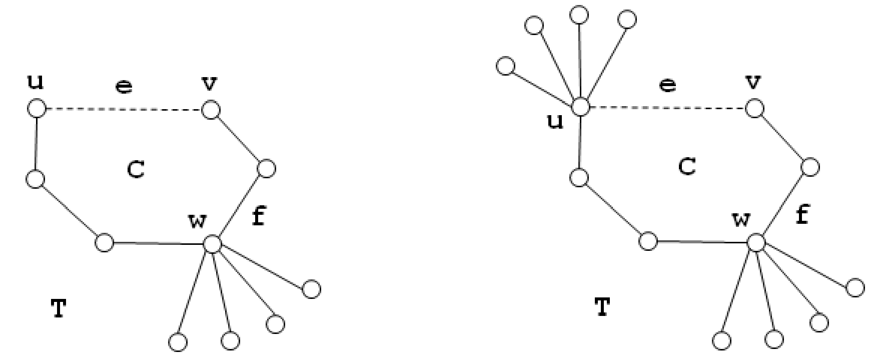


Figure 2.5: Left plot: improvement step; right plot: situation where u is a blocking for the considered edge e.

tree follows that $g_{T'}(w) = k-1$, $g_{T'}(u) \leq k-1$, and $g_{T'}(v) \leq k-1$. Such an update will be denoted as an improvement step furtheron. Moreover we say that the node w profits by this improvement step. On the other side, if we consider an edge $e \notin E(T)$, $\gamma(e) = \{u, v\}$ which does not meet one or both of the node degree constraints, i.e. $g_T(u) \geq k-1$ or $g_T(v) \geq k-1$, the step above would not improve the situation. In this case we would say that u or v (or both of them are) is a blocking for the edge e. The figure 2.5 gives a schematic view for these two principles. We formulate the following algorithm:

```
Algorithm MINGRDT

Specification: computes a spanning tree $T$ on an undirected graph such that the
maximimum degree of $T$ is at most $\Delta(T^*) + 1$.
Input: undirected graph $G = (V, E, \gamma)$, arbitrary spanning tree $T$.
Output: spanning tree $T$.

  1. Loop,
```

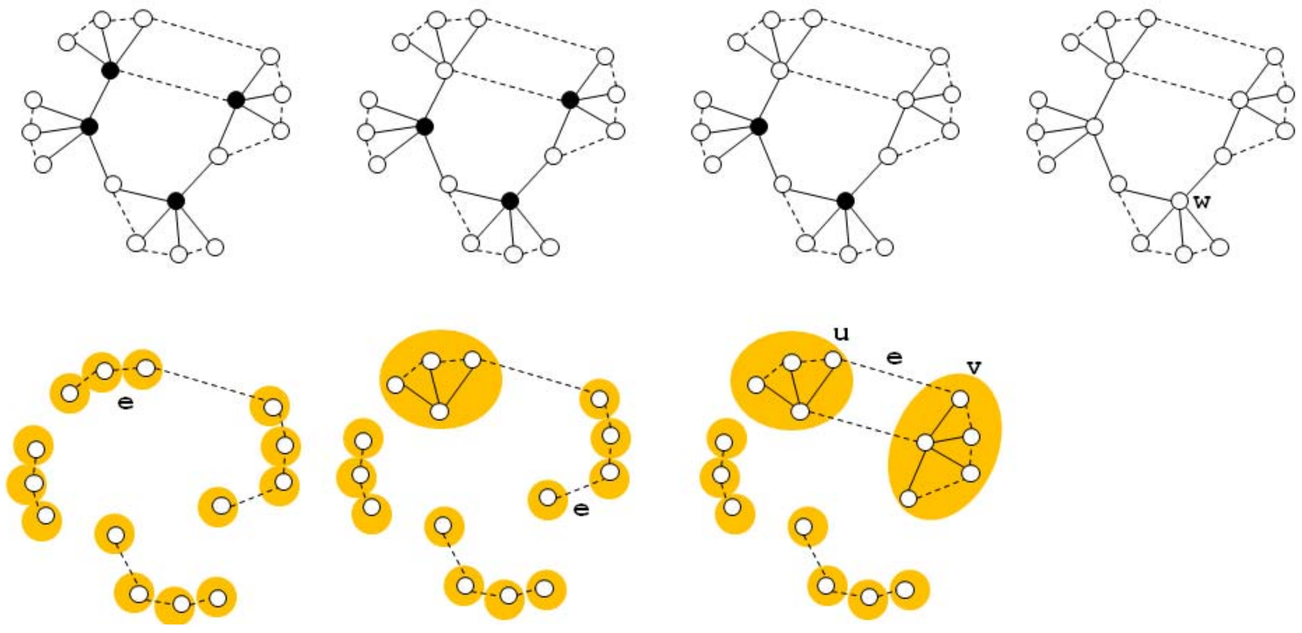


Figure 2.6: Three iterations of the inner-while loop of algorithm MINGRDT. T is shown by the solid lines; the dashed ones are edges not in T. Black filled nodes are bad ones; i.e. nodes in S. The filled areas in the lower line plots show the according connectivity components of $T - S$.

```
(a) Let $k := \Delta(T)$.
(b) Mark all nodes $v \in V$ which meet $g_T(v) \geq k-1$ to be bad ones (in the sens
    that such a $v$ represents a blocking for any incident edge). Logically
    the remaining nodes are good ones.
(c) Let $S$ be the set of bad marked nodes.
(d) Let $F := T - S$.
(e) While all nodes with degree $k$ in $T$ are marked as bad ones and there
    exists an edge $e$ in $E$, which connects different components of $F$, do
      i. Let $C$ the unique elementary cycle in $T + \{e\}$.
     ii. Remark all bad nodes in $V(C)$ to be good ones and update $S$ as well
         as $F$.
(f) If there exists a good marked node $w$ with degree $k$ in $T$, then an
    improvement step with respect to the currently considered edge $e$ is
    possible and yield an updated tree $T$.
(g) Else terminate the loop an pass back $T$ as solution.
```

End of Algorithm MINGRDT

The algorithm runs through an outer loop whereby each iteration contains an inner while loop. Obviously, the inner loop iteration does not start anymore if one of the following conditions are met:

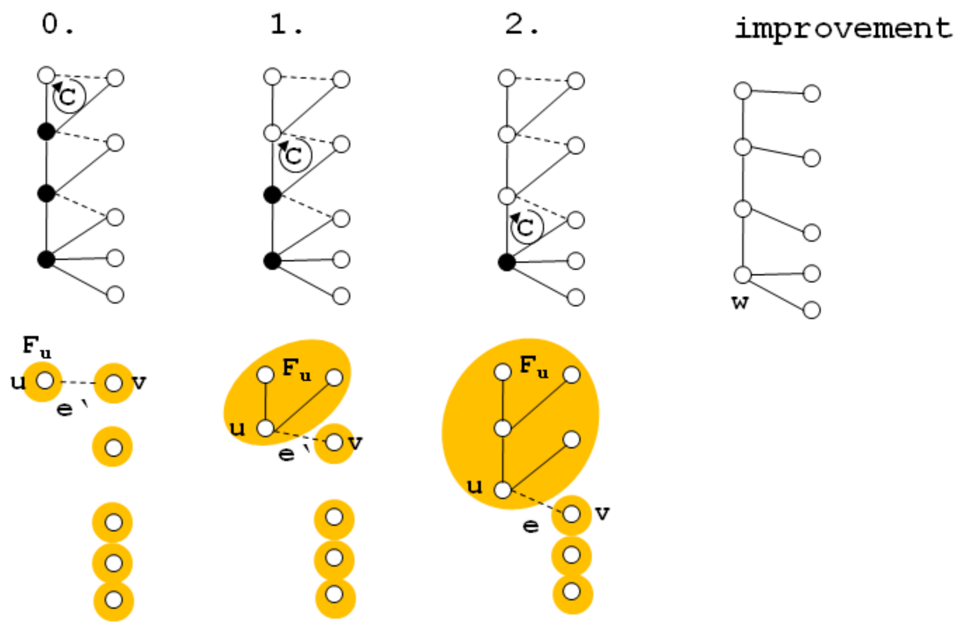


Figure 2.7: Two iterations of the inner while loop lead to a blocking for node w.

1. A node of degree k has been marked as good in the current iteration with respect to the edge e with $\gamma(e) = \{u, v\}$. This calls the step 1.f. - we will show below that this case is sufficient for the feasiblity of an improvement step.

2. If case 1 is not met then we can not find any edge e in E which connects any two of the components of $T - S$ - this lead us to the termination of the outer loop (and the whole algorithm) in step 1.g. Notice that S consists of the node set S_k containing all nodes of degree k in T (since no node of degree k which has been marked as bad initially has been remarked to be good up to now) and a subset U_{k-1} of the edge set S_{k-1} containing all nodes of degree $k - 1$ in T. In this case the current tree T meets the conditions of theorem 2.6.6.

If the inner-while loop starts and we consider an edge e, $\gamma(e) = \{u, v\}$, which connects two components of $T - S$, then the cycle C in $T + \{e\}$ contains at least one bad marked node - otherwise u and v would not be in different components of $T - S$. This observation ensures that the inner loop will terminate - either we mark a node with degree k as good or we do not find any component-connecting edges anymore. The figure 2.6 gives a schematic view for such a case - after three inner loop iterations we have found a node w of degree k which is marked as good.

We have to describe the step 1.f. in more detail. In the situation depicted in figure 2.6 neither u nor v are blockings for e and w profits from the impovement step $T := T + \{e\} - \{f\}$ whereby f is an edge in the cycle C (contained by $T + \{e\}$) and incident to w. Thus step 1.f. will be completed in this case. But let us study the situation depicted in figure 2.7 - in this case we choose another edge e, $\gamma(e) = \{u, v\}$, in the third iteration of the inner while loop. In this case both nodes, u and v, are blockings for the edge e. Let us study the situation for u in more detail - for v it still remains the same. We observe that:

- u is a good marked node - otherwise it would not be in one of the two connectivity components of $T - S$ connected by e. Let T_u be the according component containing the node u.

- Since u is a blocking, it has a degree of $k-1$ in T. Such nodes are initially marked as bad ones - thus u has been remarked during the inner-while loop iteration to be a good one. Such a node has been within a cycle contained by $T + \{e'\}$ during any inner-while loop iteration - see figure 2.7 for reference. Under the assumption that none of the nodes which are incident to e' are blockings we may perform an improvement step for u; if one of them is also a blocking (as depicted in the szenario of figure 2.7), we have to loop recursively with this local improvments for the occurring blockings. The rucursion will terminate: at least at the initial situation of the inner while loop (step 0 in figure figure 2.7) u and v are no blockings - otherwise, ii one of them has a degree grater than $k-2$, it would not be marked as good.

- After removing the blocking status of u (and optionally v) we are in the position to perform an improvement step with respect to the edge e such that w profits - this completes the step 1.f. The de-blocking procedure and improvement step are depicted in the upper line plots of figure 2.7.

Laboratory 2.6.4
Create a graph $G = (V, E, \gamma)$. Each node represents an individual of a society. Each individual has a geographic position (x and y coordinates). An edge $e = (u, v) \in E$ means that the individuals u and v are in the position to support among themselves; for example let $e \in E$ if the Euclidean distance between u and v is small enough. We are intended to find a support structure within the society such that the society is well connected by the support structure; we are not intended to create sub-societies. Moreover the effort shall be well balanced among all individuals within the intended support structure; this means that no individual shall support much more neighbours than the other ones do. For reference you will find the file `Laboratory03.m` in the directory `[Course]\Routines\Trees` as a demo implementation in `MATLAB`.

2.6.4 Optimal spanning rooted trees

We give at first a criterion which checks the existence of a spanning rooted tree:

Theorem 2.6.7
Let $G = (V, E, \alpha, \omega)$ be a directed graph and $s \in V$. Then G contains a spanning rooted tree with root s if and only if $R(s) = V$.

Proof 2.6.9
"$\Rightarrow$": trivial.
"$\Leftarrow$": G contains allways a rooted tree, e.g. $(\{s\}, 0)$ is the simplest one. Thus we have to show that a rooted tree T with $V(T) = V$ exists. We will do this by contradiction: no such tree exists and therefore T is the rooted tree with maximum set $U := V(T) \subset V$. Then obviously $(U, V \backslash U)$ represents a cut, whereby there must exist an edge $e \in \delta^+(U)$. If no such an edge exists then the nodes in $V \backslash U$ would not be reachable from any node in U and therefore also not from s - this would be a contradiction to $R(s) = V$. Thus $T + \{e\}$ would be a feasible augmentation and yield $U \subset V(T + \{e\})$ which is a contradiction. $\square$

Remark 2.6.1
This proof is constructive and allows the construction of a spanning rooted tree. Reading this proof observantly, one may recognize the algorithm `Reachable` (page 12). If we initialize $T = \{\}$ and implement the update step `2.c.iii` with $T = T + \{\langle v, w\rangle\}$ then this algorithm will yield the according rooted tree or will detect that such a tree does not exist with polynomial complexity. More generally, the depth-first- or breadth-first search algorithms discussed in section 2.5 are useful methods in order to decide whether a rooted tree exists; if so the search tree would be the intended result.

In the following we are not speciously interested in any rooted tree but rather in an optimal one. Therefore let $G = (V, E, \alpha, \omega)$ be a directed and strong connected graph with the weights $c : E \to \mathbb{R}_+$. Thus we search a spanning tree with root $s \in V$ and a minimal total weight $c(T)$ among all spanning trees with the same root. The solution of this problem is equivalent to find a minimum spanning rooted arborescence with root s on G. For a definition of this problem and a reference how to solve it we refer to the section 4 on page 71.

Laboratory 2.6.5
In the directory `[Course]\Routines\Tools` you find the the routine `GRand.m`. Use this routine in order to construct an arbitrary directed graph. For an arbitrily chosen root compute the according minimal spanning rooted tree or show that such a tree does not exist. For reference you will find the file `Laboratory04.m` in the directory `[Course]\Routines\Trees` as a demo implementation in `MATLAB`.

