Graph Theory

Lecture Notes

Winter 2014/2015

Christian Schindelhauer

Computer Networks and Telematics
Faculty of Engineering

University of Freiburg, Germany

March 30, 2015

Chapter 1

Organization and Introduction

Take the Tokyo subway route map in Fig. 1.1 as an example. Can you find the shortest path from Makanobe to Inaricho? Maybe it is already hard to find these stations. In this lecture we will learn to describe such graphs and their features and we will learn to solve basic problems concerning graphs.

1.1 Literature

We follow very closely the following books:

- Krumke, Noltemeier, "Graphentheoretische Konzepte und Algorithmen", Springer 2005, [KN09]
- Diestel, "Graph Theory", Springer 2000. [Die10]

Since the first book is only available in English, these lecture notes are supposed to cover the relevant topics of this lecture. Note that most of the material is taken from the first book, which is highly recommend for your studies.

1.2 Examples

1.2.1 Frequency Allocation

Each sender station of a radio network can choose a frequency from the set $\{1, \ldots, k\}$. Neighbored radio station should use different frequencies. This neighborhood describes a graph. This leads to the so-called **graph coloring problem**. The task is to determine the minimum number of colors such that all nodes receive a color different to each of their neighbors, see Fig. 1.2.

Especially interesting is the case of planar graphs, where famously the four color problem was proven in 1976 by Kenneth Appel and Wolfgang Haken using the first computer aided proof [AH$^+$77, AHK$^+$77].

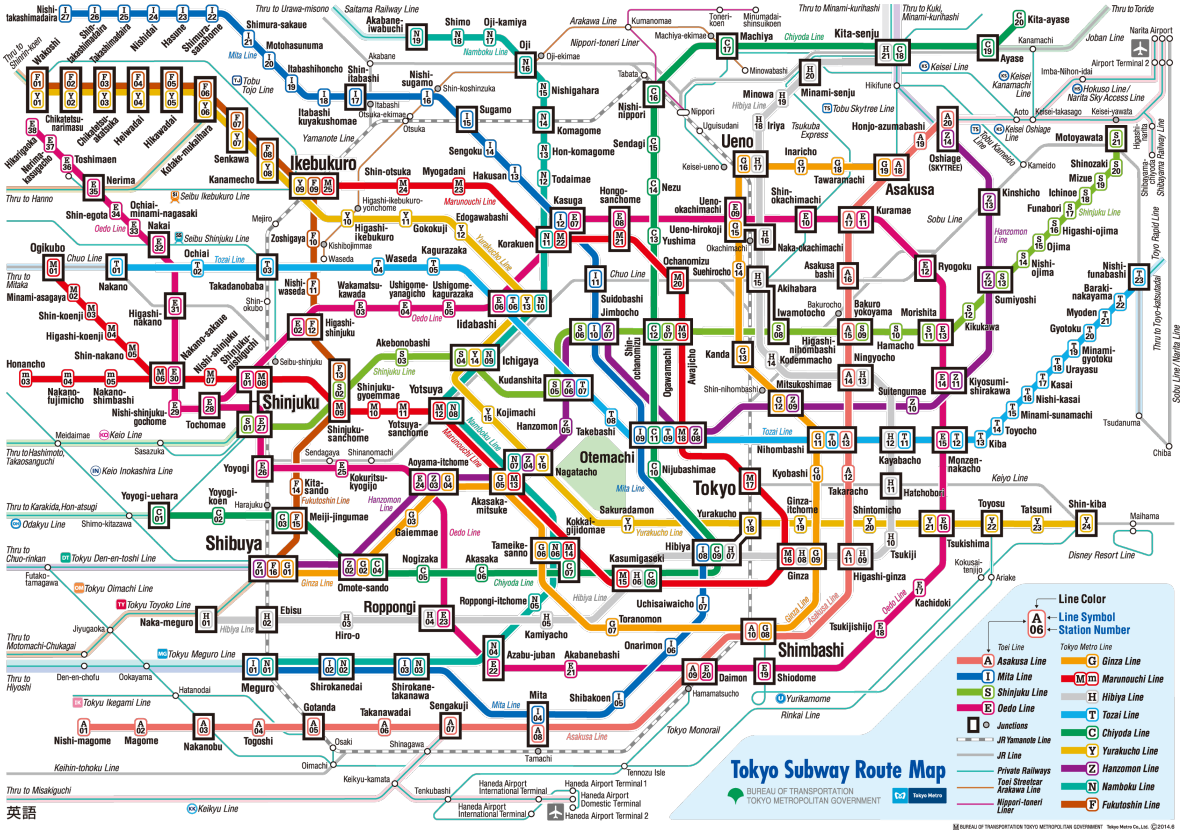


Figure 1.1: The Tokyo subway route map shows the complexity of real-world graphs in daily life (Copyright Tokyo Metro)

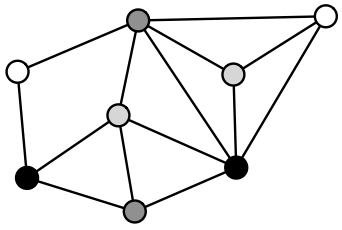


Figure 1.2: A coloring of a graph

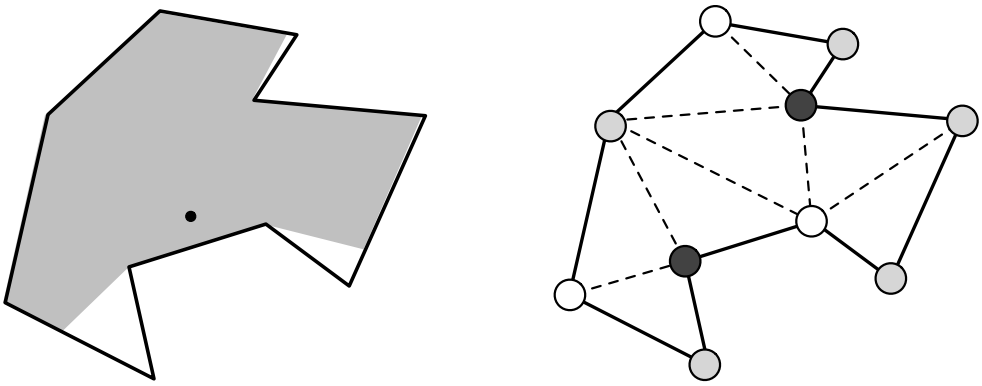


Figure 1.3: The art gallery problem and the coloring problem.

1.2.2 Art Gallery Problem

Given an art gallery with a complicated internal architectural design, how many guards need to be placed such that all walls of the museum can be watched by the guards. It can be shown that for galleries with n corners the optimal solution relies on a triangulation of the underlying polygon. The triangles need to (and can) be colored with three colors such that triangles sharing an edge have different colors. Now, choose the most seldom color and it turns out that this number n determines the minimum number of guards by $\lfloor \frac{n}{3} \rfloor$ for some worst case situations, see Fig. 1.3[1].

1.2.3 Seven Bridges of Königsberg

Formulated in 1735 by Leonhard Euler it asks whether an **Euler path** exists, i.e. a path passing all bridges of Königsberg exactly once. This is the same problem as drawing the house in Fig. 1.4 (right) without lifting the pen or drawing a line twice. Graphs with Euler paths can be characterized very precisely by the Theorem of Euler:

Theorem 1 (Euler II) *An Euler path exists if and only if the graph is connected and the degree of all except of at most two nodes is even.*

We will prove this theorem within this lecture.

1.3 Basic Notions

We need some basic mathematical notions to describe graphs formally.

[1]This construction gives the optimal bound only in the worst case. As a student correctly noted in the lecture, for some art galleries better solutions can be obtained

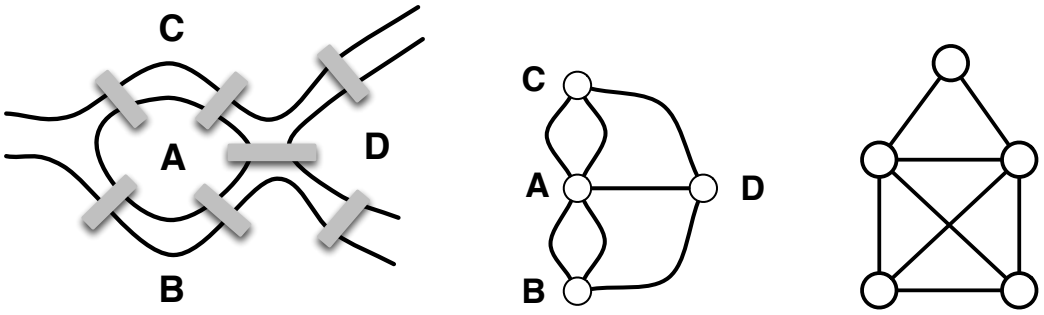
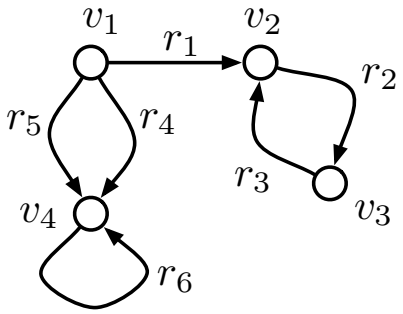


Figure 1.4: Graphs for the Euler Path problem



	α	ω
r_1	v_1	v_2
r_2	v_2	v_3
r_3	v_3	v_2
r_4	v_1	v_4
r_5	v_1	v_4
r_6	v_4	v_4

Figure 1.5: A directed graph with loops, parallels and antiparallels

1.3.1 Directed Graphs

Definition 1 (Directed Graph/digraph) *A* **directed graph** $G = (V, R, \alpha, \omega)$ *is given by*

1. *a* **node (vertex) set** V *of the graph* G
2. *a set of* **directed edges (directed arcs)** R
3. *both sets are disjoint:* $R \cap V \neq \emptyset$
4. *the functions* $\alpha : R \to V$ *and* $\omega : R \to V$*, where* $\alpha(r)$ *denotes the* **tail** *(or initial) node and* $\omega(r)$ *denote the* **head** *(or terminal) node of the directed edge* r*.*

A directed graph is **finite**, if V and R are finite. We denote by $V(G)$ the set of nodes or vertices of G, and by $R(G)$ the multi-set of directed edges or directed edges of G, see Fig. 1.5. In contrast to a set a multi-set

$$R(G) = \{\{(\alpha(r), \omega(r)) \mid r \in R\}\}$$

may have multiple identical elements and preserves the quantity of these elements while the ordering of the elements of a multi-set is not important.
Infinite graphs are also conceivable. as an example consider $V = \mathbb{N}$ and $R = \{(i, i+1) \mid i \in \mathbb{N}\}$, see Fig. 1.6.

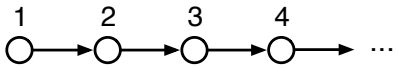


Figure 1.6: An infinite graph

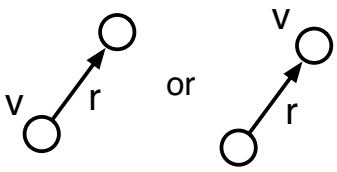


Figure 1.7: v and r are incident

Definition 2 (loops, digons, simple graph) *Let $G = (V, R, \alpha, \omega)$. Then*

- *$r \in R$ is a* **loop***, if $\alpha(r) = \omega(r)$,*
- *G is a* **loop-free** *graph, if $\forall r \in R: \ \alpha(r) \neq \omega(r)$*
- *$r, r' \in R$ are called* **parallel** *(or multiple) edges, if $\alpha(r) = \alpha(r')$ and $\omega(r) = \omega(r')$.*
- *$r, r' \in R$ are called a* **digon** *or a pair of* **antiparallel** *edges, if $\alpha(r) = \omega(r')$ and $\omega(r) = \alpha(r')$.*
- *a graph G is* **simple** *if it contains no loops and no parallel edges.*

For simple graphs G we denote $G = (V, R)$ with $R \subseteq V \times V$.

Definition 3 (Incidence, adjacency, degree, maximal degree) *Let $G = (V, R, \alpha, \omega)$. Then*

- *a vertex v and a directed edge r are called* **incident***, if $v \in \{\alpha(r), \omega(r)\}$, see Fig. 1.7.*
- *two directed edges r and r' are called* **incident***, if*

$$\{\alpha(r), \omega(r)\} \cap \{\alpha(r'), \omega(r')\} \neq \emptyset \, .$$

- *two vertices $u, v \in V$ are called* **adjacent** *or* **neighbored***, if $\exists r \in R$ such that u and r are incident and v and r are incident.*

The following notations help us to name parts of the graph. For an vertex $v \in V$ of a graph G

- *$\delta_G^+(v) := \{r \in R : \alpha(r) = v\}$ is the set of* **outgoing edges** *of v.*
- *$\delta_G^-(v) := \{r \in R : \omega(r) = v\}$ is the set of* **incoming edges** *of v.*
- *$N_G^+(v) := \{\omega(r) : r \in \delta_G^+(v)\}$ is the* **out-neighborhood/successor set** *of v.*
- *$N_G^-(v) := \{\alpha(r) : r \in \delta_G^+(v)\}$ is the* **in-neighborhood/predecessor set** *of v.*

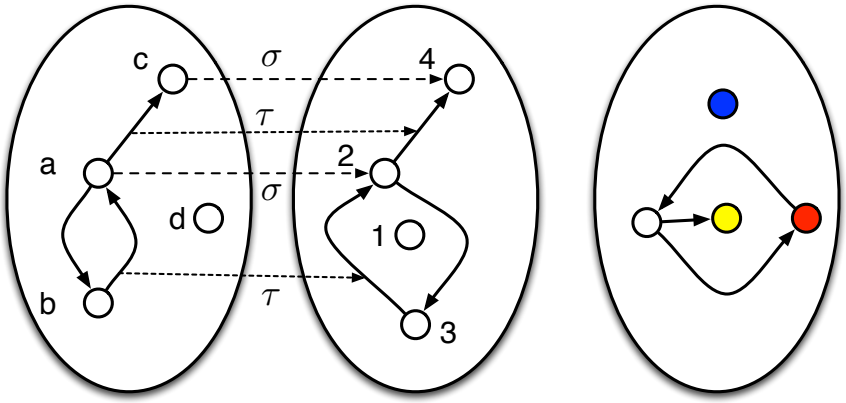


Figure 1.8: Three isomorphic digraphs

- $d_G^+(v) := |\delta_G^+(v)|$ *is the* **out degree** *of* v[2]
- $d_G^-(v) := |\delta_G^-(v)|$ *is the* **in degree** *of* v
- $d_G(v) := d_G^+(v) + d_G^-(v)$ *is the* **degree** *of* v
- $\Delta(G) := \max\{d_G(v) : v \in V\}$ *is the* **maximum degree** *of the graph* G.
- $\delta(G) := \min\{d_G(v) : v \in V\}$ *is the* **minimum degree** *of the graph* G.

The following observation turns out to be helpful

$$\sum_{v \in V} d_G^+(v) = \sum_{v \in V} d_G^-(v) = |R| \,. \tag{1.1}$$

So, we are equipped to prove the first lemma.

Lemma 1 *In a finite graph the number of vertices with odd degree is even.*

Proof: Let $U \subseteq V$ be the set of vertices with odd degree.

$$\underbrace{2|R|}_{\text{even}} = \underbrace{\sum_{v \in V} d(v)}_{\text{even}} = \underbrace{\sum_{v \in V \setminus U} d(v)}_{\text{even}} + \underbrace{\sum_{v \in U} (d(v) - 1)}_{\text{even}} + |U|$$

Therefore, $|U|$ must be an even number, too. □

1.3.2 Isomorphic Graphs

Isomorphic graphs are "equal", see Fig. 1.8.

[2]Note that we have used g (Grad) for d (degree) in the German lectures of the recent years.

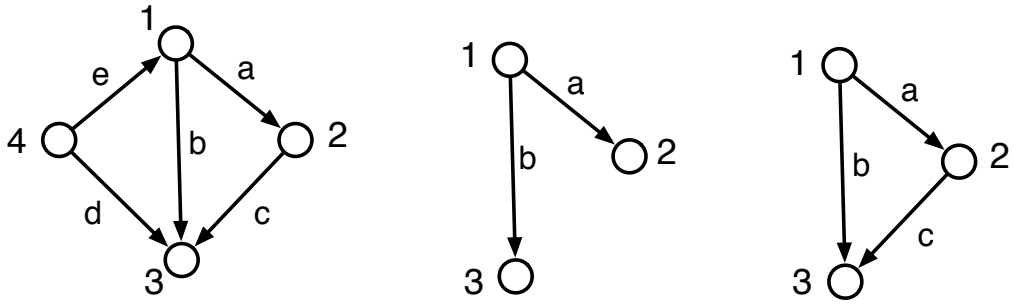


Figure 1.9: A digraph, a sub-graph of it and its induced sub-graph of vertices 1,2,3.

