Lecture Notes, Summer Term 2015

Efficient Algorithms

Dr. Arne Meier

Version of October 15, 2019

Institut für Theoretische Informatik
Leibniz Universität Hanover

Contents

Acknowledgement

The first part of these lecture notes are build from Olaf Beyersdorffs notes for his lecture *Effiziente Algorithmen.* This part has also been influenced by the two books [Cormen et al., 2001, Ottmann and Widmayer, 2012]. The second half of this lecture is about parallel algorithms and bases on the lecture *Parallele Algorithmen* which was previously given by Rainer Parchmann who kindly gave the permission to use his lecture notes. Thank you both for this.

Thanks for improvements and corrections go out to: David Bormann, Daniel Busse, Maurice Chandoo, Martin Hasselmann, Sergey Kartamyshev, Jan Kassel, Martin Lück, Nils Mangelsen, Markus Rokicki, Nicolas Tempelmeier.

1 Graph Algorithms

1.1 Foundations

An undirected graph is a pair $G = (V, E)$ with set of vertices $V = \{1, \dots, n\}$ and set of edges $E \subseteq \binom{V}{2} = \big\{\{u, v\} \mid u, v \in V, u \neq v\big\}$.

In the following we will always use n for the number of vertices and m for the number of edges in a given graph. Now let $u \in V$. Then we define the

- *neighborhood* of u as $N_G(u) =_{\mathsf{def}} \big\{v \mid \{u, v\} \in E\big\}$,
- *degree* of u as $\deg_G(u) =_{\mathsf{def}} \|N_G(u)\|$, where for a given set A its *cardinality* (size) is defined by $\|A\|$, whereas for the length of a string x we write $|x|$,
- *minimum degree* of G as $\delta(G) =_{\mathsf{def}} \min\limits_{v \in V} \deg_G(v)$, and
- *maximum degree* of G as $\Delta(G) =_{\mathsf{def}} \max\limits_{v \in V} \deg_G(v)$.
- A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap \binom{V'}{2}$. If $E' = E \cap \binom{V'}{2}$ then G' is the V'*-induced graph.*

When we speak about graph coloring we will use the following terms.

Definition. *Let* $G = (V, E)$ *be an undirected graph.*

- *A* coloring *is a mapping* $f \colon V \to \mathbb{N}$ *such that* $f(u) \neq f(v)$ *for all* $\{u, v\} \in E$.
- G *is* k-colorable *if there exists a coloring* $f \colon V \to \{1, \dots, k\}$.
- *The* chromatic number *of* G *is* $\chi(G) = \min\{k \mid G$ *is* k*-colorable*$\}$.

Definition (Complete Graphs).

- *We refer to the* complete graph *with* n *nodes, i.e.,* $\|V\| = n$ *and* $E = \binom{V}{2}$*, as* K_n *and to the* empty graph $(V, \emptyset)$ *over* n *vertices as* E_n.
- *A set* $U \subseteq V$ *is* independent *or* stable *if there are no edges between nodes in* U*, i.e.,* $E \cap \binom{U}{2} = \emptyset$. $U \subseteq V$ *is a* Clique *if any vertex in* U *is connected to every other vertex in* U*, i.e.,* $\binom{U}{2} \subseteq E$.
- *The* Clique-number *of a graph* G *is* $\omega(G) = \max\{\|U\| \mid U \subseteq V$ *is a Clique*$\}$.

Note: it holds that $\chi(G) \geq \omega(G)$.

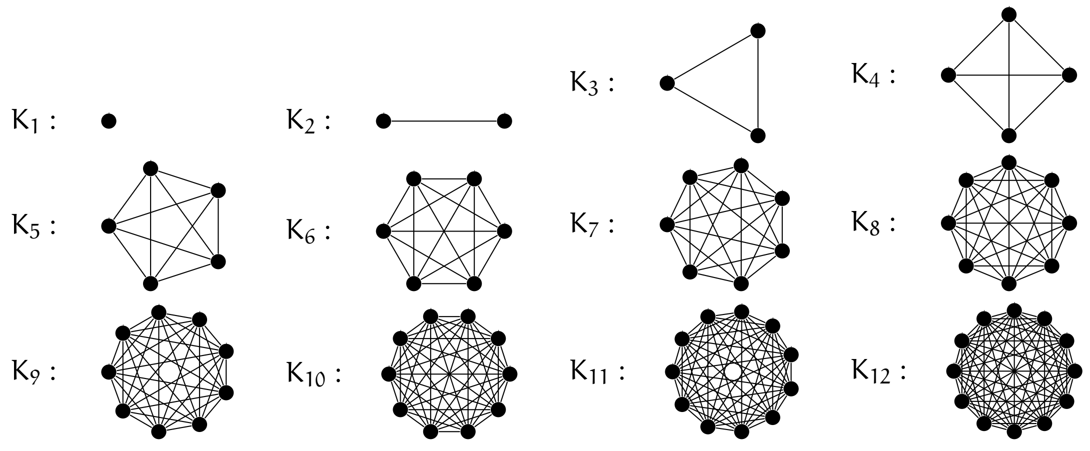


Figure 1.1: The first twelve complete graphs.

Definition (Bipartite Graphs). • *A graph* $G = (V, E)$ *is* bipartite *if the set of vertices can be divided into two distinct parts* $V_1 \uplus V_2 = V$ *such that for every* $u, v \in V_i$ *(*$i = 1, 2$*) we have* $\{u, v\} \notin E$*. Therefore one also writes* (V_1, V_2, E) *for such a graph.*

- *The complete bipartite Graph* (V_1, V_2, E) *on* $|V_1| = x$*,* $|V_2| = y$ *nodes is referred to as* $K_{x,y}$*. A graph is* bipartite *iff* $\chi(G) = 2$*.*

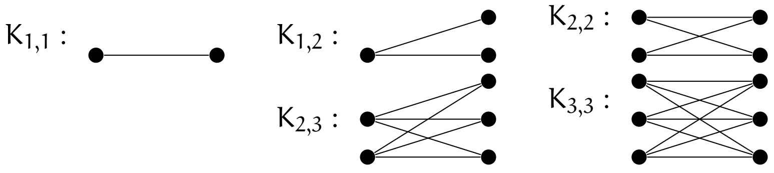


Figure 1.2: Examples for complete bipartite graphs.

Definition (Paths). • *The path of length* n *is denoted with* P_n *and defined by the number of edges. A* path *is a sequence of vertices* $\pi = v_0, \ldots, v_\ell$ *with* $\{v_i, v_{i+1}\} \in E$ *for* $i = 0, \ldots, \ell - 1$*. If for all* $1 \leq i \neq j \leq \ell$ *it holds that* $v_i \neq v_j$*, then* π *is* simple*. The* length $|\pi|$ *of the path is* ℓ*. For* $\ell = 0$ *the path is* trivial*. We refer to* $v_0, \ldots, v_\ell$ *also as* v_0*-*v_ℓ*-path. For the* i*-th element of* π *we write* $\pi(i)$ *and if* $\pi(i) = u, \pi(i + 1) = v$ *then we say* $(u, v) \in \pi$ *holds. We say that* $v_1, \ldots, v_{\ell-1}$ *are the* inner *vertices.*

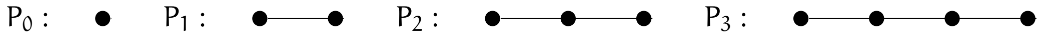


Figure 1.3: Examples for paths.

Definition (Cycles and Forests). *Let* $G = (V, E)$ *be an undirected graph, and* $u, v, v_i \in V$.

- *A* cycle *is a* u*-*v*-path of length* ≥ 3 *with* $u = v$.
- *The circle of length* n *is denoted with* C_n *and is a cycle* $v_0, \dots, v_{n-1}, v_0$ *of length* $n \geq 3$ *such that* C_n *is a simple path. The number of circles in a graph* G *is denoted with* $\nu(G)$.

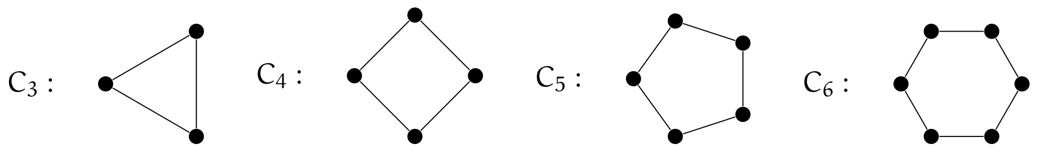


Figure 1.4: Examples for circles.

- G *is called* cycle-free *(or* acyclic *or* forest*) if there exists no circle in* G.
- *A connected forest is a* tree. *A node* u *with* $\deg(u) \geq 2$ *is an* inner *node and with* $\deg(u) \leq 1$ *is a* leaf.

Observation 1.
Every forest is bipartite.

Definition (Connected Components and Index). *Let* $G = (V, E)$ *be an undirected graph and* $G' = (V', E')$ *be a subgraph of* G. *If for every pair* $u, v \in V'$ *there exists a* u*-*v*-path then* G' *is a* connected component. *If furthermore there exists no* $w \in V \setminus V'$ *such that there exists a* $w' \in V'$ *and an* w*-*w'*-path in* G*, then* G' *is called* maximal. *The number of all maximal connected components is* $\kappa(G)$.

The index *of* G *is defined as* $\tau(G) = |E| - |V| + \kappa(G)$.

An edge e *of a graph* G *is a* bridge *if* $\kappa(G) < \kappa(G - e)$.

Observation 2.
Let e *be an edge of graph* G. *Then the following claims are equivalent.*

1. e *is a bridge.*
2. e *is not part of any circle in* G.
3. *It holds* $\kappa(G) + 1 = \kappa(G - e)$.

Theorem 1.1.
If G *is a graph and* e *an edge in* G*, then* $\kappa(G) \leq \kappa(G - e) \leq \kappa(G) + 1$.

Proof. The first inequivalence is clear and therefore we prove only the second one. Let $e = (u, v)$ and assume that $G - e$ consists of the components $G_1, G_2, \dots, G_p$ with

$p \geq \kappa(G)+2$. For convenience denote with V_i the respective set of vertices in a component. If $u, v \in V_i$ for some $1 \leq i \leq p$, we get the contradiction

$$\kappa(G) = \kappa((G-e)+e) = p > \kappa(G)\,.$$

In the remaining case $u \in V_i$ and $v \in V_j$ for $i \neq j$ we get the contradiction

$$\kappa(G) = \kappa((G-e)+e) = p-1 > \kappa(G)\,.$$

Theorem 1.2.
A graph $G = (V, E)$ is a forest if and only if every edge $e \in E$ is a bridge.

Proof. $\Rightarrow$: Let $G = (V, E)$ be a forest and $e \in E$ be an arbitrary bridge. As G is acyclic, e is not part of any circle and by Observation 2 e is a bridge.
$\Leftarrow$: If every edge is a bridge, it follows that G is acyclic and therefore a forest. □

Theorem 1.3.
Let $G = (V, E)$ be a graph and $e \in E$. Then $\kappa(G) = \kappa(G-e)$ iff e is contained in a cycle of G.

Proof. $\Leftarrow$: Let $e = (u, v)$. If there is a cycle C in G with $e \in C$, then is $C - e$ a u-v-path. Hence all paths using the edge e can then use the path $C - e$. Thus $\kappa(G) = \kappa(G - e)$.
$\Rightarrow$: Now let $\kappa(G) = \kappa(G - e)$. Hence the vertices u, v in $G - e$ are still in the same component wherefore we have a u-v-path π in $G - e$. But then the path π, e is a cycle in G. □

Theorem 1.4.
For every graph G it holds that $\tau(G) \geq 0$.

Proof. We prove this by induction on the number of edges in G. If $|E| = 0$ then G is a graph without edges, hence $\tau(G) = 0$. Now let $|E| > 0$ and e be an arbitrary edge in G. By hypothesis and Theorem 1.1 we get

$$\begin{aligned} 0 \leq \tau(G-e) &= |E| - 1 - |V| + \kappa(G-e) \\ &\leq |E| - 1 - |V| + \kappa(G) + 1 = \tau(G)\,. \end{aligned}$$

□

Theorem 1.5.
A graph G is a forest if and only if $\tau(G) = 0$.

Proof. $\Rightarrow$: Let $G = (V, E)$ be a forest. Hence by Theorem 1.2 every edge is a bridge. Observation 2 implies for the graph without edges $G_0 = (V, \emptyset)$

$$\kappa(G_0) = |V| = \kappa(G) + |E|,$$

where $\tau(G) = |E| - |V| + \kappa(G) = 0$ follows.

$\Leftarrow$: Let $\tau(G) = 0$. Now assume G contains a circle. If edge e is part of the circle, we get by Theorem 1.3 the following for the graph $G' = (V', E') = G - e$:

$$\begin{aligned}\tau(G-e) &= |E'| - |V'| + \kappa\big(G'\big)\\ &= |E| - 1 - |V| + \kappa(G)\\ &= \tau(G) - 1 = -1.\end{aligned}$$

But this a contradiction to Theorem 1.4. Hence G does not contain a circle, wherefore G is a forest. □

Definition (Equivalence Classes). *A graph* $G = (V, E)$ *is* connected *if for every pair* $\{u, v\} \in \binom{V}{2}$ *there exists a* u-v*-path. The relation*

$$Z = \left\{ (u, v) \in \binom{V}{2} \;\middle|\; \textit{there exists a } u\text{-}v\textit{-path in } G = (V, E) \right\}$$

is an equivalence relation[1] *and defines the maximal connected components in* G*, i.e.,* G/Z *is the set of all maximal connected components and* $\operatorname{index}(Z) = \kappa(G)$.

Exercise 1.
Prove Observation 1.

Exercise 2.
Prove Observation 2.

Exercise 3.
Let $G = (V, E)$ *be an undirected connected graph and* π_1, π_2 *two longest simple paths in* G*. Prove that* $\pi_1 \cap \pi_2 \neq \emptyset$*, where* $\pi_1 \cap \pi_2 := \{(u, v) \in E \mid (u, v) \in \pi_1 \textit{ and } (u, v) \in \pi_2\}$.

1.2 Directed Graphs

A *directed* graph (or digraph) is a pair $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V$. An edge $(u, u) \in E$ is a *loop.*

Definition (Set of Predecessors and Successors). *Let* $G = (V, E)$ *be a digraph.*

- *The set of* successors *of* $u \in V$ *is* $N^+(u) = \{v \in V \mid (u, v) \in E\}$.

[1] A relation $\sim\, \subseteq M \times M$ on a set M is an *equivalence relation*, if the following holds:

1. $\sim$ is reflexive, i.e., for all $x \in M$ it holds that $x \sim x$,
2. $\sim$ is symmetric, i.e. it holds that $x \sim y \Rightarrow y \sim x$ for all $x, y \in M$,
3. $\sim$ is transitive, i.e. it holds that $x \sim y$ and $y \sim z \Rightarrow x \sim z$ for all $x, y, z \in M$.

$[x]_\sim = \{y \in M\} x \sim y$ is the equivalence class of $x \in M$. $M/\sim$ is the set of all equivalence classes of $\sim$. The *index* of $\sim$ is the cardinality of $M/\sim$.

- *The set of* predecessors *of* $u \in V$ *is* $N^-(u) = \{t \in V \mid (t,u) \in E\}$.
- *The set of* neighbors *of* $u \in V$ *is* $N(u) = N^+(u) \cup N^-(u)$.
- *The* out-degree *of* $u \in V$ *is* $\deg^-(u) = \|N^+(u)\|$.
- *The* in-degree *of* $u \in V$ *is* $\deg^+(u) = \|N^-(u)\|$.

Similarly the terms of directed paths, cycles, and circles are defined.

Definition (Connectedness). *A digraph is* weakly connected *if for all pairs* $\{u,v\} \in \binom{V}{2}$ *there is a directed* u-v- *or* v-u-*path. If both paths are always existent then* g *is* strongly connected.

1.3 Shortest Paths

An important problem is the search for shortest paths in graphs. First, we just bother about the most simple case where the length of the path is measured in the number of visited edges. This is achieved via the breadth-first search (BFS) which is shown in Algorithm 1.1.

Algorithm 1.1: Breadth-first search BFS(G, s)

```
Input    : Graph $G = (V, E)$, vertex $s \in V$.
Init route[$v$] with $-$ for every $v \in V$;
queue $Q \leftarrow \emptyset$;
Q.enqueue($s$);
mark[$s$] $\leftarrow$ true;
route[$s$] $\leftarrow 0$;
while $Q$ is not empty do
    $w \leftarrow$ Q.dequeue();
    forall the edges $e = (w, v) \in E$ do
        if not mark[$v$] then
            mark[$v$] $\leftarrow$ true;
            Q.enqueue($v$);
            route[$v$] $\leftarrow$ route[$w$]+1;
```

Theorem 1.6.
Let $G = (V, E)$ *be a (di)graph. Then* route[v] *is equal to the distance of a shortest* s-v *path for all* $v \in V$.

Proof. We will prove this by induction on the length ℓ of a shortest path from s to v. Let $\ell = 0$ then $s = v$ and route[v] $= 0$. For induction step, let v be a node with distance $\ell + 1$ to s. Then there is a node $u \in N^-(v)$ with distance ℓ to s. By induction hypothesis route gets a s-u-path of length ℓ. When u is dequeued all its successors are added to Q. Therefore v is visited from u or from a previously added node z in Q. As Q is a queue

Data structure	**Requirements**
Adjacency list	$\mathsf{TIME}(\lvert V\rvert + \lvert E\rvert)$, $\mathsf{SPACE}(\lvert V\rvert + \lvert E\rvert)$
Adjacency matrix	$\mathsf{TIME}(\lvert V\rvert^2)$, $\mathsf{SPACE}(\lvert V\rvert^2)$

Table 1.1: Complexity of BFS.

route gets from s to z no longer path as from s to u. In both cases route gets a s-v-path of length $\ell + 1$. □

Observation 3.
If we use a stack instead of a queue, then the algorithm would turn into a depth-first search.

Observation 4.
If a predecessor array pred[·] is defined as pred[v] $= u$ if (u, v) is the last visited edge on a shortest path from s to v, then the route[·] array induces a predecessor subgraph $G_{route[\cdot]} = (V_{route[\cdot]}, E_{route[\cdot]})$, where $V_{route[\cdot]}$ is the set of vertices with not "−" predecessors, plus the source s:

$$V_{route[\cdot]} := \{v \in V \mid \mathit{route}[v] \neq -\} \cup \{s\}.$$

Then the edges set is defined as

$$E_{route[\cdot]} := \left\{ (\mathit{pred}[v], v) \mid v \in V_{route[\cdot]} - \{s\} \right\}.$$

1.3.1 Weighted Graphs: Jarník's algorithm

A weighted graphs consists of an additional weighting function defined on the reals.

The following algorithm goes back to the Czech mathematician Vojtěch Jarník in the 1930s and independently by Robert C. Prim in 1957. It was rediscovered by Edsger Dijkstra in 1959 and therefore is often called DJP algorithm.

Theorem 1.7.
Let $G = (V, E, w)$ a digraph and let $s \in V$. Then DJP(G, s) computes for all from s reachable nodes $t \in V$ a shortest s-t-path. This path can be tracked via route.

Proof. At first we show that all vertices reachable from s are positively marked in done[·]. Let T the by route[·] induced breadth-first-search-tree after the repeat-loop. Then, all vertices in T are positively marked in done[·]. Additionally, for every vertex $u \in T$ all direct successors of u are added to T (at the latest if u is chosen in line 6). Hence, T consists of the vertices reachable from s.

Next, we will show that from route[·] we get for every $t \in T$ a shortest s-t-path. We will prove by induction on the number k of chosen vertices prior to t the claim cost[t] $\leq d(s, t)$, where

$$d(u, v) := \min \left\{ \left\{ \sum_{i=1}^{\ell-1} w(v_i, v_{i+1}) \;\middle|\; v_1 = u, \ldots, v_\ell = v \text{ is a } u\text{-}v\text{-path.} \right\} \cup \{\infty\} \right\}.$$

Algorithm 1.2: DJP algorithm

```
Input   : A directed weighted graph $G = (V, E, w)$ and $s \in V$.
1  Init cost[$v$] with $\infty$, done[$v$] with false, and route[$v$] with $-$ for every $v \in V$;
2  Init data structure P with $(v, \infty)$ for every $v \in V$;
3  cost[s] $\leftarrow$ 0;
4  P.update(s, 0);
5  repeat
6      u $\leftarrow$ P.extractMin(), done[u] = true;
7      forall the $v \in N^+(u)$ and not done[$v$] do
8          if cost[u] + $w(u, v)$ < cost[$v$] then
9              cost[$v$] $\leftarrow$ cost[u] + $w(u, v)$;
10             route[$v$] $\leftarrow$ u;
11             P.update($v$, cost[$v$]);
12 until P.empty();
```

Induction basis $k = 0$ is $s = t$ and $\text{cost}[t] = \text{cost}[s] = 0$ ✓.

For induction step, let $s = v_0, \dots, v_\ell = t$ a shortest s-t-path in G and let v_i be the node with maximum index on this path which is chosen prior to t. By induction hypothesis it holds

$$d(s, v_i) \geq \text{cost}[v_i]. \tag{1.1}$$

Further we have

$$\text{cost}[v_{i+1}] \leq \text{cost}[v_i] + w(v_i, v_{i+1}). \tag{1.2}$$

If $t = v_{i+1}$, then it holds $\text{cost}[t] \leq \text{cost}[v_{i+1}]$ of course. If $t \neq v_{i+1}$, then t is chosen prior to v_{i+1} and we get

$$\text{cost}[t] \leq \text{cost}[v_{i+1}]. \tag{1.3}$$

Finally $v_0, \dots, v_\ell$ is a shortest s-t-path wherefore

$$d(s, v_i) + w(v_i, v_{i+1}) = d(s, v_{i+1}) \tag{1.4}$$

holds. Together we get

$$\begin{aligned} \text{cost}[t] &\overset{(1.3)}{\leq} \text{cost}[v_{i+1}] \overset{(1.2)}{\leq} \text{cost}[v_i] + w(v_i, v_{i+1}) \overset{(1.1)}{\leq} d(s, v_i) + w(v_i, v_{i+1}) \\ &\overset{(1.4)}{=} d(s, v_{i+1}) \leq d(s, t). \end{aligned}$$

□

Observation 5.
Denote with $I(\cdot), E(\cdot), U(\cdot)$ the time required for the operation Init, extractMin(), update(), *then the overall runtime of the DJP algorithm is*

$$O(I(n) + U(n) + n \cdot E(n) + m \cdot U(n)),$$

Data structure	extract-min	insert/update	init
Array	$O(n)$	$O(1)$	$O(1)$
Binary Heap	$O(\log n)$	$O(\log n)$	$O(1)$
Fibonacci Heap	$O(\log n)$ (amortized)	$O(1)$	$O(1)$

Table 1.2: Overview operation time cost for different data structures.

where $n = \|V\|$, *and* $m = \|E\|$.

Hence we get the following for different data structures

$$\begin{aligned} \textit{Array}: & \, O(\|V\|^2), \\ \textit{Binary Heap}: & \, O(\|V\| \cdot \log \|V\| + \|E\| \cdot \log \|V\|), \\ \textit{Fibonacci Heap}: & \, O(\|V\| \cdot \log \|V\| + \|E\|). \end{aligned}$$

Using an array is asymptotically optimal for dense graphs with $O(\|V\|^2)$ *edges. If we have thinner graphs, then we get a better runtime by using binary or Fibonacci heaps. An open question is wether there exists an algorithm with runtime* $O(\|V\| + \|E\|)$.