2.6.5 Further topics

- MST algorithm of Fredman and Tarjan ([5], pp.120 ff)
- MST algorithm of Boruvka ([5], pp.123 ff)

2.7 Optimal paths in graphs

2.7.1 Shortest paths

We consider directed graphs $G = (V, E, \alpha, \omega)$ whereby parallel arcs are excluded. Moreover each arc is penalized by according costs; i.e. $c : E \to \mathbb{R}$ and $c(e)$ are the costs of the arc $e \in E$. For a path P in G, $P := (v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$, $\alpha(e_i) = v_{i-1}$ and $\omega(e_i) = v_i$ for $i = 1, \ldots, k$, we define the length $c(P)$ of a path as

$$c(P) = \sum_{i=1}^{k} c(e_i)$$

The trivial path $P = (v)$ has length $c(P) = 0$. For two nodes u and v in V we define the distance $d_c(u, v, G)$ between u and v in G with respect to the cost function c as follows:

$$d_c(u, v, G) := \inf\{c(P) : P \text{ is a path from } u \text{ to } v\}$$

Notice that the index c and the graph identifier G may be omitted if it is clear in which context the distance is considered - thus we may write $d(u, v)$ instead of $d_c(u, v, G)$.

Laboratory 2.7.1
Why do we exclusively consider graphs without parallel arcs?

On the basis of this definition we observe:

- In general $d_c(u, v, G) \neq d_c(v, u, G)$.
- If $v \notin R(u)$ (there exists no path from u to v), then due to the infimum definition of the distance $d_c(u, v, G) = \infty$ holds.
- If a cycle C with $c(C) < 0$ (cycle of negative length) is reachable from a node u, then we observe that $d_c(u, v, G) = -\infty$ for all $v \in R(u)$ (obviously we may loop endlessly in C which causes an infinite number of pathes with decreasing costs).
- If $u = v$ we have two cases: if no cycle of negative length exists which contains u (or v) then $d_c(u, v, G) = 0$, otherwise $d_c(u, v, G) = -\infty$.

We fomulate the following lemma:

Lemma 2.7.1
Let $G = (V, E, \alpha, \omega)$ be a directed graph with cost function $c : E \to \mathbb{R}$. Then the following statements are equivalent:

1. $d_c(u, v, G) = -\infty$.
2. There exists a path P from u to v which contains at least one cycle of negative length (i.e. there exists at least one node w in P, which is contained by a cycle of negative length).

Proof 2.7.1
"(2)$\Longrightarrow$(1)". Proven according to the observations above.
"(1)$\Longrightarrow$(2)". Let $d_c(u, v, G) = -\infty$ and we assume that no cycle of negative length exists on any path P from u to v. Then the length of each path from u to v will decrease, if we delete a cycle from it (since all cycles are non-negative due to assumption). Thus in the case of $v \in R(u)$ the distance between u and v is determined by a cycle free path from u to v - the length of such a path is allways finite. Otherwise if $v \notin R(u)$ then the distance is positive infinite per definition. Both cases are a contradiction to $d_c(u, v, G) = -\infty$; thus the assumption is false. $\square$

2.7.2 Problems

Once again we consider a directed graph $G = (V, E, \alpha, \omega)$ with cost function $c : E \to \mathbb{R}$.

Single pair shortes path problem (SPSP). For two nodes $s, t \in V$ compute $d(s, t)$; i.e. find a path P with minimum length in G from s to t; in this context P will be denoted as the shortest s, t-path or shortest path from s to t.

Single source shortest path problem (SSSP). For a given node $s \in V$ compute $d(s, v)$ for all $v \in V$; i.e. find pathes with minimum lengths in G from s to all nodes in the graph or find the shortest s, v-paths for all $v \in V$.

Multi pairs shortest path problem (MPSP). Let $S, T \subseteq V$ and $S \cap T = 0$. Compute $\inf\{d(s,t) : s \in S, t \in T\}$; i.e. find the path with minimum length from the set S to the set T (respectively from any node $s \in S$ to any node $t \in T$) or find the shortest S, T-path.

All pairs shortest path problem (APSP). For all nodes $u \in V$ compute $d(u,v)$ for all $v \in V$; i.e. find pathes with minimum lengths in G with respect to all node pairs in the graph or find the pairwise shortest paths.

In the following we will consider the problems SSSP and APSP. Obviously, solving a SSSP will automatically yield also the solution of the according SPSP. A MPSP can be transformed to a SPSP as follows. Augment the graph G with two artificial nodes s' and t'. Insert new arcs such that s' is a predecessor of all nodes in S, $N_G^+(s') = S$, and t' is a successor of all nodes in T, $N_G^-(t') = T$. All new arcs have zero costs. Thus a shortest s', t'-path in the augmented graph implies also a shortest S, T-path in the original graph.

2.7.3 Basic observations for shortest pathes

Lemma 2.7.2
Let P be the shortest u, v-path in G determining $d(u,v)$ and $w \in V(P)$; i.e.

$$P = (u = v_0, e_1, v_1, \ldots, e_m, v_m = w, e_{m+1}, v_{m+1}, \ldots, e_k, v_k = v).$$

Then the path

$$P_{u,w} = (u = v_0, e_1, v_1, \ldots, e_m, v_m = w)$$

is also the shortest u, w-path and the path

$$P_{w,v} = (v_m = w, e_{m+1}, v_{m+1}, \ldots, e_k, v_k = v)$$

is also the shortest w, v-path.

Proof 2.7.2
Let us assume that $P_{u,w}$ is not the shortest u, w-path - there exists another u, w-path $P'_{u,w}$ with $c\left(P'_{u,w}\right) < c\left(P_{u,w}\right)$. This implies that

$$c\left(P'_{u,w} \cup P_{w,v}\right) < c(P)$$

and since $P'_{u,w} \cup P_{w,v}$ is a u, v-path this would be a contradiction to the fact that P is the shortest u, v-path. We may prove the second statemant in the same way. □

Lemma 2.7.3
Let $s \in V$. Then the inequality $d(s,v) \leq d(s,u) + c(e)$ holds for all $e \in E$ with $\alpha(e) = u$ and $\omega(e) = v$.

Proof 2.7.3
Let P be the shortest s, v-path determining $d(s,v)$ and consider any edge e with $\alpha(e) = u$ and $\omega(e) = v$. Let us consider the case $u \in V(P)$. Then lemma 2.7.2 implies that

$$d(s,v) = d(s,u) + \underbrace{d(u,v)}_{\leq c(e)} \leq d(s,u) + c(e).$$

In the case that $u \notin V(P)$ the assumption $d(s,v) > d(s,u) + c(e)$ implies that a s,v-path can be found which has a lower length than P. This contradiction shows that the assumption is false. $\square$

Definition 2.7.1
We consider a directed graph G with no parallel arcs. Then a rooted tree $T = (V', E') \subseteq G$ with root $s \in V$ is called a shortest path tree with respect to s if the following two conditions hold:

1. $V' = R(s)$.
2. For any $v \in V' \setminus \{s\}$ the unique s,v-path in T is also a shortest s,v-path in G.

The existence of such a tree can be evaluated by the following theorem:

Theorem 2.7.1
We consider a directed graph G with no parallel arcs. If $s \in V$ is a node such that each cycle which can be reached from s in G has no negative length, then there exists a shortest path tree with respect to s.

Proof 2.7.4
Let $V' = R(s)$ and $E' \subseteq E$ is the smallest set of arcs such that all shortest s,v-pathes for $v \in V'$ do exclusively use arcs in E'. Then we claim that $T' = (V', E')$ is a shortest path tree with respect to s. Thus we have only to show that T' is a rooted tree with root s. Due to theorem 2.4.3 (page 20) this means to show that $g^-_{T'}(s) = 0$ and $g^-_{T'}(v) \leq 1$ for all $v \in V'$, $v \neq s$.
Obviously, if and arc e with $\alpha(e) = v$ and $\omega(e) = s$ exists in E' for any $v \in V'$, then such an arc would not be in any shortest s,v-path - thus negleting it would yield a smaller set E' - a contradiction. In the same way we can exclude the existence of arcs e and f in E' such that $\omega(e) = \omega(f) = v$ for any $v \in V'$. $\square$

Definition 2.7.2
Let $p : V \to \mathbb{R}$ be a mapping on the nodes of a graph. Then the function

$c_p : E \to \mathbb{R}$, with

$c_p(e) := c(e) + p(u) - p(v)$ for all $e \in E$, whereby $\alpha(e) = u$ and $\omega(e) = v$,

is denoted as reduced cost function and $c_p(e)$ are the reduced costs of the arc e. The function p is denoted as a potential, if the condition $c_p(e) \geq 0$ holds for all $e \in E$, which means that the following inequalities have to hold:

$$p(v) \leq p(u) + c(e) \text{ for all } e \in E, \text{ whereby } \alpha(e) = u \text{ and } \omega(e) = v. \qquad (*)$$

Let us investigate the reduced costs in more detail. Therefore we consider a path $P = (v_0, e_1, v_1, e_2 \ldots, v_{k-1}, e_k, v_k)$ in G and compute

$$\begin{aligned} c_p(P) &= \sum_{e \in E(P)} c_p(e) = \sum_{i=0}^{k-1} c(e_i) + p(v_i) - p(v_{i+1}) = \qquad (**) \\ &= \sum_{i=0}^{k-1} c(e_i) + \sum_{i=0}^{k-1} p(v_i) - p(v_{i+1}) = c(P) + p(v_0) - p(v_k) \end{aligned}$$

Thus we see that the length of a path P from v_0 to v_k differs by a constant value (exclusively depending on the start- and the end node of the path) if we take the costs c compared to the reduced costs c_p. Thus, searching for shortest path in G with respect to the costs c yields the same solution as for the reduced costs c_p. Thus we formulate the following lemma:

Lemma 2.7.4
Let $p : V \to \mathbb{R}$ be a mapping on the nodes of a graph. Then the following statements hold:

1. P is the shortest u, v-path in G with respect to the costs c, if and only if P is also the shortest path with respect to the reduced costs c_p.
2. The equality
$$c\,(P) = c_p\,(P) + p\,(\omega\,(P)) - p\,(\alpha\,(P))$$
holds.
3. For each cycle the equality $c\,(P) = c_p\,(P)$ holds.

Moreover we formulate the following theorem:

Theorem 2.7.2
We consider a graph directed graph G with no parallel arcs. G contains a potential $p : V \to \mathbb{R}$ if and only if G does not contain any cycle of negative length.

Proof 2.7.5
"$\Rightarrow$ " : Let us assume that p is a potential. This implies that $c_p\,(e) \geq 0$ for all $e \in E$ and therefore $c_p\,(C) \geq 0$ for all cycles C in G. Due to lemma 2.7.4(3) $c_p\,(C) = c\,(C) \geq 0$ holds.
"$\Leftarrow$ " : Let us assume that G contains no cycle of negative length. Due to lemma 2.7.1 we may argue that $-\infty < d\,(u, v)$ for all node pairs u and v in G; but in general $d\,(u, v) < \infty$ does not hold for all pairs, e.g. if $v \notin R\,(u)$. Thus we augment G by an artificial node s' such that $N^+\,(s') = V$ and $c\,(e) = 0$ for all e with $\alpha\,(e) = s'$. Let us call the augmented graph as G'. Since s' can not be contained within any cycle and since G does not contain any cycle of negative length, also G' does not contain any - thus $-\infty < d\,(u, v)$ also holds in G' for all node pairs u and v. Since $R\,(s') = V$ in G' also $d\,(s', v) < \infty$ holds in G' and therefore also $-\infty < d\,(s', v) < \infty$ for all nodes v in G'. Thus

$$p\,(v) := d\left(s', v\right) \overset{\text{lemma 2.7.3}}{\leq} \underbrace{d\left(s', u\right)}_{:=p(u)} + c\,(e) \quad \text{with } e \in E : \alpha\,(e) = u, \omega\,(e) = v$$

Thus according to (*) in definition 2.7.2 p is a potential in G' and therefore also in G. $\square$

2.7.4 A skeletal structure for the SSSP

We will give an iterative procedure for computing the distances $d\,(s, v)$ for a specified $s \in V$ to all $v \in V$. In this context we will use a predecessor information $\pi : V \to V$ whereby $\pi\,(v) = u$ means that the distance $d\,(s, v)$ depends on the distance $d\,(s, u)$. Formaly we store this information in the predecessor graph $G_\pi := (V_\pi, E_\pi)$ with

$$V_\pi := \{v \in V : \pi\,(v) \neq NaN\} \cup \{s\}$$

$$E_\pi := \{e \in E : \alpha(e) = \pi(v) \text{ and } \omega(e) = v\}$$

Moreover the quantities $d(v)$ are upper bounds for the distances $d(s,v)$ for all $v \in V$. Our procedure may be initialized as follows:

Algorithm 1: `SSSPInit` (initializes the single source shortest path problem)

```
Data: A directed graph $G = (V, E, \alpha, \omega)$ without parallel arcs and arc costs $c : E \to \mathbb{R}_+$.
Result: Initialized markers $d$ and $\pi$.
1 begin
2     for $v \in V$ do
3         $d(v) = +\infty$
4         $\pi(v) = 0$
5     $d(s) = 0$
```

Moreover we will use the following module `CHECK` in order to investigate whether the upper bound $d(v)$ may be decreased when we use an arc e with $\alpha(e) = u$ and $\omega(e) = v$.