Definition 4 *Let $G_i = (V_i, R_i, \alpha_i, \omega_i)$ for $i \in \{1, 2\}$ be digraphs. Then $G_1 \cong G_2$ are* **isomorphic***, if there exists a bijective mapping $\sigma : V_1 \to V_2$ and $\tau : R_1 \to R_2$ with*

1. $\alpha_2(\tau(r)) = \sigma(\alpha_1(r)) \quad \forall r \in R_1$,
2. $\omega_2(\tau(r)) = \sigma(\omega_1(r)) \quad \forall r \in R_1$.

Definition 5 (Subgraph, supergraph) *A graph $G' = (V', R', \alpha', \omega')$ is called a* **subgraph** *of $G = (V, R, \alpha, \omega)$, i.e. $G' \sqsubseteq G$, if*

1. $V' \subseteq V$ *and* $R' \subseteq R$
2. $\forall r \in R' : \quad \alpha'(r) = \alpha(r), \omega'(r) = \omega(r)$.[3]

G is then a **super-graph** *of G'. If $V' \subset V$ or $R' \subset R$, then G is a* **proper super graph**.

See, Fig. 1.9 for an example. The following observations can be made for all G, G', G''.

1. $G \sqsubseteq G$ (reflexivity)
2. $G \sqsubseteq G',\ G' \sqsubseteq G \implies G = G$ (antisymmetry)
3. $G \sqsubseteq G',\ G' \sqsubseteq G'' \implies G \sqsubseteq G''$ (transitivity)

Hence, $\sqsubseteq$ describes a partial order.

Definition 6 *Let $G = (V, R, \alpha, \omega)$ and $V' \subseteq V$. The* **induced subgraph** *$G[V']$ is the subgraph G' with vertex set and edge set $\{r \in R : \alpha(r) \in V'$ and $\omega(r) \in V'\}$.*[4]

For $R' \subseteq R$ is $G_{R'}$ is the **induced subgraph of** *R':*

$$G_{R'} := (V, R', \alpha|_{R'}, \omega|_{R'}) \ .$$

The following notations complete this chapter

$$\begin{aligned} G - r &= \text{the induced subgraph of } G_{R(G)\setminus r'} \\ G - v &= \text{the induced subgraph of } G[V(G) \setminus \{v\}]. \end{aligned}$$

[3] other notation $\alpha|_{R'} = \alpha'$, and $\omega|_{R'} = \omega'$

[4] "and" is correct. In the lecture we erroneously stated "or"

1.3.3 Undirected Graphs

Definition 7 *An undirected graph is a triple $G = (V, E, \gamma)$ consisting of*

- *a set V of vertices (nodes),*
- *a disjoint of undirected edges E, i.e. $E \cap V = \emptyset$, and*
- *the mapping $\gamma : E \to \{X : X \subseteq V \text{ with } 1 \leq |X| \leq 2\}$ assigning the nodes $\gamma(e) \subseteq V$ to the edge e.*

The definitions of incidence, adjacency, degree, sub-graph, super-graph are analog to the definitions for directed graphs. A special case is the **loop** e where $|\gamma(e)| = 1$ and $\gamma(e) = \{v\}$ for some node v.[5]

$$\begin{array}{rcll} \delta(v) & := & \{e \in E : v \in \gamma(e)\} & \text{edges incident to } v \\ N_G(v) & := & \{u \in V : \gamma(e) = \{u, v\} \text{ for } e \in E\} & \text{nodes adjacent with } v \\ d_G(v) & := & \sum_{e \in E : v \in \gamma(e)} (3 - |\gamma(e)|) & \text{degree of } v \\ \Delta(G) & := & \max\{d_G(v) : v \in V\} & \end{array}$$

An undirected graph is **simple**, if it does not contain any loops or parallels. Then, $G = (V, E)$ with $e = \{u, v\} \subseteq V$. Then, we denote for an edge $e = [u, v]$.

In a case of an undirected graph without parallels we denote its loops by $e = [u, u]$.

Lemma 2 *The number of nodes in a finite undirected graph G with odd degree is even.*

Proof: First note that

$$\sum_{v \in V} d(v) = 2|E| \ ,$$

since we count every node twice. Let $U \subseteq V$ be the set of edges with odd degree. Then

$$\sum_{v \in U} d(v) = 2|E| - \underbrace{\sum_{V \setminus U} d(v)}_{\text{even}} \ .$$

Therefore the first term must be even as well. □

Isomorphism of Undirected graphs

Definition 8 *Let $G = (V, E, \gamma)$ and $G' = (V', E', \gamma')$ are isomorphic, i.e. $G \cong G'$, if there exist bijective functions $\sigma : V \to V'$ and $\tau : E \to E'$ such that*

$$\gamma'(\tau(e)) = \sigma(\gamma(e)) \ .$$

See Fig. 1.10 for an example.

[5] Again we use $d_G(u)$ instead of $g_G(u)$ for the degree deviating from the lecture of last year.

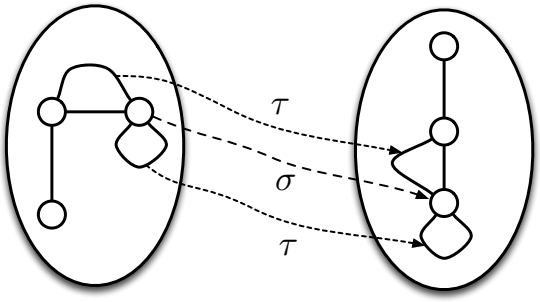


Figure 1.10: Two isomorphic undirected graphs

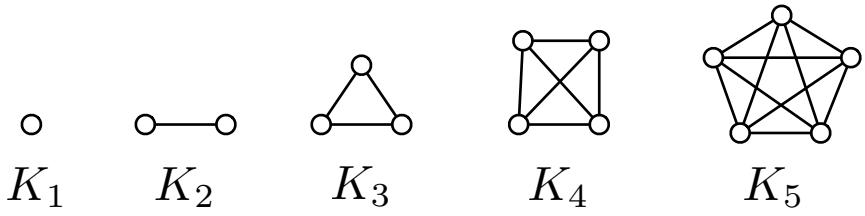


Figure 1.11: The graph family K_n of complete graphs.

Complete Graphs

The simple graph $K_n = (V_n, E_n)$ with $V_n = \{1, 2, \ldots, n\}$, $E_n = \{[i, j] \ : \ i, j \in V_n, \ i \neq j\}$ is called a complete graph, see Fig. 1.11 for the first five complete graphs.

Relating directed and undirected graphs

Definition 9 (Inverse, symmetric hull) *Let $G = (V, R, \alpha, \omega)$ be a digraph. For each directed edge $r \in R$ define a new directed edge r^{-1} as*

$$\alpha(r^{-1}) = \omega(r) \quad \textit{and} \quad \omega(r^{-1}) = \alpha(r) \ .$$

Define $R^{-1} := \{r^{-1} \ : \ r \in R\}$ with $R \cap R^{-1} = \emptyset$.

The **inverse graph** *G^{-1} is defined as*

$$G^{-1} = (V, R^{-1}, \alpha, \omega) \ .$$

The **symmetric hull** *of G is defined as*

$$G^{sym} = (V, R \cup R^{-1}, \alpha, \omega) \ .$$

The **simple symmetric hull** *of G can be obtained from the symmetric hull by removing the loops and parallel edges.*

See Fig. 1.12.

Definition 10 *Let $G = (V, R, \alpha, \omega)$ a directed graph and $H = (V, E, \gamma)$ an undirected graph. The graph H is the graph* **assigned to** *G, if*

$$E := R \quad \textit{and} \quad \gamma(e) = \{\alpha(e), \omega(e)\} \ .$$

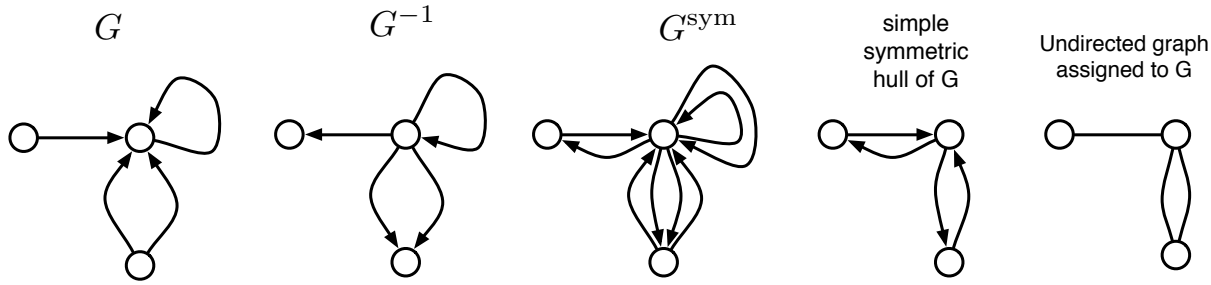


Figure 1.12: Inverse and symmetric hulls.

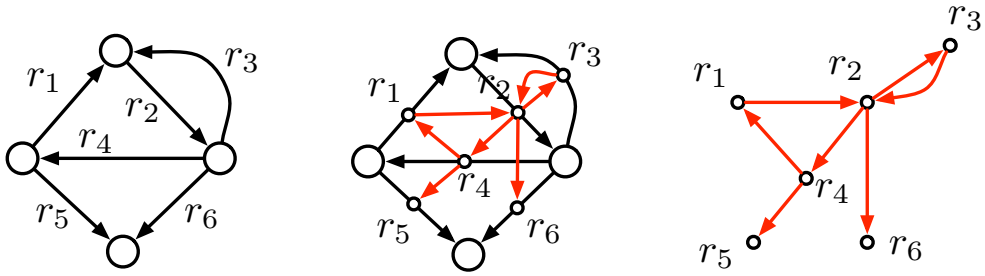


Figure 1.13: A digraph and its line graph.

1.3.4 Line Graphs

Definition 11 *Let $G = (V, R)$ be a simple finite graph with $R \neq \emptyset$. The* **Line Graph** *$L(G) = (V_L, R_L)$ is a simple graph with*

- $V_L = R$
- $R_L = \{(r, r') \ : \ r = (u, v) \in R \text{ and } r' = (v, w) \in R\}$.

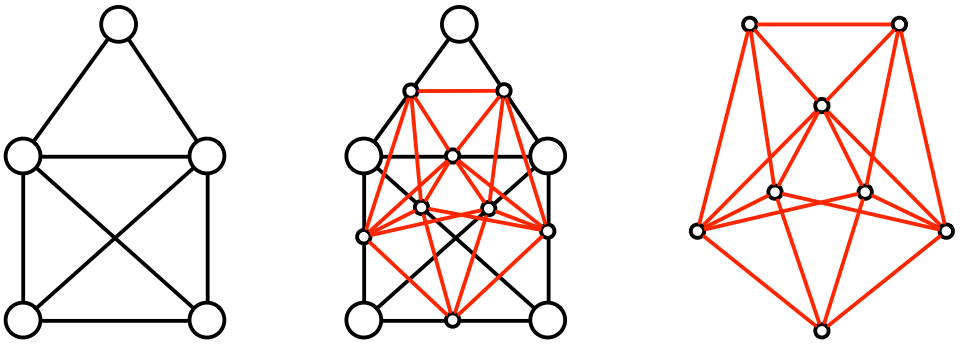


Figure 1.14: An undirected graph and its line graph.

See Fig. 1.13 for an example. Observe the following.

$$\begin{aligned} d^+_G(\omega(r)) &= d^+_{L(G)}(r) \\ d^-_G(\alpha(r)) &= d^-_{L(G)}(r) \end{aligned}$$

Another observation[6] is that an Euler circle in G implies a Hamiltonian circle in $L(G)$.

Definition 12 *For undirected simple graphs $H = (V, E)$ the* **Line Graph** $L(H) = (V_L, E_L)$ *is defined as*

- $V_L = E$
- $E_L = \{[e, e'] \ : \ e \text{ and } e' \text{ are incident in } G\}$

Note that

$$\begin{aligned} d_{L(H)}([u, v]) &= d_H(u) + d_H(v) - 2 \\ \Delta(L(H)) &\leq 2(\Delta(G) - 1) \end{aligned}$$

The number of edges incident with $v \in H$ in E_L is $\binom{d_H(v)}{2}$.

$$\begin{aligned} |E_L| &= \sum_{v \in V} \binom{d_H(v)}{2} \\ &= \frac{1}{2} \sum_{v \in V} d(v)(d(v) - 1) \\ &= \frac{1}{2} \sum_{v \in V} d(v)^2 - \frac{1}{2} \sum_{v \in V} d(v) \\ &= \frac{1}{2} \sum_{v \in V} d(v)^2 - |E| \end{aligned}$$

Different graphs can result in the same line graph.

1.4 Storing Graphs

1.4.1 Graphs for Turing Machines

Turing Machines uses tapes as a memory model, where only single characters can be placed in each cell. In order to store a digraph $G = (V, E)$ on such a TM we re-encode the node set using binary numbers:

$$V = \{1, 2, 3, 4\} = \{1, 10_2, 11_2, 100_2\}$$

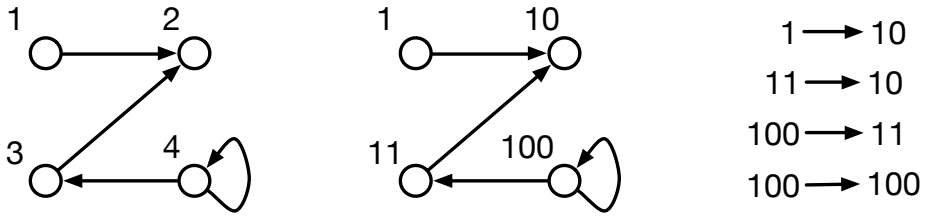
So, for an example edge set

$$E = \{[1, 2], [3, 2], [4, 3], [4, 4]\}$$

the full graph G can be stored on the tape as

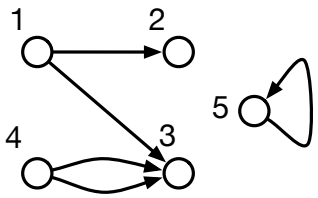
```
({1,10,11,100},{[1,10],[11,10],[100,11],[100,100]})
```

[6] we can value it as soon as we know what "Euler" and "Hamiltonian" means




```
({1,10,11,100},{(1,10),(11,10),(100,11),(100,100)})
```

Figure 1.15: How to store an undirected graph on a tape of a Turing machine



$$A(G) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 1.16: A digraph and its adjacency matrix.

1.4.2 Adjacency Matrix

From now on we consider the directed graph $H = (V, E, \gamma)$, $G = (V, R, \alpha, \omega)$ with $V = \{v_1, \ldots, v_n\}$, $R = \{r_1, \ldots, r_m\}$, or $E = \{e_1, \ldots, e_m\}$.

Definition 13 (Adjacency matrix) *For directed graphs $G = (V, R, \alpha, \omega)$ the $n \times m$ matrix $A(G)$ is called the* **adjacency matrix**

$$a_{ij} = |\{r \in R \,:\, \alpha(r) = v_i \text{ and } \omega(r) = v_j\}| \,.$$

For undirected graphs the $n \times m$ matrix $A(G)$ the **adjacency matrix** *is defined as*

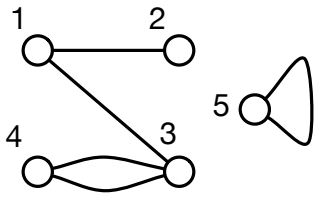
$$a_{ij} = |\{e \in E \,:\, \gamma(e) = \{v_i, v_j\}| \,.$$

Let $n = |V|$ be the number of node, $m = |R|$ or $m = |E|$ be the number of edges. Then $\Theta(n^2)$ bits are needed to store a graph without parallels using the adjacency matrix.

Weighted Graphs

For each edge $[i, j]$ in a graph without parallels a weight function $c(i, j) \in \mathbb{R}$ is defined. This is helpful to describe distances of an edge, the flow capacity, or other properties. It turns out that storing this value in the adjacency matrix of a graph without parallels results in a matrix which can be directly used to solve certain problems, e.g. shortest paths.

Then, we store $a_{ij} = c(i, j)$, if the edge $[i, j]$ exists and otherwise a special simbol $a_{i,j} =$ `nil`.



$$A(H) = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 1.17: An undirected graph and its adjacency matrix.

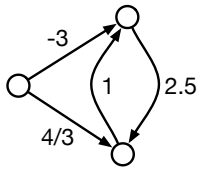


Figure 1.18: A weighted directed graph.

1.4.3 Incidence Matrix

Given a digraph G without loops we define the $n \times m$-**Incidence matrix** I as

$$i_{k\ell} := \begin{cases} 1 & \text{if } \alpha(r_\ell) = v_k \\ -1 & \text{if } \omega(r_\ell) = v_k \\ 0 & \text{elsewhere} \end{cases}$$

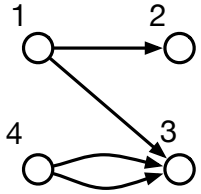
For an undirected graph H we define the Incidence matrix as:

$$i_{k\ell} := \begin{cases} 1 & \text{if } v_k \in \gamma(e_\ell) \\ 0 & \text{elsewhere} \end{cases}$$

Every quadratic submatrix of $I(G)$ is uni-modal, i.e. $\det(M) \in \{-1, 0, 1\}$.

