University of Szeged, Bolyai Institute

Graph theory

for MSc students in Computer Science

Lecture notes

By
Béla Csaba, Péter Hajnal, Gábor V. Nagy

December 31, 2019

SZÉCHENYI 2020
HUNGARIAN GOVERNMENT
European Union
European Social Fund
INVESTING IN YOUR FUTURE

Reviewers: Norbert Bogya and Fülöp Vanda

This teaching material has been made at the University of Szeged, and supported by the European Union. Project identity number: EFOP-3.4.3-16-2016-00014.

Contents

1 Basics **11**
- 1.1 Graphs and multigraphs . . . 11
- 1.2 Degrees and the handshake lemma . . . 13
- 1.3 Subgraphs . . . 15
- 1.4 Walks, tours, paths, cycles . . . 15
- 1.5 Connectivity and trees . . . 16

2 Graph realizations **20**
- 2.1 Realization by multigraphs . . . 20
- 2.2 Realization by graphs . . . 21
- 2.3 Realization by trees . . . 25

3 Enumeration of spanning trees **26**
- 3.1 Spanning trees of complete graphs . . . 26
- 3.2 Spanning trees of arbitrary graphs . . . 31

4 Network flow problems **33**
- 4.1 Network Flow Problems . . . 33
  - 4.1.1 An algorithm for finding a maximum flow . . . 36
- 4.2 Applications . . . 40
  - 4.2.1 The Menger theorems . . . 40
  - 4.2.2 The Project Selection problem . . . 42
  - 4.2.3 The Image Segmentation problem . . . 43
  - 4.2.4 Finding a maximum matching in bipartite graphs . . . 44

5 Algorithms **45**
- 5.1 Graph searching . . . 45
  - 5.1.1 Breadth-first search . . . 45
  - 5.1.2 Depth-first search . . . 46
  - 5.1.3 Applications of graph search algorithms . . . 47
  - 5.1.4 Finding shortest path from a single source in a weighted graph 48
- 5.2 The minimum spanning tree problem . . . 49

6 Matchings **52**
- 6.1 Definitions . . . 52
- 6.2 Matchings in bipartite graphs . . . 53
- 6.3 Matchings in general graphs . . . 59

6.4 Figures 67

7 Colorings **72**
7.1 Coloring the vertices of graphs 72
7.2 Coloring the edges of a graph 75

8 Planar drawings **79**
8.1 Planar multigraphs 79
8.2 Dual graph 80
8.3 Kuratowski's theorem 82
8.4 Four color theorem 84

9 Walks, tours **85**
9.1 Eulerian tours 85
9.2 Chinese postman 88

10 Paths, cycles **91**
10.1 Hamiltonian paths, Hamiltonian cycles 91
10.2 Traveling salesman problem 93

11 Extremal graph theory **97**
11.1 Independent sets and cliques 97
11.2 Turán's theorem 99
11.3 Ramsey theory 100

Description of the subject

Title of the subject: Graph theory	Credits: 5
Type of the subject: compulsory	
The ratio of the theoretical and practical character of the subject: 60-40 (credit%)	
The type of the course: lecture and practice The total number of the contact hours: 4 per week Language: English	
Type of the evaluation: 2 written tests, oral examination	
The term of the course: I. semester	
Prerequisite of the subject: None	

The aim of the subject: The courses are directed to MSc students in Computer Science with a background in combinatorics and introductory algorithm theory. It provides an overview of techniques to deal with advanced graph theoretical notions, problems. The course uses algorithmic methods. Many graph theoretical problems are formalized as optimization problem. We discuss how to find exact, or approximate solutions. We use natural ideas to design complicated algorithms, and prove basic theoretical graph theory results. Participants will study algebra, enumeration, combinatorics, combinatorial optimization to be able to handle complex graph theory problems. Some of the complexity theoretical aspects of the graph problems will be touched on.

Course description:

1. Basics
2. Graph realizations
3. Enumeration of spanning trees
4. Network flow problems
5. Algorithms
6. Matchings
7. Colorings
8. Planar drawings
9. Walks, tours
10. Paths, cycles
11. Extremal graph theory

Selected bibliography:

- L. Lovász, Combinatorial problems and exercises, AMS Chelsea Publishing, American Mathematical Society; 2nd edition (2007), ISBN 978-0821842621
- R. Diestel, Graph theory, Springer-Verlag, Heidelberg, Graduate Texts in Mathematics, Volume 173, 5th edition (2017), ISBN 978-3-662-53621-6
- B. Bollobás, Modern graph theory, Springer-Verlag, Heidelberg, Graduate Texts in Mathematics, Volume 184, Corrected 2nd printing (2002)
- D.B. West, Introduction to Graph Theory, Prentice Hall, Second edition (2001), ISBN 978-0130144003
- J.A. Bondy, U.S.R. Murty, Graph theory with applications, Elsevier, 5th printing (1982), ISBN 978-0444194510

General competence promoted by the subject:

a) Knowledge

- Understand the differences between formal and informal discussions.
- Familiar with the tools, terminology, and methods of graph theory.
- Know the computational methods of graph theory.
- Understand new or difficult concepts, algorithms. Appreciate their full mathematical precision and outcome.

b) Skills

- Able to interpret and the present the results.
- Able to apply the tools and techniques of graph theory.
- Able to recognize the coherency among the different areas of graph theory.
- Able to participate in graph theoretical projects under competent supervision.

c) Attitude

- Open to cooperate with her/his classmates.
- Ready to understand the concept graph theoretical optimization and algorithms.
- Interested in new results, techniques and methods.
- Aspires to use the abstract terminology and algorithmic methods.

d) Autonomy and responsibility

- Able to solve complex problems independently.
- Provides and requires clear explanation.
- Helps her/his classmates in the completion of their projects.
- Able to create graph theoretical models consciously.
- Able to do autonomous application of theoretical results.

Special competence promoted by the subject			
Knowledge	Skills	Attitude	Autonomy and responsibility
Know the basic notions of graph theory. Aware of the distinction between "graph" and "multigraph".	Able to determine simple parameters of small given graphs.	Willing to understand the connections between real life structures and their graph theoretical models.	
Know the definition of the degree of a vertex in a multigraph.	Able to determine the degree sequence of a small concrete graph.		Independently able to draw a realizing graph for a given sequence.
Acquire the basic knowledge about trees.	By computing a determinant able to determine the number of spanning trees of small graphs.	Understand the difference among enumeration and listing.	
Know the definition of directed graphs.	Able to determine maximal flow and minimum cut in a given network.		Independently able to model real life problems as a flow problem.
Know the elements of algorithm theory.	Able to execute simple graph theoretical algorithms.	Ready to explain what conditions are necessary to apply the basic algorithms.	
Recall the definition of an independent edge set and the corresponding optimization problem.	Able to calculate the matching parameter of a given graph.	Aspire to formulate practical problems as matching problems, and to solve it.	Discover the greedy algorithm independently.
Know the clique parameter.	Able to use basic coloring algorithms.	Open to study new techniques and methods.	Can independently determine the chromatic number of small graphs.

Knowledge	Skills	Attitude	Autonomy and responsibility
Familiarity with intuitive topology of the plane.	Able to find obstructions to planarity.	Open to draw graphs and try to improve the initial drawing.	Can independently give arguments that proves that certain graph is not planar.
Acquire the basic knowledge of the notion of a walk and a tour in a graph.	Able to determine an Euler tour in a given graph.	Open to connect the theoretical Euler theorem to practical problems.	Can independently argue that the described algorithm provides the correct output.
Remember the notion of a path and a cycle in a graph.	Able to interpret the parameter of an approximation algorithm.	Understand the difference between the necessary and sufficient conditions.	Can independently describe how one can augment a non-Hamiltonian path in a graph.
Recall the definition of a triangle and a subgraph in a graph.	Able to bound the extremal parameter in the case of small forbidden subgraphs.	Open to study new techniques and methods.	Can independently give upper and lower bounds for some Turán numbers.
Instructor of the course: Béla Csaba, PhD, associate professor			
Teachers: Gábor V. Nagy, PhD, senior lecturer; Béla Csaba, PhD, associate professor			

This lecture note was written, based on the experience and material of the previous years' Graph theory courses for MSc students in Computer Science. There are 11 numbered sections according to the weekly schedule of the course. In each section we describe basic graph theory notions and some problems related to the new notions. Through examples and simple ideas we exhibit the main steps that lead to the solution of the proposed problems. We emphasize the algorithmic way of thinking. Very simple ideas — like greedy algorithms, augmentation methods — are used to pave a natural path to complicated algorithms. All algorithms are highlighted. If possible, simple examples help the students to understand how the algorithms work. They will be able to execute them by themselves. We are sure that if students attend the classes and use this note, then they will be familiar with advanced graph theoretical notions, able to recognize several basic design disciplines

for algorithms, and can execute basic graph theoretical algorithms. With their improved mathematical skills, they will be able to attack practical problems, design graph theoretical models, and suggest solution to them.

The Authors

Chapter 1

Basics

This chapter collects the basic notions and theorems of graph theory that are required to read this book. Most of them were covered in former studies in more detail.

1.1 Graphs and multigraphs

Multigraphs are basic structures in mathematics. They can model road systems, social networks, molecules, and so on. Informally, a multigraph consists of "nodes" (that we call vertices) and "curves" (that we call edges) such that each curve connects two (not necessarily distinct) nodes. See Figure 1.1. This picture is good to keep in mind, but in fact it is just a visualization of an abstract structure, defined as follows.

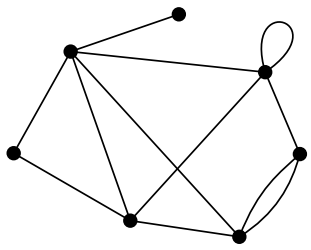

Figure 1.1: A multigraph with 7 vertices and 12 edges

Definition. A *multigraph* G is an ordered triple (V, E, ψ), where V is finite set, E is finite set, and ψ is an $E \to \mathcal{P}_2(V)$ function, where $\mathcal{P}_2(V)$ is the set of one- or two-element subsets of V, that is, $\mathcal{P}_2(V) = \{\{u, v\} : u, v \in V\}$.

The set V is called the *vertex set* of G, and the set E is called the *edge set* of G. We may write $V(G)$ and $E(G)$ for these sets if we want to indicate G in the notation. (We also write $\psi(G)$ when necessary.) The elements of V are called the *vertices* (or *nodes*) of G, the elements of E are called the *edges* of G. The number of vertices of G is commonly denoted by $v(G)$, the number of edges of G is commonly denoted by $e(G)$.

As it was foreseen in the first paragraph, this abstract structure translates to a visualization of the multigraph: The function ψ describes the incidences between

edges and vertices, namely, the *endpoints* of an edge e are exactly the elements of the set $\psi(e)$. Here we used the terminology of the visual picture about multigraphs, and we will do so in the future, too. By the above definition, every edge has one or two endpoints; but we prefer to view the one-endpoint case as "there are two endpoints that coincide", too.

Definition. An edge e is a *loop*, if its endpoints coincide.

If $\psi(e) = \{u, v\}$ for an edge $e \in E(G)$, then we will say that "the edge e connects the vertices u and v", "e is an edge between u and v" or "e is incident to u and v", etc. Two vertices are called *adjacent*, if they are connected by an edge. We say that v is a *neighbor* of u, if u and v are adjacent vertices in the multigraph. The *neighborhood* of u, denoted by $N(u)$, is the set of neighbors of u.

Definition. The edges e and f are *parallel edges* (or multiple edges), if they are incident to the same two vertices, i.e. if $\psi(e) = \psi(f)$.

Definition. A *simple graph* is a multigraph that contains no loops or parallel edges. In this book, the term *graph* is just the short form of 'simple graph'.

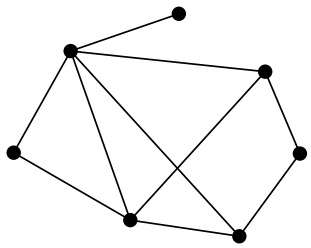

Figure 1.2: A (simple) graph

Remark 1.1. The terminology is not uniform in the literature. Some authors allow graphs to have loops or parallel edges (as they mean multigraphs or loopless multigraphs on graphs), but we do not. When we want to emphasize that loops and parallel edges are forbidden, we shall use the attribute 'simple'.

So in a graph there are no loops, and any two adjacent vertices are connected by exactly one edge. If u and v are adjacent vertices in G, we refer to the edge between them as "the edge uv", and with a slight abuse of notation, we write $uv \in E(G)$.

We end this section with some basic definitions of graph theory.

Definition. The (simple) graphs G and H are said to be *isomorphic*, if there exist a bijection $\phi\colon V(G) \to V(H)$ such that any two vertices u and v are adjacent in G if and only if the vertices $\phi(u)$ and $\phi(v)$ are adjacent in H.

Informally, isomorphic graphs are "essentially the same" (thus they are considered the same in graph theory almost always), the only difference is in the "names" of vertices. We leave the reader to the adopt the definition of graph isomorphism to multigraphs.

Definition. The complement of a graph G, denoted by $\overline{G}$, is a simple graph on the same vertex set as G, such that any two vertices are adjacent in $\overline{G}$ if and only if they are not adjacent in G.

Definition. A *complete graph* is a graph in which every pair of distinct vertices is connected by an edge. The complete graph on n vertices is denoted by K_n.

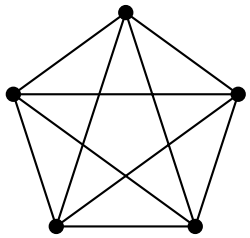

Figure 1.3: K_5, the complete graph on 5 vertices

An *empty graph* is a graph that has no edges. The empty graph on n vertices is denoted by $\overline{K_n}$.

Definition. A *bipartite (multi)graph* G is a (multi)graph whose vertices can be divided into two disjoint sets A and B such that all edges of G connects a vertex in A to a vertex in B, i.e. no two vertices within the same set are adjacent.

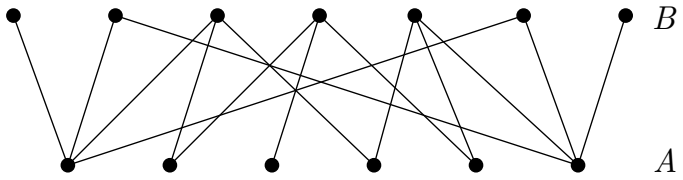

Figure 1.4: A bipartite graph

Definition. The *complete bipartite graph* $K_{m,n}$ is a bipartite graph with bipartition $V = A \cup B$ in which every vertex of A is adjacent to every vertex of B, and $|A| = m$, $|B| = n$.

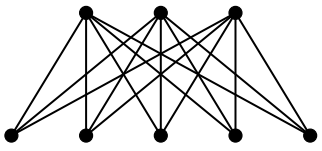

Figure 1.5: The complete bipartite graph $K_{5,3}$

1.2 Degrees and the handshake lemma

Definition. In a multigraph G, the *degree* of a vertex v is the number of edges incident to v, where the loops are counted twice. The degree of v is denoted by $\deg(v)$ or $\deg_G(v)$.

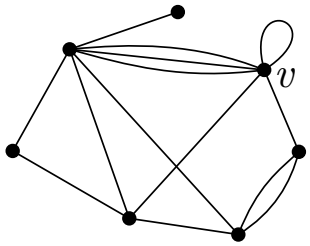

Figure 1.6: A vertex with degree 7

For example, the vertex v has degree 7 in the multigraph in Figure 1.6.

Remark 1.2. Visually speaking, in a multigraph every edge has two 'end segments', associated to the two (not necessarily distinct) endpoints of the edge. In fact, $\deg(v)$ is defined to be the number of 'end segments' incident to v, that is why loops are counted twice in the above definition.

We note that in a (simple) graph, the degree of a vertex v is simply the number of neighbors of v.

Now we are ready to state and prove the first theorem of every graph theory course.

Theorem 1.3 (Handshake lemma)**.** *For any multigraph G,*

$$\sum_{v \in V(G)} \deg(v) = 2|E(G)|.$$

That is, the sum of the degrees of all vertices of G is equal to twice the number of edges of G.

Proof. Both sides of the equation count the total number of 'end segments' of edges in G (cf. Remark 1.2):

- Since every edge has exactly two end segments, the total number of end segments in G is clearly $2|E(G)|$, the right-hand side.
- For any vertex v, the number of end segments incident to v is $\deg(v)$, so the total number of end segments is clearly the sum given in the left-hand side of the equation.

Hence the theorem follows. □

Corollary 1.4. *The sum of the degrees of all vertices is even in any multigraph. In other words, the number of vertices with odd degree is even.*

We end this section with a few definitions related to vertex degrees.

Definition. A vertex is called *isolated* if its degree is 0, i.e. if there are no edges incident to it.

Definition. A multigraph is called *regular* if all of its vertices have the same degree. If the common degree is d in a regular multigraph G, then we also say that G is *d-regular*.

1.3 Subgraphs

Definition. Let G be a multigraph, an edge $e \in E(G)$ and a vertex $v \in V(G)$ of it.

The multigraph $G - e$ obtained by *removing the edge e from G* is defined as

$$\begin{aligned} V(G-e) &:= V(G), \\ E(G-e) &:= E(G) \setminus \{e\}, \\ \psi(G-e) &:= \psi(G)|_{E(G-e)}. \end{aligned}$$

In other words, $G-e$ is the multigraph obtained from G by deleting the edge e from the edge set, and the incidences are inherited from G.

The multigraph $G - v$ obtained by *removing the vertex v from G* is defined as

$$\begin{aligned} V(G-e) &:= V(G) \setminus \{v\}, \\ E(G-e) &:= E(G) \setminus \{e \in E(G) : e \text{ is incident to } v\}, \\ \psi(G-e) &:= \psi(G)|_{E(G-v)}. \end{aligned}$$

In other words, $G - v$ is the multigraph obtained from G by deleting the vertex v (from the vertex set) and the edges incident to v (from the edge set), and the incidences are inherited from G.

The removal of several edges/vertices can be defined as a natural extension of the above definitons. For a set $X \subseteq E(G) \cup V(G)$, the multigraph obtained by removing the elements of X from G is denoted by $G - X$.

Definition. The (multi)graph H is a *sub(multi)graph* of the (multi)graph G, if H can be obtained from G by removing some (or no) edges and vertices. If H is a submultigraph of G, then we also say that G *contains* H.

The sub(multi)graph H is a *spanning sub(multi)graph* of G, if H contains all vertices of G.

The sub(multi)graph H of G is an *induced sub(multi)graph* (on S), if the vertex set of H is a subset $S \subseteq V(G)$, and H contains exactly those edges of G whose both endpoints belong to S. So the induced submultigraph H is determined by the set S, and it is denoted by $G|_S$.

1.4 Walks, tours, paths, cycles

Definition. A *walk* in a multigraph G is a sequence

$$\mathcal{W} : (v_0, e_1, v_1, e_2, v_2, e_3, v_3, \ldots, v_{\ell-1}, e_\ell, v_\ell),$$

where $v_0, v_1, \ldots, v_\ell \in V(G)$, $e_1, \ldots, e_\ell \in E(G)$, and for every e_i $(i = 1, \ldots, \ell)$ its two endvertices are v_{i-1} and v_i. We say that ℓ is the *length* of the walk. $\ell = 0$ is a possibility, '(v_0)' is a path of length 0.

A walk is *closed* iff $v_0 = v_\ell$, otherwise we call it non-closed.

Let $V(\mathcal{W}) = \{v_0, v_1, \ldots, v_\ell\}$, the vertex set of the walk. Since repeated vertices are not forbidden in the sequence of vertices along a walk, $|V(\mathcal{W})| - 1$ can be much

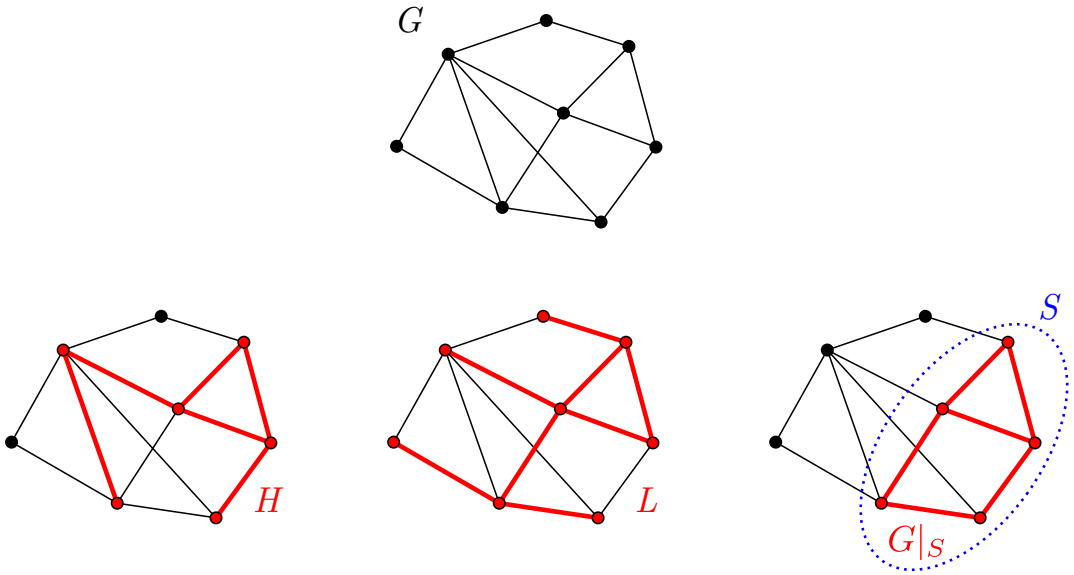

Figure 1.7: A subgraph H, a spanning subgraph L, and an induced subgraph $G|_S$

smaller than the length of $\mathcal{W}$. Let $E(\mathcal{W}) = \{e_1, \ldots, e_\ell\}$, the edge set of the walk. Note again, $|E(G)| \ll \ell$ is possible.

We emphasize a useful view of walks. We consider it as a dynamic process. The indices are denoting 'time'. At the beginning ($t = 0$) we are at vertex v_0 (initial vertex of the walk), from time i to $i+1$ we make a 'step' from v_i to v_{i+1}. The walk ends when the clock hits ℓ (end vertex of the walk), we stop in v_ℓ.

On an *xy-walk* (or *xy-path*) we mean a walk (or path) with initial vertex x and end vertex y.

Definition. A walk is called *tour* iff all e_i's are different, hence $|E(\mathcal{W})|$ is the length of the walk. The notion *trail* is also used for a tour.

A walk is called *path* iff all v_i's are different, hence $|V(\mathcal{W}) - 1|$ is the length of the walk.

A walk is called *cycle* iff $\ell > 0$, and $v_0, v_1, \ldots, v_{\ell-1}$ are different vertices, but $v_\ell = v_0$, furthermore in the case $\ell = 2$ we have $e_1 \neq e_2$.

1.5 Connectivity and trees

This section collects the definitions and fundamental theorems on connectivity as a survey, the proofs are omitted.

Definition. A multigraph G is *connected*, if for any two vertices $x, y \in V(G)$, there exists an xy-walk in G. A multigraph that is not connected is called *disconnected*.

The following lemma shows that the 'xy-walk' can be replaced to 'xy-path' in the above definition.

Lemma 1.5. *Given a multigraph G, and two vertices $x, y \in V(G)$. The following two statements are equivalent:*

(i) There exists an xy-walk in G.

(ii) There exists an xy-path in G.

The following theorem gives the structure of disconnected multigraphs.

Theorem 1.6. *Every multigraph G is a vertex-disjoint union of* connected *multigraphs $G_1 \ldots, G_k$; and this decomposition is unique. (That is, $G_1, \ldots, G_k$ are connected induced submultigraphs of G such that there is no edge in G between G_i and G_j for $i \neq j$.)*

Definition. The vertex-disjoint connected multigraphs $G_1 \ldots, G_k$ in Theorem 1.6 that G decomposes into are called the *(connected) components* of G.

We note that a multigraph is not connected, if and only if it has more than one component.

Now we give some equivalent definitions and the main properties of tree graphs.

Definition. A graph is a *tree*, if it is connected and it does not contain a cycle.

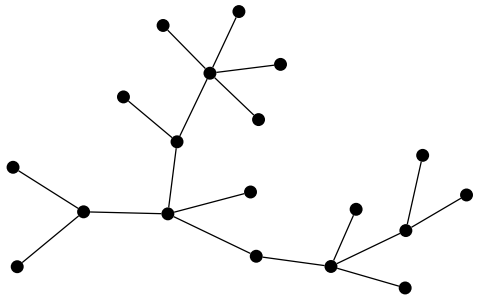

Figure 1.8: A tree graph

Theorem 1.7. *For any graph G, the following statements are equivalent.*

(i) G is a tree.

(ii) G is connected, but the removal of any edge would disconnect it (i.e. $G - e$ is disconnected for all $e \in E(G)$).

(iii) For any two vertices $x, y \in V(G)$, there exists exactly one xy-path in G.

Definition. A vertex with degree 1 in a tree is called a *leaf* of the tree.

Lemma 1.8. *Every tree with at least two vertices has a leaf.*

We will need the following operation: *Adding a pendant edge* to a graph G means that we add a new vertex $v \notin V(G)$ to the graph, and connect v by an edge to exactly one old vertex of G.

Lemma 1.9. *(a) For any leaf u of a tree T, the graph $T - u$ is also a tree.*

(b) Given a tree T, the tree T^* obtained by adding a pendant edge to T is also a tree.

Theorem 1.10 (Structure theorem of trees)**.** *A graph G is a tree if and only if it can be constructed from a single vertex (with no edges) by repeated application of "adding a pendant edge" operation. In other words, a graph G is a tree if and only if there exists a sequence $G_0, G_1, \ldots, G_k$ of graphs such that G_0 is the empty graph on one vertex, $G_k = G$, and G_i is obtained from G_{i-1} by adding a pendant edge, for $i = 1, \ldots, k$. See Figure 1.9.*

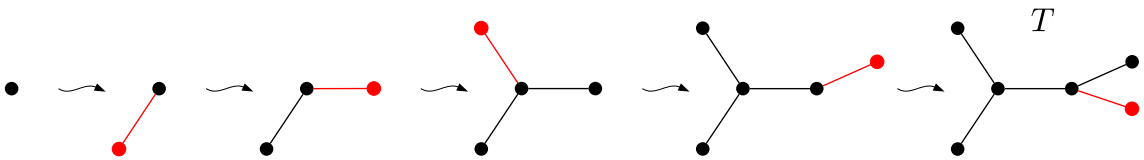

Figure 1.9: Construction of a tree T by adding pendant edges

The structure theorem has an important corollary.

Theorem 1.11. *A tree on n vertices has $n - 1$ edges.*

We will enumerate spanning trees in Chapter 3.

Definition. A *spanning tree* of a multigraph is spanning subgraph which is a tree.

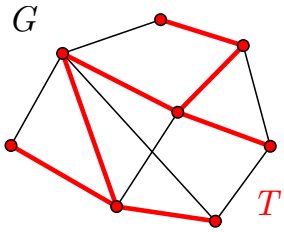

Figure 1.10: A spanning tree

Lemma 1.12. *Every connected graph has a spanning tree.*

Definition. A *rooted tree* is a tree with a designated vertex called the *root*. (Formally, a rooted tree is a pair (T, r) where T is a tree, and $r \in V(T)$.)

Theorem 1.13. *Every rooted tree (T, r) can be drawn like a family tree, as illustrated on the right-hand side of Figure 1.11: The vertices of T are arranged in levels, such that*

(i) there is exactly one vertex on the top level, the root r;

(ii) every edge of T connects two vertices on adjacent levels;

(iii) for any non-root vertex u, there is exactly one edge in T that connects u to a vertex on the level just above the level of u.

We note that the level of any vertex $v \in V(T)$ is uniquely determined. If the length of the unique rv-path in T is ℓ, then v belongs to the $(\ell + 1)^{th}$ level (from top to bottom).

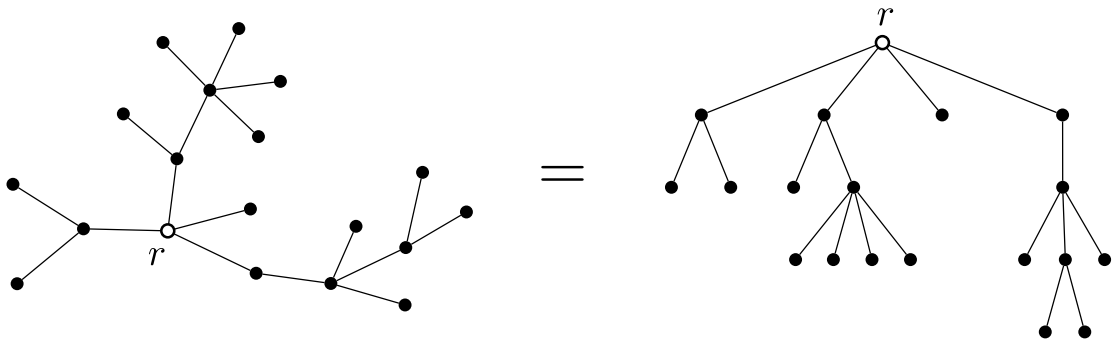

Figure 1.11: A rooted tree with root r and its family tree-like drawing

Chapter 2

Graph realizations

Definition. The *degree sequence* of a multigraph is the sequence of degrees of all its vertices, sorted in nonincreasing order. (A sequence $d_1, d_2, d_3, \ldots$ is *nonincreasing*, if $d_1 \geq d_2 \geq d_3 \geq \ldots$.)

