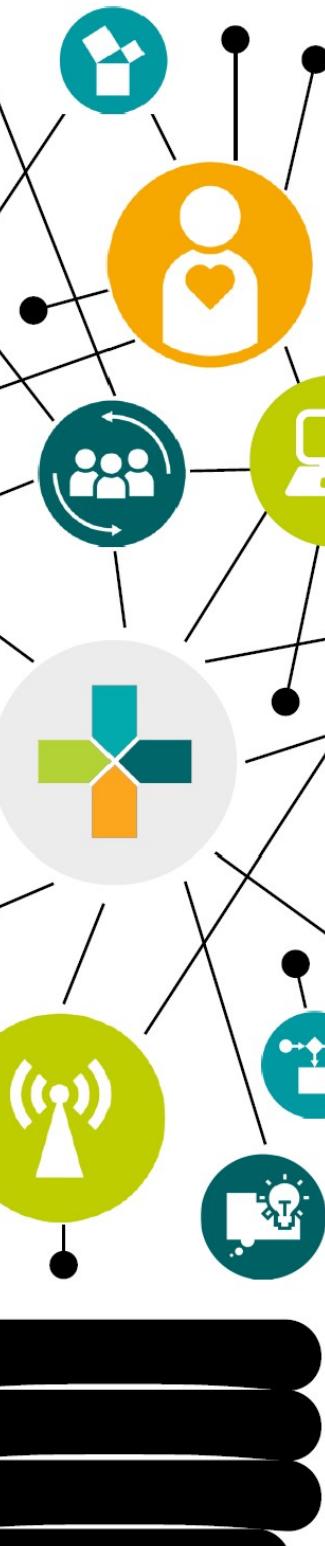


Combinatorial Optimization

- Optimierung B

Prof. Dr. Christina Büsing

Winter Term 2025/26





Contents

1. Basics and Eulerian Graphs	9
1.1. The Seven Bridges of Königsberg	9
1.2. Fundamental Properties and further Terms	17
1.3. Special Graph Classes	21
2. Minimum Spanning Tree	33
2.1. Motivation and Definition	33
2.2. Optimality Criterion	34
2.3. Kruskal's and Prim's Algorithms	41
3. Directed Graphs or Digraphs	47
4. Shortest Path Problems	57
4.1. An Optimality Criterion for Shortest Paths	57
4.2. Dijkstra's Algorithm	64
4.3. Moore-Bellman-Ford Algorithm	68
4.4. All pair shortest paths	71
5. Maximum (s, t)-Flows	77
5.1. (s, t) -Flows and the Decomposition Theorem	77
5.2. Ford-Fulkerson Algorithm	82
5.3. Max-Flow-Min-Cut Theorem	88
5.4. The Edmonds-Karp Algorithm and Dinic Algorithm with blocking flows	93
6. Minimum Cost Flow	103
6.1. Problem Formulation	103
6.2. An Optimality Criterion	106
6.3. Successive Shortest Path Algorithm	115
7. Maximum Matchings	121
7.1. Optimality Criteria and Edmonds Matching Algorithms	121
7.2. Edmonds Matching Algorithms	124
7.3. Covering Problems and Upper Bounds	134
7.4. Matchings in Bipartite Graphs	145
7.5. Postperson Problem	152
8. Linear Programming	157
8.1. Stigler's Diet Problem and Basics	157
8.2. Algebraic Interpretation of Linear Programs	166
8.2.1. The Standard Form	166
8.2.2. Bases and an Optimality Criterion	170
8.3. The Simplex Algorithm	180
8.4. Duality for Linear Programs	187
9. Basics of Integer Programming	195
9.1. Motivation, Definitions and Modeling Power	195
9.2. IPs, LPs and integer Polytopes	202
9.3. Totally unimodular Matrices	206
9.4. Branch and Bound	216
10. Computational Complexity Theory: P vs. NP	221
10.1. Encodings and Models of Computation	222
10.2. Complexity Classes	229
10.3. NP-complete problems	233

10.4. Further Complexity Results	249
11. Approximation Algorithms	257
11.1. Constant Approximation Ratio	257
11.2. Absolute Approximation Algorithms	266
11.3. Approximation Schemes	278
A. Mathematik lesen, lernen und schreiben	291
A.1. Struktur mathematischer Texte und sie verstehen	292
A.2. Überprüfung Ihres Verständnisses	296
A.3. Aufbau eines Papers/einer mathematischen Publikation	298
A.4. Schritte zum Erarbeiten eines mathematischen Textes	299
A.5. Beweistechniken	300
A.6. Beispiele und Gegenbeispiele	302
A.7. Wie man (mathematische) Probleme löst	303
A.8. Mathematik schreiben	304



Foreword to Discrete Optimization

Motivation

- We always need to make decisions
 - Where to go? What to eat? Which course to choose?
- Normally, there are several options and we can evaluate these options
- \Rightarrow objective is to find the best one (*optimization problem*)
- In discrete optimization we focus on optimization problems where the decision variables are discrete (e.g., in $\{0, 1\}$) and not continuous (e.g., in $[0, 1]$)
- These kind of problems can be found everywhere:



Logistics: Where should I route my trucks in order to transport all goods as quick as possible?



Public transportation: What is an optimal time-schedule for the trains to transport all passengers?



Communication networks: Where to allocate different mobile stations in order to obtain a good covering?



Warehousing: Where to store which goods in order to obtain them as fast as possible?



Health Care: When to schedule which surgery such that the operation theater is used all the time and in parallel we are able to cope with emergencies?



Education: When shall which lecture in which lecture hall take place?

- Therefore: Discrete Optimization is a discipline that is part of computer science, mathematics and business administration
- Classically we follow the steps in the optimization cycle in order to take a problem and to find a good solutions:

Step 1: **Problem:**

- Given from application
- Theoretical problem

Step 2: **Mathematical Formulation/Model of problem:**

- Identify the decision problem
- Obtain an initial statement of the problem with the constraints and objectives

Step 3: **Analyze structure/Design of Algorithms:**

- Analyze the structure of the problem and its complexity
- Develop different algorithmic approaches

Step 4: **Implementation of the algorithms:**

- Python, AMPL, ...
- Test the algorithms (correctness, run-time, ...)

Step 5: **Evaluate the solution:**

- Test and evaluate the solution

Step 6: **Presentation of the solution:**

- Present the solution and its implication

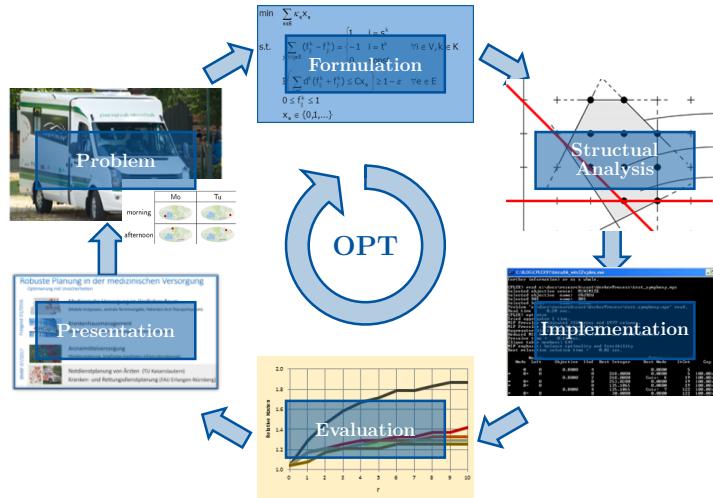
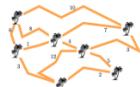


Fig. 1.: The classical process in decision making: formulate, model, solve and implement. (No. 280)

- These steps can be repeated arbitrarily often
- In this course, we provide the mathematical basis of taking the best decision in a discrete optimization setting
- Topics covered in this class:



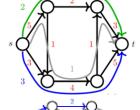
Basics and Euler: Graphs, Trees, Euler-Tour, Algorithms



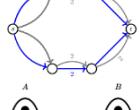
Minimum Spanning Trees: Algorithm of Kruskal, Algorithm of Prim



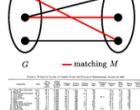
Shortest Path Problem: Dijkstras Algorithm, Bellman-Ford-Moor Algorithm



Maximum Flows: Ford&Fulkerson Algorithm, Max-Flow-Min-Cut



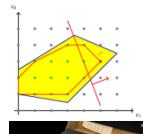
Minimum Cost Flows: cycle canceling algorithm, residual graphs



Maximum Matchings: Theorem of König, alternating paths



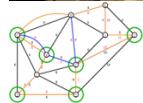
Linear Programming: Basic solutions, Simplex Algorithm, Duality



Integer Programming: Totally unimodular matrices, Branch&Bound



Complexity: P vs. NP, Turing Machine



Approximation Algorithms: constant approximation, double tree algorithm

How to Use the Lecture Material and Learn for this Class

Lecture Notes

- These notes are no book!
- There are great books on this topic:
 - [?] Korte, Bernhard and Vygen, Jens, "Combinatorial Optimization: Theory and Algorithms", Springer Publishing Company, Incorporated (2007)
 - [?] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald and Stein, Clifford, "Introduction to Algorithms", The MIT Press (2022)
- These notes/documents support me giving the lecture
- Definitions/Theorems/Algorithms are formally given
- Sometimes definition are given multiple times at different parts of the notes
- Motivations/ideas are given in key points
- Proofs are sketched in key points
 - During class: we discuss why the proofs are correct
 - Take notes!

Animation

- Visualization of concepts/algorithms/proofs
- Use them for repetition
- Don't print the document
- No. xxx connects the animation document with the lecture notes

Learning discrete Optimization

- "No one learns to think by reading the ready-written thoughts of others, but by thinking for himself." (Mihai Eminescu)
- Go through the lecture especially the proofs
 - Rethink the arguments by yourself
 - Why can we follow the next step from what we saw before?
- Theorem/Lemma: Given property 1,2,3 \Rightarrow conclusion
 - What if we drop 1,2,3. Why can't we prove the same?
- Learn the terms like vocabularies

- Use the terms and apply them to small examples
- Repeat the algorithms by yourself on small examples
- Do the exercises
- Have fun and enjoy learning great theory that can be applied to many practical problems



1. Basics and Eulerian Graphs

1.1. The Seven Bridges of Königsberg

Problem and mathematical model

- In 1736 Leonard Euler was confronted in Königsberg with the “seven Bridges of Königsberg” problem:
 - Does a Sunday stroll exist in Königsberg
 - That crosses every bridge exactly once?
- Considering this problem, Euler invented graph theory

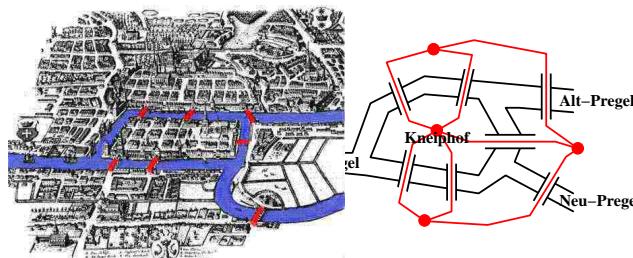


Fig. 1.1.: Map of Königsberg in 1736 and an abstract version as a graph. (No. 516)

- Euler’s idea of mathematical modeling:
 - Islands correspond to vertices
 - Bridges correspond to edges
 - Stroll corresponds to a sequence of edges also called path
 - “Sunday Stroll” is a path in the graph that contains all edges exactly once and ends in the same vertex as it starts
- We start with formal definitions of a graph and the other components of the problem

Definition 1 (Graph). An *undirected graph* $G = (V, E)$ consists of a finite set of vertices V and a multiset of edges $E \subseteq \{\{u, v\} \mid u, v \in V\}$. Two vertices u, v are *adjacent*, if an edge $\{u, v\}$ in E exists. An edge $\{u, v\}$ is *incident* to the vertices u and v .

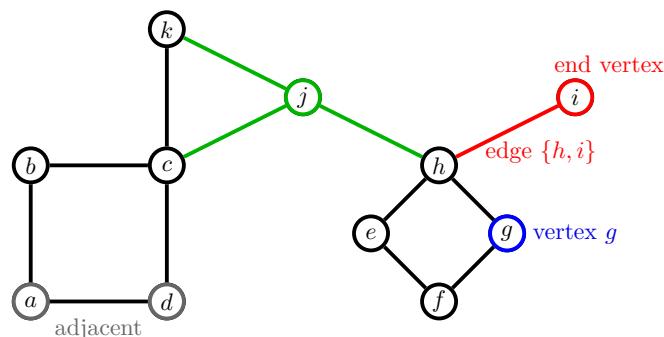


Fig. 1.2.: Representation of a graph (No. 515)

- **Note:** We almost always consider the graphical visualization of a graph
- Let's keep approaching the seven bridge problem

Definition 2 (Path). Let $s, t \in V$ be two vertices in a graph $G = (V, E)$. A *path* from s to t , also denoted as (s, t) -path, is an edge sequence $e_1 e_2 \dots e_k$ starting in s and ending in t , where for every edge e_i , $i = 1, \dots, k$, the end vertex of an edge corresponds to the start vertex of the successor edge, i.e., $e_i = v_i v_{i+1} \in E$, $i = 1, \dots, k$ with $v_1 = s$ and $v_{k+1} = t$.