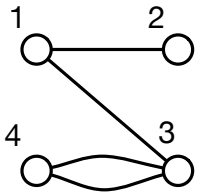
1.4.4 Adjacency List

The **Adjacency List** is an array *ADJ* consisting of n adjacency lists *ADJ*$[v]$, which is a pointered list of nodes w with $w \in N^+(v)$ for digraphs and $w \in N(v)$ for undirected graphs.



$$I(G) = \begin{pmatrix} 1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 1.19: A digraph and its incidence matrix.



$$I(G) = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 1.20: A digraph and its incidence matrix.

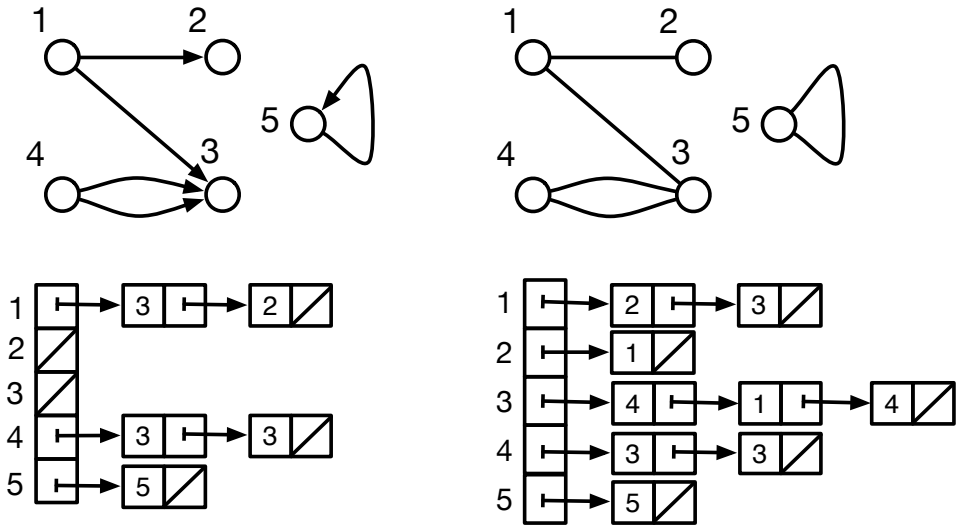


Figure 1.21: The adjacency lists of directed and undirected graphs

Inserting a node or an edge into this list can be done in constant time. Erasing an edge needs time $g^+(v)$. This data structure needs $\Theta(n+m)$ pointers. It is recommended for very sparse matrices $m \ll n^2$, i.e. for planar graphs where $m \leq 3n-6$ (for $n \geq 3$).

Chapter 2

Paths, Cycles, and Connectivitiy

2.1 Paths

For directed graphs the notion of path comes from following the directed arcs along their directions.

Definition 14 *A* **path** *in G is a finite sequence $P = (v_0, r_1, v_1, \ldots, r_k, v_k)$ for $k \geq 0$ where*

1. $v_0, \ldots, v_k \in V(G)$
2. $r_1, \ldots, r_k \in R(G)$
3. $\alpha(r_i) = v_{i-1}, \ \forall i \in \{1, \ldots, k\}$
4. $\omega(r_i) = v_i, \ \forall i \in \{1, \ldots, k\}$.

Analogously for undirected graphs a path follows incident edges.

Definition 15 *A* **path** *in H is a finite sequence $P = (v_0, e_1, v_1, \ldots, e_k, v_k)$ for $k \geq 0$ where*

1. $v_0, \ldots, v_k \in V(G)$
2. $e_1, \ldots, e_k \in E(G)$
3. $\gamma(e_i) = \{v_{i-1}, v_i\} \ \forall i \in \{1, \ldots, k\}$.

The **initial node** of P is $\alpha(P) := v_0$, the **terminal node** of P is $\omega(P) := v_k$. We say that P **connects** v_0 with v_k. The **length** of P is $|P| = k$. A path is a **cycle** if $\alpha(P) = \omega(P)$ and $|P| \geq 1$. The nodes of a path[1] are denoted by $V(P) = \{v_0, \ldots, v_k\}$. Note that empty paths with $|P| = k = 0$ starting and ending at the same node exist.

A path is **simple**, also called **a walk**, if $r_i \neq r_j$ for all $i \neq j$ (and $e_i \neq e_j$ for undirected graphs).

[1] In the German lecture this was called "Spur" and abbreviated by $s(P)$

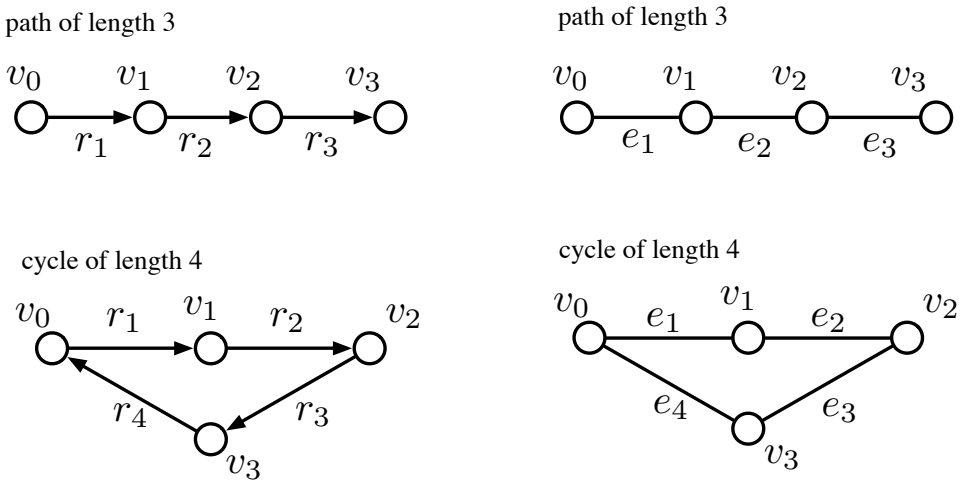


Figure 2.1: Paths and cycles in directed and undirected graphs

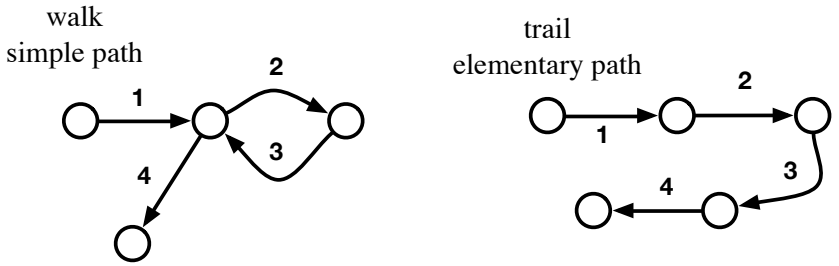


Figure 2.2: A walk and a trail

A path is **elementary**, also called **a trail**, if it is simple and all nodes are distinct with the exception of the first and last node.

For trails we have

$$|P| \leq |V(G)|$$

and if P is not a cycle we have for trails

$$|P| \leq |V(G)| - 1 \, .$$

Two paths $P = (v_0, r_1, v_1, \ldots, r_k, v_k)$, $P = (v'_0, r'_1, v'_1, \ldots, r'_k, v'_k)$ with $\omega(P) = v_k = \alpha(P') = v'_0$ are **concatenated** to

$$P \circ P' := (v_0, r_1, v_1, \ldots, r_k, \underbrace{v_k}_{=v'_0}, r'_1, v'_1, \ldots, r'_k, v'_k) \, .$$

P, P' are **sub-paths** of $P \circ P'$.

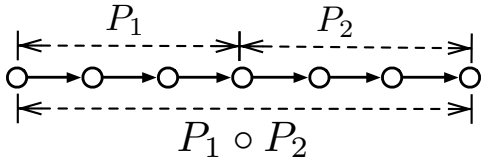


Figure 2.3: Paths $P_1, P_2, and P_1 \circ P_2$

Lemma 3 *For a finite digraph G with $\delta(G) \geq 1$, i.e. $\forall v \in V(G) \ : \ d^+(v) \geq 1$, there exists an elementary cycle in G.*

If G is simple and $\delta^+(G) \geq g \geq 1$, then there exists an elementary cycle of length of at least $g + 1$.

Proof: Let $P = (v_0, r_1, v_1, \ldots, r_k, v_k)$ the longest trail (elementary path). It exists since a single edge constitutes already a trail.

The premisse implies $g^+(v_k) \geq 1$, therefore there exists at least edges $r'_1, \ldots, r'_g$ with $\alpha(r'_j) = v_k$. Then we have the following cases for $r \in \{r'_1, \ldots, r'_g\}$ with $\alpha(r) = v_k$:

1. $\omega(r) \in \{v_0, \ldots, v_{k-1}\}$ this can happen. For simple graphs G it happens at most k times.
2. $\omega(r) = v_k$ is impossible, if G is simple, since it otherwise establishes a loop.
3. $\omega(r) \notin \{v_0, \ldots, v_k\}$ is imposible, since P is maximal.

We choose the smallest i with $\alpha(r'_j) = v_i$. In a graph with loops or parallels, we already get an elementary cycle possible of length 1, i.e. a loop.

For simple graphs we observe $i \leq k - g$. Then $(v_i, r_{i+1}, \ldots, r_k, v_k, r'_j, v_i)$ is a cycle of length of at least $g + 1$. □

2.2 Directed Acyclic Graphs

As a motivation see the work plan for an average day, see Fig. 2.4

Definition 16 *A directed graph is called* **acyclic***, if it does not contain a cycle as a sub-graph.*

We use the abbreviation **DAG** for directed acyclic graphs.

Definition 17 *Let $G = (V, R, \alpha, \omega)$ a digraph. A* **topological sorting** *of G is a bijective mapping $\sigma : V \to \{1, 2, \ldots, n\}$ with $\sigma(\alpha(r)) < \sigma(\omega(r))$ for all $r \in R$.*

Topological sortable digraphs are DAGs as the following theorem shows.

Theorem 2 *A digraph G is acyclic if and only if it can be topologically sorted.*

Proof:

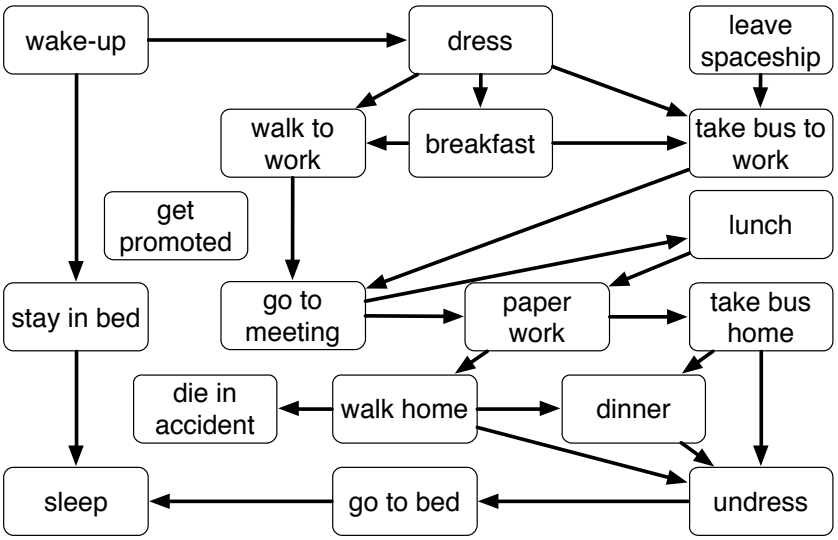


Figure 2.4: A directed acyclic graph

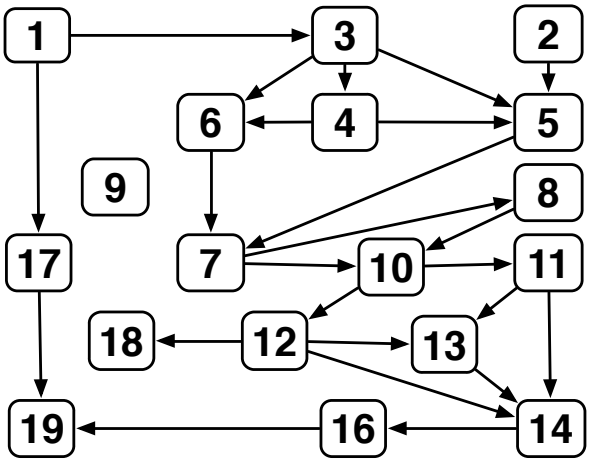


Figure 2.5: A topological sorting

"⇒" We use an induction over the number of nodes $n = |V(G)|$.

If $n = 1$, then we have a graph without loops, which can be sorted (by assigning 1 the single node).

If $n > 1$, a node $v \in V$ must exist with $d^-(v) = 0$. Since otherwise we can construct a cycle by Lemma 3.

From the induction $G' = G - v$ has a topological sorting σ'. Now define $\sigma(v) = 1$ and $\sigma(u) = \sigma'(u) + 1$ for $u \neq v$.

"⇐" Let σ be a topological sorting and $C = (v_0, r_1, v_1, r_2, \ldots, r_k, v_k)$ a cycle (i.e. $v_k = v_0$) But this implies the following contradiction

$$\sigma(v_0) < \sigma(v_1) < \ldots < \sigma(v_k) = \sigma(v_0) \; .$$

□

This proof is the basis for the following algorithm.

Algorithm 1: Topological Sorting

Input: digraph G as adjacency list;
Output: topological sorting function $\sigma : V \to \{1, \ldots, n\}$;
Compute `in-degree`$[v]$ for all $v \in V$;
$L_0 := \{v \in V : \texttt{in-degree}[v] = 0\}$;
for $i = 1, \ldots, n$ **do**
  Remove the first node $v \in L_0$;
  $\sigma(v) := i$;
  for $r \in \delta^+(v)$ **do**
    `in-degree`$[\omega(r)] :=$ `in-degree`$[\omega(r)] - 1$;
    if `in-degree`$[\omega(r)] = 0$ **then**
      Insert $\omega(r)$ into L_0
    end
  end
end

Theorem 3 *Algorithm 1 computes in time $\mathcal{O}(n + m)$ a topological sorting of G.*

2.2.1 Connectivity

Definition 18 *A node w is* **reachable** *from v, if there exists a path P with $\alpha(P) = v$ and $\omega(P) = w$ or $w = v$.*

$R_G(v)$ denotes the set of all nodes reachable from v.[2]

[2] R_G is denoted as E_G in the German lecture.

Algorithm 2: Reachable(G, s, p)

```
Input: digraph G as adjacency list, s ∈ V(G), p ∈ ℕ;
Output: R_G(s);
mark[s] := p;
for v ∈ V \ {s} do
    mark[v] := nil;
end
L := (s);
while L ≠ () do
    Remove first node v from L;
    for u ∈ Adj[v] do
        if mark[u] = nil then
            mark[u] := p;
            Insert u to the end of L;
        end
    end
end
return R_G(s) := {v ∈ V : mark[v] = p};
```

Theorem 4 *Algorithm 2 computes $R_G(s)$ in time $O(n + m)$.*

Proof: Correctness:

"$\Leftarrow$" We assume for an induction, that a node u is marked if there is a path from s to a predecessor of u or if $u = s$.

The induction is proved over the shortest path from s to u. If the assumption holds for shortest paths of length ℓ, then it follows for shortest paths of length $\ell + 1$.

"$\Rightarrow$" No nodes are marked for which there does not exist any path from s.

□

Observe that in fact we compute the shortest path, since Algorithm 2 is a breadth-first-search (BFS) algorithm, since we use the list as a queue. If we would insert v at the beginning of L, we would implement a stack. Then, the resulting algorithm performs a depth-first-search (DFS), which we will revisit later on.

Definition 19 *Let G a (directed or undirected) graph and $v, w \in V(G)$. Then, define v and w* **(strongly) connected**, *$v \leftrightarrow w$ if $v \in R_G(w)$ and $w \in R_G(v)$.*

We call the **connected component**[3] *of v*

$$CC_G(v) := \{w \in V(G) \ : \ v \leftrightarrow w\} .$$

If $CC_G(v) = V(G)$, then we call G **strongly connected**.

[3] In the German lecture we use ZK for CC.

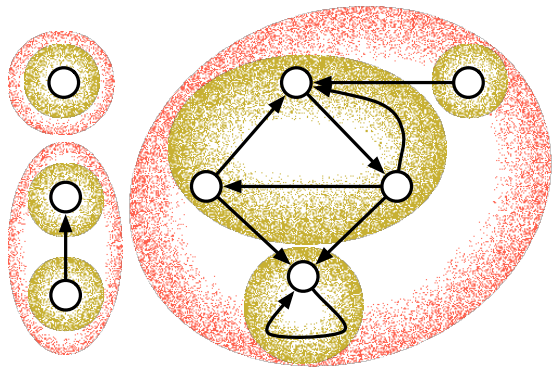


Figure 2.6: (Strongly) connected (yellow) and weakly connected (red) components of a graph

For directed and undirected graphs we use this definition. For directed graphs we also consider the notion of weak connectivity.