Example. The degree sequence of the multigraph in Figure 1.1 is

$$5, 5, 4, 4, 3, 2, 1.$$

The main problem of this chapter is to decide that whether there exists a graph whose degree sequence is $\mathbf{d}$ or not, for a given sequence $\mathbf{d}$ of integers. This problem is called the graph realization problem.

Definition. We say that a finite sequence $\mathbf{d}$ of integers *can be realized by graph*, if there exists a graph G whose degree sequence is $\mathbf{d}$. (If such a graph G exists, we say that G *realizes* $\mathbf{d}$.)

The realization by multigraph (or by loopless multigraph etc.) is defined analogously.

2.1 Realization by multigraphs

The multigraph realization problem is easy.

Proposition 2.1. *The nonincreasing sequence $d_1, d_2, \ldots, d_n$ of nonnegative integers can be realized by multigraph if and only if the sum $d_1 + d_2 + \cdots + d_n$ is even.*

Proof. Assume first that the sequence $d_1, \ldots, d_n$ can be realized by G. By the corollary of handshake lemma (Corollary 1.4), the sum of degrees is even in G, which means that $d_1 + \cdots + d_n$ is even.

For the converse, fix a nonincreasing sequence $d_1, \ldots, d_n$ of nonnegative integers with the property that $d_1 + \cdots + d_n$ is even. We construct a multigraph G on vertex set $\{v_1, \ldots, v_n\}$ such that $\deg(v_i) = d_i$, for $i = 1, \ldots, n$. The existence of such G will complete the proof. We start with the empty graph on vertex set $\{v_1, \ldots, v_n\}$, then we add $\lfloor d_i/2 \rfloor$ loops to vertex v_i, for $i = 1, \ldots, n$. At this stage, in the obtained multigraph

$$\deg(v_i) = \begin{cases} d_i, & \text{if } d_i \text{ is even} \\ d_i - 1, & \text{if } d_i \text{ is odd} \end{cases}$$

for all i. This means that for even d_i's, the corresponding vertex v_i has already had the required degree d_i, but for odd d_i's, one end segment of an edge is still missing from v_i. This latter issue will be resolved by adding new non-loop edges. Since $d_1 + d_2 + \cdots + d_n$ is even, thus the number of odd d_i's is even, and so the number of v_i's with missing end segment is even. So these v_i's can be grouped into pairs, and then we can add an edge between every two vertices belonging to the same pair. In this way all the missing end segments are added, so the obtained multigraph G has degree sequence $d_1, \ldots, d_n$, as required. The construction is illustrated in Figure 2.1. □

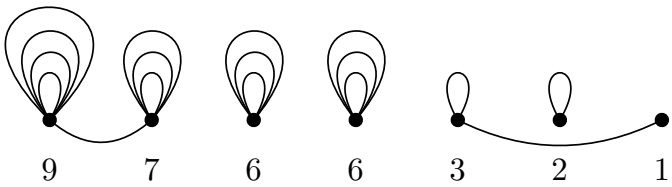

Figure 2.1: A multigraph realization of the sequence $9, 7, 6, 6, 3, 2, 1$.

.

For completeness, we present the answer to the loopless multigraphs realization problem, but we omit the proof.

Proposition 2.2. *The* nonincreasing *sequence $d_1, d_2, \ldots, d_n$ of nonnegative integers can be realized by loopless multigraph if and only if*

- *$d_1 + d_2 + \cdots + d_n$ is even, and*
- *$d_1 \leq d_2 + d_3 + \cdots + d_n$.*

2.2 Realization by graphs

Now we discuss the realization problem for graphs. This a much more difficult scenario than the previous ones. Instead of giving an explicit description of the degree sequences, first we present an algorithm to decide whether a sequence can be realized by graph, due to Havel and Hakimi. The algorithm is based on the following key observation.

Lemma 2.3 (Havel–Hakimi)**.** *The* nonincreasing *sequence of nonnegative integers*

$$d_1, d_2, \ldots, d_n$$

can be realized by simple graph, if and only if $d_1 \leq n - 1$ and the sequence

$$d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3}, \ldots, d_n$$

can be realized by simple graph (after reordering the sequence in nonincreasing order).

Remark 2.4. Note that the second sequence in the lemma can be obtained from the first one by the following operation: Remove the first element, d_1, from the sequence $d_1, \ldots, d_n$, and then in the obtained sequence $d_2, \ldots, d_n$ decrease the first d_1 elements by 1. In the future we will denote this operation by HH, so if the first sequence is denoted by $\mathbf{d}$, then the second sequence is HH($\mathbf{d}$). We apply this operation to nonincreasing sequences only.

Proof of Lemma 2.3. We denote the sequence $d_1, \ldots, d_n$ by $\mathbf{d}$, and the second sequence of the lemma by HH($\mathbf{d}$).

It is easy to see that the conditions are sufficient. Assume that the sequence

$$d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3}, \ldots, d_n$$

can be realized by a simple graph G. If we add a new vertex v and connect it to those vertices of G which have degrees $d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1$, then we obtain a simple graph G^+ whose degree sequence is $\mathbf{d}$. (The degree of v is d_1 in G^+, and its neighbors' degree has been increased by 1, becoming $d_2, \ldots, d_{d_1+1}$.) Hence the sequence $\mathbf{d}$ can be indeed realized.

For the converse direction, assume that the sequence $\mathbf{d}$ can be realized by a simple graph G_0. The vertices of G_0 are denoted by $v_1, \ldots, v_n$ such that $\deg(v_i) = d_i$, for all i. In a simple graph every degree is at most $|V|-1$, hence the inequality $d_1 \leq n-1$ clearly holds. To see that HH($\mathbf{d}$) can be realized, we will modify G_0 by adding and deleting edges (without introducing loops or parallel edges), so that in the obtained graph G the vertex degrees are unchanged and the neighbors of v_1 are precisely the vertices $v_2, v_3, \ldots, v_{d_1+1}$. Then the degree sequence of $G - v$ is clearly HH($\mathbf{d}$), which means that HH($\mathbf{d}$) can be realized.

Thus in order to complete the proof, it is enough to show that such a modification of G_0 can be done. Recall that $\deg(v_1) = d_1$. If v_1 is adjacent to all the vertices $v_2, \ldots, v_{d_1+1}$, then there is nothing to do, we are done. If this is not the case, we will just see that we can make a degree-preserving modification step on G_0 that increases the number of those vertices of $\{v_2, \ldots, v_{d_1+1}\}$ that are adjacent to v_1. And we can repeatedly apply this step until all the vertices $v_2, \ldots, v_{d_1+1}$ become adjacent to v_1, which is our goal.

So assume that there is a vertex $s \in \{v_2, \ldots, v_{d_1+1}\}$ for which $v_1 s \notin E(G_0)$. As $\deg(v_1) = d_1$, there must be a vertex $t \notin \{v_2, \ldots, v_{d_1+1}\}$ in G_0 for which $v_1 t \in E(G_0)$. Since the sequence $\mathbf{d}$ is nonincreasing, $\deg(s) \geq \deg(t)$ holds. This, together with the facts $v_1 s \notin E(G_0)$ and $v_1 t \in E(G_0)$, implies that in G_0 there exists a neighbor w of s (different from v_1 and t) for which $tw \notin E(G_0)$. In sum, we have found v_1, s, w and t, four different vertices of G_0, for which $v_1 s \notin E(G_0)$, $sw \in E(G_0)$, $wt \notin E(G_0)$ and $tv_1 \in E(G_0)$. Now we remove the edges tv_1 and sw from G_0, and add the edges $v_1 s$ and wt, see Figure 2.2. Then we obtain a simple graph G_1 *with the same vertex degrees as in* G_0, in which one more vertex of $\{v_2, \ldots, v_{d_1+1}\}$ is connected to v_1 than in G_0 (since the only changes on the neighborhood of v_1 are that s entered to it and t left it, where $s \in \{v_2, \ldots, v_{d_1+1}\}$, $t \notin \{v_2, \ldots, v_{d_1+1}\}$). □

This lemma reduces a graph realization problem to an other graph realization problem with smaller input size, so the lemma can be used to solve the problem recursively.

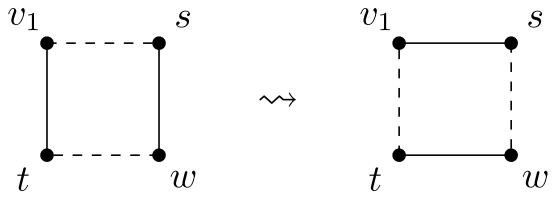

Figure 2.2: Flip of edges in the proof of Lemma 2.3

.

Theorem 2.5 (Havel–Hakimi algorithm)**.** *For a given nonincreasing sequence* $\mathbf{d}$ *of nonnegative integers, the following algorithm decides whether* $\mathbf{d}$ *can be realized by simple graph or not. Moreover, if the answer is yes, a realization graph can be easily constructed, see Remark 2.4.*

Havel–Hakimi algorithm (on input sequence $\mathbf{d}$*):*

- If $\mathbf{d}$ is a one-element sequence, then it can be realized by simple graph if and only if it is the sequence 0, and the algorithm terminates.
- If the first element of $\mathbf{d}$ is greater than or equal to the number of elements of $\mathbf{d}$, then $\mathbf{d}$ cannot be realized and the algorithm terminates.
- Otherwise, calculate HH($\mathbf{d}$).
- If the sequence HH($\mathbf{d}$) contains a negative number, then $\mathbf{d}$ cannot be realized and the algorithm terminates.
- Otherwise, reorder the sequence HH($\mathbf{d}$) in nonincreasing order, and invoke this algorithm recursively on input sequence HH($\mathbf{d}$). The obtained answer is the answer to the initial question (on input $\mathbf{d}$).

The Havel–Hakimi-algorithm is just the repeated application of Lemma 2.3. It can be best understood through examples.

Example. As a first example, we decide, using the Havel–Hakimi-algorithm, if the sequence $7, 4, 3, 3, 3, 3, 2, 1, 0$ can be realized by simple graph:

$$
\begin{array}{c}
7,4,3,3,3,3,2,1,0 \text{ can be realized} \\
\Updownarrow \\
3,2,2,2,2,1,0,0 \text{ can be realized} \\
\Updownarrow \\
1,1,1,2,1,0,0 \text{ can be realized} \\
\| \text{ (reorder)} \\
2,1,1,1,1,0,0 \text{ can be realized} \\
\Updownarrow \\
0,0,1,1,0,0 \text{ can be realized} \\
\| \text{ (reorder)}
\end{array}
$$

$$1,1,0,0,0,0 \text{ can be realized}$$
$$\Updownarrow$$
$$0,0,0,0,0 \text{ can be realized.}$$

Since the sequence $0,0,0,0,0$ can be trivially realized by the empty graph on five vertices, hence the initial sequence $7,4,3,3,3,3,2,1,0$ can realized, too. (We note that Theorem 2.5 defines the algorithm to run until it reaches to the one-element sequence 0. But, of course, we can stop at any point when we see that the actual sequence can be realized. And this is the case for all-0 sequences, for example.) □

Remark 2.6. It is important to note that in case of positive answer, the Havel–Hakimi-algorithm not only proves the existence of a graph that realizes the input sequence, but such a graph can be easily read off from the sequences that appears during the algorithm's run. The point is that if we know a realization graph G for the sequence $\mathrm{HH}(\mathbf{d})$, then a realization graph for the sequence $\mathbf{d}$ can be easily constructed from G: This is the first (easy) part of the proof of Lemma 2.3 (add a new vertex to G and connect it to the vertices with decreased degrees). So we can go through the sequences appearing during the algorithm's run in reverse order, and starting from the trivial realization of the final sequence, reach to a realization of the input sequence of the algorithm. In the example above, from the (empty graph) realization of $0,0,0,0,0$, we can obtain a realization for $1,1,0,0,0,0$ by the above method. And from the realization of $1,1,0,0,0,0$, we can obtain a realization for $2,1,1,1,1,0,0$, and so on.

Example. As a second example, we decide if the sequence $8,8,6,6,6,5,3,2,2$ can be realized by simple graph:

$$8,8,6,6,6,5,3,2,2 \text{ can be realized}$$
$$\Updownarrow$$
$$7,5,5,5,4,2,1,1 \text{ can be realized}$$
$$\Updownarrow$$
$$4,4,4,3,1,0,0 \text{ can be realized}$$
$$\Updownarrow$$
$$3,3,2,0,0,0 \text{ can be realized}$$
$$\Updownarrow$$
$$2,1,-1,0,0 \text{ can be realized.}$$

Clearly, the sequence $2,1,-1,0,0$ cannot be realized by graph as it contains a negative number, hence the inital sequence cannot be realized neither. □

There is an explicit description of degree sequences of simple graphs, due to Erdős and Gallai. We omit the proof.

Theorem 2.7 (Erdős-Gallai)**.** *The* nonincreasing *sequence $d_1, d_2, \ldots, d_n$ of nonnegative integers can be realized by simple graph if and only if*

- *$d_1 + \cdots + d_n$ is even, and*

- *for all* $k \in \{1, \ldots, n\}$,

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k).$$

2.3 Realization by trees

We end this chapter with the realization problem on trees.

Proposition 2.8. *For $n \geq 2$, the nonincreasing sequence $d_1, \ldots, d_n$ of nonnegative integers can be realized by tree if and only if $\sum_{i=1}^{n} d_i = 2(n-1)$ holds and $d_i > 0$ for all i.*

Proof. If the sequence can be realized by a tree T, then by the handshake lemma,

$$\sum_{i=1}^{n} d_i = 2|E(T)| = 2(n-1),$$

using also the fact that a tree on n vertices has $n-1$ edges. And a tree (on at least two vertices) cannot have isolated vertex, so $d_i > 0$ holds for all i, too.

Conversely, the statement "if $\sum_{i=1}^{n} d_i = 2(n-1)$, and $d_i > 0$ for all i, then the sequence $d_1, \ldots, d_n$ can be realized by tree" is proved by induction on n. The base case $n = 2$ can be verified easily. $(1, 1$ is the only sequence that satisfies the conditions, which can be clearly realized.) For the inductive step, assume that the sequence $d_1, \ldots, d_n$ satisfies the conditions. Then observe that the average of the numbers $d_1, \ldots, d_n$ is between 1 and 2, because

$$\frac{1}{n}\sum_{i=1}^{n} d_i = \frac{1}{n} \cdot 2(n-1) = 2 - \frac{1}{n}$$

and $n \geq 2$. This implies that d_1, the largest element, must be at least 2, and d_n, the smallest element, must be at most 1. Since $d_n > 0$ by the conditions, we conclude that $d_n = 1$. Hence the $(n-1)$-element sequence $d_1 - 1, d_2, d_3, \ldots, d_{n-1}$ satisfies the conditions of the theorem, as $d_1 - 1 > 0$ and the sum of its elements is $2(n-2)$. So, by the induction hypothesis, this sequence can be realized by a tree T' on $n-1$ vertices. If we add a new vertex to T' and join it to the vertex of degree $d_1 - 1$, then we obtain a tree T that realizes the sequence $d_1, \ldots, d_n$, completing the proof. □

Remark 2.9. The inductive part of the proof can be easily algorithmized to construct a tree that realizes a given nonincreasing sequence $d_1, \ldots, d_n$ satisfying the conditions of Proposition 2.8: Invoke the algorithm recursively on the sequence $d_1 - 1, d_2, d_3, \ldots, d_{n-1}$ (after reordering it in nonincreasing order), and then add a new vertex to the output tree T' and connect it to the vertex of degree $d_1 - 1$. The recursion terminates when the number of vertices reaches 2.

In the next chapter we will prove even more on this subject, Corollary 3.7 gives the exact number of trees realizing a given sequence.

Chapter 3

Enumeration of spanning trees

This chapter deals with the number of spanning trees of a given graph. (Recall the definion of spanning trees from Chapter 1.) In this subject, two spanning trees are considered the same if and only if they have the same edge set.

3.1 Spanning trees of complete graphs

The following classical enumeration result gives the number of spanning trees in complete graphs, cf. Figure 3.1.

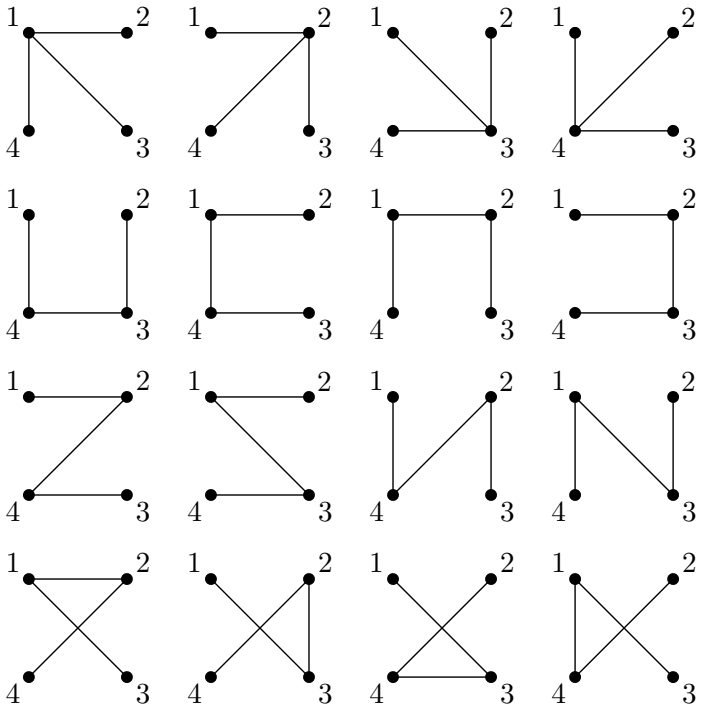

Figure 3.1: The spanning trees of K_4

Theorem 3.1 (Cayley)**.** *The number of spanning trees of the n-vertex complete graph is* n^{n-2}*.*

We will prove Cayley's theorem soon by giving a bijection between the set of spanning trees of K_n and the set of sequences of $n-2$ elements of $\{1, 2, \ldots, n\}$.

But first we look at the spanning trees of the complete graph K_n from a different viewpoint. As every two vertices are adjacent in K_n, a spanning tree of K_n is just a tree on the vertex set of K_n. We can assume that $V(K_n) = \{1, \ldots, n\}$, so a spanning tree of K_n is just a tree on vertex set $\{1, \ldots, n\}$. We will use this viewpoint if we do not want to deal with the underlying graph K_n, and we even use a different terminology then.

Definition. A *labeled tree on n vertices* is a tree on vertex set $\{1, \ldots, n\}$. In order to be consistent with the spanning tree enumeration problem, the labeled trees T_1 and T_2 (on n vertices) are considered the same if and only if exactly the same pairs of vertices are adjacent in T_1 as in T_2.

Remark 3.2. The name 'labeled tree' reflects the fact that we usually think of vertex i (where $i \in \{1, \ldots, n\}$) as a node labeled with the number i, see Figure 3.2. Hence sometimes we refer to vertex i as vertex with *label* i, too.

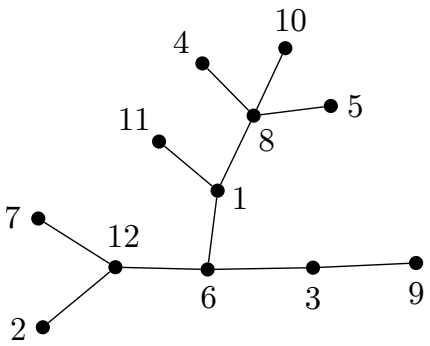

Figure 3.2: A labeled tree on 12 vertices

With this terminology, Cayley's theorem can be formulated as follows.

Theorem 3.3 (Cayley)**.** *The number of labeled trees on n vertices is n^{n-2}.*

As promised, we encode now the labeled trees on n vertices into sequences of $n-2$ elements of $\{1, 2, \ldots, n\}$, due to Prüfer.

Definition (Prüfer encoding)**.** Let T be a labeled tree on n vertices, where $n \geq 2$. The *Prüfer code* of T, denoted by $\Pr(T)$, is a sequence defined by the following procedure:

- Find the leaf of T which has the smallest label among the leaves of T; and let us denote this leaf by u_1. Add the label of u_1's (unique) neighbor to $\Pr(T)$ as first element, and then remove the leaf u_1 from T.
- Repeat the previous step on the obtained labeled tree $T - u_1$ (cf. Lemma 1.9.a): Find the leaf of $T - u_1$ with smallest label, say u_2, and add the label of u_2's neighbor to $\Pr(T)$ as second element, then remove the leaf u_2.

- And so on, in the k^{th} step, we define the k^{th} element of $\Pr(T)$ to be the label of the neighbor of the smallest-labeled leaf in $T^{(k-1)}$, where $T^{(k-1)}$ is the tree obtained from T after removing the vertices $u_1, \ldots, u_{k-1}$ in the former steps; then remove the smallest-labeled leaf u_k from this tree.

- The process terminates after $n-2$ step, i.e. when the number of vertices in the actual tree is 2. (At that point the modification step is not executed further.)

We note that it is obvious from the above description that $\Pr(T)$ has $n-2$ elements and its elements are from the set $\{1, \ldots, n\}$.

Example. For example, the Prüfer code of the labeled tree in Figure 3.2 is

$$12, 8, 8, 12, 3, 6, 8, 1, 1, 6.$$

Cayley's theorem (Theorem 3.3) is directly implied by the following lemma.

Lemma 3.4 (Prüfer)**.** *Let n be a fixed integer at least 2. The mapping $T \mapsto \Pr(T)$ is a bijection from the set of labeled trees on n vertices to the set of $(n-2)$-element sequences of $\{1, \ldots, n\}$.*

Proof. We will prove a slightly more general statement. For any finite set $V \subset \mathbb{N}$, we can consider labeled trees on V, by extending the original definition to arbitrary vertex (label) set: A labeled tree on V is just a tree on vertex set V (interpreting its vertices as labels). The definition of Prüfer encoding also applies to labeled trees on vertex set V without any modification. We will prove the following statement by induction on n: "For any fixed finite set $V \subset \mathbb{N}$ with $n \geq 2$ elements, the mapping $T \mapsto \Pr(T)$ is a bijection from the set of labeled trees on V to the set of $(n-2)$-element sequences of V."

The case $|V| = 2$ is obvious, so we can assume that $|V| > 2$. We have to prove that for any $(n-2)$-element sequence $\mathbf{s}$ of V, there exists a unique labeled tree on V whose Prüfer code is $\mathbf{s}$. To this end, fix an arbitrary $(n-2)$-element sequence $\mathbf{s} = (s_1, \ldots, s_{n-2})$ of V, and assume that $T_{\mathbf{s}}$ is a labeled tree on V for which $\Pr(T_{\mathbf{s}}) = \mathbf{s}$.

Observe first that vertex v of the labeled tree T is a leaf if and only if v is not contained in the sequence $\Pr(T)$. This is because if v is a leaf, then we never remove the neighbor of v during the Prüfer encoding (and hence v is never added to the Prüfer code), because we always remove a leaf, and the neighbor of v can be also a leaf only if there are no other vertices in the tree (when the process already terminates). And if v is not a leaf, then during the Prüfer enconding the degree of v must decrease at some point (because v will become a leaf eventually, no matter whether v will be deleted from the tree or v is one of the two surviving vertices); and when the degree of a vertex decreases then the vertex is added to the Prüfer code.

This means that we know an important information on the tree $T_{\mathbf{s}}$ (with Prüfer code $\mathbf{s}$): The leaves of $T_{\mathbf{s}}$ must be exactly those vertices from V that are not contained in the sequence $\mathbf{s}$. (At least two such vertices exist, because while V has n vertices, $\mathbf{s}$ has only $n-2$ elements.) Let u denote the smallest of the leaf vertices. We know that in the first step of the Prüfer encoding of $T_{\mathbf{s}}$, the removed leaf is u and it was

connected to vertex s_1. And then the obtained tree $T_{\mathbf{s}} - u$, a labeled tree on vertex set $V \setminus \{u\}$, was Prüfer-encoded into $(s_2, \ldots, s_{n-2})$, a sequence of $n-3$ elements of $V \setminus \{u\}$. By the induction hypothesis, there exists exactly one labeled tree on $V \setminus \{u\}$, say T', whose Prüfer code is $(s_2, \ldots, s_{n-2})$. So $T_{\mathbf{s}} - u$ must be this tree T', and hence $T_{\mathbf{s}}$ can only be the tree obtained from T' by adding vertex u and connecting it to s_1.

It is straightforward to verify that the Prüfer code of this unique tree T_s is indeed $\mathbf{s}$. (We have to check that after connecting u to vertex s_1 of T', u *indeed* becomes the smallest leaf (label) in the obtained tree $T_{\mathbf{s}}$ – this is required to verify that u is the first removed vertex in the Prüfer encoding of $T_{\mathbf{s}}$, as expected. This can be easily done by determining the leaves of T' from its Prüfer code $(s_2, \ldots, s_{n-2})$, as discussed in the third paragraph of this proof.) □

The above proof is a bit terse, an example will shed more light on the recursive reconstruction of the labeled tree from its Prüfer code.

Example (The inverse of Prüfer encoding)**.** As an example, we will find the unique labeled tree T (on 9 vertices) whose Prüfer code is the following 7-element sequence of $\{1, \ldots, 9\}$:

$$4, 1, 4, 2, 4, 9, 2.$$

For shortness, vertex v and label v will be identified (like in the formal definition of labeled trees). And $T^{(i)}$ will denote the tree obtained from T after processing the first i steps (vertex removals) of the Prüfer encoding of T.

1. The leaves of T are exactly those elements of $V(T) = \{1, \ldots, 9\}$ which are not contained in the sequence $4, 1, 4, 2, 4, 9, 2$; so the leaves of T are precisely $3, 5, 6, 7, 8$. The smallest of them, vertex '3', was removed in the first step of the Prüfer endoding, and it was adjacent to the first element of the sequence, vertex '4'. So we conclude that 34 is an edge of T. (Here 34 denotes an edge connecting the vertices '3' and '4'.)

2. The leaves of $T^{(1)}$, the tree obtained from T after removing vertex '3', are exactly those elements of $V(T^{(1)}) = \{1, \ldots, 9\} \setminus \{3\}$ which are not contained in the truncated sequence $1, 4, 2, 4, 9, 2$; i.e. the leaves of $T^{(1)}$ are precisely $5, 6, 7, 8$. This means that in the second step of the Prüfer encoding, the smallest of them, vertex '5' was removed from $T^{(1)}$, and it was adjacent to '1', so 51 is also an edge of T.

3. In the third step of the Prüfer encoding of T, the removed leaf of $T^{(2)}$ was '1', the smallest element of $V(T^{(2)}) = \{1, \ldots, 9\} \setminus \{3, 5\}$ which is not contained in the sequence $4, 2, 4, 9, 2$. The neighbor of the removed leaf '1' is '4', so 14 is an edge of T.

4-7. And so on, we can figure out in a similar way that the vertices '6', '7', '4' and '8' were removed in the 4^{th}, 5^{th}, 6^{th} and 7^{th} steps of the Prüfer encoding, respectively. The second row of Table 3.1 contains the sequence of removed vertices; the vertex in the i^{th} position was removed in the i^{th} step. (Remark 3.5 will discuss a mechanical way to fill this table.) The neighbors can be read off from the Prüfer code, so we found the edges 62, 74, 49 and 82 in T.

Prüfer code	4	1	4	2	4	9	2
removed leaf	3	5	1	6	7	4	8

Table 3.1: The steps of reconstruction

8. Finally, we know that after removing the seven vertices determined above, we end up with a 2-vertex tree, i.e. the two remaining vertices, '2' and '9' are connected by edge, and hence 29 is also an edge in T. Now we have determined all edges of T: they are $34, 51, 41, 62, 74, 49, 82$ and 29, so we conclude that T is the labeled tree on Figure 3.3.

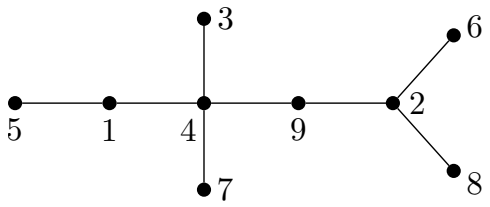

Figure 3.3: The solution to the exercise

We note that the reasonings in steps 1-8 do not show *why* the determined edges form a tree. That follows from the inductive argument of the proof of Lemma 3.4. □

Remark 3.5. It is easy to see that both the Prüfer encoding and its inverse can be implemented efficiently on computer. For example, the construction of Table 3.1, the heart of the above inversion algorithm, can be summarized as follows. The elements of the second row are filled from left to right, such that the i^{th} element of the second row is the smallest number in $\{1, \ldots, n\}$ which occurs neither among the first $i-1$ elements of the second row nor among the last $n-i+1$ elements of the first row.

The observation made in the proof of Lemma 3.4 can be extended, which can be used to count the number of trees with a given degree sequence.