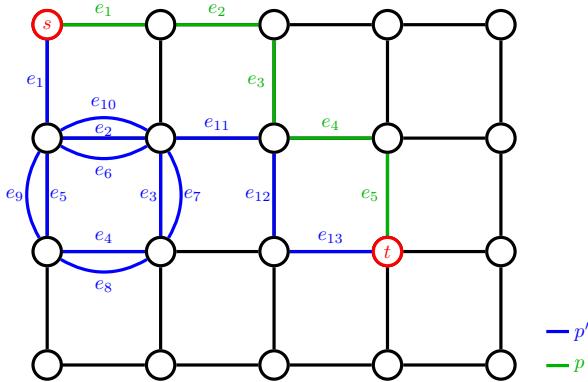


Fig. 1.3.: Paths in a graph (No. 753)

- Often, we denote a path p by a sequence of vertices, i.e., $p = v_1 v_2 \dots v_k v_{k+1}$
- The vertex v_{i-1} is called *predecessor* of v_i and v_{i+1} is the *successor* of the vertex v_i
- We denote with $V(p)$ and $E(p)$ the set of vertices and edges of a path p in a considered graph $G = (V, E)$
- Special paths:
 - *Elementary path* is a path without vertex repetition
 - *Simple path* is a path without edge repetition and different start- and endvertices
 - *Eulerian path* is a path which visits every edge of the considered graph exactly once

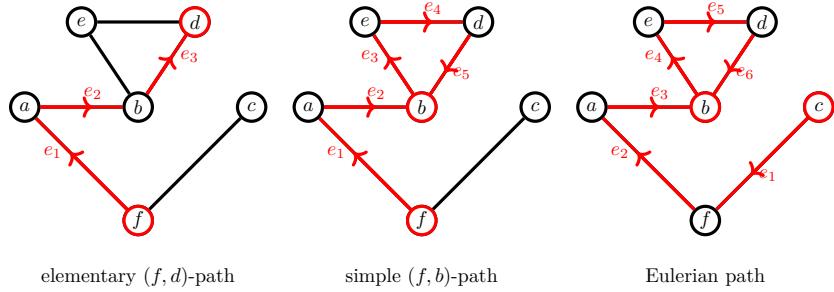


Fig. 1.4.: Special paths (No. 517)

- A path whose start vertex is the same as the end vertex is called a *cycle*

- **Note:** If not stated otherwise, we consider *simple cycles*, i.e., cycles that contain every edge at most once.
- A cycle that visits every edge exactly once is called *Eulerian cycle* or *Eulerian tour*

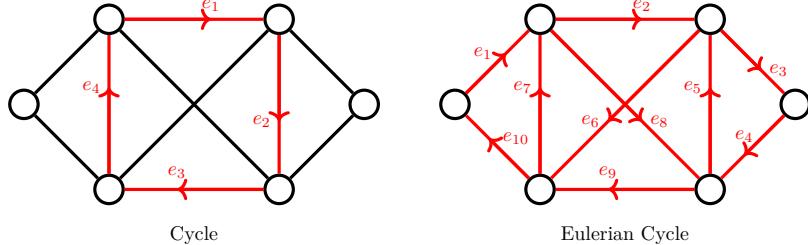


Fig. 1.5.: Cycle vs. Eulerian cycle (No. 518)

Definition 3. Eulerian cycle problem

Given: Undirected graph $G = (V, E)$

Find: An Eulerian cycle if one exists, i.e., a cycle that contains every edge exactly once.

Mathematical analysis and structural properties

- When does an Eulerian cycle exist?

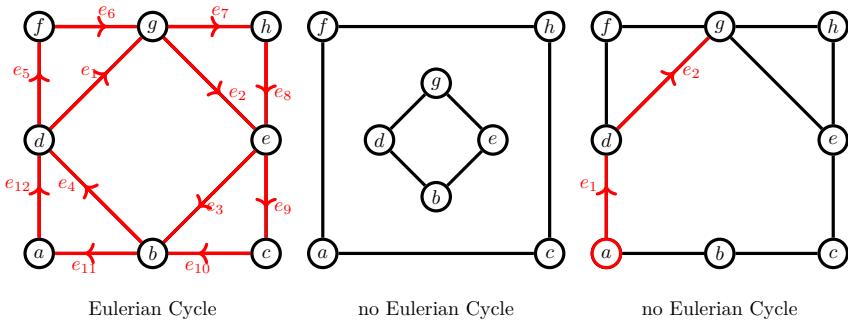


Fig. 1.6.: Eulerian cycle? (No. 519)

- First necessary condition: we need to get from any vertex to any other

Definition 4 (connected graph). Graph $G = (V, E)$ is called *connected* if either $|V| = 1$ or there is a (u, v) -path for all $u, v \in V$.

- Connectivity is not sufficient
- Consider the number of edges that contain a certain vertex

Definition 5 (Degree of a vertex). The *degree* $d(v) \in \mathbb{N}_0$ of a vertex $v \in V$ denotes the number of edges which are incident to v . An edge $e = uv$ is *incident* to u and v . A vertex with $d(v) = 0$ is called an *isolated vertex*.

- If an Eulerian cycle exists the degree is even

Theorem 6 (Euler 1736, Hierholzer 1873). *Let G be a graph without an isolated vertex. Then the graph G has an Eulerian cycle if and only if G is connected and the degree of each vertex is even.*

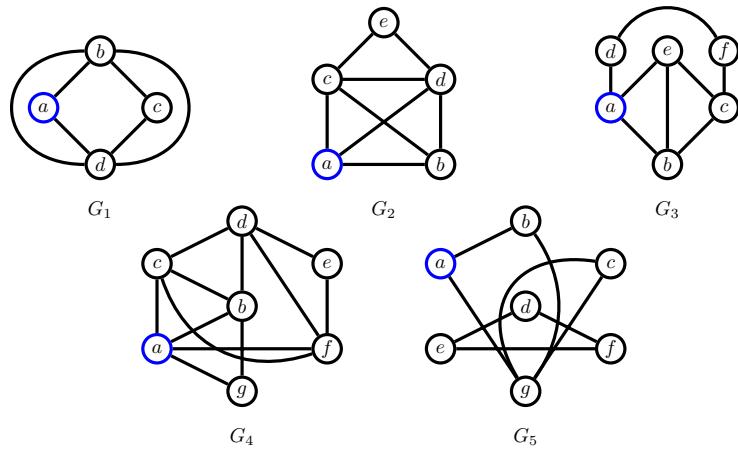


Fig. 1.7.: Does an Euler Cycle exist? (No. 948)

- In order to prove Theorem 6, we need the following property:

Lemma 7 (Closed-walk). *Let $G = (V, E)$ be a graph such that the degree of every vertex $v \in V$ is even. Let p be a simple path which starts in v_0 and ends in v_k . If every incident edge of v_k is also a part of p , then p is a cycle, i.e., $v_0 = v_k$.*

Proof. Counting the edges

- Let $p = v_0v_1 \dots v_k$ be a simple path such that all incident edges of v_k are a part of p
- $\Rightarrow p$ cannot be extended
- Assume $v_0 \neq v_k$
- Since v_k is the end vertex, an odd number of incident edges of v_k are a part of p
- By assumption there exists an edge e' that is incident to v_k and which is not a part of p
- \Rightarrow We can extend path p by e' without using an edge twice \Rightarrow contradiction

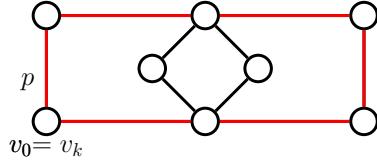


Fig. 1.8.: Path p isn't extendable in vertex v_k , however this doesn't mean that p is an Eulerian cycle (No. 531)

□

- We now proceed with proof of Euler's Theorem

Proof. Use the Closed-walk Lemma 7

- (\Rightarrow): Given: There is an Eulerian Cycle
Prove: The graph is connected and every vertex has an even degree
Let T be an Eulerian cycle \Rightarrow the graph is connected
- Let v be a vertex which is crossed k times by T
- Every time the cycle crosses v two incident edges are visited
- $\Rightarrow 2k$ edges are incident to v , i.e., $d(v) = 2k$
- \Rightarrow all vertices have an even degree

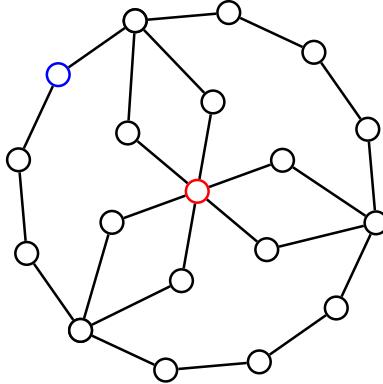


Fig. 1.9.: Every vertex has an even degree (No. 653)

- (\Leftarrow): Given: The graph is connected and every vertex has an even degree
Prove: There is an Eulerian Cycle
- Let G be connected and let $d(v)$ be even $\forall v \in V$
- Let $p = v_0v_1 \dots v_k$ be a simple path in G with maximum length, i.e., there is no longer simple path in G
- $\Rightarrow p$ cannot be extended, i.e., all incident edges of v_k are part of p
- $\Rightarrow p$ is a cycle, i.e., $v_0 = v_k$ (Closed-walk Lemma 7)
- Case 1: p uses every edge of G
 - $\Rightarrow p$ is an Eulerian cycle
- Case 2: assume there exists an edge $e = uv \notin E(p)$
 - Case 2.1: e is incident to a vertex of p , i.e. $v = v_i \in V(p)$ for an $i = 0, \dots, k$

- We can define a path

$$p' = uv_i \dots v_k v_0 v_1 \dots v_{i-1} v_i$$

with larger length \Rightarrow contradiction

- Case 2.2: e is not incident to a vertex of p

- G is connected
- \Rightarrow there exists a path \bar{p} from v to a vertex $v_i \in V(p)$ with $E(\bar{p}) \cap E(p) = \emptyset$
- Consider the edge $e' \in E(\bar{p})$ which is incident to v_i , i.e. $e' = v'v_i$
- \Rightarrow setting as in Case 2.1 \Rightarrow contradiction

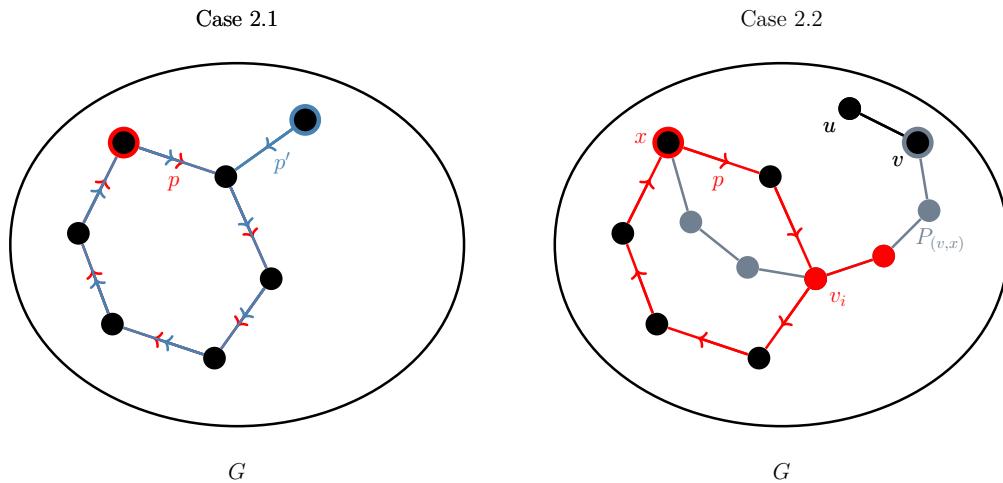


Fig. 1.10.: Proof Eulerian cycle (No. 526)

□

- **Note:** A graph which contains an Eulerian cycle is called *Eulerian graph*
- Euler only provided a proof for the necessary conditions
- With this, the seven bridges of Königsberg problem was solved
- However, if a cycle exists it is not clear how to find it

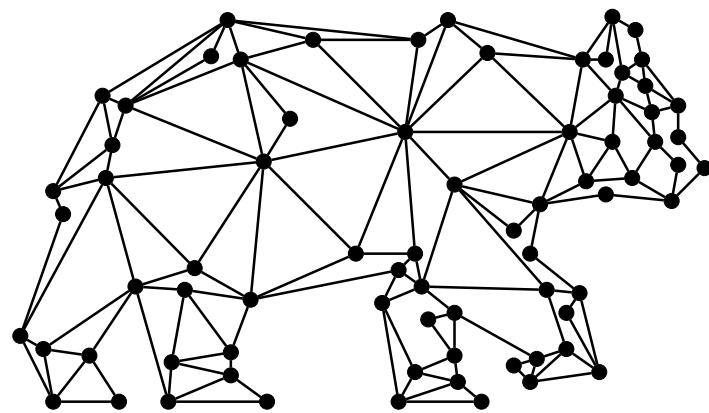


Fig. 1.11.: If the graph is big, it is difficult to construct an Eulerian cycle (No. 813)

- Hierholzer developed the first algorithm to construct an Eulerian cycle in 1871

Hierholzer's Algorithm

- *Algorithms* are finite descriptions of steps to get out of an input a desired output (most times a solution to a considered problem)
- Examples:
 - Lego construction manual
 - Recipes
 - IKEA instructions



Fig. 1.12.: Algorithms (No. 959)

- Carl Hierholzer was a German mathematician who habilitated in Königsberg
- After his death in 1871 two colleges published in 1873 his algorithm that solves the Eulerian cycle problem
- Idea of Hierholzer's algorithm
 - Choose a vertex v_0
 - Construct a cycle by choosing unused edges and mark the edges as visited
 - If the cycle doesn't contain all edges, choose a vertex on the cycle which is incident to an unused edge and construct a new cycle
 - Merge both cycles

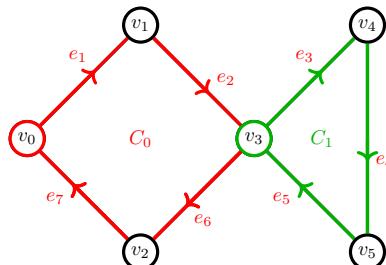


Fig. 1.13.: Hierholzer's algorithm (No. 527)

Algo. 1.1 Hierholzer's algorithm

Input: Connected graph $G = (V, E)$ with $d(v)$ even $\forall v \in V$ Output: Eulerian cycle K

Method:

- Step 1
 - Choose a vertex v_0
 - Successively select unused edges until we obtain a cycle K
 - Step 2 **If** K is an Eulerian cycle **then**
 Stop and Return K
 - Step 3
 - Set $K' = K$
 - Choose a vertex $v_i \in V(K')$ which is incident to an unused edge
 - As in Step 1, construct a cycle K'' starting from v_i with $E(K'') \cap E(K') = \emptyset$
 - Merge K' and K'' to a new cycle K as follows: pass all vertices from v_0 to v_i , pass through K'' and then pass the rest of K'
 - Go to Step 2
-

Theorem 8. Let G be an undirected, connected graph such that all vertices have an even degree. Then, Hierholzer's algorithm constructs an Eulerian cycle.

Proof. Feasibility of each step

- Step 1: Construction of a path which doesn't use an edge twice and which either is a cycle or isn't extendable
- \Rightarrow we get a cycle (Closed-walk Lemma 7)
- Step 2: Algorithm terminates if an Eulerian cycle is found
- Step 3: Prove: At the end of Step 3, we have a cycle
- *Claim 1:* If K' is not an Eulerian cycle, then $\exists v_i \in V(K')$ which is incident to $e \in E \setminus E(K')$.