Observation 6.

Similarly to Observation 4, now we get a shortest-path tree containing shortest paths from the source s *to every vertex that is reachable from* s. *Let* $G = (V, E, w)$ *a weighted digraph containing no negative-weight cycles reachable from* s *(so that shortest paths are well defined). A shortest-path tree rooted at* s *is a directed subgraph* $G' = (V', E')$, *where* $V' \subseteq V$ *and* $E' \subseteq E$, *s.t.*

1. V' *is the set of vertices reachable from* s *in* G,
2. G' *forms a rooted tree with root* s, *and*
3. *for all* $v \in V'$, *the unique simple path from* s *to* v *in* G' *is a shortest path from* s *to* v *in* G.

1.3.2 Application: Timetable Algorithms

A *timetable* is a graph with vertices (stations, bus stops, ...) and edges (train-/bus-connections) with departure and arrival times. Hence edges are quadruples $e = (s, t, d, a)$ with

- start vertex s and target vertex t,
- departure time d in station s, and
- arrival time a in station t.

A *change* between two connections $e_1 = (s_1, t_1, d_1, a_1)$ and $e_2 = (s_2, t_2, d_2, a_2)$ is possible if $t_1 = s_2$ and $a_1 \leq d_2$. A *connection* between stations s and t is a sequence of edges $(e_1, \ldots, e_n)$, where

- s is the start vertex of e_1,
- t is the target vertex of e_n, and
- the change from e_i to e_{i+1} is possible for $i = 1, \dots, n-1$.

An addition operation is defined on the set of time labels T. Often T consists of integers between 0 and 1439 and represents the number of minutes since midnight.

The *earliest arrival problem* EAP consists of triples (s, t, d) with

- start vertex s,
- target vertex t, and
- an earliest departure time d.

The task is to compute a connection between s and t, such that s does not earlier depart than d and arrives as soon as possible at t. EAP has a *realistic* and a *simplified* version. The realistic one considers a change time at each station. The simplified one assumes a change time of 0. In the following we will consider the simplified version of EAP.

Another problem is the *minimum number of transfers problem* MNTP. For this problem the instance consists of a departure station s and a arrival station t. The task is to compute a connection with as few as possible changes.

Time-Expanded Model

Our DJP algorithm is not immediately applicable for timetables because edges are only available within a specific time interval. One of the most used approaches is the transformation of the timetable into a graph through the *time-expanded model.*

The time-expanded model follows the following procedure. For every edge $e = (s, t, d, a)$ get a copy of the start vertex s labeled with departure time d, and get a copy of the target vertex t labeled with arrival time a. The edge e is an edge between these two copies.

For every station s of the original timetable all copies of s are ordered w.r.t. their time labels. If $v_1, \dots, v_k$ are the copies of s in ascending order, then we introduce *waiting edges* (v_i, v_{i+1}) for $i = 1, \dots, k-1$. Figure 1.5 shows a transformation of a timetable into the time-expanded model.

EAP in Time-Expanded Model

In the time-expanded model every vertex is of the form (v, t) where v is a vertex in the original network and t is a time label. To solve EAP with the DJP-algorithm we allocate the weight

$$w_{\mathrm{EAP}}\big((v_1, t_1), (v_2, t_2)\big) = t_2 - t_1$$

to every edge in the time-expanded model. The weighting function on edges corresponds to the difference between arrival and departure time of the respecting connection. An

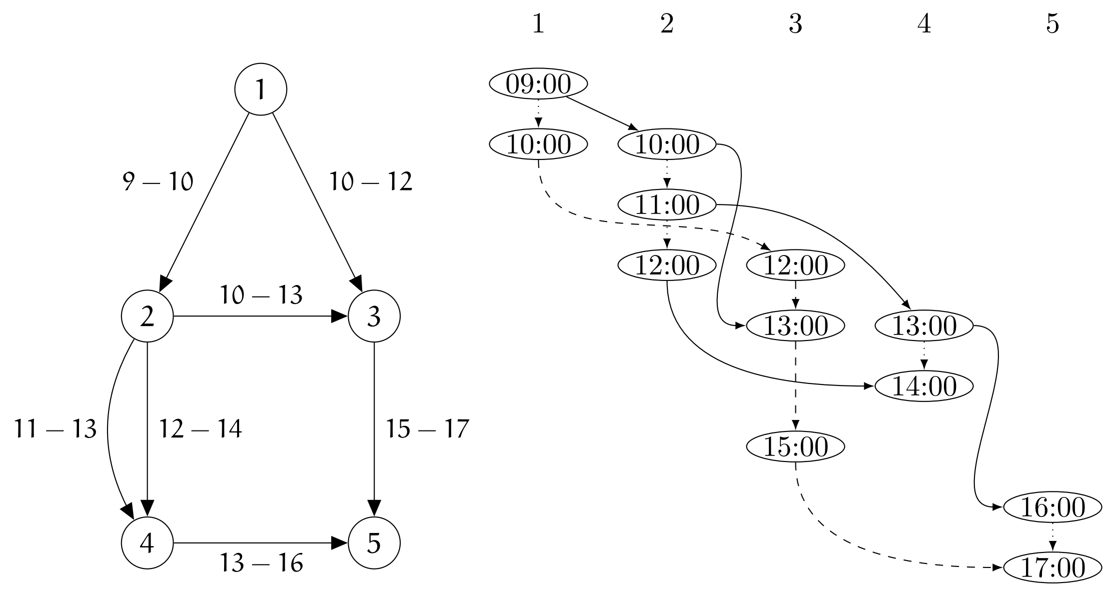


Figure 1.5: A timetable and the transformation into the time-expanded model. The dashed lines show a solution of the EAP-request (1,5,9:30).

EAP instance (s, t, d) can be solved if we search the node $s' = (s, t')$ with minimal time value $t' \geq d$ in the time-expanded model and then call the DJP-algorithm with initial node s'. The shortest path to a copy of the node t with smallest possible time label leads to a solution of the EAP instance (s, t, d).

MNTP in Time-Expanded Model

If we want to use MNTP in the time-expanded model with the DJP-algorithm we consider the weighting

$$w_{\mathrm{MNTP}}\big((v_1, t_1), (v_2, t_2)\big) = \begin{cases} 1 & \text{, if } v_1 \neq v_2, \\ 0 & \text{, otherwise.} \end{cases}$$

We solve the MNTP instance (s, t) through using the DJP-algorithm in the time-expanded model with the earliest copy of s. A shortest path to an arbitrary copy of t leads to a solution of the MNTP instance.

1.4 Negative-weight edges

Sometimes one has to work with negative edges weightings (e.g., if a specific connection has some benefits compared to others). In such situations one usually searches a shortest *simple path*, where vertices may not occur twice. The computational complexity of the problem basically relies on the fact if directed cycles of negative length are allowed (see

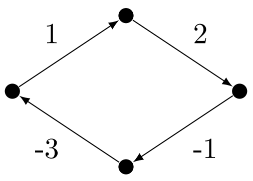


Figure 1.6: A negative cycle.

Figure 1.6). If such cycles are allowed then the problem becomes NP-hard. Otherwise there exist efficient algorithms which will be examined in the following.

The Ford-algorithm is very similar to the DJP-algorithm but can visit a vertex more than once. Therefore the algorithm revisits an edge $e = (u, v)$ such that the current distance value cost$[v]$ gets smaller due to the inequivalence cost$[u] + w(u, v) <$ cost$[v]$. Such a step is called *relaxation* (or *relaxing*) of e.

1.4.1 The Bellman-Ford-Algorithm

The Bellman-Ford-algorithm gets all shortest paths from a given initial vertex s in a weighted digraph $G = (V, E, w)$ without negative cycles. The running time is $O(|V| \cdot |E|)$ wherefore for dense graphs we get a running time of $O(|V|^3)$.

Algorithm 1.3: Bellman-Ford-algorithm

```
   Input : A non-empty connected weighted graph $G = (V, E, w)$ and $s \in V$.
1  Init cost[$v$] with $\infty$, and route[$v$] with $-$ for every $v \in V$;
2  cost[$s$] $\leftarrow 0$;
3  for $i = 1$ to $|V| - 1$ do
4      forall the $(u, v) \in E$ do
5          if cost[$u$] $+ w(u, v) <$ cost[$v$] then        // relaxation
6              cost[$v$] $\leftarrow$ cost[$u$] $+ w(u, v)$;
7              route[$v$] $\leftarrow u$;
8  forall the $(u, v) \in E$ do
9      if cost[$u$] $+ w(u, v) <$ cost[$v$] then
10         return There exists a negative cycle.
```

Exercise 4.
Prove the upper-bound property:
Let $G = (V, E, w)$ *be a weighted digraph and* $s \in V$ *be a vertex. Then cost*$[v] \geq d(s, v)$ *for all* $v \in V$ *and this invariant is maintained over any sequence of relaxation steps on the edges of* G*. Moreover, once cost*$[v]$ *achieves its lower bound* $d(s, v)$*, it never changes.*

Exercise 5.
Prove the convergence property:
Let $G = (V, E, w)$ *be a weighted digraph and let* $(u, v) \in E$*. Then, immediately after relaxing edge* (u, v) *in the **if** block we have cost*$[v] \leq$ *cost*$[u] + w(u, v)$*.*

Exercise 6.
Prove the path-relaxation property:
If $\pi = (\nu_0, \nu_1, \dots, \nu_k)$ *is a shortest path form* $s = \nu_0$ *to* ν_k*, and the edges of* p *are relaxed in the order* $(\nu_0, \nu_1), (\nu_1, \nu_2), \dots, (\nu_{k-1}, \nu_k)$*, then cost*$[\nu_k] = d(s, \nu_k)$*. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of* π*.*

Exercise 7.
Prove the no-path property:
Suppose that in a weighted, digraph $G = (V, E, w)$*, no path connects a source* $s \in V$ *to a given vertex* $\nu \in V$*. Then we have cost*$[\nu] = d(s, \nu) = \infty$ *after initialization, and this equality is maintained as an invariant over any sequence of relaxation steps on the edges of* G*.*

Exercise 8.
Prove the cost-array correctness:
Let $G = (V, E, w)$ *be a weighted digraph and* $s \in V$*, and assume that* G *contains no negative cycles reachable from* s*. Then, after* $|V| - 1$ *iterations of the* ***for*** *loop it holds cost*$[\nu] = d(s, \nu)$ *for all vertices* ν *that are reachable from* s*.*

Lemma 1.8.
Let $G = (V, E, w)$ *be a weighted digraph,* $s \in V$*, and assume that* G *contains no negative-weight cycles reachable from* s*. Then, after the initialization, the predecessor subgraph* $G_{route[\cdot]}$ *forms a rooted tree with root* s*, and any sequence of relaxation steps on edges of* G *maintains this property as an invariant.*

Proof. We start with only $s \in V_{\text{route}[\cdot]}$ and the claim is trivially true. Now consider a predecessor subgraph $G_{\text{route}[\cdot]}$ arising after a sequence of relaxation steps. First we prove that $G_{\text{route}[\cdot]}$ is acyclic. Assume for the sake of contradiction that some relaxation step creates a cycle in the graph $G_{\text{route}[\cdot]}$. Let this cycle be $c = (\nu_0, \nu_1, \dots, \nu_k)$, where $\nu_k = \nu_0$. Then route$[\nu_i] = \nu_{i-1}$ for $1 \leq i \leq k$, and w.l.o.g., assume that it was the relaxation of edge (ν_{k-1}, ν_k) that created the cycle in $G_{\text{route}[\cdot]}$.

We claim that all vertices on c are reachable from s. This holds, because each vertex on c has a not $-$ predecessor, and so each vertex on c was assigned a finite shortest-path estimate when it was assigned its not $-$ route$[\cdot]$ value. By the upper-bound property, each vertex on cycle c has a finite shortest-path weight, which implies that it is reachable from s.

We examine the shortest-path estimates on c just prior to the relaxation of (ν_{k-1}, ν_k) and show that c is a negative-weight cycle leading to a contradiction. Just before the call, we have route$[\nu_i] = \nu_{i-1}$ for $1 \leq i \leq k - 1$. Hence the last update to cost$[\nu_i]$ was by the assignment

$$\text{cost}[\nu_i] \leftarrow \text{cost}[\nu_{i-1}] + w(\nu_{i-i}, \nu_i).$$

If cost$[\nu_{i-1}]$ changed since then, it decreased. Therefore just before the relaxation call, we have

$$\text{cost}[\nu_i] \geq \text{cost}[\nu_{i-1}] + w(w_{i-1}, \nu_i) \qquad , 1 \leq i \leq k - 1. \tag{1.5}$$

Because route$[v_k]$ is changed by the call, immediately beforehand we also have the strict inequality

$$\text{cost}[v_k] > \text{cost}[v_{k-1}] + w(v_{k-1}, v_k).$$

Summing these two inequalities, we obtain the sum of the shortest-path estimates around cycle c:

$$\begin{aligned}\sum_{i=1}^{k} \text{cost}[v_i] &> \sum_{i=1}^{k} \big(\text{cost}[v_{i-1}] + w(v_{i-1}, v_i)\big) \\ &= \sum_{i=1}^{k} \text{cost}[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).\end{aligned}$$

$$\text{But } \sum_{i=1}^{k} \text{cost}[v_i] = \sum_{i=1}^{k} \text{cost}[v_{i-1}],$$

since each vertex in c appears exactly once in each summation. This equality implies

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Thus, the sum of weights around the cycle c is negative, which provides the contradiction.

Now we have proven that $G_{\text{route}[\cdot]}$ is an acyclic digraph. To show that it forms a tree rooted at s, it suffices to prove that for each vertex $v \in V_{\text{route}[\cdot]}$ there is a unique path from s to v in $G_{\text{route}[\cdot]}$.

First we show that a path from s exists for each vertex in $V_{\text{route}[\cdot]}$. The vertices in $V_{\text{route}[\cdot]}$ are those without $-$ values in route$[\cdot]$ plus s. This can be proven by induction on path lengths from s to all vertices in $V_{\text{route}[\cdot]}$.

To complete the claim proof, we need to show that for any vertex $v \in V_{\text{route}[\cdot]}$ there is at most one path from s to v in $G_{\text{route}[\cdot]}$. Suppose otherwise. That is, suppose that there are two simple paths from s to some vertex v : π_1, which can be decomposed into $s, ..., u, ..., x, z, ..., v$ and π_2, which can be decomposed into $s, ..., u, ...y, z, ..., v$, where $x \neq y$ (see Figure 1.7). But then, route$[z] = x$ and route$[z] = y$ implying the contradiction that $x = y$. We conclude that there exists a unique simple path in $G_{\text{route}[\cdot]}$ from s to v, and hence $G_{\text{route}[\cdot]}$ forms a rooted tree with root s.

Exercise 9.
Prove the predecessor-subgraph property:
Let $G = (V, E, w)$ *be a weighted digraph,* $s \in V$*, and assume* G *contains no negative-weight cycles reachable from* s*. After the initialization of the algorithm execute any sequence of relaxation steps producing* cost$[v] = d(s, v)$ *for all* $v \in V$*. Then, the predecessor subgraph* $G_{route[\cdot]}$ *is as shortest-paths tree rooted at* s*.*

Theorem 1.9.
The BF-algorithm is correct.

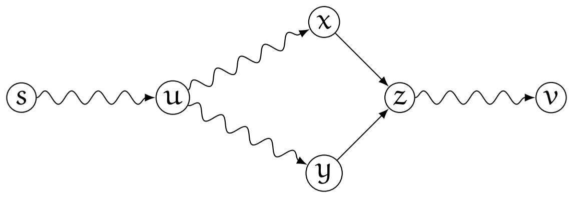


Figure 1.7: Showing uniqueness for paths in $G_{\text{route}[\cdot]}$.

Proof. Suppose that G contains no negative-weight cycles that are reachable from s. We first prove the claim that at termination, $\text{cost}[v] = d(s, v)$ for all $v \in V$. If v is reachable from s, then Exercise 8 proves this claim. If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven. The predecessor-subgraph property, along with the claim, implies that $G_{\text{route}[\cdot]}$ is a shortest-path tree. Now use the claim to show $\mathsf{BF}(G, s)$ does not return "There exists a negative cycle.". At termination, we have for all edges $(u, v) \in E$,

$$\begin{aligned}\text{cost}[v] &= d(s, v)\\ &\leq d(s, u) + w(u, v) \qquad \text{(by triangle inequality)}\\ &= \text{cost}[u] + w(u, v),\end{aligned}$$

and so none of the **if** blocks is fulfilled. Conversely, suppose that G contains a negative-weight cycle reachable from s; let this cycle be $c = (v_0, v_1, \ldots, v_k)$ with $v_0 = v_k$. Then

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0. \tag{1.6}$$

Assume for contradiction that the algorithm does not return "There exists a negative cycle.". Thus $\text{cost}[v_i] \leq \text{cost}[v_{i-1}] + w(v_{i-1}, v_i)$ for $1 \leq i \leq k$. Summing the inequalities around c gives us

$$\begin{aligned}\sum_{i=1}^{k} \text{cost}[v_i] &\leq \sum_{i=1}^{k} \big(\text{cost}[v_{i-1}] + w(v_{i-1}, v_i)\big)\\ &= \sum_{i=1}^{k} \text{cost}[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).\end{aligned}$$

Since $v_0 = v_k$ each vertex in c appears exactly once in each summation of $\sum_{i=1}^{k} \text{cost}[v_i]$ and $\sum_{i=1}^{k} \text{cost}[v_{i-1}]$, and so

$$\sum_{i=1}^{k} \text{cost}[v_i] = \sum_{i=1}^{k} \text{cost}[v_{i-1}].$$

Moreover, as for each vertex $v \in V$ there is a path from s to v iff the algorithm terminates with cost[v] $< \infty$, cost[v_i] is finite for $1 \le i \le k$. Thus,

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

which contradicts Equation (1.6). Hence the algorithm returns "There exists a negative cycle." iff it contains a negative-weight cycle. □

A slight variant of the BF-algorithm is the Bellman-Ford-Moore-algorithm, which improves in most cases the runtime. It uses the following idea: in every iteration step only the vertices are considered which have changed in the previous step. Precondition is a weighted digraph $G = (V, E, w)$ with given $s \in V$ and no negative cycles.

For acyclic graphs the runtime can be improved to $O(\|V\| + \|E\|)$ by choosing the successor in topological sorting.

1.4.2 Computation of all Shortest Paths

Sometimes one does not need to get just the shortest paths starting in a specific vertex s, but one wants the shortest paths between every pair of two vertices. Of course one can use the previous algorithms n-times, but a more simple way is to use the Floyd-Warshall-algorithm.

Let $G = (\{1, \dots, n\}, E, w)$ be a weighted digraph without negative cycles. In particular, for every loop $(i, i) \in E$ it holds $w(i, i) \ge 0$. Let $d_k(i, j)$ the length of the shortest, nontrivial path from i to j whose inner vertices are all in the set $\{1, \dots, k\}$. Then

$$d_0(i, j) = \begin{cases} w(i, j) & , \text{ if } (i, j) \in E, \\ \infty & , \text{ otherwise.} \end{cases}$$

If we are interested in only nontrivial paths, then set $d_0(i, i) = \infty$ for all $i \in V$. If trivial paths are under consideration then set $d_0(i, i) = 0$. It holds that

$$d_k(i, j) = \min\bigl\{d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j)\bigr\}.$$

The runs in time of Algorithm 1.4 is $O(|V|^3)$ and in space of $O(|V|^2)$. G does contain a negative cycle if there is an i such that $d_k(i, i) < 0$.

1.5 Flow networks

Definition. *A* network $N = (V, E, s, t, c)$ *consists of a digraph* $G = (V, E)$ *with a* source $s \in V$ *(i.e.,* $N^-(s) = 0$*) and a sink* $t \in V$ *(i.e.,* $N^+(t) = 0$*) as well as a capacity function* $c \colon V \times V \to \mathbb{N}$. *Every edge* $(u, v) \in E$ *must have a positive capacity* $c(u, v) > 0$ *and all non-edges* $(u, v) \notin E$ *do have capacity* $c(u, v) = 0$.

Definition. *A* flow *of a network* N *is a function* $f \colon V \times V \to \mathbb{Z}$ *with*

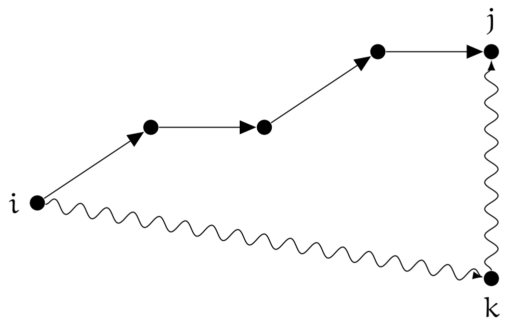


Figure 1.8: Case distinction for computation of $d_k(v, v)$.

Algorithm 1.4: Floyd-Warshall-Algorithm

```
Input : weighted digraph $G = (\{1, \dots, n\}, E, w)$.
1 forall the $u \in V$ do
2     forall the $v \in V$ do
3         if $(u, v) \in E$ then $d_0(u, v) \leftarrow w(u, v)$;
4         else $d_0(u, v) \leftarrow \infty$;
5 for $k = 1$ to $n$ do
6     forall the $i = 1$ to $n$ do
7         forall the $j = 1$ to $n$ do
8             $d_k(i, j) \leftarrow \min\{d_{k-1}(i, j), d_{k-1}(i, k) + d_{k-1}(k, j)\}$
```

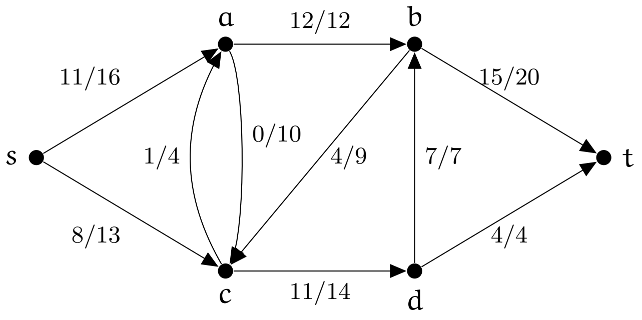


Figure 1.9: An example network. x/y denotes a flow of x and a capacity of y.

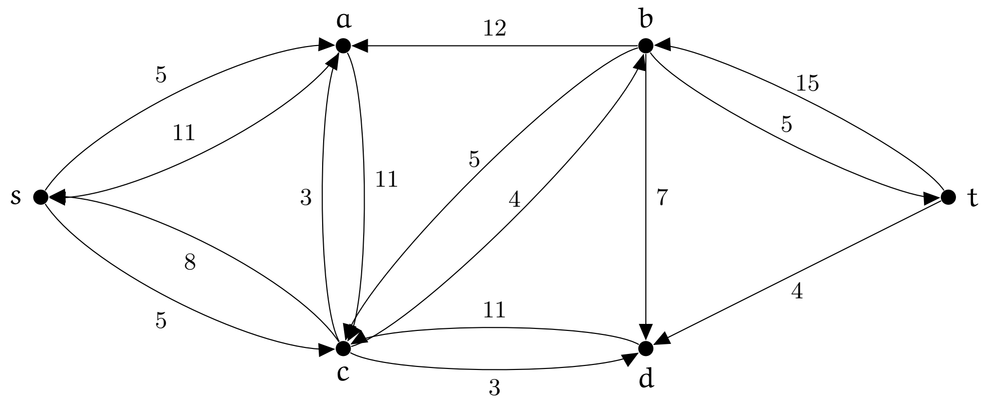


Figure 1.10: The residual network of Figure 1.9.