Lemma 3.6. *For an arbitrary labeled tree T, any vertex v of T occurs exactly* $\deg(v)-1$ *times in the Prüfer code of T.*

Proof. The proof is left to the reader as an exercise. □

Corollary 3.7. *For $n \geq 2$, let $d_1, \ldots, d_n$ be a sequence of integers that can be realized by tree, that is, by Proposition 2.8, a sequence for which $\sum_{i=1}^{n} d_i = 2(n-1)$ holds and $d_i > 0$ for all i. Then the number of those (labeled) trees on vertex set $\{1, \ldots, n\}$ in which $\deg(i) = d_i$ holds for $i = 1, \ldots, n$, is*

$$\frac{(n-2)!}{(d_1-1)!(d_2-1)!\ldots(d_n-1)!}.$$

Proof. Since the Prüfer encoding encodes trees on vertex set $\{1, \ldots, n\}$ into $(n-2)$-element sequences of $\{1, \ldots, n\}$ bijectively, it is enough to count those $(n-2)$-element sequences of $\{1, \ldots, n\}$ which belong to trees satisfying the degree conditions $\deg(i) = d_i$. Fortunately, the degrees of vertices can be easily read off from the Prüfer code by Lemma 3.6: We have to count those $(n-2)$-element sequences in which the element i occurs exactly $d_i - 1$ times, for all $i \in \{1, \ldots, n\}$. (This makes sense because $d_i - 1 \geq 0$ for all i, and $\sum_{i=1}^{n}(d_i - 1) = n - 2$, as implied by the conditions.) These sequences are exactly the permutations of the $(n-2)$-element multiset

$$\{\underbrace{1, 1, \ldots, 1}_{d_1 - 1 \text{ times}}, \underbrace{2, 2, \ldots, 2}_{d_2 - 1 \text{ times}}, \ldots, \underbrace{n, n, \ldots, n}_{d_n - 1 \text{ times}}\}.$$

The well-known formula on the number of permutations of a multiset yields the answer

$$\frac{(n-2)!}{(d_1 - 1)!(d_2 - 1)! \ldots (d_n - 1)!}$$

to the enumeration problem. □

3.2 Spanning trees of arbitrary graphs

Without a proof, we present a famous theorem that expresses the number of spanning trees of a given graph as a determinant. What makes it applicable in practice is that determinants can be calculated effectively by computer. In addition, the theorem has many theoretical consequences, too – as an application, we deduce Cayley's theorem from it.

Theorem 3.8 (Kirchhoff's matrix tree theorem)**.** *A graph G on vertex set $\{v_1, \ldots, v_n\}$ is given. The $n \times n$ matrix L_G is defined as follows. $(L_G)_{ij}$, the element lying in the i^{th} row and the j^{th} column of L_G, is*

$$\begin{cases} \deg_G(v_i), & \text{if } i = j \\ 0, & \text{if } i \neq j, \text{ and } v_i \text{ is not adjacent to } v_j \\ -1, & \text{if } i \neq j, \text{ and } v_i \text{ is adjacent to } v_j. \end{cases}$$

In words, the diagonal elements of L_G are the degrees of vertices, and every off-diagonal element is equal to (-1) times the number of edges between the two corresponding vertices. And let $L_G^{(-i)}$ denote the $(n-1) \times (n-1)$ matrix obtained from L_G by deleting the i^{th} row and the i^{th} column of it.

Then the determinant of $L_G^{(-i)}$ counts the number of spanning trees of G, for any fixed $i \in \{1, \ldots, n\}$.

Example (Cayley's theorem as a corollary of the matrix tree theorem)**.** As an application, we deduce the number of spanning trees of the complete graph K_n again, now using the matrix tree theorem.

Following the notations of the theorem, we clearly have

$$L_{K_n} = \begin{pmatrix} n-1 & -1 & -1 & \cdots & -1 \\ -1 & n-1 & -1 & \cdots & -1 \\ -1 & -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & n-1 \end{pmatrix}_{n\times n},$$

and hence

$$L_{K_n}^{(-1)} = \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix}_{(n-1)\times(n-1)}.$$

By the matrix tree theorem, the number of spanning trees of K_n is equal to $\det(L_{K_n}^{(-1)})$, so we have to calculate this determinant.

$$\begin{aligned} \det(L_{K_n}^{(-1)}) &= \begin{vmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{vmatrix}_{(n-1)\times(n-1)} \\ &= \begin{vmatrix} 1 & 1 & \cdots & 1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{vmatrix}_{(n-1)\times(n-1)} = \begin{vmatrix} 1 & 1 & \cdots & 1 \\ 0 & n & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & n \end{vmatrix}_{(n-1)\times(n-1)} \\ &= n^{n-2}. \end{aligned}$$

The second determinant was obtained from the first one by adding the $2^{\text{nd}}, 3^{\text{rd}}, \ldots, (n-1)^{\text{th}}$ rows to the 1^{st} row, and then the third determinant was obtained from the second one by adding the 1^{st} row to the $2^{\text{nd}}, 3^{\text{rd}}, \ldots, (n-1)^{\text{th}}$ rows. (These operations do not change the value of the determinant.) Finally, we used the linear algebra fact that the determinant of an upper triangular matrix is equal to the product of the elements in its main diagonal. □

Chapter 4

Network flow problems

4.1 Network Flow Problems

The mathematical abstraction of networks is particularly useful. One can model a water pipe network or currents in an electrical network (and many other real life problems) using network flows, and it is also a valuable tool for studying certain combinatorial optimization problems. In this chapter we will present the basics of the area and some applications.

First a notation. Given a directed multigraph $G = (V, E)$ and any vertex $v \in V$, we let $E^+(v)$ denote the set of edges leaving v, and $E^-(v)$ denotes the set of edges entering v. Hence, the outdegree of v is $deg^+(v) = |E^+(v)|$ and the indegree of v is $deg^-(v) = |E^-(v)|$.

Definition. Let $G = G(V, E)$ be a directed multigraph with two distinguished vertices, s and t such that $s \neq t$. Let $c : E \longrightarrow \mathbb{R}^+$ be the *function*. In a network $\mathbb{N}(G, s, t, c)$ with underlying directed braph G we call s the source and t the sink, and c is the capacity function of the edges of G.
Given a function $f : E \longrightarrow \mathbb{R}$ we say that f is a feasible flow in $\mathbb{N}$ if the following conditions hold:

Capacity constraints : for every $e \in E(G)$ we have $0 \leq f(e) \leq c(e)$.

Conservation constraints: for every $v \in V - \{s, t\}$ we have

$$\sum_{e \in E^+(v)} f(e) = \sum_{e \in E^-(v)} f(e)$$

The value of a feasible flow f is defined to be $val(f) = \sum_{e \in E^+(s)} f(e) - \sum_{e \in E^-(s)} f(e)$.

Given subsets $S \subset V, s \in S$ and $T \subset V, t \in T$ such that $S \cup T = V$ and $S \cap T = \emptyset$ is called an $[S, T]$-cut (or source/sink-cut). The edge set of the cut is $E(\{S, T\})$ that contains exactly those edges with one endpoint in S and the other endpoint in T. The edge set $E([S, T])$ can naturally be divided into two disjoint subsets: $\overrightarrow{E} = \overrightarrow{ST}$

contains the edges that point from S towards T, while $\overleftarrow{E} = \overrightarrow{TS}$ contains the edges with tail in T and head in S.

Given an $[S,T]$-cut and a feasible flow f in a network $\mathbb{N}$ the value of the cut is defined to be $val(S,T) = \sum_{e \in \overrightarrow{ST}} f(e) - \sum_{e \in \overrightarrow{TS}} f(e)$.

Our goal is to find a feasible flow having maximum value. The following helps in achieving this goal.

Lemma 4.1.

$$val(S,T) = val(f).$$

Proof. By induction on the cardinality of S. It clearly holds when $|S| = 1$, that is, when $S = \{s\}$. One only have to verify that the value does not change when placing an arbitrary vertex other than t from T to S. □

Definition. The capacity of an $[S,T]$-cut is the total capacity of edges in $\overrightarrow{ST}$, that is, we sum up the capacities of edges with tail in S and head in T. It is denoted by $c(S,T)$.

Easy to see:

$$\max_{f \text{ is a flow}} val(f) \leq \min_{[S,T]-\text{cut}} c(S,T).$$

The theorem below shows that much more is true (we will prove this later).

Theorem 4.2. *[Maximum Flow–Minimum Cut theorem] Let $\mathbb{N}(G,s,t,c)$ be a network. Then*

$$\max_{f \text{ is a flow}} val(f) = \min_{[S,T]-\text{cut}} c(S,T).$$

We will refer to this result as the MFMC theorem.

Remark 4.3. When the two sides equal for some feasible flow f and an $[S,T]$-cut as in the MFMC theorem, then the edges from S to T "work" at full capacity, while f is zero on every edge that goes from T to S.

Definition. Let P be an *undirected* $s-t$ path, i.e., a path which leads from s to t in the graph we obtain from G by making its edges undirected. Assume that f is a feasible flow in $\mathbb{N}(\overrightarrow{G},s,t,c)$. We divide the edges of P into two disjoint subsets, $E^{fwd}(P)$ and $E^{bwd}(P)$. The edge $e \in E(\overrightarrow{G}) \cap E(P)$ belongs to $E^{fwd}(P)$ if we traverse e according to its orientation when going from s to t along P. Otherwise, if we traverse e against its orientation, then $e \in E^{bwd}(P)$. We say that P is an augmenting path with respect to flow f if

- $f(e) < c(e)$ for every $e \in E^{fwd}(P)$ and

- $f(e) > 0$ for every $e \in E^{bwd}(P)$.

Let

$$\delta^{fwd} := \min\{c(e) - f(e) : e \in E^{fwd}(P)\},$$
$$\delta^{bwd} := \min\{f(e) : e \in E^{bwd}(P)\},$$

and

$$\delta := \min\{\delta^{fwd}, \delta^{bwd}\}.$$

We have the following lemma.

Lemma 4.4. *Let f be a feasible flow in the network $\mathbb{N}(G, s, t, c)$. If $\mathbb{N}$ has an augmenting path P with respect to f, then the value of f is not optimal, one can find another feasible flow f' in $\mathbb{N}$ for which*

$$val(f) + \delta = val(f'),$$

where δ is as defined above.

Proof. The proof essentially consists of two observations. The first one is that f' is a feasible flow using the definition of δ, neither the capacity constraints, nor the conservations constraints are violated. Secondly, if we take an arbitrary $[S, T]$-cut, its value increases precisely by δ. These observations prove what was desired. □

One might think that the following scenario is possible: there exists some network $\mathbb{N}(G, s, t, c)$ and a feasible flow f in $\mathbb{N}$ such that f is not optimal, but there is no augmenting path in $\mathbb{N}$ with respect to f. Fortunately, this is not the case.

Theorem 4.5. *Let f be a flow in network $\mathbb{N}(G, s, t, c)$. The following are equivalent:*

1. *f is a maximum flow*
2. *there exists an $[S, T]$-cut, for which $val(f) = c(S, T)$*
3. *there is no augmenting path in $\mathbb{N}(G, s, t, c)$ with respect to f*

Observe that the MFMC theorem is implied by Theorem 4.5. In order to prove Theorem 4.5 we need a new notion and a lemma.

Definition. Let P be a path, with one endpoint being s, in the graph we obtain from G by making its edges undirected. The edge set of P is divided into the disjoint sets $E^{fwd}(P)$ and $E^{bwd}(P)$, as before. We say that P is a partial augmenting path in $\mathbb{N}(G, s, t, c)$ if the followings hold: (i) for every $e \in E^{fwd}(P)$ we have $f(e) < c(e)$ and (ii) for every $e \in E^{bwd}(P)$ we have $f(e) > 0$.

Lemma 4.6. *Let f be a maximum flow in network $\mathbb{N}(G, s, t, c)$. Let S be the set of those vertices that can be reached from s by some partial augmenting path. Finally, let $T = V - S$. Then $[S, T]$ is a minimum cut with capacity $c(S, T) = val(f)$.*

Proof. If f is a maximum flow then, clearly, there is no augmenting path with respect to it in $\mathbb{N}(G, s, t, c)$. Hence, $t \notin S$, and therefore $T \neq \emptyset$. Moreover, if e is an edge from S to T then we must have $f(e) = c(e)$, and if e goes from T to S then we must have $f(e) = 0$ – otherwise S were larger. Hence, $val(S, T) = c(S, T)$. By Lemma 4.1 we have that $val(S, T) = val(f)$, hence, $c(S, T) = val(f)$. □

It is easy to see that the above imply Theorem 4.5, and therefore the MFMC theorem, Theorem 4.2.

4.1.1 An algorithm for finding a maximum flow

Below we present the Ford-Fulkerson algorithm for finding a maximum flow in a network $\mathbb{N}(G, s, t, c)$. It was published in 1955. Since then many algorithms were developed as the problem is of great practical and theoretical importance. The Ford-Fulkerson algorithm is certainly not the fastest among them, but it helps a lot in understanding other notions and results in graph theory and combinatorial optimization.

The Ford-Fulkerson algorithm

1. Initialization: let $f \equiv 0$ (this is always a feasible flow; if we have another feasible flow, that could also be the one we start with)

2. $S := \{s\}$, $B^{fwd} := \emptyset$, $B^{bwd} := \emptyset$

3. Let

$$B^{fwd} = \{x \in V - S : \exists y \in S \text{ such that } \overrightarrow{yx} \in E \text{ and } f(\overrightarrow{yx}) < c(\overrightarrow{yx})\}$$

and

$$B^{bwd} = \{x \in V - S : \exists y \in S \text{ such that } \overrightarrow{xy} \in E \text{ and } f(\overrightarrow{xy}) > 0\}$$

4. If $t \in (B^{fwd} \cup B^{bwd})$, then going backwards from t towards s we can find an augmenting path P. Augment the flow along P and continue with Step (2).

5. if $t \notin (B^{fwd} \cup B^{bwd})$, and $B^{fwd} \cup B^{bwd} \neq \emptyset$

   (a) Let $x \in B^{fwd} \cup B^{bwd}$ arbitrary

   (b) Let $S := S + x$, and continue with Step (3)

6. If $B^{fwd} \cup B^{bwd} = \emptyset$, then STOP – f is a maximum flow[a]

[a]In this case there is no augmenting path with respect to f.

Remark 4.7. If every capacity is an integer[1], then the Ford-Fulkerson algorithm stops in a finite number of steps, since we augment in every step by at least 1 unit. On the other hand, it is possible to construct a network having real number capacities so that the Ford-Fulkerson algorithm does not stop in a finite number of steps, moreover, in this case the values of the flows given by the algorithm do not even converge to the maximum flow value.

The Edmonds-Karp version: Always use shortest path (which contains the least number of edges) for augmentation. This can be done by using breadth-first search. This modified version of the algorithm will stop and find the optimal solution in $O(v(G)^4)$ steps.

There are faster methods to find the optimal solution than the Edmonds-Karp version. Many are not even based on augmenting paths, but on other ideas. At the moment the best general algorithm is by Orlin, that works in $O(|V(G)| \cdot |E(G)|)$ steps. Depending on the structure of the underlying graph (for example, G is dense or sparse) there are other methods which could be even better than Orlin's.

Below we give an example for the Ford-Fulkerson algorithm on a network. The list of consecutive figures show how the algorithm works.

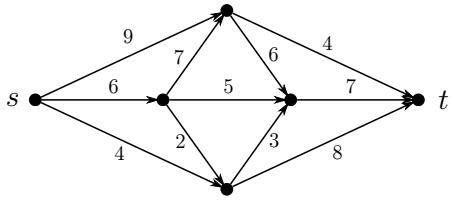

Figure 4.1: The network; the numbers on the edges are the corresponding capacities.

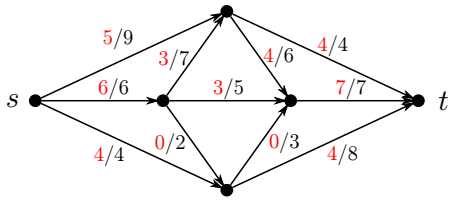

Figure 4.2: State of the network after some augmenting steps: the numbers on the left of the slashes indicate the amount of flow on the edge, while on the right we have the capacity.

[1]In case of rational numbers, one can make them integers by multiplying them with the least common multiple of the denominators.

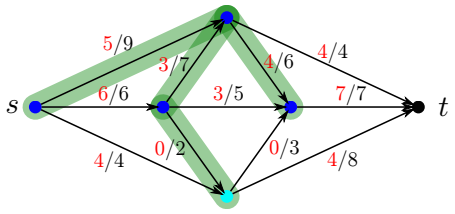

Figure 4.3: Looking for an augmenting path.

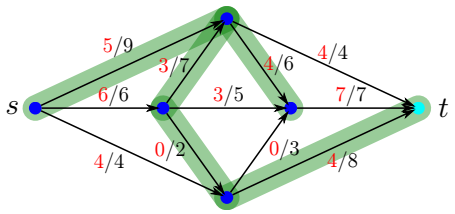

Figure 4.4: One can reach t from s via an augmenting path.

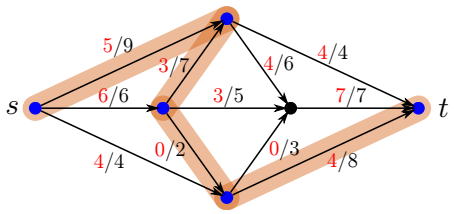

Figure 4.5: Since there is an augmenting path from s to t, indicated by the red edges, one can augment the flow.

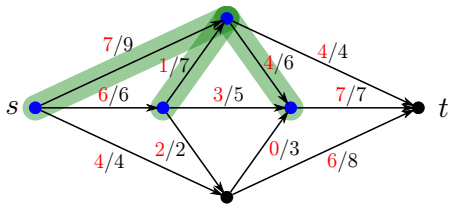

Figure 4.6: The vertices that can be reached via some partial augmenting path. Note that t and another vertex cannot be reached from s this way.

Definition. The network $\mathbb{N}(\overrightarrow{G}, s, t, c)$ is a *uniform network,* if the capacity of every edge of $\overrightarrow{G}$ is one unit, that is, $c \equiv 1$. Let $F \subseteq E(\overrightarrow{G})$, then $deg^+_F(x) = |\{e \in F \cap E^+(x)\}|$ and $deg^-_F(x) = |\{e \in F \cap E^-(x)\}|$.

Let $f : E(G) \longrightarrow \{0, 1\}$ be a function such that $f(e) = 1$ if and only if $e \in F$. Observe, that f satisfies the capacity constraints whenever $deg^+_F(x) = deg^-_F(x)$ for every $x \in V - \{s, t\}$. We can say more below.

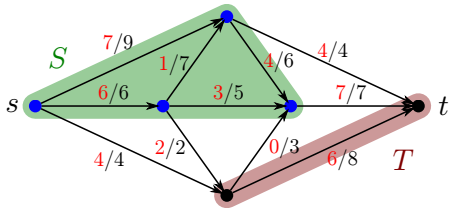

Figure 4.7: The $[S,T]$-cut shown must have minimum capacity since $c(S,T) = val(f)$.

Lemma 4.8. *Let $F \subset E$ be a subset of edges in G. Then the followings are equivalent.*

1. *F determines a flow*

2. *if $x \neq s,t$, then $deg_F^+(x) = deg_F^-(x)$*

3. *F can be decomposed as follows:*

$$F = \cup_i P_i \bigcup \cup_j Q_j \bigcup \cup_k C_k,$$

where the P_i, Q_j and C_k edge sets are disjoint, every P_i is a directed $s-t$ path, every Q_j is a directed $t-s$ path, and every C_k is the set of edges of a directed cycle.

Proof. First we note that (1) and (2) are equivalent by the definition of a flow. Next observe that if one deletes all the edges of some directed cycle of F, then the new edge set determines a flow if and only if the original F determined a flow, moreover, the values of the two flows are equal. This way we can get rid of all the directed cycle from F so that we do not change the value of the corresponding flow.

If in the remaining graph there is any edge left, then we can find either a directed $s-t$ path or a directed $t-s$ path as follows. The degree condition of (2) implies that whenever we enter a vertex v from a vertex w such that $v, w \notin \{s,t\}$, then there must be an edge that leaves v and another one that enters w. This enables us to extend the wv edge in both directions. Since there is no directed cycle left we must get a directed path having endpoints s and t by repeating this procedure. This path either goes from s to t, or from t to s. Next delete all the edges of the $s-t$ path we have just found and check if there is any edge left. If so, then the whole procedure can be repeated. We are able to do so because when deleting the edges of the path the indegrees and outdegrees are decreased by one for every inner vertex of the path, hence, (2) still holds. □

Let us call an edge set F *simple* if it has no directed cycles.

Lemma 4.9. *Let $\mathbb{N}(G, s, t, c)$ be a uniform network. Then $\mathbb{N}$ has such an optimal f flow, which is determined by a simple edge set F. Moreover, the optimal flow value is exactly the number of directed $s-t$ paths in the above decomposition of F.*

Proof. Using the Ford-Fulkerson algorithm, we see that there is an optimal flow f such that either $f(e) = 0$ or $f(e) = 1$ for every edge e, since $\mathbb{N}$ is a uniform network. The simple edge set is determined by those edges e for which $f(e) = 1$. The second statement of the lemma follows easily from the definition of the value of a flow. □

Definition. We call $L \subseteq E(G)$ a cutting edge set in $\overrightarrow{G}$, if there is no directed $s-t$ path in $\overrightarrow{G} - L$.

Lemma 4.10. *The edge set $\overrightarrow{ST}$ of every $[S,T]$-cut (that is, the edges that point from S to T) is a cutting edge set. Furthermore, every cutting edge set L contains a subset which is exactly the set of edges of $\overrightarrow{ST}$ in some $[S,T]$-cut.*

Proof. It is clear that if one deletes the edges of $\overrightarrow{ST}$ for some $[S,T]$-cut, then there remains no directed $s-t$ path. Assume now that L is a cutting edge set. Let S denote the set of vertices that can be reached from s on a directed path in $G - L$, and let $T = V - S$. Clearly, L must contain every edge that points from S towards T, otherwise S were larger and T were smaller, hence, $\overrightarrow{ST} \subseteq L$. This proves the other direction of the lemma. □

4.2 Applications

In this section we consider applications of network flow theory. Some are theoretical results, others are important for "real life" problems.

4.2.1 The Menger theorems

Lemma 4.9 and Lemma 4.10 can be used to prove the famous theorems of Karl Menger. Originally, he showed only the first among the four results below, however, the second, third and fourth ones each follow from the first one quite easily.[2] We will discuss them as corollaries of Theorem 4.11.

Theorem 4.11 (Menger)**.** *Let $\overrightarrow{G}$ be a directed graph, and $s, t \in V(\overrightarrow{G})$ such that $s \neq t$. Then*

$$\max\{k : P_1, \ldots, P_k \text{ are edge} - \text{disjoint } \overrightarrow{s-t} \text{ path}\} =$$

$$\min\{|L| : L \subseteq E(\overrightarrow{G}),\ \overrightarrow{G} - L \text{ has non } \overrightarrow{s-t} \text{ path}\}.$$

[2] We remark that Menger's theorem is implied by a result of Dénes König on maximum matchings and minimum covers in bipartite graphs. We will consider König's theorem later.

Proof: It is clear that if one deletes less edges from G then the number of edge-disjoint $s-t$ paths, then there must remain an intact $s-t$ path, therefore the right hand side is always at least as large as the left hand side. In order to prove equality, we define a uniform network $\mathbb{N}(G, s, t, 1)$. In $\mathbb{N}$ there is a set of edge-disjoint $s-t$ paths corresponding to every integral flow, moreover, as we have seen already, the maximum flow value equals the maximum number of edge-disjoint $s-t$ paths. By the MFMC theorem, we have that the capacity of a minimum $[S,T]$-cut equals the value of this maximum flow. As $c(S,T) = |\overrightarrow{ST}|$ in the uniform network $\mathbb{N}$, using Lemma 4.10 we get what was desired. □

Corollary 4.12. *Let G be an undirected graph and $s, t \in V(G)$ such that $s \neq t$. Then*

$$\max\{k : P_1, \ldots, P_k \text{ edge} - \text{disjoint } s-t \text{ paths}\} =$$
$$\min\{|L| : L \subseteq E(G),\ G - L \text{ has no } s-t \text{ path}\}.$$

Proof. We construct a directed graph $\mathcal{G}$ from G as follows. The two graphs have the same vertex set. Whenever $uv \in E(G)$ for some $u, v \in V$, then we have both directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$ in $\mathcal{G}$. Applying Theorem 4.11 for $\mathcal{G}$ easily finishes the proof. □

Corollary 4.13. *Let $\overrightarrow{G}$ be a directed graph and $s, t \in V(G)$ such that $s \neq t$. Assume that $\overrightarrow{st} \notin E(G)$. Then*

$$\max\{k : P_1, \ldots, P_k \text{ vertex} - \text{disjoint } \overrightarrow{s-t} \text{ paths}\} =$$
$$\min\{|U| : U \subseteq V(G) - \{s,t\},\ G - U \text{ has no } \overrightarrow{s-t} \text{ path}\}.$$

Proof. We construct a directed graph $\mathcal{G}$ from G again. For every vertex u where $u \neq s, t$ in $\mathcal{G}$ we will have two vertices u_{in} and u_{out}, and a directed edge that goes from u_{in} to u_{out}. Whenever an edge enters u in G, the corresponding edge will enter u_{in} in $\mathcal{G}$, and if an edge leaves u in G then the corresponding edge will leave u_{out} in $\mathcal{G}$. Notice that edge-disjoint paths in $\mathcal{G}$ correspond to vertex-disjoint paths in G. Hence, applying Theorem 4.11 finishes the proof. □

Corollary 4.14. *Let G be an undirected graph and $s, t \in V(G)$ such that $st \notin E(G)$. Then*

$$\max\{k : P_1, \ldots, P_k \text{ vertex} - \text{disjoint } s-t \text{ paths}\} =$$
$$\min\{|U| : U \subseteq V(G) - \{s,t\},\ G - U \text{ has no } s-t \text{ path}\}.$$

Proof. The proof consists of applying the previous two corollaries as follows. First construct a directed graph $\mathcal{G}_1$ from G as in the proof of Corollary 4.12. Next construct the directed graph $\mathcal{G}_2$ from $\mathcal{G}_1$ as in the proof of Corollary 4.13. Finally apply Theorem 4.11. The edge-disjoint disjoint paths in $\mathcal{G}_2$ translate to vertex-disjoint directed paths in $\mathcal{G}_1$, which in turn translate to vertex-disjoint undirected paths in G. □

Definition. Let k be a natural number. We say that graph G is k-edge-connected, if G remains connected even after removing up to $k-1$ edges from it arbitrarily. Similarly, we say that graph G is k-vertex-connected or briefly k-connected if no matter how one removes up to $k-1$ vertices from it, what is left remains connected.

Theorem 4.15. *Let G be a graph and k be a natural number. We have the following.*

(1) G is k-edge-connected if and only if there exists k edge-disjoint $x-y$ paths for every $x, y \in V(G)$ (here $x \neq y$).

(2) G is k-connected if and only if $v(G) > k+1$ and for every $x, y \in V(G)$, $x \neq y$ there exists k pairwise vertex-disjoint $x-y$ paths in G.

Proof. The proof of (1) easily follows from Corollary 4.12, while (2) follows from Corollary 4.14. □

Definition. Let G be a connected graph. We define $\kappa_e(G)$ to be the largest k for which G is k-edge-connected. Similarly, we define $\kappa(G)$ to be the largest k for which G is k-connected. If G is disconnected, then we let $\kappa_e(G) = \kappa(G) = 0$.

Remark 4.16. We have $\kappa_e(G) \geq \kappa(G)$ for every graph G.

4.2.2 The Project Selection problem

Assume that we are given k projects, $P_1, \ldots, P_k$, and l equipments $Q_1, \ldots, Q_l$. In order to perform a project, we may need certain equipments. The revenue of project P_i is denoted by $r(P_i)$, the cost of equipment Q_j is denoted by $c(Q_j)$ for every $1 \leq i \leq k$ and $1 \leq j \leq l$.

As we noted above, each project may require certain equipments, moreover, equipments can be shared by several projects. Our task is to determine which projects and corresponding equipments should be selected and purchased, respectively, in order to maximize profit. Observe that if projects didn't share equipments, this task would be very easy to solve.

Denote $\mathcal{P}$ the set of selected project, and $\mathcal{Q}$ the set of equipments required by the projects of $\mathcal{P}$. Then our goal is to find an $\mathcal{P}$ which maximizes the following expression:

$$\sum_{P \in \mathcal{P}} r(P) - \sum_{Q \in \mathcal{Q}} c(Q).$$

Clearly, we have

$$\sum_{P \in \mathcal{P}} r(P) - \sum_{Q \in \mathcal{Q}} c(Q) = \sum_{i \geq 1} r(P_i) - \sum_{P \in \mathcal{P}'} r(P) - \sum_{Q \in \mathcal{Q}} c(Q),$$

where $\mathcal{P}' = \{P_1, \ldots, P_k\} - \mathcal{P}$, i.e., the complement of $\mathcal{P}$. Observe, that the first term does not depend on $\mathcal{P}'$ and $\mathcal{Q}$, so we can formulate an equivalent minimization problem: find

$$\min_{\mathcal{P}'}\{\sum_{P\in\mathcal{P}'} r(P) - \sum_{Q\in\mathcal{Q}} c(Q)\}.$$

Next we construct a network $\mathbb{N}(G, s, t, c)$ so that the minimum cut in the network corresponds to the optimal solution of the above minimization problem. More precisely, if $[S, T]$ is the minimum cut, then S contains those projects that one has to select in order to maximize profit. Since a minimum cut can be found by first solving the maximum flow problem in the network, and then determining those vertices that can be reached from s by some partial augmenting path, the project selection problem can be solved efficiently. This is not an obvious fact! A naive approach could be to check all possible subsets of the projects in order to find the optimal solution. This however would take exponentially many steps.