Proof of Claim:

- K' isn't an Eulerian cycle, i.e. $\exists e = uv \in E(G) \setminus E(K')$
- Case 1 : $u \in V(K')$ or $v \in V(K') \Rightarrow$ claim holds true
- Case 2: $u \notin V(K')$ and $v \notin V(K')$
 - G is a connected graph
 - Find a path p from v to a vertex of K'
 - The end vertex of p corresponds to a vertex is incident to an edge $e \in E \setminus E(K')$ $\square C1$

- Due to the close-walked Lemma, K'' is a cycle

- *Claim 2:* Combining K' and K'' leads to a new cycle.

Proof of Claim

- The graph $G' = (V, E')$ with $E' = E \setminus E(K')$ has an even degree for all vertices
- Thus, G' satisfies the property of the Closed-walk Lemma 7
- By construction, K'' is a cycle (Closed-walk Lemma 7)
- Let $K' = v_0v_1 \dots v_i \dots v_0$ and $K'' = w_0w_1 \dots w_jw_0$ be two cycles with $w_0 = v_i$

- \Rightarrow

$$K := v_0 \dots v_i w_1 \dots w_j v_i \dots v_0$$

is a cycle

\square C2

- Since Step 3 is executed at most $\frac{1}{2}|E|$ times, the algorithm terminates with Step 2

\square

- Fleury introduced in 1883 another important algorithm to solve the Eulerian cycle problem

- Here, the Eulerian cycle is constructed in one run by choosing the next edge wisely

The optimization cycle

Step 1: Problem:

- Seven Bridges of Königsberg of 1736

Step 2: Mathematical Formulation/Model of problem:

- Definition of graphs and paths, the Eulerian cycle problem

Step 3: Analyze structure/Design of Algorithms:

- Euler's Theorem: \exists Eulerian cycle if and only if $d(v)$ even and G connected
- Hierholzer's algorithm

Step 4: Implementation of the algorithms:

- skipped

Step 5: Evaluate the solution:

- skipped

Step 6: Presentation of the solution:

- Euler convinced Königsberg to give up searching for a Sunday Stroll

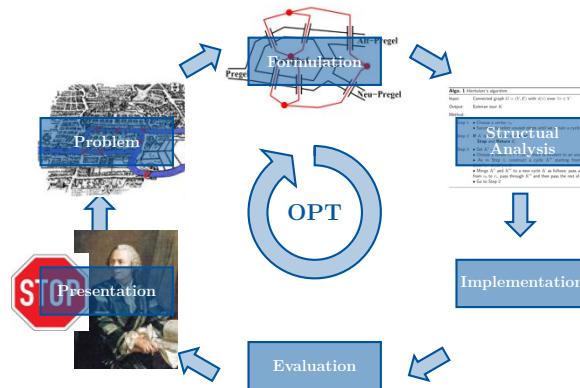


Fig. 1.14.: The optimization cycle according to the Eulerian cycle problem (No. 814)

1.2. Fundamental Properties and further Terms

- More general: a graph models relations (edges) between entities (vertices)

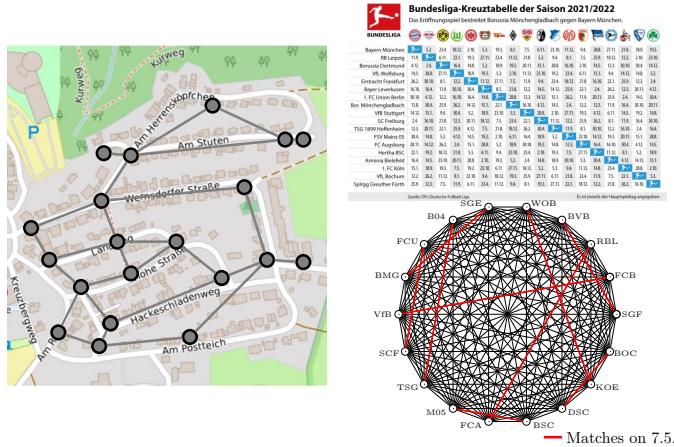


Fig. 1.15.: Graphs are powerful models (No. 815)

Reading Material

- Special edges:
 - *Loops*: start and end vertex are the same, i.e., $e = vv$, $v \in V$
 - *Parallel edges*: various edges have the same end vertices
- Special vertices:
 - Vertices v with $d(v) = 0$ are called *isolated* vertices

Definition I (Simple Graph). A *simple graph* has neither loops nor parallel edges.

- **Note:** If not stated otherwise, we consider simple graphs in this course
- Let's start with the most basic property of a graph

Lemma II (Handshaking-Lemma). *Let $G = (V, E)$ be a graph. Then*

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

Proof. Counting argument

- Every edge is counted twice, since every edge is incident to two vertices
- \Rightarrow

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

□

- A nice and often used property is the following:

Corollary III. *Let $G = (V, E)$ be a graph. Then the number of vertices with odd degree is even.*

Proof. Handshaking-Lemma

- We consider all vertices with even and odd degrees separately:

$$2|E| \stackrel{(1)}{=} \sum_{v \in V} d(v) = \sum_{\substack{v \in V \\ d(v) \text{ even}}} d(v) + \sum_{\substack{v \in V \\ d(v) \text{ odd}}} d(v)$$

(1): Handshaking Lemma ??

- $\Rightarrow \sum_{\substack{v \in V \\ d(v) \text{ odd}}} d(v)$ is even, but all summands are odd
- \Rightarrow the number of vertices with odd degree is even

□

- Often subgraphs are considered instead of the whole graph

Definition IV (Subgraph and connected component). Let $G = (V, E)$ be an undirected graph.

1. A *subgraph* of G is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. We often write $G' \subseteq G$.
2. Let $V' \subseteq V$, then $G' := (V', E')$ with $E' := \{uv \in E \mid u, v \in V'\}$ is the subgraph *induced* by V' .
3. A maximal connected, induced subgraph G' of G is called *connected component*. Maximal means that adding any vertex yields a disconnected subgraph.

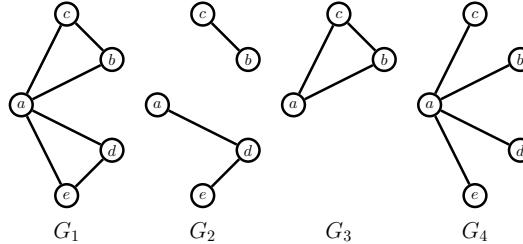


Fig. 1.16.: Subgraph (No. 524)

- An edge that separates a graph into two components is called a bridge

Definition V (bridge). Let $G = (V, E)$ be a graph. An edge e is called *bridge* if $G - e$ has one more connected component than G . With $G - e$ we denote a graph $G' = (V', E')$ with $V' = V$ and $E' = E \setminus \{e\}$.

- Bridges are easy to characterize

Theorem VI. An edge e is not a bridge if and only if it is part of a cycle.

Proof. Exercise

□

Algorithms

- Important components of our algorithms:

Input	Given instances to perform an algorithm
Example:	Graph $G = (V, E)$, edge weight $c(e) \in \mathbb{R}$, $\forall e \in E$
Output	Solution which is generated by the algorithm
Example:	(s, t) -path
$\leftarrow/=$	Assignment
Example:	$x \leftarrow 3$ means that variable x is set to value 3
//	Comments for the readers
While -Loop	While a statement is true do the following
Example:	While \exists a vertex $v \in V$ with $d(v) \geq 3$ and v is not red do
	<ul style="list-style-type: none"> Color v red

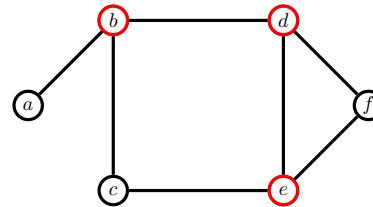


Fig. 1.17.: The algorithm colors vertices with $d(v) \geq 3$ red. (No. 521)

If-Query	If a statement is true then do the following Else do the following
Example:	Choose $v \in V$
	If $d(v) \geq 3$ then <ul style="list-style-type: none"> Color v red
	Else <ul style="list-style-type: none"> Color v blue

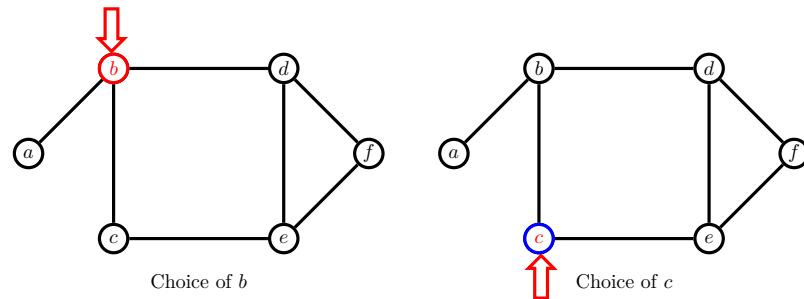
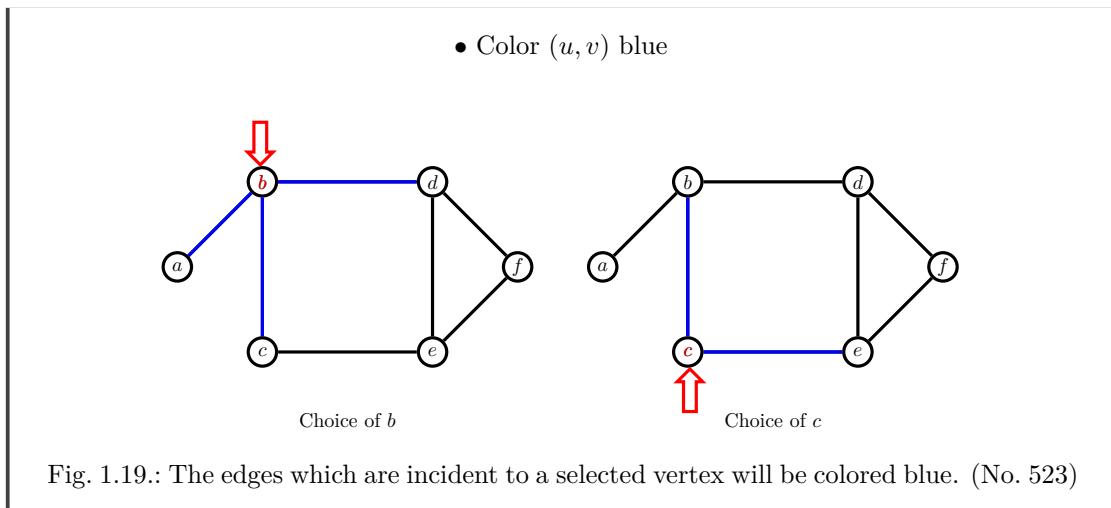


Fig. 1.18.: Depending on the choice of the vertex it will be colored red or blue. (No. 522)

For-Loop	For a set do the following
Example:	Choose $v \in V$
	For $(u, v) \in E$ do



1.3. Special Graph Classes

- Often graphs have special structures

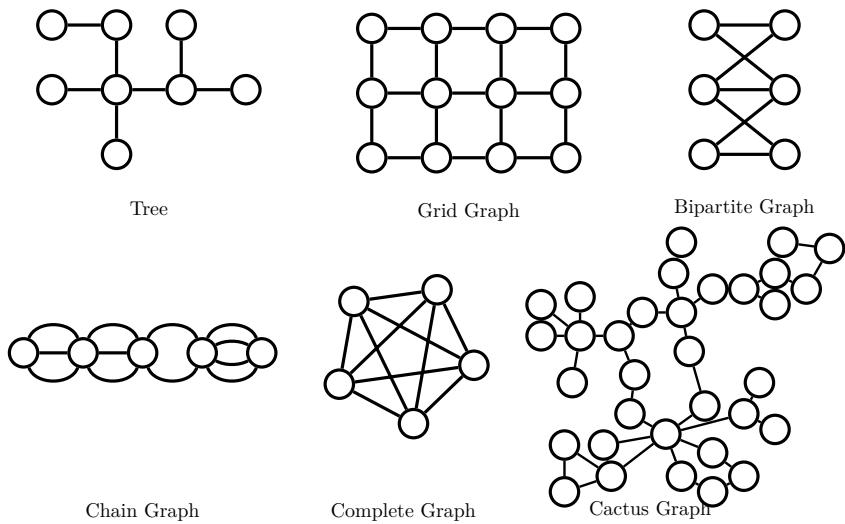


Fig. 1.20.: Different graphs (No. 754)

- According to these structures
 - algorithms may behave differently
 - problems become more easy to solve
 - or optimal solutions may have certain properties

Trees

Definition 9 (Tree). A connected graph $G = (V, E)$ is called a *tree* if it contains no cycle. The *leaves* of a tree are the vertices with degree one.

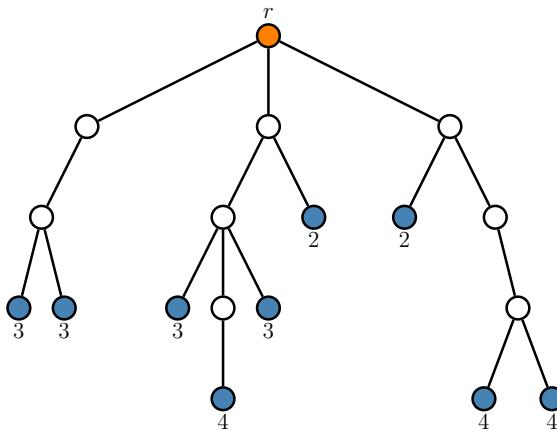


Fig. 1.21.: Tree with one root (orange) and seven leaves (blue) (No. 534)

- Trees are the basic structure of a connected graph
- They combine two opposite concepts:
 - Connection \Rightarrow (many) edges are necessary
 - No cycles \Rightarrow not too many edges are allowed
- Often, we select one arbitrary vertex and call it the *root* of the tree
- By means of a root r , we can address other vertices or leaves v with their *distances* to r , i.e., the number of edges which are between root r and vertex v in tree T . We write $\text{dist}_T(r, v)$

Lemma 10. *Let G be a tree with at least two vertices, then G has at least two leaves.*

Proof. Exercise □

- An important property, we often use for trees, is the following

Lemma 11. *Let G be a graph. Then G is a tree if and only if there exists a unique path between any two vertices in G .*

Proof. Exercise □

- A slight extension of a tree is a forest
- A *forest* is a graph whose connected components are trees.