Definition 20 *Let G a digraph, $v, w \in V(G)$. Then, u, v are* **weakly connected***, i.e. $u \overset{w}{\leftrightarrow} v$, if $u \leftrightarrow v$ in G^{sym}.*

We call the **weak connected component** *of v*

$$WCC_G(v) := \{u \in V(G) \ : \ u \overset{w}{\leftrightarrow} v\} \ .$$

If $WCC_G(v) = V(G)$, then we call G **weakly connected***.*

Lemma 4 *The relation $\leftrightarrow$ is an equivalence relationship.*

Proof:

1. Reflexivity: $v \leftrightarrow v$ for all $v \in V(G)$ follows by definition.
2. Symmetry: $v \leftrightarrow w \quad \Leftrightarrow \quad w \leftrightarrow v$ follows by definition.
3. Transitivity: $u \leftrightarrow z$ and $z \leftrightarrow w$ implies $u \leftrightarrow w$.

   There is a path from u to z, called P_1, and there is a path from z to w, called P_2. So, $P_1 \circ P_2$ is a path from u to v. By the same argument we find a path from v to u.

□

Note the fact, since R is equivalence relation for H, then the equivalence classes describe a partition of the graph. So, for the connected components $C_1, C_2, \ldots, C_k$ we have

$$\bigcup_{i \in \{1,\ldots,k} C_i = V(G) \ , \quad C_i \cap C_j = \emptyset \ , \ \forall i \neq j \ .$$

Algorithm 3: Connected components of an undirected graph G

Input: undirected graph G;
Output: connected components $V_1, \ldots, V_p$;
$\forall v \in V \ : \ \texttt{mark}[v] := \texttt{nil}$;
$p := 1$;
for $j := 1, \ldots, n$ **do**
  if $\texttt{mark}[v_j] = \texttt{nil}$ **then**
    p:= p+1;
    Reachable(G, v_j, p);
  end
end
return G has p connected components $V_1, \ldots, V_p$, where $V_i = \{v \in V \ : \ \texttt{mark}[v] = i\}$;

Theorem 5 *The connected components of a graph can be computed in time $O(n + m)$.*

Due to lack of time we do not prove this theorem based on Algorithm 3, see [KN09].

Lemma 5 *Let $u, v \in V(G)$ with $u \in CC_G(v)$.*

1. *Every path from u to v visits only nodes in $CC_G(v)$.*
2. *There exists a cycle[4], which visits all edges in $CC_G(v)$, if $|CC_G(v)| > 0$.*

2.3 Euler Paths and Cycles

Definition 21 *A path P is called an* **Euler path***, if P is a simple path visiting all edges of the graph G. If P is a cycle, then P is an* **Euler cycle***.*

A graph G is called **Eulerian***, if there exists an Euler cycle within G.*

Theorem 6 (Euler I) *For finite directed and weakly connected graph G with at least one edge we have is Eulerian, if*

$$G \text{ is Eulerian if and only if } \quad \forall v \in V \ : \ d^+(v) = d^-(v) \ .$$

Proof:

"⇒" Let G be Eulerian and $|R| \ > 0$, then let $P = (v_0, r_1, v_1, \ldots, r_k, v_0)$ be an Eulerian cycle. Now, every node with $d^+(v) > 0$ or $d^-(v) > 0$ must occur at least once. Every occurrence contributes to an exactly one increment to the in-degree and one increment to the out-degree. Therefore $d^+(v) = d^-(v)$ for all nodes v.

[4] which is not necessary simple nor elementary

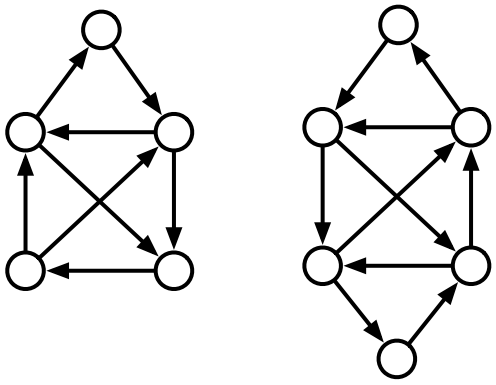


Figure 2.7: A graph with an Euler path and a graph with an Euler cycle

"$\Leftarrow$" We use an induction over $|R|$ with basis $|R| = 1$. For this edge r we must observe $\alpha(r) = \omega(r)$ which is a loop and also the smallest possible Euler cycle.

For the inductive step assume that G is weakly connected and $g^+(v) = g^-(v)$ for all $v \in V$, where $|R| = k > 1$. We now proof that such graphs contain an Eulerian cycle.

We choose a node v_0 and an edge r_1 with $\alpha(r_1) = v_0$. We construct a simple path as follows. Let $v_1 = \omega(r_1)$. Now from v_i, we continue and choose an arbitrary unused edge r_{i+1}, reaching a node v_{i+1} until we face only used edges at the node v_k. Then, we have returned to v_0, i.e. $v_0 = v_k$.

The last statement is true, since otherwise, i.e. $v_k \neq v_0$, the number of outgoing edges at v_k must be smaller than the number of incoming edges at v_k, i.e. $d^+(v_k) < d^-(v_k)$, since we used at v_k incoming and outgoing edges at the same quantity. This would contradict the precondition.

This cycle is not necessarily an Euler cycle. If it is, the proof is complete, otherwise consider the sub-graph $G' = G - \{r_1, \ldots, r_k\}$ and its connected connectivity components $C_1, \ldots, C_k$. If there is a connectivity component C' which cannot be reached from P in the corresponding undirected graph, then G is not weakly connected. By induction each of the induced graphs containing at least one edge has a Eulerian cycle, since $d^+_{G'(C_i)}(v) = d^-_{G'(C_i)}(v)$ for all nodes v.

We can combine all these $k + 1$ cycles to a single cycle, since G is weakly connected (in fact we prove that it is now even strongly connected). This cycle is Eulerian completing the proof.

□

Corollary 1 *Every finite directed and weakly connected graph with $d^+(v) = d^-(v)$ for all $v \in V$ is strongly connected.*

Note that the above theorem gives a linear time algorithm to construct Eulerian paths. For this we use an array `current` : $V \mapsto$ first entry in the adjacency lists `ADJ`$[v]$.

Algorithm 4: Explore (G, v, u)

Input: directed graph G, $v, u \in V(G)$;
Output: Cycle C.;
$K := (v, r_{vu}, u)$;
while $u \neq v$ **do**
  $w :=$ `current`$[u]$ (* the next entry in `ADJ`$[v]$ *);
  Advance `current`$[u]$ in the adjacency lists `ADJ`$[u]$;
  $K := K \circ (u, r_{uw,}, w)$;
  $u := w$;
end
return K;

Algorithm 5: Euler (G)

Input: directed graph G, $v \in V(G)$;
Output: Euler cycle K;
for $v \in V$ **do**
  `current`$[v]$:= first element in `ADJ`$[v]$ or `nil` if it is empty;
end
Choose any $v_0 \in V$;
$C := (v_0)$;
$v := v_0$;
repeat
  while `current`$[v]$ $\neq$ `nil` **do**
    K:= Explore(G,v, `current`[v]) (* `current`[v] is changed by Explore *);
    Insert K in C at the position v;
  end
  $v :=$ is the node following v in C;
until $v = v_0$;
return C;

There is a second Euler theorem which deals with Euler paths.

Theorem 7 (Euler II) *For a finite directed weakly connected graph G we have that G hosts an Euler path, which is no cycle, if and only if there exists $s, t \in V(G)$ such that*

$$\begin{aligned} d^+(s) &= d^-(s) + 1 \,, \\ d^+(t) &= d^-(t) - 1 \,, \\ d^+(v) &= d^-(v) \quad \forall v \in V \setminus \{s, t\} \,. \end{aligned}$$

Then s and t are the initial and terminal nodes of the Euler paths.

Proof:

"$\Rightarrow$" This can be proved analogously to the first theorem of Euler by counting in- and out-degrees.

"⇐" Insert an edge r^* with $\alpha(r^*) = t$ and $\omega(r^*) = s$, then $d^+(v) = d^-(v)$ for all $v \in V$ and the first theorem of Euler implies that there is an Euler cycle in G'. By removing r^* we get the Euler path in G.

□

Theorem 8 (Euler III) *A finite undirected connected graph with at least one edge*

1. *is Eulerian, if and only if all nodes have even degree.*
2. *possesses a Euler path, which is no cycle, if and only if exactly two nodes have odd degree.*

Proof: (sketch)

"⇒" Again we can prove it by counting the in- and out-degree.

"⇐" Let $P = (v_0, \ldots, v_k)$ be a simple path of maximum length. Assume that P is not Eulerian cycle. Then, if P is not a cycle, then P is not maximal. If there is an edge e in G with $e \notin V(P)$. This would imply that G is not connected. There are nodes in P with an edge not in P then P is not maximal.

□

2.4 Hamiltonian Graphs

Definition 22 *An elementary path in G is called* **Hamiltonian***, if every node of G occurs in P. A* **Hamiltonian cycle** *is an elementary cycle visiting all nodes (exactly once).*

A graph is **Hamiltonian** *if it contains a Hamiltonian cycle.*

Currently, no efficient algorithm is known to decide whether a given graph is Hamiltonian. For special cases this question can be decided.

Theorem 9 (Theorem of Dirac) *Let G be a simple undirected graph with $\delta(G) \geq \frac{n}{2}$ and $n := |V| \geq 3$. Then G is Hamiltonian.*

Proof: First we prove that G is connected. Consider the smallest connected component C. If $|C| < \frac{n}{2}$ this would contradict the assumption that $\delta(G) = \min_v\{d(v)\} \geq \frac{n}{2}$ and that G is simple. If $|C| \geq \frac{n}{2}$ then either there exists a smaller connected component (which contradicts the assumption that C is the smallest one) or $C = V$. So, only the latter case can be true.

Let P be the longest elementary path in C. Let $P = (v_0, \ldots, v_k)$. First observe that $k \geq \frac{n}{2}$, since otherwise P is not maximal, since v_k has at least $\frac{n}{2}$ neighbors.

Because v_0 and v_k have at least $n/2$ neighbors, there must be an $i \in \{0, \ldots, k-1\}$ such that $(v_0, v_{i+1}) \in E$ and $(v_i, v_k) \in E$ by a counting argument.

Now, we claim that $C = (v_0, v_{i+1}, v_{i+2}, \ldots, v_k, v_i, v_{i-1}, \ldots, v_0)$ is a Hamiltonian cycle. Otherwise note that since G is connected there is an edge from a node of C to a node $v \notin V(C)$. In that case the elementary path was not optimal, since the cycle C could have split up at the neighbored node of v in C and increased to length $k + 1$.

□

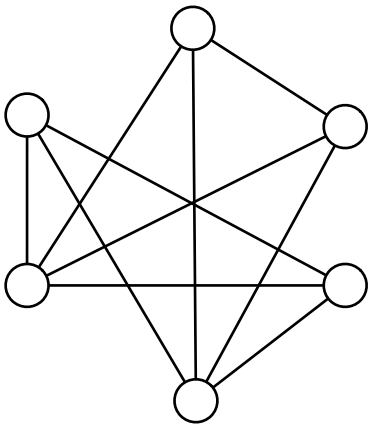


Figure 2.8: A graph G with $\delta(G) \geq n/2$

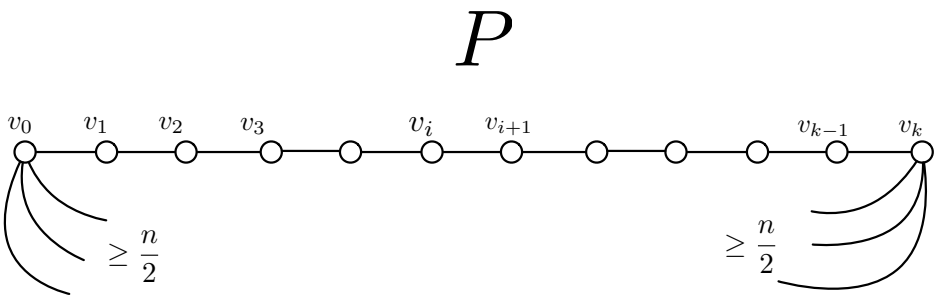


Figure 2.9: The longest elementary path P

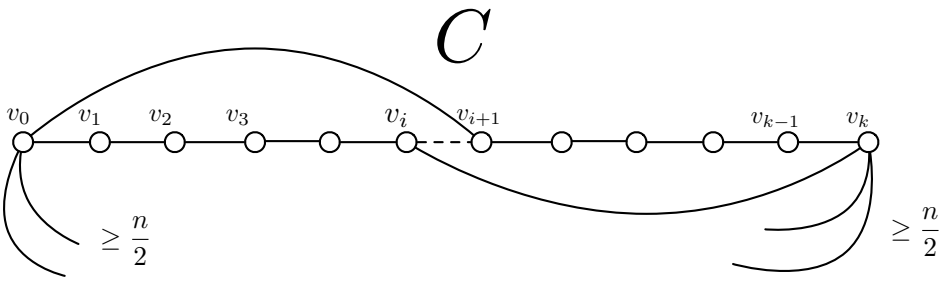


Figure 2.10: An Hamiltonian cycle V built from P

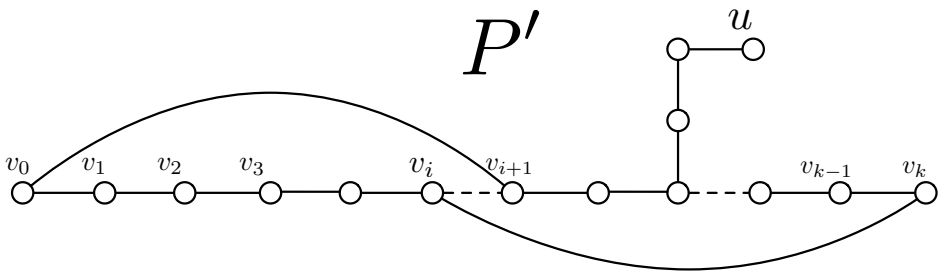


Figure 2.11: If C is not Hamiltonian, then P' would be longer than P

Chapter 3

Trees

3.1 Trees and Forests

Definition 23 *An undirected graph G is called a* **forest***, if G is acyclic.*
If G is also connected, it is called a **tree***.*

Theorem 10 *The following statements are equivalent for undirected trees*

1. *G is a tree.*
2. *G contains no elementary cycle, but every proper super graph of G with the same vertex set contains an elementary cycle.*
3. *For every pair $u, v \in V$ there exists exactly one path with $\alpha(P) = u$, $\omega(P) = v$.*
4. *G is connected and $|E| = |V| - 1$.*
5. *G contains no elementary cycle and $|E| = |V| - 1$.*

Definition 24 *A directed graph G is called r-***rooted tree** *if*

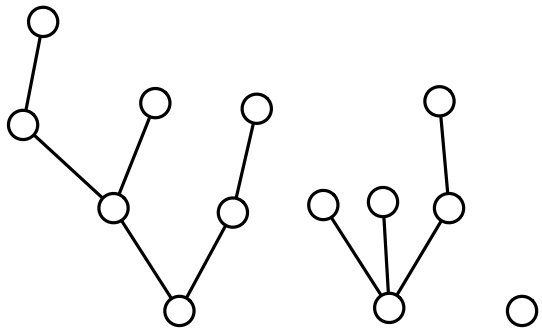


Figure 3.1: Three undirected trees or a first of three tree

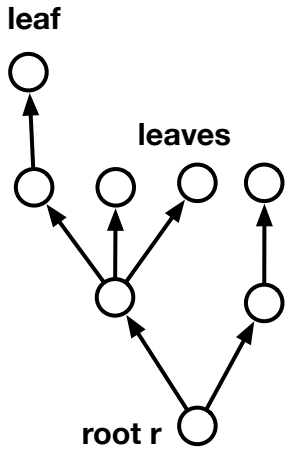


Figure 3.2: A rooted tree

- *G is simple,*
- *the corresponding undirected graph is a tree,*
- *there exists root node $r \in V(G)$ for which $\forall v \in V : v \in R_G(r)$.*

The leaves are the nodes with $d^+(v) = 0$. v is the predecessor of u, if v is on the path from r to u. u is the successor of v, if v is the predecessor of u.

u is the father of v if $N_G(u) = v$. v is the son of u if $N_G(u) = v$.

Theorem 11 *The following claims are equivalent for directed trees*

1. *G is a s-rooted tree.*
2. *The corresponding undirected graph G is a tree, $d^-(s) = 0$, $\forall v \in V \setminus \{s\} : d^-(v) = 1$.*
3. *$d^-(s) = 0$, $\forall v \in V \setminus \{s\} : d^-(v) \leq 1$, and $R_G(s) = V$.*

3.2 Depth First Search

The goal of the DFS-algorithm is to visit all nodes in a graph. There are two basic strategies for exploration starting from a node. First explore the depth of the tree leading to Depth first search (DFS) or to explore all nodes in the same depth and then proceed to deeper nodes: Breadth first search (BFS).