The definition of $\mathbb{N}$ is as follows. The vertex set consists of a source s, a sink t, the projects and the equipments. There is a directed edge from s to project P_i having capacity $r(P_i)$ for every i. Similarly, there is a directed edge from equipment Q_j to t having capacity $c(Q_j)$ for every j. Finally, whenever equipment Q_j is required for project P_i we include a directed edge from P_i to Q_j having infinite capacity. Note that when in an optimal $[S, T]$-cut a project $P_i \in S$, then all the equipments required for this project must also be in S by the infinite capacity of the P_iQ_j edges. This also implies that in a minimum $[S, T]$-cut the capacity of the cut is given by the sum of capacities of those edges that go from s to all the projects in $\mathcal{P}'$ plus the sum of capacities of those edges that go from equipments in $\mathcal{Q}$ to t. As the capacity of such a cut is precisely what we have to minimize, we proved that the optimal subset of projects can be found efficiently using network flow theory.

4.2.3 The Image Segmentation problem

In the image segmentation problem we are given a (usually rectangular grid) graph $H = (V, E)$, every vertex in V represents a pixel. Each pixel v can either be assigned a foreground value f_v or a background value b_v. There is a penalty of p_{uv} if pixels u, v are adjacent, i.e., $uv \in E$, and have different assignments. The problem is to assign all pixels to foreground or background such that the sum of their values minus the penalties is maximum.

More formally, let $\mathcal{F}$ be the set of pixels assigned to foreground and $\mathcal{B}$ be the set of points assigned to background, here of course $V = \mathcal{F} \cup \mathcal{B}$. Let $\mathcal{S}$ denote the set of those edges of H that connect a vertex in $\mathcal{F}$ to a vertex in $\mathcal{B}$. The problem can be formulated as follows:

$$\max\{\sum_{u\in\mathcal{F}} f_u + \sum_{v\in\mathcal{B}} b_v - \sum_{uv\in\mathcal{S}} p_{uv}\}.$$

Similarly to the project selection problem, this maximization problem can be formulated as a minimization problem instead, that is,

$$\min\{\sum_{u\in\mathcal{B}} f_u + \sum_{v\in\mathcal{F}} b_v + \sum_{uv\in\mathcal{S}} p_{uv}\}$$

(notice that the role of $\mathcal{F}$ and $\mathcal{B}$ are switched in the first two terms of the above sum).

The solution of this minimization problem can be obtained similarly to the project selection problem. That is, we construct a network $\mathbb{N}(G, s, t, c)$ so that the minimum cut in the network corresponds to the optimal solution of the minimization problem.

The set of vertices of $\mathbb{N}$ includes V, a source s and a sink t. There is a directed edge from the source to all vertices $v \in V$ having capacity f_v, and there is an edge from every vertex $v \in V$ to the sink having capacity b_v. For every edge $uv \in H$ we include two directed edges: one goes from u to v, the other from v to u. The capacity of both edges is p_{uv}. The minimum $[S, T]$-cut then represents the optimal assignment of pixels assigned to the foreground $\mathcal{F}$ or the background $\mathcal{B}$. Again, one can find the minimum $[S, T]$-cut efficiently by solving the maximum flow problem on $\mathbb{N}$.

4.2.4 Finding a maximum matching in bipartite graphs

We consider the matching problem in greater detail in another chapter, hence, we do not repeat definitions here. Our goal is to give the description of an effective method for finding a maximum cardinality matching in a bipartite graph $H = H(A, B, E)$.

First we construct a network $\mathbb{N}(G, s, t, c)$. The vertex set $V(G)$ includes $A \cup B$ together with the new vertices s and t. The network is uniform, so $c \equiv 1$. The edge set of G is determined as follows. We have every edge between s and A, all oriented towards A. We also have every edge between B and t, all pointing from B towards t. Finally, every edge of H is included, these edges are oriented towards their endpoint in B.

Find the maximum flow f in $\mathbb{N}$ using the Ford-Fulkerson algorithm so that f is either 0 or 1 at every edge. By Lemma 4.9 the number of directed $s - t$ paths is $val(f)$. These paths have no common vertices in $A \cup B$, hence, the intersection of the simple edge set F determined by f and $E(H)$ must be a matching of H.

If M is a maximum matching in H, then it is easy to turn it into a feasible flow. Repeat the following for every $xy \in M$, where $x \in A$ and $y \in B$: set $f(sx) = f(xy) = f(yt) = 1$. The f function we obtain this way must be a feasible flow by the definition of matchings.

Putting these together, the Ford-Fulkerson algorithm can be used to find a maximum matching in a bipartite graph. We remark that one can also use the MFMC theorem in order to prove the König-Hall theorem on the cardinality of maximum matchings in bipartite graphs.

Chapter 5

Algorithms

In this chapter we consider a few basic algorithmic problems for graphs. While the problems discussed here are among the most important ones, our goal is not to give a complete landscape, rather to let the reader to catch a glimpse of the area.

5.1 Graph searching

Given a graph $G = (V, E)$ it is a natural question to ask if two vertices $u, v \in V$ belong to the same component, and if so, find a path that connects u and v. There are two basic strategies for solving this problem, breadth-first search and depth-first search. We will discuss both. We will also consider a closely related problem: given a graph with non-negative edge weights and a source vertex u find the shortest path from the source to every other vertex of G.

5.1.1 Breadth-first search

Let $G = (V, E)$ be a graph, and $u \in V$ be a vertex. In every step the algorithm has a set S the elements of which are searched, and another set R containing those vertices that are reached, but not searched. The set R is a First-In-First-Out (FIFO) list, that is, we add elements to the back of the list, and remove only the first element.

When R becomes the emptyset, the algorithm stops. For every $v \in S$ we also compute the length of the shortest path denoted by $d(u, v)$ between u and v. The details are as follows.

Breadth-first search

Input: a graph $G = (V, E)$ and a starting vertex $u \in V$.

Step 1. Initialization: let $R = u$, $S = \emptyset$ and set $d(u, u) = 0$

Step 2. If $R \neq \emptyset$

Step 2.a Let v be the first vertex of R

Step 2.b For every $w \in N(v) - (S \cup R)$ we add w to the back of R, and let $d(u, w) = d(u, v) + 1$

Step 2.c Remove v from the list R, and place it into S

Step 2.d Continue with Step 2.

Step 3. STOP

Next we prove that the above algorithm correctly calculates the distances from the start vertex u.

Theorem 5.1. *When the Breadth-first search algorithm stops, for every vertex v in the connected component of u we have that $d(u, v)$ equals the length of the shortest path from u to v.*

Proof. We prove the theorem by induction on the distance from u. Clearly, it holds for $v = u$, since $d(u, u) = 0$. It also holds for every $v \in N(u)$, as for these vertices the algorithm sets $d(u, v) = 1$.

Now assume that $d(u, v)$ has the correct value for every v which is at distance at most k, for some $k \geq 1$. Let w be a vertex which is at distance $k + 1$ from u. Let $u - v_1 - v_2 - \ldots - v_k - w$ be a path of shortest length from u to w. Then for $d(u, v_i) = i$ for every $1 \leq i \leq k$ by induction. We also have $v_1, v_2, \ldots, v_k \in S$, moreover, before we placed v_k into S, we added w to R, and set $d(u, w) = k + 1$. Hence, $d(u, w)$ also has the correct value. □

Remark 5.2. During the execution of the algorithm we can construct a rooted tree T, the so-called breadth-first search or BFS tree, as follows. The root of T is the start vertex u. We include an edge vw in T, when we add w to R and replace v from R to S.

Remark 5.3. It is very easy to modify the algorithm in order to make it work for directed graphs. The only task is to change Step 2.b a little bit: in the directed version we add w to the back of R if it belongs to $N^+(v) - (S \cup R)$, where $N^+(v)$ denotes the out-neighborhood of v.

5.1.2 Depth-first search

Depth-first search and breadth-first search has some common features. For example we have two sets, R and S, that have similar roles. However, this time R will be a Last-In-First-Out (LIFO) list, which changes the behaviour of the algorithm.

Depth-first search

Input: a graph $G = (V, E)$ and a starting vertex $u \in V$.

Step 1. Initialization: let $R = u$, $S = \emptyset$

Step 2. If $R \neq \emptyset$

Step 2.a Let v be the last vertex of R

Step 2.b If $N(v) - (S \cup R) = \emptyset$, then remove v from R and add it to S

Step 2.c If $N(v) - (S \cup R) \neq \emptyset$, then pick a $w \in N(v) - (S \cup R)$ and add w to the back of R

Step 2.d Continue with Step 2

Step 3. STOP

The depth-first search algorithm finds every vertex in the component of the start vertex. One can prove this very similarly to the proof of Theorem 5.1, we leave the details for the reader.

Remark 5.4. During the execution of the algorithm we can construct a rooted tree T, the so-called depth-first search or DFS tree, as follows. The root is the start vertex u. Whenever we place w into R (that is, when we choose w from $N(v) - (S \cup R)$), we include the edge vw in T.

Remark 5.5. Depth-first search can also be adjusted to work in a directed graph. Analogously to the breadth-first search algorithm, the only modification is that we select a vertex in $N^+(v) - (S \cup R)$.

5.1.3 Applications of graph search algorithms

Let us give a few simple applications of the above discussed algorithms. First of all, let $G = (V, E)$ be an undirected graph, and $u \in V$ be a vertex of it. Run the BFS or the DFS algorithm. At the end the set S will contain precisely the vertices of the connected component of u. In particular, if $S = V$, then G is a connected graph. If G is not connected, then for finding another connected component of it just take a vertex from the complement of the already found component. One can use this method for exploring every component of a graph.

In case $G = (V, E)$ is a directed graph, we may ask if it is strongly connected, that is, if there is a directed path from any vertex to any other vertex of G. One possibility is the following. For every $u \in V$ run the directed version of the BFS (or DFS) algorithm with u as a start vertex. If in every run we can reach every vertex from the start vertex, then clearly G is strongly connected.

But there is a much faster method! First, take an arbitrary vertex $u \in V$ as start vertex, and run the directed version of one of the search algorithms. Next reverse the orientation of every edge in G. Run the directed version of a search algorithm from the same start vertex u. It is not hard to see that if in both cases the set S equals V, then G is strongly connected, otherwise not.

Let us mention a final application, without detailed description. Using the DFS algorithm for an undirected and 2-edge-connected graph G, one can orient its edges so that the resulting directed graph is strongly connected. This goes as follows. Build the DFS tree, and orient every edge in the tree from the parent towards its child. For the rest of the edges the orientation is based upon the order in which

the vertices were placed into R. If v preceded w and $vw \in E(G)$, then we orient the edge from w towards v. Note that if G is not 2-edge-connected, then there is no such orientation.

5.1.4 Finding shortest path from a single source in a weighted graph

The problem we discuss here is an everyday problem. Say that you drive your car in an unknown town, and want to find the shortest route from your current position to some restaurant other people recommended. A mobile phone application can tell the answer, but how? We will consider the mathematics behind this question. Note that if every street block were of the same length, then the BFS algorithm could find us the optimal route. However, usually this is not the case, we need a more sophisticated, although somewhat similar method.

Assume that we are given a graph $G = (V, E)$ in which every edge vw has a weight $l(v, w) \geq 0$. We extend l to paths in the natural way. If P is a path, then $l(P) = \sum_{e \in E(P)} l(e)$. One more assumption for convenience: we set $l(v, w) = \infty$ for every $vw \notin E$.

The following algorithm for finding the shortest paths from a source vertex $u \in G$ to every other vertex was discovered by Dijkstra.

Dijkstra's algorithm

Input: an edge-weighted graph $G = (V, E)$ and a starting vertex $u \in V$.

Step 1. Initialization: let $S = u$, $t(u) = 0$ and for every $v \in V$ let $t(v) = l(u, v)$

Step 2. If $S \neq V$

Step 2.a Choose a vertex $v \in V - S$ such that $t(v) = \min\{t(w) : w \in V - S\}$

Step 2.b If $t(v) = \infty$, then continue with Step 4

Step 2.c Let $S = S + v$

Step 2.d Let $t(w) = \min\{t(w), t(v) + l(v, w)\}$ for every $w \in N(v) \cap (V - S)$

Step 3. Let $l(u, v) = t(v)$ for every $v \in V$

Step 4. STOP

Let us give some explanation for the algorithm. First, the $t(v)$ values are tentative distances from u to v. As the algorithm proceeds, in every step we determine the length of the shortest path to some vertex v from u, where v is the closest to u among vertices in $V - S$. At the end the distances are given by the $l(u, \cdot)$ values.

There are two reasons why the algorithm may stop: either $S = V$, or the algorithm recognizes that the graph is disconnected.

Theorem 5.6 (Dijkstra (1959))**.** *Given an edge-weighted graph G and a starting vertex u, Dijkstra's algorithm correctly computes the $l(u, v)$ values for every $v \in V$.*

Proof. We will show that at every point in time $t(v) = l(u, v)$ for every $v \in S$, moreover, for $w \notin S$ the value of $t(w)$ is the length of the shortest path from u to w having internal vertices only from S. We do this by induction on the cardinality of S. When $|S| = 1$, then clearly, $t(u) = l(u, u) = 0$.

Assume now that the theorem holds for $|S| \leq k$ for some natural number $k \geq 1$. First we show that if a vertex $v \in V - S$ is chosen in Step 2.a, then $t(u)$ equals the distance from u to v.

Assume that this is not the case, there is a vertex v, chosen at Step 2.a such that $t(v)$ is larger, than the distance of u and v. Consider the shortest path P from u to v. By assumption $l(P) < t(v)$. But then there must be a vertex on P which does not belong to S by the induction hypothesis. Let w be the first vertex from $V - S$ on P. Note that the value of $t(w)$ is the length of the shortest path from u using only vertices from S, hence, $t(w) \geq t(v)$. From w there is non-negative length portion of P, so $l(P) \geq t(v)$, contradicting to the assumption.

Next we have to show that for every $w \in V - S$ the value of $t(w)$ is the length of the shortest path from u when we are only allowed to use internal vertices from S. This holds for $|S| \leq k$. After increasing S by a vertex v in Step 2.c we may change the value of $t(w)$. But we only change it when $t(v) + l(v, w) < t(w)$. So before changing $t(w)$ was the smallest distance using vertices of $S - v$ by the induction hypothesis, and then it has become the smallest when using vertices from $S - v$ and v. This finishes the proof. □

Remark 5.7. It is vey easy to obtain Dijkstra's algorithm for directed graphs: the only change is that in Step 2.d use $N^+(v)$ instead of $N(v)$.

Finally we mention that there is an algorithm, that can work with negative edge weights as well, the Bellman-Ford algorithm. However, when there is a cycle with negative total weight, there is no minimum weight path for every pair of vertices, as one can wind around the cycle arbitrary number of times, always decreasing the total weight.

5.2 The minimum spanning tree problem

In many real life problems one faces with the question of connecting n points with "wires" of some length such that a sign can travel between any two points through wires, and the total wire length is minimal. In one of the earliest appearances of this problem, Otakar Boruvka, an engineer formulated and solved this problem when devising the electrical network system in some portion of Czechslovakia. Boruvka published his result in 1926. Since then many solutions were found, at present the fastest is due to Bernard Chazelle. Chazelle's algorithm is very sophisticated and complex, theoretically is very important, but perhaps not the one to be presented

here. Instead we will discuss the algorithm by Joseph Bernard Kruskal below. The latter has a very clear formulation, and can be analyzed easily.

It is clear from the question that an optimal solution must be a tree on n vertices with minimum total wire length. First, it must contain every vertex, and second, if it had a cycle, then we could make the total length shorter by eliminating the longest edge on the cycle, still keeping the network connected.

Definition. Let $G = (V, E)$ be a simple graph such that for every $e \in E$ we have an edge weight $w(e)$. We extend the edge weights to subgraphs of G in the natural way. If $H \subseteq G$ is any subgraph, then

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

In the minimum spanning tree problem the goal is to find a spanning tree T of minimum total weight $w(T)$.

Kruskal's algorithm (1956)

Assume, that the edges of G are sorted according to their weight, ties are broken arbitrarily: $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$, where $E = \{e_1, e_2, \ldots, e_m\}$.

Step 1. Initialization: let T be the empty graph, and $i = 1$

Step 2. If $T + e_i$ is acyclic, then let $T = T + e_i$

Step 3. If $i < m$, then let $i = i + 1$, and continue with Step 2.

Step 4. Output: minimum spanning tree T

We are going to prove that the above algorithm finds the optimal solution.

Theorem 5.8. *Given a connected edge weighted graph $G = (V, E)$, Kruskal's algorithm finds a minimum weight spanning tree of it.*

Proof. First we have to show that the output T is indeed a spanning tree of G. It is clear from the way we build T that it is acyclic. Assume that T is not connected, it has at least two components, C_1 and C_2. Since G is connected, there is at least one edge e_k of G that goes between C_1 and C_2. But then we must have added e_k to T as $T + e_k$ is acyclic! This shows that T is a spanning tree of G.

Now let us assume that G has another spanning tree T' for which $w(T') < w(T)$. Denote e_l the first edge according to the ordering of edges that belongs to T' but not to T. Adding e_l to T we create a cycle C. Since T' is a tree, we must have at least one edge $e_t \in C$ which belongs to T but not to T'. Moreover, for every edge e_i of C we have $w(e_i) \leq w(e_l)$, since otherwise we would have added e_l to T. Let

T_1 be the tree we obtain from T by adding e_l to it and deleting e_i from it. Clearly, $w(T) \leq w(T_1)$, and T_1 has one more common edges with T' than T.

If T' has another edge e_s that does not belong to T, then we repeat the above procedure: add e_s to T_1, and delete an edge e_q that does not belong to T'. Call the new tree we obtain this way. Then T_2 is "closer" to T', than T_1 was, it has one more common edges with it. We also have that $w(T_2) \geq w(T_1)$.

Repeating the above procedure at most $n-1$ times we get a sequence of trees $T, T_1, T_2, \ldots, T_p, T'$ such that $w(T) \leq w(T_1)$, in general $w(T_j) \leq w(T_{j+1})$, and finally, $w(T_p) \leq w(T')$. This shows that T must be a minimum weight spanning tree. □

Let us have a remark on the time complexity of Kruskal's algorithm. If the edges are given as an ordered sequence according to the weights, Kruskal's algorithm runs in $O(m)$ steps, and is in fact a greedy algorithm. Sorting the edges requires $\Theta(m \log m)$ steps, this is the most time demanding part of the algorithm.

And a final remark: if one wants to find any spanning tree of a connected undirected graph, then can use the BFS, the DFS algorithms, as well as Kruskal's algorithm. For the latter one can mimic if the edges of the graph had some weights, for example, every edge can have unit weight.

Chapter 6

Matchings

6.1 Definitions

Definition. A set of edges $M \subseteq E(G)$ in a multigraph G is called a *matching*, if no two edges of M have a common endpoint, and there are no loops in M.

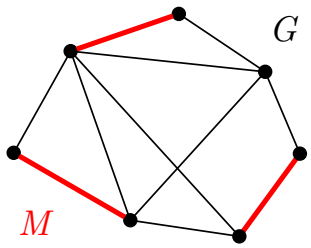

Figure 6.1: A matching

For any $S \subseteq E(G)$, the set of endpoints of edges in S is denoted by $V(S)$, i.e.

$$V(S) = \{v \in V(G) : v \text{ is incident to some edge of } S\}.$$

We note that $M \subseteq E(G)$ is a matching in G if and only if $|V(M)| = 2|M|$.

We review some terminologies. For a matching M, we say that M *covers* (precisely) the vertices of $V(M)$. We also call the vertices in $V(M)$ the *matched* vertices of G, and the vertices in $V(G) \setminus V(M)$ are the unmatched vertices. Moreover, if $uv \in M$, then we say that M *matches* u to v.

Definition. A *maximum matching* (or maximum-cardinality matching) is a matching that contains the largest possible number of edges. The size of a maximum matching of G is denoted by $\nu(G)$, i.e.

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

(Recall that $|M|$, the size of M, is the number of edges in M by definition.) The parameter $\nu(G)$ is sometimes called the matching number of G.

A *perfect matching* in G is a matching that covers all vertices of G.

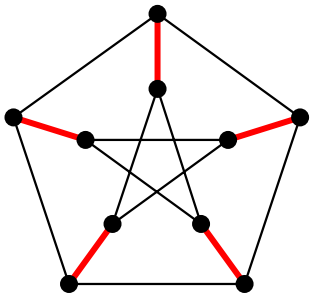

Figure 6.2: A perfect matching in the Petersen graph

Of course, not all graphs have a perfect matching. For example, graphs with an odd number of vertices do not have a perfect matching, because every matching covers an even number of vertices. Among other things, the next sections present efficient algorithms and powerful theorems to decide whether a graph has a perfect matching.

It is evident that

$$\nu(G) \leq \frac{|V(G)|}{2}, \tag{6.1}$$

as

$$2|M| = |V(M)| \leq |V(G)|,$$

and so

$$|M| \leq \frac{|V(G)|}{2}$$

for any matching M of G. Equality occurs in (6.1) if and only if G has a perfect matching.

6.2 Matchings in bipartite graphs

Recall the definition of bipartite graphs from Chapter 1. This section deals with the problem of determining the parameter $\nu(G)$ of bipartite graphs G. Throughout this section $G(A, B)$ will denote a bipartite graph G with bipartition $V = A \cup B$.

It is clear that for any matching M of $G(A, B)$,

$$|A \cap V(M)| = |B \cap V(M)| = |M|,$$

since every edge of M matches a vertex of A with a vertex of B. This means that the size of A (or B) is an upper bound for the size of M. Since this is true for maximum matchings too, we obtained the following.

Observation 6.1. *For any bipartite graph $G(A, B)$ we have that*

$$\nu(G) \leq |A|.$$

Now we are going to characterize bipartite graphs for which $\nu(G) = |A|$ occurs, i.e. in which there exists a matching that covers the partite set A. If $\nu(G) = |A|$, then there is always a quick way to prove it in theory, it is enough to present a

matching in G covering A. But how can we argue in the case when $\nu(G) < |A|$, i.e. how can we find a short proof of the *non-existence* of such a matching? It turns out that there exists a universal argument that can be *always* applied when $\nu(G) < |A|$. Now we are going into the details.

For a subset $X \subseteq A$, the *neighborhood* $N(X)$ of X is defined as

$$N(X) := \{v \in B : v \text{ is adjacent to some vertex of } X\},$$

in other words, $N(X)$ is the union of the neighborhoods $N(x)$, where $x \in X$.

Lemma 6.2. *Given a bipartite graph $G(A, B)$. If $|N(X)| < |X|$ for some $X \subseteq A$, then G does not contain a matching covering A.*

Proof. The setting is illustrated in Figure 6.3. Pick an arbitrary matching M in G. By the definition of $N(X)$, every vertex of X can be matched to a vertex of $N(X)$ only. This means that at least $|X| - |N(X)| > 0$ vertices of X (and so of A) are unmatched, otherwise there would exist two edges in M, starting from X, which have common endpoint in $N(X)$, by the pigeonhole principle. □

In the above proof, we obtained a lower bound on the number of unmatched vertices in A, i.e. an upper bound on $\nu(G)$.

Observation 6.3. *Assume that $|N(X)| < |X|$ for some $X \subseteq A$, in a bipartite graph $G(A, B)$.*

(a) Then any matching of G leaves at least $|X| - |N(X)|$ unmatched vertices in A.

(b) This implies that

$$\nu(G) \leq |A| - (|X| - |N(X)|).$$

We call a set $X \subseteq A$ a *Kőnig set*, if $|N(X)| < |X|$ holds. (It is named after Dénes Kőnig.)

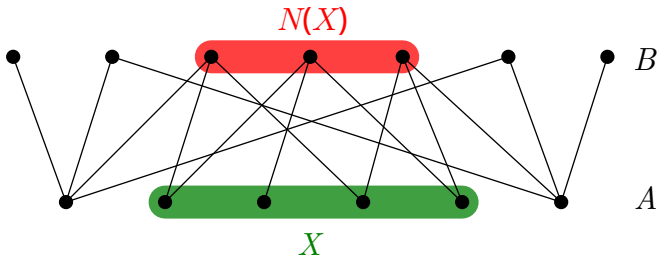

Figure 6.3: A Kőnig set X

We saw in Lemma 6.2 that if a Kőnig set exists in G, then G does not have a matching covering A. Actually, the converse is also true, which means that the non-existence of a matching covering A can be always proven by presenting a Kőnig set.

Theorem 6.4 (Marriage theorem, or Kőnig–Hall-theorem)**.** *The bipartite graph $G(A, B)$ contains a matching covering A if and only if there is no Kőnig set in G, i.e. if $|N(X)| \geq |X|$ for all $X \subseteq A$.*

Lemma 6.2 is the easy direction of this theorem; the other direction is more difficult. We will give a full proof later on page 59.

Now we can also characterize bipartite graphs that contain a *perfect* matching.

Theorem 6.5 (Kőnig–Frobenius)**.** *The bipartite graph $G(A, B)$ contains a perfect matching if and only if $|A| = |B|$ and there is no Kőnig set in G (i.e. $|N(X)| \geq |X|$ for all $X \subseteq A$).*

Proof. Bipartite graphs with $|A| \neq |B|$ clearly do not have a perfect matching, because the number of matched vertices in A is the same as in B, for any matching. For bipartite graphs with $|A| = |B|$, perfect matchings are exactly the matchings covering A. and so Theorem 6.4 can be applied. □

The following notion plays an important role in the theory of matchings, and it will be useful in the non-bipartite case, too.

Definition. Given a (not necessarily bipartite) graph G, and a matching M in G. We say that a path

$$P : (v_0, e_1, v_1, e_2, v_2, e_3, v_3, \ldots, v_{2k}, e_{2k+1}, v_{2k+1})$$

in G is an *augmenting path* with respect to M, if P satisfies the following conditions:

(i) $v_0 \notin V(M)$,

(ii) $e_1, e_3, e_5, \ldots, e_{2k+1} \notin M$,

(iii) $e_2, e_4, e_6, \ldots, e_{2k} \in M$, and

(iv) $v_{2k+1} \notin V(M)$,

In words, the two end vertices of P are unmatched, and the edges belong alternately to M and not to M. See the upper path in Figure 6.4.

We say that P is a *partial augmenting path* (with respect to M), or PAP for short, if it satisfies conditions (i)-(iii). The length of a PAP is allowed to be even (while the length of an augmenting path is always odd). The attribute 'partial' in the name reflects that a PAP might be extendable to an augmenting path. We say that $v_0, v_2, v_4, \ldots$ are the *outer* vertices, and $v_1, v_3, v_5, \ldots$ are the *inner* vertices of the partial augmenting path P.

The following statement explains the attribute 'augmenting' in the above definition.

Lemma 6.6. *If there exists an augmenting path with respect to the matching M in a (not necessarily bipartite) graph G, then M is not a maximum matching in G.*

Proof. Let $P : (v_0, e_1, v_1, e_2, v_2, e_3, v_3, \ldots, v_{2k}, e_{2k+1}, v_{2k+1})$ be and an augmenting path. It is easy to check that

$$M' := M \setminus \{e_2, e_4, e_6, \ldots, e_{2k}\} \cup \{e_1, e_3, e_5, \ldots, e_{2k+1}\}$$

(c.f. Figure 6.4) is a matching of G that has one more edges than M. □

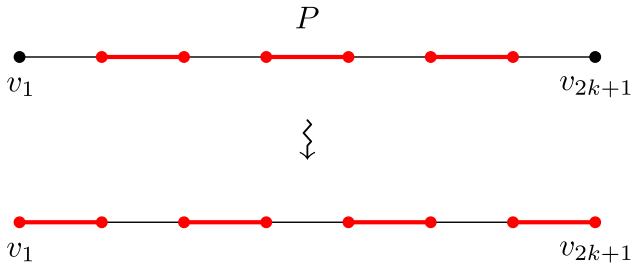

Figure 6.4: Illustration of the proof of Lemma 6.6

We will see in the next section (in Theorem 6.18) that it is also true that if no augmenting path exists, then M is a maximum matching.

Now we are in a position to present an algorithm which determines the parameter $\nu(G)$ of bipartite graphs in polynomial time. In fact, we want to find a maximum matching M, together with a proof of its maximality. Then $\nu(G) = |M|$ is obtained.

Goal 6.7. Our goal is to construct a polynomial-time algorithm that finds an augmenting path in $G(A, B)$ with respect to M, if such a path exists.

- So the INPUTs of the algorithm are a bipartite graph $G(A, B)$ and a matching M in G that *does not cover* A. (Trivially, augmenting paths cannot exist when M covers A, because then M is maximum.)
- The expected OUTPUT of the algorithm is the following.
  - If M is not a maximum matching, then the output is an augmenting path with respect to M.
  - If M is a maximum matching, then the output is *"M is a maximum matching, no augmenting path exists"* (by Lemma 6.6), together with a Kőnig set $X \subseteq A$ such that $|X| - |N(X)| = |A| - |M|$, which proves the maximality of M by Observation 6.3.

Observation 6.8. *Any algorithm $\mathcal{A}$ that fulfils the requirements in Goal 6.7 can be used as a subroutine to find a maximum matching in an input bipartite graph G (and so determine $\nu(G)$), together with a proof of its maximality, in polynomial time as follows.*

- *Start with a trivial matching, for example, set $M = \emptyset$ or $M = \{e\}$ initially, for a non-loop edge e of G.*
- *Invoke $\mathcal{A}$ on input M (and G). If M is not maximum, then $\mathcal{A}$ finds an augmenting path P, and M can be augmented to obtain a one edge larger matching M', as seen in the proof of Lemma 6.6. Then invoke $\mathcal{A}$ on input matching M', and so on, keep repeating this to obtain larger and larger matchings, until we*
  - *(i) either reach to a matching M_1 that covers A,*
  - *(ii) or reach to a matching M_2 that does not cover A and $\mathcal{A}$ does not find an augmenting path with respect to M_2.*

- *In case (i), M_1 is obviously a maximum matching in G, we are done, the output is M_1, and $\nu(G) = |M_1| = |A|$. In case (ii), M_2 is a maximum matching in G, and the output of $\mathcal{A}$ on M_2, a Kőnig set X, is a proof of the maximality of M_2. We are done, the output is M_2, its maximality is justified by X, and we have that $\nu(G) = |M_2|$.*

The above algorithm runs at polynomial time, because the polynomial-time $\mathcal{A}$ is invoked at most $\nu(G) \leq |V(G)|/2$ times, and every augmentation can be done in polynomial time.