- $f(u, v) \leq c(u, v)$ *(capacity constraints)*
- $f(u, v) = -f(v, u)$ *(skew symmetry)*
- *For all* $u \in V \setminus \{s, t\} : \sum_{v \in V} f(u, v) = 0$ *(flow conservation); the net flow to a node is zero, except for the source which produces flow and the sink which consumes flow.*

The size *of* f *is* $|f| := \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$.

Goal: We want to compute flows of maximum size.

Definition. *Let* $N = (V, E, s, t, c)$ *be a network and* f *be a flow of* N*. The corresponding* residual network *is* $N_f = (V, E_f, s, t, c_f)$ *where*

$$c_f = c(u, v) - f(u, v),$$

and

$$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}.$$

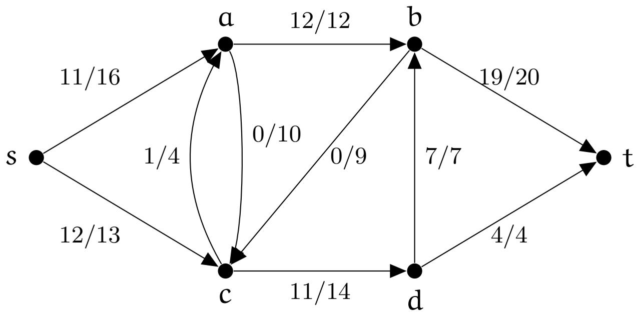


Figure 1.11: The network of Figure 1.9 extended by the augmenting path.

Definition. *Let* $N_f = (V, E_f, s, t, c_f)$ *be a residual network. Then every* s-t-*path* π *in* (V, E_f) *is an* augmenting path *for* N_f. *The capacity of* $\pi \in N_f$ *is*

$$c_f(\pi) = \min\{c_f(u, v) \mid (u, v) \in \pi\}.$$

The flow of π *in* N_f *is defined as*

$$f_\pi(u, v) = \begin{cases} c_f(\pi) & , \text{ if } (u, v) \in \pi, \\ -c_f(\pi) & , \text{ if } (v, u) \in \pi, \\ 0 & , \text{ otherwise.} \end{cases}$$

Hence $\pi = (u_0, \ldots, u_k)$ *is an augmenting path in* N_f *if and only if*

- $u_0 = s$,
- $u_k = t$,
- $u_0, \ldots, u_k$ *are pairwise different, and*
- $c_f(u_i, u_{i+1}) > 0$ *for* $i = 0, \ldots, k - 1$.

Ford-Fulkerson-Algorithm

The Ford-Fulkerson-Algorithm is used to find maximum flows and depicted in Algorithm 1.5.

Algorithm 1.5: Ford-Fulkerson-Algorithm

```
  Input    : Flow network $N = (V, E, s, t, c)$
1 forall the $(u, v) \in V \times V$ do $f(u, v) \leftarrow 0$ ;
2 while there exists an augmenting path $\pi$ for $N_f$ do $f \leftarrow f + f_\pi$ ;
```

Now we need to prove correctness of the algorithm, that is, showing that the algorithm computes a maximum flow. Additionally we need to estimate the overall runtime.

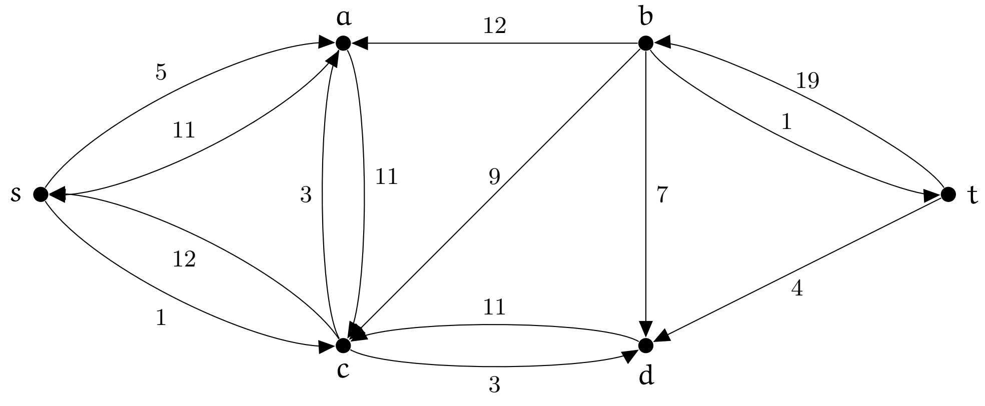


Figure 1.12: The residual network of Figure 1.11. No other augmenting path available.

Definition. *Let* $N = (V, E, s, t, c)$ *be a network and* $S, T \subseteq V$ *with* $S \dot{\cup} T = V$. *We say* (S, T) *is a* cut *through* N, *if* $s \in S$ *and* $t \in T$. *The* capacity of a cut (S, T) *is*

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v),$$

and if f *is a flow of* N, *we say to*

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

the flow through the cut (S, T).

Lemma 1.10.
For every cut (S, T) *and every flow* f *it holds* $|f| = f(S, T) \le c(S, T)$.

Proof. We prove $|f| = f(S, T)$ by induction on $k = \|S\|$.
Induction start. $k = 1$, hence $S = \{s\}$ and $T = V \setminus \{s\}$. Then we have

$$|f| = \sum_{v \in V \setminus \{s\}} f(s, v) = \sum_{u \in S, v \in T} f(u, v) = f(S, T).$$

For the induction step $k - 1 \to k$ let (S, T) be a cut with $|S| = k > 1$ and let $w \in S \setminus \{s\}$. Consider the cut (S', T') with $S' = S \setminus \{w\}$, $T' = T \cup \{w\}$. Then it holds:

$$f(S, T) = \sum_{u \in S, v \in T} f(u, v) = \sum_{u \in S', v \in T} f(u, v) + \sum_{v \in T} f(w, v)$$
$$f(S', T') = \sum_{u \in S', v \in T'} f(u, v) = \sum_{u \in S', v \in T} f(u, v) + \sum_{u \in S'} f(u, w)$$

Due to $f(w,w) = 0$ we have

$$\sum_{u \in S'} f(u,w) = \sum_{u \in S} f(u,w)$$

and therewith

$$\begin{aligned} f(S,T) - f(S',T') &= \sum_{v \in T} f(w,v) - \sum_{u \in S'} f(u,w) \\ &= \sum_{u \in V \setminus S} f(w,u) - \sum_{u \in S} f(u,w) = \sum_{v \in V} f(w,v) = 0 \end{aligned}$$

By induction hypothesis we get $|f| = f(S',T') = f(S,T)$.
The inequivalence $f(S,T) \leq c(S,T)$ follows by

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

□

Theorem 1.11 (Min-Cut-Max-Flow-Theorem).
Let f be a flow for a network $N = (V, E, s, t, c)$. Then the following claims are equivalent

(i) $|f|$ is maximal.

(ii) In N_f exists no augmenting path.

(iii) There is a cut (S,T) with $c(S,T) = |f|$.

Proof. (i) $\longrightarrow$ (ii): is clear as the existence of an augmenting path leads to an increase of f.
(ii) $\longrightarrow$ (iii): Consider the cut (S,T) with

$$S = \{u \in V \mid u \text{ is from } s \text{ reachable in the residual network } N_f\}.$$

As in N_f we have no augmenting path then $s \in S$, $t \in T$ and $c_f(u,v) = 0$ holds for all $u \in S$, $v \in T$. Due to $c_f(u,v) = c(u,v) - f(u,v)$ it follows that

$$|f| \overset{\text{Lem. 1.10}}{=} f(S,T) = \sum_{u \in S, v \in T} f(u,v) = \sum_{u \in S, v \in T} c(u,v) = c(S,T).$$

(iii) $\longrightarrow$ (i): In the case $c(S,T) = |f|$ it follows for every flow f'

$$|f'| \overset{\text{Lem. 1.10}}{=} f'(S,T) \overset{\text{Lem. 1.10}}{\leq} c(S,T) = |f|.$$

□

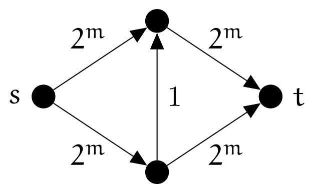


Figure 1.13: Example leading to exponential runtime for unmodified Ford-Fulkerson.

Runtime estimation

- The computation of the augmenting path π in breadth-first-search costs time $O(\|V\| + \|E\|)$.
- The computation of the residual network requires time $O(\|V\|)$ through updates along the edges in π for the next residual network.
- Let $c_0 = \sum_{v \in V} c(s, v)$ be the capacity of the cut $S = \{s\}$. Then the Ford-Fulkerson-algorithm runs through the while-loop at most c_0-times.
- By this we get a complete runtime of $O(c_0 \cdot (\|V\| + \|E\|))$.
- The size of the input is $O(\|V\|^2 \cdot \log(c_0))$, hence in the worst case we get an exponential runtime. This can be the case as well for badly chosen augmenting paths[2].

This exponential runtime can be avoided through the following strategies:

1. Consider only augmenting paths with suitable chosen minimum capacity. This leads to polynomial runtime in $\|V\|$, $\|E\|$ and $\log c_0$.
2. *Edmonds-Karp-strategy*: Always choose the shortest augmenting path in the residual network. Leads to a runtime of $O(\|V\| \cdot \|E\|^2)$ (independently from the capacity function!).
3. *Algorithm of Dinic*: The difference to Edmonds-Karp is that not only a single s-t-path is augmented, instead larger s-t-flows are considered composed of several shortest s-t-paths. Runtime: $O(\|V\|^2 \cdot \|E\|)$.

1.6 Matchings in Bipartite Graphs

Definition. *Let* $G = (V, E)$ *be an undirected graph.*

- *A set of edges* $M \subseteq E$ *is said to be a* matching*, if for all* $e, e' \in M$ *it holds*

$$e \neq e' \quad \textit{implies} \quad e \cap e' = \emptyset.$$

We then say M *is a matching of* G.

[2]The crucial point is that an augmenting path must not be a shortest path. An example is shown in Figure 1.13. Here one may use 2^m paths which always use the middle edge with capacity 1.

- *A vertex is* matched *(or saturated), if it is contained in a matching.*
- *The* matching number *of* G *is* $\mu(G) := \max\{\|M\| \mid M$ *is a matching in* $G\}$*, i.e., the size of a maximum matching in* G*. If* M *is a matching of* G *and* $\|M\| = \mu(G)$*, then we say* M *is maximum matching.*
- *A matching is* (inclusion) maximal*, if it is not contained in a larger matching.*

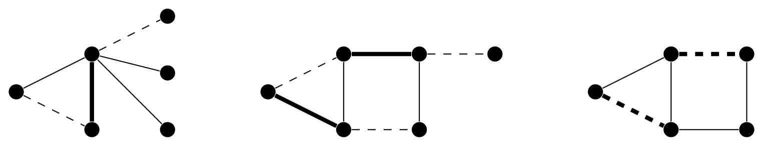


Figure 1.14: Maximal matchings denoted with thick edges. Maximum matchings with dashed edges. Hence maximal matchings not need to be perfect.

- *For a set of vertices* $U \subseteq V$ *and an set of edges* $F \subseteq E$ *denote with*

$$U_F := \{u \in U \mid \exists v \in V \text{ and } \{u, v\} \in F\}$$

the set of vertices which are covered by F *in* U*. A matching* M *is said to be* perfect *if* $V_M = V$*.*

Theorem 1.12.
If G *is a bipartite graph, then the task to compute a maximum matching can be done in polynomial time.*

Proof. We will reduce the problem of searching for maximum matchings to the following flow problem. Let $G = (U, V, E)$ be a bipartite graph. Consider the network $N(G) = (V', E', s, t, c)$ where

$V' = U \cup V \cup \{s, t\}$ with fresh vertices s, t,
$E' = \{(s, u) \mid u \in U\} \cup \{(u, v) \mid u \in U, v \in V, \{u, v\} \in E\} \cup \{(v, t) \mid v \in V\}$, and
$c(e) = 1$ for all $e \in E'$.

Then we can assign to every flow f in $N(G)$ a matching $M_f = \{\{u, v\} \in E \mid f(u, v) = 1\}$. Vice versa for every matching M we get a flow

$$f_M(v, v') = \begin{cases} 1 & \text{, if } (v, v') \in E_M, \\ -1 & \text{, if } (v', v) \in E_M, \text{ and} \\ 0 & \text{, otherwise ,} \end{cases}$$

where

$$E_M = \{(s, u) \mid u \in U_M\} \cup \{(u, v) \mid u \in U, v \in V, \{u, v\} \in M\} \cup \{(v, t) \mid v \in V_M\}.$$

Hence, the maximum flow in the residual network $N(G)$ computed by Ford-Fulkerson gets a maximum matching in G.

The runtime of this algorithm is polynomial because the size of f is bounded by $\|U\|$ and therefore the while-loop in Ford-Fulkerson is used at most $\|U\|$ times. □

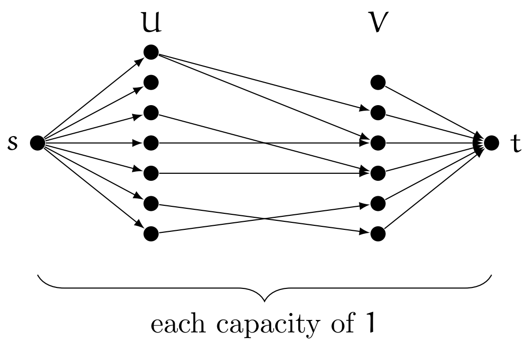


Figure 1.15: If we add source and sink we can use a maximum flow to find a maximum matching.

Let $G = (V, E)$ be a graph and $A \subseteq V$ be a set of vertices. Define the set of *neighbors* of A as

$$\Gamma(A) := \{v \in V \mid \exists u \in A \text{ such that } v \in N_G(u)\}$$

The following theorem proves a characterization of the existence of matchings in bipartite graphs, where all vertices in partition U find a matching partner.

Theorem 1.13 (Hall's marriage theorem, 1935).
For a bipartite graph $G = (U, V, E)$ *it holds that* $\mu(G) = \|U\|$ *if and only if for all subsets* $A \subseteq U$ *it holds* $\|\Gamma(A)\| \geq \|A\|$.

Proof. "$\Rightarrow$": clear, because every subset $A \subseteq U$ needs at least $\|A\|$ matching partners in $\Gamma(A)$.

"$\Leftarrow$": We will prove:

$$\mu(G) \geq \|U\| - \max_{A \subseteq U}\{\|A\| - \|\Gamma(A)\|, 0\}.$$

Here it suffices to show that there exists an $A \subseteq U$ with $\|A\| - \|\Gamma(A)\| \geq \|U\| - \mu(G)$.

The maximum flow f in the network $N(G)$ has the size $|f| = \mu(G)$. From the Min-Cut-Max-Flow-Theorem (Theorem 1.11) the existence of a cut S follows which has capacity $\mu(G)$. We choose $A = S \cap U$. Then it holds that

$$\mu(G) = c(S) \overset{(\star)}{\geq} \|U \setminus S\| + \|\Gamma(S \cap U)\| = \|U\| - \|A\| + \|\Gamma(A)\|. \tag{1.7}$$

The estimation $(\star)$ holds because every vertex $u \in U \setminus S$ adds the edge (s, u) to the capacity $c(S)$ and as every vertex $v \in \Gamma(S \cap U)$ adds an edge e to $c(S)$:

1st case: $v \in S \quad \Longrightarrow \quad e = (v, t)$

2nd case: $v \notin S \quad \Longrightarrow \quad e = (u, v)$ for an $u \in S \cap U$.

From Equation (1.7) we get by shifting

$$\|A\| - \|\Gamma(A)\| \geq \|U\| - \mu(G),$$

whereby the claim follows. □

1.7 Maximum Matchings in Arbitrary Graphs

Definition. *Let* $G = (V, E)$ *be a graph and* $M \subseteq E$ *a matching in* G.

- *Every edge* $e \in M$ *is* bound.
- *Every edge* $e \in E \setminus M$ *is* free.
- *Every vertex incident to a bounded edge is* bound*, other vertices are* free.
- *A path in* G *whose edges are alternating between bound and free, is an* alternating path *w.r.t.* M.
- *An alternating path with two free vertices on both ending edges is* augmenting *w.r.t.* M.

Theorem 1.14.
Let $G = (V, E)$ *be a graph and* $M \subseteq E$ *be a matching in* G*. Then* G *w.r.t.* M *has an augmenting path if and only if* M *is not a maximum matching.*

Proof. "$\Rightarrow$": clear, as augmenting paths can enlarge the matching through swapping the edges along the augmenting path.

"$\Leftarrow$": Let M_{max} be a maximum matching, but M is not a maximum matching. Let $k = ||M_{\text{max}}|| - ||M|| > 0$ and

$$M_{\text{sym}} = M \Delta M_{\text{max}} = (M \setminus M_{\text{max}}) \cup (M_{\text{max}} \setminus M).$$

Every vertex in V incidences at most with two vertices in M_{sym}. The subgraph (V, M_{sym}) contains no cycles of odd length, as edges of M and M_{max} must alternate. Therefore (V, M_{sym}) contains only cycles of even length or paths of arbitrary length. Each such cycle or path is an alternating path w.r.t. M, because the edges are alternating between free and bound (w.r.t. M). Due to $k = ||M_{\text{max}}|| - ||M||$ and as M_{sym} contains all edges from $M_{\text{max}} \cup M$ except the common edges, M_{sym} contains exactly k edges more from M_{max} than from M.

Every cycle in (V, M_{max}) contains equivalent many edges from M as from M_{max}. Hence on paths (without cycles) there are even k edges more from M_{max} than from M. Hence there must be at least k paths which start and end from an edge in M_{max} and alternate on the edges in M_{max} and M.

As M_{max} is a matching, all these paths are disjunct w.r.t. to the vertices and as well augmenting w.r.t. M, because both ending vertices are free w.r.t. M. □

The last theorem delivers the main idea for an algorithm to compute maximum matchings:

Though the most important question is: *how can augmenting paths be found?* We will discuss two strategies which will not immediately lead to the desired success.

```
M ← ∅;
while there is an augmenting path $\pi$ do
    M ← M$\Delta\pi$;
```

Idea 1: Breadth-first-search. This strategy can be used for bipartite graphs. We will illustrate this technique on the example graph from Figure 1.16 which leads with the corresponding breadth-first-search-tree to the augmenting path $y_3, x_2, y_4, x_4, y_5, x_6$. Generally this cannot be done in this way as shown in Figure 1.17.

Idea 2: Vertices will be visited twice. This does not work as well, because one can find in the graph of Figure 1.17 (a) the path $1, 2, 3, 6, 4, 3, 2, 7$ which is not an augmenting path. The main problem seems to be the presence of *blossoms*, as shown in Figure 1.18.

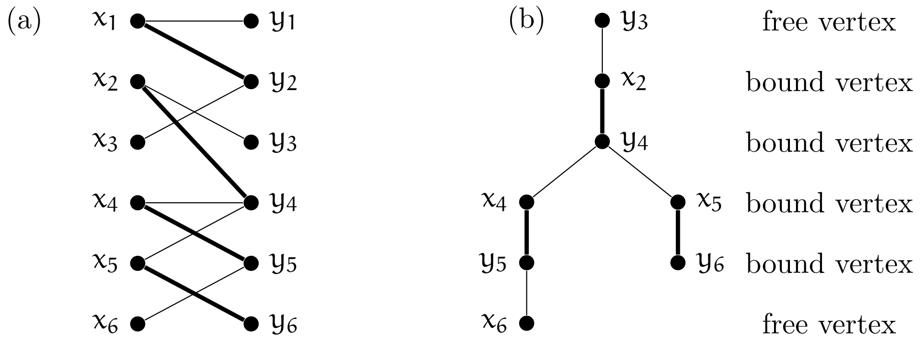


Figure 1.16: (a) Bipartite graph with matching (thick edges). (b) Breadth-first-search-tree to find augmenting paths.

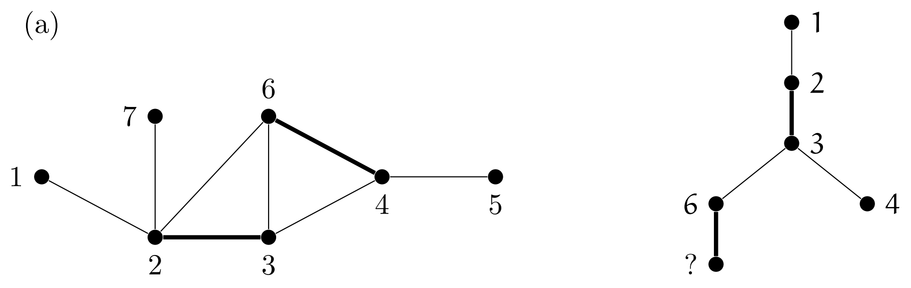


Figure 1.17: (a) Graph with matching (thick edges). (b) Augmenting path search does not work with breadth-first-search. Problem is that vertex 4 has already been visited.

Formally we define blossoms as follows.

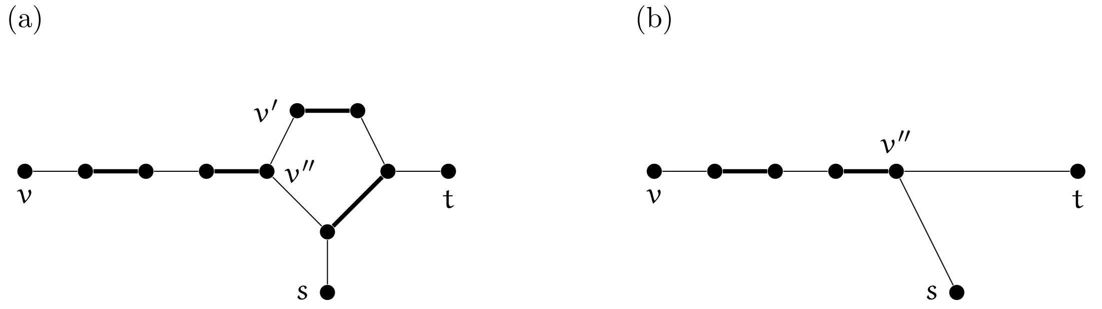


Figure 1.18: (a) A blossom (with additional vertices s, t). (b) Shrunken blossom from (a).

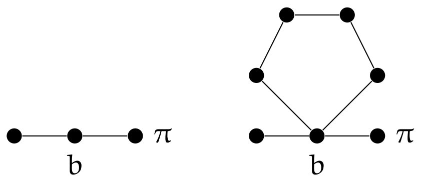


Figure 1.19: Case 2 of the proof of the Shrinking Blossom Theorem. Left the path π in G', right the path with an expanded blossom.

Definition. *Let* M *be a matching in a graph* $G = (V, E)$*. Let* π *w.r.t.* M *be an alternating path from a free vertex* v *to a vertex* v' *such that (i) there exists a vertex* v'' *which is on* π *at even distance to* v*, and (ii)* v' *is connected to* v'' *by a free edge (cf. Figure 1.18). Then we say* blossom *to a path consisting of the subpath of even length from* v'' *to* v' *together with the edge* (v', v'')*, and* stem *to the subpath from* v *to* v''*.*

If we find a blossom w.r.t. to a matching M in G, then we can transform G into G' by *shrinking* the blossom: the blossom will become a single vertex (cf. Figure 1.18 (b)). It is important to retain the alternating paths as shown in the example.

The following theorem will show that shrinking blossoms retains augmenting path existence.