For DFS we temporarily color the nodes with three colors:

- white: At the beginning all nodes are white, which means that they are unexplored.

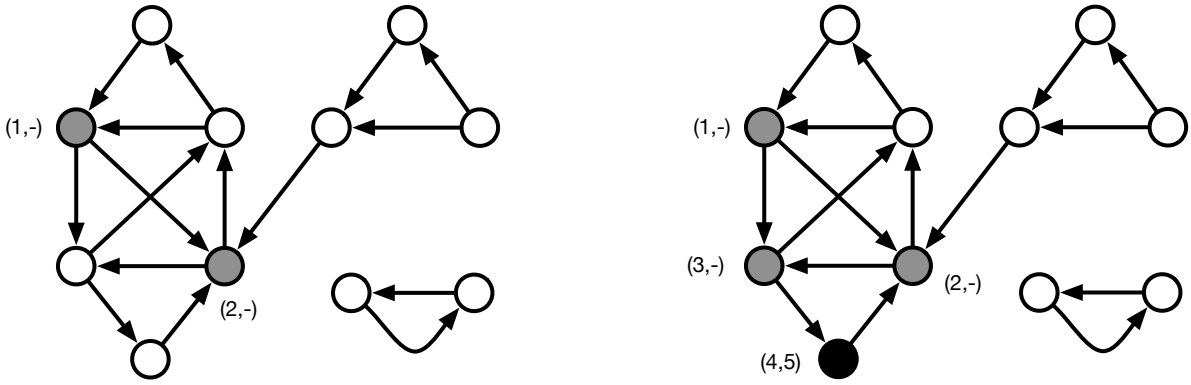


Figure 3.3: Start of the DFS algorithm discovery and finishing times in brackets

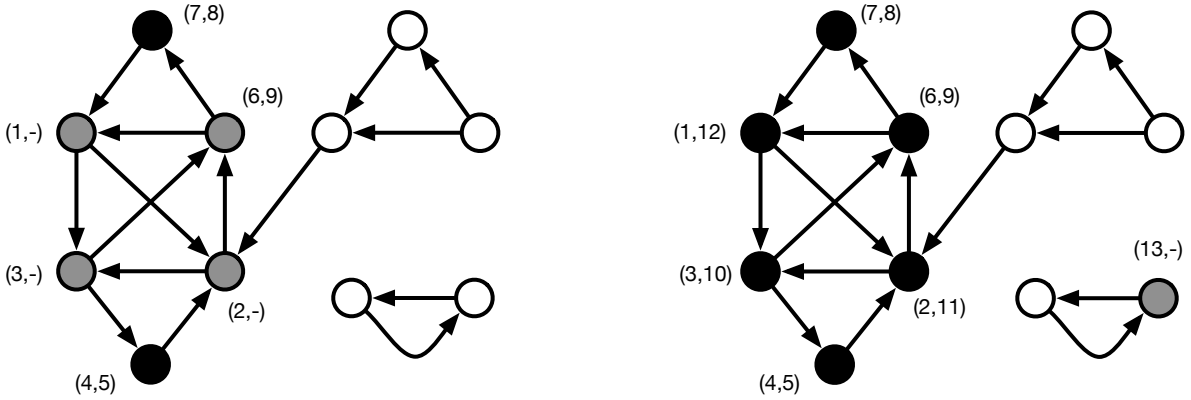


Figure 3.4: Black, grey and white nodes

- grey: When we explore and handle a node we color it grey in order to show that we are working on it and on all its neighbors. So, grey means it is currently explored and there is work in progress.

- black: When we have explored all nodes neighbored of the current grey node, we color this node black.

We keeping track of the exploration time. For this we increment a time counter whenever we color a node and write down the time in the arrays d and f. We denote by $d[v]$ the discovery time from $\{1, 2, \ldots, n\}$ and $f[v]$ denotes the finishing time. This is the time, when $N(v)$ has been processed. When the current time is in the interval $I(v) = [d(v), f(v)]$, then v is grey.

The DFS-algorithm consists of a main part shown as Algorithm 6, which invokes the DFS-VISIT Algorithm 7 for each node (if the node has been already visited before). The result is a simple directed graph $G_\pi = (V, R_\pi)$.

Algorithm 6: DFS main part

Input: directed graph $G = (V, R, \alpha, \omega)$;
Output: $G_\pi = (V, R_\pi)$;
for $v \in V$ **do**
  color$[v]$:= `white`;
  $\pi[v]$:= `nil`;
end
$R_\pi := \emptyset$;
time := 0;
for $v \in V$ **do**
  if *color*$[v]$ = `white` **then**
    DFS-VISIT(v);
  end
end

Algorithm 7: DFS-VISIT(u)

Input: directed graph $G = (V, R, \alpha, \omega)$, $v \in V$;
color$[u]$:= `grey`;
$d[u]$:= *time*;
time := *time* + 1;
for $v \in$ *ADJ*$[u]$ **do**
  if *color*$[v]$ = `white` **then**
    $\pi[v] := u$; /* u is the predessor of v in the DFS forest */
    $R_\pi := R_\pi \cup \{r_{uv}\}$;
    DFS-VISIT(v);
  end
end
color$[u]$:= `black`;
$f[u]$:= *time*;
time := *time* + 1;

Observation: The run-time of DFS (Algorithm 6 using Algorithm 7) is $O(n + m)$.

Theorem 12

1. *G_π is a forest.*
2. *Every weak connected component of G_π is a rooted tree.*

Proof: We have $d^-_{G_\pi}(v) \leq 1$ for all $v \in V$, because an edge r_{uv} is only added, if v has been white before, i.e. the node is added only once.

For every edge $r \in R_\pi$ we have

$$d[\alpha(r)] < d[\omega(r)] \ .$$

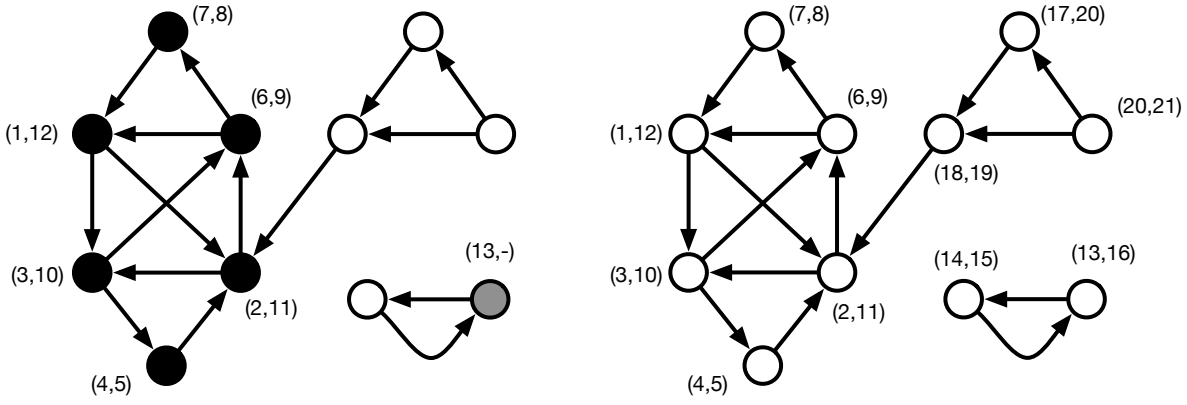


Figure 3.5: Left: first connected component is explored the main part has to start agin. Right: DFS is finished.

Hence, G_π is acyclic.

Let T be a weakly connected component in G_π. Because G_π is acyclic there exists a node with $d^-(s) = 0$.

We want to prove: $R_T(s) = V(T)$. In the corresponding undirected graph of T there is exactly a path from s to $v \in V[T]$:

$$P = (\underbrace{s}_{v_0}, r_1, v_1, r_2, v_2, \ldots, r_k, \underbrace{v}_{v_k}) .$$

From $d^-(s) = 0$ it follows $r_1 = (v_0, v_1)$, from $d^-(v_1) \leq 1$ follows $r_2 = (v_1, v_2)$ etc. Therefore P is a directed path from s to v.

So, $d^-(s) = 0$ and $d^-(v) \leq 1$ for all $v \in V$, and $E_T(s) = V(T)$ implies that T is a s-rooted tree. □

Theorem 13 (Interval theorem) *Let $u, v \in V$ with $d[u] \leq d[v]$ then after a DFS search we have either*

1. *$I(v) \subset I(u)$ and v is sucessor of u or*
2. *$I(v) \cap I(u) = \emptyset$.*

Due to lack of time we omit the proof, which can be found in [KN09].

Theorem 14 (Theorem of the white path) *The node v is a successor of u in the DFS-tree G_π if and only if, at the time point $d[u]$ a path from u to v exists consisting only of white nodes.*

We refer to [KN09] for a proof.

Corollary 2 *All nodes in a strong connected component of G lie in the same DFS-rooted tree of G_π.*

Classification of edges in G:

- **tree edges**: edges of R_π
- **back edges**: edges where $\alpha(r)$ is a successor of $\omega(r)$
- **forward edges**: $r \in R \setminus R_\pi$: $\omega(r)$ is successor of $\alpha(r)$
- **cross edges**: the rest

Note that using the time intervals $[d[u], f[f]]$ the DFS algorithm can be modified to classify all nodes in time $O(n+m)$.

Algorithm 8: DFS-VISIT(u) that colors the nodes

```
Input: directed graph G = (V, R, α, ω), v ∈ V;
color[u] := grey;
d[u] := time;
time := time + 1;
for v ∈ ADJ[u] do
    if color[v] = grey then
        Mark [u, v] as back edge;
    else if color[v] = black and d[u] < d[v] then
        Mark [u, v] as forward edge;
    else if color[v] = black and d[u] > d[v] then
        Mark [u, v] as cross edge;
    else if color[v] = white then
        Mark [u, v] as tree edge;
        π[v] := u;        /* u is the predessor of v in the DFS forest */
        R_π := R_π ∪ {r_uv};
        DFS-VISIT(v);
    end
end
color[u] := black;
f[u] := time;
time := time + 1;
```

Theorem 15 *DFS produces back edges if and only if the underlying graph contains cycles.*

Proof:

"⇒" If r is a back edge, then $\omega(r)$ is a predecessor of $\alpha(r)$. Then, there exists in G_π and thus in G an elementary path from $\omega(r)$. Therefore $P \circ (\alpha(r), r, \omega(r))$ is an elementary cycle.

"⇐" Let C be a cycle in G with nodes $(v_0, v_1, \ldots, v_k = v_0)$ (in this order). Let $v_i \in V(C)$ be the first node of the cycle that is found by DFS. At time point $d[v_i]$ there is a white path from v_i to v_{i-1} (via the nodes of the cycles). Then by the Theorem of the white path v_{i-1} becomes a successor of v_i in G_π. Hence, the edge (v_{i-1}, v_i) is a back edge.

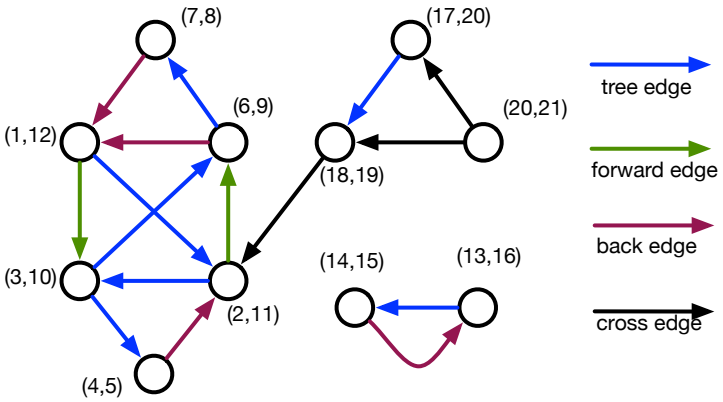


Figure 3.6: Tree, back, forward, and cross edges

□

Corollary 3 *Using DFS one can test in time $O(n+m)$ whether a given graph is acyclic, and if not DFS outputs an elementary cycle.*

We can use DFS to find an alternative way to topologically sort a DAG. For this we have a countdown variable i starting from n and if a node v turns black, we set $\sigma(v) := i$ and decrement i.

Theorem 16 *The above technique computes a topological sorting in time $O(n+m)$ provided that G is acyclic.*

For a proof sketch note that a node v can only turn black, if all nodes that could be reached from it, are already processed (and turned black). Further notice that the finishing time constitutes the reverse ordering of the topologic sorting.

3.3 Computing strong connected components

The observation of the ordering behavior can be used to compute the strong connected components of a graph with the following algorithm.

Algorithm 9: Computing strong connectivity components

Input: directed graph $G = (V, R, \alpha, \omega)$;
Output: strong connected components of G;
Invoke *DFS*(G) to compute $f[v]$ for $v \in B$;
Compute the inverse graph G^{-1};
Invoke *DFS*(G^{-1}) where the nodes are sorted with decreasing $f[u]$;
Output all *DFS*-trees of the last steps as strong connected compontents;

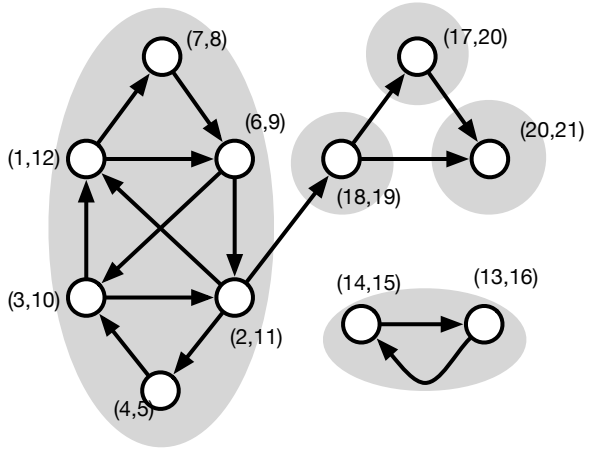


Figure 3.7: Computing the connected components using G^{-1}.

Theorem 17 *Algorithm 9 computes in time $O(n+m)$ the strong connected components of G.*

Proof sketch:

The run-time follows from the observation that all three major steps need run-time $O(n+m)$.

Consider a strong connected component C. DFS-visit will visit all nodes of the strong connected component. However, it is possible that during it processes the nodes of C it also visits some neighbored strong connected components.

For two neighbored strong connected components C_1, C_2, where all edges point from C_1 towards C_2 the interval theorem states, that all nodes of C_2 have finishing times, which are smaller than those from all nodes from C_1, if the DFS starts is in C_1 and then visits C_2. However, if the DFS starts in C_2, then it is finished and has to restart in some other connected component in order to reach C_1. So, the finishing times of nodes in C_2 al also smaller than those of C_1.

Now, if we consider G^{-1} the connected components stay the same and therefore it is not possible to reach a neighbored unexplored connectivity component any more. Since in the DAG given by the connected components and the induced graph, all edges point from smaller finishing times to larger finishing times. □

3.4 Minimum Spanning Trees

Remember that the partial graph is the induced subgraph based on the node set. So, the DFS algorithm computes a tree in an undirected component, which connects all possible nodes. This property is called *spanning*.

Definition 25 *A partial graph $H = (V, E', \gamma')$ of an undirected graph $G = (V, E, \gamma)$ is a* **spanning tree***, if H is a tree.*

A partial graph $H = (V, E', \gamma')$ is a **spanning forest** *of $G = (V, E, \gamma)$, if every connected component of H is a spanning tree of a connected component of G,*

Definition 26 *The* **Minimum Spanning Tree (MST)** *problem is the following: Given an undirected connected graph $G = (V, E, \gamma)$ with cost functions $c : E \to \mathbb{R}$ find a spanning tree T with minimal overall costs*

$$c(T) = \sum_{e \in E(T)} c(e) \ .$$

Definition 27 *The* **Minimum Spanning Forest (MSF)** *problem is the following: Given an undirected graph $G = (V, E, \gamma)$ with cost functions $c : E \to \mathbb{R}$ find a spanning forest F with minimal overall costs*

$$c(F) = \sum_{e \in E(F)} c(e) \ .$$

We want to construct such minimal trees, which can be done by a several intuitive algorithms.

Algorithm 10: Algorithm of Kruskal

Input: undirected graph $G = (V, R, \gamma)$, $c : E \to \mathbb{R}$;
Output: edges E_F of the MSF;
Sort edges according to weight, such that $c(e_1) \leq c(e_2) \leq \ldots \leq c(e_m)$;
$E_F := \emptyset$;
for $i := 1, \ldots, n$ **do**
  if $(V, E_F \cup \{e_i\}, \gamma)$ *is acyclic* **then**
    $E_F := E_F \cup \{e_i\}$;
  end
end
return E_F;

In order to prove the correctness of Kruskal's algorithm we need some notation and some understanding of minimum spanning trees.