Algorithm 2: `CHECK` (checks an arc if it is in the shortest path tree)

```
Data: An arc $e \in E$ such that $\alpha(e) = u$ and $\omega(e) = v$ and marker values $d(u), d(v)$.
Result: Updated markers $d$ and $\pi$ for $u$.
1 begin
2     if $d(v) \geq d(u) + c(e)$ then
3         Update $d(v) \leftarrow d(u) + c(e)$
4         Update $\pi(v) = u$
```

On the basis of these two modules we formulate the following theorem:

Theorem 2.7.3
We consider a directed graph G with no parallel arcs and a specified node $s \in V$. After the initialization with algorithm `SSSP_Init` and an arbitrary number of iterations with algorithm `CHECK` the following conditions hold:

1. The reduced costs $c_d(e)$ with respect to the upper bounds d hold:
$$c_d(e) := c(e) + d(u) - d(v) \leq 0 \text{ for all } e \in E_\pi \text{ whereby } \alpha(e) = u \text{ and } \omega(e) = v.$$
2. The quantities d remain upper bounds, i.e. $d(v) \geq d_c(s,v)$ for all $v \in V$.
3. Each cycle in G_π has negative length with respect to the reduced costs c_d and the costs c.
4. If in G no cycle of negative length is reachable from s, then
   (a) $G_\pi = (V_\pi, E_\pi)$ is a rooted tree with root s.
   (b) In G_π each (unique) path P from s to $v \in V_\pi$ has at most the length $d(v)$. Notice that the unique path P is given by the recursion
$$P = \left(s := \pi^k(v), e_k, \pi^{k-1}(v), e_{k-1}, \pi^{k-2}(v), \ldots, \pi^2(v), e_2, \pi^1(v), e_1, \pi^0(v) := v\right)$$

whereby

$$\pi^{i}(v) := \pi\left(\pi^{i-1}(v)\right) \text{ for } i > 0 \text{ and } \pi^{i-1}(v) \neq NaN.$$

Proof 2.7.6
We show the theorem by induction. Initialy, after the run of the algorithm `SSSP_Init`, the theorem is trivially fulfilled. Let us assume that the theorem is also met after several runs of the routine `CHECK`. Then we have to show, that a current run do not violate any of the four statements. If $d(v) \leq d(u) + c(e)$ with $\alpha(e) = u$ and $\omega(e) = v$ then nothing happens in the current run; thus we assume that $d(v) > d(u) + c(e)$ for the following investigations.
Ad 1.
After the update $d(v) := d(u) + c(e)$ we see that $c_d(e) = c(e) + d(u) - d(v) = 0$ for the arc e. Let us study an arc f with $\alpha(f) = v$ and $\omega(f) = w$. Before the current run $c_d(f) \leq 0$ holds due to assumtion. Since $c_d(f) = c(f) + d(v) - d(w)$ and $d(v)$ will be decreased in the current run, the inequality also holds after the run. Other arcs will not be affected by the current run. We have chosen e arbitrarily in E and E_π is a subset of E.
Ad 2.
The following inequalities hold:

$$d(v) = d(u) + c(e) \geq d(s,u) + c(e) \overset{\text{lemma 2.7.3}}{\geq} d(s,v)$$

Ad 3.
Since $c_d(e) \leq 0$ holds for all $e \in E_\pi$ each cycle C has a negative length with respect to the reduced costs. Due to lemma 2.7.4 the costs of a cycle are equal to the reduced costs.
Ad 4.
First we want to show that the recursion in G_π allways stops at the node s. Therefore let us assume that this case not happens; i.e. the recursion produces a cycle

$$C = \left(\ldots, v_j = \pi^{j+l}(v), e_{j+l}, \pi^{j+l-1}(v), \ldots, e_{j+1}, \pi^{j}(v) := v_j, e_j, \ldots, , \pi^{1}(v), e_1, \pi^{0}(v) := v\right)$$

in G_π. Due to point 3 the cycle has negative length. A node u in the cycle C is obviously also a node in V_π - in order to get to $V_{\pi i}$ the node u has a predessor $\pi(u)$ which also implies that $d(u) < \infty$ and therefore also $d(s,u) \leq d(u) < \infty$ holds. Thus we have found a node which is within a cycle of negative length and which is also reachable from s - this is a contradiction and therefore our assumption has to be false.
Due to lemma 2.7.4 we have

$$c(P) = \underbrace{c_d(P)}_{\leq 0} + d(v) - d(s) \leq d(v) - \underbrace{d(s)}_{\geq d(s,s)} \leq d(v) - \underbrace{d(s,s)}_{=0} = d(v)$$

Finaly we have to show that G_π is a rooted tree with root s. Due to theorem 2.4.3 (page 20) we have to show that $R_{G_\pi}(s) = V_\pi$, $g^-_{G_\pi}(s) = 0$ and $g^-_{G_\pi}(v) \leq 1$ for all $v \in V_\pi$. We have shown how to recursively construct a path P from s to all $v \in V_\pi$; thus clearly $R_{G_\pi}(s) = V_\pi$ holds in G_π. Since each node v has exactly one predecessor $\pi(v)$ in G_π the condition $g^-_{G_\pi}(v) = 1$ holds. The assumption $g^-_{G_\pi}(s) \geq 0$ yield a cycle in G_π which must be of negative length (due to point 3) - this is a contradiction to the fact that no such cycle is reachable from s. $\square$

Corollary 2.7.1
Thus if no cycle of negative length is reachable from s and if we could construct an algorithm which initializes with the sub-routine `SSSP_Init` and which iterates, whereby at the end of each iteration the sub-routine `CHECK` will be performed, such that at the end of all iterations $d(v) \leq d(s,v)$ hold for all $v \in V$, then we can argue that

- $d(v) = d(s,v)$ for all $v \in V$.
- G_π is a shortest path tree with root s.

2.7.5 Solving the SSSP for non-negative arc costs - Dijkstra's algorithm

Let us consider directed graphs with no parallel edges and non-negative arc costs; i.e. $c : E \to \mathbb{R}_+ \cup \{0\}$. Moreover let $V^* \subseteq V$ be the subset of nodes such that for each $v \in V^*$ the condition $d(v) \leq d(s,v)$ already holds. Thus if $V = V^*$ at the end of an algorithm we have found the shortest path tree (due to the observation above). Let W be the complementary set of V^* in V. Then we formulate the following algorithm:

Algorithm 3: `SSSPDijkstra` (solves the SSSP for non-negative arc costs)

```
Data: A directed Graph $G = (V, E, \alpha, \omega)$ with a cost function $c : E \to \mathbb{R}_+$.
Result: Shortest path tree $G_\pi = (V_\pi, E_\pi)$.
begin
    Call algorithm SSSPInit
    Initialize $W \leftarrow V$
    while $W \neq \{\}$ do
        Determine $u = \arg\min \{d(v) : v \in W\}$
        Update $W \leftarrow W - \{u\}$
        for $v \in Adj(u) \cap W$ do
            Call algorithm CHECK
```

Theorem 2.7.4
At the iteration stop of the algorithm `SSSPDijkstra` the following situation holds:

1. $d(v) \leq d(s,v)$ for all $v \in V$.
2. G_π is a shortes path tree for the source $s \in V$.
3. The complexity of `SSSPDijkstra` is $O\left(|V|^2\right)$.

Proof 2.7.7
We observe that the termination criterion is feasible - $W = \{\}$ implies that $V^* = V$.
Ad 1) We have to show, that whenever a node u will be eliminated from W (which means that u changes to the set V^*), the inequality $d(u) \leq d(s,u)$ holds (notice: due to the positve arc costs we do not have any cycles of negtive lengths). We show this by induction with respect to the number of iterations. In the first iteration $u = s$ will be eliminated from W; obviously $d(s) := 0 \leq d(s,s) = 0$ holds. Now study the situation when we eliminate u in the current

iteration; let assume that the inequality $d(v) \leq d(s,v)$ holds for all v which have been eliminated in all iterations before; but now $d(u) > d(s,u)$ is assumed for the current iteration. Let P be the shortest path from s to u, formaly:

$$P = (s = v_0, v_1, \dots v_{i-1}, v_i, \dots, v_{k-1}, v_k = u) \text{ and } c(P) = d(s,u).$$

Choose the minimal $i \in \{1, 2, \dots, k\}$ such that $v_i \in W$; notice that $i > 0$ holds, since $v_0 = s$ has been already eliminated from W. Thus $v_{i-1} \in V^*$ holds and therefore $d(v_{i-1}) \leq d(s, v_{i-1})$ due to the assumption. After processing the sub-routing `CHECK` in that iteration, when v_{i-1} has been eliminated from W, we have the following situation:

$$d(v_i) \leq d(v_{i-1}) + c(e) \text{ with } \alpha(e) = v_{i-1} \text{ and } \omega(e) = v_i$$

$$d(v_i) \leq d(v_{i-1}) + c(e) \leq d(s, v_{i-1}) + c(e) = d(s, v_i)$$

This means for the current iteration that:

$$d(v_i) \leq d(s, v_i) \leq d(s,u) \overset{(*)}{<} d(u) \qquad (*) \dots \text{ due to assumption!}$$

Since u should be the node in W with the smallest d-value in the current iteration, this is a contratiction - thus also $d(u) \leq d(s,u)$ holds.
Ad 2) See corollary 2.7.1 on page 46.
Ad 3) In the step 5 we have to do $|W| - 1$ comparisions in order to find the correct u. Since $|W| \leq |V|$ and we have to perform this step at most $|V|$ times we get an overall complexity for the step 5 of $|V|^2$. In all calls of the step 7 we consider each arc of the graph once only; this is easy to see: since u will be eliminated from W in an iteration (and never been inserted in W anymore) we will never consider the arcs starting from this eliminated node.Thus we get an overall complexity of

$$|V|^2 + |E| \leq |V|^2 + |V|^2 = 2|V|^2 = O\left(|V|^2\right).$$

This competes the proof. $\square$

Remark 2.7.1
The algorithm `SSSPDijkstra` is applicable to undirected graphs G without any modification if we consider for each edge $e \in E$ with $\gamma(e) = \{u, v\}$ two arcs f with $\alpha(f) = u$, $\omega(f) = v$ and g with $\alpha(g) = v$, $\omega(g) = u$, both with costs $c(e)$.

Laboratory 2.7.2
Create a directed graph and compute the shortest path tree for any source node and interpret the result. For reference you find the routine `ssspuree.m` in the folder `...\Routines\Exercises`.

Laboratory 2.7.3
In the course directory `...\Routines\Exercises` you find the file `46305_Klagenfurt_Mitte.ist` - graph instance file in the RTR_R2008a format, see section 6.1.3. The graphs represents the spatial topology of the city-net domain Klagenfurt-Mitte.

- Realize a routine which allows the computation of the shortest path (with respect to the coded edge costs or the Euclidean edge lengths) between two arbitrarily chosen positions in this city-net domain (SPSP problem).

- Realize a routine which allows the computation of the shortest path (with respect to the coded edge costs or the Euclidean edge lengths) between any node with an easting position lower than 500 meters and any node with an easting position greater than 2500 meters.

For reference you find the routine `navi.m` in the folder `...\Routines\Exercises`.

There exists various techniques for implementing the algorithm `SSSPDijkstra`. Most of them use a priority queue in the following sens: we do not store the node set W explicitely, but rather a queue Q of detected nodes in W. Thereby a node u is queued before a node v if and only if $d(u) \leq d(v)$ holds. Thus we may formultate the following algorithm:

Algorithm 4: `SSSPDijkstraQ` (solves the SSSP for non-negative arc costs)

```
Data: A directed Graph $G = (V, E, \alpha, \omega)$ with a cost function $c : E \to \mathbb{R}_+$.
Result: Shortest path tree $G_\pi = (V_\pi, E_\pi)$.
1  begin
2      Call algorithm SSSPInit
3      Initialize $Q \leftarrow [s]$
4      while $Q \neq []$ do
5          $u = Q[1]$ and update $Q \leftarrow Q[2, 3, \ldots, end]$
6          for $v \in Adj(u)$ do
7              Initialize $\bar{d} = d(v)$
8              if $d(v) = +\infty$ then
9                  Update $Q[end + 1] = v$
10             Call algorithm CHECK
11             if $d(v) < \bar{d}$ then
12                 Relocate $v$ to the $k$-th position in the queue $Q$ such that the condition
                   $d(Q[k-1]) \leq d(Q[k] = v) \leq d(Q[k+1])$ holds.
```

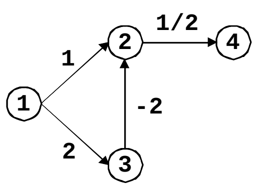
Remark 2.7.2
The algorithm `SSSPDijkstraQ` obviously corresponds to the algorithm `SSSPDijkstra`, but with one difference: since we do not consider the node set W explicitely, we call the routine `CHECK` for all adjacent nodes $v \in Adj(u)$; thus also for $v \in V^*$ - compare to step 7 in `SSSPDijkstra`. But whenever $v \in V^*$ then we have already found the optimal path from s to v; thus in this case the value $d(u) + c(e)$, $\alpha(e) = u$ and $\omega(e) = v$, would never be lower than $d(v)$ and therefore the call of `CHECK` woud not have any affect.

Remark 2.7.3
The complexity of `SSSPDijkstraQ` depends on the implementation technique of the queue Q; e.g. using a Fibonacci-Heap we get the complexity $O(|E| + |V| \log |V|)$.

2.7.6 The SSSP problem negative arc costs

In general the algorithm `SSSPDijkstra` fails in cases of negative arc costs:

Laboratory 2.7.4
Show that the algorithms `SSSPDijkstra` and `SSSPDijkstraQ` fail if we compute the shortest path tree for the source node $s = 1$ in the graph shown below.



Algorithm 5: `SSSPBellmanFord` (solves the SSSP for arbitrary arc costs)

```
    Data: A directed Graph $G = (V, E, \alpha, \omega)$ with a cost function $c : E \to \mathbb{R}$.
    Result: Shortest path tree $G_\pi = (V_\pi, E_\pi)$.
1   begin
2       Call algorithm SSSPInit
3       for $k = 1, 2, \ldots, | V |$ do
4           for $e \in E$ do
5               Let $u = \alpha(e)$ and $v = \omega(e)$
6               if $v \neq s$ then
7                   Call algorithm CHECK
```

Remark 2.7.4
Obviously $d(s) = 0$ and $p(s) = s$ is required. Notice that the if-condition in the routine `CHECK` would never been met for any arc $e \in E$ with $u = \alpha(e)$ and $s = \omega(e)$ in case of positive arc weights. In the considered case of negative arc weights, the additional if-condition in step 6 guarantees that the initial requirement will never been violated.