Remark 6.9. The algorithm $\mathcal{A}^*$ described in Observation 6.8 has a very useful property. The algorithm provides a *proof* for the correctness of its output, so the user do not have to know or understand how the algorithm works, the correctness can be verified without these details. (The user only have to check that the output matching is indeed a matching, and the output Kőnig set indeed proves its maximality.)

Now we present the Hungarian method (named in honor of the Hungarian mathematicians Dénes Kőnig and Jenő Egerváry), an algorithm that achieves Goal 6.7.

Theorem 6.10 (Hungarian method)**.** *The following algorithm fulfils the requirements of Goal 6.7.*

Hungarian method:

INPUT: A bipartite graph $G(A, B)$ and a matching M in G that does not cover A. (The unmatched vertices of A are denoted by $r_1, \ldots, r_k$.)

VARIABLES: The algorithm uses three (essential) variables: F, O and I.

- F is a rooted subforest in G with roots $r_1, \ldots, r_k$; that is, a vertex-disjoint union of k rooted subtrees of G where the root of the i^{th} subtree is r_i $(i = 1, \ldots, k)$.
- The vertices of F are partitioned into the (disjoint) sets O and I. The vertices in O are called the *outer* vertices, the vertices in I are called the *inner* vertices of F.

THE ALGORITHM:

(I) Initially, let F be the rooted forest consisting of the isolated vertices $r_1, \ldots, r_k$, where the r_i's are considered as one-vertex rooted trees. And set $O := \{r_1, \ldots, r_k\}$, $I := \emptyset$.

(II) Then repeteadly perform the following steps (1)-(2), until neither of these steps can be performed or the algorithm terminates.

// The steps (1)-(2) are illustrated in Figures 6.8-6.9 at the end of this chapter, where the edges of M are red.

(1) If some *outer* vertex u of F is adjacent in G to some *unmatched* vertex v of B, then there exists an augmenting path in G (with respect to M): Let T be the (tree) component of F containing u, and let r be

the root of T. Let P be the rv-path obtained by the concatenation of the (unique) ru-path in T and the edge uv. Then P is an augmenting path, we are done, the OUTPUT is P, and the algorithm terminates.

(2) If some *outer* vertex u of F is adjacent in G to some *matched* vertex $v \in V(M) \setminus V(F)$ *not in* F, then let w be the vertex v is matched to by M, and add v as inner vertex and w as outer vertex to F (and set $I := I \cup \{v\}$, $O := O \cup \{w\}$), together with the edges uv and vw. Then repeat step (II) with the new forest F.

(III) If neither (1) nor (2) can be performed, then we are done. The OUTPUT is *"M is a maximum matching, no augmenting path exists. This is justified by the Kőnig set O."*, and the algorithm terminates.

Proof. Before going into the details, we note that, roughly speaking, the algorithm builds a forest F in a greedy way that consists of partial augmenting paths starting from the unmatched vertices $r_1, \ldots, r_k$ of A. The structure of F is also illustrated in Figures 6.8-6.9.

We also note that in step (2), w does not belong to F either (so the step is well defined), because the endpoints of an edge of M are always added to F at same time in (2), and so if v did not belong to F, then neither did w.

We begin the proof with some easy observations. During the algorithm's run, $O \subseteq A$ and $I \subseteq B$ always hold. And for every vertex u of F, the unique ru-path P_u in F is a partial augmenting path, where r is the root of the (tree) component of F that contains u. Moreover if u is an outer (resp. inner) vertex of F, then u is an outer (resp. inner) (end)vertex of P_u; in other words, outer vertices are even distance from the root of their component, inner vertices are odd distance apart. All these properties can be verified by induction: they hold after the initial step (I), and the forest F is extended in step (2) so that these properties are preserved.

Since the graph G is finite, the forest F cannot grow infinitely, and so either step (1) or step (III) will be performed at some point.

It is obvious from the above discussion that if step (1) is performed, then the path P defined there (which is the concatenation of P_u and the edge uv) is indeed an augmenting path, because u is an outer vertex, and $v \notin V(M)$.

When step (III) is performed, $N(O) \subseteq I$ holds. This is because, no vertex of O is adjacent to a vertex in $V(G) \setminus V(F)$, as neither of steps (1)-(2) can be performed; and no vertex of O is adjacent to an other vertex of O, as $O \subseteq A$ and there is no edge between vertices of the same partite set of G. This means that all neighbors of a vertex of O are contained in I, as stated. In fact, $N(O) = I$ holds, because at the moment when an inner vertex v is added to F in step (2), v becomes a neighbor of a vertex in O (for example, of its "parent" u), which means that $N(O) \supseteq I$. Recall that k denotes the number of unmatched vertices in A. We have that

$$|O| - |N(O)| = |O| - |I| = k = |A| - |M|,$$

where the equality $|O| - |I| = k$ can be verified by an easy induction (it holds after the initial step (I), and later in steps (2) always exactly one new inner and one new outer vertex is introduced). Hence O is indeed a Kőnig set that proves the maximality of M.

Now the correctness of the algorithm is verified, and we leave the reader to check that this algorithm can be implemented in polynomial time. These imply that the requirements of Goal 6.7 are fulfilled. □

The existence of an algorithm achieving Goal 6.7 has some important theoretical consequences.

Proof of the marriage theorem (Theorem 6.4). We saw in Lemma 6.2 that if G contains a matching covering A, then no Kőnig set can exist in G.

Otherwise, if G does not contain a matching covering A, then pick a maximum matching M, run the Hungarian method on input M, and it will find a Kőnig set (cf. Goal 6.7). □

A more detailed analysis of the Hungarian method gives the following.

Theorem 6.11. *If M is a maximum matching in a bipartite graph $G(A, B)$ such that M does not cover A, then there exists a Kőnig set $X \subseteq A$ which proves the maximality of M, i.e. for which*

$$|X| - |N(X)| = |A| - |M|.$$

Proof. Run the Hungarian method on the input matching M, and it will find a suitable Kőnig set X, by Goal 6.7. □

Observation 6.3, Theorem 6.11 and Theorem 6.4 can be summarized as follows (the details are left to the reader).

Theorem 6.12 (Kőnig's formula)**.** *For any bipartite graph $G(A, B)$,*

$$|A| - \nu(G) = \max_{X \subseteq A}\{|X| - |N(X)|\},$$

or equivalently,

$$\nu(G) = |A| - \max_{X \subseteq A}\{|X| - |N(X)|\}.$$

Note that setting $X = \emptyset$ gives that $\max_{X \subseteq A}\{|X| - |N(X)|\}$ is always nonnegative.

6.3 Matchings in general graphs

The methods developed in the previous section for bipartite graphs can be extended to general graphs. It turns out that a maximum matching can be found in polynomial time in general graphs, too. This also means that $\nu(G)$ can be always determined efficiently.

We begin with the discussion of a generalization of marriage theorem.

Lemma 6.13. *Given a graph G, suppose that there exists a set $X \subset V(G)$ for which $o(G - X) > |X|$, where $o(G - X)$ denotes the number of odd components of $G - X$ (that is, components with an odd number of vertices). Then the followings hold.*

(a) There is no perfect matching in G.

(b) *Moreover, any matching of G leaves at least $o(G-X)-|X|$ unmatched vertices in $V(G)$.*

(c)

$$\nu(G) \leq \frac{1}{2}\Big(|V(G)| - (o(G-X) - |X|)\Big).$$

Proof. The setting is illustrated in Figure 6.5. The key observation here is that whenever every vertex of an odd component $\mathcal{C}$ of $G-X$ is matched by a matching M, then at least one vertex of $\mathcal{C}$ is matched to a vertex of X. This is because, on the one hand, it cannot happen that every vertex of $\mathcal{C}$ is matched to an other vertex of $\mathcal{C}$, because then the edges of M lying inside $\mathcal{C}$ would form a perfect matching in $\mathcal{C}$, which is not possible because $\mathcal{C}$ has an odd number of vertices. So there is a vertex u in $\mathcal{C}$ that is matched to a vertex v not in $\mathcal{C}$. On the other hand, v must be in X, because $\mathcal{C}$ is a component of $G-X$, thus, in G, every edge leaving $\mathcal{C}$ must end at X.

So we obtained that for every odd component $\mathcal{C}$ of $G-X$ whose vertices are all matched by M, there is an edge $e_{\mathcal{C}}$ in M which connects a vertex of $\mathcal{C}$ to a vertex of X. As these edges $e_{\mathcal{C}}$ cannot share a common vertex in X, the number of such edges $e_{\mathcal{C}}$ can be at most $|X|$, and so the number of "completely matched" odd components $\mathcal{C}$ is at most $|X|$. So there are at least $o(G-X)-|X|$ odd components of $G-X$ with some unmatched vertices, thus the number of unmatched vertices in G is at least $o(G-X)-|X| > 0$, for any matching. The statements (a) and (b) are now proven.

As a maximum matching matches $2\nu(G)$ vertices, and the number of matched vertices is at most $|V(G)|-(o(G-X)-|X|)$ by (b), hence statement (c) follows. □

We call a set $X \subset V(G)$ a *Tutte set* (named after the mathematician William Thomas Tutte), if $o(G-X) > |X|$, using the notation in Lemma 6.13. (We note that $X = \emptyset$ can be a Tutte set, this occurs when G has some odd components.)

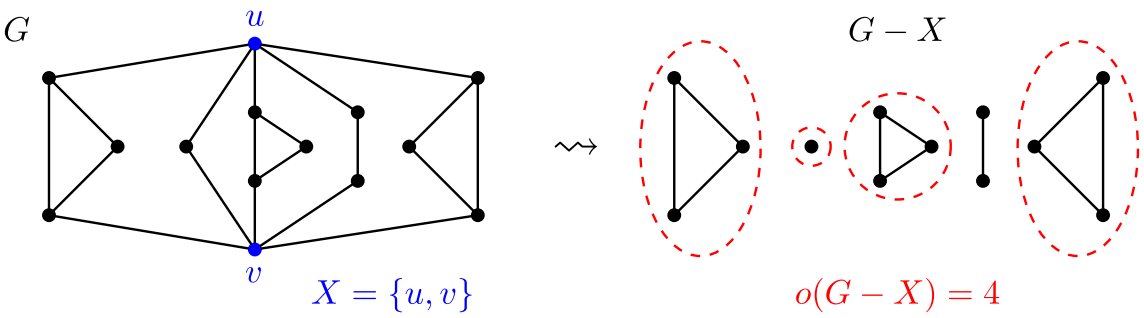

Figure 6.5: A Tutte set X

In this terminology, part (a) of the above lemma says that if a Tutte set exists in G, then no perfect matching exists in G. Actually, the converse is also true, so we have an "if and only if" theorem here.

Theorem 6.14 (Tutte)**.** *A graph G has a perfect matching if and only if there is no Tutte set in G, i.e. if $o(G-X) \leq |X|$ for all $X \subset V(G)$, where $o(G-X)$ denotes the number of odd components in $G-X$.*

We will prove Tutte's theorem on page 66 after developing an algorithm that finds a maximum matching in graphs. Analogously to the bipartite case, we set the following goal.

Goal 6.15. Our goal is to construct a polynomial-time algorithm that finds an augmenting path in G with respect to M, if such a path exists.

- So the INPUTs of the algorithm are a graph G and a *non-perfect* matching M in G. (Trivially, augmenting paths cannot exist when M is a perfect matching.)
- The expected OUTPUT of the algorithm is the following.
  - If M is not a maximum matching, then the output is an augmenting path with respect to M.
  - If M is a maximum matching, then the output is *"M is a maximum matching, no augmenting path exists"* (by Lemma 6.6), together with a Tutte set $X \subset V(G)$ such that $o(G-X) - |X| = |V(G)| - 2|M|$, which proves the maximality of M by Lemma 6.13.b.

Analogously to Observation 6.8, if we have an algorithm achieving Goal 6.15, then a maximum matching can be found in graphs (and so $\nu(G)$ can be determined) in polynomial time, together with a proof of correctness. Thus we present such an algorithm now, which is an improvement of the Hungarian method.

The greedy fashion of the Hungarian method heavily relies on the fact that if a vertex v is an inner vertex of a partial augmenting path (starting from A), then v cannot be an outer vertex of a partial augmenting path (starting from A). This is not the case for non-bipartite graphs, which makes things more complicated. As an illustration, see the bossom shaped subgraph Y in Figure 6.6, where the red edges represent the edges of a matching of M, and $r \notin V(M)$. The vertices of the odd cycle C, except x, can be both inner and outer vertices of a suitable partial augmenting path starting from r, depending on which arc of C is used when the vertex is reached from r. If we want to consider all PAPs starting from r, all vertices of C should be treated as outer vertices, because any edge that connects a vertex of C to a vertex outside of Y can be used to continue the corresponding PAP. (This scenario could not be handled by the greedy approach of Hungarian method, some vertices of C would be designated as inner vertices irrevocably.) That is why we will treat the whole cycle C as one single (outer) vertex. This is made precise in the following definition, which will be used in the algorithm below.

Definition. Given a multigraph G and a cycle C in it. The multigraph G/C obtained from G by *contracting the cycle* C is defined as

- $V(G/C) := (V(G) \setminus V(C)) \cup \{c\}$, where $c \notin V(G)$ is a new vertex,
- $E(G/C) := E(G) \setminus E(C)$, where $E(C)$ denotes the set of edges of C.
- The edge-vertex incidences are inherited from G with the modification that all vertices of C are replaced to c in these incidences. More precisely, for every

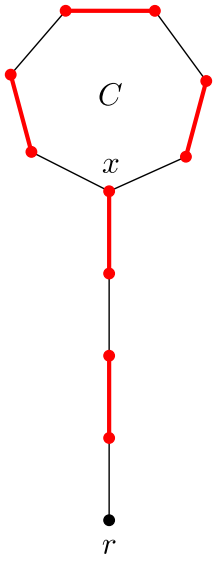

Figure 6.6: A "blossom"

edge $e \in E(G/C)$, if e connects the vertices u and v in G, then e connects the vertices $\hat{u}$ and $\hat{v}$ in G/C by definition, where

$$\hat{u} = \begin{cases} u, & \text{if } u \notin V(C) \\ c, & \text{if } u \in V(C). \end{cases}$$

Theorem 6.16 (Edmonds' blossom algorithm)**.** *The following algorithm fulfils the requirements of Goal 6.15.*

Blossom algorithm:

INPUT: A graph G_{input} and a non-perfect matching M_{input} in G_{input}. (The unmatched vertices in G_{input} are denoted by $r_1, \ldots, r_k$.)

VARIABLES: The algorithm uses five (essential) variables: G, M, F, O and I.

- The algorithm will modify the input graph G_{input} and the input matching M_{input}; the variables G and M are the actual states of G_{input} and M_{input}.
- The variables F, O and I have the same meaning as in the Hungarian method, keeping in mind that the "container" (multi)graph G can also vary.
- We must be able to restore older values of these variables. So technically, these variables should be sequences/arrays (we should introduce a sequence $G_0, G_1, G_2, \ldots$ which stores the states of G, for example), or we should use recursion etc., but we do not want to complicate the theoretical description.

THE ALGORITHM:

(I) Initially, set $G := G_{\text{input}}$, $M := M_{\text{input}}$, and let F be the rooted forest

consisting of the isolated vertices $r_1, \ldots, r_k$, where the r_i's are considered as one-vertex rooted trees. And set $O := \{r_1, \ldots, r_k\}$, $I := \emptyset$.

(II) Then repeteadly perform the following steps (1)-(3), until none of these steps can be performed or the algorithm terminates.

// The steps (1)-(3) are illustrated in Figures 6.10-6.12 at the end of this chapter, where the edges of M are red.

(1) If some *outer* vertex u of F is adjacent in G to some *outer* vertex v of a *different* component of F, then there exists an augmenting path P (with respect to M) in the *actual* G: Let T_u and T_v be the (tree) components of F containing u and v, and let r_u and r_v be the roots T_u and T_v, respectively. Then P can be defined as the concatenation of the unique $r_u u$-path in T_u, the edge uv, and the unique vr_v path of T_v. This augmenting path P can be "transformed back" to an augmenting path $\widetilde{P}$ of G_{input} with respect to M_{input} (see the Proof for more details), the output is $\widetilde{P}$, and the algorithm terminates.

(2) If some *outer* vertex u of F is adjacent in G to some (matched) vertex $v \in V(G) \setminus V(F)$ *not in* F, then extend F with vertices v and w (where $vw \in M$) and edges uv and vw in the same way as in step (2) of Hungarian method, and designate v and inner vertex, w as outer vertex of F. Then repeat step (II) with the new forest F.

(3) If some *outer* vertex u of F is adjacent in G to some *outer* vertex v of the *same* component of F, then perform a cycle contraction: Let T be the (common) component of F containing u and v, and let r be the root of T. Let $x \in V(T)$ be the last common vertex of the unique ru-path and rv-path in T, and let C be the (odd) cycle determined by the edge uv and the unique xu-path and xv-path in T. Contract the cycle in G, i.e. set $G := G/C$, and update M, F, O and I accordingly ($M := M \setminus E(C)$, $I := I \setminus V(C)$, and so on), designate the new vertex c (the contracted cycle) as outer vertex. Then repeat step (II) with the new setting G, M, F, O, I.

(III) If none of the steps (1)-(3) can be performed, then we are done. The OUTPUT is *"M_{input} is a maximum matching, no augmenting path exists. This is justified by the Tutte set I."*, and the algorithm terminates. (We note that there is no "contracted cycle" vertex in I, because such vertices belong to O, and so the final state of I defines a set of vertices in the input graph G_{input}, too.)

Sketch of proof. The full proof is rather lengthy, so we skip the details of a few straightforward arguments.

The forest F is constructed in the same way in step (2) as in the Hungarian method, so F will have the same alternating property: For any vertex u of F, the unique ru-path P_u in F is a PAP, where r is the root of the (tree) component of F containing v; and if $u \in O$, then u is an outer (end)vertex of P_u, and if $u \in I$, then u is an inner (end)vertex of P_u.

It is straightforward to check that this alternating property of F is preserved when a cycle contraction is performed in step (3), and F gets modified (as well as G, M, O, I).

It easy to verify that the cycle C contracted in step (3) is always of odd length, and $M \cap E(C)$ matches all but one vertex of C (the unmatched vertex is the vertex denoted by x in (3)).

It is obvious from the alternating property of F that the path P defined in step (1) is an augmenting path in the actual G with respect to the actual M. However, G might have been obtained from G_{input} by applying a number of cycle contractions, and we need an augmenting path in G_{input} with respect to M_{input}. That is why we undo the cycle contractions (in reverse order of their execution), and prove in a separate Lemma 6.17 that when a cycle contraction is undone, then an augmenting path can be constructed in the obtained ("blown-up") graph using the known augmenting path in the original ("contracted") graph. So when all cycle contractions are undone, we end up with an augmenting path $\widetilde{P}$ in G_{input} with respect to M_{input}, as stated,

Finally, we verify that in step (III) the set I is indeed a Tutte set that proves the maximality of M_{input}. When step (III) is performed, then $o(G - I) = |O|$. This is because the vertices of O are all isolated in $G - I$ (as none of the steps (1)-(3) can be performed), and the other components of $G - I$ has an even number of vertices (as [the restriction of] M gives a perfect matching in $G - V(F)$). Since we are going to undo the cycle contractions, we employ the notations $G_{\text{fin}} := G$, $I_{\text{fin}} := I$ and $O_{\text{fin}} := O$ to indicate the *final states* of G, I and O (when step (III) is reached). We have that

$$o(G_{\text{fin}} - I_{\text{fin}}) - |I_{\text{fin}}| = |O_{\text{fin}}| - |I_{\text{fin}}| = k = |V(G_{\text{input}})| - 2|M_{input}| \tag{6.2}$$

(recall that k is the number of unmatched vertices in G_{input}), because $|O| - |I| = k$ holds after the initial step (I) and it is preserved during the algorithm's run, so $|O_{\text{fin}}| - |I_{\text{fin}}| = k$ also holds. Now we undo the cycle contractions to obtain G_{input} from G_{fin}. As noted in the theorem, $I_{\text{fin}} \subset V(G_{\text{input}})$ also holds, i.e. no vertex of I_{fin} will be blown up to a cycle. The point is that $o(G_{\text{input}} - I_{\text{fin}}) = o(G_{\text{fin}} - I_{\text{fin}})$, because the value of $o(G - I_{\text{fin}})$ does not change when undoing a cycle contraction, because exactly one component of $G - I_{\text{fin}}$ is affected by the "blow-up", and its number of vertices is increased by an even number (as the length of the contracted cycle is always odd). Hence, by (6.2), we obtained that

$$o(G_{\text{input}} - I_{\text{fin}}) = |V(G_{\text{input}})| - 2|M_{input}|,$$

which means that the I_{fin} is indeed a Tutte set in G_{input} that proves the maximality of M_{input}, cf. Lemma 6.13.b.

The proof of correctness of the algorithm is now complete, and it is easy to check that the blossom algorithm can be implemented by a program with polynomial time bound. □

Now we present the lemma used in the above proof.

Lemma 6.17. *Let G be a graph, and let M be a matching in G. Let C be a cycle of odd length in G such that $E(C) \cap M$ matches all but one vertex of C. Assume that*

there is an augmenting path P in the contracted graph G/C with respect to M/C, where M/C is the matching in G/C obtained from M after performing the cycle contraction. Then there is also an augmenting path in G with respect to M, that can be constructed in polynomial time.

Proof. The unmatched vertex of C is denoted by x in G. Let $c \in V(G/C)$ denote the vertex in G/C that corresponds to the contracted cycle C. If c is not on P, then P is clearly an augmenting path in G, too, we are done.

So assume that c is on P. The proof is illustrated in Figure 6.7. The end vertices of P are denoted by v_1 and v_2. The vertex c divides P into two subpaths, a v_1c-path and a v_2c-path, denoted by P_1 and P_2, respectively. Observe that P_1 and P_2 are partial augmenting paths in G/C. Consider the PAPs $\widetilde{P}_1$ and $\widetilde{P}_2$ in G that correspond to P_1 and P_2, respectively (i.e. $\widetilde{P}_i$ is the path in G that consists of the same edges as P_i, for $i = 1, 2$). By the definition of G/C, $\widetilde{P}_1$ is a v_1w_1-path and $\widetilde{P}_2$ is a v_2w_2-path, where w_1 and w_2 lie on C. Since P has odd length, either P_1 or P_2 has even length; we may assume that P_1 has even length, and then so does $\widetilde{P}_1$. Since the last edge of $\widetilde{P}_1$ (the edge incident to w_1) is in the matching M, thus the vertex w_1 must be x, otherwise more than one edge of M would be incident to w_1. (If P_1 is a 0-length path, i.e. if $P_1 = (c)$, then $\widetilde{P}_1$ is *defined* to be the 0-length path (x).) It is easy to see that one of the two xw_2-arcs of C connects the PAPs $\widetilde{P}_1$ and $\widetilde{P}_2$ so that the obtained v_1v_2-path is an augmenting path in G, as desired. □

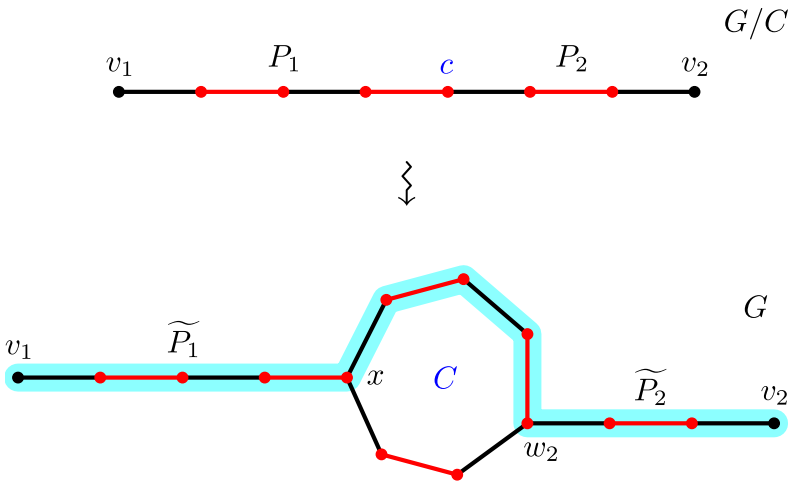

Figure 6.7: Illustration of the proof

We end this section with some noteworthy theoretical consequences of the existence of an algorithm achieving Goal 6.15.

For example, it is far from obvious that every non-maximum matching can be augmented along a suitable augmenting path, but that is the case because the blossom algorithm will always find an augmenting path for non-maximum (non-perfect) matchings, cf. Goal 6.15. Hence, taking also account of Lemma 6.6, we obtained the following corollary.

Theorem 6.18 (Berge)**.** *The matching M is a non-maximum matching in G, if and only if there is an augmenting path in G with respect to M.*

We are also in a position to prove Tutte's fundamental theorem.

Proof of Tutte's theorem (Theorem 6.14). The "only if" direction is Lemma 6.13.a.

So assume that there is no Tutte set in G, and consider a maximum matching M in the graph. M must be perfect, otherwise we could invoke the blossom algorithm on input M, and it would find a Tutte set, by Goal 6.15. □

The following corollary says that there is a standard terse proof for the maximality of a matching.

Theorem 6.19. *If M is a non-perfect maximum matching in the graph G, then there exists a Tutte set $X \subset V(G)$ which proves the maximality of M, i.e. for which*

$$o(G - X) - |X| = |V(G)| - 2|M|.$$

Proof. The blossom algorithm on input M will find a suitable Tutte set X (cf. Goal 6.15). □

Lemma 6.13.c, Theorem 6.19, and Theorem 6.14 can be summarized as follows (the details are left to the reader).