Definition 28 *A subset $F \subseteq E$ is* **error free***, if there exists a MSF F^* of G such that $F \subseteq E(F^*)$. An edge is* **safe** *for F, if $F \cup \{e\}$ is error free.*

Definition 29 *A cut (A, B) is a node disjoint partition of the node set V, i.e. $A, B \subseteq V$, and $A \cup B = V$, $A \cap B = \emptyset$. We use the following definition for the edges on the cut:*

$$\delta(A) := \{[u, v] \in E : u \ \in A, v \in V \setminus A\}$$

Lemma 6 *Let $F \subseteq E$ be error free, and (A, B) be a cut in G with $\delta(A) \cap F = \emptyset$. If $e \in \delta(A)$ has the smallest cost $c(e)$ in $\delta(A)$, then e is safe for F.*

Proof: Consider the MSF with $F \subseteq F^*$ and $e \notin F^*$, where $c(e)$ is minimal for $\delta(A)$.

If we add e to F^* a cycle would exist, see Fig. 3.8. We can restore the forest property by removing an edge of the cut $\delta(A)$. If this edge e' has larger weight than e then the new forest has smaller weight and thus F^* was not minimal, which is a contradiction.

If this edge e' has smaller weight than e, then $c(e)$ was not minimal.

If this edge e' has the same weight, we get a new forest F' which is also minimal. This implies that e was safe for the tree. □

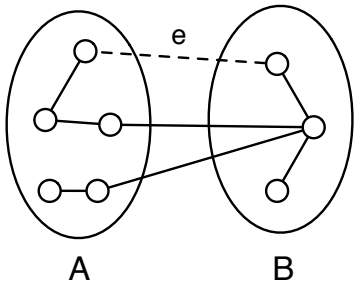


Figure 3.8: The edge with smallest weight between A and B is safe

Corollary 4 *Let $F \subseteq E$ be error free and let U be a connected component of (V, F). If e is an edge with minimal weight of $\delta(U)$ then e is safe for F.*

Now we can prove the correctness of Kruskal's algorithm:

Theorem 18 *The algorithm of Kruskal computes a MSF. If G is connected, it computes a MST.*

Proof: Since F is acyclic, F is a forest. If C is a connected component of G and F is not spanning for this component, then there is a subset $A \subset V(C)$ which is covered by F. Since C is connected, there is at least an edge between A and $V(C) \setminus A$. This edge does not introduce a cycle and from this edge set $\delta(A)$ at least one is minimal. Therefore $A = V(C)$ and therefore F is spanning.

Now this algorithm only adds safe edges, therefore the resulting forest is the MSF. □

Note that the run time depends on the data structure. We need an operation, which decides whether two elements are in the same set, while sets keep on getting unified. This is the Union-Find-problem and there is no linear time solution for it. However, one can show that for $m \geq n$ the run-time is $O(m \cdot \alpha(n))$, where $\alpha(n)$ denotes the inverse of the Ackermann function. Since $\alpha(n) \leq 4$ for all $n \leq 10^{684}$ and $\alpha(n) = o(\log \log n)$ the main contribution for the running time of $O(m \log m)$ is caused by sorting the edges at the beginning.

Another solution to the minimum spanning tree problem is the Prim's Algorithm 11.

Algorithm 11: Algorithm of Prim

Input: undirected connected graph $G = (V, R, \gamma)$, $c : E \to \mathbb{R}$;
Output: edges E_T of the MST;
select an arbitrary node $s \in V(G)$;
$E_T := \emptyset$;
$S := \{s\}$;
while $S \neq V$ **do**
    select an edge $[u, v] \in \delta(S)$ with $u \in S$, $v \in V \setminus S$ and $c([u, v])$ minimal;
    $E_T := E_T \cup \{[u, v]\}$;
    $S := S \cup \{v\}$;
end
return E_T;

Theorem 19 *The algorithm of Prim computes a MST.*

Proof: follows by Corollary 4. □

Using an appropriate data structure it is possible to implement Prim's algorithm within time $O(m + n \log n)$.

For unique cost functions we can use the algorithm of Borůvka 12, which is very well suited for parallel computation. So, we assume that for all $K \subseteq E$, there is only one edge $e \in K$ with $c(e) = \min\{c(e'), e' \in\}$, which we denote by $e = \arg\min_{e' \in K}\{c(e')\}$.

Algorithm 12: Algorithm of Borůvka

Input: undirected connected graph $G = (V, R, \gamma)$, $c : E \to \mathbb{R}$, $V = \{v_1, \ldots, v_n\}$;
Output: edges E_T of the MST;
$E_T := \emptyset$;
while (V, E_T) *contains* $p > 1$ *connected components* **do**
  let $V_1, \ldots, V_p$ be the node sets of the p connected components;
  for $i := 1, \ldots, p$ **do**
    $e_i := \arg\min_{e \in \delta(V_i)}\{c(e)\}$;
  end
  $E_T := E_T \cup \{e_1, e_2, \ldots, e_p\}$;
end
return E_T;

Theorem 20 *The algorithm of Borůvka computes a MST within run time* $O(m \log n)$.

Proof: Regarding the runtime we can determine the edges $e_1, \ldots, e_p$ within time $O(n + m) = O(m)$ by scanning over all edges.

Every connecting component V_i has at least an edge $e_i \in \delta(V_i)$, since the graph is connected. Therefore in the next round the set is going to be unified with (at least) a neighbor set. Therefore the number of nodes of the minimum connected component has increased by at least a factor of two. So, we need at most $\log n = \lceil \log_2 n \rceil$ rounds until there is only one connected component.

For correctness we show that only "safe" edges are added. Without loss of generality (wlog) we can assume that in a step of the while loop the edges $e_1, \ldots, e_p$ are sorted regarding the weight function $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_p)$ (note that some of the edges must be the same, i.e. $e_i = e_{i+1}$ and therefore the equality $c(e_i) = c(e_{i+1})$ holds simply despite c is injective).

Now, e_i is safe for $E_T \cup \{e_{i+1}, \ldots, e_p\}$, since we assume the edges to be sorted. So, $c(e_i) \leq c(e)$ for all $e \in \delta(V_{i+1} \cup \ldots \cup V_p)$, since $c(e_j)$ is minimal for $\delta(V_j)$ and

$$\delta(V_{i+1} \cup \ldots \cup V_p) \subseteq \delta(V_{i+1}) \cup \ldots \cup \delta(V_p) \; .$$

Now we insert the edges in opposite order, i.e. $e_p, e_{p-1}, \ldots, e_1$. □

Chapter 4

Flow Problems

4.1 Flows, Cuts and Permissible Flows

In nature, production flow, and transport systems we very often encounter so-called flows, which we can model by a directed graph. The flow problem is defined as follows.

Definition 30 *Given a finite directed grapht $G = (V, R, \alpha, \omega)$, a weighting function $h : R \to \mathbb{R}$, extended to sets $R' \subseteq R$:*

$$h(R') := \sum_{r \in R'} h(r) \ .$$

Then $f : R \to \mathbb{R}$ is the **flow value** *of an edge*

$$\begin{aligned} f(\delta^+(v)) &= \sum_{r \in \delta^+(v)} f(v) \\ f(\delta^-(v)) &= \sum_{r \in \delta^-(v)} f(v) \end{aligned}$$

Then, we define $\text{excess}_f(v) := f(\delta^-(v)) - f(\delta^+(v))$ *as the* **flow excess** *of v under f.*

Definition 31 (flow) *Given nodes $s, t \in V(G)$, then a (s,t)-***flow** *is a function $f : R \to \mathbb{R}$ with*

$$\text{excess}_f(v) = 0 \quad \forall v \in V \setminus \{s, t\} \ .$$

We call s the source, t the sink and define $\textbf{val}(f) := \text{excess}_f(t)$ *as the* **flow value**.

For a capacity function $c : R \to \mathbb{R}$ we call a flow **permissable** *if $0 \leq f(v) \leq c(r)$ for all $r \in R$.*

Consider a cut (S, T) with $s \in S$, $t \in T$, $S \dot\cup T = V$. We call such a cut an (s,t)-cut.

Definition 32 *The forward edges set of an (s,t)-cut (S, T) is defined as*

$$\delta^+(S) := \{r \in R : \alpha(r) \in S \wedge \omega(r) \in T\}$$

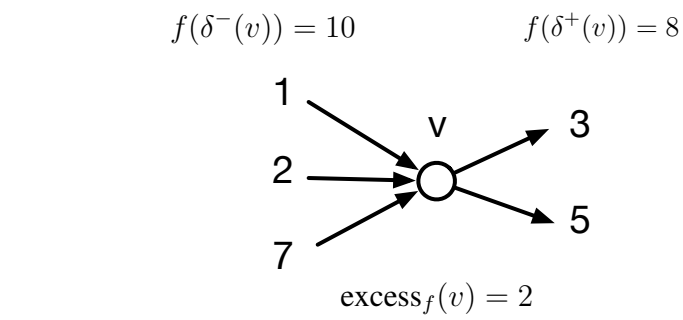


Figure 4.1: The definition of flow and excess

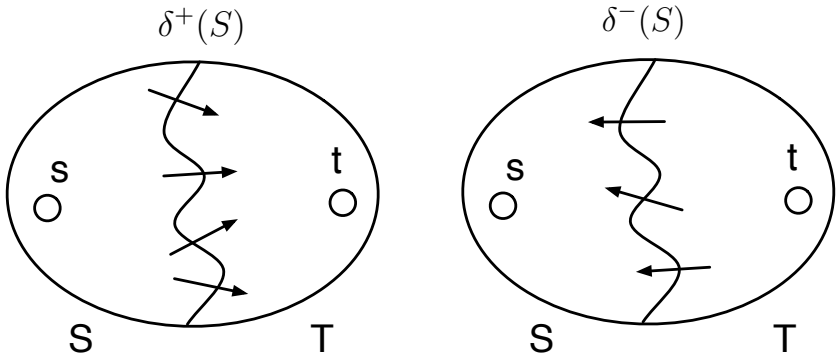


Figure 4.2: Forward $\delta^+(S)$ and backward edges set $\delta^-(S)$ of a cut (S, T)

The backward edge set is defined as

$$\delta^-(S) := \{r \in R : \omega(r) \in S \wedge \alpha(r) \in T\}$$

Define:

$$\begin{aligned} \text{excess}_f(S) &:= f(\delta^-(S)) - f(\delta^+(S)) \\ \text{excess}_f(T) &:= f(\delta^-(T)) - f(\delta^+(T)) \end{aligned}$$

Lemma 7 *For all flows $f : R \to \mathbb{R}$:*

$$\text{excess}_f(S) = \sum_{v \in S} \text{excess}_f(v) \ .$$

Proof:

$$\sum_{v \in S} \text{excess}_f(v) = \sum_{v \in S} \left(\sum_{r \in \delta^-(v)} f(r) - \sum_{r \in \delta^+(v)} f(r) \right)$$

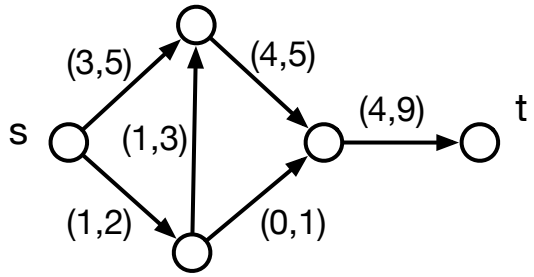


Figure 4.3: A permissible flow, were each edge r is labeled with $(f(r), c(r))$

If both start and terminal node of an edge r are in S, then $f(r)$ occurs twice: positive and negative. Therefore:

$$= \sum_{r \in \delta^-(S)} f(r) - \sum_{r \in \delta^+(S)} f(r) = f(\delta^-(S)) - f(\delta^+(S))$$

□

Analogously, we can prove the following.

Lemma 8 *For all flows $f : R \to \mathbb{R}$:*

$$\text{excess}_f(T) = \sum_{v \in T} \text{excess}_f(v) \ .$$

Lemma 9 *For all (s,t)-cuts (S,T) with and (s,t)-flows f we have*

$$\begin{aligned} \mathit{val}(f) &= f(\delta^+(S)) - f(\delta^-(S)) \\ &= \mathit{excess}_f(t) \\ &= -\mathit{excess}_f(S) \end{aligned}$$

Proof:

$$\begin{aligned} \text{val}(f) &= \text{excess}_f(t) \\ &= \sum_{v \in T} \text{excess}_f(v) && (4.1) \\ &= \text{excess}_f(T) && (4.2) \\ &= f(\delta^-(T)) - f(\delta^+(T)) && (4.3) \\ &= f(\delta^+(S)) - f(\delta^-(S)) && (4.4) \end{aligned}$$

(4.1) follows from $\text{excess}_f(v) = 0$ for all $v \in V \setminus \{s,t\}$.
(4.2) follows by Lemma 7.
(4.3) follows by definition.
(4.4): Note that $\delta^-(T) = \delta^+(S)$, $\delta^-(S) = \delta^+(T)$ □

Definition 33 *The capacity of a (S,T)-cut is defined as*

$$c(\delta^+(S)) := \sum_{r \in \delta^+(S)} c(r) \; .$$

This naming is reasonable as the following lemma shows.

Lemma 10 *If f is a permissible flow then val$(f) \leq c(\delta^+(S))$.*

Proof:

$$\text{val}(f) = \underbrace{f(\delta^+(S))}_{\leq c(\delta^+(S))} - \underbrace{f(\delta^-(S))}_{\geq 0} \leq c(\delta^+(S))$$

□

We will consider lower and upper capacity bounds given by 0 and $c(r)$ such that

$$0 \leq f(r) \leq c(r)$$

Now if val$(f) = c(\delta^+(S))$, then f must be a maximum flow for any (S,T)-cut.

4.2 Residual Graph and Augmenting Paths

Let $G_f = (V, R_f, \alpha', \omega')$ be the **residual graph**, where $R_f \subseteq \{+r_1, +r_2, \ldots, +r_m, -r_1, -r_2, \ldots, -r_m\}$ is defined as follows.

- If $r \in R$, $f(r) < c(r)$, then $+r \in R_f$ and $\alpha'(+r) = \alpha(r)$, $\omega'(+r) = \omega(r)$,
- if $r \in R$, $f(r) > \ell(r)$, then $-r \in R_f$ and $\alpha'(-r) = \omega(r)$, $\omega'(+r) = \alpha(r)$.

An **augmenting flow** is a path P in G_f from s to t using forward edges $+r$ and backward edges $-r$. Note that

1. Positive edges indicate a flow increase.
2. Negative edges indicate a flow reduction.

So, we define for $\sigma_f \in R_f$ the potential capacity $c : R' \to \mathbb{R}$ for augmentation as:

$$\begin{aligned} c_f(+r) &:= c(r) - f(r) \\ c_f(-r) &:= f(r) \end{aligned}$$

Definition 34 *A path from s to t in a residual graph G_f is a* **flow augmenting path**. *Define*

$$\Delta(P) := \min_{\sigma_r \in P} c_f(\sigma_r)$$

Note that if a flow augmenting path for f exists, then f is no maximum flow.

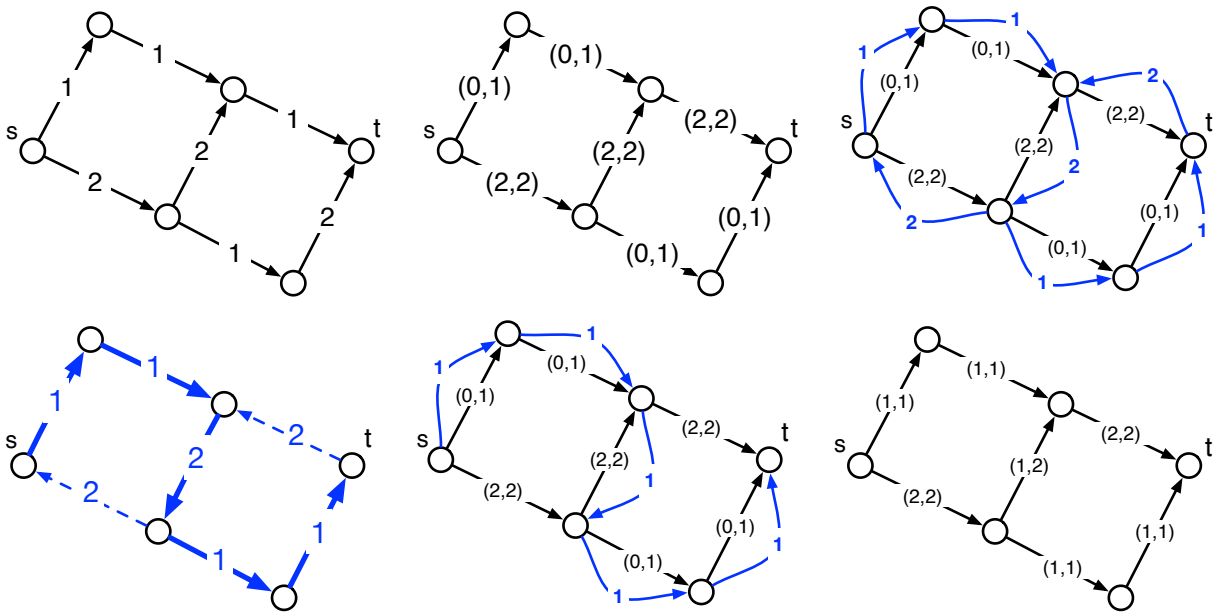


Figure 4.4: A bounded capacity graph; with a non optimal flow; the residual graph; an augmenting path; the optimal flow

Lemma 11 *If f is a permissible flow and f^* is a maximum (s,t)-flow. Then*

$$val(f^*) \leq val(f) + c_f(\delta^+_{G_f}(S))$$

for all (s,t)-cuts (S,T) in G_f.