Lemma 2.7.5
The following condition holds at the end of the k-th run of the step 2 of the algorithm `SSSPBellmanFord`:

$$d(v) \leq \inf\{c(P) : P \text{ is a } s, v\text{-path}, \ |E(P)| \leq k\} \tag{*}$$

Proof 2.7.8
For all $v \notin R(s)$ the condition $d(v) \leq \inf\{\} = +\infty$ trivialy holds. For all nodes with $v \in R(s)$ we show this lemma by induction w.r.t. k. For $k = 0$, once again the condition $d(v) \leq \inf\{\} = +\infty$ trivialy holds. Let us assume that the condition holds in the case $k - 1$, i.e.

$$d(v) \leq \inf\{c(P) : P \text{ is a } s, v\text{-path}, \ |E(P)| \leq k - 1\} \tag{**}$$

after the $k - 1$-th iteration of step 2 and show the correctness of (*) after the k-th iteration. It is sufficient to show that

$$d(v) \leq \inf\{c(P) : P \text{ is a } s, v\text{-path}, \ |E(P)| = k\}$$

holds; notice: all paths with less than k arcs meet the criterion (**) after the $k-1$-th run and $d(v)$ would never been increased in the k-th run due to the construction of `CHECK`. Thus we consider an arbitrary path P from s to v with k arcs:

$$P = (s = v_0, e_0, v_1, e_1, v_{2,} \dots, e_{k-2}, v_{k-1}, e_{k-1,} v_k = v)$$

Then the path

$$W = (s = v_0, e_0, v_1, e_1, v_{2,} \dots, e_{k-2}, v_{k-1}) \subset P$$

is a s, v_{k-1}-path with $k-1$ arcs and due to (**) the condition $d(v_{k-1}) \leq c(W)$ holds after the $k-1$-th iteration. After the call of `CHECK` in the k-th iteration it follows that

$$d(v_k) \leq d(v_{k-1}) + c(e) \text{ with } \alpha(e) = v_{k-1} \text{ and } \omega(e) = v_k$$

and therefore

$$d(v) = d(v_k) \leq c(W) + c(e) = c(P)$$

This completes the proof. $\square$

Theorem 2.7.5
If G does not contain any cycle of negative length, which is reachable from s, then after the termination of the algorithm `SSSPBellmanFord` the following statements are true:

1. $d(v) = d(s, v)$ for all $v \in V$.
2. G_π is a shortest path tree with respect to the source node s.

The complexity of `SSSPBellmanFord` is $O(|V| |E|)$.

Proof 2.7.9
Due to corollary 2.7.1 it is sufficient to show that $d(v) \leq d(s, v)$ for all $v \in V$. For all $v \notin R(s)$ the condition $d(v) \leq d(s, v) = +\infty$ trivially holds. Let us consider a node $v \in R(s)$ and let

$$P = (s = v_0, e_0, v_1, \dots, e_{k-1}, v_k = v)$$

be a shortest s, v-path with k arcs; i.e. $c(P) = d(s, v)$. We may argue that P does not contain any cycle; if a cycle would exist in P we may delete this cycle (with non-negative costs) and get a new s, v-path with lower costs which contradicts to $c(P) = d(s, v)$. Thus P is an elementary path with k-arcs and thererfore $|V(P)| = k + 1 \leq |V|$ or $k \leq |V| - 1$ holds. Due to lemma 2.7.5 on page 49 we may argue that after the k-th iteration of step 2 the algorithm `SSSPBellmanFord` yiels a marker $d(v) \leq c(P) = d(s, v)$. The estimate of the complexity is obvious. $\square$

Laboratory 2.7.5
Show that the algorithm `SSSPBellmanFord` works correctly for the problem in laboratory 2.7.4 on page 49.

We have seen that the algorithm `SSSPBellmanFord` is able to deal with negative arc costs as long as we do not have any cycle of negative length which is reachable from the source node $s \in V$. In the case that such a cycle of negative length is reachable from s we have the situation that $d(s, v) = -\infty$ for all $v \in R(s)$. Thus we need an oracle which is able to check the existence of such a cycle - such a tool is provided by the following algorithm `CHECKNegCycle`. Notice the algorithm uses the output of the algorithm `SSSPBellmanFord`. Thus if we accecpt or reject the output of `SSSPBellmanFord` depending on the result of `CHECKNegCycle`.

Algorithm 6: `CHECKNegCycle` (checks if a cycle of negative length is reachable from a node $s \in V$)

Data: A directed Graph $G = (V, E, \alpha, \omega)$ with a cost function $c : E \to \mathbb{R}$ and distance markers $d(v)$ for all $v \in V$ at the end of algorithm `SSSPBellmanFord`.

Result: A boolean decission variable $C \in \{true, false\}$ whereby $C = true$ means that such a cycle exits.

```
1 begin
2     Initialize $C \leftarrow false$
3     for $e \in E$ do
4         Let $u = \alpha(e)$ and $v = \omega(e)$
5         if $d(v) > d(u) + c(e)$ then
6             Update $C \leftarrow true$ and terminate the algorithm
```

Theorem 2.7.6
The algorithm `CHECKNegCycle` checks (after the termination of the algorithm `SSSPBellmanFord`) correctly the existence of a cycle of negative length which is reachable from $s \in V$. The complexity of `CHECKNegCycle` is $O(|E|)$; thus the overall complexity of `SSSPBellmanFord` and `CHECKNegCycle` is $O(|V|\,|E| + |E|) = O(|V|\,|E|)$.

Proof 2.7.10
We show the following equivalence:

$$\texttt{CHECKNegCycle} \text{ yields } C = false. \Leftrightarrow \text{ No cycle of negative length is reachable from } s \in V.$$

"$\Leftarrow$".
If no cycle of negative length, which is reachabel from s, exists we may argue that $d(v) = d(s, v)$ for all $v \in V$ due to theorem 2.7.5 on page 50. Lemma 2.7.3 on page 41 tells us that

$$d(s, v) \leq d(s, u) + c(e) \quad \text{for } \alpha(e) = u \text{ and } \omega(e) = v$$

Thus for all arcs e with $\alpha(e) = u$ and $\omega(e) = v$ the condition $d(v) \leq d(u) + c(e)$ holds after the termination of `SSSPBellmanFord` and `CHECKNegCycle` will yield $C = false$.
"$\Rightarrow$".
Let us assume that `CHECKNegCycle` yields $C = false$. At the end of the algorithm `SSSPBellmanFord` we get $d(v) < \infty$ for all $v \in R(s)$ due to lemma 2.7.5 on page 49. Therefore the subgraph $G[U]$ with $U = \{v \in V : d(v) < \infty\}$ conatains all nodes which are reachable from s; so it is sufficient to show, that $G[U]$ does not contain any cycle of negative length. Since `CHECKNegCycle` yields $C = false$ we have the situation that $d(v) \leq d(u) + c(e)$ for all arcs $e \in E$ with $\alpha(e) = u$ and $\omega(e) = v$. Thus due to definition 2.7.2 on page 42 we see that d is a potential in $G[U]$; theorem 2.7.2 on page 43 tells us that the existence of a potential is sufficient for the non-existence of a cycle of negative length. □

For several applications it is also of interest to determine a cycle of negative length if `CHECKNegCycle` has detected the existence of such a cycle. Thus we assume that after the termination of `SSSPBellmanFord` an arc $e \in E$, $\alpha(e) = u$ and $\omega(e) = v$, exists such that

$$d(v) > d(u) + c(e). \tag{*}$$

Then we start at u and construct the path

$$P = (s = v_i, v_{i-1}, \ldots, v_2, v_1, v_0 =: u) \text{ with } v_k := \pi(v_{k-1})$$

Notice that P is always a s,u-path since G_π is a rooted tree with root s. Then two cases may happen:

1. In the construction process we detect that $v_j = v_{j-k}$ for some $k > 0$; i.e. we detect a node which is already within the path. Thus $C = \{v_j, \ldots, v_{j-k}\}$ is a cycle in G_π. Due to theorem 2.7.3 on page 44, point 3, we may argue that $c(C) < 0$. Thus C is a cycle of negative length which is reachable from s.

2. P is elementary. In this case we claim that $v \in V(P)$, i.e. $v_j = v$ for any $j : 0 \leq j \leq i$. In order to show this we assume the contrary; i.e. that v is not in P. Then

$$P' = (s = v_i, \ldots, v_2, v_1, v_0 =: u, v)$$

is an elementary path from s to v and therefore consists of at most $|V| - 1$ arcs. We consider the reduced costs

$$c_d(f) := c(f) + d(v_i) - d(v_{i-1}) \text{ whereby } \alpha(f) = v_i \text{ and } \omega(f) = v_{i-1}$$

and we know from theorem 2.7.3, page 44, point 1, that $c_d(f) \leq 0$ for all $f \in E_\pi$; therefore

$$c_d(P') = \underbrace{c_d((s, \ldots, v_2, v_1, u))}_{\leq 0} + c_d(e) \leq c_d(e) \text{ for } \alpha(e) = u \text{ and } \omega(e) = v.$$

Due to (**) on page 42 we see that

$$c_d(P') = c(P') + d(s) - d(v)$$

or

$$c(P') = c_d(P') + d(v) - d(s) = c_d(P') + d(v) \quad (\text{since } d(s) = 0).$$

$$c(P') \leq d(v) + c_d(e) = d(v) + (c(e) + d(u) - d(v)) \text{ for } \alpha(e) = u \text{ and } \omega(e) = v.$$

$$c(P') \leq d(u) + c(e) < d(v) \text{ due to (*) on page 51.}$$

Thus we see that P' is a path from s to v with at most $|V| - 1$ and $c(P') < d(v)$ - this is a contradiction to lemma 2.7.5 on page 49.
Thus we may concretize the path P as follows:

$$P = (s = v_i, \ldots, v_j = v, v_{j-1}, \ldots, v_2, v_1, v_0 =: u) \quad \text{with } i > j \text{ and } v_k := \pi(v_{k-1})$$

and see immediately that

$$C = (v_j = v, v_{j-1}, \ldots, v_2, v_1, v_0 = u, v)$$

is a cycle which is reachable from s. Since $c_d(f) \leq 0$ with $\alpha(f) = v_i$ and $\omega(f) = v_{i-1}$ and $c_d(e) = c(e) + d(u) - d(v) < 0$ due to (*) on page 51 we see that $c_d(C) < 0$. Due to (**) on page 42 also $c(C) < 0$ holds.

Algorithm 7: GETNegCycle (yields a cycle of negative length)

Data: A directed graph $G = (V, E, \alpha, \omega)$ with arc costs $c : E \to \mathbb{R}$. Distance markers $d(v)$ for all $v \in V$ at the end of the algorithm SSSPBellmanFord.

Result: A cycle of negative length $C = (C_1, C_2, \ldots, C_k)$ which is reachable from $s \in V$ or the information that no such cycle exists.

```
1  begin
2      Initialize $C = ()$
3      for $e \in E$ do
4          Let $u = \alpha(e)$ and $v = \omega(e)$
5          if $d(v) > d(u) + c(e)$ then
6              Initialize $w = u$, $C = (w)$ and $k = 1$
7              while $w \neq s$ do
8                  Update $w \leftarrow \pi(w)$
9                  if $C_j = w$ for any $j = 1, \ldots, k$ then
10                     Update $C \leftarrow (w, C_1, C_2, \ldots, C_{j-1}, C_j)$ and terminate the algorithm
11                 else
12                     Update $k \leftarrow k + 1$
13                     for $j = k, k-1, \ldots, 2$ do
14                         Update $C_j \leftarrow C_{j-1}$
15                     $C_1 = w$
16             Find an index $j$ such that $C_j = v$
17             Update $C \leftarrow (C_j, C_{j+1}, \ldots, C_k, v)$ and terminate the algorithm.
```

Theorem 2.7.7
The algorithm `GETNegCycle` determines (after the termination of the algorithm `SSSPBellmanFord`) correctly a cycle of negative length which is reachable from $s \in V$ or yield the information that no such cycle exists. The complexity of `GETNegCycle` is $O(|E|)$; thus the overall complexity of `SSSPBellmanFord` and `GETNegCycle` is $O(|V||E| + |E|) = O(|V||E|)$.

Remark 2.7.5
The algorithm `SSSPBellmanFord` is applicable to undirected graphs G without any modification if we consider for each edge $e \in E$ with $\gamma(e) = \{u, v\}$ two arcs f with $\alpha(f) = u$, $\omega(f) = v$ and g with $\alpha(g) = v$, $\omega(g) = u$, both with costs $c(e)$.

Laboratory 2.7.6
In the folder `...\Routines\Exercises` you will find the instance file `ssspdigraph01.ist` storing a directed graph in the basic instance format described in section 6.1.1 on page 81. Study the situation for the SSSP with $s = 17$; delete an arbitrary chosen arc from a detected cycle of negative length as long as no such a cycle can be detected anymore. Specify the shortest path tree for the accordingly modified graph. For reference you find the routine `ssspnegcosts.m` in the folder `...\Routines\Exercises`.

Remark 2.7.6
We have seen that we can find a cycle of negative length which is reachable fom a specified node

$s \in V$ or decide that no such cycle exists. In some applications (considering network flows) it is neccessary to decide independently from any specified root node whether a graph does contain any cycle of negative length or not. Thus we introduce a new node s, i.e. $V \leftarrow V + \{s\}$ and new arcs $e : \alpha(e) = s$ and $\omega(e) = v$ for all $V - \{s\}$ with costs $c(e) = 0$. Running `GETNegCycle` with root node s yields the intended answer.