Theorem 1.15 (Shrinking Blossom Theorem).
Let $G = (V, E)$ *be a graph and* $M \subseteq E$ *be a matching in* G*. If* G' *results from shrinking a blossom in* G*, then* G *contains an augmenting path w.r.t.* M *if and only if* G' *contains an augmenting path w.r.t.* M*.*

Proof. "$\Leftarrow$": Let π be an augmenting path in G' and b be the vertex resulting from the shrunken blossom.

Case 1: b is not used in π. Then π is an augmenting path in G.

Case 2: π uses b and the expanded blossom in G has exactly one vertex with π in common. Then π is an augmenting path in G (see Figure 1.19).

(b)

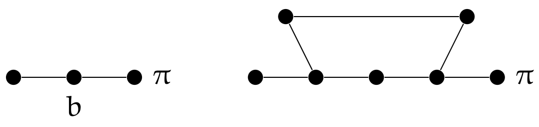


Figure 1.20: Case 3 of the proof of the Shrinking Blossom Theorem. Left the path π in G', right the path with an expanded blossom.

Case 3: π uses b and the expanded blossom in G has more than one vertex with π in common. Then the expansion of π is of one of the two paths through the blossom b in an augmenting path in G (see Figure 1.20).

"$\Rightarrow$": This direction is harder to prove. We will show this constructively with the algorithm of Edmonds which finds augmenting paths (Algorithm 1.6). The algorithm explores the graph by shrinking blossoms whenever they are encountered. During the procedure the algorithm constructs a forest which consists of trees of alternating paths rooted at the free vertices. Therefore we replace undirected edges by the two corresponding directed ones.

In the algorithm we will use the following strategy and notion:

- Construct a forest of trees with alternating paths, starting with a free vertex.
- If blossoms are found, then shrink them.
- Every vertex has three possible states: unreached, even, or odd.
- For every vertex v we save its predecessor $\mathsf{pred}(v)$ in the recently constructed tree.
- For every bound vertex v let $\mathsf{partner}(v)$ be the matching partner of v. □

Now Edmonds-Algorithm provides us with a technique to find augmenting paths and thereby we can compute maximum matchings in arbitrary graphs. The correctness follows from the previous theorem. The runtime of the algorithm is $O(\|V\| \cdot \|E\|)$, if one uses a specific data structure to maintain disjunct subsets which is used for blossoms and subtrees. However there exists an improved algorithm by Micali and Vaziran with a runtime of $O(\sqrt{\|V\|} \cdot \|E\|)$.

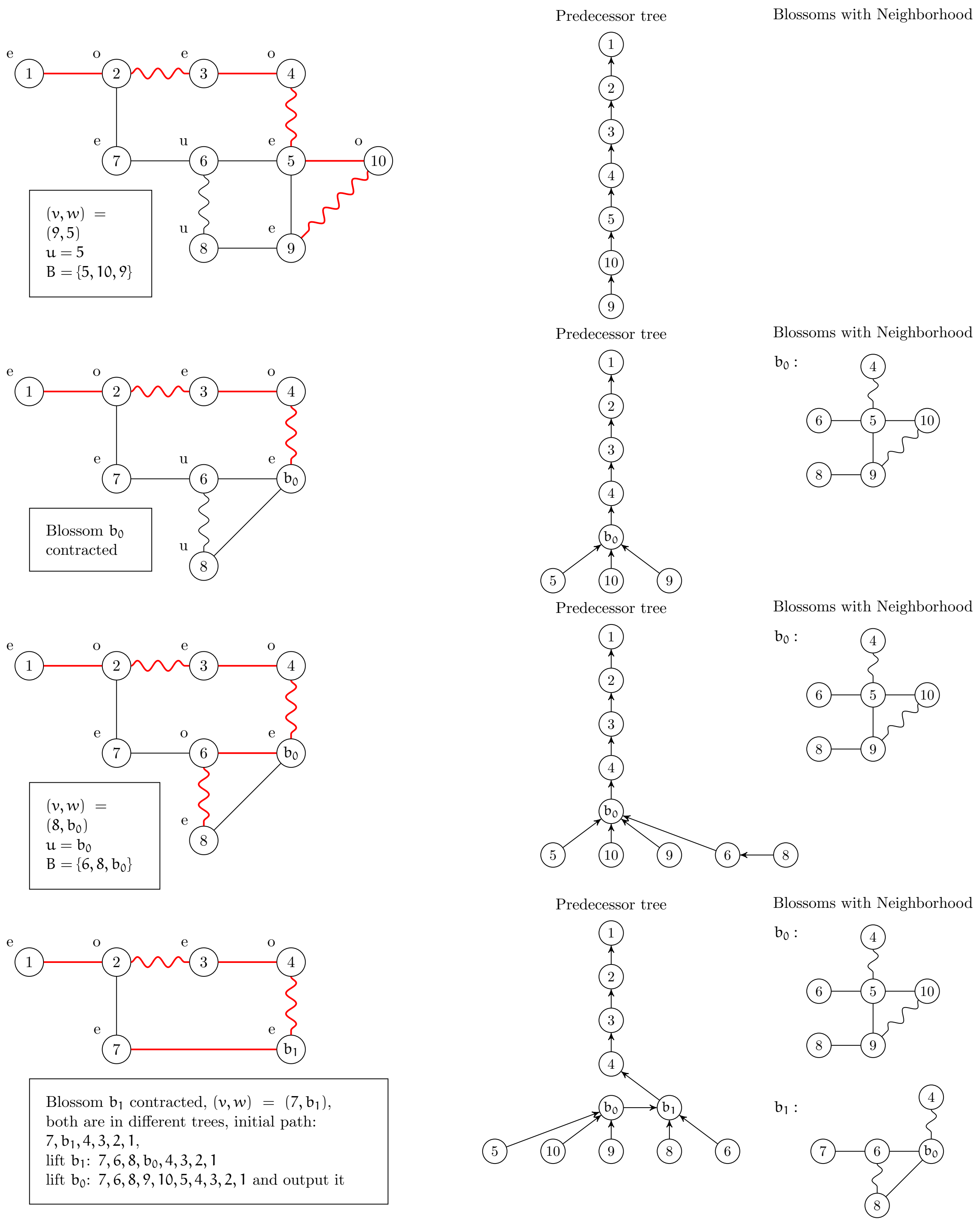
1.8 Maximum Weighted Matchings

Now we will consider undirected graphs $G = (V, E)$ with a weighting function $w\colon E \to \mathbb{Z}$. The task is to find a matching M with maximum weight

$$w(M) = \sum_{e \in M} w(e).$$

Algorithm 1.6: Edmonds-Algorithm for the computation of augmenting paths.

```
   Input    : Graph $G = (V, E)$ with a matching $M$
   Output   : an augmenting path in $(G, M)$ or $\emptyset$ if none exists
1  forall the $v \in V$ such that $v$ is free do state($v$) $\leftarrow$ even;
2  forall the $v \in V$ such that $v$ is bound do state($v$) $\leftarrow$ unreached;
3  forall the $v \in V$ do pred($v$) $\leftarrow$ nil;
4  $i \leftarrow 0$ blossom id;
5  $\mathcal{B} \leftarrow \emptyset$ is the set of constructed blossoms plus their neighborhood during the algorithm;
6  while there is an unmarked edge $(v, w)$ with state($v$) is even do
7      mark $(v, w)$;
8      if state($w$) = odd then // Case 1: cycle of even length found
9          do nothing;
10     else if state($w$) = unreached then // Case 2: extend path
11         state($w$) $\leftarrow$ odd;
12         state(partner($w$)) $\leftarrow$ even;
13         pred($w$) $\leftarrow$ $v$;
14         pred(partner($w$)) $\leftarrow$ $w$;
15     else if state($w$) = even then
16         if $v$ and $w$ are in the same tree w.r.t. pred($\cdot$) then // Case 3: shrink the blossom
17             $u \leftarrow$ nearest common ancestor of $v$ and $w$ in the tree;
18             $B \leftarrow \{v \mid v$ is a descendant of $u$ and ancestor of $v$ or $w\}$;
19             add blossom $G|_B$ with its neighborhood to $\mathcal{B}$ and mark it with $i$;
20             replace $G|_B$ by $b_i$ in $G$ and connect neighborhood of $B$ to $b_i$ (if
               $(v, w_1), \ldots, (v, w_k) \in E$ and all $w_i \in B$ then $(v, b)$ is matched if one $(v, w_i)$ is
               matched);
21             pred($b_i$) $\leftarrow$ pred($u$), pred($x$) $\leftarrow$ $b_i$ for each vertex $x \in B$;
22             pred($u$) $\leftarrow$ $b_i$, for each $u$ with pred($u$) $= x$ for some $x \in B$;
23             state($b_i$) $\leftarrow$ even, $i \leftarrow i + 1$;
24         else // Case 4: augmenting path found
25             $\pi$ is the path from the root of the tree containing $v$ to the root of the tree
               containing $w$ while $\pi$ contains blossoms do
26                 let $(u, b_j, v)$ be the subpath in $\pi$, where $j$ is maximum;
27                 let $B$ be the corresponding $j$th blossom in $\mathcal{B}$;
28                 $\pi' \leftarrow u, w_1, \ldots, w_n, v$ path of even length through $B$ and replace $b_j$ by $\pi'$
                   in $\pi$;
29             return $\pi$;
30 return $\emptyset$;
```



$(v, w) = (9, 5)$
$u = 5$
$B = \{5, 10, 9\}$

Blossom b_0 contracted

$(v, w) = (8, b_0)$
$u = b_0$
$B = \{6, 8, b_0\}$

Blossom b_1 contracted, $(v, w) = (7, b_1)$, both are in different trees, initial path: $7, b_1, 4, 3, 2, 1$,
lift b_1: $7, 6, 8, b_0, 4, 3, 2, 1$
lift b_0: $7, 6, 8, 9, 10, 5, 4, 3, 2, 1$ and output it

Figure 1.21: Example of Edmonds-Algorithm

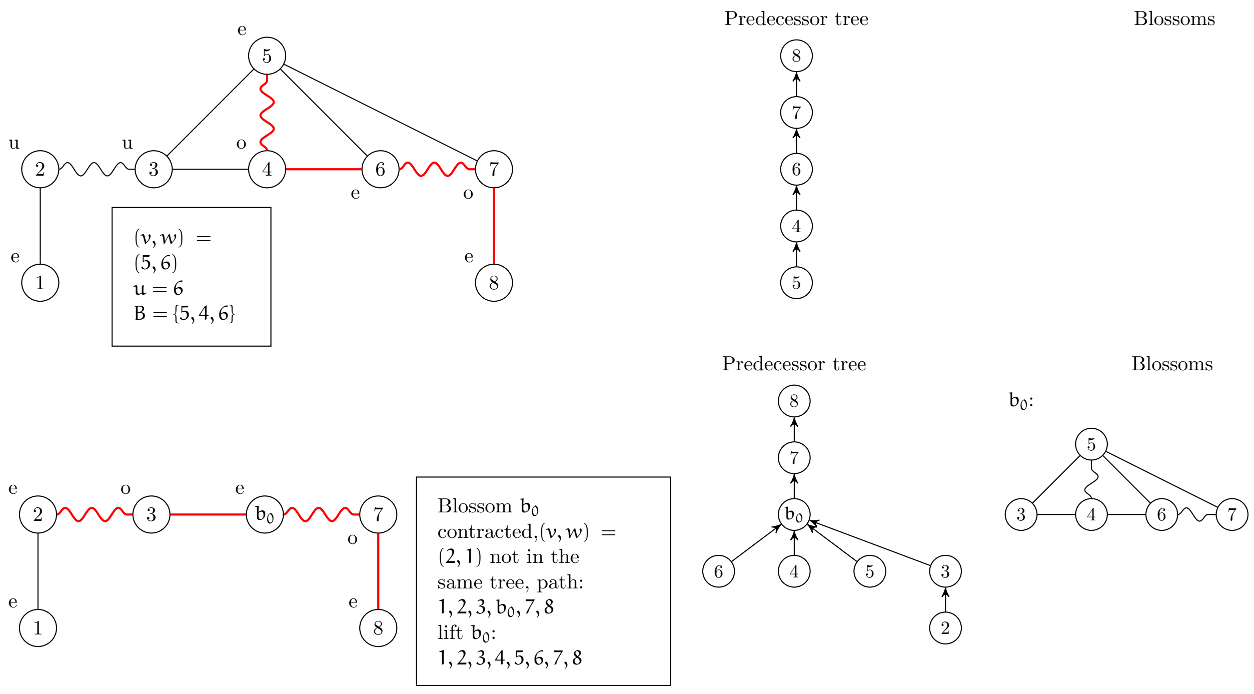

Predecessor tree
Blossoms
(v, w) = (5, 6)
u = 6
B = {5, 4, 6}
Blossom b_0 contracted, (v, w) = (2, 1) not in the same tree, path: 1, 2, 3, b_0, 7, 8
lift b_0: 1, 2, 3, 4, 5, 6, 7, 8
b_0:

Figure 1.22: Another example of Edmonds-Algorithm

Note that we do not compute a maximum weighted perfect matching. The matching we are looking for not need to be of maximum size.

We will compute maximum weighted matchings through the search of augmenting paths, however, we will need to consider cycles, because they can increase the weight.

Let π be an alternating path or cycle w.r.t. the matching M. Then we set the weight of π as the sum weight of the free edges minus the sum weight of the bound edges:

$$w(\pi) = \sum_{e \in \pi, e \notin M} w(e) - \sum_{e \in \pi \cap M} w(e).$$

Algorithm 1.7 then computes maximum weighted matchings. The correctness will follow from the next two theorems.

Theorem 1.16.
Let M be a matching with maximum weight w.r.t. to all matchings with $||M||$ edges, and let π be an augmenting path with maximum weight w.r.t. M. Then $M \Delta \pi$ is a matching with maximum weight w.r.t. all matchings with $||M|| + 1$ edges.

Proof. Let M_{max} be a matching with maximum weight under all matchings with $||M||+1$ edges. Consider the symmetric difference $M_{\text{sym}} = M \Delta M_{\text{max}}$.

Claim. *Every path or cycle π in (V, M_{sym}) of even length has weight 0 w.r.t. M.*

Proof (of claim). M and M_{max} are both matchings with maximum weight. If $w(\pi) > 0$ w.r.t. M, then M could be augmented. If $w(\pi) < 0$, then M_{max} could be augmented. □

Algorithm 1.7: Algorithm to compute maximum weighted matchings.

```
1 $M \leftarrow \emptyset$, $w \leftarrow 0$, $w' \leftarrow 0$;
2 while $w' \leq w$ do
3     $w' \leftarrow w$;
4     $M' \leftarrow M$;
5     Compute augmenting path $\pi$ with the help of Edmond's algorithm of maximum weight
      $w(\pi)$;
6     $M \leftarrow M\Delta\pi$;
7     $w \leftarrow w(M)$;
8 return $M'$;
```

As $||M_{\max}|| - ||M|| = 1$ we know that M_{sym} has exactly one edge more from $M_{\max}$ than from M.

Claim (Path pairs). *There exists a path π^* of odd length in (V, M_{sym}) which has exactly one edge more from M_{max} than from M. Except for this path π^* the paths of odd length in (V, M_{sym}) can be combined to pairs in such a way that for every pair equally many edges from M and M_{max} are used and the weight of this pair w.r.t. M is 0.*

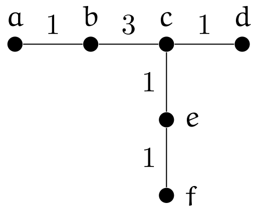
Proof (of claim). As an augmenting path adds exactly one edge to M there is exactly one edge more from $M_{\max}$ than from M present in M_{sym}. □

The path π^* can be used for augmentation of M, i.e., we set $M' = M\Delta\pi^*$. Then M' is a matching with $||M|| + 1$ edges and the same weight as $M_{\max}$. □

For the correctness of the algorithm we will need to make sure that the termination criterion of the **repeat**-loop is correct. The following theorem will do this.

Theorem 1.17.
Let M be a matching of maximum weight among matchings of size $||M||$, let π be an augmenting path for M of maximum weight, and let M' be the matching formed by augmenting M using π. Then M' is of maximum weight among matchings of size $||M|| + 1$.

Proof. Let $M_{\max}$ be a matching of maximum weight among matchings of size $||M|| + 1$. Consider the symmetric difference $M\Delta M_{\max}$. Define the weight of a path of cycle in $M\Delta M_{\max}$ with respect to M. Any cycle or even-length path in $M\Delta M_{\max}$ must have weight zero; a cycle or path of positive or negative weight contradicts the choice of M or $M_{\max}$, respectively. $M\Delta M_{\max}$ contains exactly one more edge from $M_{\max}$ than from M. Thus we can pair all but one of the odd-length paths so that each pair has an equal number of edges in M and in $M_{\max}$. Each pair of paths must have total weight zero; a positive or negative weight pair agains contradicts the choice of M or $M_{\max}$. Augmenting M using the remaining path gives a matching of size $||M|| + 1$ and of the same weight as $M_{\max}$. □



i	M	M'	w	w'
0	$\emptyset$	$\emptyset$	0	0
1	bc	$\emptyset$	3	0
2	bc/ef	bc	4	3
3	ab/cd/ef	bc/ef	3	4

1.9 Coloring Planar Graphs

In this part we are going to talk about coloring of plane graphs. Historically, it goes back until the mid of the 18th century. There it was firstly mentioned by Francis Guthrie who postulated the four color conjecture, i.e., any map can be colored with four colors such that adjacent regions sharing a common border do not receive the same color. In 1879 Alfred Kempe published a paper where he claimed to prove this 4 color conjecture by Guthrie and for a decade the problem was considered solved.

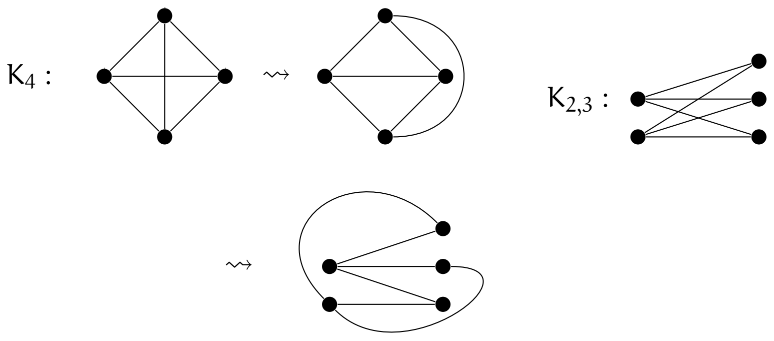
In 1890 though, Percy John Heawood showed that Kempe's proof was flawed. However Headwood was able to prove the five color theorem. In the following century a vast amount of work was built around the four color conjecture. Finally in 1976 it was proven by Kenneth Appel and Wolfang Haken. Remarkably, they used several old ideas from Heawood and Kempe and ignored the current developments. Further this proof is also very prominent for being the first major computer-aided proof.

We will now introducing the necessary notion for plane graphs and colorings. Then we will prove Euler's polyhedron formula and visit the five color theorem. Finally we will consider the interesting generalization list-coloring and prove a list-coloring result for five colors.

1.9.1 Foundations

Definition. *A graph is* planar *if it can be embedded in the plane such that two different edges do not meet in a point other their common end. Such graphs are also called* plane graphs.

Examples of planar graphs are K_4 and $K_{2,3}$:



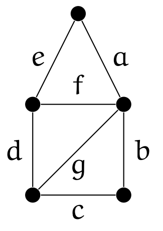


Figure 1.23: A planar graph $G = (V, E, R)$ together with its frontier set $R = \big\{\{a, f, e\}, \{f, d, g\}, \{b, g, c\}, \{a, b, c, d, e\}\big\}$.

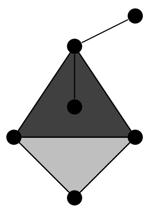
Examples of non-planar graphs are K_5 and $K_{3,3}$ whose non-planarity will be proven later. We will see, that these two graphs are the "typical" non-planar graphs (Theorem of Kuratowski).

The edges of a planar graph divide the plane into *regions* from which one is unbounded. We will stick to the following notion.

Definition. *Let* $G = (V, E)$ *be a graph. Then*

- $r(G) = r$ *denotes the number of regions of* G*,*
- $m(G) = m$ *is the number of edges, and*
- $n(G) = n$ *is the number of vertices.*

The *frontier of a region* is the set of edges which meet that region. For a region r we denote with $d(r)$ the number of edges at the frontier of r, where included edges count twice.



$d(■) = 5$

$d(□) = 3$

Apparently it holds that

$$\sum_{r \text{ is a region}} d(r) = 2 \cdot m(G).$$

A planar graph is denoted by the triple $G = (V, E, R)$ where R consists of the frontiers of all regions.

1.9.2 Euler's polyhedron formula

Theorem 1.18 (Euler's polyhedron formula, 1750).
Let G *be a planar, connected and non-empty graph. Then it holds that* $n(G) - m(G) + r(G) = 2$.

Proof. We prove the result by induction on $\mathsf{m}(\mathsf{G})$ the number of edges.

Ind. beg. $m = 0$. As G is connected: $n = 1$ and $r = 1$. Hence $n - m + r = 1 - 0 + 1 = 2$. ✓

Ind. step. $m \to m + 1$. Let G be a planar, connected graph with $m + 1$ edges.

Case 1. G is a tree. Choose leaf v. Let u be the parent of v. Remove v and the edge $\{v, u\}$ in G. The resulting graph G' has m edges and is connected and planar. By IH we have $n(G') - m(G') + r(G') = 2$. Further it holds that $n(G') + 1 = n(G)$, $m(G') + 1 = m(G)$, and $r(G) = r(G') = 1$. Together

$$n(G) - m(G) + r(G) = n(G') + 1 - m(G') - 1 + r(G') = 2. \quad ✓$$

Case 2. G is not a tree. Then G must consist of a cycle. Choose an edge $\{u, v\}$ on this cycle. G' results from G by removing $\{u, v\}$. Then it holds $m(G') = m(G) - 1$, G' is planar and connected. Further we know that $n(G') = n(G)$ and $r(G') = r(G) - 1$. From IH we have $n(G') - m(G') + r(G') = 2$ and therefore also

$$\begin{aligned} n(G) - m(G) + r(G) &= n(G') - m(G') - 1 + r(G') + 1 \\ &= n(G') - m(G') + r(G') = 2. \quad ✓\square \end{aligned}$$

Corollary 1.19.
Let $G = (V, E)$ be a planar graph with $n \geq 3$ vertices. Then it holds that $m \leq 3n - 6$.

Proof. W.l.o.g. let G be connected. As $n \geq 3$ every region of G is surrounded by a least 3 edges, i.e., $d(r) \geq 3$ for every region r. Then it holds that

$$2 \cdot m = \sum_r d(r) \geq 3 \cdot r \Leftrightarrow r \leq \frac{2 \cdot m}{3}.$$

With the help of the polyhedron formula we get

$$m = n + r - 2 \leq n + \frac{2 \cdot m}{3} - 2 \Leftrightarrow \frac{1}{3} m \leq n - 2,$$

and finally $m \leq 3 \cdot n - 6$. $\square$

Corollary 1.20.
K_5 *is not planar.*

Proof. We know that $n(K_5) = 5$ and $m(K_5) = 10$. As $10 \not\leq 3 \cdot 5 - 6 = 9$ it follows that K_5 is not planar. $\square$

Corollary 1.21.
$K_{3,3}$ *is not planar.*

Proof. Assume that $K_{3,3}$ is planar. Then it must obey the polyhedron formula. Hence we get from $n(K_{3,3}) = 6$ and $m(K_{3,3}) = 9$ that the number of regions must be $9 + 2 - 6 = 5 = r(K_{3,3}) =: r$.