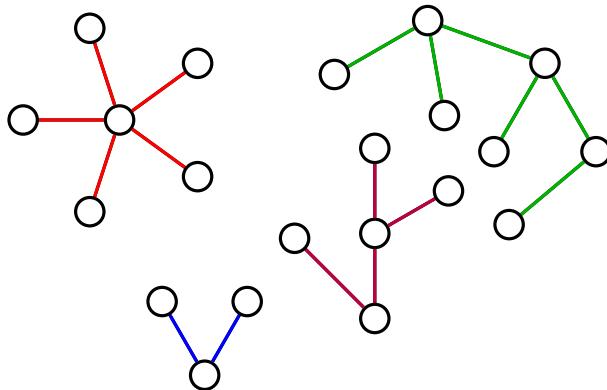
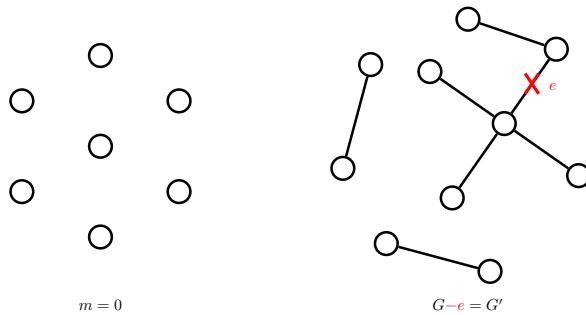


Fig. 1.22.: A forest (No. 755)

Lemma 12. Let G be a forest on n vertices with m edges and p connected components. Then $n = m + p$.

- With $G + e$ and $G - e$ we denote the graph $G = (V, E)$ with an additional edge $e \notin E$ and without the edge $e \in E$, respectively, i.e. $G + e = (V, E \cup \{e\})$ and $G - e = (V, E \setminus \{e\})$

Proof. Induction on the number of edges m

Fig. 1.23.: In forests: $n = m + p$ (No. 654)

- I.B.: $m = 0 \Rightarrow$ Each vertex is a separate connected component, i.e. $n = p$
 - I.H.: For every forest on n vertices with m edges, it holds $n = m + p$
 - I.S.: Let G be a forest on n vertices with $m + 1$ edges and p connected components
 - Delete an arbitrary edge e to obtain $G' = G - e$
 - $\Rightarrow G'$ is also a forest on n vertices with m edges and $p + 1$ connected components
 - \Rightarrow

$$n \stackrel{I.H.}{=} m + p + 1 = (m + 1) + p$$
 - By the principle of induction, $n = m + p$ is true
-
- For trees, we obtain with this a nice characterization via the number of edges

Theorem 13 (Important characteristics of trees). *Let G be a graph on n vertices. Then the following statements are equivalent:*

1. G is a tree (i.e. is connected and has no cycles).
2. G has $n - 1$ edges and no cycles.
3. G has $n - 1$ edges and is connected.

Proof. Number of connected components vs. number of edges

- (1) \Rightarrow (2):

Given: Graph G is a tree, i.e. G has no cycles and is connected by definition.

Prove: $m = n - 1$

- By Lemma 12, $n = m + p \Leftrightarrow m = n - p$
- Since $p = 1 \Rightarrow m = n - 1$

- (2) \Rightarrow (3):

Given: Graph G has no cycles and $n - 1$ edges.

Prove: $p = 1$

- Since G has no cycles, G is a forest
- Lemma 12: $m = n - p \Rightarrow p = n - m = n - (n - 1) = 1$
- $\Rightarrow G$ is connected

- (3) \Rightarrow (1):

Given: Graph G is connected with $n - 1$ edges.

Prove: G has no cycles

- Assume that G has a cycle $\Rightarrow \exists$ edge such that $G - e$ is connected as well
- Delete edges until the new graph G' has no cycles and is connected

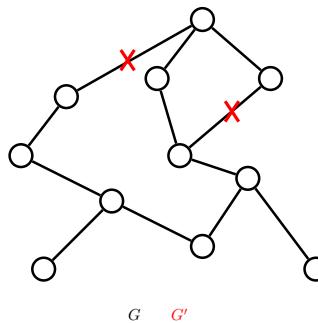


Fig. 1.24.: Delete until acyclic (No. 655)

- Set $m' = |E(G')|$
- $\Rightarrow m' < m$
- G' is a tree by construction
- $\Rightarrow m' = n - 1$ (proven above)
- However, $m = n - 1$, contradiction

□

Spanning trees

- Consider the following gold nugget search problem

Given: A building with gold nuggets in it but locked doors

Find: A way to get all rooms opening as few doors as possible

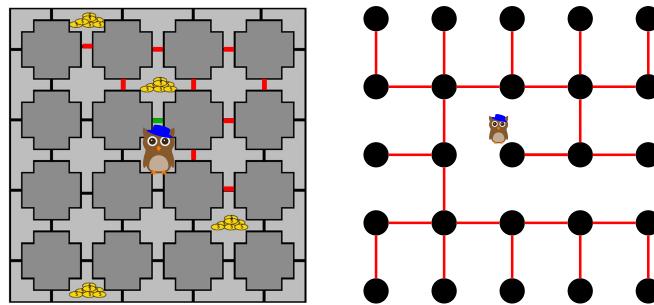


Fig. 1.25.: Finding all gold (No. 1083)

- Modeling

- Every junction and every dead end is a vertex
- Connected junctions form an edge
- Find: a path to every vertex
- More easily: a tree connecting all nodes

- Summary of the gold nugget search problem

Given: An undirected graph $G = (V, E)$

Find: a spanning tree in G

Definition 14. Let G be a connected graph. A subgraph $T \subseteq G$ is called a *spanning tree* of G if T is a tree which covers all vertices of G , i.e., $V(T) = V(G)$.

- The question is: How do we find such a spanning tree?

Algorithm 1.1 Generic graph searchInput: Connected Graph $G = (V, E)$, root vertex $r \in V$ Output: Spanning Tree T

Method:

Step 1 Initialization

- Set $R = \{r\}$ as the set of visited vertices
- Set $\text{pred}[r] = 0$ and $\text{pred}[v] = \text{NIL} \forall v \in V \setminus \{r\}$
- Set $L = \{r\}$ as the list of candidates for a visit

Step 2 Search procedure

```

While  $L \neq \emptyset$  do
  • chose  $v \in L$ 
  If  $\exists w \in V \setminus R$  with  $vw \in E$  do
    • chose  $w \in V \setminus R$  with  $vw \in E$ 
    • add  $w$  to  $R$ , set  $\text{pred}[w] = v$ , add  $w$  to  $L$ 
  Else delete  $v$  from  $L$ 

```

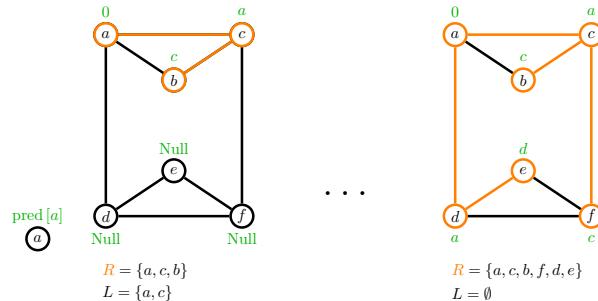
Step 3 **Return** Tree $T = (R, E)$ with $E = \{\{u, \text{pred}(u)\} \mid u \in R \setminus \{r\}\}$.

Fig. 1.26.: General graph search (No. 672)

- The output is a tree due to the construction
- The output of the generic graph search algorithm depends on the order in which the vertices are chosen from L
- The most famous ones are

BFS Breadth First Search: the vertices are always added at the end of L and the first vertex is chosen to be visited next, i.e., First-In-First-Out (FIFO) implemented by a queue

DFS Depth First Search: the vertices are added at the beginning of L and the first vertex is chosen to be visited next, i.e., Last-In-First-Out (LIFO) implemented by a stack

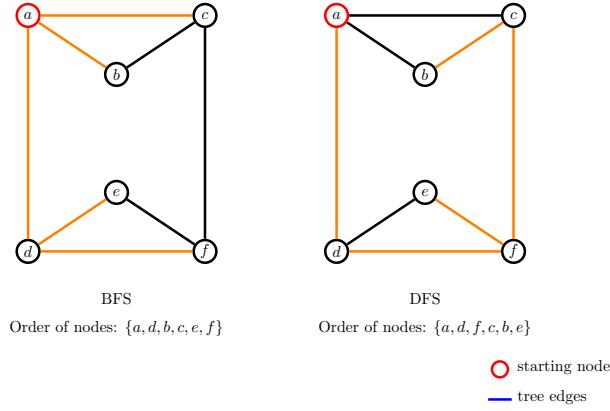


Fig. 1.27.: BFS and DFS (No. 673)

- Using BFS or DFS on the same graph computes spanning trees with different properties

Lemma 15. Let G be an undirected graph and r the root vertex.

1. Let T_{BFS} be a BFS-tree. Then any (r, v) -path is a shortest path with respect to the number of edges.
2. Let T_{DFS} be a DFS-tree. Then any edge that is not in T_{DFS} connects vertices along a path starting in r .

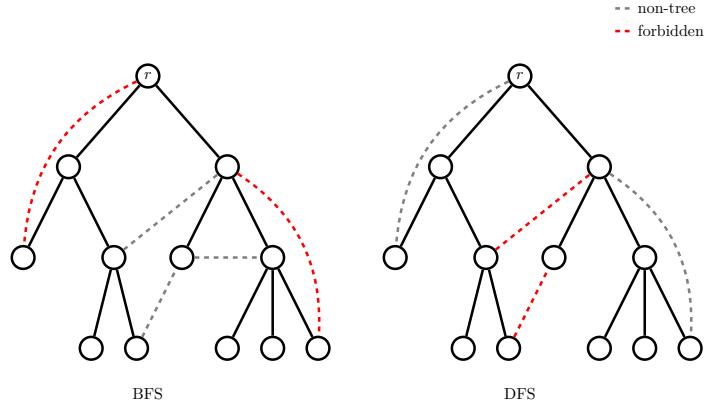


Fig. 1.28.: Properties of BFS and DFS (No. 669)

Proof. Construction of the tree

- Property (1): In T_{BFS} any path starting in r is a shortest path.
 - Let $\text{dist}(v)$ be the shortest path length from r to v in G w.r.t. the number of edges
 - Let $\text{level}(v)$ be the path length from r to v in T_{BFS}
 - *Claim:* $\text{level}(v) = \text{dist}(v) \ \forall v \in V$.
Proof of Claim:
 - Since T_{BFS} is a subgraph of G , $\text{level}(v) \geq \text{dist}(v)$ for all $v \in V$

- Assume, $\exists w \in V$ with $\text{dist}(w) < \text{level}(w)$ and minimum $\text{dist}(w)$ value, i.e., the “first” vertex that violates the condition according to $\text{dist}(v)$

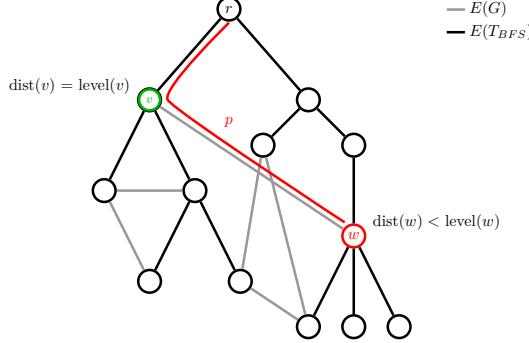


Fig. 1.29.: w is smallest criminal (No. 670)

- Let p be a shortest path in G from r to w
- Let vw be the last edge on p
- $\Rightarrow \text{dist}(v) = \text{level}(v)$
- Then,

$$\begin{aligned} \text{level}(w) &> \text{dist}(w) = \text{dist}(v) + 1 \\ &= \text{level}(v) + 1 \end{aligned}$$

- Since $\text{level}(v) < \text{level}(w)$, v is added to L before w
- (BFS): w is added the latest to L when scanning v , since $vw \in E$
- $\Rightarrow \text{level}(w) \leq \text{level}(v) + 1$, contradiction $\square C$

- Property (2): Any non-tree edge $e \in E \setminus E(T_{\text{DFS}})$ connects only vertices along a path starting in r
 - Assume uv connects two different paths, i.e., $v \notin p_{[r,u]}$ and $u \notin p_{[r,v]}$ with $p_{[a,b]}$ being the path in T connecting a and b
 - Let w be last vertex with $w \in p_{[r,v]} \cap p_{[r,u]}$
 - W.l.o.g. u is added to L before v
 - (DFS): u is scanned before w
 - $\Rightarrow v$ is added to L when u is scanned and $\text{pred}(v) = u$
 - $\Rightarrow uv \in T$, contradiction

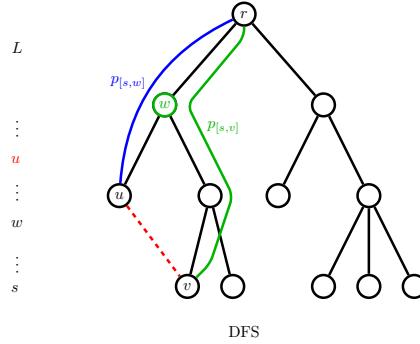


Fig. 1.30.: uv does not exist (No. 671)

□

- BFS and DFS are often subroutines in different, more complex algorithms

BFS	DFS
<ul style="list-style-type: none"> • shortest path computation • computation of maximum flows • choice of shortest cycles 	<ul style="list-style-type: none"> • test of planarity • topological sorting • construction of strong connectivity components

Bipartite Graphs

- In real world applications, vertices often represent groups and edges consist just between different groups
 - worker vs. jobs
 - seats vs. persons/people
- Such situations yield so called bipartite graphs

Definition 16 (Bipartite Graph). A graph $G = (V, E)$ is *bipartite* if there exists a partition of the vertices $V = U \cup W$ such that every edge $e = uw \in E$ has one end vertex in U and one in W .

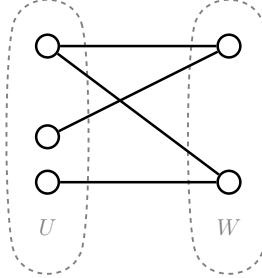


Fig. 1.31.: Bipartite graph (No. 528)

Theorem 17. A graph G is bipartite if and only if G has no cycle of odd length.