2.7.7 The APSP problem

We consider a graph $G = (V, E, \alpha, \omega)$ with arc costs $c : E \to \mathbb{R}$. Thus we allow negative arc costs, but exclude the existance of any cycle of negative length in G; we can check this condition with complexity $O(|V|\,|E|)$ as discussed in the section above. Obviously we can call `SSSPBellmanFord` for all nodes in order to solve the APSP problem - this would yield a complexity $O(|V|^2\,|E|)$ and with $|E| = O(|V|^2)$ the worst case ratio $O(|V|^4)$. However a more efficient way for solving this problem can be found:

Let us assume that the nodes in V are numbered; i.e. $V = \{v_1, v_2, \ldots, v_n\}$ for $|V| = n$. Moreover, for $k = 1, 2, \ldots, n$ the measure $d_k(v_i, v_j)$ are the costs of a shortest v_i, v_j-path in the induced subgraph $G[\{v_i, v_j\} \cup \{v_1, v_2, \ldots, v_k\}]$. We observe that:

- For $k = 0$ it follows that:
$$d_0(v_i, v_j) = \begin{cases} c(e) & \text{if } e \in E : \alpha(e) = v_i \text{ and } \omega(e) = v_j \\ +\infty & \text{otherwise} \end{cases}$$
- For $k = n$ it follows that $d_n(v_i, v_j) = d(v_i, v_j)$; i.e. $d_n(v_i, v_j)$ equal to the costs of the shortest v_i, v_j-path in G. Thus we are interested to find a recursion F such that:
$$d_{k+1}(v_i, v_j) = F(d_k(v_i, v_j)) \text{ for all pairs } (v_i, v_j) \in V \times V \text{ and for } k = 0, 1, \ldots, n-1.$$

Let us investigate this measure in more detail - therefore we assume that P is the shortest v_i, v_j-path in the induced subgraph $G[\{v_i, v_j\} \cup \{v_1, v_2, \ldots, v_k, v_{k+1}\}]$ and therefore $c(P) = d_{k+1}(v_i, v_j)$. Then we can consider two cases:

1. $v_{k+1} \notin V(P)$.
   This means that P is also a shortest v_i, v_j-path in $G[\{v_i, v_j\} \cup \{v_1, v_2, \ldots, v_k\}]$ and therefore
$$c(P) = d_{k+1}(v_i, v_j) = d_k(v_i, v_j).$$
2. $v_{k+1} \in V(P)$.
   Since we have excluded the existence of cycles of negative lengths we may argue that P is an elementary path. Thus the union of the two subpathes P' from v_i to v_{k+1} and P'' from v_{k+1} to v_j yield the same path as P and therefore:
$$d_{k+1}(v_i, v_j) = c(P) = c(P') + c(P'').$$
   Due to lemma 2.7.2 on page 41 we may argue that:
   - P' is a shortest v_i, v_{k+1}-path in the induced subgraph
$$G[\{v_i, v_{k+1}\} \cup \{v_1, v_2, \ldots, v_k, v_{k+1}\}] = G[\{v_i, v_{k+1}\} \cup \{v_1, v_2, \ldots, v_k\}]$$
     and therefore $c(P') = d_k(v_i, v_{k+1})$.

- P'' is a shortest v_{k+1}, v_j-path in the induced subgraph

  $$G\left[\{v_{k+1}, v_j\} \cup \{v_1, v_2, \ldots, v_k, v_{k+1}\}\right] = G\left[\{v_{k+1}, v_j\} \cup \{v_1, v_2, \ldots, v_k\}\right]$$

  and therefore $c\left(P''\right) = d_k\left(v_{k+1}, v_j\right)$.

Thus we get

$$d_{k+1}\left(v_i, v_j\right) = d_k\left(v_i, v_{k+1}\right) + d_k\left(v_{k+1}, v_j\right)$$

Summing up the two cases we get the following recursion:

$$d_{k+1}\left(v_i, v_j\right) = \min\left\{d_k\left(v_i, v_j\right), d_k\left(v_i, v_{k+1}\right) + d_k\left(v_{k+1}, v_j\right)\right\}$$

This is the basis of the following algorithm with complexity $O(|V|^3)$:

2.8 Matroids

- [5], pp. 107 ff., Knapsack problem.

2.9 Colorings

- [5], pp. 55 ff.

2.10 Matchings

- [5], pp. 271 ff.
- Christophides heuristic for the TSP, [5], pp. 291 ff.

Algorithm 8: APSPFloydWarshall (solves the APSP)

Data: A directed graph $G = (V, E, \alpha, \omega)$ with arc costs $c : E \rightarrow \mathbb{R}$ and no cycles of negative length.

Result: A distance- and a predecessor matrix $d(u, v)$ and $p(u, v)$, respectively. $d(u, v)$ is the length of the shortest u, v-path in G and $p(u, v)$ is the predecessor node of v on the shortest u, v-path in G.

```
1  begin
2      Initialize $n \leftarrow \mid V \mid$
3      for $i = 1, 2, \ldots, n$ do
4          for $j = 1, 2, \ldots, n$ do
5              if $j = i$ then
6                  $d(v_i, v_j) = 0$
7                  $p(v_i, v_j) = v_i$
8              else
9                  if $e \in E$ such that $\alpha(e) = v_i$ and $\omega(e) = v_j$ then
10                     $d(v_i, v_j) = c(e)$
11                     $p(v_i, v_j) = v_i$
12                 else
13                     $d(v_i, v_j) = +\infty$
14                     $p(v_i, v_j) = 0$
15     for $k = 0, 1, \ldots, n-1$ do
16         for $i = 1, 2, \ldots, n$ do
17             for $j = 1, 2, \ldots, n$ do
18                 if $i \neq j$ then
19                     Compute $d' = d(v_i, v_{k+1}) + d(v_{k+1}, v_j)$
20                     if $d' < d(v_i, v_j)$ then
21                         Update $d(v_i, v_j) \leftarrow d'$
22                         Update $p(v_i, v_j) \leftarrow p(v_{k+1}, v_j)$
```

Part II

Network flows

- Network simplex
- Dynamic flows [5], pp. 305 ff
- Single commodity flow, Multi commodity flow

Part III

Network design

Chapter 3

Design of trees

3.1 Steiner Trees

We consider the undirected graph $G = (V, E; c)$ and the node subset $K \subseteq V$. Then a Steiner tree $T \subseteq G$ is a tree which is spanning all nodes in K. We call the nodes in K terminals and the nodes $v(T) - K$ as Steiner points. The Steiner tree with lowest costs $c(T)$ is called the minimum Steiner tree. In the technical sens we can take the proglem of finding a minimum Steiner tree as the task to find a cost optimal connection of terminals via a given hub structure. The edge costs may be distances, rental costs or similar ones.

Laboratory 3.1.1
Consider the complete graph G with five nodes $V = \{v_1, \ldots, v_5\}$. The nodes $v_1, \ldots, v_5$ have the positions $(0,0)$, $(1,0)$, $(1,1)$, $(0,1)$ and $(1,1)$, respectively. Let $K = \{v_1, v_2, v_4, v_5\}$ be the set of terminals. The edge costs are defined by the according euclidian distances. Is it correct to say: 'The minimal Steiner tree is given by the minimum spanning tree on the graph $G[K]$ induced by the set of terminals K.'?

The computation of minimum Steiner trees is NP-complete. Obviously there are existing two polynomial solvable cases of this problem: for $|K| = 2$ and $K = V$ the Steiner tree problem corresponds to the shortes path and the minimum spanning tree problem, respectively. For all other cases we give a quiete intuitive approximation heuristic for this problem.

```
Algorithm SteinerMST
  Specification: Finds the Steiner tree close to the minimum Steiner tree.
  Input: Graph $G = (V, E; c)$, set of terminals $K \subset V$.
  Output: Steiner tree $T \subseteq G$

  1. Let $H = (K, E'; d)$ be the complete graph with edge costs defined as cost
     distances in $G$.

  2. Let $T'$ be the minimum spanning tree in $H$.

  3. Initialize $E'' = \{\}$.

  4. For all edges $f = (u, v) \in e(T')$ do:
        (a) Let $P$ be the according shortest between the terminals $u$ and $v$ in $G$
            corresponding to $f$.
        (b) Update $E'' := E'' \cup e(P)$
    5. Compute a minimum spanning tree $T$ on the graph $G[E'']$ induced by the edge
       set $E''$.
    6. As long as a non-terminal node with degree one exists in $T$ do:
        (a) Delete the node and the according incident edge.
End of Algorithm SteinerMST
```

Theorem 3.1.1
The algorithm `SteinerMST` finds a Steiner tree T with costs $c(T)$ at most $2(1-1/|K|)$ times the costs of the minimum Steiner tree. The runtime complexity of `SteinerMST` is $O\left(nm+n^2\log n\right)$.

Proof 3.1.1
Let T^* be the minimal Steiner tree for the terminals $K=\{t_1,\dots,t_k\}$ - see figure 3.1 for a schematic view. Further let D be the multigraph resulting from doublicating all edges in T^*. D has the following properties: it ist connected and each node in D ist has even degree. Then it can be shown [3] (p 14) it is shown, that D contains an Eulerian cycle C; the costs $c(C)$ are obviously $2c(T^*)$. Clearly, since all terminals t_i has degree two in D they must be visited on the Eulerian tour C. Let us assume that the terminals $t_1,\dots,t_k$ are visited in ascending order, starting from terminal t_1 (This can allways be forced by cyclic renumbering of the terminals). Then we can

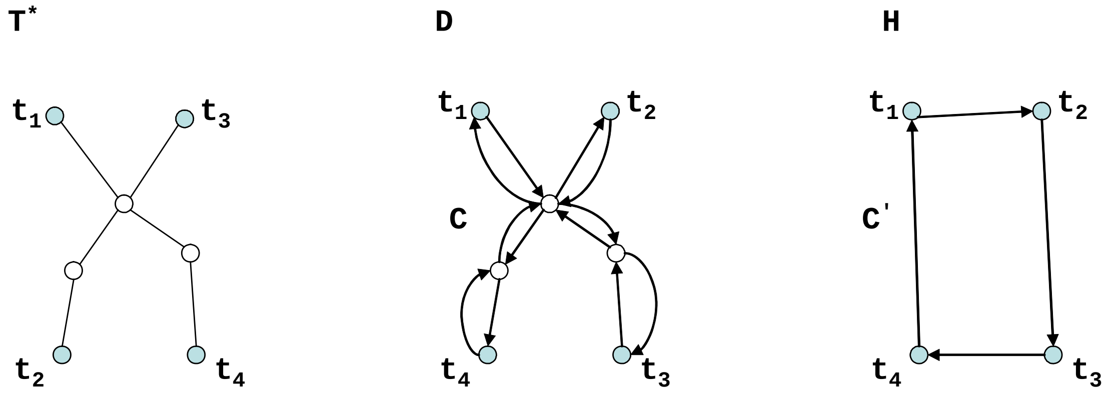


Figure 3.1: Some schematic views for the proof 3.1.1.

write $C=P_1\cup P_2\cup\cdots\cup P_k$, whereby P_i is the path from terminal t_i to terminal t_{i+1} in D and $t_{k+1}:=t_1$. Thus $c(C)=\sum_{i=1}^{k}c(P_i)$ and $c(P_i)\geq d(t_i,t_{i+1})$. Considering the Hamiltonian cycle $C'=\{t_1,t_2,\dots,t_k\}$ in the graph H, thus we find that

$$d\left(C'\right)=\sum_{i=1}^{k}d(t_i,t_{i+1})\leq c(C)=2c(T^*) \qquad \text{(i)}$$

Let $e = (t_i, t_{i+1})$ be the most expensive edge in C'. Then with

$$d(e)\left|C'\right| \geq d\left(C'\right) \Leftrightarrow d(e) \geq \frac{d(C')}{|C'|} = \frac{d(C')}{k}$$

the subgraph $\tilde{T} := C' - \{e\}$ is a spanning tree in H with costs

$$d\left(\tilde{T}\right) = d\left(C'\right) - d(e) \leq \left(1 - \frac{1}{k}\right) d\left(C'\right) \overset{\text{(i)}}{\leq} 2\left(1 - \frac{1}{k}\right) c\left(T^*\right)$$

Since T' (as computed in step 2 of algorithm `SteinerMST`) is a minimum spanning tree on H, it follows that $d(T') \leq d\left(\tilde{T}\right)$. Replacing the edges of T' by according pathes in G yields the edge set E'' with $c(E'') \leq d(T')$. Finaly computing a minimum spanning tree on the subgraph $G[E'']$ yields a Steiner tree T with $c(T) \leq c(E'')$.
In order to proof the runtime complexity we take the following results. Step 1: as shown in section **??** (page **??**) the the computation of pairwise distances d for a graph with n nodes and m edges has the complexity $O\left(nm + n^2 \log n\right)$. Step 2: computing a minimum spanning tree on H is of complexity $O(|E(H)| + |V(H)| \log |V(H)|) = O\left(|V(H)|^2\right) = O\left(|K|^2\right) \subseteq O\left(n^2\right)$ if we use the algorithm of Prim; see section **??** (page **??**). Step 4: T' has at most $n-1$ edges each of which represents a path in G with at most m edges; thus the complexity is $O(nm)$. Step 5 and 6 are of complexity $O(m + n \log n)$ and $O(n)$. □

Laboratory 3.1.2
Implement the algorithm `SteinerMST` on a platform of your chioce. Run your program on `SteinLib` instances provided in the subdirectory `[Course]\Routines\SteinerSolver` and compare the practical performance ration $c(T)/c(T^*)$, wherby $c(T^*)$ are the costs of the minimal Steiner tree, with the worst case ratio given in theorem 3.1.1. For reference you will find the file `Laboratory.m` in the directory `[Course]\Routines\SteinerSolver` as a demo implementation in `MATLAB`.

Laboratory 3.1.3
(Multicast Cost Sharing). You are in the position to be an information provider. There exists a pool of potential information users. Formally, let us define the graph $G = (V, E; c)$ with $s \in V$ and $V - \{s\}$ represents you (as a source) and the users, respectively. $c(u, v)$ are costs you have to pay to route information from node u to node v, independet from the amount of shifted information (e.g. fixed rental costs for leased line). You know also the sales $\pi(v)$ of user $v \in V$ if it is supplied with information from you; $w(s) = 0$. Thus you can optionally embark on two bussiness strategies:

Marginal cost strategy: Your profit for dealing with the users $Q \subseteq V$, $s \in Q$, is given by

$$win(Q) := \sum_{v \in Q} \pi(v) - c(T_Q)$$

whereby T_Q is the Steiner tree for the terminal set Q. Thus you search a bussiness model which maximizes your profit:

$$Q^* := \arg\max_{Q \subseteq V} win(Q)$$

So, if you have found your bussiness model it is the task to quote prizes for delivering information to the participating users. Since $win\left(Q^*\right) - win\left(Q^* - \{v\}\right)$ can be regarded as the marginal return of user $v \in Q^*$ let

$$q(v) := \pi(v) - \left(win\left(Q^*\right) - win\left(Q^* - \{v\}\right)\right) \text{ for } v \in Q^*$$

be the quote of the users.