Theorem 6.20 (Berge's formula)**.** *For any graph G,*

$$\nu(G) = \frac{1}{2}\Big(|V(G)| - \max_{X \subset V(G)} \{o(G - X) - |X|\}\Big).$$

6.4 Figures

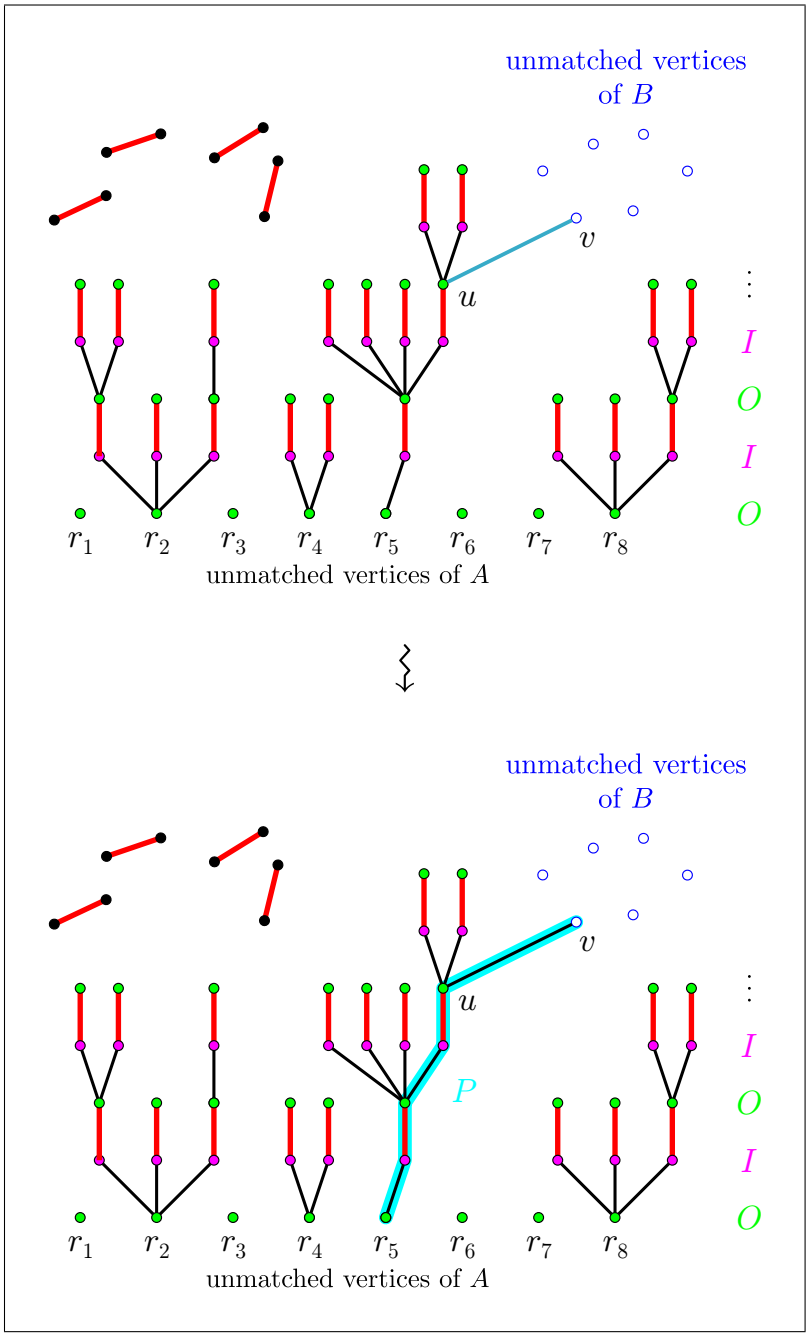

Figure 6.8: Illustration of step (1) of the Hungarian method

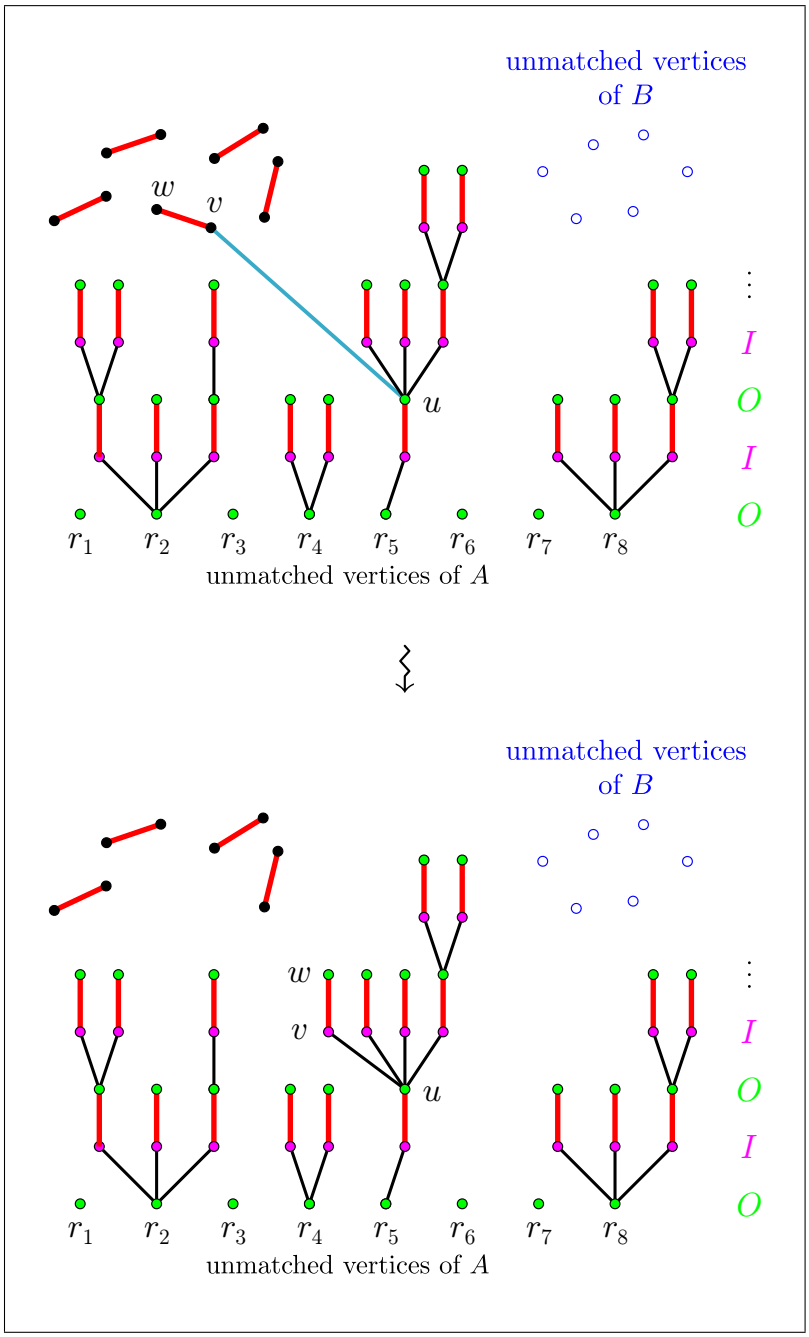

Figure 6.9: Illustration of step (2) of the Hungarian method

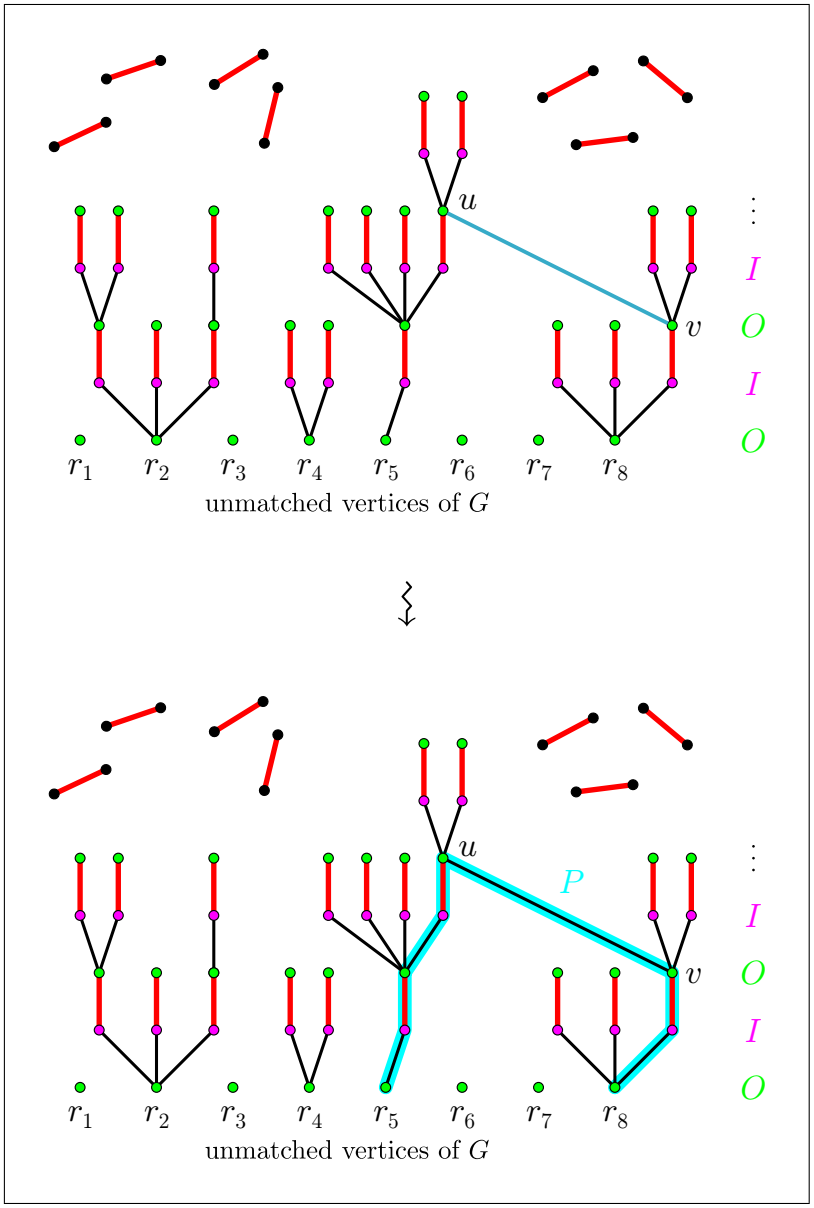

Figure 6.10: Illustration of step (1) of the blossom algorithm

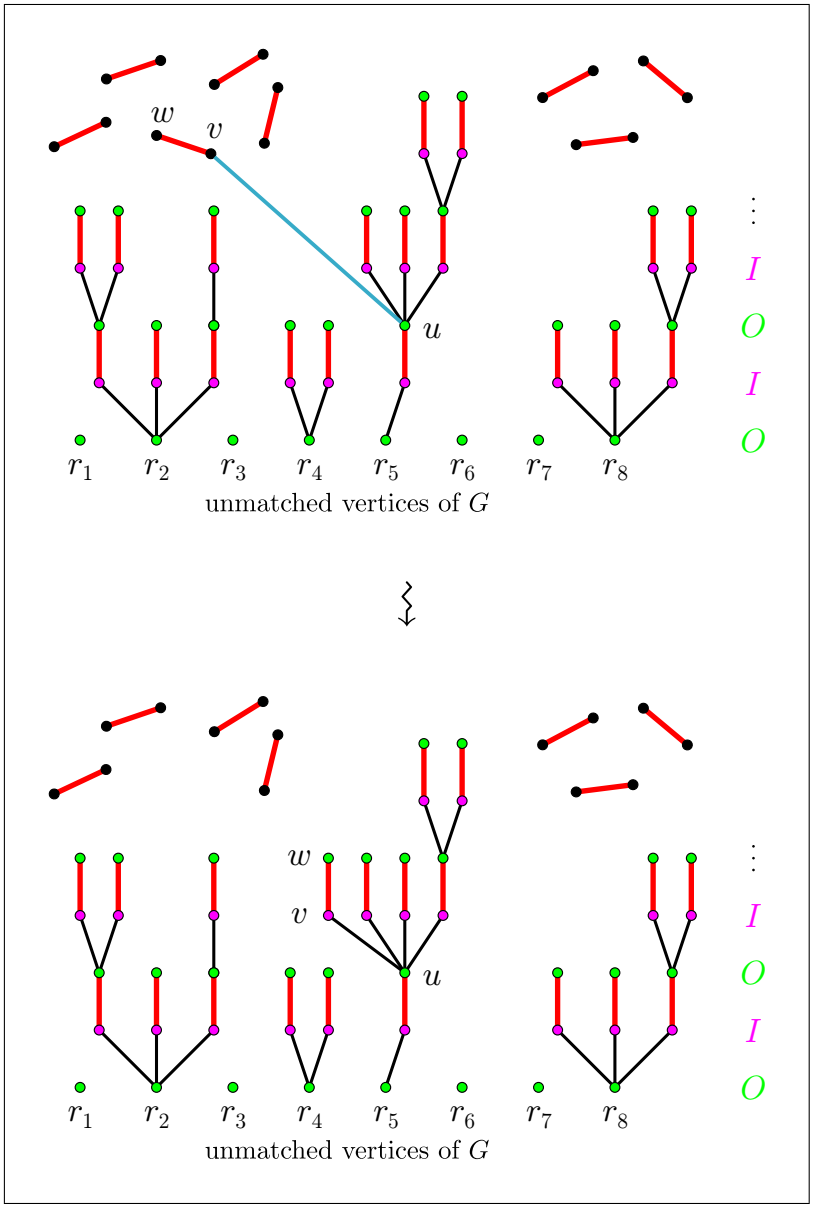

Figure 6.11: Illustration of step (2) of the blossom algorithm

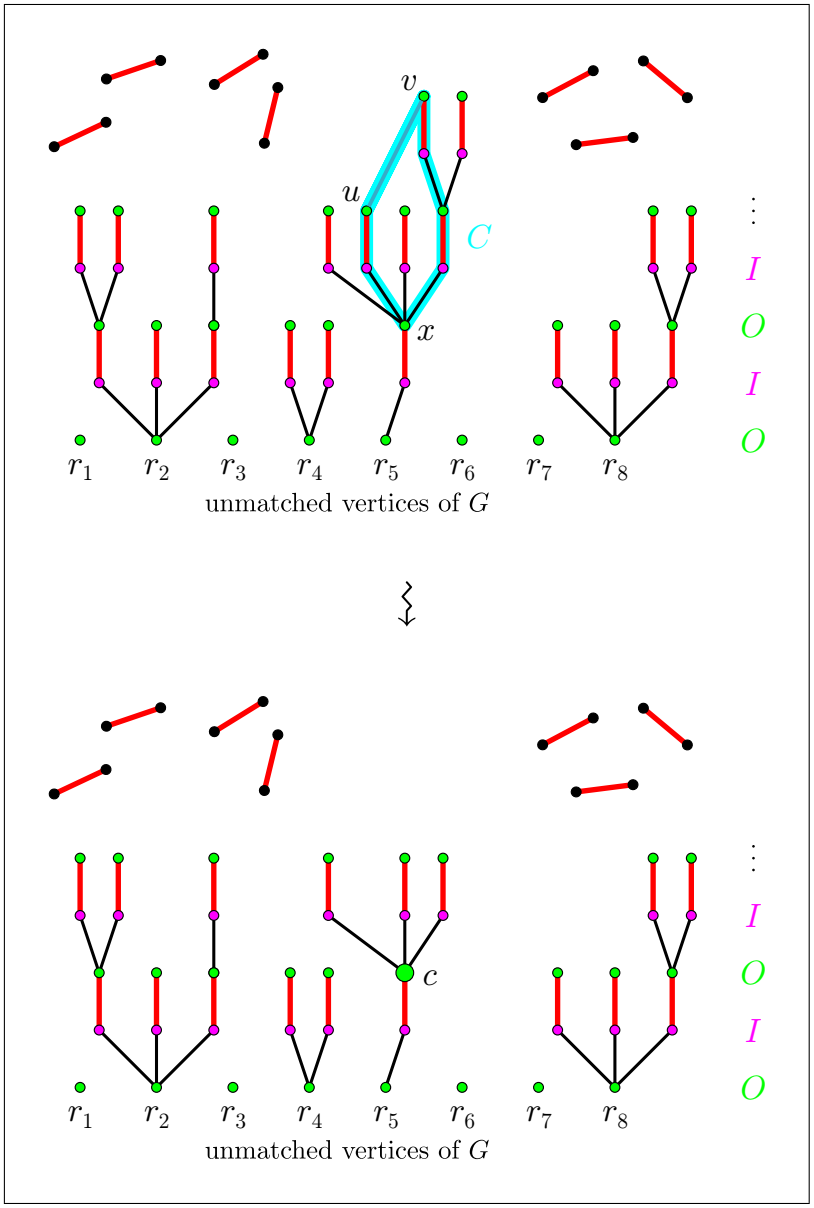

Figure 6.12: Illustration of step (3) of the blossom algorithm

Chapter 7

Colorings

7.1 Coloring the vertices of graphs

In this chapter every graph is loopless multigraph.

Definition. Let $G = (V, E)$ be a graph. A function $c : V(G) \to \mathbb{N}$ is a proper coloring of G if $c(x) \neq c(y)$ whenever $xy \in E$. [1] If $|\{c(x) : x \in V\}| = k$, then c is a k-coloring of G. We say that G is k-colorable if it has a proper k-coloring. The chromatic number $\chi(G)$ is the smallest k for which G is k-colorable. The set of vertices that have the same color are called the color classes of G.

Example. Let V denote a set of tasks. There could be pairs of tasks that cannot be fulfilled at the same time, while there could be pairs without a conflict. We construct a graph G on vertex set V that reflects this situation as follows: whenever tasks x and y are in conflict with each other, we connect them by an edge. If not, x and y will be non-adjacent. Then the minimum number of steps needed to fulfil all tasks is the chromatic number of G.

Example. There are regular memory cells in a computer, and a limited number of index registers. The processor can work much faster, if the required data is in the index register, so it does not have to bring it in from regular memory. How many index registers are needed for a given computer program? For the answer we construct a graph, the vertices represent variables of the program. Two vertices are adjacent if and only if the corresponding variables must be stored in overlapping time intervals in index registers during the execution of the program. The chromatic number of this graph can tell, how many index registers are needed in an optimal setup.

Remark 7.1. We have $\chi(G) = 1$ if and only $e(G) = 0$. If G is a graph and we obtain H by deleting some set of its edges and vertices, then $\chi(H) \leq \chi(G)$.

If T is a tree, then $\chi(T) = 2$, since one can find a proper 2-coloring of T easily as follows: starting from some arbitrarily designated root r, we let $c(x) = 1$ if its

[1] Observe that if G has a loop edge, then it cannot have a proper coloring.

distance from r is odd, otherwise we let $c(x) = 2$. Bipartite graphs are also 2-colorable, the bipartition is itself a good 2-coloring of the graph. If G has an odd cycle, then $\chi(G) \geq 3$.

Lemma 7.2. *A graph G is bipartite if and only if it has no odd cycle.*

Proof. One direction of the lemma is easy to see: if G is bipartite then it cannot have an odd cycle. For the other direction note, that without loss of generality we may assume that G is connected. Let T be any spanning tree of G, and consider a proper 2-coloring c of T as is given by Remark 7.1. Then every edge of $G - T$ must connect two vertices with opposite colors, otherwise G had an odd cycle. Hence c is a proper 2-coloring of G as well. □

Greedy coloring algorithm

Given a vertex ordering $\pi = v_1, \ldots, v_n$ of $V(G)$ we color v_i by the smallest available natural number that is not used by any of the neighbors of v_i which precede[a] it. We let $\chi_\pi(G)$ to be the number of colors the above greedy algorithm uses.

[a] These are those neighbors of v_i that have already been colored before v_i.

It is easy to see that the greedy coloring algorithm always gives a proper coloring, and that $\chi(G) \leq \chi_\pi(G)$ for every π.

Lemma 7.3. *Let G be a graph. Then the greedy coloring algorithm will use at most $\Delta(G) + 1$ colors (here $\Delta(G)$ denotes the maximum degree of G).*

Proof. The lemma follows from the fact that for every $v \in V$ and ordering π the number of neighbors of v that precede v in π is at most $\Delta(G)$. Hence, when the greedy algorithm colors v, at most $\Delta(G)$ colors are not available for v in the set $\{1, \ldots, \Delta(G), \Delta(G) + 1\}$. □

Theorem 7.4 (Brooks)**.** *Let G be a connected graph that is not the complete graph or an odd cycle. Then we have $\chi(G) \leq \Delta(G)$.*

We are not going to prove Brooks theorem, but a weakened version instead, which is easier to show. For that recall, that a graph is r-regular for some natural number r if the degree of every vertex is r.

Theorem 7.5. *Let G be a connected graph that is not the complete graph or an odd cycle. Assume further, that G is not a regular graph. Then we have $\chi(G) \leq \Delta(G)$.*

Proof. We show that $V(G)$ has an ordering π such that if we apply greedy coloring then we never need more than $\Delta(G)$ colors. Assume that $v \in V$ has the smallest degree in G, in particular, $\deg(v) < \Delta(G)$. Let T be any spanning tree of G. Let $dist_T(u)$ denote the distance of u and v in T. Order the vertices of G according to their distance from v so that if $dist_T(w) < dist_T(u)$, then u gets a smaller index

than w. If $dist_T(u) = dist_T(w)$, then one may fix any order for the two so that it satisfies the previous condition.

Denote the above ordering by π. Clearly, v will be the last vertex of π. Moreover, if u is any other vertex, than there will be a neighbor of u which follows u in π. Hence, if we use greedy coloring according to π, then there will always be at least one available color in the set $\{1, \ldots, \Delta(G)\}$ for every vertex. If $u \neq v$ then this follows from the fact that u precedes at least one of its neighbors. We must also have an available color in the set $\{1, \ldots, \Delta(G)\}$ for v since $\deg(v) < \Delta(G)$. This proves what was desired. □

Remark 7.6. Finding a good coloring a graph G by $\chi(G)$ colors is a very hard question. Even determining the chromatic number is an NP-complete problem. The theorem of Brooks is not sharp in general, in many cases the chromatic number of a graph is much smaller than its maximum degree.

The proof of Theorem 7.5 suggests a heuristic: order the vertices of a graph by their degree, starting with the largest degree vertices.

It is clear, that if G is a complete graph on n vertices, then $\chi(G) = n$. Similarly, if G contains a complete graph on q vertices, i.e. a q-clique, then $\chi(G) \geq q$. Let $\omega(G)$ denote the number of vertices in the largest complete subgraph of G, we sometimes call this the clique number of the graph. We have that $\chi(G) \geq \omega(G)$. One might have the intuition that $\chi(G) = \omega(G)$ always holds.

However, this is not the case in general, as the following simple example shows. Let G be a graph on 8 vertices which includes a triangle and a cycle on 5 vertices so that they are vertex-disjoint. We also have every edge that connects a vertex of the triangle with a vertex of the 5-cycle, and no other edges. It is an easy exercise to show that $\omega(G) = 5$, while $\chi(G) = 6$. We remark that one can construct examples in which the gap between the number of vertices in the largest clique and the chromatic number is arbitrarily large.

There is an important class of graphs, the so called *interval graphs,* for which the chromatic number and the clique number are equal. We call a graph G an interval graph, if the following holds. One can assign non-empty intervals of $\mathbb{R}$ to vertices of G so that two (different) vertices of G are adjacent if and only if the corresponding two intervals intersect. This is called the interval representation of the graph. We have the following.

Theorem 7.7. *If G is an interval graph, then $\chi(G) = \omega(G)$.*

Proof. We have already seen that $\chi(G) \geq \omega(G)$. Hence, it is sufficient to show that G has a good coloration that uses $\omega(G)$ colors.

For that we apply greedy coloring. The vertices of G are ordered according to the left endpoints of their intervals in the interval representation. Let v be any vertex of G, and assume that we color v by c. Let ℓ denote the left endpoint of the interval of v. From the greedy coloring method we get that there must be at least $c - 1$ neighbors of v that precede v in the above ordering. All the intervals of those neighboring vertices must contain ℓ. Hence, all these intervals intersect (in at least one point), which implies that G contains a clique on c vertices. □

Another result that may be useful for bounding the chromatic number of a graph was obtained by three researchers independently.

Theorem 7.8 (Gallai - Roy - Vitaver)**.** *Let G be a graph, and denote D a directed graph which we obtain by orienting the edges of G in an arbitrary way. Denote the length of the longest directed path of D by $\ell(D)$. Then $\chi(G) \leq \ell(D)+1$. Furthermore, there is an orientation D for every G such that $\chi(G) = \ell(D) + 1$.*

Proof. Let D' be the maximal spanning subgraph of D which is acyclic. One can obtain D' for example by repeatedly deleting an arbitrary edge from the directed cycle of D. When we stop, no directed cycles are left. Let $c(v)$, the color of v, be 1 plus the number of edges in the longest directed path of D' that ends at v.

The theorem is implied by the following simple observation. Assume that v is the first vertex of some directed path P. Then if P' is a path in D' that ends at v, then P' cannot have any other vertex in P, otherwise we would have a directed cycle in D'. Hence, the color of the vertices of $P - v$ are larger than $c(v)$.

Now assume that $uv \in E(D)$. If $uv \in E(D')$, then $c(u) < c(v)$ using the previous observation. If $uv \in E(D) - E(D')$, then there must be a directed path from v to u in D' using its maximality. But then we have that $c(v) < c(u)$, similarly as above. This proves the first part of the theorem.

For the other direction consider a coloring c of G by $\chi(G)$ colors. We orient the edges as follows. Assume that $uv \in E(G)$ and $c(u) < c(v)$, then the edge will point from u towards v. It is clear that the length of the longest directed path that starts at a vertex of color k is precisely of length $\chi(G) - k$. This finishes the proof of the theorem. □

The Gallai-Roy-Vitaver theorem enables us to prove a result of László Rédei in a very easy way. For stating Rédei's theorem we need a definition: a *tournament* is a complete graph in which every edge is oriented.

Theorem 7.9 (Rédei)**.** *Let T be a tournament on $n \geq 2$ vertices. Then T contains a directed Hamiltonian path, that is, a directed path on n vertices.*

Proof. It is clear that $\chi(K_n) = n$. Orient the edges of K_n so that we obtain the tournament T. By the Gallai-Roy-Vitaver theorem the longest directed path in T must have length at least $n - 1$, and since longer paths are not possible, T must contain a directed Hamiltonian path. □

We remark that since determining the chromatic number is an NP-complete problem, finding an orientation of the edges of a graph so that the longest directed path has the smallest length must also be NP-complete.

7.2 Coloring the edges of a graph

Definition. Let $G = (V, E)$ be a graph. A function $c : E \to \mathbb{N}$ is a proper edge coloring of G, if for every $xy, xz \in E$ we have $c(xy) \neq c(xz)$, that is, if two edges have a common endpoint, then their colors must be different. Notation:

$$\chi'(G) = \min\{k : \exists c : E \to [k] \text{ proper edge coloring}\},$$

so this is the smallest number of colors necessary for properly coloring the edges of G. We call $\chi'(G)$ the chromatic index or edge-chromatic number of G.

It is easy to see that $\Delta(G) \leq \chi'(G)$, if G is a simple graph. There are graphs for which the chromatic index is larger than the maximum degree. For example, let G be any cycle with odd length. Then $\Delta(G) = 2$, but, as is easily seen, $\chi'(G) = 3$.

Similarly to vertex coloring, if G has a loop, then it does not allow a proper edge coloring. Having parallel edges, on the other hand, makes sense, unlike in vertex coloring. Soon we will talk more about this.

Lemma 7.10. *We have that* $\chi'(G) \leq 2\Delta(G) - 1$.

Proof. First construct the line graph $L(G)$ of G. Recall, that the vertex set of $L(G)$ is $E(G)$, and two vertices, e_1 and e_2 are adjacent in $L(G)$, if the edges e_1 and e_2 have at least one common endpoint in G. Clearly, $\Delta(L(G)) \leq 2(\Delta(G) - 1)$, and applying the greedy vertex coloring we end up using at most $\Delta(L(G)) + 1 = 2\Delta(G) - 1$ colors. □

In case G is bipartite, we can say more.

Theorem 7.11 (Dénes König (1916))**.** *If* $G = (A, B, E)$ *is a bipartite graph, then* $\chi'(G) = \Delta(G)$.

Proof. The theorem follows from the fact that if G is an r-regular bipartite graph for some positive integer r, then it has a perfect matching. This in turn is an easy consequence of the König-Hall theorem for the existence of perfect matchings in bipartite graphs, we leave it as an exercise.

Now if G was r-regular, find a perfect matching M_1 in it, and then delete its edges from G. The resulting graph $G - M_1$ is $(r-1)$-regular, hence, if $r - 1 \geq 1$, then $G - M_1$ has a perfect matching M_2. We may continue this way, and stop only when there is no edge left. At this point we have the perfect matchings $M_1, \ldots, M_r$. Clearly, if $e, e' \in M_i$ for some $1 \leq i \leq r$, then they may get the same edge color without creating a conflict. Hence, if G is an r-regular bipartite graph, then its edges can be properly colored by r colors. This must be an optimal coloring, as less colors are not sufficient.

Finally assume that G is not r-regular for $r = \Delta(G)$. If G is not balanced, say, $|A| > |B|$, then we add $|A| - |B|$ new vertices to B to make the graph balanced. We are going to add edges to the graph to make it regular. Assume that both $x \in A$ and $y \in B$ have degrees less than r. Then we add the edge xy to the graph – note, that this way we might create multiple edges between x and y. We repeat the above procedure, and when we finish, every vertex will have degree r, so the König-Hall theorem can be applied for finding a perfect matching. □

Let us remark, that if originally G was a simple graph, then we may just create a simple r-regular graph from it as well. This goes as follows. If x, y belong to different vertex classes, are non-adjacent, and both have degrees less than r, then we add the edge xy to G. This is repeated until it is just possible. When we stop and find two vertices from different vertex classes that both have small degrees, then they must

be adjacent. For such an x, y pair of vertices we add a *gadget* to the graph. This gadget is a $K_{r,r}$. We delete one of the edges of the gadget having endpoints u and v. Say, u and x belong to the same vertex class. Then we add the edges uy and vx to the graph. With this we increased the degrees of x and y, and every vertex in the gadget still have degree r. It is easy to see that repeating this procedure results in a possibly much larger graph G', but what is important for us, G' is r-regular, hence, its edges can be properly colored by $r = \Delta(G') = \Delta(G)$ colors.

In general it is possible that $\chi'(G) > \Delta(G)$, but surprisingly, the difference of the two is always small. The result below was discovered first by Vizing, then independently, by Gupta. It is usually referred to as Vizing's theorem in the literature.

Theorem 7.12 (Vizing (1964), Gupta (1966))**.** *Let G be a simple graph. Then $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$.*

We do not prove Vizing's theorem, but mention an interesting fact: while $\chi'(G)$ may take on only two different values, it is NP-hard to decide, if $\Delta(G)$ or $\Delta(G) + 1$ is the edge-chromatic number of G.

The following is a generalization of Vizing's theorem for graphs having parallel edges, it was proved by the same authors. We introduce a new notion. We let $\mu(x, y)$ denote the multiplicity of the x, y pair (this is the number of edges connecting x and y), and let $\mu(G)$ denote the maximum of the multiplicities of edges of G.

Theorem 7.13 (Vizing (1964), Gupta (1966))**.** *Let G be a graph. Then $\chi'(G) \leq \Delta(G) + \mu(G)$.*

Since $\mu(G) = 1$ if G is a simple graph, the above generalizes Theorem 7.12. This also shows that the theorem is sharp.

There is another bound for the chromatic index of graphs by Shannon.

Theorem 7.14 (Shannon (1949))**.** *If G is a graph, then $\chi'(G) \leq \frac{3}{2}\Delta(G)$.*

We will prove a slightly weaker result, also by Shannon, which uses a theorem of Petersen. We need a definition before stating this it: a 2-*factor* of a graph is a spanning subgraph of it in which every vertex has degree 2. It is easy to see that a 2-factors is a collection of disjoint cycles.

Theorem 7.15 (Petersen (1890))**.** *Assume that G is a $2k$-regular graph, where k is a positive integer. Then G can be decomposed into k edge-disjoint 2-factors.*

Proof. Since every degree in G is even, we have an Eulerian circuit in it. Traversing this Eulerian circuit gives a natural orientation to every edge. Since we leave and enter every vertex k times, the in-degree and out-degree of every vertex will be k. Let us construct an auxiliary bipartite graph $H = (V_1, V_2, E(H))$. Whenever $u \in V(G)$, we have two copies of u in H, $u_1 \in V_1$ and $u_2 \in V_2$. If an edge vw of G is oriented from v towards w, then we have the edge v_1w_2 in $E(H)$. It is easy to see that H is a k-regular bipartite graph.