Lets count the number of edges around any region. Start at some vertex $v \in U$. Now we want to walk along the edges to reach v. This must be possible because we assumed $K_{3,3}$ to be planar. It cannot be one edge because we do not have loops. And it cannot be two edges because this would imply multi-edges. Also it cannot be three edges, because after two edges we turn back to U as we are bipartite. So no region can be surrounded by three edges.

The minimum number of edges required is four. If there are at least four edges per region then the number of times an edge is counted as the border of a region is at least $4 \cdot r$.

Any edge can only sit between two regions. Hence each edge can only count twice as a region frontier, so there are at least $\frac{4r}{2} = 2r$ edges.

We know that $r = 5$ which means we need at least $2 \cdot 5 = 10$ edges. However $m(K_{3,3}) = 9$ which leads to the contradiction. □

Definition. *An edge contraction is an operation which removes an edge from a graph while simultaneously merging the two vertices it used to connect. A graph* G *is a* minor *of a graph* H *if* G *is isomorphic to a subgraph from* H *constructed by edge contractions.*

Definition. *The subdivision of some edge* $e = \{u, v\}$ *yields a graph containing one new vertex* w *with an edge set replacing* e *by the two new edges* $\{u, w\}$ *and* $\{w, v\}$*.*

A subdivision of a graph G *(or sometimes expansion) is a graph resulting from the subdivision of edges in* G*.*

A graph H *is called a topological minor of a graph* G *if a subdivision of* H *is isomorphic to a subgraph of* G*.*

Theorem 1.22 (Wagner's Conjecture, Theorem of Robertson & Seymour).
Given an infinite countable list $G_1, G_2, \ldots$ *of finite graphs, then there always exist two indices* $i < j$ *such that* G_i *is a minor of* G_j*.*

Theorem 1.23 (Kuratowski 1930; Wagner 1937).
The following assertions are equivalent for graphs G*:*

1. G *is planar.*
2. G *contains neither* K_5 *nor* $K_{3,3}$ *as a minor.*
3. G *contains neither* K_5 *nor* $K_{3,3}$ *as a topological minor.*

1.9.3 Colorings of Maps

In this section we will turn towards the five color theorem which will be easier to prove than the four color theorem. At first we need to prove this lemma.

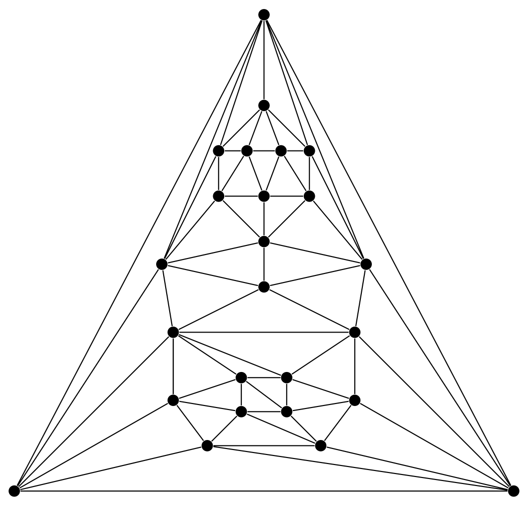


Figure 1.24: Counterexample of Heawood.

Lemma 1.24.
Let G *be a planar graph. Then in* G *there is a vertex with degree* ≤ 5*.*

Proof. For the number of vertices $n \leq 6$ this claim is obviously true. Let $n > 6$ and assume $\delta(G) \geq 6$. Then it holds

$$m = \frac{1}{2} \sum_{u \in V} \deg u \geq \frac{1}{2} \sum_{u \in V} 6 = 3n$$

contradicting Corollary 1.19. □

The gap in Kempes 4-color-proof in 1879 is settled in the previous lemma. He believed the validity of the lemma also for degree ≤ 4, i.e., he believed that every planar graph has a vertex of degree ≤ 4. However Lemma 1.24 cannot be improved to this as shown by the counterexample of Heawood shown in Figure 1.24. A function $f \colon V \to \mathbb{N}$ is a coloring function. A graph $G = (V, E)$ is c-colorable iff there exists a coloring function f such that for every edge $(u, v) \in E$ it holds that $f(u) \neq f(v)$ and there are at most c different values of $f(v)$ for all $v \in V$.

Still Kempe's argumentation leads together with Lemma 1.24 to the following theorem.

Theorem 1.25 (5-Color-Theorem).
Any planar graph is 5-colorable.

Proof. Induction on $n(G)$.

Induction beginning. $n \leq 5$ clear. ✓

Induction step. $n \to n + 1$: Let G be a planar graph with $n + 1$ vertices. From Lemma 1.24 G has a vertex v with $\deg v \leq 5$. Let G' be the graph resulting from G by removing v. By IH G' is 5-colorable.

Case 1: $\deg \nu < 5$. Then one color remains for ν because the neighbors of ν only need ≤ 4 colors. Hence ν gets this color and G is therefore 5-colorable.

Case 2: $\deg \nu = 5$. Let $u_1, \dots, u_5$ the neighbors of ν. We need to distinguish again two cases:

(a) $u_1, \dots, u_5$ are colored in G' with at most 4 colors. Then ν can get the free color.

(b) $u_1, \dots, u_5$ all need different colors. W.l.o.g. assume that u_i is colored with color i (in the coloring of G'). Let $G_{i,j}$ be the subgraph of G which is induced by all vertices with colors i and j. We need to differ again two cases:

   (i) In $G_{1,3}$ exists *no* path from u_1 to u_3. Then the colors 1 and 3 in the CC of u_1 in $G_{1,3}$ can be switched. By this we color u_1 with 3 and ν can be colored with color 1.

   (ii) In $G_{1,3}$ exists *a* path from u_1 to u_3. Then there cannot be a path from u_2 to u_4 in $G_{2,4}$ due to planarity. As in the previous case we can switch the colors 2 and 4 in the CC of u_2 in $G_{2,4}$. By this u_2 gets color 4 and ν is colored with 2.

$\square$

1.9.4 List Coloring

In this section we will follow Diestel's book about graph theory [Diestel, 2005]. Here we will consider a recent generalization of colorings. Suppose we are given a graph $G = (V, E)$ and for each vertex of G we have a list of its allowed colors at this particular vertex. When can we color G so that each vertex receives only a color from its list?

Definition. *Let $(S_\nu)_{\nu \in V}$ be a family of sets. We call a vertex coloring c of G with $c(\nu) \in S_\nu$ for all $\nu \in V$ a coloring from the list S_ν. The graph G is called* k-list-colorable *or* k-choosable *if for every family $(S_\nu)_{\nu \in V}$ with $|S_\nu| = k$ for all ν there is a vertex coloring of G from the lists S_ν. The least integer k for which G is k-choosable is the* list-chromatic number *or the* choice number *$ch(G)$ of G.*

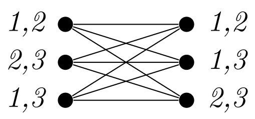
Principally, showing that a given graph is k-choosable is more difficult than proving it to be k-colorable: the latter is just a special case of the former where all lists are equal to $\{1, \dots, k\}$. Hence

$$\mathrm{ch}(G) \geq \chi(G)$$

is true for all graphs G.

A first intuition between the correlation of list-coloring and usual coloring could be that every graph which is k-colorable is as well k-choosable. However this intuition is wrong as can be seen by the following example (thank you to Fabian and Martin for the example!).

Example. *The graph $K_{3,3}$ is 2-colorable, as it is bipartite. Though it is not 2-choosable as the following counter-example visualizes:*



If the first left is 1, then the first right is 2 and second right is 3. Hence no color is available for the second left. If the first left is 2, then the first right is 1 and the last right is 3. Hence no color is available for the third left.

Definition. *Let $G = (V, E)$. A plane graph G is called* maximally plane*, or just* maximal *if we cannot add a new edge to form a plane graph $G' \supsetneq G$ with $V(G') = V(G)$.*

We say G is a plane triangulation if every region in G is bounded by a triangle.

An edge which joins two vertices of a cycle but is not itself an edge of the cycle is a chord *of that cycle.*

Lemma 1.26.
A plane graph of at least 3 vertices is maximally plane iff it is a plane triangulation.

Proof. Not proven. □

Theorem 1.27 (Thomassen 1994).
Every planar graph is 5-choosable.

Proof. We will prove the following proposition for all plane graphs $G = (V, E)$ with at least 3 vertices:

Proposition. *Suppose that every inner region of G is bounded by a triangle and its outer region by a cycle $C = v_1, \dots, v_k, v_1$. Suppose further that v_1 has already been colored with the color 1, and v_2 with color 2. Finally assume that with every other vertex of C a list of at least 3 colors is associated, and with every vertex of $G \setminus C$ a list of at least 5 colors. Then the coloring of v_1 and v_2 can be extended to a coloring of G from the given lists.*

First we need to verify that the proposition implies the theorem. Let any planar graph be given, together with a list of 5 colors for each vertex. At first add edges to this graph until we have a maximal planar graph G. Now by Lemma 1.26 G is a plane triangulation. Let $v_1 v_2 v_3 v_1$ be the boundary of its outer region. Now we will color v_1 and v_2 differently from their lists and extend this color with the help of the proposition to a list-coloring of G.

Proof (of proposition). We will prove the result by induction on $||V||$. If $||V|| = 3$ then $G = C$ and the assertion is trivial.

Induction step. Now let $||V|| \geq 4$.

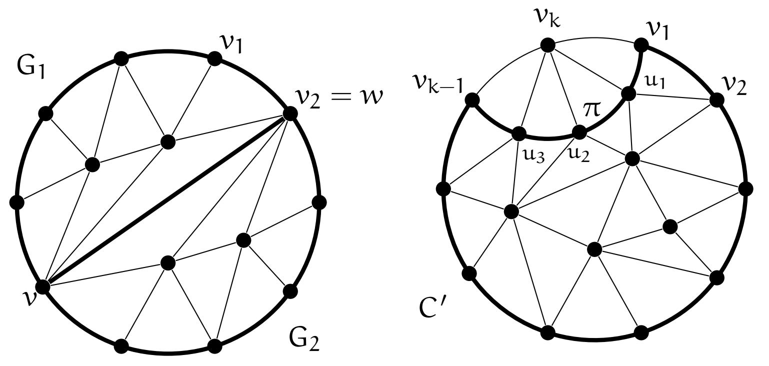


Figure 1.25: Induction step of the proof of Theorem 1.27 with a chord (left) and without a chord (right).

C **has a chord:** Then this chord vw lies on two unique cycles $C_1, C_2 \subseteq C + vw$ with $v_1v_2 \in C_1$ and $v_1v_2 \notin C_2$. For $i = 1, 2$ let G_i denote the subgraph of G induced by the vertices lying on C_i or in its inner region. Apply IH to G_1 and then (v and w have new colors now) to G_2. This will yield the desired coloring of G.

C **has no chord:** Let $v_1, u_1, u_2, \dots, u_m, v_{k-1}$ be the neighbors of v_k in their natural cyclic order around v_k. By definition of C all the u_i's lie in the inner region of C. As the inner regions of C are bounded by triangles, $\pi := v_1, u_1, u_2, \dots, u_m, v_{k-1}$ is a path in G and $C' := \pi \cup (C \setminus v_k)$ is a cycle.

Now we just choose two different colors $j, \ell \neq 1$ from v_k's list and delete these colors from the lists of each vertex u_i. Then every list of a vertex on C' has at least 3 colors so by IH we may color C' and its interior $G \setminus v_k$. At least one of the two colors j, ℓ is not used for v_{k-1} and we may assign that color to v_k. □

□

2 Parallel Algorithms

Until this point we have investigated solely sequential algorithms. Criteria to compare them are required space and time. Also it might be possible to compare simplicity of implementation. Now we want to turn towards parallel algorithms. For this type of algorithms we can introduce other parameters which allow us to compare them more deeply. That are, number of processors, size of local storage, net topology, and communication concepts.

Let P be an arbitrary problem, and n be the size of the input. The (optimal) sequential time complexity of P let denote us by $T^*(n)$, i.e., there is a sequential algorithm solving an instance of P for input size n in $T^*(n)$ time units and one can show that no sequential algorithm solves P faster than $T^*(n)$.

Let A be a parallel algorithm which solves P in time $T_p(n)$ on a parallel computer with $p \in \mathbb{N}$ processors. Then the *speedup* of A is defined as

$$\mathrm{sp}(n) = \frac{T^*(n)}{T_p(n)}.$$

Theorem 2.1.
The speedup of a parallel algorithm cannot be larger than the number of processors, i.e., the p-times time of a parallel algorithm can never be less than the time of the serial algorithm:

$$sp(n) \leq p \qquad T^*(n) \leq p \cdot T_p(n).$$

Proof. If we simulate the computations of the p processors step wisely on a one processor machine, then we get a new sequential algorithm running in $p \cdot T_p(n)$ steps. Clearly this cannot be faster than $T^*(n)$ as $T^*(n)$ is already the optimal runtime. □

We are interested in algorithms whose speedup is approximately p, hence a *linear speedup*:

$$\mathrm{sp}(n) = \frac{T^*(n)}{T_p(n)} = p \Longrightarrow T^*(n) = p \cdot T_p(n).$$

Another measure of performance is *efficiency*:

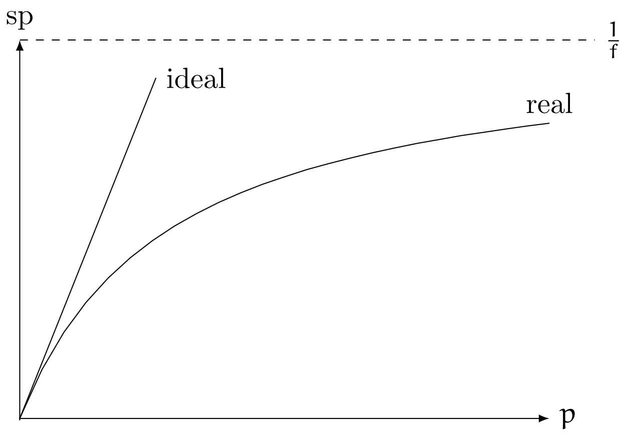
$$E_p(n) = \frac{T^*(n)}{p \cdot T_p(n)} = \frac{1}{p} \cdot \mathrm{sp}(n) \leq 1.$$

2.1 Amdahl's Law

Let $0 \leq f \leq 1$ be the fraction of operations in an algorithm which has to be done sequentially. If one uses a parallel computer with $p \in \mathbb{N}$ processors, then the maximum speedup is

$$\mathrm{sp}(n) \leq \frac{T^*(n)}{f \cdot T^*(n) + \frac{(1-f)\cdot T^*(n)}{p}} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f},$$

e.g., for 1% of sequential operations the speedup for a computer with $p = 10$ processors $\leq 9{,}17$ and for $p = 100$ processors we get a speedup of $\leq 50{,}25$.



Observe that this approach does not consider processor caches which may allow to store the complete problem instance in it if enough processors are used. This would result into a super-linear speedup. However the maximum speedup bound is still valid.

2.1.1 Sieve of Eratosthenes

This is a usual approach in finding all prime numbers $\leq n$. The algorithm terminates if the chosen number is $> \sqrt{n}$ and does not consider synchronization problems or inefficiencies (e.g., two processors work on the same non-prime).

If we scale on the time required to mark a field in the array, and if there are k prime numbers $\pi_1, \ldots, \pi_k \leq \sqrt{n}$, then a single processor spends

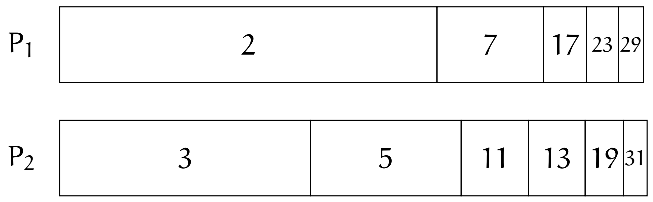
$$\left\lfloor \frac{n-\pi_1}{\pi_1} \right\rfloor + \left\lfloor \frac{n-\pi_2}{\pi_2} \right\rfloor + \cdots + \left\lfloor \frac{n-\pi_k}{\pi_k} \right\rfloor = \left\lfloor \frac{n-2}{2} \right\rfloor + \left\lfloor \frac{n-3}{3} \right\rfloor + \cdots + \left\lfloor \frac{n-\pi_k}{\pi_k} \right\rfloor$$

time units.

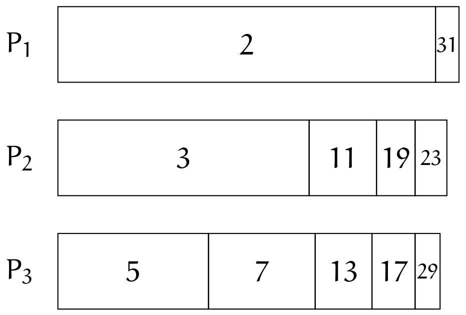
If $n = 1000$ we get

primes π_i	2	3	5	7	11	13	17	19	23	29	31
$\left\lfloor \frac{1000-\pi_i}{\pi_i} \right\rfloor$	499	332	199	141	89	75	57	51	42	33	31

Hence the sum corresponding to the time required using only a single processor is 1549 time units. If we consider the case for 2 processors we get:



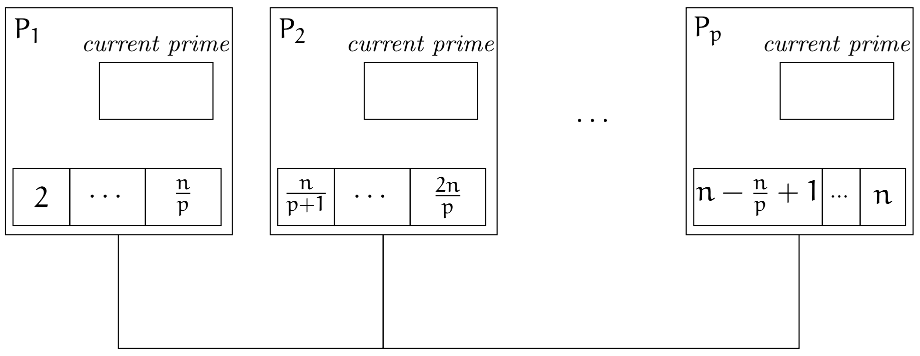
As speedup we get $\frac{1549}{777} \approx 2$. For three processors we get an improvement of



Thus a speedup of $\frac{1549}{530} \approx 2{,}92$. For more than three processors the speedup stays at $\frac{1549}{499} \approx 3{,}1$.

2.1.2 A Better Approach

In the following we will not use a shared memory. However, now we communicate through messages from processor to processor. Every processor will be responsible for $\leq \left\lceil \frac{n}{p} \right\rceil$ numbers. If $p < \sqrt{n}$ then all primes $\leq \sqrt{n}$ are in the field controlled by processor P_1. P_1 will send the next prime number π to P_2 up to P_p. Every processor then deletes the multiples from his array. Therefore we will consider the time to mark a number μ and the communication time λ.



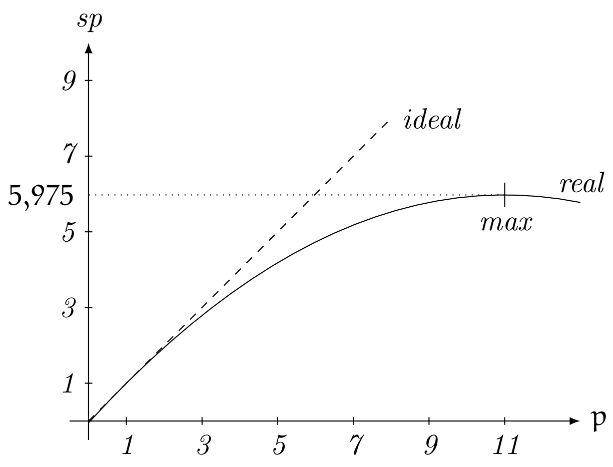
communication channel

There are k primes $\leq \sqrt{n}$ denoted by $\pi_1, \ldots, \pi_k$. Thus every processor requires at most

$$\left(\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{2} \right\rceil + \left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{3} \right\rceil + \cdots + \left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{\pi_k} \right\rceil \right) \cdot \mu$$

to mark the primes and $k(p-1)\lambda$ time units for sending messages.

Example. $\lambda = 100\mu s, n = 10^6$, *there are* $k = 168$ *primes* $\leq \sqrt{10^6} = 1000$ *and* $\pi_k = 997$.



2.2 Important concepts of parallel computers

Parallel Random Access Machine (PRAM)

These machines have the following properties:

1. Every processor is a usual random access machine (RAM) with local memory for local data and the program.
2. The instructions correspond to the ones of a usual RAM.
3. The processors are connected through a shared memory. The data which is stored in shared memory is called *global data*.
4. Every processor is identified by its processor number $0 \leq p < n$. These identifiers as well as the total number of processors is known by every processor.

A PRAM runs synchronously, i.e., all processors run under the control of a global clock frequency.

Example. *Consider the following code:*

```
if $p \leq \frac{n}{2}$ then part $i$;
else part $ii$;
locvar := 0;
```

*All processors run in parallel the **if**-condition. The first half will run part* i *and the second half will run part* ii *(in parallel!). Then all processors will set their local variable **locvar** in parallel to* 0.

In order to clearly negotiate the read and write rights for the shared memory there exist three common versions of PRAMs:

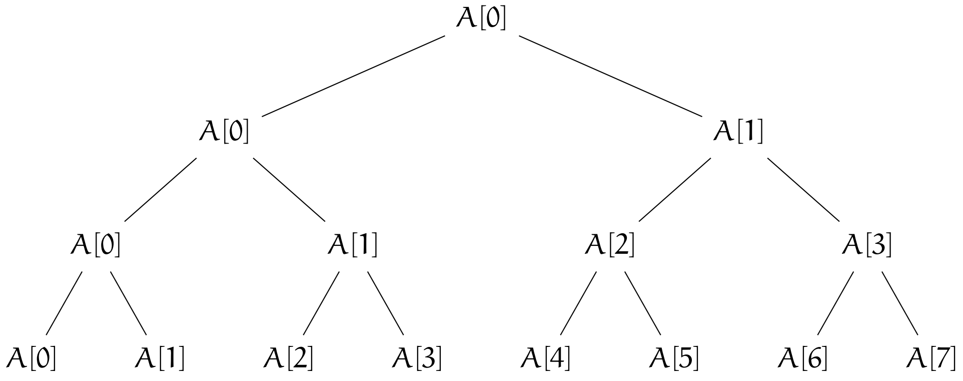
EREW-PRAM (exclusive read, exclusive write). Neither parallel read, nor parallel write on the same memory cell is allowed.

CREW-PRAM (concurrent read, exclusive write). All processors may parallel read a memory cell, however parallel write on the same memory cell is not allowed.

CRCW-PRAM (concurrent read, concurrent write). All processors may parallel read a memory cell. For possible writing conflicts there exist three other rules:

- common CRCW-PRAM: All processors that write to the same memory cell have to write the same (standard interpretation).
- priority CRCW-PRAM: A set distinct priority for the processors decides which processor is allowed to write.
- arbitrary CRCW-PRAM: Randomly one of the processors is allowed to write.