Shapley value strategy: We consider the cost optimized connection $T(Q)$ of the users $Q \subseteq V$ to the information provider $s \in Q$. For an edge e in $T(Q)$ we split the costs $c(e)$ to all participating users, which are supplied via the edge e. Repeating this procedure for all edges we get the quote $q(v)$ which can be regarded as costs for connecting the user $v \in Q$ (via other users) to the provider s. We decide that Q is a feasible bussiness model if and only if $q(v) \leq \pi(v)$ for all $v \in Q$. The Shapely Value Strategy searches for a feasible model with maximum number of participating users.

In the directory `[Course]\Routines\MCS_SVS` you find three `*.ist` instances of G (basic `*.ist` file; see section 6.1 on page 80); $s := 1$. Find out for each instance: is it possible to earn money with that bussiness and if it is: what is the best busssiness model Q^* and what are the according quotes q for the participating users? Hint: in order to find all user supplied by an edge of an rooted tree use the interval theorem of section **??** (page **??**). In the according directory you find the demo implementation `Laboratory.m`.

3.2 k-Spanning Networks

Suppose you have to design a communication network for n locations $v_1, \ldots, v_n$. Let $c(v_i, v_j)$ be the implementation costs and assume that the communication effort (e.g. damping on copper cables) is proportional to $c(v_i, v_j)$. Thus, if you design your network as a minimum spanning tree you may get unacceptable efforts between some pairs (v_i, v_j). Thus you are requested to insert some additional links between these pairs of nodes, which motivates the following definition:

Definition 3.2.1 (k-spanning subgraph)
A subgraph H of the graph $G = (V, E; c)$ is denoted as a k-spanning subgraph, if the condition

$$d_H\left(v, v'\right) \leq k d_G\left(v, v'\right)$$

holds for all $(v, v') \in V \times V$ for some $k \geq 1$. H is called as the minimal k-spanning subgraph, if the costs $c(H) = \sum_{e \in E(H)} c(e)$ are the minimal ones among all k-spanning subgraphs.

The following theorem charakterizes k-spanning subgraphs:

Theorem 3.2.1
The subgraph $H \subseteq G = (V, E; c)$ is a k-spanning subgraph if and only if

$$d_H\left(v, v'\right) \leq k c\left(v, v'\right) \quad \text{for all } \left(v, v'\right) \in V \times V.$$

Proof 3.2.1
'⇒': If H is a k-spanning graph then $d_H(v,v') \leq kd_G(v,v') \leq kc(v,v')$ holds for all pairs $(v,v') \in V \times V$.

'⇐': We consider an arbitrary pair $(v,v') \in V \times V$. Let $P = \{v = v_1, v_2, \ldots, v_k = v'\}$ be the shortes path between v and v' in G; thus $c(P) = d_G(v,v')$. Applying the triangle inequality it follows that

$$d_H(v,v') = d_H(v_1,v_k) \leq d_H(v_1,v_2) + \underbrace{d_H(v_2,v_k)}_{\leq d_H(v_2,v_3)+d_H(v_3,v_k)}$$

and thus

$$d_H(v,v') \leq \sum_{i=1}^{k-1} d_H(v_i,v_{i+1})$$

Due to assumpton we know that $d_H(v_i,v_{i+1}) \leq kc(v_i,v_{i+1})$ for $i = 1,\ldots,k-1$. Thus

$$d_H(v,v') \leq k\sum_{i=1}^{k-1} c(v_i,v_{i+1}) = kc(P) = kd_G(v,v')$$

This completes the proof. □

The theorem 3.2.1 motivates (and obviously proofs the correctness of) the following algorithm:

```
Algorithm kSpSubGr
  Specification: Finds a $k$-spanning subgraph.
  Input: Graph $G = (V,E;c)$.
  Output: $k$-spanning subgraph $H$.

  1. Sort all edges $e_i \in E$ such that the relation $c(e_i) \leq c(e_{i+1})$ holds for all $i =$
     $1,\ldots,|E|-1$.
  2. Initialize $H = \{\}$.
  3. For $i = 1,\ldots,|E|$ do:
       (a) Let $e_i = (v,v')$.
       (b) If $d_H(v,v') > kc(e_i)$ then do:
             i. $E(H) := E(H) + \{e_i\}$.
End of Algorithm kSpSubGr
```

The computation of minimum k-spanning networks is NP-complete. Nevertheless in [4] a linear algorithm which yields a compromise between the 'minimum spanning tree' and a 'shortes distances tree' is proposed - the according tradeoff is quantified and therefore controlable.

3.3 Median of a tree

Let $T = (V = \{v_1, \ldots, v_n\}, E; c, \pi)$ be the backbone of a(n) (information) distribution network. The mapping $c : V \times V \to \mathbb{R}_+$ are the costs for passing one unit of information through and edge $(v_i, v_j) \in V \times V$; the mapping $\pi : V \times V \to \mathbb{R}_+$ represent the information demand of user $v_i \in V$. Thus if we consider the node v_i as the distribution center of our backbone, the overall transmission costs are given by

$$c(v_i) = \sum_{j=1}^{n} d_T(v_i, v_j)\, \pi(v_j) = \sum_{\substack{j=1 \\ j \neq i}}^{n} d_T(v_i, v_j)\, \pi(v_j)$$

Definition 3.3.1
The node $v^* \in V$ is called the median of tree $T = (V, E; c, \pi)$ if

$$v^* = \arg \min_{i=1,\ldots,n} c(v_i)$$

```
Algorithm TrMedian
  Specification: Finds the median of a tree.
  Input: Tree $T = (V = \{v_1, \ldots, v_n\}, E; c, \pi)$.
  Output: median $v^* \in V$.

 1. Initialize $w := \sum_{i=1}^{n} \pi(v_i)$.
 2. While $n > 0$ do:
    (a) Select an aritrary leaf $v_i$ in $T$.
    (b) If $\pi(v_i) \geq \frac{w}{2}$ then do:
          i. $v^* := v_i$ and terminate the algorithm.
    (c) Otherwise do:
          i. Let $e = (v_i, v_j)$ be the (unique) edge which is incident to $v_i$.
         ii. Update $\pi(v_j) := \pi(v_j) + \pi(v_i)$.
        iii. Update $T := T - \{v\}$, $n := n - 1$.
End of Algorithm TrMedian
```

Theorem 3.3.1
The algorithm `TrMedian` finds the median of a tree in linear time.

Proof 3.3.1
The linear complexity is obvious. We show the correctness of the algorithm by induction.
Initially, there is nothing to do for $n = 1$ and the algorithm is correct for $n = 2$: consider the nodes v_1, v_2 with $w = \pi(v_1) + \pi(v_2)$. Let us assume that $\pi(v_1) \geq \pi(v_2)$ and therfore

$\pi(v_2) \leq \frac{w}{2} \leq \pi(v_1)$. Thus the algorithm would yield $v^* = v_1$ as median - this is correct since: $c(v_1) = d_T(v_1, v_2) * \pi(v_2) \leq c(v_2) = d_T(v_1, v_2) * \pi(v_1)$. The case $\pi(v_1) \leq \pi(v_2)$ is analogous; just exchange indizes.
Let us now consider the general case n assuming that the algorithm works correct for $n-1$. Let v_i be an arbitrary leaf of the tree and (v_i, v_k) be the according incident edge. The costs of v_i are

$$c(v_i) = \sum_{\substack{j=1 \\ j \neq i}}^{n} d_T(v_i, v_j)\, \pi(v_j)$$

Consider the vertual node v' which results, if we shift v_i by ε units, $0 < \varepsilon \leq c(v_i, v_k)$, towards v_k. If we use v' as distribution center, we have costs

$$c(v') = \varepsilon\pi(v_i) + \sum_{\substack{j=1 \\ j \neq i}}^{n} (d_T(v_i, v_j) - \varepsilon)\, \pi(v_j)$$

and thus a cost difference

$$\begin{aligned} \Delta c &= c(v') - c(v_i) = \varepsilon\pi(v_i) + \sum_{\substack{j=1 \\ j \neq i}}^{n} (-\varepsilon)\, \pi(v_j) = \left(\pi(v_i) - \textstyle\sum_{\substack{j=1 \\ j \neq i}}^{n} \pi(v_j) \right) \varepsilon \\ &= (\pi(v_i) - (w - \pi(v_i)))\, \varepsilon = (2\pi(v_i) - w)\, \varepsilon \end{aligned}$$

If $\pi(v_i) > \frac{w}{2}$ the costs will increase if we choose v_k for $\varepsilon = c(v_i, v_k)$ as a candidate; if $\pi(v_i) = \frac{w}{2}$ the node v_k for $\varepsilon = c(v_i, v_k)$ will produce the same costs as v_i. In the case where $\pi(v_i) < \frac{w}{2}$ the node v_k will obviously yield lower costs and is therfore a better candidate than v_i. Eliminating v_i and updating $\pi(v_k) := \pi(v_k) + \pi(v_i)$ yields a new instance with $n-1$ nodes which will be treated correctly by assumption. Notice that the costs of this new instance are decreased by a constant factor $c(v_i, v_k)\, \pi(v_i)$ which does not affect the argument of minimization. $\square$

Laboratory 3.3.1
In the directory `[Course]\Routines\TrMedian` you find the distribution network `G01.ist` (basic `*.ist` file; see section 6.1 on page 80). Search the best distribution center for your bussiness using the algorithm `TrMedian`. Crosscheck the result by computation of the distribution costs of all other potential distribution centers. What is is the best center if we consider euclidian distances as edge costs in the according network?

3.4 Network routing

bla, bla ...

Part IV

Network analysis

Part V

Linear- and integer programming models

- Advanced models for network design problems
  - General multicommodity flow model for network design [1], pp 69-74: Application 'Design fixed cost communication system' (minimum spanning tree, Steiner tree, private network leasing, network survivability), Application 'Local access telephone network capacity expansion'
  - Primal dual approach for network design problems: NETQUEST Arbeitsbericht 05/2003 (minimum spanning tree, shortest path, Steiner tree)

Part VI

Appendix

Chapter 4

Advanced topics

Branchings and arborescences

Definition 4.0.1 (Arborescence and branching)
A rooted tree $T \subseteq G$ is also denoted as an arborescence. A forest of arborescences is called a branching. A branching which contains all nodes in V is called a spanning branching.

In the following we give an algorithm for computing the maximum spanning branching problem on a graph G. This means to find a spanning branching with a maximum sum of edge costs in G, if such a branching exists. If we are able to solve this problem we are also in the position to treat the related problems:

1. We can solve the minimum spanning branching problem by replacing the edge costs by their negatives.

2. We are in the position to find a maximum spanning arborescence, i.e. a spanning arborescence for G with a maximum sum of edge costs, if any exists. If we assume positive edge costs the algorithm for solving the maximum branching problem is intended to include as many edges into the solution as possible. Obviously no branching can contain more edges as a spanning arborescence; thus the algorithm solving the the maximum branching problem yield automatically a spanning arborescence, if one exists for the considered graph. Moreover this arborescence would be a maximum one. In the case of negative edge costs, just add a positive constant M which is large enough to the edge costs.

3. We are in the position to find a minimum spanning arborescence by replacing all edge costs by their negatives and by adding a large enough positive constant M to the edge costs in analogy to point 2.

4. We can also in the position to solve the rooted maximum (minimum) spanning arborescence problem, if any solution exists. In this case we are searching for a spanning arborescence with a specified root node s. We introduce an auxiliary node s' and an edge (s', s) with arbitrary costs. Then, since s' has indegree zero, each spanning arborescence in the graph has to be rooted at s'. This also implies that such an arborescence is also a spanning one rooted at s in the original graph. Thus the algorithm would yield a maximum spanning rooted arborescence in the case of positive edge costs; if negative edge costs exists we have to add a large enough positive constant as described above. In order to find a minimum

spanning rooted arborescence we once again replace the edge costs by their negatives and add a large enough positive constant.

Initially we formulate the maximum branching algorithm and show the correctness afterwards.

```
Algorithm MAXBR

Specification: computes a maximum branching for a directed graph.
Input: directed graph $G = (V, E, \alpha, \omega)$ with weight function $c$, $G$ does not contain self loops.
Output: maximum branching $B_0$.

1. Initialize $G_0 := G$, $V_0 := V$, $E_0 := E$, $c_0 := c, \alpha_0 := \alpha$, $\omega_0 := \omega$, $B_0 := \{\}$, $i = 0$,
   as well as the node marker $M(v) = false$ for all $v \in V_0$.

2. While any $v \in V_i$ with $M(v) = false$ exists, do

   (a) Select a $v \in V_i$ with $M(v) = false$.
   (b) If $E_i = \{\}$ or $c_i(e) \leq 0$ for all $e \in \delta^-(v) \subseteq E_i$, then set $M(v) := true$.
   (c) Else set $f := \arg\max\{c_i(e) : e \in \delta^-(v) \subseteq E_i\}$.
       i. If $B_i \cup \{f\}$ does not contain a cycle (i.e. it is a branching), then
          update $B_i :\leftarrow B_i \cup \{f\}$ and $M(v) := true$.
       ii. Else do:
          $\alpha$) Let $C_i$ and $W_i$ be the arcs and the nodes of the directed cycle
              in $B_i \cup \{f\}$, respectively.
          $\beta$) Merge $W_i$ to a new pseudo-node $p_i$ and construct the graph
              $G_{i+1} := (V_{i+1}, E_{i+1}, \alpha_{i+1}, \omega_{i+1})$ with the weight function $c_{i+1}$:
              $V_{i+1} := V_i - W_i + \{p_i\}$.
              $E_{i+1} := E_i - \{e \in E_i : \alpha_i(e) \in W_i \wedge \omega_i(e) \in W_i\}$.
              for all $e \in E_i$ with $\alpha_i(e) \in W_i \wedge \omega_i(e) \notin W_i$ do:
                  $\alpha_{i+1}(e) := p_i$, $\omega_{i+1}(e) := \omega_i(e)$, $c_{i+1}(e) := c_i(e)$
              for all $e \in E_i$ with $\alpha_i(e) \notin W_i \wedge \omega_i(e) \notin W_i$ do:
                  $\alpha_{i+1}(e) := \alpha_i(e)$, $\omega_{i+1}(e) := \omega_i(e)$, $c_{i+1}(e) := c_i(e)$
              for all $e \in E_i$ with $\alpha_i(e) \notin W_i \wedge \omega_i(e) \in W_i$ do:
                  $\alpha_{i+1}(e) := \alpha_i(e)$, $\omega_{i+1}(e) := p_i$, $c_{i+1}(e) := c_i(e) + c_i(g_i) - c_i(h_i)$
                  with $g_i := \arg\min\{c(e') : e' \in C_i\}$, $h_i \in C_i : \omega_i(h_i) = \omega_i(e)$
          $\gamma$) All nodes in $V_{i+1} - \{p_i\}$ keep their markers; $M(p_i) := false$.
          $\delta$) Update $B_{i+1} := B_i - C_i$, $i :\leftarrow i + 1$.

3. If $i = 0$ then terminate the algorithm ($B_0$ is maximum branching).

4. While $i > 0$ do:

   (a) If $p_{i-1} \in V_i$ is a root of an arborescence in $B_i$ ($\gamma^-(p_{i-1}) = 0$ in $B_i$)then
       update $B_{i-1} := B_i + (C_{i-1} - \{g_{i-1}\})$ wherby $g_{i-1} := \arg\min\{c_{i-1}(e) : e \in C_{i-1}\}$.
```

```
(b) Else for $e \in B_i : \omega_i(e) = p_{i-1}$ let $h_{i-1} \in C_{i-1} : \omega_{i-1}(e) = \omega_{i-1}(h_{i-1})$ and
    update $B_{i-1} := B_i + (C_{i-1} - \{h_{i-1}\})$.
(c) Update $i :\leftarrow i-1$.

End of Algorithm MAXBR
```

Example 4.0.1
The figures 4.1 and 4.2 give an example for the run of algorithm MAXBR.