Hence, by Theorem 7.11 we have a decomposition of $E(H)$ into k perfect matchings, $M_1, \ldots, M_k$. In every M_i, every vertex of G appears exactly twice, once in both copies of V. Therefore, for every i the edges of M_i induce a spanning subgraph in which every vertex of G has degree exactly 2. □

The weaker version of Theorem 7.14 is as follows.

Theorem 7.16 (Shannon (1949))**.** *Let G be a graph. If $\Delta(G)$ is even, then $\chi'(G) \leq \frac{3}{2}\Delta(G)$. If $\Delta(G)$ is odd, then $\chi'(G) \leq \frac{3}{2}(\Delta(G)+1)$.*

Proof. We begin with a preprocessing of G in case $\Delta(G) = 2k-1$ for some positive integer k. Similarly to the proof of 7.11 we first turn G into a $2k-1$-regular graph by adding edges and vertices to G between points that have degree less than $2k-1$. Next we add an arbitrary perfect matching to the graph. After this preparation we obtain a $2k$-regular graph G'.

Now we are ready to use Theorem 7.15. We can find the k edge-disjoint 2-factors $F_1, \ldots, F_k$. Clearly, the edges of a 2-factor can be colored by at most 3 colors – if every cycle has even length, 2 colors are sufficient, otherwise we need 3. In total we need at most $3 \cdot k$ colors. This proves what was desired. □

The theorem of Shannon is tight, as the following example shows. Let u, v and w be the vertices of a graph, and assume that between any two of the vertices we have 3 parallel edges. So this graph is a "multitriangle".

We remark, that depending on the graph, either the bound $\chi'(G) \leq \Delta(G)+\mu(G)$ by Vizing and Gupta or the one $\chi'(G) \leq \frac{3}{2}\Delta(G)$ by Shannon is sharper.

Chapter 8

Planar drawings

8.1 Planar multigraphs

Definition. A *drawing* of a multigraph G is a pair (ρ, γ) where $\rho\colon V(G) \to \mathbb{R}^2$ is an injective function that maps each vertex $v \in V(G)$ to a point $\rho(v)$ in the plane, and γ is a function that maps each edge $e = uv \in E(G)$ to a continuous plane curve γ_e between $\rho(u)$ and $\rho(v)$, such that γ_e does not contain $\rho(w)$ as an interior point for any $w \in V(G)$. We refer to the points $\rho(v)$ as the *points* of the drawing of G, and the curves γ_e are referred as *edge curves*.

An *edge crossing* in a drawing is a point on the plane which is contained in two (or more) different edge curves γ_e, γ_f as interior points. A multigraph G is *planar*, if it has a drawing without edge crossings. Such a drawing is called a *planar drawing* (or planar embedding) of G.

Example. For example, the complete graph K_4 is planar, as justified by the second and third drawings in Figure 8.1.

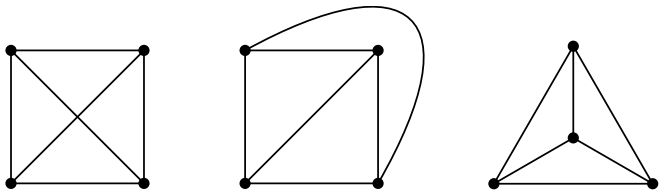

Figure 8.1: A non-planar and two planar drawings of K_4

As an other example, we note that all trees are planar. This follows from the structure theorem of trees (Theorem 1.10) and the observation that a pendant edge can be always added to a planar drawing without introducing edge crossings.

Polyhedral graphs are also planar. (A polyhedral graph is a graph formed from the vertices and edges of a 3-dimensional convex polyhedron.) For example, a planar drawing of the cube is shown in Figure 8.2; and K_4, our first example of planar graphs, is also a polyhedral graph (of a tetrahedron).

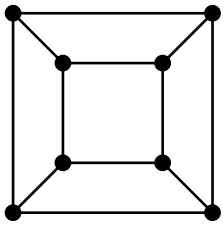

Figure 8.2: The cube graph

8.2 Dual graph

Definition. A *plane (multi)graph* is a planar embedding of a planar (multi)graph. Given a plane multigraph, its curves partition the plane into regions, including an unbounded region. These regions are open, they do not contain any point that is used in the embedding. We call these regions *faces*. The length of a face is the length of a closed walk that bound the face.

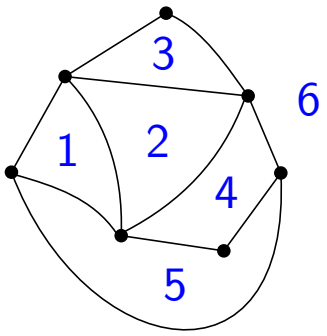

Figure 8.3: The faces of a plane graph

Every planar drawing of a K_4 has four faces. This can be verified considering only a few cases for the drawing of a K_4. For much larger graphs trying all possible drawings is not possible, but the number of faces still can be given easily.

Theorem 8.1 (Leonhard Euler (1758))**.** *Let G be a connected plane multigraph on n vertices and e edges. Then $n - e + f = 2$, where f denotes the number of faces of G.*

Proof. We will prove the theorem by induction on the number of vertices. The base case is when $n = 1$. If $e = 0$, then G has only one face, the unbounded one, and we obtain the true inequality $1 - 0 + 1 = 2$. Observe, that even if $n = 1$, we may have several edges, these then must be loops. In a planar drawing these loops do not cross. If $e = 1$, we have two faces, the unbounded face, and one bounded by the loop edge. Hence, the formula holds in this case as well. We can add more loops one-by-one, if necessary. Adding a new loop increases the number of faces by one[1]. So the formula holds for the base case $n = 1$.

For $n \geq 2$ we first observe that G must contain a non-loop edge xy, otherwise it were not connected. Contract this edge, that is, identify its two endpoints, call

[1]While this looks natural, the rigorous proof follows from the non-trivial Jordan Curve Theorem.

the resulting new vertex Z, and if any vertex v was adjacent to x or y, then vZ is included in G'. It is easy to see that G' is planar multigraph. Moreover, it has the same number of faces as G has – when contracting the edge xy we only decreased the number of edges of two faces, or one face (the unbounded one). Since G' has $n' = n - 1$ vertices, by induction we have that $n - 1 - (e - 1) + f = 2$. Hence, we proved what was desired. □

Sometimes we need to construct another plane graph from a plane graph.

Definition. Given a plane multigraph G one can form another plane multigraph G^*, called the *dual* of G, as follows. To every face of G we have a vertex of G^*, and two vertices u, v of G^* are adjacent if the corresponding two faces have a common edge e that bound both faces, moreover, there are as many edges between u and v in G^* as the number of such common edges e. See Figure 8.4 for an illustration.

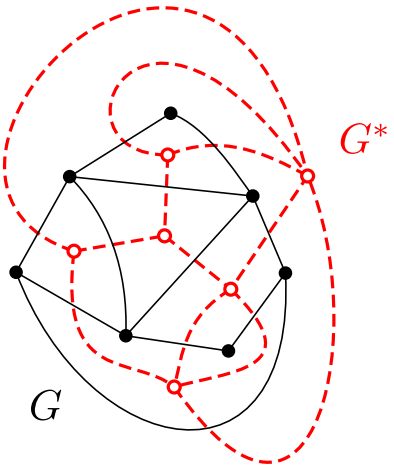

Figure 8.4: A dual graph

Theorem 8.2. *The dual G^* of a plane multigraph is planar.*

Proof. Let G be a plane multigraph. We construct a planar drawing (ρ, γ) of its dual G^*.

For each $u \in V(G^*)$, i.e. for each face u of G, the point $\rho(u)$ is defined to be an arbitrary point inside the face u. For each edge $e \in E(G)$, we have an edge e^* in G^* that connects u and v, where u and v are the two faces of G that e bounds (it is possible that $u = v$, then e^* is a loop). Then γ_{e^*} is defined to be a curve that connects $\rho(u)$ and $\rho(v)$ so that it crosses the edge curve e exactly once, and it crosses no other edge curves of G. This can be done such a way that the edge curves of G^* do not cross each other. (The edge curves starting from a point $\rho(v)$ must form a star-like shape inside the region v, as illustrated in Figure 8.4. This is possible to achieve, by intuition, but we skip the details of a rigorous proof, as it requires some knowledge of topology.) □

The following lemma is very similar to the handshake lemma.

Lemma 8.3. *Let $l_1, \ldots, l_f$ denote the length of the faces of a planar multigraph G. Then $\sum_{i=1}^{f} l_i = 2e(G)$.*

Proof. Take the dual multigraph G^*. It has f vertices, and the degree of a vertex of the dual is exactly the length of the corresponding face in G. Then the statement of the lemma translates to the handshake lemma for G^*. □

An important implication of Euler's formula is the following.

Theorem 8.4. *Let G be a connected simple planar graph on $n \geq 3$ vertices. Then $e(G) \leq 3n - 6$, and if G is triangle-free, then $e(G) \leq 2n - 4$.*

Proof. Note first that ever face length is at least 3, since G is simple and $n \geq 3$. Using Lemma 8.3 and its notation we get that $2e(G) = \sum l_i \geq 3f$. By Theorem 8.1 we have that $2e(G) \geq 3f = 3(2-n+e(G))$, implying the first formula $e(G) \leq 3n-6$.

If G is triangle-free, then every face of it has at length at least 4, therefore $2e(G) \geq 4f$ in this case. Simple calculation finishes the proof of the theorem. □

8.3 Kuratowski's theorem

In theory, if a multigraph G is planar, then there is always a short proof of this fact: a planar drawing of G. But how can we argue if G is non-planar, i.e. how can we prove that no planar drawing of G *exists*? Fortunately, there is always a short proof for the non-planarity, too, due to a nice characterization theorem of Kuratowski.

First we give the two basic examples of non-planar graphs.

Theorem 8.5. *The complete graph K_5 and the complete bipartite graph $K_{3,3}$ are non-planar. (We note that $K_{3,3}$ is often called the "three houses-three wells" graph.)*

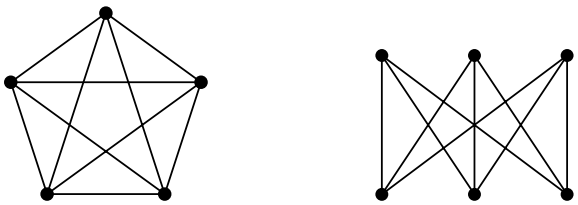

Figure 8.5: The two basic examples of non-planar graphs

Proof. This theorem is a direct corollary of Theorem 8.4. It can be proved very easily by just counting the number of vertices and edges in these graphs. For K_5, we use the first formula, for $K_{3,3}$, we use the second one. □

From the two basic examples we can construct many non-planar graphs.

Lemma 8.6. *All submultigraphs of a planar multigraph are also planar.*

Proof. Let H be a submultigraph of a planar multigraph G. Let $X \subseteq V(G) \cup E(G)$ denote the set of vertices and edges by whose removal H is obtained from G, i.e. for which $H = G - X$. A planar drawing of H can be obtained from a planar drawing of G by deleting the points and edge curves corresponding to the vertices and edges in X. □

Definition. A *subdivision* of a multigraph G is a multigraph S obtained from G by replacing the edges of G by internally vertex-disjoint paths with the same endpoints. (So the new vertices have degree 2 in S, as they belong to exactly one edge of G.) See Figure 8.6 for an illustration. When an edge $e \in E(G)$ is replaced to a path P_e of length ℓ in S, we say that $\ell - 1$ new vertices are inserted into e.

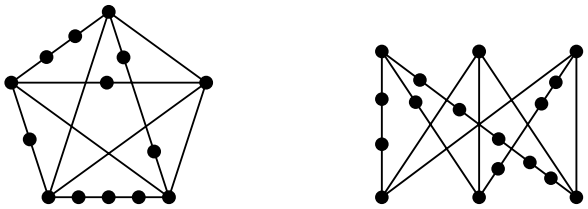

Figure 8.6: A subdivision of K_5, and a subdivision of $K_{3,3}$

Lemma 8.7. *Let S be a subdivision of a multigraph G. Then S is planar if and only if G is planar.*

Proof. Assume that G is planar, and consider a planar drawing (ρ, γ) of G. The subdivision S is obtained from G by inserting new vertices into edges of G. These vertex insertions can be performed in the planar drawing (ρ, γ), too: When an edge $e \in E(G)$ is replaced to a path P_e of length ℓ in S, then we subdivide the corresponding edge curve γ_e at $\ell - 1$ distinct interior points in the drawing (these interior points represent the internal vertices of P_e, and the curve segments represent the edges of P_e). In this way, we obtain a planar drawing of S.

For the converse, assume that S is planar. Then G can be obtained from S by "deleting" the new vertices from the edges of S. Analogously to the above, these vertex deletions can be performed in a planar drawing of S, too, yielding a planar drawing of G. □

Combining Theorem 8.5 with Lemma 8.6 and Lemma 8.7, we proved that every multigraph containing a subdivision of K_5 or $K_{3,3}$ is non-planar. By the following beautiful theorem, there are no other non-planar multigraphs. (We omit the proof of the difficult direction.)

Theorem 8.8 (Kuratowski)**.** *A multigraph G is planar if and only if G does not contain a subgraph that is a subdivision of K_5 or $K_{3,3}$.*

Example. The Petersen graph is non-planar, because it contains a subdivision of $K_{3,3}$, see Figure 8.7.

We note that there exist *linear-time* algorithms to decide whether a graph is planar or not. The first linear-time planarity testing was proposed by Hopcroft and Tarjan in 1974.

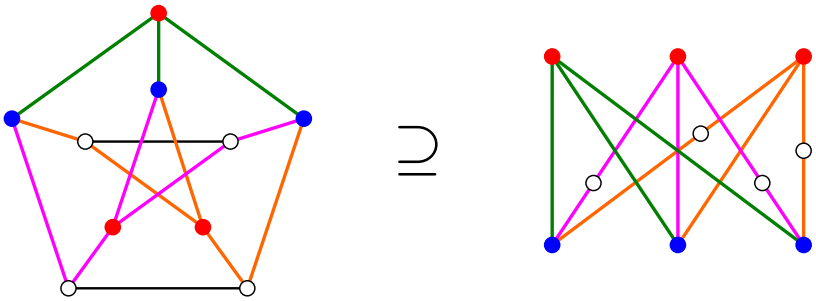

Figure 8.7: A subdivision of $K_{3,3}$ in the Petersen graph

8.4 Four color theorem

We end this chapter with one of the famous theorems in graph theory, the four color theorem.

Theorem 8.9 (Four color theorem, dual version)**.** *Let G be a plane multigraph, such that every edge of G bounds two different faces of G. Then the faces of G can be colored using four colors such a way that no two adjacent faces have the same color. (Two faces are adjacent if their boundaries have a common edge.)*

In other words, the above theorem states that G^* is 4-colorable, if G^* loopless. Since G^* is planar by Theorem 8.2, hence Theorem 8.9 is implied by the following.

Theorem 8.10 (Four color theorem)**.** *Every loopless planar multigraph is 4-colorable.*

This is a groundbreaking result in mathematics. It was first proved by Appel and Haken in 1976. Their original proof is 139 long, and it is computer-assisted: They were able to reduce the problem to check that 1936 special graphs have a certain property, and this verification was done by computer after 1200 hours of running time. Robertson, Sanders, Seymour and Thomas obtained a more compact proof in 1997, their proof took 35 pages, and they provided an open-source C program for the computer-assisted part. Nowadays there is no doubt that the four color theorem is true, but all known proofs use computer.

Chapter 9

Walks, tours

9.1 Eulerian tours

In the 18th century the *Bridges of Königsberg* problem was formulated. The problem is about the downtown of Königsberg. A river crosses the city. The two banks and two islands form the center, and there are seven bridges, connecting them in the manner seen on the next picture.

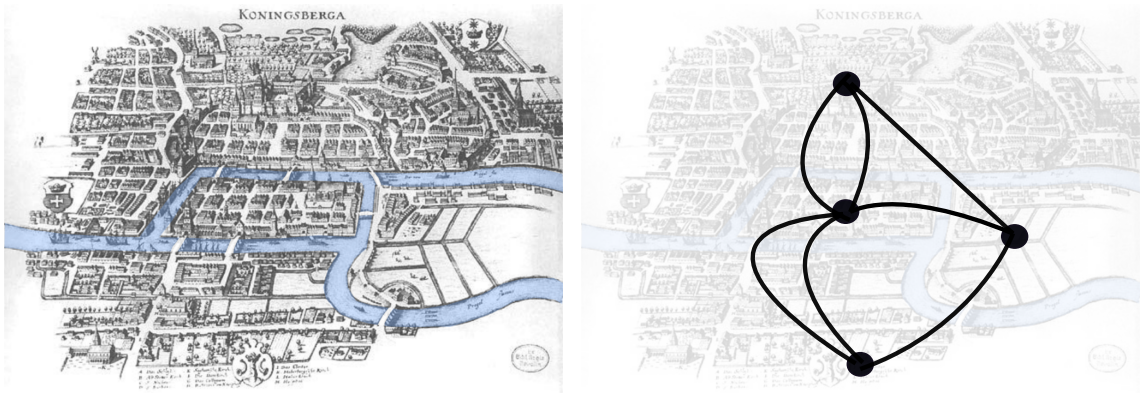

Figure 9.1: On the left the historical center of Königsberg, on the right the graph of Königsberg. (Source of the left picture is https://scilogs.spektrum.de/, article *The Bridges of Königsberg.*)

The citizens of Königsberg asked the following question: Is it possible to walk in the center and meanwhile traverse every bridge exactly once? It is obvious that the problem is about a multigraph G (vertices are the two banks, and two islands; edges are the bridges; the multigraph is on the right hand side of the above picture). The citizens asked for a tour in G with the property that its edge set is $E(G)$.

The problem was considered by Euler. He generalized the question and solved the general form.

Definition. Let G be a multigraph. $\mathcal{T}$ tour is Eulerian tour in G iff $E(\mathcal{T}) = E(G)$ and $V(\mathcal{T}) = V(G)$.

The second condition is technical, its only reason is to exclude isolated vertices (we can have isolated vertices if we only require $E(\mathcal{T}) = E(G)$).

Theorem 9.1 (Euler theorem, first version)**.** *Let G be an arbitrary multigraph. G has a closed Eulerian tour iff the following two conditions hold:*

(E1) G is connected,

(E2) all degrees in G are even.

Assume that G has a closed Eulerian tour, $\mathcal{T}$. $\mathcal{T}$ must contain all vertices, specially G contains tour between any two vertices, i.e. G is connected.

We take any vertex v and walk through $\mathcal{T}$. We visit v certain many times, say m times. (If v is the initial vertex of the tour, then leaving it at the beginning and returning it at the end are considered as one visit.) It is easy to see that any visit contributes 2 to the degree of v, and these contributions add up (we walk through a tour). The Eulerian property gives us that all edges incident to v are counted, so $d(v) = 2m$. We have proved that the two conditions are sufficient to have an Eulerian tour.

The harder part is the reverse direction. We assume (E1) and (E2). We must find a closed Eulerian tour.

For this we introduce the *greedy tour building* process.

Greedy tour building. We are given a $v_0 \in V$, initial vertex.

- *(Initialization): $a = v_0$. // a is the actual vertex of our tour.*
- *(Step): Choose an edge that is incident to a, and our walk has not traversed it so far. If there is no such edge, then STOP. If we find an edge $e = aa^+$, then we traverse it, and update the actual vertex: $a \leftarrow a^+$. Repeat (Step). // Since every edge is traversed at most once, the tour building will stop. The last vertex of the walk/tour is called terminal vertex.*

The next simple claim will be crucial.

Observation 9.2. *Assume that all vertices have even degree. We run a greedy tour building starting at an arbitrary initial vertex, v. Then the terminal vertex is necessarily is v, i.e. the tour, we built is closed.*

Indeed. If the actual vertex of the walk is u (different from v), then we visited u certain times, say m times, finally we entered it. The number of traversed edges, incident to u is odd. Hence there must be an edge, incident to u that is not traversed, the walk is not stopped.

Based on this observation we can easily (without any idea) can find a closed tour in G. The problem is that in general for the constructed $\mathcal{T}$ the Eulerian property (visit all vertices and edges) won't be satisfied. Since G is connected, in this case we have a vertex v, that is visited by $\mathcal{T}$, and an edge e, that is incident to v, but is not traversed.

Start a greedy tour building process, starting at v, in the multigraph formed by the edges that not traversed by $\mathcal{T}$. Note that this multigraph also satisfies that any vertex has even degree (although it is not necessarily connected). Hence our observation guarantees that we end up with a closed tour, $\mathcal{T}'$.

Let $\mathcal{T}^+$ denote the following walk: We walk through $\mathcal{T}$, but we stop at v, and we traverse $\mathcal{T}'$. When we are done (we must be at v), then we finish the walk through $\mathcal{T}$. Since $\mathcal{T}$ and $\mathcal{T}'$ are tours, $\mathcal{T}'$ is using edges not traversed by $\mathcal{T}$ the obtained $\mathcal{T}^+$ is a tour. We say that $\mathcal{T}^+$ is constructed by *the insertion process.*

So we can enlarge a non-Eulerian closed tour. This proves the first version of Euler's theorem. The proof also provide an algorithm that finds a closed Eulerian tour in G when (E1) and (E2) are satisfied:

Euler's algorithm: We are given a graph G satisfying (E1) and (E2), furthermore a $v_0 \in V$, an arbitrary initial vertex.

- *(Initial closed tour): Do a greedy tour building, starting v_0. Let $\mathcal{T}$ denote the closed tour we constructed.*
- *(Insertion step): Until $E(\mathcal{T}) \neq E(G)$ repeat the following: Find a vertex v, that is visited by $\mathcal{T}$, and an edge e, that is incident to v, but is not traversed so far. Starting from v, traversing e first we do a greedy tour building using only edges of $E(G) - E(\mathcal{T})$. After obtaining $\mathcal{T}^+$, perform an insertion process.*

Now we characterized multigraphs having closed Eulerian tours. The case of non-closed Eulerian tours is an easy consequence of it. Its proof is an easy exercise.

Theorem 9.3 (Euler theorem, second version)**.** *Let G be an arbitrary multigraph. G has a non-closed Eulerian tour iff the following two conditions hold:*

(E1) G is connected,

(E2) all degrees in G are even, except two (i.e. the number of vertices with odd degree is 2).

We mention that (E1) and (E2) are satisfied then the two vertices of odd degree must be the first and last vertex of the Eulerian tour guaranteed by the theorem.

The two versions can be unified.

Theorem 9.4 (Euler theorem, full version)**.** *Let G be an arbitrary multigraph. G has an Eulerian tour iff the following two conditions hold:*

(E1) G is connected,

(E2) the number of vertices of odd degree is 0 or 2.

We mention a weakness of the full version. It hides the fact that the two possibilities in (E2) and the closed/non-closed option for the tour are closely related.

Finally we intorduce an important notion.

Definition. A multigraph G is called Eulerian iff every vertex has even degree.

9.2 Chinese postman

Mei-Ko Kwan, Chinese mathematician, introduced a weighted version of the Eulerian tour problem. It is called Chinese postman problem.

The Chinese postman problem (CPP) is as follows: We are given a connected multigraph, a distinguished vertex s, and a length function on its edge set: $\ell : E(G) \to \mathbb{R}_{++}$. Easy to extend the length function to walks. To compute the length of a walk we take the sequence of edges (that is a set of edges with multiplicities) and add their length. CCP asks to determine the shortest closed walk traversing all edges and starting at s.

s is called *post office*. Easy to see that it plays no role in this problem.

If G is an Eulerian multigraph, the problem is obvious. Any Eulerian tour (easy to determine one) is an optimal solution. Any feasible postman walk $\mathcal{W}$ is such that $\ell(\mathcal{W}) \geq \ell(E(G))$ ($\ell(E(G))$ is the sum of all edge lengths in G).

Notation. Let G be a multigraph with a length function on its edge set. $\ell(G)$ denotes the sum of the edge lengths in G.

We expand this idea. Let $\mathcal{W}$ be a feasible walk for CPP. In the sequence of edges along $\mathcal{W}$ each $e \in E(G)$ must have a positive multiplicity. $1 + \mu_+(e)$ denotes this multiplicity ($\mu_+ : E(G) \to \mathbb{N} = \{0, 1, 2, \ldots\}$). We introduce two auxiliary multigraphs: First, let $\widehat{G}$ be the multigraph that we obtain from G by adding for each $e \in E(G)$ $\mu_+(e)$ parallel edges next to e. Second, let G_+ be the multigraph that we obtain on the vertex set $V(G)$ by adding for each $e = uv \in E(G)$ $\mu_+(e)$ parallel edges, connecting u and v. Note that $G, G_+ \subset \widehat{G}$. If G is Eulerian, and $\mathcal{W}$ is an Eulerian tour, then $\widehat{G} = G$ and $G_+ = E_{V(G)}$ (the empty graph on $V(G)$). In general $\widehat{G} = G \dot{\cup} G_+$, where $\dot{\cup}$ is the edge disjoint union of the two multigraphs on the same vertex set.

Easy to see that based on $\mathcal{W}$ we can construct an Eulerian tour in $\widehat{G}$, specially $\widehat{G} = G \dot{\cup} G_+$ is an Eulerian multigraph, i.e. all degrees are even. This is equivalent to the fact that the set of vertices of odd degree in G is the same as in G_+. O denotes this common set.

Based on an arbitrary feasible walk $\mathcal{W}$ we made a lot of observations. In some sense these logical steps can be reversed: Take any G_+ multigraph on $V(G)$ that

(i) the edges connect pairs of vertices, that are connected in G,

(ii) the set of vertices of odd degree is $O = \{v \in V(G) : d_G(v) \text{ is odd}\}$.

Take $G \dot{\cup} G_+$, an Eulerian multigraph. Find an Eulerian tour, $\mathcal{T}$ in it. One can project this into G, and obtain a feasible solution, $\mathcal{W}$ for CPP. The length of $\mathcal{W}$ is $\ell(G) + \ell(G_+)$. We summarize our claims.

Observation 9.5. *Given G, a connected multigraph with a length function on $E(G)$. The CPP for it is equivalent to the following:*

Consider the multigraphs G_+ with property (i) and (ii). Minimize $\ell(G_+)$: find a feasible G_+ with minimal length.

This observation with the next one are crucial for solving CPP.

Observation 9.6. *Let H be a multigraph, O is the set of vertices of odd degree. $k \in \mathbb{N}$ denotes $|O|/2$. Then there exist k edge disjoint path in H, that the set of endvertices of the paths is exactly O.*

Proof. Assume that $k > 0$. It is easy to find a uv path in H for some $u \neq v \in O$: Take an arbitrary $u \in O$. In its component there must be at least one other vertex of O, say v (apply the Handshake Lemma for this component). There is a path P connecting these two vertices in the same component.

Finding only one path seems too little. This is not the case. Let H_0 be the graph we obtain by deleting the edges of P from H. H_0 is a multigraph, the set of vertices of odd degree in H_0 is $O - \{u, v\}$. I.e. the halved cardinality of the set of vertices of odd degree in H_0 is $k - 1$.

A simple induction or a recursive algorithm finishes the proof. □

The first Observation rephrased the CPP: We have to find an optimal G_+ graph. The second Observation says that it is enough to look for the optimal G_+ among the matching path systems (each path of a system defines a vertex pair, its two endvertices; these pairs must form a perfect matching of the elements of O). Specially the edge set of an optimal G_+ is nothing else than an edge set of a matching path system for O. Also a path in an optimal path system must be the shortest path among its endvertices. These facts immediately lead to an algorithm.

Chinese postman algorithm (Edmonds). Given a multigraph G and a length function on it edge set.

- *(Odd degree step) Determine the set of vertices, O. // Note that $|O|$ is even, see Handshake lemma.*
- *(Shortest path step): Determine the shortest paths between any pair of vertices from O. // One possibility is to use Dijkstra's algorithm in multiple times.*
- *(Auxiliary graph): Let A be the complete graph on O, with edge weights: the $w(e)$ weight of the edge $e = uv$ is the length of the shortest uv path, determined in the (Shortest path step). Let P_e an optimal path connecting the two endvertices of e.*
- *(Matching step): Determine a minimal cost perfect matching in (A, w): M_{alg}. // Let $\mathcal{P}$ the set $\{P_e : e \in M\}$.*
- *(Auxiliary graph 2): Consider the graph $\widehat{G}$ on $V(G)$ with the edge set*

$$E(G) \dot{\cup} \dot{\bigcup}_{P \in \mathcal{P}} E(P).$$

  // Note that $\widehat{G}$ is an Eulerian graph.

- *(Eulerian step): Find a closed Eulerian tour in $\widehat{G}$, and project it to G. $\mathcal{W}$ denotes the closed walk we obtain this way. Output $\mathcal{W}$.*

The correctness of the algorithm (the optimality of the output) is straight forward from the previous discussion.

Chapter 10

Paths, cycles

10.1 Hamiltonian paths, Hamiltonian cycles

Definition. A path $\mathcal{P}$ is a Hamiltonian path iff $V(\mathcal{P}) = V(G)$, i.e. $\mathcal{P}$ visits all vertices.

A cycle $\mathcal{C}$ is a Hamiltonian cycle iff $V(\mathcal{C}) = V(G)$, i.e. $\mathcal{C}$ visits all vertices.