Example. *An array of length* $m = 2^k$ *is stored in the shared memory of an EREW-PRAM. The task is to compute the sum* $S = A[0] + \cdots + A[m-1]$ *with* $n = \frac{m}{2}$ *processors.*
Idea: $m = 2^3 = 8, n = 4$*:*



Algorithm 2.1: sum ⟨EREW-PRAM⟩.

```
global A: array[0, ..., m − 1] of real; S: integer; m: integer;
local n, p: integer;
for h = 1 to log(2n) do
    if $p < \frac{2n}{2^h}$ then A[p] := A[2p] + A[2p + 1] ;
if p = 0 then S := A[0] ;
```

Observations:

- *As* m *is a global integer and the model is an EREW-PRAM we cannot use a for loop from* $h = 0$ *to* $\log m - 1$.

- *'$A[p] := A[2p] + A[2p+1]$' is interpreted as a sequence of read and write operations.*
- $S := A[0]$ *must only be done by a single processor.*
- *If we do not want to overwrite the array A we can use a blank array.*

The runtime of the algorithm is $O(\log m)$.

Example. *Multiplication of two matrices A, B with $m = 2^r$ for some $r \in \mathbb{N}$ on a CREW-PRAM with $m^3 = n$ processors. The product matrix C is then defined by*

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj}.$$

Idea: identify a processor with the triple (i, j, k) for $0 \leq i, j, k < m$.

Algorithm 2.2: matmul ⟨CREW-PRAM⟩

```
global $m$: integer;
global $A, B, C$: array$[0, ..., m-1][0, ..., m-1]$ of real;
global $D$: array$[0, ..., m-1][0, ..., m-1][0, ..., m-1]$ of real;
local $i, j, k$: integer, $n, p$: integer, $0 \leq p < n$;
Compute $(i, j, k)$ from $p$ and $m$;
$D[i, j, k] := A[i, k] \cdot B[k, j]$;
for $h = 1$ to $\log m$ do
    if $k < \frac{m}{2^h}$ then $D[i, j, k] := D[i, j, 2k] + D[i, j, 2k+1]$;
if $k = 0$ then $C[i, j] := D[i, j, 0]$ ;
```

Observations:

- *In line 6 m processors read in parallel $A[i, j]$. Therefore a CREW-PRAM is required.*
- *The **if**-condition in line 9 ensures that only one processors writes into $C[i, j]$.*
- *The runtime is $O(\log m)$.*

Definition. *Let $\mathcal{A}, \mathcal{B} \in \{C, E\}$, then $\mathcal{A}R\mathcal{B}W(p(n), t(n))$ is the set of all problems which can be solved by a $\mathcal{A}R\mathcal{B}W$-PRAM with $p(n)$ processors in $O(t(n))$ parallel steps.*

Example. *Computing the sum of elements of an array is in $\mathsf{EREW}(n, \log n)$. Matrix multiplication is in $\mathsf{CREW}(m^3, \log m)$.*

Theorem 2.2.
$\mathsf{EREW}(p(n), t(n)) \subseteq \mathsf{CREW}(p(n), t(n)) \subseteq \mathsf{CRCW}(p(n), t(n)) \subseteq \mathsf{SEQ}(p(n) \cdot t(n))$*, where $\mathsf{SEQ}(f(n))$ is the set of all problems solvable by a usual RAM in $O(f(n))$ steps.*

Remark. *One can show that the first two inclusions are strict.*

Definition. *Let $T_p(n)$ the time which an algorithm for problems of input size n requires on a parallel computer with $p(n)$ processors. Then we say to the product $C(n) = T_p(n) \cdot p(n)$ the* costs *of the algorithm. A parallel algorithm A is said to be* cost optimal *(or only optimal) if A's cost $C(n) = \Theta(T^*(n))$ where $T^*(n)$ is the time in which an optimal sequential algorithm on a RAM needs.*

Example. *For Algorithm 2.1 it holds*

$$C(n) = \frac{n}{2} \cdot \Theta(\log n) = \Theta(n \cdot \log n),$$

whence the algorithm is not optimal.

Definition. *The* work $W(n)$ *which a parallel algorithm for a problem input of size n requires is the sum of the runtimes of each processor used in the algorithm.*

Example. *For Algorithm 2.1 we get*

$$W(n) = \Theta\left(\frac{n}{2} + \frac{n}{4} + \cdots + 1\right) = \Theta(n).$$

2.3 Theorem of Brent

Theorem 2.3.
The cost $C(n)$ of a parallel algorithm are always bounded by the serial complexity:

$$T^*(n) \leq C(n).$$

Proof. From Theorem 2.1 we have

$$\mathrm{sp}(n) = \frac{T^*(n)}{T_p(n)} \leq p$$

hence we have $T^*(n) \leq p \cdot T_p(n) = C(n)$. □

Theorem 2.4.
The costs $C(n)$ of a parallel algorithm are at least in the order of the work: $W(n) \leq C(n)$.

Proof. Let $W_i(n)$ the number of (parallel) steps which are executed in step i of the algorithm, i.e., $W(n) = \sum_{i=1}^{T_p(n)} W_i(n)$. In a single step the number of operations are bounded by the number of processors: $W_i(n) \leq p$. Hence

$$W(n) = \sum_{i=1}^{T_p(n)} W_i(n) \leq \sum_{i=1}^{T_p(n)} p = p \cdot T_p(n) = C(n).$$

□

Theorem 2.5 (Brent).
A parallel algorithm with p *processors running in time* $T_p(n)$ *and having a work of* $W(n)$ *can be executed on* $p' < p$ *processors in time*

$$T_{p'}(n) \leq T_p(n) + \left\lfloor \frac{W(n)}{p'} \right\rfloor .$$

Proof. In step i of the algorithm p processors do $W_i(n)$ operations. With p' processors we need for these $W_i(n)$ operations at most

$$t^i_{p'}(n) \leq \left\lceil \frac{W_i(n)}{p'} \right\rceil$$

steps. In order to execute the complete workload $W(n) = \sum_{i=1}^{T_p(n)} W_i(n)$ we need to track all steps of the p-processors PRAM:

$$\begin{aligned} T_{p'}(n) = \sum_{i=1}^{T_p(n)} t^i_{p'}(n) \leq \sum_{i=1}^{T_p(n)} \left\lceil \frac{W_i(n)}{p'} \right\rceil \leq \sum_{i=1}^{T_p(n)} \left\lfloor \frac{W_i(n)}{p'} \right\rfloor + 1 \\ \leq \left\lfloor \frac{1}{p'} \cdot \sum_{i=1}^{T_p(n)} W_i(n) \right\rfloor + T_p(n) = \left\lfloor \frac{W(n)}{p'} \right\rfloor + T_p(n). \end{aligned}$$

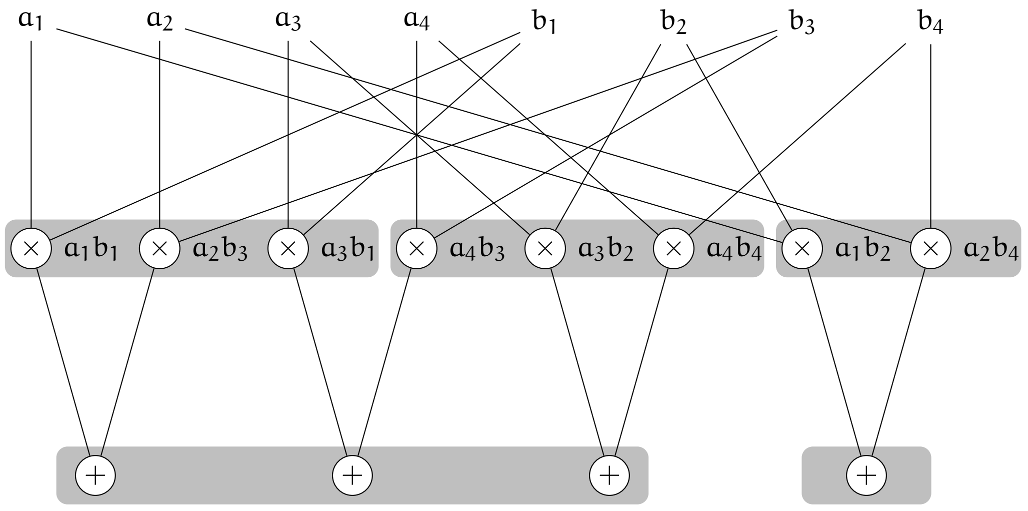
□

Definition. *A computation graph is a directed acyclic graph (dag), where the vertices correspond to the input-, output-values, and the operations. The edges encode the dependencies between the operations and operands.*

The depth of a vertex is the maximum distance to an input vertex. The depth of the computation graph is the maximum depth of all vertices. The size of the computation graph is the number of operations.

Note that loops are unfurled unless the number of runs is unpredictable. In that case the algorithm cannot be described by computation graphs.

Example (Application of Brent's theorem to computation graphs). *Multiplication of two* 2×2 *matrices.*



Depth is 2, size is 12. With 3 processors 5 steps are required.

Theorem 2.6 (Brent's theorem for computation graphs).
Every computation graph with depth t and size w with constant in-degree for every vertex can be solved by an n-processor CREW in $O(\frac{w}{n} + t)$ steps.

Proof. The input values are in the global memory of the PRAM. For every operator and every output vertex a storage in the global memory is used. One processor of the PRAM can compute the value of an operator vertex in $O(1)$. Reading of the operands in $O(1)$, operation and saving $O(1)$. □

Theorem 2.7 (Eckstein).
An EREW-PRAM with p processors can simulate a priority-CRCW-PRAM with a runtime blowup of $O(\log p)$.

Proof. In order to simulate the concurrent read of the CRCW one designated processor reads the cell and distributes the information tree-like–every processor transfers the information to two others processors. In that way all processors can be notified in $\log p$ steps.

For the concurrent write the tree is processed bottom-up from the leaf level. Every processor communicates with his neighbors about the value which has to be written. After $\log p$ steps the final value is determined. □

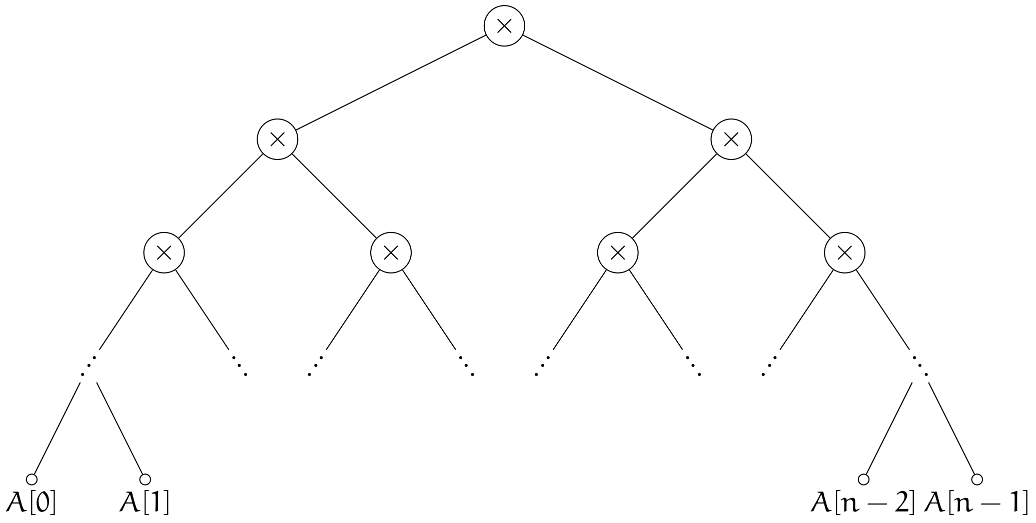
2.4 Fundamental techniques of parallel algorithms

In the following subsection we will visit several important techniques used by parallel algorithms. Our underlying theoretical model will be the parallel random access machine PRAM defined in the previous section.

2.4.1 The method of balanced binary trees

The underlying principle is the construction of the tree by its input and requires possibly an extension of the input to a power of two. Then one runs bottom-up from the leafs to the root.

Example. *Given an array* $A[0,\dots,n-1]$ *of elements,* $n = 2^k$. *Associative operation* $\times$ *on the elements. Task: compute* $\bigtimes_{i=0}^{n-1} A[i]$:



After $\log n$ *steps the root is reached. During the computation at most* $\frac{n}{2}$ *processors are simultaneously busy. If* $\times = +$ *then we showed that it is in* $\mathsf{EREW}(n, \log n)$ *(see Algorithm 2.1). The algorithm is not optimal as*

$$C(n) = \frac{n}{2}\log n = \Theta(n \log n),$$

and the naive sequential algorithm runs in $\Theta(n)$ *steps.*

Another technique leads to a more efficient algorithm. We divide the array into N *(number of processors) segments* $S[i]$ *of length* $\leq \lceil \log n \rceil$, *where* $N = \left\lceil \frac{n}{\log n} \right\rceil$. *This division can be done in* $O(1)$ *steps if all processors know* n. *Otherwise in* $O(\log N)$ *steps.*

Then processor P_i *(*$0 \leq i < N$*) computes* $\bigtimes_{j \in S[i]} a[j]$ *in* $O(\log n)$ *steps. Afterwards the result will be determined by the technique of the balanced tree. Hence* $\Theta\left(\log\left(\frac{n}{\log n}\right)\right)$ *steps with*

$$\Theta\left(\log\left(\frac{n}{\log n}\right)\right) = \Theta(\log n - \log\log n) = \Theta(\log n).$$

Thus the problem is in $\mathsf{EREW}(\frac{n}{\log n}, \log n)$. *This algorithm is optimal as*

$$C(n) = \Theta\left(\frac{n}{\log n} \cdot \log n\right) = \Theta(n).$$

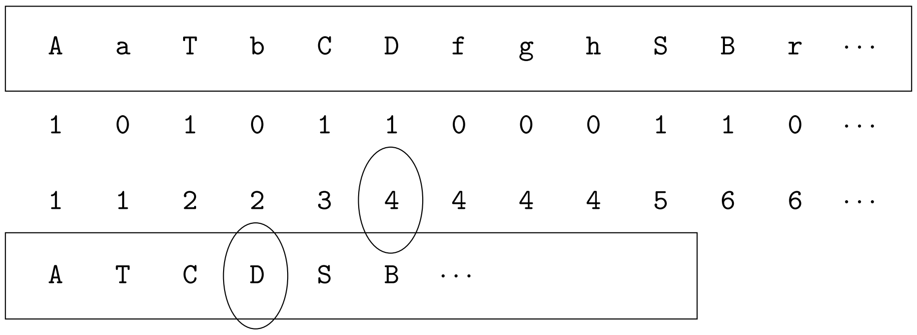
2.4.2 A parallel algorithm to compute all prefix sums

Assume n is a power of two. Given an array $A[0,\dots,n-1]$ and an associative operation $\times$ on the elements of A. We search for the field $S[0,\dots,n-1]$ with

$$S[j] = \bigtimes_{i=0}^{j} A[i]$$

for $0 \le j < n$. The sequential algorithm solves the problem in $\Theta(n)$ steps.

Possible application: given an array with upper and lower case letters. The task is to return the array without the lower case letters and no gaps in between.



Idea: Use balanced tree to compute the partial sums bottom up and then compute the prefix sums top down.

Algorithm 2.3: prefixsum $\langle$CREW$\rangle$

```
1 global A: array[0,...,n − 1] (input, n power of two),
           S: array[0,...,n − 1] (prefix sums),
           Y, Z: array[0,...,n − 1] (auxiliary arrays);
2 local p, n: integer (0 ≤ p < n);
3 if n = 1 and p = 0 then S[0] := A[0] and terminate ;
4 if p < n/2 then Y[p] := A[2·p] × A[2·p + 1] ;
5 Compute recursively the prefix sum of Y[0,...,n/2 − 1] and store the result in Z[0,...,n/2 − 1];
6 if p < n then
7     if p = 0 then S[0] := A[0] ;
8     else if p ≡₂ 0 then S[p] := Z[(p−2)/2] × A[p] ;
9     else S[p] := Z[(p−1)/2] ;
```

Notation: $\langle i,j\rangle_A$ abbreviates $\bigtimes_{r=i}^{j} A[r]$. Obviously it holds that $\langle i,i\rangle_A = A[i]$ and $\langle i,k\rangle_A = \langle i,j\rangle_A \times \langle j+1,k\rangle_A$ for $j \in [i,k-1]$

Claim. *Algorithm 2.3 computes correctly the prefix sum.*

Proof. Induction on k with $n = 2^k$.

Induction beginning. $k = 0$ hence $n = 1$. ✓

$$\begin{tikzcd}
a_0 \arrow[dr] \arrow[ddd] & a_1 \arrow[d] & a_2 \arrow[dr] \arrow[ddd] & a_3 \arrow[d] & a_4 \arrow[dr] \arrow[ddd] & a_5 \arrow[d] & a_6 \arrow[dr] \arrow[ddd] & a_7 \arrow[d] \\
& y_0 & & y_1 & & y_2 & & y_3 \arrow[d, dashed, bend left, "\text{recursive computation}"] \\
& z_0 \arrow[d] \arrow[dr] & & z_1 \arrow[d] \arrow[dr] & & z_2 \arrow[d] \arrow[dr] & & z_3 \arrow[d] \\
S_0 & S_1 & S_2 & S_3 & S_4 & S_5 & S_6 & S_7
\end{tikzcd}$$

Figure 2.1: Computation of prefix sums as in Algorithm 2.3. a_i, y_i, z_i, S_i denote the arrays in the algorithm.

Induction step. The algorithm works correctly for $n = 2^k$. Consider $n = 2^{k+1}, k > 0$. At first we compute

$$Y[i] = A[2i] \times A[2i+1] = \langle 2i, 2i+1 \rangle_A \qquad \text{for } 0 \leq i \leq \frac{n}{2}.$$

The recursive call gets by IH:

$$Z[j] = \bigtimes_{i=0}^{j} Y[i] = \langle 0, j \rangle_Y = \langle 0, 2j+1 \rangle_A.$$

Further it is $S[0] = A[0]$ and for odd i it holds

$$S[i] = Z\left[\frac{i-1}{2}\right] = \langle 0, i \rangle_A.$$

For even i we get

$$S[i] = Z\left[\frac{i-2}{2}\right] \times A[i] = \langle 0, i-1 \rangle_A \times \langle i, i \rangle_A = \langle 0, i \rangle_A \quad \checkmark$$

$\square$

The runtime of the algorithm is $\Theta(\log n)$ wherefore the problem is in CREW$(n, \log n)$.

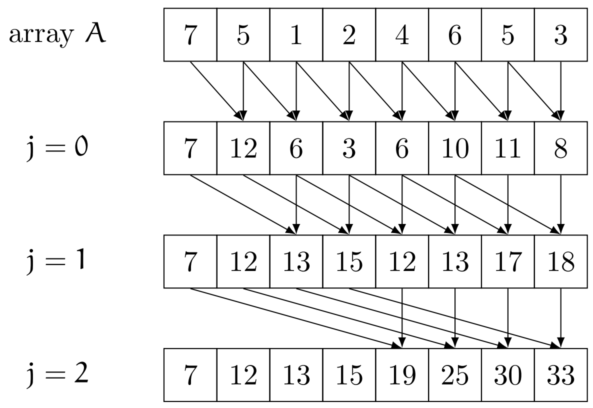
Note that it is possible to computed the prefix sum on a EREW-PRAM in $O(\log n)$ steps.
An alternative non-recursive implementation gets to Algorithm 2.4.
The invariant for Algorithm 2.4 says:

- If $i \leq 2^j - 1$ then $A[i] = \langle 0, i \rangle_A$ ($A[i]$ is the prefix sum from 0 to j), and
- if $i > 2^j - 1$ then $A[i] = \langle i - 2^j + 1, i \rangle_A$ ($A[i]$ is the sum of elements $i - 2^j + 1$ to i, hence the prefixsum up to i where the elements 0 to $i - 2^j$ are missing).

Algorithm 2.4: prefixsumB $\langle\mathsf{EREW}\rangle$

```
global $A$: array$[0,\ldots,n-1]$ ;            // $n$ is not required to be a power of 2
local $p, n, j$: integer;
for $j := 0$ to $\lceil \log n \rceil - 1$ do
    ⋆⋆⋆ Invariant ⋆⋆⋆
    if $2^j \le p < n$ then $A[p] := A[p] \times A[p-2^j]$ ;
```

If $j = 0$ then $A[i] = \langle i, i\rangle_A$ for $i \geq 1$ and for $j = 1$ we get $A[i] = \langle i-1, i\rangle_A$ if $i \geq 3$.



Claim. *The invariant is correct.*

Proof. IB. Let $j = 0$ then $A[i] = \langle i, i\rangle_A$ for $i \geq 0$. ✓

IS. $j \to j+1$. Let $p \geq 2^j$. Then (in parallel) we compute $A[p] = A[p] \times A[p-2^j]$. If $p \leq 2^{j+1}-1$, then

$$\begin{aligned} A[p] &= \underbrace{\langle p-2^j+1, p\rangle_A}_{\textbf{IH.}} \times \underbrace{\langle 0, p-2^j\rangle_A}_{\textbf{IH.}} \\ &= \langle 0, p\rangle_A \quad \checkmark \end{aligned}$$

If $p > 2^{j+1}-1$ then

$$\begin{aligned} A[p] &= \underbrace{\langle p-2^j+1, p\rangle_A}_{\textbf{IH.}} \times \underbrace{\langle (p-2^j)-2^j+1, p-2^j\rangle_A}_{\textbf{IH.}} \\ &= \langle p-2^{j+1}+1, p\rangle_A \quad \checkmark \end{aligned}$$

□

Observation 7.
Algorithm 2.4 requires $\Theta(\log n)$ *steps and* $(n-2^0)+(n-2^1)+\cdots+(n-2^{\lfloor \log n \rfloor}) = \Theta(n \cdot \log n - n) = \Theta(n \cdot \log n)$ *operations.*

2.4.3 The technique of pointer jumping

In order to demonstrate the technique of pointer jumping we will introduce a new problem first.

> In parallel algorithms, the list ranking problem involves determining the position, or rank, of each item in a linked list. That is, the last item in the list should be assigned the number 1, the second-to-last item in the list should be assigned the number 2, etc. Although it is straightforward to solve this problem efficiently on a sequential computer, by traversing the list in order, it is more complicated to solve in parallel. As [Anderson and Miller, 1990] write, the problem was viewed as important in the parallel algorithms community both for its many applications and because solving it led to many important ideas that could be applied in parallel algorithms more generally.
>
> —`http://en.wikipedia.org/wiki/List_ranking`

More formally:

Problem List Ranking

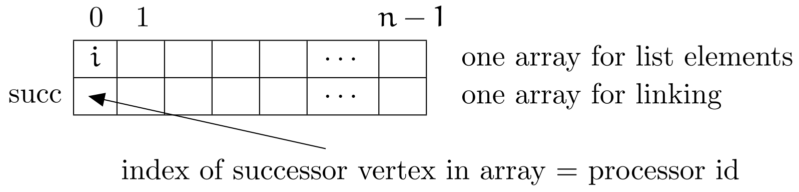
Given: a list stored in the memory of a PRAM.

Task: compute the distance of every element of the list to the end of the list, i.e., for every element i we need to compute the value

$$d[i] := \begin{cases} 0 & \text{, if } i \text{ is the last element of the list,} \\ d[\mathrm{succ}[i]] + 1 & \text{, otherwise} \end{cases}$$

where succ[i] is the successor of i in the list.

Possible implementation of the lists:


0
1
n − 1
i
· · ·
succ
· · ·
one array for list elements
one array for linking
index of successor vertex in array = processor id

Note that the $d[i]$-values can be then set in $\Theta(n)$.