Theorem 4.0.1
The i-th iteration of step 2 in algorithm MAXBR yields a branching B_i and with $B_i \cup \{f\}$ a cycle C_i containing the node set W_i. Hence in the current graph $G_i = (V_i, E_i)$ with weight function c_i there exists a maximum branching B^* such that

$$\left|B^* \cap \left(\bigcup\nolimits_{v \in W_i} \delta^-(v) - C_i\right)\right| \leq 1 \tag{4.1}$$

and

$$|B^* \cap C_i| = |C_i| - 1. \tag{4.2}$$

Proof 4.0.1
We show (4.1) by contradiction - see also figure 4.3. Therefore let B^* be a branching in G_i and

$$\Delta_i := \bigcup\nolimits_{v \in W_i} \delta^-(v) - C_i, \quad \Delta := B^* \cap \Delta_i$$

and let us further assume that B^* is the branching with the smallest value $|\Delta| > 1$. We mark this assumption with (i). In order to simplify the notation we describe C_i by the sequence of nodes in it; i.e. $C_i := (1, 2, \ldots, r) \in W_i$. Thus

$$\Delta = \{(u_j, v_j) : j = 1, 2, \ldots, k\} \text{ and } k > 1.$$

The indices j are (without a loss of generality) chosen in such a way, that $v_j = v_{j-1} + n$ with $n \in \mathbb{N}$. Let us furhter investigate the sets

$$B^*_j := B^* - \{(u_j, v_j)\} + \{(v_j - 1, v_j)\} \qquad j = 1, 2, \ldots, k$$

None of the sets B^*_j can be a branching in G_i. If a B^*_j would be a branching, then due to the selection strategy in step 2.c the inequality

$$c_i\left(B^*_j\right) - c_i\left(B^*\right) = c_i\left(v_j - 1, v_j\right) - c_i\left(u_j, v_j\right) \geq 0$$

and hence $c_i\left(B^*_j\right) \geq c_i\left(B^*\right)$ holds. This would imply that B^*_j is a maximum branchin with less arcs in Δ_i which is a contratiction to the assumption (i). Thus for each B^*_j there mus exist a cycle containing the arc $(v_j - 1, v_j)$. Since in B^*_j no pair of arcs has the same end nodes this cycle is a directed one which implies that there exists a directed $v_j, v_j - 1$-path P_j in B^*_j for all j. Since B^*_j and B^* exclusively differ in the arcs $(v_j - 1, v_j)$ and (u_i, v_j) and both arcs ar not in P_j, the pathes P_j are also in B^*. The pathes P_j do not exclusively consit of arcs which are in C_i. If they would, then $P_j = C_i - \{(v_j - 1, v_j)\}$ which means that with exception of v_j no other node from W_i could be an end node of any arc in a feasible branching B^*. This would be

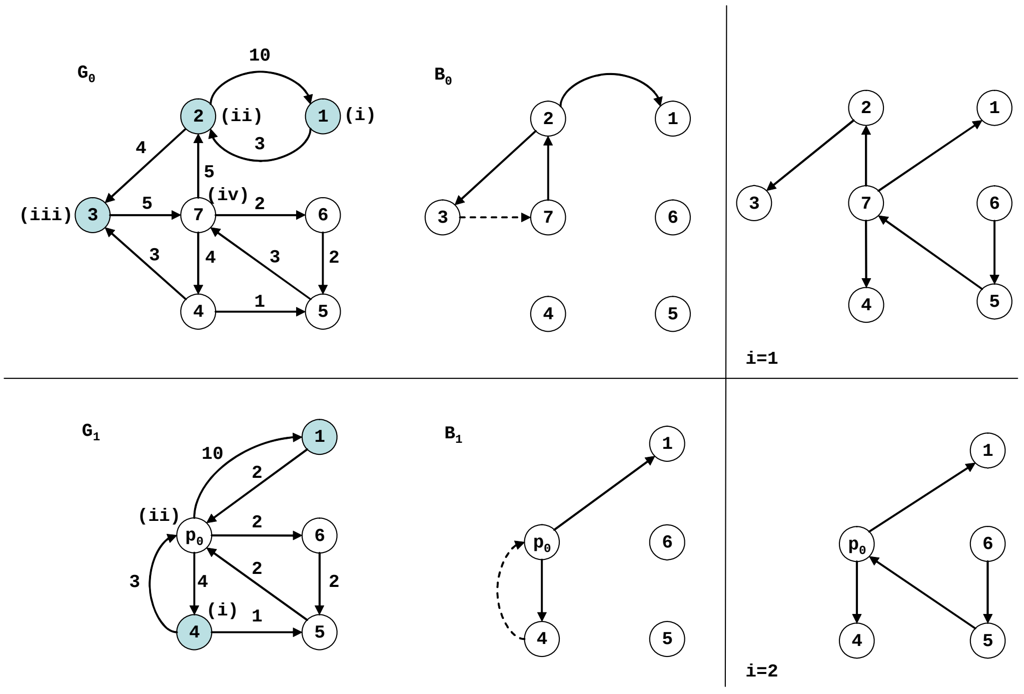


Figure 4.1: Example for algorithm `MAXBR`. The roman numbers show the sequence of selected not marked nodes; marked ones are depicted in filled mode.

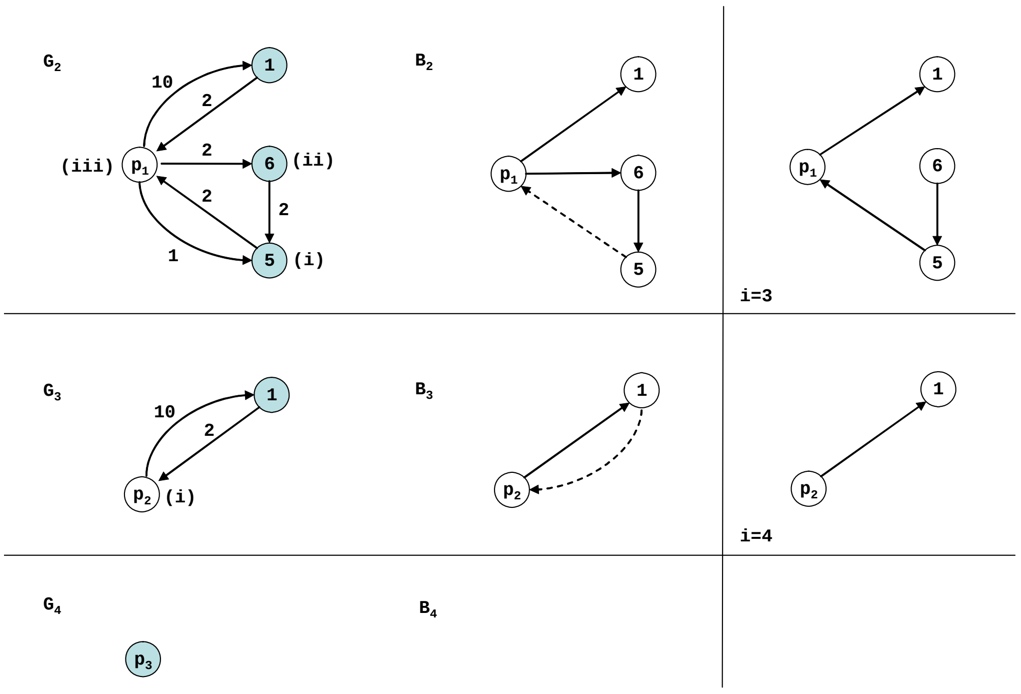


Figure 4.2: Example for algorithm `MAXBR` - continuation of figure 4.1.

a contradiction to $k > 1$. On the other side P_j must have one arc in common with Δ. Obviously each directed path in B^* from a node $v \in V_i - \{v_j - 1, v_j - 2, \ldots, v_{j-1}\}$ to the node $v_j - 1$ has to contain the subpath $(v_{j-1}, v_{j-1} + 1, \ldots, v_j - 1)$ and since v_{j-1} is exclusively reachable via the arc (u_{j-1}, v_{j-1}), this arc is the one which P_j has common with Δ. Thus we can construct in B^* a path $\tilde{P}_j$ from v_j to v_{j-1} for all j and therfore the cycle

$$v_k \tilde{P}_k v_{k-1} \tilde{P}_{k-1} v_{k-2} \ldots v_2 \tilde{P}_2 v_1 \tilde{P}_1 v_k$$

in B^* - this is a contradiction to the assumption that B^* is a branching. Thus the assumption (i) has to be false.

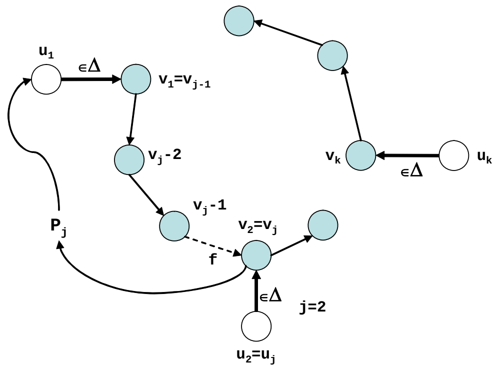


Figure 4.3: Sketch for proofing (4.1). The filled nodes are in W_i, the arcs are within B^*, the dashed arc is not in B^* but in C_i.

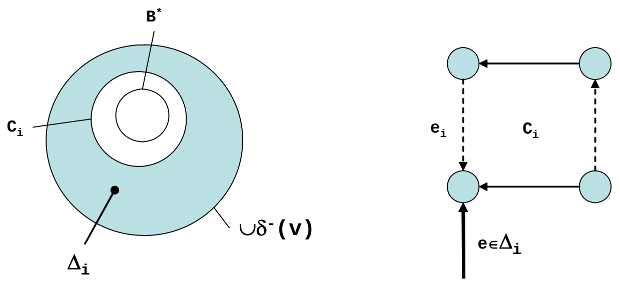


Figure 4.4: Sketch for proofing (4.2). Right plot: the filled nodes are in W_i, the arcs are within B^*, the dashed arc is not in B^* but in C_i.

In order to show (4.2) we consider a maximum branching B^* which meets (4.1), i.e. $|B^* \cap \Delta_i| \leq 1$ and which is maximal w.r.t. $|B^* \cap C_i|$ - see also figure 4.4. Then distinguish two cases:
In the first case we set $B^* \cap \Delta_i = \{\}$. Since $C_i \subseteq \bigcup_{v \in W_i} \delta^-(v)$ and $B^* \cap \left(\bigcup_{v \in W_i} \delta^-(v) - C_i \right) = \{\}$ it follows that $B^* \subset C_i$ and therefore $|B^* \cap C_i| < |C_i|$. Since $|B^* \cap C_i|$ is assumed to be maximal it follows $|B^* \cap C_i| = |C_i| - 1$.

Now let $B^* \cap \Delta_i =: \{e\}$. Let us further assume that $|B^* \cap C_i| < |C_i| - 1$. Then obviously replacing $e \in B^*$ by the edge $e_i \in C_i$ with $\omega_i(e_i) = \omega_i(e)$ yields a new branching B' with $c_i(B') \geq c_i(B)$ (due to step 2.c.). Hence B' is a maximal branching, but with more arcs in common to C_i than B^* - this contradicts to the fact that B^* is maximal w.r.t. $|B^* \cap C_i|$. Thus the assumption $|B^* \cap C_i| < |C_i| - 1$ has to be false and therefore $|B^* \cap C_i| = |C_i| - 1$. $\square$

Theorem 4.0.2
The algorithm `MAXBR` yield a maximum branching for all directed graphs $G = (V, E, \alpha, \omega)$ with weight function c.