- The length of a cycle equals the number of edges

Proof. Exercise

□

- Using the idea in the proof, we can easily derive an algorithm to test whether a graph is bipartite

Complete Graphs

- Complete graphs contain the maximum number of edges

Definition 18. A graph $G = (V, E)$ in which each two vertices are adjacent is called a *complete graph*.

- We denote the complete graph on n vertices with K_n

- K_n has exactly $\binom{n}{2}$ edges

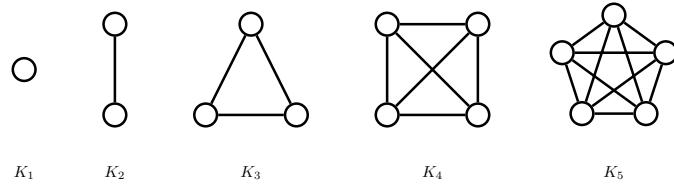


Fig. 1.32.: Complete graphs K_1, \dots, K_5 (No. 530)

- For some problems, we assume that a complete graph is given
- The most famous of these problems is the Traveling Salesperson Problem (TSP)

Definition 19. Traveling Salesperson Problem (TSP)

Given: Undirected complete graph $G = (V, E)$ and edge costs $c : E \rightarrow \mathbb{Z}$

Find: A cycle $C = e_1 e_2 \dots e_n$ which visits every vertex of graph G exactly once with minimum cost

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

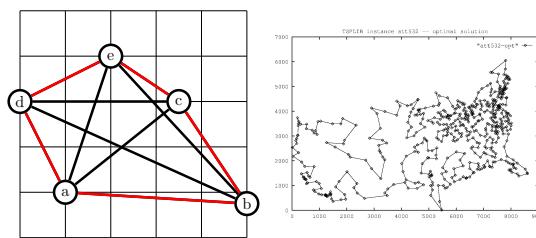


Fig. 1.33.: TSP (No. 529)

- The TSP problem has always attracted researchers and there are still many challenges in this direction going on

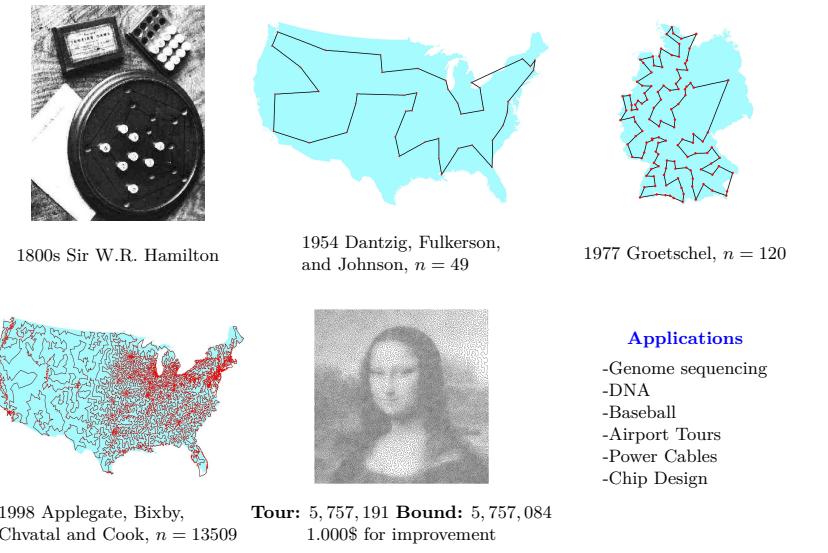


Fig. 1.34.: A short history of the TSP (No. 756)

- Size of search space for possible solutions:
 - Each of n cities can be at each position $\Rightarrow n!$ different ways
 - Cost for tours are the same regardless from which city you start and in which direction you are going
 - $\Rightarrow \frac{n!}{2n} = \frac{(n-1)!}{2}$ possible solutions
- Enumeration of all solutions to chose the best tour is no option

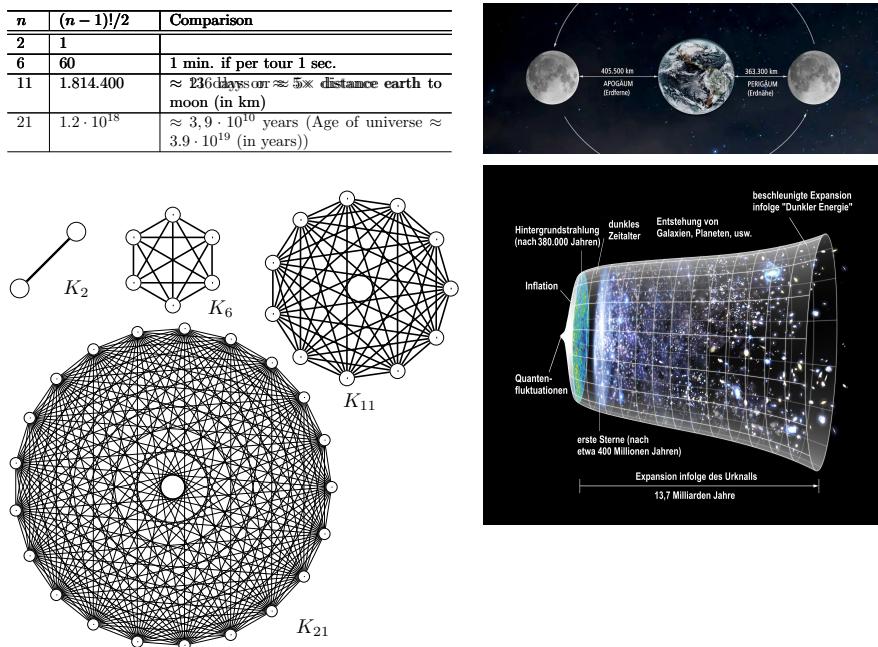


Fig. 1.35.: Combinatorial explosion (No. 757)



2. Minimum Spanning Tree

2.1. Motivation and Definition

- Consider the oases problem:
 - Desert state consists of seven oases which are connected through trade routes
 - Trade routes are damaged because of wind and sun \Rightarrow renovation of the roads is necessary
 - Furthermore, they are willing to install a new technology that enables traveling along one road in 0s
 - The cost for installing the technology are given along each road
 - However, not every trade road should be repaired and upgraded, it's sufficient to make sure that we can reach every oases from every other oases
 - Which roads should be renovated in order to minimize the renovation cost?

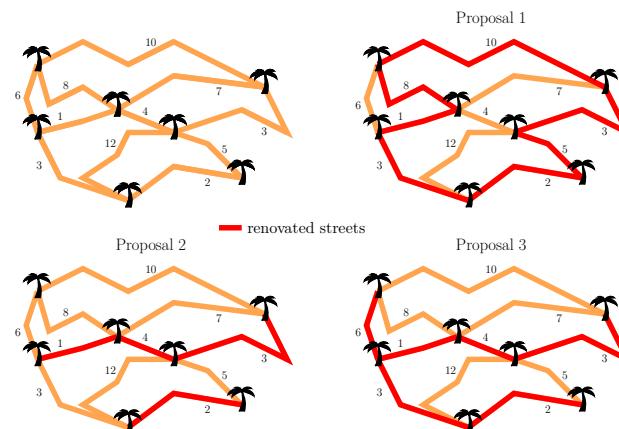


Fig. 2.1.: The oases problem (No. 538)

- Idea of a mathematical model:
 - oases correspond to vertices
 - routes correspond to edges
 - vertices and edges form a graph
 - renovation cost correspond to cost on the edges
 - renovation corresponds to choosing a subgraph
 - “reach every oasis from every other oasis”: Is there a sequence of edges also called path from any node to any other within the subgraph

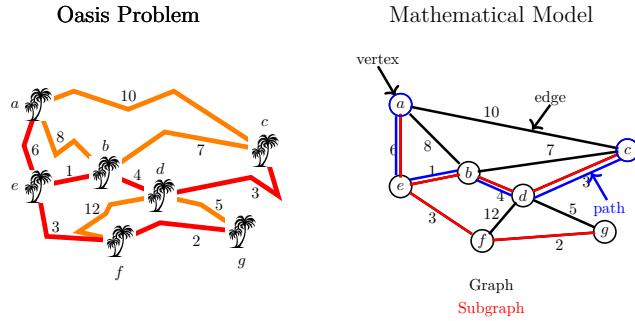


Fig. 2.2.: Modelling the problem as a graph (No. 989)

- Objective: Find a subgraph that connects all nodes with minimum cost
 - In the subgraph a path exists between every two vertices
 - There is no cycle in the subgraph
- In order to obtain a good infrastructure for the oases problem, a spanning tree with minimum cost is the best choice

Definition 20. Minimum Spanning Tree (MST) Problem

Given: Undirected, connected graph $G = (V, E)$ and edge cost $c : E \rightarrow \mathbb{R}$

Find: A spanning tree T with minimum cost $c(T)$ with

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

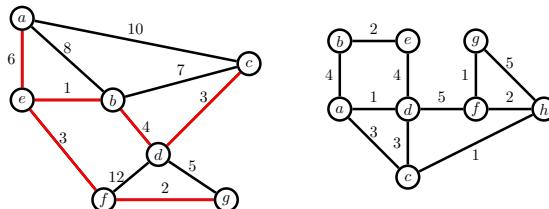


Fig. 2.3.: The left graph shows a minimum spanning tree. Find in the right one a spanning tree and compute its cost (No. 1057)

- In 1889, Caley proves that enumeration is a bad idea to find an MST:
 - There exist n^{n-2} different spanning trees in a complete graph on n vertices
 - Assume that we can enumerate 10^6 trees per second:
for $n = 30$ we have $\frac{30^{28}}{10^6}$ sec of computational time, which is roughly $7.25 \cdot 10^{27}$ years

2.2. Optimality Criterion

- Can we somehow characterize minimum spanning trees?
- In other words: Is there a simple criterion with which we can decide whether a given tree is a minimum spanning tree?

- Such a criterion is called an *optimality criterion*

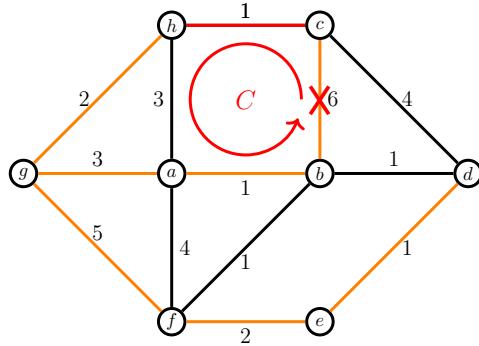


Fig. 2.4.: Are the colored edges a MST? (No. 535)

- If we can add a (cheap) edge and delete a more expensive one but keep a tree, our tree is not optimal
- More formally:

Definition 21 (Fundamental Cycle). Let $T = (V, E(T))$ be a spanning tree in a graph $G = (V, E)$ and $e = E \setminus E(T)$ be a *non-tree edge*. The created *fundamental cycle* C_e w.r.t. T is the unique cycle which results from $T + e$.

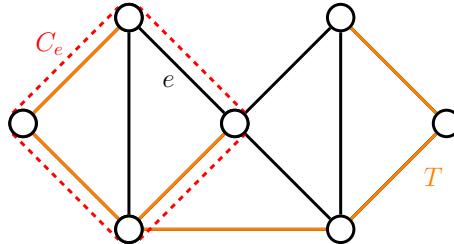


Fig. 2.5.: Fundamental cycle (No. 540)

- We first need to prove that this definition makes sense

Lemma 22 (Fundamental Cycle-Lemma). Let T be a spanning tree of $G = (V, E)$. The fundamental cycle C_e w.r.t. T which is created by the non-tree edge $e \in E(G) \setminus E(T)$ is well-defined.

Proof. Properties of a tree

- Let $e = uv$ be a non-tree edge
- *Claim 1:* If we add e to T , we obtain a cycle.
Proof of Claim:
 - T is a spanning tree $\Rightarrow \exists$ a path p from u to v in T

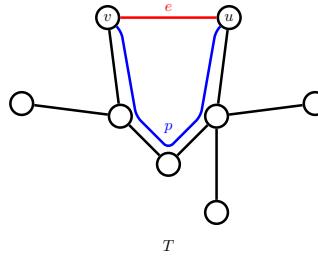


Fig. 2.6.: Fundamental cycle exists (No. 657)

- $p + e$ is a cycle which contains e $\square C1$
- \Rightarrow there exists a fundamental cycle in $T + e$
- *Claim 2:* The fundamental cycle C_e is unique.
Proof of Claim:
 - Assume there exist two different cycles C_1 and C_2 in $T + e$
 - Both cycles have to contain the edge e since T is a tree
 - Delete edge e from both cycles

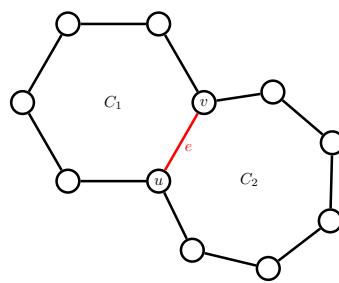


Fig. 2.7.: A fundamental cycle is unique (No. 658)

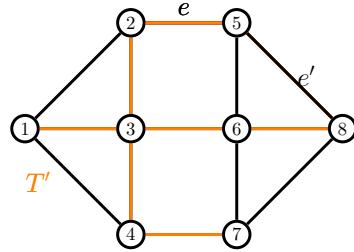
- \Rightarrow there remain two different simple paths from u to v in T
- Contradiction to T being a tree $\square C2$
- \square
- Next, we need to formulate the “exchange” argument (cheap non-tree edge vs. expensive tree edge)

Lemma 23 (Cycle Criterion). *Let T be a spanning tree of G . If T is a minimum spanning tree, then for all non-tree edges $e \in E(G) \setminus E(T)$ this edge is the most expensive edge in the corresponding fundamental cycle, i.e., $c(e') \leq c(e) \forall e' \in C_e$.*

Proof. Edge-exchange

- Assume there exists an edge $e \in E(G) \setminus E(T)$ and an edge $e' \in C_e$ with $c(e') > c(e)$
- Define the graph $T' = T - e' + e$
- T' contains $n - 1$ edges and no cycles, as the unique cycle in $T + e$ is destroyed
- $\Rightarrow T'$ is a tree (Theorem Important characteristics of trees ??)