A basic question: Given a graph G. Decide whether G has a Hamiltonian path/cycle.

For those who studied complexity theory we mention that the Eulerian tour problem, the Chinese postman problem are solvable in polynomial time. The Hamiltonian cycle problem is $\mathcal{NP}$-complete, hence it is considered intractable (see one of the seven Millennium prize problems).

In spite of the theoretical difficulties we might try to attack the problem with some heuristical ideas. Assume that we are given, $\mathcal{P}$ a path in G. Our strategy is to lengthen this $\mathcal{P}$ until we are able to do so. We hope that we obtain a Hamiltonian path.

For this we introduce the *greedy path building* process.

Greedy path building. We are given a $v_0 \in V$, initial vertex.

- *(Initialization):* $a = v_0$. *// a is the actual vertex of our tour.*
- *(Step): Choose a neighbor of a, that is not visited so far. If there is no such such neighbor, then STOP. If we find an edge $e = aa^+$, that a^+ is not visited then we traverse e, and update the actual vertex: $a \leftarrow a^+$. Repeat (Step). // Since every vertex is visited at most once, the walk/path will stop. The last vertex of the walk is called terminal vertex.*

It is obvious that if we built path, $\mathcal{P}$ by the greedy process than all the neighbors of t, the terminal vertex of $\mathcal{P}$, are on the path. Take an arbitrary neighbor of t, say s (note that s is on the path). The next Figure shows this situation.

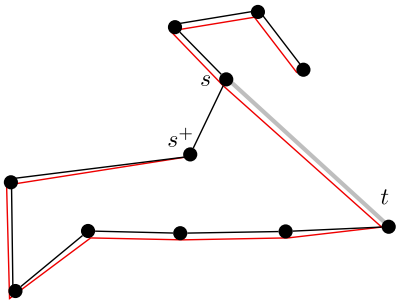

Figure 10.1: The greedy algorithm got stuck and provided $\mathcal{P}$ (black edges), the terminal vertex of it is t. A neighbour of t is presented, s ($ts \in E$ is a gray edge). s^+ is the vertex following s on the path $\mathcal{P}$. The red path starts as $\mathcal{P}$, but from s it goes to t, then follow $\mathcal{P}$ in reversed order till s^+.

On the Figure we see a (red) alternative path of $\mathcal{P}$. We have $d(v)$ many such alternative paths (note that $\mathcal{P}$ itself is an alternative path, we obtain it when $s = t^-$, the vertex just preceeding t on the path $\mathcal{P}$). We summarize the properties of these alternatives:

Observation 10.1. *The alternative paths of $\mathcal{P}$ have the following porperties:*

- *they start at the same vertex,*
- *the visit the same set of verices, hence they have the same length,*
- *their terminal vertices are different.*

Now we can extend our greedy path building procedure. When we get stuck, we try to find an alternative path, that can be continued by the greedy step. We might be lucky and find a Hamiltonian path, or at least a long path in G. The next theorem says that this success is guaranteed if our graph has large degrees.

Theorem 10.2 (Dirac's theorem)**.** *Let G be a graph with minimal degree at least $|V|/2$. Then G has a Hamiltonian path.*

Proof. Let $\mathcal{P}$ a path that we found by the greedy path building process. If $\mathcal{P}$ is a Hamiltonian path, then we are done. Assume that we have a vertex u, that is not on $\mathcal{P}$: $u \notin V(\mathcal{P})$.

The degree of the terminal vertex is at least $|V|/2$. Hence we have at least $|V|/2$ alternative paths. Let T be the terminal vertices of these paths. We know that $|T| \geq |V|/2$. Let N be the set of neighbors of u. We know that $|N| \geq |V|/2$. T and N are two subsets of $V - \{u\}$. Their sizes guarantee a common vertex c. One of the alternatives of $\mathcal{P}$ starts at v and leads to c, that can be continued to u. We obtained a longer path than $\mathcal{P}$. We might have a Hamiltonian path. If this is not the case then we might use the greedy method or if that is not possible, then the method described above the lengthen our path.

We just describe an algorithm that leads to a Hamiltonian path in G. □

Easy to note that we used an arbitrary initial vertex v, and after it we only constructed paths starting at v. The claim of the theorem can be strengthened.

Corollary 10.3. *Let G be a graph with minimal degree at least $|V|/2$, and v is an arbitrary vertex. Then G has a Hamiltonian path starting at v.*

Our ideas guarantee many alternative Hamiltonian paths starting at the same vertex v. Since v has high degree — by the conditions of our theorem — we have an alternative Hamiltonian path, with terminal vertex, that is a neighbor of v. We can state a new strengthened claim.

Corollary 10.4. *Let G be a graph with minimal degree at least $|V|/2$. Then G has a Hamiltonian path, such that its initial vertex and final vertex are connected.*

The guaranteed Hamiltonian path and the edge in the theorem is "almost" a Hamiltonian cycle. The case $|V| = 2$ is the only one, when the above theorem doesn't "give" us a Hamiltonian cycle. We obtain the following central theorem of graph theory.

Theorem 10.5 (Dirac's theorem, cycle form)**.** *Let G be a graph with minimal degree at least $|V|/2$. Assume that $|V| \neq 2$. Then G has a Hamiltonian cycle.*

10.2 Traveling salesman problem

The traveling salesman problem (denoted as TSP) is a weighted version of the Hamiltonian cycle problem: Given a graph with a cost function on its edge set ($c : E(G) \to \mathbb{R}_{++}$). The cost function can be easily extended to walks (we did this type of step in the case of length function). We want to find the cheapest Hamiltonian cycle.

We note that a missing edge — two non-connected vertex, u, v — means that while walking in the graph we cannot make a step from u to v. This restriction can 'simulated' by adding an uv edge with very high cost. So we can assume that our underlying graph is a complete graph. When we discuss TSP we use this assumption: $G = K_n$, i.e. n denotes the number of vertices.

Since TSP is a generalization of the Hamiltonian cycle problem it is a hard problem. We only consider a special case of it: We assume that the cost function satisfies the triangle inequality:

$$c(uv) \leq c(uw) + c(wv), \quad \text{for any three different vertices.}$$

If the cost is proportional to a geometrical distance then this assumption is satisfied. We also mention that in many applications the triangle inequality doesn't hold. If the property is satisfied we write c is a Δ-cost function.

Now on we only consider Δ-cost functions. In spite of this relaxation we will give only approximation algorithms for the problem.

Approximation algorithms are designed to handle optimization problems. We discuss only the case of minimization (maximization is completely analogous). We are given a set $\mathcal{F}$ of feasible solutions (usually a set described by certain constrains) and an objective function over the set of feasible solutions $o : \mathcal{F} \to \mathbb{R}_{++}$. Our goal is to find an $\mu \in \mathcal{F}$ such that the value of o is minimal: $o(\mu) \leq o(\omega)$ for every $\omega \in \mathcal{F}$, i.e. $o(\mu) = \min\{o(\omega) : \omega \in \mathcal{F}\}$.

Notation. If we are talking about a specific optimization problem $\mu_{\text{opt}} \in \mathcal{F}$ always denotes an optimal solution

$$o(\mu_{\text{opt}}) = \min\{o(\omega) : \omega \in \mathcal{F}\}.$$

Many of the basic optimization problems are very hard, we do not expect efficient algorithm for solving them. In some cases we are satisfied by an algorithm $\mathcal{A}$ that efficiently computes a $\mu_{\text{alg}} \in \mathcal{F}$ solutions and gives a guarantee that

$$o(\mu_{\text{alg}}) \leq (1 + \alpha)o(\mu_{\text{opt}}).$$

In this case we say '$\mathcal{A}$ is an α-approximation algorithm'.

Assuming triangle inequality in TSP has an important consequence, that is summarized in the next claim.

Observation 10.6. *Assume that we are given a Δ-cost function c on $E(K_n)$ and a $\mathcal{W}$ closed walk with $V(\mathcal{W}) = V(K_n)$. Then we can efficiently construct a Hamiltonian cycle $\mathcal{H}$ with low cost:*

$$c(\mathcal{H}) \leq c(\mathcal{W}).$$

The claim says that we can think about closed walks when we want to visit all vertices. The next claim is also very easy.

Observation 10.7. *We are given T, a spanning tree of K_n. Then we can construct a closed walk $\mathcal{W}_T$, that traverses each edge of T exactly twice and doesn't involve any other edge. Hence the $c(\mathcal{W}_T) = 2c(T)$, where the cost of a tree (or in general the cost of an edge set) is the sum of the cost of its edges.*

Indeed. Take T (its the vertex set is $V(K_n)$ and for each $e \in E(T)$ add one new copy of this edge: e'. The multigraph, we obtain this way has an Eulerian tour. This tour can be projected into K_n, and we obtain the desired $\mathcal{W}_T$.

Now we are ready to present our first approximation algorithm for the TSP with Δ-cost function.

$\mathcal{A}_1$: Given $G = K_n$, a Δ-cost function on $E(G)$.

- *(Spanning tree step): Find a minimum cost spanning tree T_{alg} of G. // We can do it by Kruskal algorithm.*
- *(Walk step): Use T to construct a $\mathcal{W}_{alg}$ closed walk, visiting every vertex. // See Observation 2.*
- *(Simplification step): From $\mathcal{W}_{alg}$ construct a Hamiltonian cyle $\mathcal{H}_{alg}$, and output it. // See Observation 1.*

The essence of any approximation algorithm is the guarantee for its performance and its proof.

Theorem 10.8. *$\mathcal{A}_1$ is a 1-approximation algorithm, i.e.*

$$c(\mathcal{H}_{alg}) < 2 \cdot c(\mathcal{H}_{opt}).$$

Indeed. H_{opt} contains a Hamiltonian path, a special spanning tree. Hence $c(T_{\text{alg}}) \leq c(H_{\text{opt}})$. On the other side $c(\mathcal{H}_{\text{alg}}) \leq c(\mathcal{W}_{\text{alg}}) = 2c(T_{\text{alg}})$.

Easy to realize that there is room to improve the algorithm. Taking twice the edges of T_{alg} is very 'childish' way to obtain an Eulerian graph (achieve even degrees). We can do better: in the complete graph we can take a "cheap" perfect matching for the vertices of odd degree. Adding these matching edges to T_{alg} gives us an Eulerian graph. We exhibit an improved algorithm:

$\mathcal{A}_2$ (Christofides (1976)): Given $G = K_n$, a Δ-cost function on $E(G)$.

- *(Spanning tree step): Find a minimum cost spanning tree T_{alg} of G. // We can do it by Kruskal algorithm.*
- *(Odd degree step): Determine the set of vertices of T_{alg} with odd degree in T_{alg}. Let O_{alg} be this set. // Note that $|O_{alg}|$ is even.*
- *(Matching step): Find the minimum cost perfect matching M_{alg} in $K_n|_{O_{alg}}$. // This step requires an extension of Edmonds algorithm for the weighted case. This can be done efficiently.*
- *(Walk step): Use T_{alg} and M_{alg} to construct a connected Eulerian multigraph (take disjoint union of the two edge set). Find a closed Eulerian tour in it and project to K_n to obtain $\mathcal{W}_{alg}$ closed walk, that visits every vertex. // See Observation 2.*
- *(Simplification step): From $\mathcal{W}_{alg}$ construct a Hamiltonian cyle $\mathcal{H}_{alg}$, and output it. // See Observation 1.*

The costly (but polynomial) maximum weigthed matching algorithm used by Christofides provides a significant theoretical improvement.

Theorem 10.9 (Christofides)**.** *$\mathcal{A}_2$ is a 1/2-approximation algorithm, i.e.*

$$c(\mathcal{H}_{alg}) \leq \frac{3}{2} \cdot c(\mathcal{H}_{opt}).$$

Proof. We have $c(T_{\text{alg}}) \leq c(\mathcal{H}_{\text{opt}})$ and $c(\mathcal{H}_{\text{alg}}) \leq c(\mathcal{W}_{\text{alg}}) = c(T_{\text{alg}}) + c(M_{\text{alg}})$. We need to prove that $2c(M_{\text{alg}}) \leq c(\mathcal{H}_{\text{opt}})$.

As walking along $\mathcal{H}_{\text{opt}}$ we see a circular ordering of O: $v_1, v_2, \ldots, v_{|O|}$ (the indices reflect this order). This circular ordering defines two perfect matchings of O. The first $M_1 : v_1v_2$, v_3v_4, …, $v_{|O|-1}v_{|O|}$, and the second one $M_2 : v_2v_3$, v_4v_5, …, $v_{|O|}v_1$.

When walking along $\mathcal{H}_{\text{opt}}$ as we go from v_i to v_{i+1} we pass an arc α_i. Using the triangle inequality we have

$$c(M_1) \leq c(\alpha_1) + c(\alpha_3) + \ldots + c(\alpha_{|O|-1}),$$

$$c(M_2) \leq c(\alpha_2) + c(\alpha_4) + \ldots + c(\alpha_{|O|}).$$

The sum of these inequalities is

$$c(M_1) + c(M_2) \leq c(\mathcal{H}_{\text{opt}}).$$

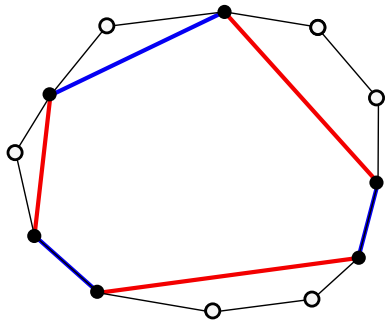

Figure 10.2: The cycle on the figure is a Hamiltonian cycle of K_n, hence it contains all the vertices. The elements of O are typified by black vertices. The red and blue edges form M_1 and M_2.

The (Matching step) of our algorithm guarantees that $c(M_{alg}) \leq c(M_1), c(M_2)$. We are done. □

Chapter 11

Extremal graph theory

11.1 Independent sets and cliques

Definition. Let G be a simple graph. The set $H \subset V(G)$ is an independent set in G, if $e(G[H]) = 0$, that is, if there is no edge between vertices of H in G. Notation:

$$\alpha(G) = \max\{|H| : e(G[H]) = 0\},$$

so $\alpha(G)$ denotes the cardinality of the largest independent set of G. We say that $H \subset V(G)$ is a clique, if $e(G[H]) = \binom{|H|}{2}$, that is, any two vertices of H are adjacent in G. Notation:

$$\omega(G) = \max\left\{|H| : e(G[H]) = \binom{|H|}{2}\right\},$$

so $\omega(G)$ denotes the cardinality of the largest clique of G.

Remark 11.1. It is easy to see, that $\alpha(G) = \omega(\overline{G})$.

As with some other important graph parameters, computing the value of $\alpha(G)$ and $\omega(G)$ are NP-complete problems. In order to get some estimation we may use a greedy algorithm.

Greedy algorithm for lower bounding $\alpha(G)$

1. fix an arbitrary ordering π of the vertices of G
2. $I = \emptyset$, $T = V(G)$
3. UNTIL $T \neq \emptyset$
   (a) let $x \in T$ be the first vertex of T according to π
   (b) let $I = I + x$ and $T = T - x - N(x)$
4. output the independent set I

One can easily see, that I must be an independent set, since after including any vertex x in I, we immediately delete all its neighbors from T. Hence, we can never choose a vertex y for I which is adjacent to any vertex already in I.

Since at step 3.*b* we delete at most $\Delta(G)+1$ vertices, we must always have

$$\alpha(G) \geq \frac{|G|}{\Delta(G)+1}.$$

In general more is true. The theorem below was discovered by Caro, and independently, by Wei.

Theorem 11.2 (Caro, Wei)**.** *We have that*

$$\alpha(G) \geq \sum_{v \in V(G)} \frac{1}{d(v)+1}.$$

Proof. One can prove this theorem by analyzing the (deterministic) greedy algorithm above. Instead we choose a probabilistic approach, a randomized version of the above greedy algorithm, which perhaps gives some insight to randomized algorithms as well.

First, we choose the ordering π uniformly at random from the $n!$ possible orderings of $V(G)$. Next we define indicator random variables. For every $v \in V(G)$ we let X_v denote the random variable which is 1, if v precedes all of its neighbors in the random ordering; otherwise we let $X_v = 0$. We need one more random variable:

$$X = \sum_{v \in V(G)} X_v.$$

It is easy to see, that X is a lower bound for the cardinality of the independent set what is found by the greedy algorithm if the vertices are ordered according to π. Hence, $\mathbb{E}[X]$, the expected (loosely speaking, average) value of X, is a lower bound for $\alpha(G)$.

By linearity of expectation we have

$$\mathbb{E}[X] = \sum_{v \in V(G)} \mathbb{E}[X_v].$$

We claim that $\mathbb{E}[X_v] = 1/(d(v)+1)$. For proving this notice that for determining X_v we don't need to consider other vertices than v and its neighbors $N(v)$. Altogether there are $(d(v)+1)!$ orderings of these vertices, out of them there are exactly $d(v)!$, in which v precedes all its neighbors. Since every ordering π is equally possible, we have that the probability that v precedes all its neighbors is exactly

$$\Pr(X_v = 1) = \frac{d(v)!}{(d(v)+1)!} = \frac{1}{d(v)+1}.$$

Hence,

$$\mathbb{E}[X_v] = 1 \cdot \Pr(X_v = 1) + 0 \cdot \Pr(X_v = 0) = \frac{1}{d(v)+1},$$

which finishes the proof of the theorem. □

Corollary 11.3. *Let $\overline{d}$ denote the average degree of an n-vertex graph G. Then we have that*

$$\alpha(G) \geq \frac{n}{\overline{d}+1}.$$

Proof. By Jensen's inequlity, we have

$$\sum_{v \in V(G)} \frac{1}{d(v)+1} \geq \sum_{v \in V(G)} \frac{1}{\overline{d}+1},$$

which proves what was desired. □

11.2 Turán's theorem

Definition. Let $r, n \in \mathbb{N}$ such that $2 \leq r \leq n$. The $T_{n,r}$ Turán[1] graph consists of r disjoint sets: $A_1, \ldots, A_r$, where $||A_i| - |A_j|| \leq 1$. Note that if r divides n, then all the A_i sets have equal cardinality. The edges of $T_{n,r}$ are as follows: if $1 \leq i \neq j \leq r$, then every $x \in A_i$ is adjacent to every $y \in A_j$. There are no other edges.

It is easy to see that while a $T_{n,r}$ graph contains plenty of cliques on r vertices, it has no clique on $r+1$ vertices. We also say that the $T_{n,r}$ graphs are K_{r+1}-free. The special case $r = 2$ of the theorem below was proved by Mantel in 1909, the general case was proved by Turán.

Theorem 11.4 (Turán (1941))**.** *Let $r, n \in \mathbb{N}$ with $2 \leq r \leq n$. If G is an n-vertex simple K_{r+1}-free graph, then*

$$e(G) \leq e(T_{n,r}).$$

Before proving this result, let us mention, that sometimes a slightly weaker corollary of the above theorem is called Turán's theorem. We will prove the the theorem and the corollary, using two different methods.

Corollary 11.5. *Let $r, n \in \mathbb{N}$ with $2 \leq r \leq n$. If G is an n-vertex simple K_{r+1}-free graph, then*

$$e(G) \leq \left(1 - \frac{1}{r}\right) \frac{n^2}{2}.$$

Proof. (of the corollary) We will use Corollary 11.3. Notice that if G is K_{r+1}-free, then $\omega(G) \leq r$. Hence, $\alpha(\overline{G}) \leq r$. By Corollary 11.3 we have that

$$\frac{n}{d'+1} \leq r,$$

where d' denotes the average degree of $\overline{G}$. This implies that $d' \geq \frac{n}{r} - 1$, hence,

$$e(\overline{G}) \geq \left(\frac{n}{r} - 1\right) \frac{n}{2}.$$

[1]These graphs are named after their discoverer, Pál Turán, a famous hungarian mathematician of the 20th century. Turán mostly worked in number theory, but made excursions into graph theory.

Since $e(G) + e(\overline{G}) = \binom{n}{2}$, we get that

$$e(G) \leq \binom{n}{2} - \left(\frac{n}{r} - 1\right)\frac{n}{2} = \left(1 - \frac{1}{r}\right)\frac{n^2}{2},$$

proving what was desired. □

Now let us consider the proof of Theorem 11.4. This theorem has several proofs, we chose the one given by Zykov, as it includes an important technique.

Proof. (of Theorem 11.4:) The proof is by induction on r. The base case is $r = 2$. Let $v \in V(G)$ be a vertex having $d(v) = \Delta(G)$. Since G is K_3-free, the neighborhood $N(v)$ has no edges. Consider the following bipartite graph H. Its vertex set is $V(G)$, which we divide into two vertex classes, $N(v)$ and $V(G) - N(v)$, the latter contains v. We include every edge between the two vertex classes.

The crucial observation is that $d_G(u) \leq d_H(u)$ for every $u \in V(G)$. This holds for v since its neighborhood has not changed, and also holds for every other vertex of $V(G) - N(v)$, since in H each such vertex has degree $\Delta(G)$. Suppose that $u \in N(v)$. Using that $N(v)$ is an independent set, u could have degree at most $n - 1 - d(v)$ in G, and this is exactly its degree in H. Hence, $e(G) \leq e(H)$, and clearly $e(H) \leq T_{n,2}$.

For $r \geq 3$ we use the ideas in the previous case. Let $v \in V(G)$ be a vertex having $d(v) = \Delta(G)$. Since G is K_{r+1}-free, the neighborhood $N(v)$ has at most $e(T_{d(v),r-1})$ edges. We may assume that $G[N(v)]$ is Turán graph $T_{d(v),r-1}$, as this has the most edges by the induction hypothesis.

Consider the following r-partite graph H. Its vertex set is $V(G)$, which we divide into r vertex classes. The first $r - 1$ classes are the vertex classes of $T_{d(v),r-1}$ on $N(v)$, the rth one is $V(G) - N(v)$. We include every edge between any two different vertex classes. As before, one can easily check that for every $u \in V(G)$ we have $d_G(u) \leq d_H(u)$. There is only one final step left: to show that $e(H) \leq e(T_{n,r})$. We leave this for the reader. □

11.3 Ramsey theory

The area is named after Frank Plumpton Ramsey (1903-1930), a british mathematician and economist. The whole area was initiated by his very influential paper "On a problem of formal logic." In a sense Ramsey theory says, that some kind of order is unavoidable even in "chaos."

We will formulate his question in modern language. The question is about edge-colorings of infinite complete graphs. For a warm-up we discuss the finite version first. Let $n \geq 2$ be a natural number, and consider K_n, the complete graph on n vertices. Color every edge of K_n either red or blue. Call a subset S of its vertices monochromatic, if every edge going in between the vertices of S has the same color. How large a monochromatic set can we avoid? Clearly, even coloring a K_2 (an edge) we are going to have a monochromatic set on 2 vertices.

What if $|S| = 3$? One can color a K_5 easily so that there is no monochromatic triangle in it. Just decompose the edges of the K_5 into two cycles of length 5 (two

edge-disjoint Hamilton cycles), color the edges of the first one blue, the edges of the second one red.

However, when we color the edges of a K_6, we will always have a monochromatic triangle. Take an arbitrary vertex u. It has degree 5. It has at least 3 incident edges of the same color, say, this is blue. If in the blue neighborhood of u there is at least one blue edge, we obtain a blue triangle. If every edge is red in this neighborhood, then we get a red triangle.

By now it has probably become clear that the order we are looking for is a large monochromatic set of the complete graph. Let us consider $K_{\mathbb{N}}$, the infinite complete graph with vertex set $\mathbb{N}$, in which every two natural numbers are adjacent. Color the edges of red and blue. Can we avoid to have an infinite monochromatic set?

Theorem 11.6 (Ramsey (1930))**.** *Whenever the edges of the infinite complete graph $K_{\mathbb{N}}$ are colored red and blue, there always exists an infinite monochromatic subset $S \subset \mathbb{N}$.*

Proof. Pick an arbitrary $x_1 \in \mathbb{N}$. Then there exists an infinite set $A_1 \subset \mathbb{N} - x_1$ such that x_1a for all $a \in A_1$ have the same color, say, c_1 (of course, here c_1 is either red or blue). Next pick an $x_2 \in A_1$. Then there exists an infinite set $A_2 \subset A_1 - x_2$ such that x_2a for all $a \in A_2$ have the same color, say, c_2.

Repeating this method one can obtain an infinite sequence $x_1, x_2, x_3, \ldots$ of numbers and an infinite sequence $c_1, c_2, c_3, \ldots$ of colors such that x_ix_j has color c_i whenever $i < j$. Each c_i is either red or blue. Hence, infinitely many of the c_is are the same, say, $c_{i_1} = c_{i_2} = c_{i_3} = \ldots$. where $i_1 < i_2 < i_3 < \ldots$. Therefore we can take $S = \{x_{i_1}, x_{i_2}, x_{i_3}, \ldots\}$. □

Remark 11.7. It is easy to see that the same proof applies for any number $k \in \mathbb{N}$ of colors. Hence, if the edges of $K_{\mathbb{N}}$ are colored by k colors, one can always find an infinite monochromatic set $S \subset \mathbb{N}$.

Remark 11.8. Let $q \in \mathbb{N}$, and set $n = 4^q$. Color the edges of a K_n. The proof method of Ramsey's theorem works, and shows that we cannot avoid to have a monochromatic set of cardinality at least q.

Definition. Let $p, q \in \mathbb{N}$ with $p, q \geq 2$. The Ramsey number $R(p, q)$ denotes the smallest integer n such that no matter how one colors the edges of K_n by red and blue, we have either blue set of cardinality p, or a red set of cardinality q. The diagonal Ramsey numbers are those with $p = q$.

One can formulate Ramsey's question in another, equivalent way. Color the edges of a K_n red and blue. Let the graph G be determined by the blue edges. Then $\overline{G}$ includes the red edges. In this formulation the largest blue set has cardinality $\omega(G)$, the largest red set has cardinality $\omega(\overline{G}) = \alpha(G)$.

Using Remark 11.8 we get the following: if G is a simple graph on n vertices, then $\max\{\omega(G), \alpha(G)\} \geq \frac{1}{2}\log_2 n$. This is not the best upper bound we know. We prove a slightly better bound below.

Theorem 11.9. *We have $R(p, q) \leq R(p-1, q) + R(p, q-1)$.*

Proof. Let $n = R(p-1,q) + R(p,q-1)$. Let x be an arbitrary vertex of K_n. If its blue neighborhood B has at least $R(p-1,q)$ vertices, then we have two cases: either B contains a blue set of size at least $p-1$, and so with x we have the blue set of size p, or the largest blue subset of B has at most $p-2$ vertices. In the latter case we must have a red set of size at least q in B. If R, the red neighborhood of x has at least $R(p,q-1)$ vertices, then using a very similar argument we get either a blue set of size p, or a red set of size q. Observe, that we cannot have $|B| < R(p-1,q)$ and $|R| < R(p,q-1)$ at the same time, so we proved what was desired. □

Using Theorem 11.9 and that $R(p,2) = R(2,p) = p$, one can obtain that

$$R(p,q) \leq \binom{p+q-2}{p-1}.$$

In particular, we get the following upper bound for the diagonal Ramsey numbers:

$$R(p,p) \leq \binom{2p-2}{p-1} = c\frac{4^p}{\sqrt{p}}$$

for some $c > 0$ constant.

Below we show a lower bound for the diagonal Ramsey numbers by Pál Erdős. His proof played an important role in developing his probabilistic method, which has since become one of the most powerful tools in combinatorics.

Theorem 11.10 (Erdős (1947))**.** $R(p,p) \geq (1+o(1))\frac{1}{e\sqrt{2}}p2^{p/2}$.

Proof. We will show that there exists a graph $G = (V,E)$ on n vertices for which $\alpha(G), \omega(G) > p$, here n is the largest integer for which the following inequality holds:

$$\binom{n}{p}2^{1-\binom{p}{2}} < 1.$$

Our "construction" of G is probabilistic. Let $V = \{1,2,\ldots,n\}$. For every pair $ij \in \binom{V}{2}$ we randomly, independently flip a coin. If the result is heads, we include ij in E, otherwise $ij \notin E$.

Let $S \subset V$ be a set with $|S| = p$. We compute the probability that $G[S]$ is a clique or an independent set:

$$\Pr\left(e(G[S]) = \binom{p}{2}\right) = \Pr\left(e(G[S]) = 0\right) = 2^{-\binom{p}{2}}.$$

Let A_S denote the event that $G[S]$ is either and independent set or a clique. Using the above we have

$$\Pr(A_S) = 2^{1-\binom{p}{2}}.$$

Finally we will estimate the probability that for some set S with $|S| = p$ the event A_S holds:

$$\Pr\left(\bigcup_{S:|S|=p} A_S\right) \leq \sum_{S:|S|=p} \Pr(A_S) = \binom{n}{p}2^{1-\binom{p}{2}}.$$

Here we used the so called union bound for upper bounding the probability of the union of some events, and that the number of p element subsets of V is $\binom{n}{p}$. Since by the definition of n the probability that neither of the A_S events hold is positive, there must exist a graph G on n vertices with the claimed properties. Using the Stirling formula for approximating the factorials gives the bound of the theorem. $\square$

The above bound of Erdős, although old, is still only a constant factor away from the currently best bound. Estimating the Ramsey numbers is a notoriously hard problem, for most of the cases the current best known upper and lower bounds for the diagonal Ramsey numbers are far away from each other, even asymptotically.

Let us mention that Ramsey type results have several applications in combinatorics, combinatorial geometry or computer science. It is still a rapidly growing area with lots of ramifications, with many interesting and deep questions and results.