Proof: Consider $\epsilon \geq 0$ defined by

$$\text{val}(f^*) = \text{val}(f) + \epsilon \leq c(\delta(S)) \ .$$

Then, we have

$$\begin{aligned}
\epsilon &\leq c(\delta^+(S)) - f(\delta^+(S)) + f(\delta^-(S)) \\
&= \sum_{r \in \delta^+(S)} (c(r) - f(r)) + \sum_{r \in \delta^-(S)} f(r) \\
&= \sum_{+r \in \delta^+(S)} c_f(+r) + \sum_{-r \in \delta^+(S)} c(-r) \\
&= c_f(\delta^+_{G_f}(S))
\end{aligned}$$

□

4.3 The Max-Flow-Min-Cut-Theorem

Theorem 21 *In a directed graph G with capacity $c : R \to R^+$ we have*

$$\max_{f \text{ is a permissible flow}} val(f) = \min_{(S,T) \text{ is } (s,t)\text{-cut}} c(\delta^+(S)) \ :$$

Proof: Let f^* be a maximum flow. Then there exists no flow augmenting path for f^*, which implies that t is not reachable in G_{f^*} starting from s.

Define

$$\begin{aligned} S &:= \{v \in V \ : \ v \text{ is reachable from } s \text{ in } G_{f*}\} \\ T &:= \{v \in V \ : \ v \text{ is not reachable from } s \text{ in } G_{f*}\} \end{aligned}$$

Then, $s \in S$ and $t \ \in T$.

Consider an edge $r \in \delta^+(S)$. This implies that $f^*(r) = c(r)$, since otherwise there is a path in G_{f^*} from s to t. Then, $f^*(\delta^*(S)) = c(\delta^+(S))$.

Now consider an edge $r \in \delta^-(S)$. Then, $f^*(r) = 0$, since otherwise there is a path from s to t. Hence, $f^*(\delta^-(s)) = 0$.

This implies

$$c(\delta^+(S)) = f^+(\delta^+(S)) - \underbrace{f^*(\delta^-(S))}_{0} = \text{val}(f^*) \ .$$

The last equation follows by Lemma 9. □

Theorem 22 (Integer Max Flow) *A graph $G = (V, R, \alpha, \omega)$ with integer capacities $c : R \to \mathbb{N}_0$ has a maximum flow f such that for all $r \in R$: $f(r) \in \mathbb{N}_0$.*

Proof: First note, that the minimum cut (S, T) is given as an integer $c(S)$. Then, start with an empty flow $f_0(r) = 0$ for all $r \in R$ an construct the residual graph G_{f_0}. If there exists a path from s to t in G_{f_0} with non-zero weight, then it has an integer capacity. Using this path we construct the flow f_1, and so on.

Hence, after at most $c(S)$ this iteration will terminate yielding an integer max flow. □

Chapter 5

Matching and Vertex Cover

5.1 The Marriage Theorem

We need the following notations for undirected graphs.

Definition 35 *An undirected simple graph is* **bipartite***, if there exists a node partition $V = A \cup B$ with $A \cap B = \emptyset$, such that every edge is incident with a node from A and B.*

Definition 36 *A* **matching** *is a subset of the edges of a given undirected simple graph G such that no two edges are incident. The* **maximum maching number** *$\nu(G)$ is defined as*

$$\nu(G) := \max\{|M| \ : \ M \text{ is a matching for } G\} \ .$$

A matching M is **perfect** *if every nodes $v \in V(G)$ is incident with an edge of the matching M.*

Consider a bipartite graph G with node set $V = M \cup W$, where $|M| = |W|$, $M \cap W = \emptyset$ and where edges $E \subseteq M \times W$ denote the sympathies between male (gentleman) and female (women) persons. The question arises, when there is a perfect matching, i.e. if all male and female persons can marry without performing bigamy.

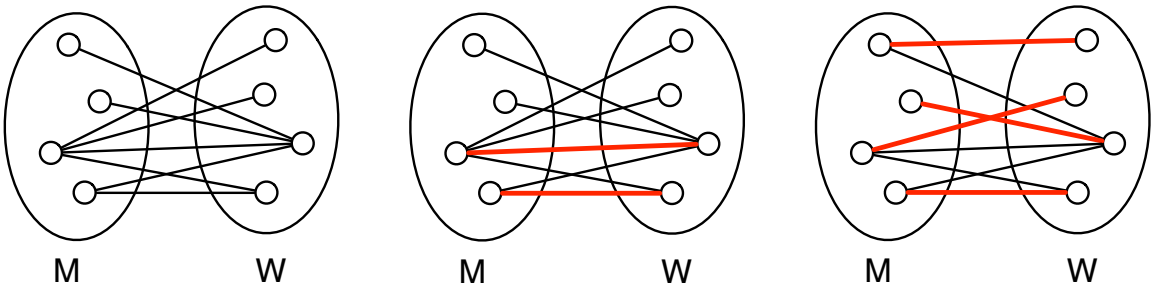


Figure 5.1: A bipartite graph, a matching, and a perfect matching

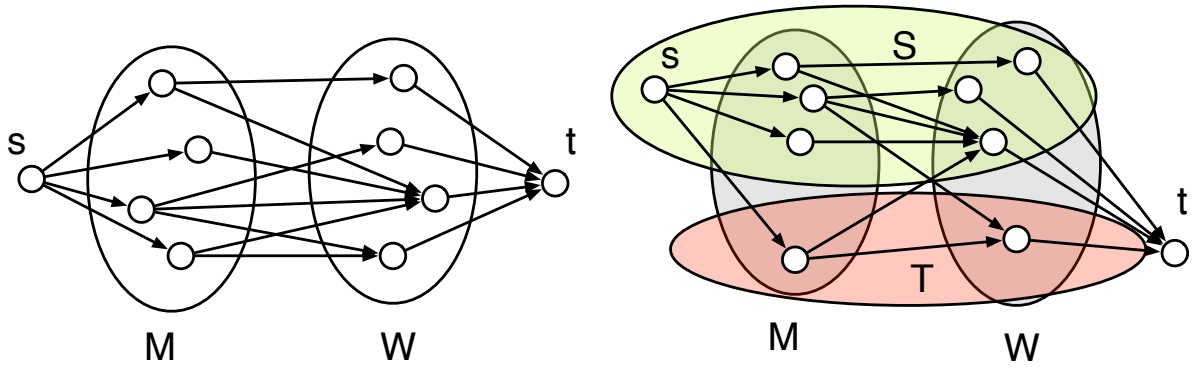


Figure 5.2: The corresponding flow problem and the (s,t)-cut (S,T)

Theorem 23 *There is a perfect matching for the sympathy graph M, if for every subset $W' \subseteq W$ of women at least $|W'|$ acceptable men are available, i.e.*

$$|N^-(W')| \geq |W'| \quad \textit{for all } W' \subseteq W \ , \tag{5.1}$$

where $N^-(W') := \{h \in H : [h,w] \in E$, for a $w \in W'\}$.

Proof: Condition (5.1) is necessary, since otherwise by the pigeonhole principle, there is no matching for a set of women W' with $|N^-(W')| < |W'|$ of women.

We now show that such a matching must exist. For this we transform this problem to a flow problem. So, we add two nodes s,t to the graph with disjoint node sets M,W. We now define the following flow problem for the graph $G' = (V', R, \alpha, \omega)$, where

$$\begin{aligned} V' &= M \cup W \cup \{s,t\} \\ R &= \{s\} \times M \ \cup \ E' \ \cup \ W \times \{t\} \end{aligned}$$

where E' are the directed edges of from M to W corresponding to the undirected edges of $E(G)$. The capacity for all edges $c(r) = 1$, for all $r \in R$.

If there is a flow with value $|M| = |W|$ from s to t, this implies that there is an integer flow with the same value. Consider the graph F consisting of all edges with flow 1. Every node $m \in M$ has exactly on incoming and one outgoing edge, in F. The some holds for every node $w \in W$. Therefore, there the edges $E(F) \cap E'$ describe a perfect matching.

Now consider a (s,t)-cut (S,T) and define

$$\begin{aligned} M_S &:= S \cap M \\ M_T &:= T \cap M \\ W_S &:= S \cap W \\ W_T &:= T \cap W \end{aligned}$$

Then, the capacity of this cut can be compute by all edges connecting

1. s and M_T (with capacity $|M_T|$)

2. M_S and W_T (capacity $|N^-(W_T) \cap M_S|$)

3. W_S and t (with capacity $|W_S|$).

So, we can lower bound the capacity as follows

$$\begin{aligned} c(\delta^+(S)) &= \sum_{r \in \delta^+(S)} c(r) \\ &= |M_T| \quad + \quad |N^-(W_T) \cap M_S| + |W_S| \\ &\geq |N^-(W_T) \cap M_T| + |N^-(W_T) \cap M_S| + |W_S| \\ &= |N^-(W_T)| + |W_S| \\ &\geq |W_T| + |W_S| \\ &= |W| = |M| \end{aligned}$$

So, every cut has at least a capacity of $|W|$ and therefore a maximum matching exists. □

5.2 Vertex Cover

Definition 37 *A subset $S \subseteq V$ is called a* **vertex cover** *if S incident with every edge. The minimum vertex cover is defined as*

$$\tau(G) := \min\{|S| \ : \ S \textit{ is a vertex cover}\} \ .$$

Theorem 24 *Let G be a graph. For all matchings M and all vertex covers S of G we have $|M| \leq |S|$. This implies*

$$\nu(G) \leq \tau(G) \ .$$

Proof: Every edge e of the matching is incident with a vertex in S. Since M is a matching, all vertices of S are different. Therefore, $|M| \ \leq |S|$. □

Note that the cardinality of the matching can be larger than those of the vertex cover.

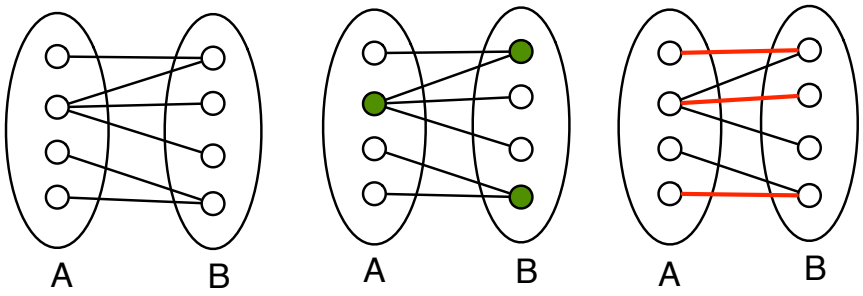


Figure 5.3: A bipartite graph, a minimum vertex cover $\tau(G)$, and a maximum matching $\nu(G)$.

Algorithm 13: Approximation algorithm for computing the smallest vertex cover S

Input: undirected graph G ;
Output: vertex cover S ;
$M := \emptyset$;
$S := \emptyset$;
for $v \in V$ **do**
  for $e = \{v, u\} \in \delta(v)$ **do**
    if $M \cup \{e\}$ *is a matching* **then**
      $M := M \cup \{e\}$;
      $S := S \cup \{u, v\}$;
    end
  end
end
return S

Theorem 25 *Algorithm 13 computes a vertex cover of size $2\tau(G)$ in linear runtime.*

Proof: Note that the the inner loop can be computed in constant time, if the graph is given as incidence matrix. M is maximal with respect to inclusion. Every edge $e \notin M$ is incident with at least edge from M. Therefore all vertices of S of the matchings from a valid vertex cover. Therefore $|S| \leq 2|M| \leq 2\tau(G)$. □

For bipartite graphs the minimum vertex cover and the maximum matching have the same size.

Theorem 26 (König 1931) *For every bipartite graph we have*

$$\tau(G) = \nu(G) \ .$$

Proof: Let $G = (V, E)$ a bipartite graph with node sets $V = A \cup B$, $A \cap B = \emptyset$. Again we construct a new graph with additional nodes s, t and directed edge set R containing (s, a) for all $a \in A$, (a, b) for all $[a, b] \in E$ and for all $b \in B$, (b, t).

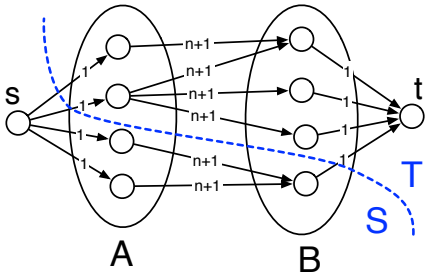


Figure 5.4: The flow problem used in the Theorem of König and a minimum cut.

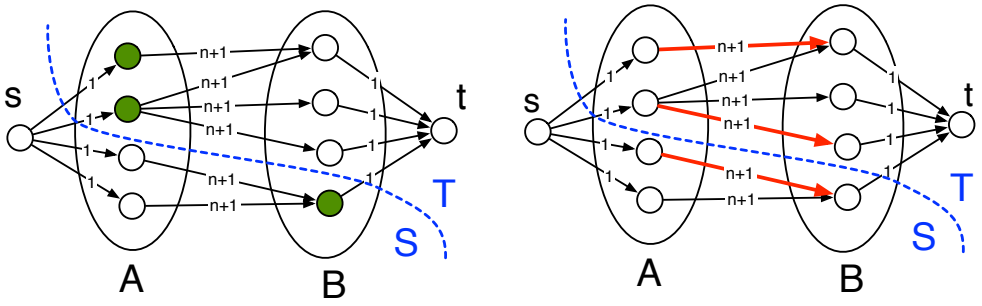


Figure 5.5: The minimum cut describes the minimum vertex cover and the maximum flow a maximum matching..

Let $n = |A|$ and wlog. $n = |A| \leq |B|$. For a flow problem the edge capacity of edges $c(s, a) = c(b, t) = 1$ and $c(r) = n + 1$ for $r \in A \times B \cap R$. The edges with positive weight of an integer flow from s to t corresponds to a matching in G.

Let (S, T) be the minimum (s, t)-cut in G'. Since the capacity $c(\delta^+(s)) = n$, in the edge set between S and T there is no edge in R, since each such edge has capacity $M > 1$. Hence, $C := (A \cap T) \cup (B \cap S)$ is a vertex cover in G, which size $|C|$ is equivalent to a matching and therefore $\tau(G) = \nu(G)$. □

Chapter 6

Coloring and Chordal Graphs

6.1 Coloring, Independence, Clique Partitionings

The Coloring problem is the classic graph theory problem, where the nodes of an undirected graph have to be colored such that no neighbored nodes receive the same color.

Definition 38 *For $k \geq 0$ a k-***Coloring** *of a graph is a surjective mapping $f : V \to \{1, \ldots, k\}$ such that if $u, v \in V$ are adjacent, then $f(v) \neq f(v')$.*

For $i \in \{1, \ldots, k\}$ let $f^{-1}(i) \subseteq V$ denote the **color class** i.

An example for the graph coloring is the choice of different radio frequencies of radio stations, such that neighbored radio stations do not interfere. Clearly, it is interesting to choose as few colors as possible. Therefore we define the **chromatic number** $\chi(G)$ as follows.

$$\chi(G) := \min\{k : \text{there exists a } k\text{-coloring for } G\} .$$

Definition 39 *A set of nodes $C \subseteq V$ is called* **clique***, if all nodes $u, v \in C, u \neq v$ are adjacent.*

The maximum size of a clique is called the **clique number** $\omega(G)$

$$\omega(G) := \max\{|C| \ : \ C \text{ is a clique in } G\} .$$

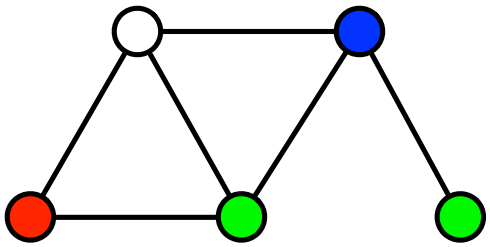


Figure 6.1: A (non minimum) vertex coloring of a graph with clique number 3.