Proof 4.0.2
Let us consider the branching $B_k \subseteq G_k = (V_k, E_k, \alpha_k, \omega_k)$ with weigth function c_k which results from the last (i.e. the k-th) iteration of step 2. Then, due to step 2.c.i for an $f \in B_k$ the inequality $c_k(f) \geq c_k(e)$ holds for all $e \in E_k : \omega_k(e) = \omega_k(f)$. Thus we may say that each $v \in V_k$ which is an end node of any arc f in B_k is reached by the maximum weighed arc f. On the other side, if $v \in V_k$ is not reached by any arc of B_k, then due to step 2.b $c_k(e) \leq 0$ holds for all $e \in \delta^-(v)$. Summarizing these two observations we may argue that B_k is a maximum branching for G_k w.r.t. c_k.
We show by induction that if B_i is a maximum branching for G_i w.r.t. c_i, then B_{i-1}, constructed according to step 4, is also a maximum branching for G_{i-1} w.r.t. c_{i-1}. We have initialized the induction for $i = k$ in the last paragraph.
Let B_i be the maximal branching in the graph G_i, $C_{i-1} \subseteq B_{i-1} + \{f\}$ is the cycle constructed in step 2.c in the $i-1$-th iteration and g_{i-1} is the cost-minimal arc within this cycle w.r.t. the cost function c_{i-1}. We assume furthermore that B^* is a maximal branching in G_{i-1}. Thus it meets the statements of theorem 4.0.1 and we have to consider two cases:

1. $\left|B^* \cap \delta^-(W_{i-1})\right| = 1$. Thus let $B^* \cap \delta^-(W_{i-1}) =: \{e\}$. See figure ? for a schematic view. Then we may argue that

$$B^* = B' + \{e\} + (C_{i-1} - \{e_{i-1}\})$$

whereby $e_{i-1} \in C_{i-1}$ with $\omega_{i-1}(e_{i-1}) = \omega_{i-1}(e)$ and $B' \subseteq E_{i-1} - \left(\delta^-(W_{i-1}) + E(W_{i-1})\right)$ (due to $|B^* \cap C_{i-1}| = |C_{i-1}| - 1$ - theorem 4.0.1). Obviously $B' + \{e\}$ represents a branching in G_i; notice that $c_i(B' + \{e\}) \leq c_i(B_i)$ holds since B_i is maximal in G_i. Moreover we have to consider that due to step 2.c.ii the costs for e are as follows:

$$c_i(e) = c_{i-1}(e) + c_{i-1}(g_{i-1}) - c_{i-1}(e_{i-1})$$

Combining these observations, it follows that

$$\begin{aligned} c_{i-1}(B^*) &= c_{i-1}\left(B' + \{e\} + (C_{i-1} - \{e_{i-1}\})\right) = \\ &= c_i\left(B' + \{e\}\right) - c_{i-1}(g_{i-1}) + c_{i-1}(e_{i-1}) + c_{i-1}(C_{i-1} - \{e_{i-1}\}) = \\ &\leq c_i(B_i) + c_{i-1}(C_{i-1} - \{g_{i-1}\}) \end{aligned} \tag{4.3}$$

Then we habe to discuss two subcases:

(a) $B_i \cap \delta^-(W_{i-1}) = 0$. Then according to step 4.a it follows that

$$B_{i-1} = B_i + (C_{i-1} - \{g_{i-1}\})$$

and considering the fact that in this case according to step 2.c.ii $c_i(B_i) = c_{i-1}(B_i)$ holds we see with (4.3) that

$$c_{i-1}(B^*) \leq c_{i-1}(B_i) + c_{i-1}(C_{i-1} - \{g_{i-1}\}) = c_{i-1}(B_i + (C_{i-1} - \{g_{i-1}\})) = c_{i-1}(B_{i-1})$$

Thus B_{i-1} has to be a maximal branching in G_{i-1}.

(b) $B_i \cap \delta^-(W_{i-1}) = \{d\}$ (this case is depicted in figure ?). Furthermore let $d_{i-1} \in C_{i-1}$ such that $\omega_{i-1}(d_{i-1}) = \omega_{i-1}(d)$. Thus according to step 4.b we get

$$B_{i-1} = B_i + (C_{i-1} - \{d_{i-1}\})$$

Once again the costs for d have been manipulated in step 2.c.ii such that

$$c_i(d) = c_{i-1}(d) + c_{i-1}(g_{i-1}) - c_{i-1}(d_{i-1})$$

Thus we get

$$\begin{aligned} c_{i-1}(B_{i-1}) &= c_{i-1}(B_i + (C_{i-1} - \{d_{i-1}\})) = \\ &= c_{i-1}(B_i) + c_{i-1}(C_{i-1} - \{d_{i-1}\}) = \\ &= c_i(B_i) - c_{i-1}(g_{i-1}) + c_{i-1}(d_{i-1}) + c_{i-1}(C_{i-1} - \{d_{i-1}\}) = \\ &= c_i(B_i) + c_{i-1}(C_{i-1} - \{g_{i-1}\}) \underset{(4.3)}{\geq} c_{i-1}(B^*) \end{aligned}$$

and therefore B_{i-1} maximal in G_{i-1}.

2. $\left|B^* \cap \delta^-(W_{i-1})\right| = 0$. This case can be shown in analogy to the first case. □

4.1 Construction of cycles of negative lengths

....

....

Chapter 5

Additional techniques

5.1 Proving techniques

In order to prove statements several techniques will be available - the most common ones are:

5.1.1 Indirect prove

Considering two logical expressions A and B the logical combinations $A \to B$ (if A then B) and $\ulcorner B \to \ulcorner A$ are logically equivalent. Thus in order to show $A \Rightarrow B$ one may also show that $\ulcorner B \Rightarrow \ulcorner A$.

Example 5.1.1
Show that whenever n^2 is even than also n is even for all $n \in \mathbb{N}$. Thus let A and B be the expressions "n^2 is even" and "n is even", respectively. Then we have to show that $A \Rightarrow B$; instead of this we may show that $\ulcorner B \Rightarrow \ulcorner A$: $\ulcorner B$ means that n is odd; thus there exists a $k \in \mathbb{N}$ such that $n = 2k + 1$. Now compute:

$$n^2 = (2k+1)^2 = 4k^2 + 4k + 1 = 2\left(2k^2 + 2k\right) + 1$$

Since $k \in \mathbb{N}$ also $2k^2 + 2k =: k' \in \mathbb{N}$ holds - thus if n is odd, there exists a $k' \in \mathbb{N}$ such that $n^2 = 2k' + 1$ which is obviously odd - thus $\ulcorner B \Rightarrow \ulcorner A$, q.e.d.

5.1.2 Prove by contradiction

Considering the logical table of $A \to B$ (if A then B) for two logical expressions A and B, there exists only one case when $A \to B$ is not true: namly the case A is true and B is false. Thus in order to show $A \Rightarrow B$ one may exclude this case. In order to do this we assume this case and show that this yield a contradiction to the assumption, i.e. formaly we show $\ulcorner(A \wedge \ulcorner B)$.

Example 5.1.2
Show that $\sqrt{2} \notin \mathbb{Q}$. Thus we have to show the following implication:

$$A : x = \sqrt{2} \Rightarrow B : \ulcorner\left(\exists\, p, q \in \mathbb{N} : x = \frac{p}{q},\ p \text{ and } q \text{ are coprime}\right)$$

In order to show this we show $A \wedge \ulcorner B$ yields a contradiction - thus $\ulcorner(A \wedge \ulcorner B)$ holds. $A \wedge \ulcorner B$ means that:

$$x = \sqrt{2} \wedge \exists\, p, q \in \mathbb{N} : x = \frac{p}{q}, p \text{ and } q \text{ are coprime}$$

respectively $\sqrt{2} = \frac{p}{q}$. Thus we see that $p^2 = 2q^2$. Since $q^2 \in \mathbb{N}$ it follows that p^2 has to be even which implies that also p has to be even (see example above). Thus there exists a $k \in \mathbb{N}$ such that $p = 2k$ which means that $p^2 = 4k^2 = 2q^2$ or $q^2 = 2k^2$. Thus also q^2 is even and therefore also q, due to the example above. Since p and q are both even, they have the common factor 2. This is a contradiction to assumption that p and q are prime to each other. Thus the case $A \wedge \ulcorner B$ can be excluded and therefore $A \rightarrow B$ is allways true; i.e. $A \Rightarrow B$.

5.1.3 Prove by complete induction

The complete induction bases on the 5th axiom of Peano: if a subset of $A \subseteq \mathbb{N}_0$ contains the number zero and and if $n \in A$ then also $n+1 \in A$, then $A = \mathbb{N}_0$. If we have a logical expression $A(n)$ such that the boolean value of A depends on the number $n \in \mathbb{N}_0$ we may argue on the basis of this axiom: if $A(0)$ is true (base case) and if $A(n)$ is true (assumption) then also $A(n+1)$ is true (inductive step), then $A(n)$ is true for all $n \in \mathbb{N}_0$.

Example 5.1.3
Show that

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \quad \text{for all } n \in \mathbb{N}_0.$$

Let us study the base case:

$$A(0) : \sum_{i=0}^{0} i = 0 = \frac{0(0+1)}{2}$$

We see that $A(0)$ is true. Under the assumption that also $A(n)$ is true let us investigate the inductive step:

$$A(n+1) : \sum_{i=0}^{n+1} i = \left(\sum_{i=0}^{n} i\right) + n + 1 = \underbrace{\frac{n(n+1)}{2}}_{A(n) \text{ is true}} + n + 1 = \frac{n(n+1)}{2} + \frac{2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

Thus we see that also $A(n+1)$ is true and therefore $A(n)$ is true for all $n \in \mathbb{N}_0$.

Chapter 6

MATLAB Implementations

6.1 Coding graphs and instances

We consider a graph $G = (V, E)$ which may be directed or undirected. Basically we store instances of graphs as follows:

Coding the node set V: we address the nodes by running integers, i.e. $V = \{1, 2, \ldots, n-1, n\}$. Thus we do not store the node set V explicitely, but rather the number of nodes n.

Coding the arc or edge set E: We use two different structures for storing E:

- Linear arc or edge identifiers store the positions in the adjacency matrix $A \in Mat\,(n, n)$ where an according arc or edge flag has been set. Position means to start at the cell $A_{1,1}$, run through the columns and increase a position marker by one till we reach the considered flag position. Depending whether we consider arcs or edges we get the identifiers id

  $$id = (v-1)\,n + u \text{ for } e \in E \text{ with } \alpha\,(e) = u \text{ and } \omega\,(e) = v$$

  or

  $$id = (\max\{u, v\} - 1)\,n + \min\{u, v\} \text{ for } e \in E \text{ with } \gamma\,(e) = \{u, v\}$$

  Notice that we do not store a symmetric adjacenc matrix for undirected graphs, but rather the according upper triangular representative. The inverse direction is:

  $$u = \begin{cases} \operatorname{mod}(id, n) & \text{if } \operatorname{mod}(id, n) > 0 \\ n & \text{otherwise} \end{cases} \quad \text{and} \quad v = \frac{(id - u)}{n} + 1$$

  for both, arcs and edges. Listing identifers multiple times means to consider parallel arcs or edes.

- Adjacency lists store all arcs or edges which are incident to the nodes of the graph. Depending whether we consider arcs or edges we get

  $$v \in Adj\,(u) \text{ and } -u \in Adj\,(v) \text{ for } e \in E \text{ with } \alpha\,(e) = u \text{ and } \omega\,(e) = v$$

  or

  $$v \in Adj\,(u) \text{ and } u \in Adj\,(v) \text{ for } e \in E \text{ with } \gamma\,(e) = \{u, v\}$$

You may use the routines `AdjList.m` and `Adj2E.m` in order to convert linear arc or edge identifiers to adjacency lists and vice versa.

6.1.1 Instance files in the basic format

The according `*.ist` file can be loaded with the MATLAB routine `RdInst`. The call

```
>> istfl = <path><filename>.ist;      (full path name of the according *.ist file)
>> [E,n,P,c] = RdInst(istfl);
```

creates the following MATLAB variables:

- `E` (array 1×m of integers) stores the arcs or edges of the graph as linear arc or edge identifiers.
- `n` (interger) is the number of nodes.
- `P` (array $n \times 2$) stores the (spatial) positions of the nodes; $P(k,1)$ and $P(k,2)$ are the x- and the y-value of the node k for $k = 1, \ldots, n$.
- `c` (array $1 \times m$) stores the costs of the arcs or edges. The value $c\,(k)$ means the costs of the arc or edge $E\,(k)$ for $k = 1, \ldots, m$.

6.1.2 Instance files in the RTR_R2007a format

–> `RdInstRb.m`

6.1.3 Instance files in the RTR_R2008a format

The according `*.ist` file can be loaded with the MATLAB routine `RdInstRc`. The call

```
>> istfl = <path><filename>.ist;      (full path name of the according *.ist file)
>> [Inst,report] = RdInstRc(istfl,[]);
```

creates a MATLAB structured array `Inst` which contains the loaded instance. The following structures are essential:

- `Inst.dim.dimension.n` (integer) is the number of nodes in the graph; the nodes are identified by running integers from 1 up to `Inst.dim.dimension.n`.
- `Inst.dim.dimension.m` (integer) is the number of edges in the graph.
- `Inst.dim.edge_type.t` (cell array 1× [≥ 1] of strings) stores a set of land use classes.
- `Inst.edge.e` (array 1×`Inst.dim.dimension.m` of integers) stores the edges of the graph, whereby `Inst.edge.e(k)` is the k-th edge coded as linear edge identifier.
- `Inst.edge.c` (array 1×`Inst.dim.dimension.m` of integers) stores the edge costs of the graph, whereby `Inst.edge.c(k)` are the costs of the k-th edge `Inst.edge.e(k)`.
- `Inst.edge.omega` (cell array 1×`Inst.dim.dimension.m` of array 1× [≥ 1] of integers) stores the land use classes which are assigned to the edges. The k-th edge of the graph

  ```
  >> Inst.edge.e(k)
  ```

```
>> m = length(Inst.edge.omega{k})
```

belongs to the land use classes

```
>> for j = 1:m, disp(Inst.dim.edge_type.t{Inst.edge.omega{k}(j)}); end;
```

- `Inst.node.x` (array 1×`Inst.dim.dimension.n` of double) the easting positions of the nodes, whereby the k-th node has the position `Inst.node.x(k)`.
- `Inst.node.y` (array 1×`Inst.dim.dimension.n` of double) the northing positions of the nodes, whereby the k-th node has the position `Inst.node.y(k)`.

Bibliography

[1] Ahuja, R.K., Applications of Network Optimization, In: Ball, M.O., Eds., Handbooks in Operations Research and Mangagement Science, Vol7, Network Models, North-Holland, 1995

[2] Optimization Algorithms for Networks and Graphs, Marcel Dekker, 1992

[3] Jungnickel, D., Graphs, Networks and Algorithms, Springer Verlag, 1999

[4] Khuller, S., et.al., Balancing minimum spanning and and shortest path trees, Proc. 4.th ACM-SIAM Symp. on Discrete Algorithms, 1993

[5] Krumke, S.O., Graphentheoretische Konzepte und Algorithmen, Teubner Verlag, 2005

[6] The SteinLib Testdata Library; http://elib.zib.de/steinlib/steinlib.php

Index