- $\Rightarrow c(T') < c(T)$, contradiction

Fig. 2.8.: T' is cheaper than T (No. 539)

□

- Is there another criterion to make sure that we have an MST?

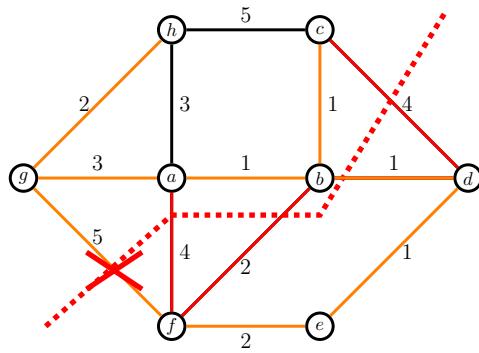
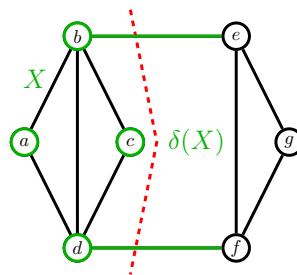


Fig. 2.9.: Is the above spanning tree an MST? (No. 536)

- If we can delete an edge from the tree and add a cheaper one to reconnect the two parts, our tree is not optimal
- Let $\emptyset \neq X \subsetneq V$. A $cut \delta(X)$ is the set of edges with exactly one end vertex in X , i.e., $\delta(X) = \{uv \in E \mid u \in X, v \notin X\}$
- We call the cut $\delta(X)$ *induced by* X

Fig. 2.10.: A cut is the edge set that divides the set of vertices X from the other vertices (No. 543)

Definition 24 (Fundamental Cut). Let $T = (V, E(T))$ be a spanning tree of graph $G = (V, E)$ and $e = uv \in E(T)$ be a tree edge. Let X_e be the set of vertices which are reachable from u in $T - e$. The *fundamental cut* w.r.t. T which is created by e is the cut $\delta(X_e)$ in G .

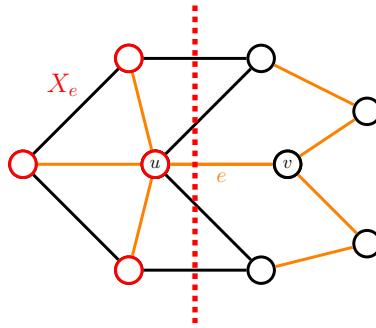


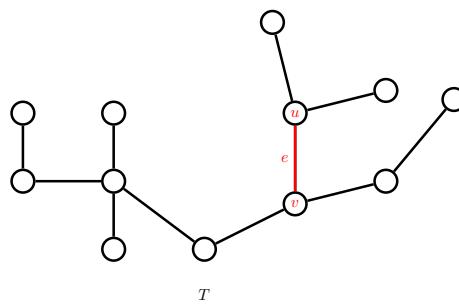
Fig. 2.11.: Fundamental cut (No. 537)

- As for fundamental cycles, we need to prove that the definition of fundamental cuts is valid

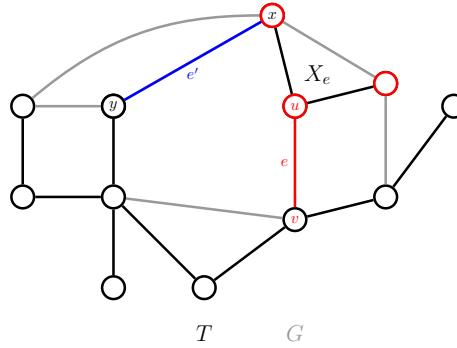
Lemma 25 (Fundamental Cut-Lemma). *Let T be a spanning tree of $G = (V, E)$. The induced fundamental cut $\delta(X_e)$ w.r.t. T which is created by the tree edges $e = uv \in E(T)$ is well-defined. Furthermore, e is the unique tree edge in $\delta(X_e)$, i.e., $\{\delta(X_e) \cap E(T)\} = \{e\}$.*

Proof. Properties of a tree

- Let $e = E(T)$ be a tree edge
- *Claim 1:* $X_e \neq \emptyset$ and $X_e \subsetneq V$.
Proof of Claim:
 - There exists a unique path between two vertices in a tree T
 - Delete edge $e = uv \Rightarrow$ there exists no longer a path between vertices u and v
 - W.l.o.g. $v \notin X_e$ and $u \in X_e$

Fig. 2.12.: Set X_e induces a cut (No. 659)

- $\Rightarrow \emptyset \neq X_e \subsetneq V$ $\square C1$
- \Rightarrow the cut induced by X_e is well defined
- *Claim 2:* The edge e is the unique edge of the tree in $\delta(X_e)$.
Proof of Claim:
 - Assume $e' = xy \in E(T) \cap \delta(X_e)$ and $e' \neq e$
 - W.l.o.g. $x \in X_e$ and $y \notin X_e$

Fig. 2.13.: e is the only tree edge (No. 660)

- $\Rightarrow x$ is reachable in $T - e$ from vertex u by definition of X_e

- $\Rightarrow y \in X_e$, contradiction

□C2

□

- With these two concepts we can define two optimality criteria

Theorem 26 (Optimality criteria for MST). *Let $G = (V, E)$ be a graph, $c : E \rightarrow \mathbb{R}$ an edge cost function and T a spanning tree of G . Then the following is equivalent:*

1. *The tree T is a minimum spanning tree.*
2. *For every non-tree edge e , it is true that e is one of the most expensive edges in the fundamental cycle w.r.t. T which is created by e . (Cycle-criterion)*
3. *For every tree edge e , it is true that e is one of the cheapest edges in the fundamental cut w.r.t. T which is created by e . (Cut-criterion)*

Proof. Ring closure and exchange arguments

- (1) \Rightarrow (2):

Given: T is a minimum spanning tree.

Prove: e is the most expensive edge in C_e

- See Cycle Criterion Lemma 23

- (2) \Rightarrow (3):

Given: Every non-tree edge is one of the most expensive edge in its fundamental cycle.

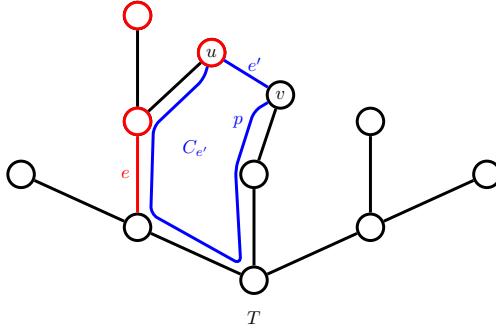
Prove: Every tree edge is one of the cheapest edges in its fundamental cut

- Let $e = xy$ be a tree edge with fundamental cut X_e , $x \in X_e$

- Assume there exists an edge $e' = uv \in \delta(X_e)$ with $c(e') < c(e)$

- By fundamental cut lemma, e' is a non-tree edge in $\delta(X_e)$

- $\Rightarrow e'$ closes a fundamental cycle $C_{e'}$ w.r.t. T

Fig. 2.14.: tree edge e is the cheapest in $\delta(X_e)$ (No. 661)

- *Claim:* $e \in C_{e'}$.

Proof of Claim:

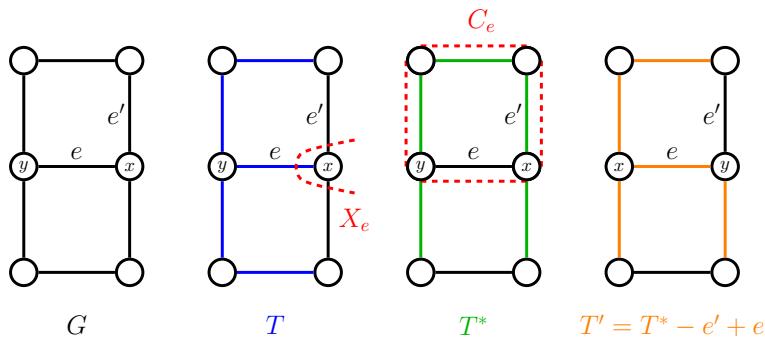
- $e' = uv \in \delta(X_e)$
- Let w.l.o.g. $u \in X_e$
- $\Rightarrow (u, v)$ -path p in T contains an edge in $\delta(X_e)$
- Since e is the unique tree edge in $\delta(X_e)$, e is part of the (u, v) -path in T
- $\Rightarrow e \in C_{e'}$ □C
- $\Rightarrow c(e') \geq c(e)$, contradiction to choice of e'

- (3) \Rightarrow (1):

Given: Every tree edge is one of the cheapest edges in its fundamental cut.

Prove: T is a minimum spanning tree.

- Let T^* be an MST such that $|E(T^*) \cap E(T)|$ is maximum
- Assume $T^* \neq T$
- $\Rightarrow \exists e = xy \in T$ and $e \notin T^*$
- $\Rightarrow e$ creates a fundamental cut $\delta(X_e)$ in T and a fundamental cycle C_e in T^*

Fig. 2.15.: G , T , T^* and T' (No. 542)

- *Claim:* $\exists e' \in C_e \setminus \{e\}$ with $c(e') = c(e)$.

Proof of Claim:

- $C_e \setminus \{e\}$ connects x and y
- $\Rightarrow \exists e' \in C_e \setminus \{e\}$ which is part of $\delta(X_e)$
- e' is a tree edge of T^* and a non-tree edge of T
- e is a non-tree edge of T^* and $e' \in C_e \Rightarrow c(e) \geq c(e')$ since T^* is optimal
- e is tree edge of T and $e' \in \delta(X_e) \Rightarrow c(e) \leq c(e')$ by assumption

- $\Rightarrow c(e) = c(e')$ $\square C$
- $T' = T^* - e' + e$ is an MST since $c(T^*) = c(T')$
- and $|E(T') \cap E(T)| = 1 + |E(T^*) \cap E(T)|$, contradiction to the choice of T^*

 \square

- Using these criteria we can test efficiently, whether a given tree T is an optimal minimum spanning tree (or not)

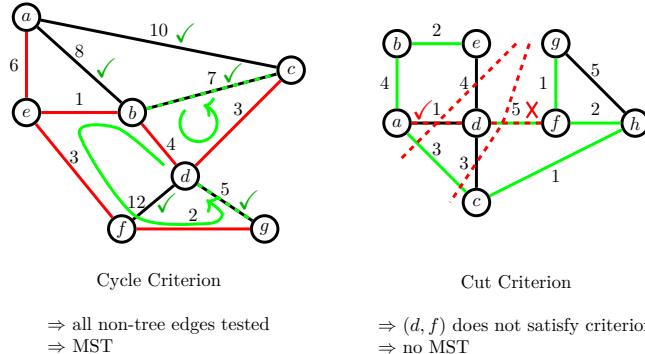


Fig. 2.16.: MST? (No. 1163)

2.3. Kruskal's and Prim's Algorithms

- The most classical algorithms to solve the minimum spanning tree problem are those by Kruskal and Prim
- In both cases, the idea is to start with an empty graph and extend the graph until we obtain a spanning tree
- Such methods are called a greedy algorithm
- After each extension, either the cycle-criterion (Kruskal) or the cut-criterion (Prim) remains satisfied
- Thus, we end up with a minimum spanning tree

Kruskal's Algorithm

- Kruskal, an American mathematician, published his algorithm in 1956

Algo. 2.1 Kruskal's algorithm

Input: Undirected, connected graph $G = (V, E)$ and edge cost $c : E \rightarrow \mathbb{R}$

Output: Spanning tree T

Method:

- Step 1
- Set $m = |E|$ and $n = |V|$
 - Set $E(T) = \emptyset$ and sort all edges by their cost in non-decreasing order, i.e., $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$
 - Set $i = 1$
- Step 2 **While** $|E(T)| \neq n - 1$ **do**
- If** e_i does not close a cycle in $T = (V, E(T))$ **do**
 - Add e_i to $E(T)$
 - Set $i = i + 1$
- Return** $T = (V, E(T))$
-

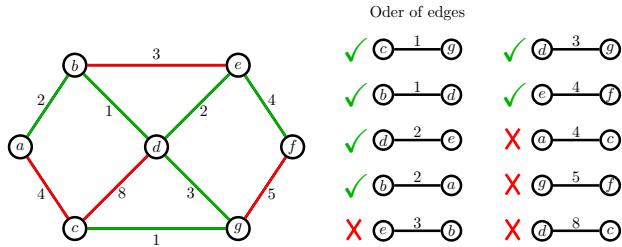


Fig. 2.17.: Kruskal's Algorithm (No. 662)

Theorem 27. Kruskal's algorithm computes a minimum spanning tree.

Proof. Cycle-criterion

- Let T be the spanning tree computed by Kruskal's algorithm
- Let e be a non-tree edge of T

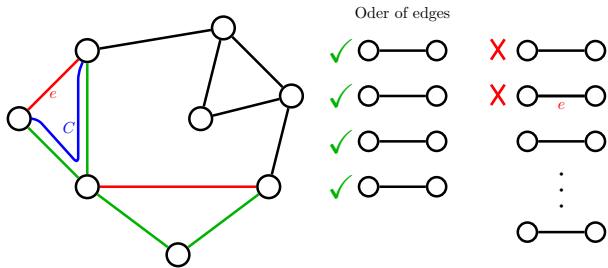


Fig. 2.18.: Kruskal's Algorithm constructs MST (No. 663)

- Since e is not part of T , it closes a cycle C when it was considered in the algorithm
- Since the fundamental cycle of e w.r.t. T is unique, C is the fundamental cycle C_e
- All other edges in C_e were added before e

- $\Rightarrow c(e) \geq c(e') \forall e' \in C_e$
- \Rightarrow the cycle-criterion is fulfilled
- $\Rightarrow T$ is an MST

□

- The run-time of the algorithm heavily depends on the implementation and the data structure
- In the \mathcal{O} -notation, one estimates the number of steps needed (more Details: See Section on Complexity)

Theorem 28. Kruskal's algorithm can be implemented with a run-time of $\mathcal{O}(m \cdot \log m + n^2)$ where m denote the number of edges and n the number of nodes of the considered graph.