Observation 8.
The algorithm works destructively on the list. If this not desired one must create a copy of the list in $O(1)$ *time.*

*The implicitly assumed synchronization enforces that all processors run synchronously after the **while** loop. But also the resetting of the **succ**-pointers must be done synchronously, i.e., read operations must be finished before write operations.*

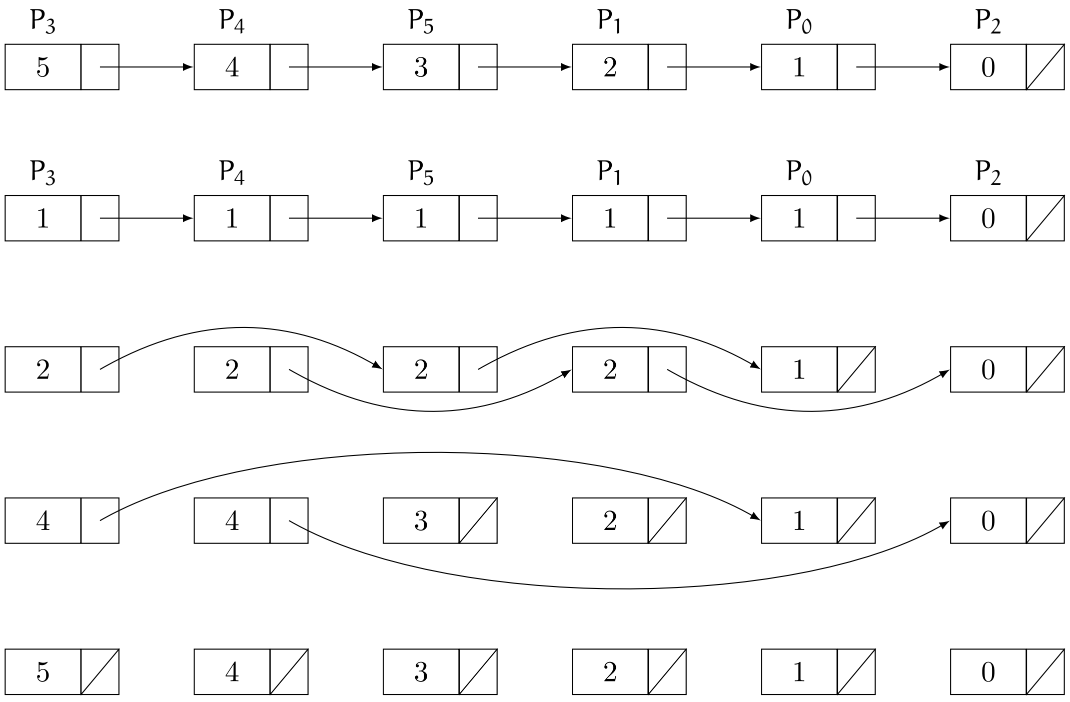


Figure 2.2: Example pointer jumping.

Claim. *The algorithm works correct.*

Proof. Consider the invariant: if one sums the d-values of all vertices of one sublist starting with p then one gets the distance p to the end of the original list. In every loop one part of the list is released but also added its distance. Hence the invariant stays valid. □

The initialization requires $O(1)$ steps, one loop requires $O(1)$ steps. As in every loop every list is split into two (almost) equal sized lists after pointer jumping and the procedure ends at lists of length of 1 the loop requires $O(\lceil \log n \rceil)$ steps.

Algorithm 2.5: listranking ⟨EREW⟩.

```
global d, succ: array of integer;
local p, N: integer ($0 \le p < N$);
if succ[p] = nil then d[p] := 0 ;
else d[p] := 1 ;
⋆⋆⋆ Invariant ⋆⋆⋆
while succ[p] ≠ nil do
    d[p] := d[p] + d[succ[p]];
    succ[p] := succ[succ[p]] (pointer jumping);
```

Hence the algorithm runs in EREW($n, \log n$) and is not optimal. But with the usual knack $N = \frac{n}{\log n}$ one can define an optimal algorithm.

2.4.4 Computing prefix sums through pointer jumping

Given an array A of length n with elements on which an associative operation $\times$ is defined. Compute $S[j] = \bigtimes_{i=0}^{j} A[i] = \langle 0, j\rangle_A$.

Algorithm 2.6: prefixsumpointer ⟨EREW⟩

```
1 global $A, S$: array[$0, \dots, n-1$], succ: array[$0, \dots, n-1$] of integer;
2 local $p, n$: integer;
  // Construction of the list
3 if $p = n - 1$ then  succ[p] = nil ;
4 else succ[p] = $p + 1$;
5 $S[p] := A[p]$;
  // Compute prefix-sum with pointer jumping
6 while succ[p] $\neq$ nil do
7     $S[succ[p]] := S[p] \times S[succ[p]]$;
8     succ[p] := succ[succ[p]];
```

Claim. *Algorithm 2.6 works correct.*

Proof. At the beginning of the `while`-loop the following invariant is true: The cpu with id p which is assigned to the kth element $0 \le k < n$ has assigned to the S-array the value $\langle \max\{0, k - 2^t + 1\}, k\rangle$ after t steps in the loop. The succ-array pointers to the $k + 2^t$th element in the list if $k + 2^t < n$ otherwise `nil`.

IB. $t = 0$ then $\langle \max\{0, k\}, k\rangle = \langle k, k\rangle$ end succ pointers to the $k + 1$st element unless $k = n - 1$.

IS. $t \to t + 1$. Let p be assigned the kth list element. If succ[p] $\neq$ `nil` then $k + 2^t < n$ (acc. to IH). At first the S-value of the $k + 2^t$th vertex is computed. This value is

$$\begin{aligned} &\langle \max\{0, k - 2^t + 1\}, k\rangle \times \langle \max\{0, (k + 2^t) - 2^t + 1\}, k + 2^t\rangle \\ =\ &\langle \max\{0, k - 2^t + 1\}, k + 2^t\rangle \\ =\ &\langle \max\{0, (k + 2^t) - 2^{t+1} + 1\}, k + 2^t\rangle \end{aligned}$$

The succ-array is set to the list element which succeeds the $k + 2^t$th vertex. This is the $(k + 2^t) + 2^t = k + 2^{t+1}$st vertex or `nil`.

For $t \ge \lceil \log n\rceil$ the value in the S-array of the kth list element is $\langle \max\{0, k - 2^t + 1\}, k\rangle = \langle 0, k\rangle$ and in the succ-array is `nil`. □

The runtime is $\Theta(\log n)$.

2.5 Decomposition techniques in parallel algorithms

We want to divide a problem into *independent* subproblems of approximately similar size (decomposition phase), then we want to solve the subproblems in parallel and construct a complete solution from the solutions of the subproblems (merge phase).

Therefore we will investigate sorting algorithms on PRAMs. Let Σ be a non-empty set of elements obeying an ordering $\leq$. Now we search for efficient algorithms which sort an arbitrary set S of n elements from Σ w.r.t. $\leq$. Sequentially the best sorting algorithms run in $O(n \cdot \log n)$ steps (Heapsort, Mergesort).

Algorithm 2.7: mergesort $\langle \mathsf{SEQ} \rangle$.

```
Input     : Sequence S
1 $S_1, S_2, R_1, R_2$ : sequences;
2 divide $S$ into two sequence $S_1$ and $S_2$;
3 $R_1 :=$ mergesort($S_1$);
4 $R_2 :=$ mergesort($S_2$);
5 return merge($R_1, R_2$);
```

2.5.1 A simple merge algorithm

Definition. *Let* $X = (x_1, \dots, x_n)$ *be a tuple of elements from* Σ*, and let* $z \in \Sigma$*. Now define the* rank of z in X *as*

$$\text{rank}\,(z, X) := |\{x_i \mid x_i < z\}|.$$

Let $Y = (y_1, \dots, y_s)$ *be a tuple of elements from* Σ*. Then define*

$$\text{rank}\,(Y, X) := (z_1, \dots, z_s),$$

where $z_i = \text{rank}\,(y_i, X)$ *for* $1 \leq i \leq s$.

Example. *Let* $\Sigma = \mathbb{Z}$*,* $X = (26, 45, -12, 8, -3, 14)$*, and* $Y = (-16, 11, 22)$*. Then we get* $\text{rank}\,(Y, X) = (0, 3, 4)$.

Now take A and B as two given tuples and assume that A and B are disjunct, i.e., no element occurs more than once. AB is then the tuple which is constructed by attaching B at A. If x is in AB then $\text{rank}\,(x, AB)$ is the index of x in the sorted sequence (index from 0). As $\text{rank}\,(x, AB) = \text{rank}\,(x, A) + \text{rank}\,(x, B)$ holds, one can solve the merge-problem of two sorted sequences if one knows $\text{rank}\,(A, B)$ and $\text{rank}\,(B, A)$.

If A is sorted and is $b_i \in B$ then $\text{rank}\,(b_i, A)$ can be determined by binary search in $O(\log |A|)$ steps. Hence two disjunct sorted sequences A and B can be merged on a CREW-PRAM with $|A| + |B|$ processors in $O(\log |A| + \log |B|)$ steps.

Algorithm 2.8: merge $\langle$CREW$\rangle$.

```
1  global A: array[0,…,n−1] ($n = 2^k$; the first sorted sequence is in A[0,…,$\frac{n}{2}-1$], the
   second sorted sequence is in A[$\frac{n}{2}$,…,n−1], the merge result is then in A afterwards);
2  local p, N: integer (N = n),
       high, low, i: integer, x: element;
3  if $p \geq \frac{N}{2}$ then                    // processor p works on the left sequence
4      high := $\frac{N}{2} - 1$;
5      low := 0;
6  else                                  // processor p works on the right sequence
7      high := N − 1;
8      low := $\frac{N}{2}$;
9  x := A[p];                                                // determine rank of x
10 repeat
11     ⋆⋆⋆ low ≤ high and A[low − 1] < x < A[high + 1], extend A on left/right by ±∞ ⋆⋆⋆
12     i := $\left\lfloor \frac{high+low}{2} \right\rfloor$;
13     if x < A[i] then high := i − 1 ;
14     else low := i + 1 ;
15 until low > high;
16 A[p + low − $\frac{N}{2}$] := x;                                 // it holds x < A[low]
```

Observation 9.
If $p < \frac{N}{2}$ *then the final value of low is between* $\frac{N}{2}$ *and* N*, i.e.,*

$$\operatorname{rank}\left(A[p], A\left[\frac{n}{2},\dots,n-1\right]\right) = \mathit{low} - \frac{N}{2} \quad \textit{and} \quad \operatorname{rank}\left(A[p], A\left[0,\dots,\frac{n}{2}-1\right]\right) = p.$$

Hence the rank of $A[p]$ *can be computed in the sorted array to* $p + \mathit{low} - \frac{n}{2}$*. If* $p \geq \frac{N}{2}$*, then the final value of low is between* 0 *and* $\frac{N}{2}$*, hence it holds*

$$\operatorname{rank}\left(A[p], A\left[0,\dots,\frac{n}{2}-1\right]\right) = \mathit{low} \quad \textit{and} \quad \operatorname{rank}\left(A[p], A\left[\frac{n}{2},\dots,n-1\right]\right) = p - \frac{n}{2}.$$

The cost of the algorithm are $\Theta(n \cdot \log n)$ whence the algorithm is not optimal.

Example. $N = 8$ *and the given array is*

Index	0	1	2	3	4	5	6	7
Element	4	8	9	12	2	3	7	10

The situation for processor 0 *is* $x = 4$ *and*

high	7	7	5
low	4	6	6

,

hence low $= 6$*, position of* 4 *in the sorted array is* $p + \mathit{low} - \frac{n}{2} = 0 + 6 - 4 = 2$.

The situation for processor 6 *is* $x = 7$ *and*

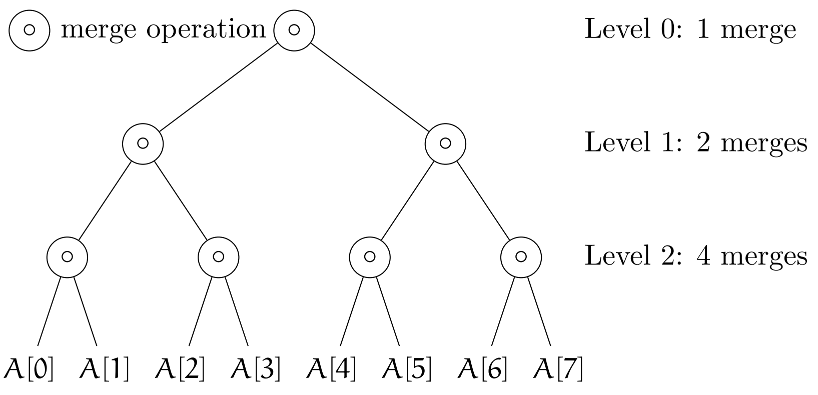
high	3	0	0
low	0	0	1

,

hence low $= 1$*, position of* 7 *in the sorted array is* $p + \mathit{low} - \frac{n}{2} = 6 + 1 - 4 = 3$.

2.5.2 A simple sort algorithm

We will use the known technique of the balanced binary tree.



The merging is computed level wise from bottom to top. In level $0 \leq i < \log n$ we execute 2^i merge operations on the length $2^{\log n - i - 1} = \frac{n}{2^{i+1}}$ in parallel. Therefore we need $2^i \cdot \left(\frac{n}{2^{i+1}} \cdot 2\right) = n$ processors. The number of parallel steps is $O((\log n)^2)$:

#merges	level	list length	steps
1	0	$\frac{n}{2}$	$\log\left(\frac{n}{2}\right)$
2	1	$\frac{n}{4}$	$\log\left(\frac{n}{4}\right)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\frac{n}{2}$	$\log n - 1$	$\frac{n}{n} = 1$	$\log\left(\frac{n}{n}\right)$

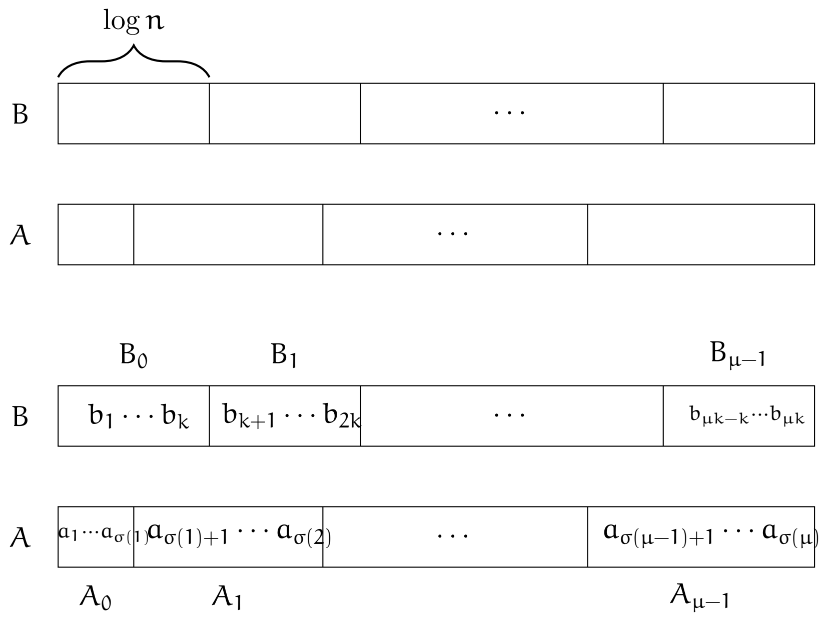
Hence the number of parallel steps are:

$$\begin{aligned} &\approx (\log n - 1) + (\log n - 2) + \cdots + (\log n - \log n) \\ &= \log n \cdot \log n - \frac{\log n(\log n + 1)}{2} = O\Big((\log n)^2\Big). \end{aligned}$$

The sort problem is hence in $\text{CREW}\big(n, (\log n)^2\big)$, however not optimal.

2.5.3 An optimal merge algorithm

Let A and B the sequences that have to be merged. The idea is to divide both sequences into blocks of approximately equal length which can be independently merged.



Algorithm 2.9: partition ⟨CREW⟩

Input : Given two arrays $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_m]$, with $n, m > 1$ sorted ascending order. W.l.o.g. let $\log m = k$ and $\mu = \left\lceil \frac{m}{\log m} \right\rceil$ an integer.

Output : μ pairs (A_i, B_i) of subsequences with the following properties:

1. $|B_i| = \log m = k$, $0 \le i < \mu$,
2. $A = A_0 A_1 \ldots A_{\mu-1}$ and $B = B_0 B_1 \ldots B_{\mu-1}$,
3. Every $x \in A_i$ is larger than every $y \in B_{i-1}$ and vice versa.

Therefore let $\sigma(\mu) = n$ and $\sigma(i) = \text{rank}\,(b_{k \cdot i}, A)$, $0 < i < \mu$. The $\sigma(i)$'s can be determined in parallel with binary search in $O(\log n)$ steps by μ processors.

Example. $A = [4, 6, 7, 10, 12, 15, 18, 20]$, *and* $B = [3, 9, 16, 21]$. *Then* $m = 4, k = 2 = \mu$.
$B_0 = [3, 9], A_0 = [4, 6, 7]$,
$B_1 = [16, 21], A_1 = [10, 12, 15, 18, 20]$,
(A_0, B_0) *and* (A_1, B_1) *can be independently from each other merged and define* C_0*, resp.,* C_1*. It holds*

$$\text{merge}\,(A, B) = C_0 C_1.$$

Lemma 2.8.
Let $m \ge 3$*. Algorithm 2.9 runs in* $O(\log n)$ *steps and requires* $O(n + m)$ *operations on a CREW-PRAM.*

Proof. The runtime of $O(\log n)$ is clear and follows from the binary tree. For the second claim observe that from $(1 + \frac{n}{m})^m < m^n$ follows $(n+m)^m < m^{m+n}$ through multiplication by m inside. Now we apply the logarithm to that and get $m \cdot \log(m+n) < (m+n) \cdot \log(m)$. From monotonicity of logarithm we know $m \cdot \log(n) < m \cdot \log(m+n)$ hence together we have

$$\begin{aligned} & m \log(n) < m \cdot \log(m+n) < (m+n) \cdot \log(m) \\ \Leftrightarrow \quad & \frac{m \log(n)}{\log(m)} < \frac{m \cdot \log(m+n)}{\log(m)} < (m+n), \end{aligned}$$

and thus

$$O(\mu \log n) = O\left(\frac{m}{\log m} \log n\right) = O(n+m).$$

Observation 10.
Let C_i *be the sequence that is defined by merging* A_i *and* B_i*. Then we have* $\text{merge}\,(A, B) = C_0 \ldots C_{\mu-1}$.

How do we merge two sequences of equal length, i.e., $n = m$? *At first we run Algorithm 2.9 in a runtime of* $O(\log n)$*. If* $|A_i| \leq \log n$*, then we use the sequential merge algorithm in* $O(\log n)$ *time. Otherwise we divide* A_i *in blocks of length* $\log n$ *and use Algorithm 2.9 on* A_i *and* B_i *separately. This is possible for the* A_i*'s with* μ *processors in time* $O(\log \log n)$*. Then, again, we use the sequential merge algorithm in* $O(\log n)$ *steps.*

Hence two sequences of length n *can be merge on a CREW-PRAM with* $\frac{n}{\log n}$ *processors in* $O(\log n)$ *steps, i.e.,*

$$\textit{merge} \in \mathsf{CREW}\left(\frac{n}{\log n}, \log n\right),$$

which is optimal. A transfer of these results on the previous section shows that merge-sort $\in$ *CREW*$\left(\frac{n}{\log n}, (\log n)^2\right)$.

Example. *In the following we will go through a complete example for the optimal merge algorithm. The to be sorted list is*

$$[5, 3, 4, 35, 1, 42, 19, 13, 50, 2, 25, 33, 47, 89, 16, 7]$$

The relevant values are $n = 16, \log n = 4, n/\log n = 4$*. At first the list is divided into 4 parts (one for each processor)*

$$\begin{aligned} 0 &: [5, 3, 4, 35] \\ 1 &: [1, 42, 19, 13] \\ 2 &: [50, 2, 25, 33] \\ 3 &: [47, 89, 16, 7] \end{aligned}$$

Then each cpu sorts sequentially in $O(\log n \cdot \log(\log n))$ *steps these lists:*

$$\begin{aligned} 0 &: [3, 4, 5, 35] \\ 1 &: [1, 13, 19, 42] \\ 2 &: [2, 25, 33, 50] \\ 3 &: [7, 16, 47, 89] \end{aligned}$$

Now we start to merge. Run partition on the pairs in parallel:

1. *Partition input:* $A = [3, 4, 5, 35]$ *and* $B = [1, 13, 19, 42]$
2. *Partition input:* $A = [2, 25, 33, 50]$ *and* $B = [7, 16, 47, 89]$

In parallel these partition calls happen:

1. $m = n = 4, \log m = 2, \mu = 2$ *and*
$$B_0 = [1, 13], B_1 = [19, 42], A_0 = [3, 4, 5], A_1 = [35]$$
2. $m = n = 4, \log m = 2, \mu = 2$ *and*
$$B_0 = [7, 16], B_1 = [47, 89], A_0 = [2], A_1 = [25, 33, 50]$$

The overall parallel runtime is $O(\log(\log(n)))$.

As for all respective A_i *their length is* $\leq \log n$ *we just use the sequential merge algorithm for all pairs running in* $O(\log n)$ *parallel time:*

1. *merge(A,B)* $= C_0C_1 = [1, 3, 4, 5, 13][19, 35, 42]$
2. *merge(A,B)* $= C_0C_1 = [2, 7, 16][25, 33, 47, 50, 89]$

Now we have to do the last partition call with
$$A = [1, 3, 4, 5, 13, 19, 35, 42] \quad \textit{and} \quad B = [2, 7, 16, 25, 33, 47, 50, 89].$$

The values are $m = n = 8, \log m = 3, \lceil \mu \rceil = \lceil 8/3 \rceil = 3$. *The subarrays are*
$$B_0 = [2, 7, 16], B_1 = [25, 33, 47], B_2 = [50, 89]$$
$$A_0 = [1, 3, 4, 5, 13], A_1 = [19, 35, 42], A_2 = []$$

Parallel runtime to get them is $O(\log n)$.

However, the length of A_0 *is too long, hence to merge* A_0 *and* B_0, A_0 *is split up by partition into blocks of length* $3 = \log n$ *and then sequentially merged with* B_0 *in time* $O(\log \log n)$. *The others are sequentially merged in* $O(\log n)$. *The result is*

merge(A, B) $= C_0C_1C_2 = [1, 2, 3, 4, 7, 13, 16][19, 25, 33, 35, 42, 47][50, 89]$

Overall we have a parallel runtime of: $O(\log n \cdot \log(\log n)) + O(\log(\log(n))) + O(\log n) + O(\log n) = O((\log n)^2)$ *many steps with* $4 = n/\log n$ *processors.*

2.6 Parallel algorithms for graphs

2.6.1 Connected components

From the definition on page 7 we are familiar with connected components. From exercise sheet 2 we know that the algorithm determining the connected components of a given graph utilizes the breadth-first-search technique and therefore runs in $O(n+m)$, resp., $O(n^2)$ steps where n is the number of vertices and m is the number of edges.