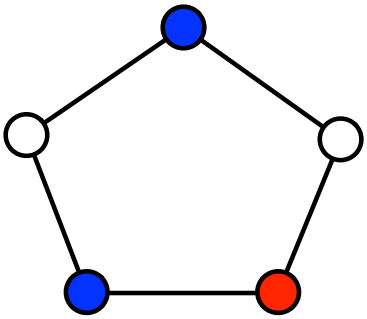


Figure 6.2: A graph with $\omega(G) = 2$ and $\chi(G) = 3$

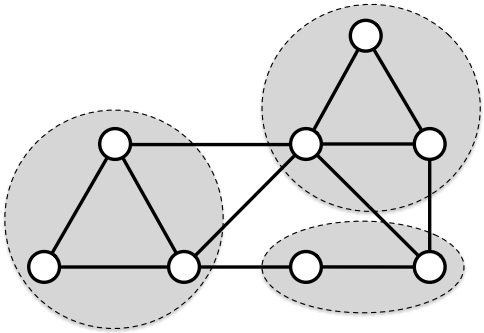


Figure 6.3: A graph clique partition number $\overline{\chi}(G) = 3$

Every clique of size c can be colored with c colors, but not with any smaller number of colors. So the following observation can be made.

$$\chi(G) \geq \omega(G) \; .$$

However, the equality is not true in general, which can be seen in the following graph, see Fig. 6.2.

Note that the node set of any graph can be partitioned in disjoint cliques. We are interested in the smallest number of such cliques.

Definition 40 Clique partition number

$$\overline{\chi}(G) := \min\{k \;\; : \;\; V(G) = C_1 \dot{\cup} C_2 \dot{\cup} \cdots \dot{\cup} C_k \; , \textit{where } C_1, \ldots, C_k \textit{ are cliques in } G\}$$

If $\overline{\chi}(G) = 1$, then we have a graph, which forms a clique. For $\overline{\chi}(G) = n$, then the graph g does not contain any edge.

Definition 41 *A set $U \subseteq V$ is called an* **independent set** *if for all $u, v \in U$, $[u, v] \notin U$.*

The **independence number** $\alpha(G)$ of a graph is defined as

$$\alpha(G) := \max\{|U| : U \text{ is independent set in } V\} \; .$$

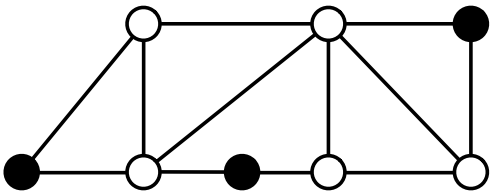


Figure 6.4: The black nodes are independent

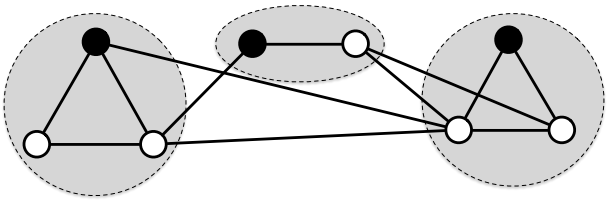


Figure 6.5: From each clique of a clique partition at most one node can occur in the independent set.

Now we observe that

$$\alpha(G) \leq \overline{\chi}(G) \; .$$

For this we consider a maximum set of cliques $C_1, \ldots, C_k$ partitioning G. Every independent set U can have at most one element from each clique. So the overall number of independent nodes is at most k.

Definition 42 *The* **complement graph** $\overline{G} = (V, \overline{E})$ *of a graph* $G = (V, E)$ *is defined as by the edge set*

$$\overline{E} := \{[u, v] : u, v \in V, u \neq v, [u, v] \notin E\}$$

For the complement graph we observe that

1. $|E| + |\overline{E}| = \binom{n}{2} = \frac{n(n-1)}{2}$.
2. $\overline{\overline{G}} = G$
3. $S \in V$ is a clique in G if S is an independent set in $\overline{G}$
4. $\omega(G) = \alpha(\overline{G})$ and $\omega(\overline{G}) = \alpha(G)$

We now relate the independence number to the chromatic number.

Lemma 12

$$\chi(G) \geq \frac{n}{\alpha(G)}$$

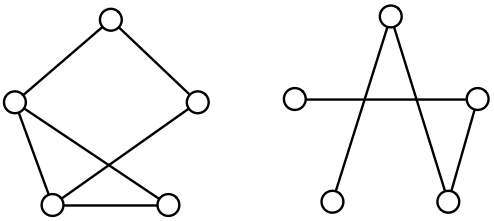


Figure 6.6: A graph G and its complement graph $\overline{G}$

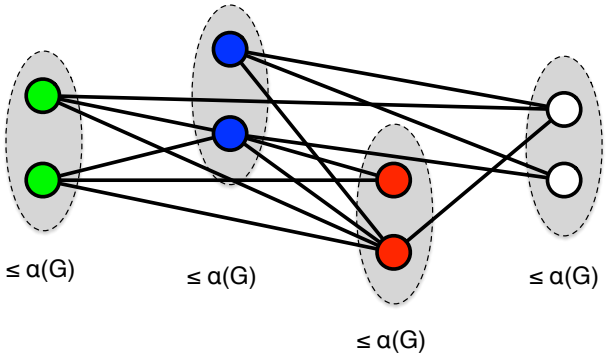


Figure 6.7: Nodes with the same color form an independent set.

Proof: For $k = \chi(G)$ and the maximum coloring f consider the sets $f^{-1}(1), f^{-1}(2), \ldots f^{-1}(k)$.

Each of these sets are independent. Therefore $|f^{-1}(i)| \leq \alpha(G)$. Let $\alpha_i := |f^{-1}(i)|$, then $\sum_{i=1}^{k} \alpha_i = n$ and $\alpha_i \leq \alpha(G)$. So,

$$n = \sum_{i=1}^{\chi(G)} \alpha_i \leq \sum_{i=1}^{\chi(G)} \alpha(G) = \alpha(G)\chi(G)$$

□

Now for the minimum number of edges, there has to be at least one edge between two nodes of $f^{-1}(i)$ and $f^{-1}(j)$ for $i \neq j$ if f is a minimum coloring of G. So, for the minimum number of edges of a graph with m colors we get for $k = \chi(G)$.

$$m \geq \frac{k(k-1)}{2}$$

When we solve this equation for k we get for the chromatic number in a graph with m edges:

$$\chi(G) = k \leq \frac{1}{2} + \sqrt{2m + \frac{1}{4}}\,.$$

For clique graphs this inequality becomes an equation.

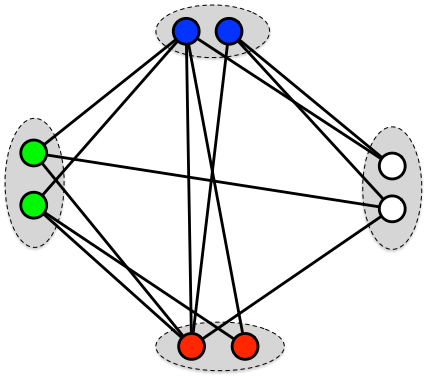


Figure 6.8: In a minimum coloring there is at least an edge between the node sets with the same color.

6.2 Perfect Graphs

For perfect graphs the clique number is not only equal to the chromatic number, it is the case for all induced subgraphs.

Definition 43 *An undirected simple graph $G = (V, E)$ is* **perfect***, if for all induced sub-graphs H*

$$\omega(H) = \chi(H) \; .$$

As an example take the square graph for a perfect graph.

Theorem 27 (Lovasz) *A graph G is perfect, iff $\overline{G}$ is perfect.*

This theorem is not proved in this lecture and can be found in [Lov72].

Definition 44 *A* **hole** *is cycle C_{2k+1} for $k \geq 2$, such that no other edges than the cycle edges exist.*

An **anti-hole** *is the complement graph of a hole, i.e. a node set V which induces a hole in the complement graph of G.*

Holes and anti-holes are not perfect. So, the lack of holes and anti-holes are a necessary feature of holes. It was long conjectured that the non-existence of holes and anti-holes in graphs implies the perfectness of a graph.

This could be proved in 2002 [CRST02] resulting in the following theorem.

Theorem 28 (Strong Perfect Graph Theorem) *A graph is perfect, if and only if every induced subgraph is neither a hole nor an anti-hole.*

Due to its complexity this proof is omitted.

In perfect graphs $\alpha(G), \omega(G)\chi(G), \overline{\chi(G)}$ can be computed in polynomial time [GLS84].

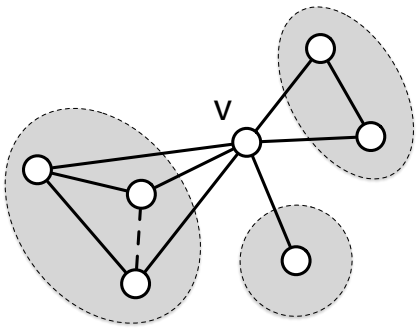


Figure 6.9: If $n > 1$, the chordal graph G is connected and there exists a node $u \in V$ such that every node is in the neighborhood of u, then all neighbors of u are simplicial nodes.

6.3 Chordal Graphs

Definition 45 *An undirected graph is* **chordal***, if every elementary cycle of size at least* 4 *contains at least a chord.*

This means for $k \geq 4$ that for $C = (v_0, e_1, v_1, e_2, \ldots, d_k, v_k = v_0) \in G$ then there exists $[v_i, v_j] \in E$ for $j \notin \{i-1, i-i+1\}$. So, every graph is triangulized, i.e. consists of triangle.

Note the following observation: If G is chordal, then every induced sub-graph is chordal.

Definition 46 *A node* $v \in V(G)$ *is* **simplicial***, if its neighborhood* $N_G(v)$ *is a clique.*

The relationship between simplicial nodes and chordal graphs is given by the Theorem of Dirac.

Theorem 29 (Dirac) *Every chordal graph contains a simplicial node.*

Proof: We use an induction over $n = |V|$ and consider the following cases.

1. G is complete. Then, every node is simplicial.
2. $n = 1$. Then the $N_G(v) = \emptyset$ is simplicial.
3. $n > 1$ and G is not connected. Then the claim follows by induction, since each connected component hast less than n nodes.
4. $n > 1$ and there exists $u \in V$ such that $G - v$ is not connected and there exists a connected component C in $G - v$ where all nodes are in the neighborhood of v. Then all nodes of this connected component are adjacent. Assume otherwise, then there are two nodes $u, u' \in C$. Consider the shortest path in C from u to u' and join it with the edges $[u, v], [v, u']$. This elementary cycle does not contain a chord. By this contradiction, it follows that all nodes in C are connected, and therefore each node of C is a simplicial node.

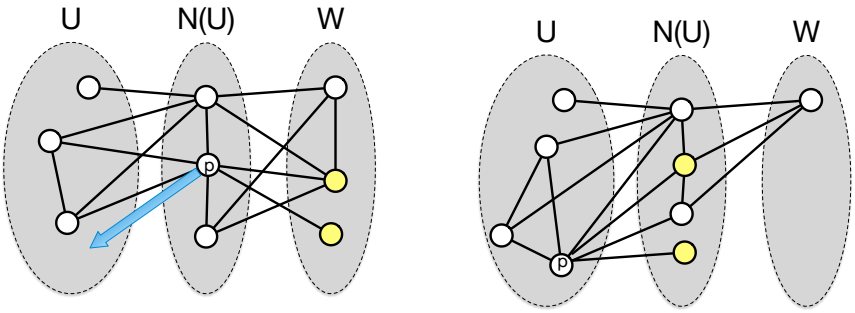


Figure 6.10: The node p can be moved from $N(U)$ to U, if there exists a node in W which is not adjacent to p

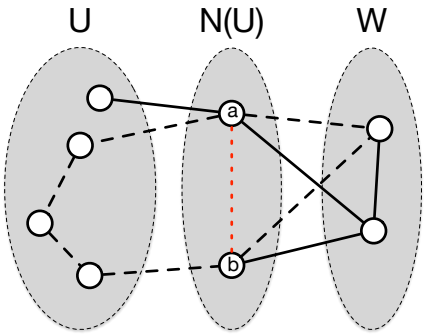


Figure 6.11: Nodes a and b are adjacent, since otherwise the chordal graph property would have been violated.

5. $n > 1$ and G is connected, there is no node u such that all nodes are adjacent to u.

   Consider a connected set U such that $U \subseteq V$ and $U \cup N(U) \neq V$. Such a set must exist by the assumption.

   Choose U to be the maximum such set and let $u, v \in V$ such that $[u, v] \notin E$.

   Consider U, $N(U)$ and $W = V \setminus (U \cup N(U))$. Now, every node of W is a simplicial node. Every node $p \in N(U)$ is adjacent to all nodes in W. Otherwise p could be added to U, while W remains non-empty. Then, U would have not been maximal, see Fig. 6.10.

   Assume $a, b \in N(U)$ are not adjacent. Then consider the shortest path from a to b through U. Note that this path combined with the edge $[a, w]$, $[b, w]$ with some $w \in W$ forms a cycle of length at least four. Since, G is a chordal graph a chord must exist. Since P is the shortest path and w and all nodes in U are disconnected, the only possible chord is $[a, b]$, which is a contradiction, see Fig. 6.11.

   Therefore all nodes in $N(U)$ are connected and $N(U)$ forms a clique and therefore all nodes in W are simplicial.

□

Corollary 5 *Every chordal graph is perfect.*

Proof: We only have to prove that $\omega(G) = \chi(G)$, since every sub-graph of G is also chordal.

For this we use an induction over n. Now, for $n = 1$ we have $\omega(G) = \chi(G) = 1$.

For $n > 1$ we know from Theorem 29, that G contains a simplicial node v. So, the graph induced by $\{v\} \cup N(v)$ is a clique, which implies $|N(v)| \leq \omega(G) - 1$.

From the induction hypothesis we have $\chi(G - v) = \omega(G - v) \leq \omega(G)$. So, we can color $N(v)$ with $\omega(G)$ colors. Then, the neighbors of v have at most $\omega(G) - 1$ colors, which allows to color v with the remaining color. This results in a $\omega(v)$-coloring and therefore

$$\chi(G) \leq \omega(G) \leq \chi(G) \ ,$$

which implies the claim. □

Definition 47 *A* **perfect elimination scheme** *of an undirected graph G is a bijection $\sigma : V \rightarrow \{1, \ldots, n\}$ in G, such that $\sigma^{-1}(i) = v_i$ is a simplicial node for $G[\sigma^{-1}(i), \ldots, \sigma^{-1}(n)]$.*

In fact, the existence of perfect elimination schemes and the chordal graph property are equivalent.

Theorem 30 *A graph is chordal if and only if it possesses a perfect elimination scheme.*

Proof: "⇒": From Theorem 29 we know that every chordal graph has a simplicial node $\sigma^{-1}(1) = v_1$. By induction the chordal graph $G - \{v_1\}$ has a perfect elimination scheme $(v_2 = \sigma^{-1}(2), \ldots, v_n = \sigma^{-1}(n)\}$.

"⇐": Consider an elementary cycle of length $k \geq 4$ and let v_i be the node with the smallest number in the elimination scheme. Then, its neighbors are adjacent, since v_i is a simplicial node for the rest of the graph. So, every cycle of length $k \geq 4$ has a chord. □

Perfect elimination schemes can be found in polynomial time. For this, one can test each node whether it is a simplicial node. Testing the neighborhood for the clique property takes at most $O(g^2)$ steps, if g is the degree of the graph. This has to repeated n times in order to find a simplicial node. From the proof of the theorem above, it is clear that a perfect elimination scheme can start from any simplicial node. So, iteratively removing a simplicial node results in an overall $O(n^2 g^2)$-time bounded algorithm. Since $g \leq n - 1$ we have a run-time of at most $O(n^4)$.

A perfect elimination scheme allows to determine the maximum clique and the coloring of a graph in linear time. For this, an algorithm colors the node in this order: $\sigma^{-1}(n), \sigma^{-1}(n - 1), \ldots, \sigma^{-1}(1)$, where each node receives the smallest possible color. Since $v_i = \sigma^{-1}(i)$ has at most $\omega(G) - 1$ neighbors in $\{v_{i+1}, \ldots, v_n\}$ (since $\{v_i\} \cup (N(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ is a clique in G), there is always one of the $\chi(G) = \omega(G)$ colors left to color v_i).

Bibliography

[AH+77] Kenneth Appel, Wolfgang Haken, et al. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

[AHK+77] Kenneth Appel, Wolfgang Haken, John Koch, et al. Every planar map is four colorable. part ii: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 1977.

[CRST02] Maria Chudnovsky, Neil Robertson, PD Seymour, and Robin Thomas. Progress on perfect graphs. 2002.

[Die10] R. Diestel. *Graph Theory*. Graduate Texts in Mathematics. Springer Berlin Heidelberg, 2010.

[GLS84] Martin Grötschel, Laszlo Lovász, and Alexander Schrijver. Polynomial algorithms for perfect graphs. *North-Holland mathematics studies*, 88:325–356, 1984.

[KN09] S.O. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. Leitfäden der Informatik. Vieweg + Teubner, 2009.

[Lov72] L. Lovasz. Normal hypergraphs and the perfect graph conjecture. *Discrete Mathematics*, 2(3):253 – 267, 1972.