Proof. Data structure to label nodes of the same connected component

- The run-time depends on how fast, we can
 1. Sort all edges
 2. Test if a cycle is closed
 3. Add an edge to the tree
- Sorting of m elements can be done in $\mathcal{O}(m \cdot \log m)$
- 2. and 3. can be done nicely in the following way:

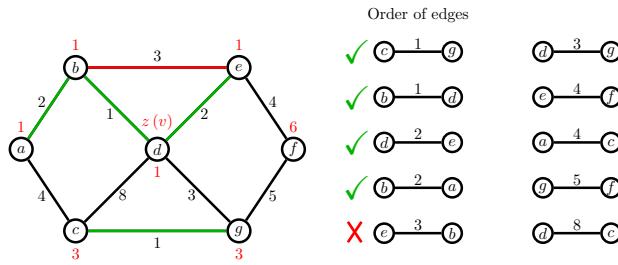


Fig. 2.19.: Labeling with Kruskal (No. 664)

- Label each vertex with $z(v)$
- The label $z(v)$ corresponds to a connected component, i.e., all vertices v with $z(v) = x$ are at that step in the algorithm in the same connected component of the current tree
- Initialization: Set $z(v_i) = i$ for $i = 1, \dots, n$ (some numbering of the vertices)
- Let $e = uv$ be some edge we consider in Step 2
- If $z(u) = z(v) \Rightarrow u$ and v are already connected $\Rightarrow e$ closes a cycle
- If $z(u) \neq z(v) \Rightarrow u$ and v are in different components \Rightarrow we need to add e to our tree and update z
- Update z according to $e = uv$: Set $z(w) = z(u)$ for all $w \in V$ with $z(w) = z(v)$
- In total: For each update, we need $\mathcal{O}(n)$ time
- Updating is done $m = (n - 1)$ -times

- Total run-time is: $\mathcal{O}(m \log m + n^2)$

□

- For an efficient implementation see [?]
 - $\mathcal{O}(m \log n)$ if sorting is needed
 - $\mathcal{O}(m \cdot \alpha(m, n))$ if sorting is given, whereby $\alpha(m, n)$ is the inverse of the Ackermann function¹, i.e., a quasi constant

Prim's Algorithm

- A different algorithm is based on the cut-criterion
- The algorithm was developed in 1930 by the Czech mathematician Vojtěch Jarník
- Later it was rediscovered and republished by the computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959.
- Therefore, it is also sometimes called the Jarník's algorithm, Prim-Jarník algorithm, Prim-Dijkstra algorithm or the DJP algorithm

Algo. 2.2 Prim's algorithm

Input: Undirected, connected graph $G = (V, E)$ and edge cost $c : E \rightarrow \mathbb{R}$

Output: Minimum spanning tree T

Method:

Step 1 Choose an arbitrary $v \in V(G)$ and set $T = (\{v\}, \emptyset)$

Step 2 **While** $V(T) \neq V(G)$ **do**

- Choose $e \in \delta(V(T))$ with minimum cost $c(e)$
- Set $E(T) = E(T) + e$

Return $T = (V, E(T))$

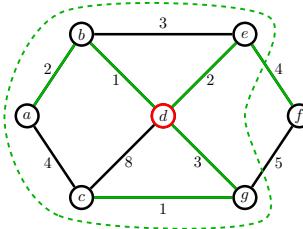


Fig. 2.20.: Prim's Algorithm (No. 665)

Theorem 29. *Prim's algorithm computes a minimum spanning tree.*

Proof. Exercise

□

- The run-time depends again on a “good” data structure, i.e., that we can choose $e \in \delta(V(T))$ with minimum cost efficiently

¹Ackermann functions are simple function that grows very rapidly and is often used in computability theory

Theorem 30. *Prim's algorithm can be implemented in $\mathcal{O}(n^2)$ where n denotes the number of nodes of the considered graph.*

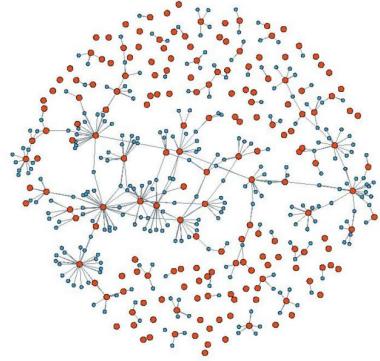
Proof. Exercise □

- There are several extensions of the above algorithms to speed them up



3. Directed Graphs or Digraphs

- So far, an edge represented a relation between two objects



Social Network



Street Network

Fig. 3.1.: Undirected graphs do not model everything (No. 809)

- Often relations are not the same for both objects
 - a likes b but b does not like a
 - one-way streets can only be traversed in one direction
- In order to model these aspects, we orient the edges

Definition 31. A graph $G = (V, A)$ with an orientation is called *directed graph* or *digraph*. An orientation assigns to every edge an explicit start and an end vertex. For an edge that is oriented from u to v we write (u, v) . The graph without orientation is called the *underlying undirected graph*.

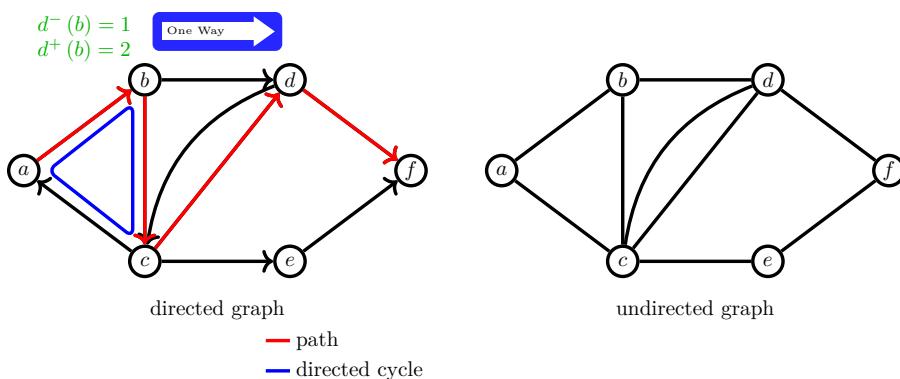


Fig. 3.2.: Directed graphs (No. 601)

- When considering digraphs, we also say *arcs* instead of edges
- Most of the definitions of undirected graphs can be transferred easily:

- A directed path is a sequence of arcs a_1, \dots, a_k with $a_i = (v_i, u_i)$ and $v_i = u_{i-1}$ for $i = 2, \dots, k$
- A *directed cycle* is a closed directed path
- $d^+(v)$ and $d^-(v)$ are the *out-degree* and *in-degree* of vertex v
- $\delta^+(X) = \{(u, v) \in A \mid u \in X, v \notin X\}$ and $\delta^-(X) = \{(u, v) \in A \mid u \notin X, v \in X\}$ are *directed out-cuts* or *in-cuts*, respectively, and $\delta(X) = \delta^+(X) \cup \delta^-(X)$

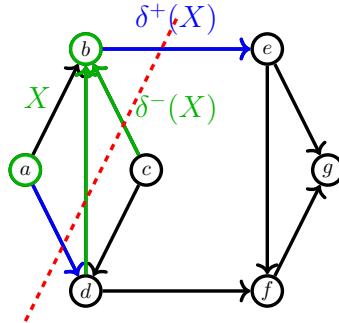


Fig. 3.3.: A directed cut divides the set of vertices X from the other vertices (No. 558)

- If $\delta(X) = \delta^+(X)$ or $\delta(X) = \delta^-(X)$, we call X a *directed cut*
- Furthermore, problems and algorithms can often be transferred

Connectivity

- This term can not be transferred easily and still depends in the weak form on the undirected version of the graph

Definition 32. A directed graph is (*weakly*) *connected* if the underlying undirected graph is connected.

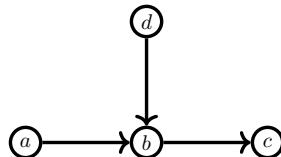


Fig. 3.4.: In a connected digraph, two vertices are not necessarily connected via a directed path (No. 548)

- A slightly advanced concept is called *strong connectivity*

Definition 33. A digraph $G = (V, A)$ is called *strongly connected* if for each two vertices $s, t \in V$ there exists a directed path from s to t and from t to s .

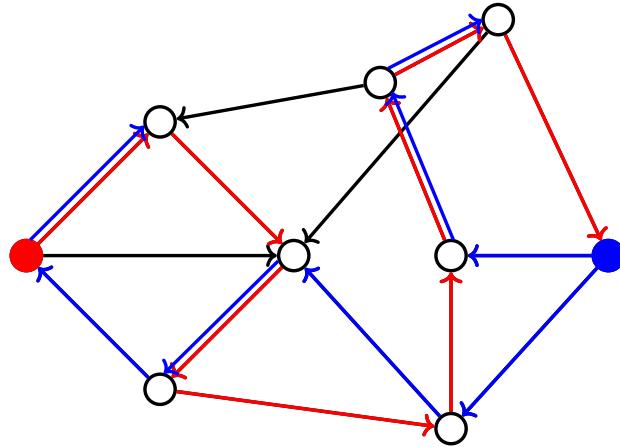


Fig. 3.5.: Strongly connected graph (No. 810)

- When is a directed graph strongly connected?

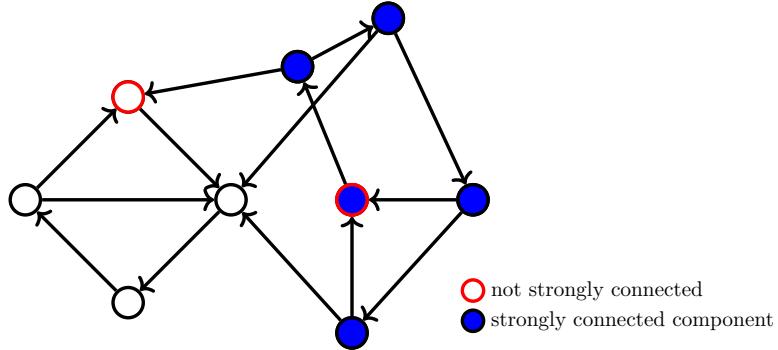


Fig. 3.6.: Strongly connected components (No. 602)

Theorem 34. Let G be a digraph. Then the following three statements are equivalent:

1. G is strongly connected.
2. G doesn't contain a directed cut, i.e., $\emptyset \neq X \subsetneq V$ s.t. $\delta^-(X) := \{(v, u) \in G \mid u \in X, v \notin X\}$ with $\delta^-(X) = \emptyset$
3. G is connected and every arc is part of a directed cycle.

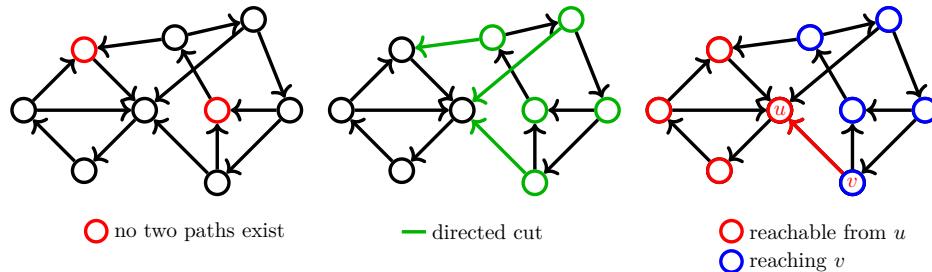


Fig. 3.7.: strongly connected? (No. 603)

Proof. Proof by contradiction

- (1) \Rightarrow (2):

Given: G is strongly connected.

Prove: G does not contain a directed cut

- Assume there exists a directed cut $\delta(X) = \delta^+(X)$ with $\emptyset \not\subseteq X \subsetneq V$
- Let $a' = (v, w) \in \delta^+(X)$ with $v \in X$ and $w \in V \setminus X$
- Since G is strongly connected \Rightarrow a path p from w to v exists
- $\Rightarrow \exists a = (x, y) \in p$ with $x \in V \setminus X$ and $y \in X$
- $\Rightarrow a \in \delta^-(X)$, contradiction

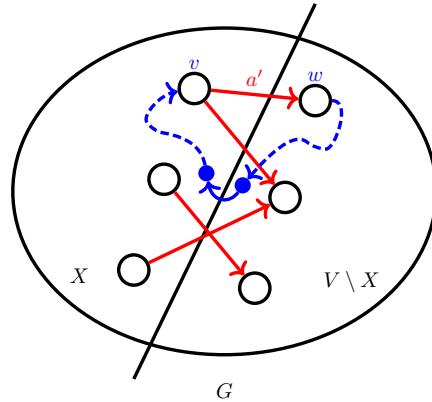


Fig. 3.8.: directed cut (No. 604)

- (2) \Rightarrow (3):

Given: G doesn't contain a directed cut.

Prove: G is connected and every arc is part of a directed cycle

- Assume G isn't connected.
- \Rightarrow let X be the set of vertices of one of the connected components

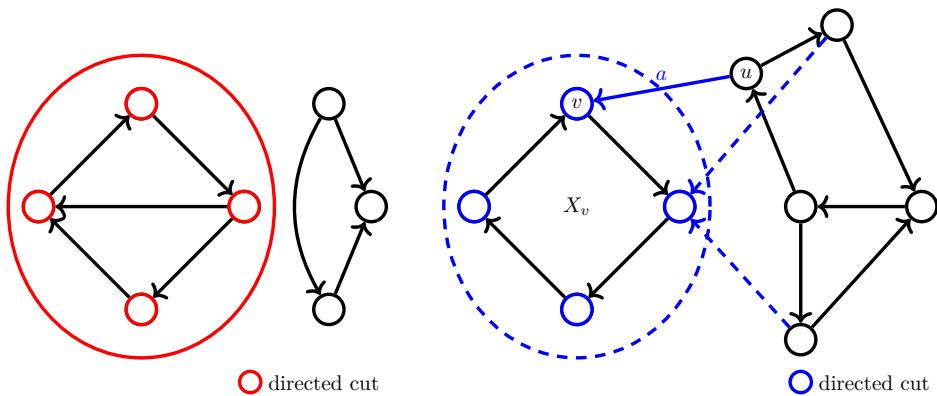


Fig. 3.9.: Every arc is part of cycle (No. 605)

- $\Rightarrow \delta^-(X) = \emptyset$ and is a directed cut, contradiction

- *Claim:* Every arc is part of a directed cycle.