Definition (Sparse and dense graphs). *A given graph* $G = (V, E)$ *is* dense *if the number of edges* m *is close to the number of* n^2*, i.e.,* $m \sim n^2$*.*

G *is said to be* sparse *if the number of edges* m *is much smaller than* n^2*, i.e.,* $m \ll n^2$*.*

Definition (Pseudo-tree). *A* pseudo-tree *is a directed graph* $G = (V, E)$ *such that the following holds:*

- *The out-degree of every vertex is* 1*.*
- *The corresponding undirected graph (any directed edge is considered to be undirected) is connected.*

Lemma 2.9.
A pseudo-tree has exactly *one cycle.*

Proof. Induction on $||V|| = n$. If $n = 1$ the claim is clear. ✓

Now let $n > 1$. Search for a vertex with in-degree 0. If there is no such vertex, then we have found a cycle. Otherwise remove this vertex and apply induction hypothesis.

If we found more than one cycle, then an edge of a cycle must connect to another cycle. Hence the out-degree would be larger than 1 which is a contradiction to the definition of pseudo-trees. □

Definition. *A* pseudo-tree with root r *is a pseudo-tree plus there is an edge* (r, r)*. A* root star *is a pseudo-tree with root* r *plus every path from a vertex* $v \neq r$ *to* r *has length* 1*.*

Theorem 2.10.
Every function $f \colon V \to V$ *defines a pseudo-forest* (V, F) *with* $F = \{(v, f(v)) \mid v \in V\}$*.*

Proof. Induction on $||V|| = n$. If $n = 1$ then $f(v) = v$. ✓

Induction step. $n \to n+1$. Let $V' = \{f(v) \mid v \in V\}$ and $||V|| = n + 1 > 1$. We need to distinguish two cases:

Case 1: If $||V'|| < ||V||$ then choose a $v' \in V \setminus V'$. By IH $f|_{V \setminus \{v'\}}$ defines a pseudo-forest. If we then add v' and the edge $(v', f(v'))$ again we get a pseudo-forest.

Case 2: If $||V|| = ||V'||$, then f is a permutation and the pseudo-forest consists of one or more (distinct) cycles. □

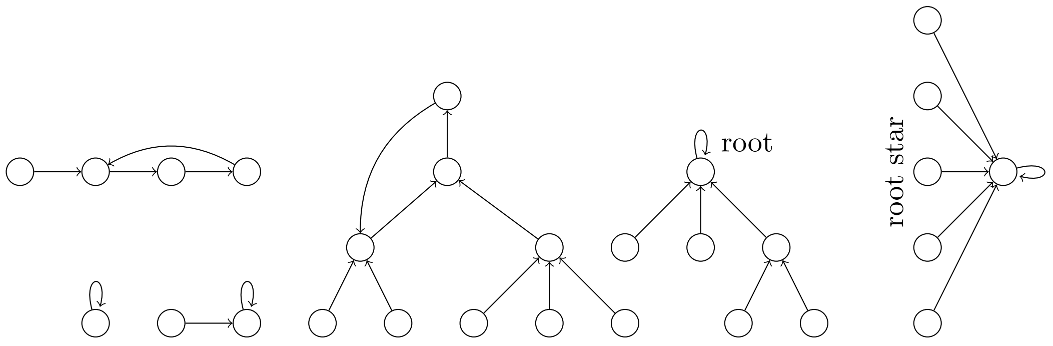


Figure 2.3: Examples of pseudo-trees (with root) and a root star.

2.6.2 Connected components in dense graphs

Theorem 2.11.
Let A *be the adjacency matrix of the undirected graph* $G = (V, E)$ *with* $V = \{1, \dots, n\}$*. Further define the function* $\mu \colon V \to V$ *as*

$$\mu(u) := \begin{cases} u & , \textit{ if } A[v,u] = 0 \textit{ for all } v \in V \\ \min\{v \mid A[v,u] = 1\} & , \textit{ otherwise} \end{cases}$$

which generates a pseudo-forest (V, E') *where the vertices of the pseudo-trees* $T_1, \dots, T_s$ *fix a partition of* V *into* $V_1, \dots, V_s$*. Then it holds that*

1. *All vertices in* V_i *are part of a connected component in* G*.*
2. *The cycle in* T_i *has length* ≤ 2 *and contains the smallest vertex in* V_i*.*

Proof. We will prove both cases with induction on $\|V_i\| = n_i$.

1. $n_i = 1$ then v is an isolated vertex. ✓

   Induction step. $n_i > 1$ and let $u, v \in V_i$ and $u \neq v$. Then $(u, \mu(u))$ and $(v, \mu(v))$ are contained in the pseudo-tree T_i. Every edge in T_i corresponds to an edge in G.

2. If $n_i = 1$ then the cycle has length 1. ✓

   Induction step. If $n_i > 1$ and let r be the smallest vertex in V_i. Let $\mu(r) = u$. Hence $\{r, u\} \in E$ and therefore $\mu(u) = r$ (as r is minimal). Thus there exists a cycle around $(r, u), (u, r)$ of length 2. We claim that this is the only cycle in T_i. Assume the opposite, i.e., there is another cycle C in T_i. On this cycle C there is some minimal vertex $r' \geq r$.

   If $r' = r$ then the out-degree of r is > 1. $\lightning$ Hence $r' > r$ holds. However, as C is in T_i C must be connected then with u getting either a contradiction for the out-degree of u or for the out-degree of one vertex $v \neq r'$ in C. $\square$

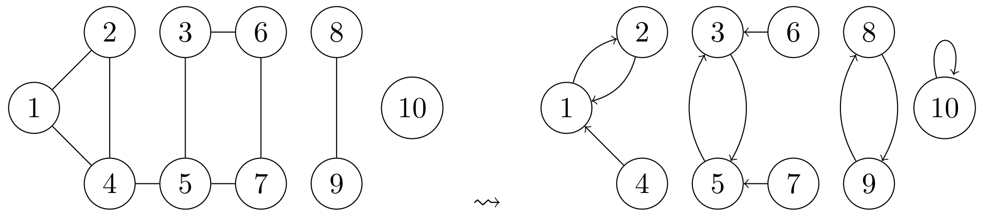


Figure 2.4: Example for μ-function.

In the following we want to compute the connected components of a given undirected graph by using a recursive technique of the previous theorem. Thereby we want to merge the vertex set V_i to a *super node* and want to repeat the technique recursively on the reduced graph until all super nodes are isolated. Every super node then corresponds to one CC in G. The representative of a super node is the smallest vertex in it. As the explicit construction of new adjacency matrices is too expensive we define a new function C as follows:

$$\forall u, v \in V\colon C[u] = C[v] \iff u \text{ and } v \text{ are in the same super node.}$$

The function C will be iteratively improved and defines after at most $\lceil \log n \rceil$ iterations super nodes which are isolated vertices because in every iteration non-isolated vertices will be merged with at least one other one. Hence the set of vertices will be halved in every iteration step.

Algorithm 2.10 uses n^2 processors on a CREW-PRAM. In step 1 the elements of M are determined in parallel (therefore we need n^2 processors!) and entered into an auxiliary array $D[i, j]$ (which is not specified in the pseudo code and represents the set M). Afterwards in $O(\log n)$ steps the computation of the minimum (technique of balanced binary tree). Analogously step 2. Step 3 runs in $O(1)$ steps. Step 4 runs in $O(\log n)$ steps. Hence we get $\mathsf{compCC} \in \text{CREW}(n^2, (\log n)^2)$. However this is not optimal as in the sequential case we need $O(n^2)$ steps.

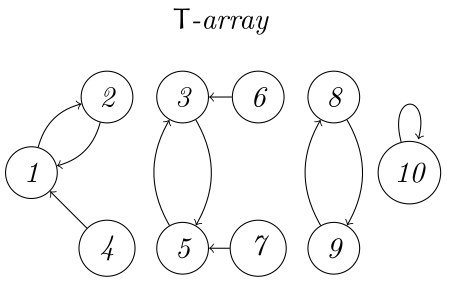
Using less processors improves the cost. If we are interested in using less than n^2 processors, e.g., only n many then step 1 will require linear time instead of constant. By this we get an overall cost of the algorithm of $n \cdot \log(n) \cdot n = n^2 \cdot \log n$ which is better than $n^2 \cdot \log(n)^2$ but has a worse runtime of $\log(n) \cdot n$ compared to $\log(n)$.

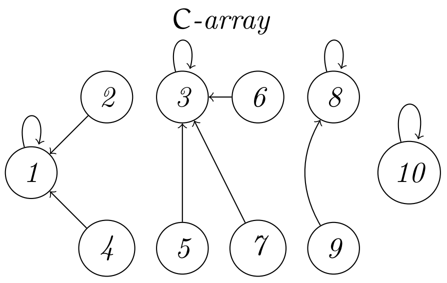
Example. *Consider the graph from Figure 2.4. After the initialization we get*

vertex v	1	2	3	4	5	6	7	8	9	10
C-*array:*	1	2	3	4	5	6	7	8	9	10

In the following we depict each step of the loop in the algorithm. Iteration 1:

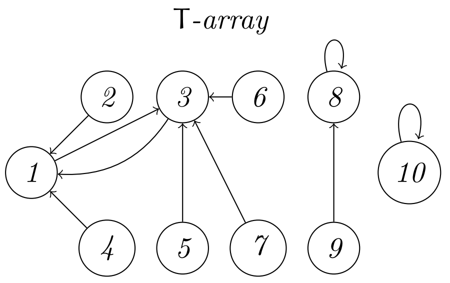
vertex ν	1	2	3	4	5	6	7	8	9	10
Step 1 T*-array:*	2	1	5	1	3	3	5	9	8	10
Step 2 T*-array:*	2	1	5	1	3	3	5	9	8	10
Step 3 C*-array:*	1	1	3	1	3	3	5	8	8	10
Step 4 C*-array:*	1	1	3	1	3	3	3	8	8	10

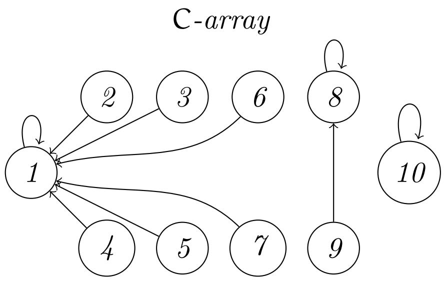




Iteration 2:

vertex ν	1	2	3	4	5	6	7	8	9	10
Step 1 T*-array:*	1	1	3	3	1	3	3	8	8	10
Step 2 T*-array:*	3	1	1	1	3	3	3	8	8	10
Step 3 C*-array:*	1	1	1	1	3	3	3	8	8	10
Step 4 C*-array:*	1	1	1	1	1	1	1	8	8	10





Iteration 3:

vertex ν	1	2	3	4	5	6	7	8	9	10
Step 1 T*-array:*	1	1	1	1	1	1	1	8	8	10
Step 2 T*-array:*	1	1	1	1	1	1	1	8	8	10
Step 3 C*-array:*	1	1	1	1	1	1	1	8	8	10
Step 4 C*-array:*	1	1	1	1	1	1	1	8	8	10

Algorithm 2.10: compCC ⟨CREW⟩

```
global $A$: array$[1,\dots,n][1,\dots,n]$ of integer, $C, T$: array$[1,\dots,n]$ of integer;
local $n, p$: integer, $M$: set;
$C[p] \leftarrow p$;                          // every vertex is its own representative
for count $\leftarrow 1$ to $\lceil \log n \rceil$ do
    ⋆⋆⋆ Step 1 ⋆⋆⋆
    // $M$ is set of the super node representatives connected with vertex p
    $M \leftarrow \{C[j] \mid A[p,j] = 1$ and $C[j] \neq C[p]\}$;
    if $M = \emptyset$ then $T[p] \leftarrow C[p]$                // Determine $\mu$-function ;
    else $T[p] \leftarrow \min(M)$         // $T[p]$ could be a representative candidate ;

    ⋆⋆⋆ Step 2 ⋆⋆⋆
    $M \leftarrow \{T[j] \mid C[j] = p$ and $T[j] \neq p\}$;
    // Check for all vertices of super node p if in step 1 connections
    have been determined. If so create the connection to the smallest
    representative (out degree $= 1$!)
    if $M = \emptyset$ then $T[p] \leftarrow C[p]$ ;
    else $T[p] \leftarrow \min(M)$ ;

    ⋆⋆⋆ Step 3 ⋆⋆⋆
    if $C[p] = p$ then $C[p] \leftarrow T[p]$          // transfer representative to root ;
    if $C[C[p]] = p$ then $C[p] \leftarrow \min(C[p], p)$               // break the loops ;

    ⋆⋆⋆ Step 4 ⋆⋆⋆
    for count' $\leftarrow 1$ to $\lceil \log n \rceil$ do $C[p] \leftarrow C[C[p]]$ // distribute repres. number ;
```

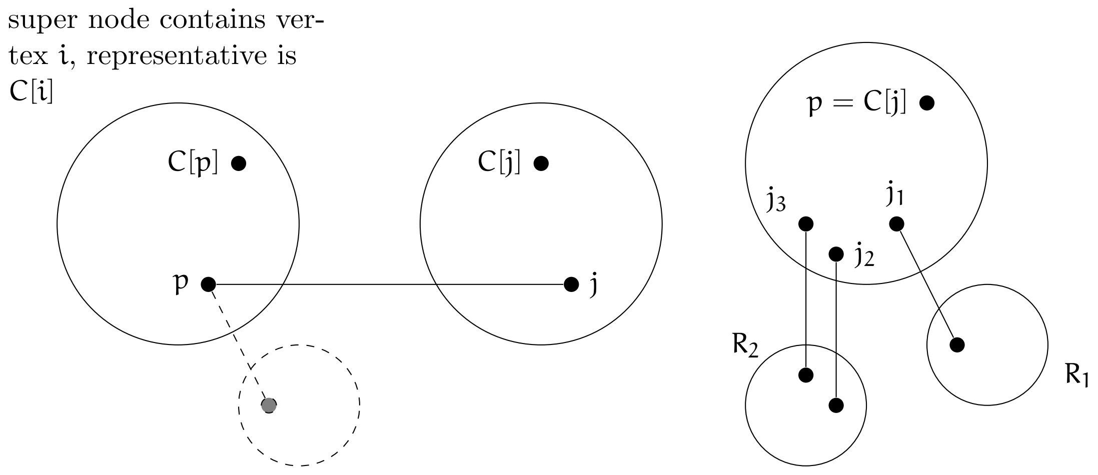


Figure 2.5: Schema of step 1 (left) and step 2 (right) in Algorithm 2.10

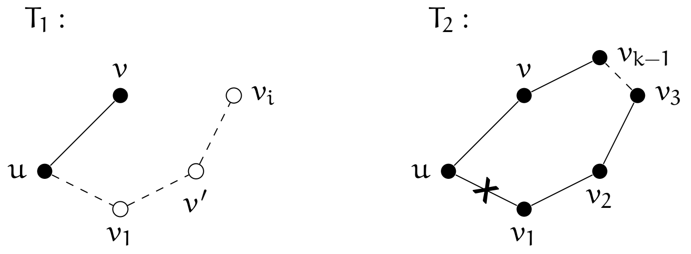
2.6.3 Minimal spanning trees (MST)

Let $G = (V, E)$ be an undirected, connected graph and $w\colon E \to \mathbb{N}$ a weighting function. A spanning tree of G is a subgraph $T = (V, E')$ with $E' \subseteq E$ which is a tree. The weight of T, defined by $w(T) = \sum_{e \in E'} w(e)$, is the sum of edge weigths of E'. An MST is a spanning tree with minimum weight.

Without loss of generality we always assume that all edge weights are different. E.g., this can be realized by assigning the weight $(w(e).k)$ where $k \in \mathbb{N}$ is a serial number. By this the weighting function becomes $w\colon E \to \mathbb{Q}^+$.

Lemma 2.12.
Let $G = (V, E)$ be a connected, undirected graph with weighting function w which assigns different weights to each edge. Then there is exactly one minimum spanning tree.

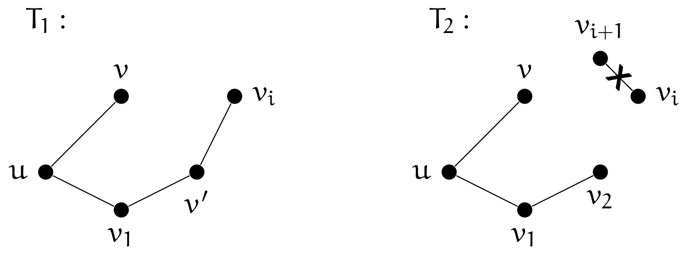
Proof. Let $T_1 \neq T_2$ be minimum spanning trees with $w(T_1) = w(T_2)$. Let (u, v) be an edge of minimum weight which is in exactly one of both trees T_1 and T_2. W.l.o.g. let $(u, v) \in T_1$ and $(u, v) \notin T_2$.



Then there exists in T_2 a path $(u, v_1, v_2, \ldots, v_{k-1}, v_k = v)$ which connects u with v and $v_1 \neq v$.

1. If $w(u, v) < w(u, v_1)$ then replace in T_2 the edge (u, v_1) by (u, v). Then we get a spanning tree T_2' with $w(T_2') < w(T_2)$ which is a contradiction.

2. If $w(u, v) > w(u, v_1)$ then the edge (u, v_1) is in T_1 due to minimality of $w(u, v)$. Consider the first edge (v_i, v_{i+1}) on the path from u to v in T_2 whose weight is larger than $w(u, v)$. This edge must exists because otherwise there would be a cycle in T_1. Replace this edge with (u, v) in T_2 and get a contradiction again.



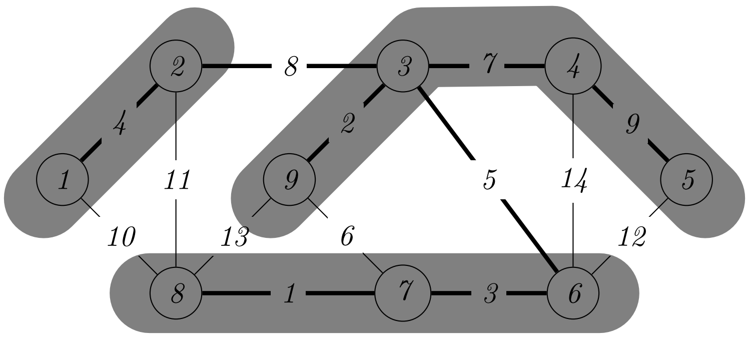
□

Lemma 2.13.
Let $G = (V, E)$ *be an undirected, connected graph with weighting function* w*. For every* $u \in V$ *let* $\mu(u) = v$ *if* $w(u, v) = \min\{w(u, v') \mid (u, v') \in E\}$ *holds. Then it holds that*

1. $(u, \mu(u))$ *is an edge in the MST of* G*.*
2. *Every pseudo-tree in the* μ*-defined pseudo-forest has a cycle of length* ≤ 2*.*

Proof.
1. Let T be a MST of G and let $(u, \mu(u)) \notin T$. Let $\pi = (u, v_1, \ldots, v_k = \mu(u))$ be the path which connects u with $\mu(u)$ in T. Replace edge (u, v_1) by $(u, \mu(u))$ we get a spanning tree with weight $< w(T)$. $\lightning$
2. If a pseudo-tree T' contains a cycle of length > 2 then in the corresponding undirected graph T_1 there exists a cycle. As all edges of T_1 are edges in the MST (due to 1.), we get a contradiction. $\square$

Example. *Consider the following graph:*



Then we get

u	1	2	3	4	5	6	7	8	9
$\mu(u)$	2	1	9	3	4	7	8	7	3

.

Lemma 2.14.
Let $V = \biguplus_{i=1}^{r} V_i$ *be a partition of* V *into* r *non-empty sets* $V_1, \ldots, V_r$*. Let* e_i *be the edge of minimum weight which connects a vertex* $u \in V_i$ *with a vertex* $v \in V \setminus V_i$*. Then* e_i *is in the MST of* $G = (V, E)$*.*

Proof. Let $e_i = (u, v)$ and T be the MST of G. For contradiction assume e_i is not an edge in T. There exists a path $\pi = (u = v_0, v_1, \ldots, v_s = v)$ in T which connects u with v. Let t be the index with $v_{t-1} \in V_i$ and $v_t \in V \setminus V_i$. It holds that $w(v_{t-1}, v_t) > w(e_i) = w(u, v)$. If we remove (v_{t-1}, v_t) from T and add (u, v) instead, then we get a spanning tree of weight $< w(T)$. $\lightning$ $\square$

Sollins algorithm starts with a pseudo-forest F_0 in which every pseudo-tree is a single vertex. Then it successively builds trees through subsets of V until a single tree is constructed containing all vertices.

In every iteration an edge of minimum weight is added to *each* tree in the forest. This requires $O(\log|V|)$ steps. The algorithm uses a weighting matrix W such that

$$W[i,j] = \begin{cases} w(i,j), & \text{if } (i,j) \in E \\ \infty, & \text{otherwise} \end{cases}$$

and further we assume that $V = \{1, \dots, n\}$.

Algorithm 2.11: Sollins algorithm to compute a minimum spanning tree. $\langle \mathsf{CREW} \rangle$

```
global $A$: array$[1,\dots,n][1,\dots,n]$ of integer, $W$: array$[1,\dots,n][1,\dots,n]$ of integer;
global $C, T$: array$[1,\dots,n]$ of integer;
local $n, p$: integer;
$C[p] \leftarrow p$;                          // every vertex is its own representative
for count $\leftarrow 1$ to $\lceil \log n \rceil$ do
    ⋆⋆⋆ Step 1 ⋆⋆⋆
    $M \leftarrow \{C[j] \mid A[p,j] = 1$ and $C[j] \neq C[p]\}$;
    if $M = \emptyset$ then $T[p] \leftarrow C[p]$ ;
    else $T[p] \leftarrow \min(M)$ ;
    // $M$ set of weights of all edges from super node $C[p]$ to other super
    nodes. $T[p]$ specifies the vertex $j$ outside of super node $C[p]$ to which
    an edge of minimum weight exists.

    ⋆⋆⋆ Step 2 ⋆⋆⋆
    // Determine $\mu$-function.
    $M \leftarrow \{(j, T[j]) \mid C[j] = p$ and $C[T[j]] \neq p\}$;
    if $M = \emptyset$ or $\min(M) = \infty$ then $T[p] \leftarrow C[p]$ ;
    else
        Compute $j$ such that $W[j, T[j]]$ is minimal with respect to $M$;
        $T[p] \leftarrow C[T[j]]$;
        mark edge $(j, T[j])$;                 // Edge $(j, T[j])$ is part of the MST
    // For every super node the edge to other super nodes will be added to
    $M$. The edge with minimum weight in $M$ then determines which super
    nodes are merged. This edge is part of the MST.

    ⋆⋆⋆ Step 3 ⋆⋆⋆
    if $C[p] = p$ then $C[p] \leftarrow T[p]$        // transfer representative to root ;
    if $C[C[p]] = p$ then $C[p] \leftarrow \min(C[p], p)$          // break the loops ;

    ⋆⋆⋆ Step 4 ⋆⋆⋆
    for count' $\leftarrow 1$ to $\lceil \log n \rceil$ do $C[p] \leftarrow C[C[p]]$  // distribute repres. number ;
```

As we can see Algorithm 2.11 is a slight modification of Algorithm 2.10.

Bibliography

[Anderson and Miller, 1990] Anderson, R. J. and Miller, G. L. (1990). A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269 – 273.

[Cormen et al., 2001] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2001). *Introduction to Algorithms.* MIT Press, second edition.

[Diestel, 2005] Diestel, R. (2005). *Graph theory.* Springer Verlag.

[Ottmann and Widmayer, 2012] Ottmann, T. and Widmayer, P. (2012). *Algorithmen und Datenstrukturen.* Springer DE.