Proof Claim:

- Assume arc $a = (u, v)$ is not a part of any directed cycle
- Let $X_v \subseteq V$ be the set of nodes that are reachable by a directed path from v

- $u \notin X_v$ since otherwise there exists a directed path from v to u and therefore, a is a part of a directed cycle
- Consider the cut $\delta(X_v)$
- By definition of X_v all arcs are orientated from $V \setminus X_v$ to X_v
- $\Rightarrow \delta(X_v)$ is a directed cut, contradiction $\square C$
- (3) \Rightarrow (1):
Given: G is connected and every arc is part of a directed cycle.
Prove: G is strongly connected.
 - Assume $r, w \in V$ are not connected via a directed path
 - Let X_r be the set of vertices which are reachable from r
 - $\Rightarrow w \notin X_r$
 - By definition, there exists no arc from X_r to $V \setminus X_r$
 - Since G is connected, at least one arc a connects X_r and $V \setminus X_r$, i.e. $a = (u, v)$ with $u \in V \setminus X_r$ and $v \in X_r$
 - a is part of a directed cycle
 - $\Rightarrow \exists$ a directed path p which connects v with u
 - Since $|A(p) \cap \delta^+(X_r)| \geq 1$, $\exists a' = (x, y) \in A(p) \cap \delta^+(X_r)$
 - But $x \in X_r$ and $y \in V \setminus X_r$, contradiction

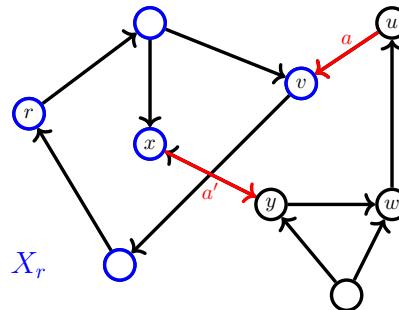


Fig. 3.10.: At least one arc between X_r and $V \setminus X_r$ must exist (No. 546)

\square

- Using these properties, we can test whether a graph is strongly connected

Orienting undirected Graphs

- We will next consider what happens if we can actually choose the orientation
- The objective is to orient the edges in such a way, that the obtained graph is strongly connected

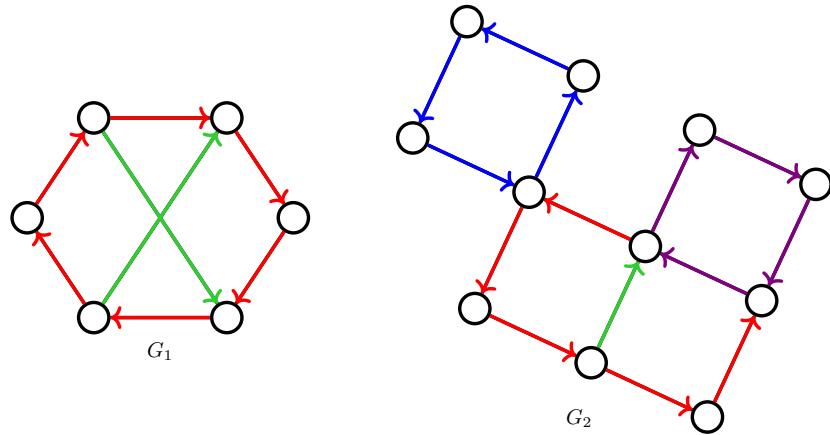


Fig. 3.11.: How to orient the edges? (No. 811)

- In order to prove a simple criteria we need one more definition

Definition 35. Let $G = (V, E)$ be a graph and $V' \subseteq V$. The *contraction* $G_{V'}^c$ of G is defined in the following way:

- $V(G_{V'}^c) = (V \setminus V') \cup \{v'\}$
- $E(G_{V'}^c) = E_1 \cup E_2 \cup E_3$ with $E_1 = \{(u, v) \in E \mid v, u \in V \setminus V'\}$, $E_2 = \{(u, v') \mid \exists u \in V \setminus V' \text{ and } v \in V' \text{ with } (u, v) \in E\}$ and $E_3 = \{(v', u) \mid \exists u \in V \setminus V' \text{ and } v \in V' \text{ with } (v, u) \in E\}$

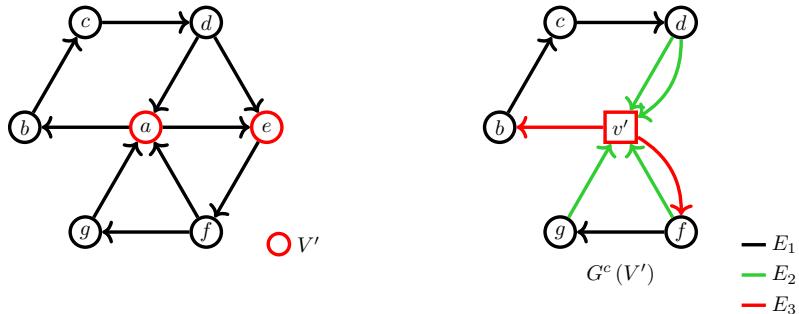


Fig. 3.12.: Contraction (No. 606)

- Note that the contraction may contain parallel and anti-parallel edges
- The following theorem provides a simple criterion to decide whether an orientation exists such that the graph is strongly connected

Theorem 36. An undirected graph can be orientated to a strongly connected digraph if and only if the graph is connected and has no bridge.

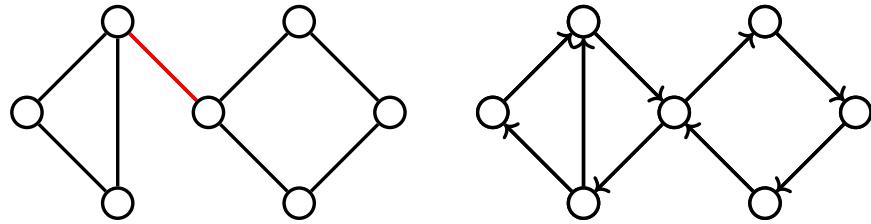
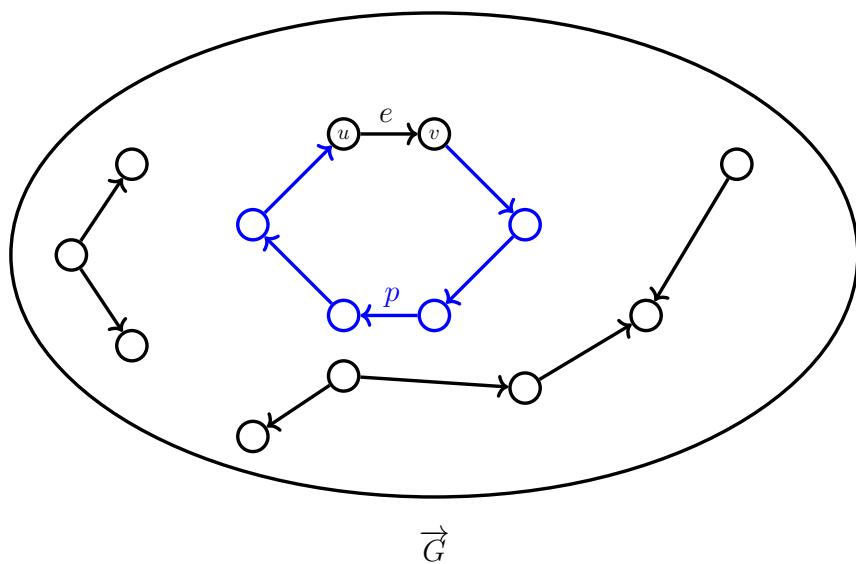


Fig. 3.13.: Bridge (No. 607)

Proof. Simple construction via cycles and contraction

- (\Rightarrow): Let \vec{G} be a strongly connected digraph of G and $e = (u, v)$ be an arbitrary edge

Fig. 3.14.: If strongly connected, e is part of a cycle (No. 812)

- Since \vec{G} is strongly connected there exists a path in \vec{G} from v to u
- $\Rightarrow e$ is part of a cycle and therefore no bridge (Theorem ??)
- (\Leftarrow): Consider the following procedure
 - Step 1 Find a cycle in G
 - Step 2 Orient all edges C in one direction
 - Step 3 **If** C contain all vertices, orient all non-oriented edges in an arbitrary fashion
Stop
Else Contract G by $V(C)$ and go to Step 1 with the new graph

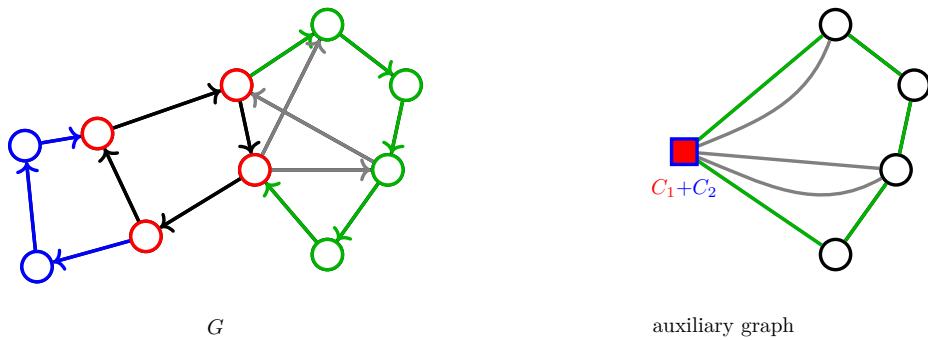


Fig. 3.15.: Strong Orientation (No. 608)

- Since G has no bridge $\Rightarrow \exists$ a cycle in G
 - Since G has no bridge $\Rightarrow G_{V(C)}^c$ has no bridge
 - \Rightarrow in every step, a cycle exists and thus can be found
 - If the algorithm ends, all vertices are part of at least one cycle
 - The orientation works in such a way that we can reach any vertex from any other
 - \Rightarrow the graph is strongly connected according to Theorem 34
 - The algorithm stops after at most $n-3$ iterations, since C contains at least 2 vertices

1

Eulerian Tours

- Similar to other graph problems, there exists a directed version of Eulerian graphs

Definition 37. A digraph $G = (V, A)$ is *Eulerian* if and only if there exists a directed cycle in G which contains every arc of the graph exactly once. We call such a cycle an *Eulerian-Tour*.

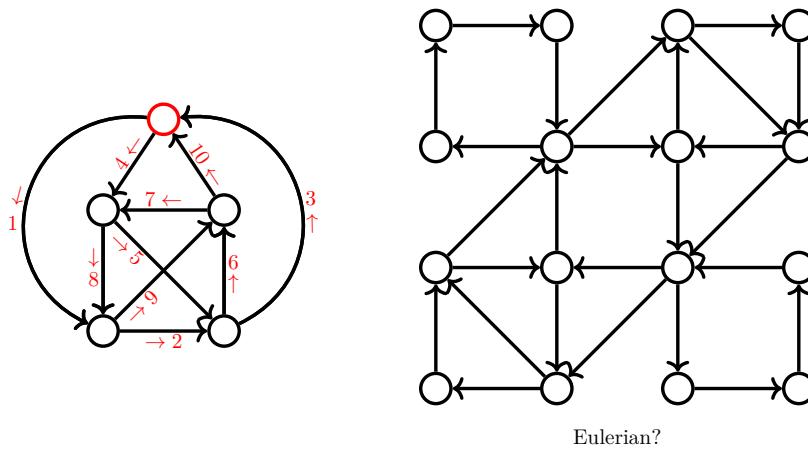


Fig. 3.16.: Eulerian directed graphs (No. 609)

- The property under which such a tour exists is similar to the undirected version

Theorem 38. A digraph $G = (V, A)$ is Eulerian if and only if the underlying undirected graph is connected and $d^+(v) = d^-(v)$ for every vertex $v \in V$.

Proof. Analogous to the proof of Euler's theorem □

- Also the Hierholzer's algorithm can easily be transferred
- However, here is a nice version which constructs a tour without gluing cycles
- The main idea of the algorithm was introduced by a Fleury, a french mathematician, in 1883
- **Note:** we use here an adaption of the graph search algorithm for directed graphs in order to obtain a directed spanning tree

Algo. 3.1 Fleury's Algorithm

Input: Eulerian digraph $G = (V, A)$

Output: Directed Eulerian tour

Method:

- Step 1
 - Let G' be the digraph which results by switching the orientation of G
 - Choose a vertex $v \in V$ and create a spanning tree T' of G' with root v
 - Step 2
 - Let T be the tree which results from T' by switching the orientation of the arcs $\Rightarrow T$ contains a path to v for all vertices $u \in V \setminus \{v\}$
 - Determine an arbitrary order of the non-tree arcs
 - Step 3
 - Create an Eulerian Tour from v by choosing the lowest unvisited arc for all $u \in V$ – according to the order in Step 2 – and only then a tree arc
-

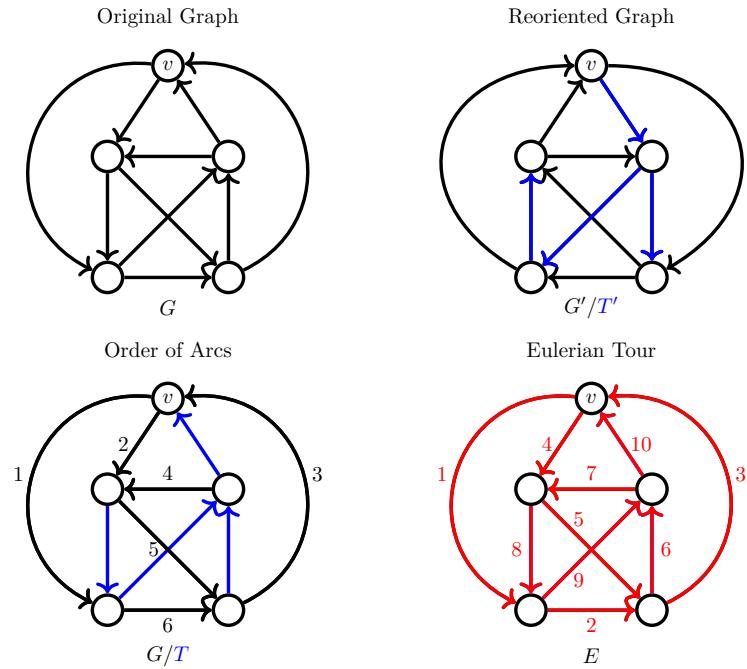


Fig. 3.17.: Construction of orientation (No. 549)

Theorem 39. *If a digraph is Eulerian, Fleury's Algorithm constructs an Eulerian-Tour.*

Proof. Exercise □



4. Shortest Path Problems

4.1. An Optimality Criterion for Shortest Paths

- Nowadays, everyone is using openstreetmaps.org, google.maps or other services to obtain the shortest route between two places
- The question is: How do they obtain these routes?
- A little more general: Given a location, how do we find a shortest path to all other locations?

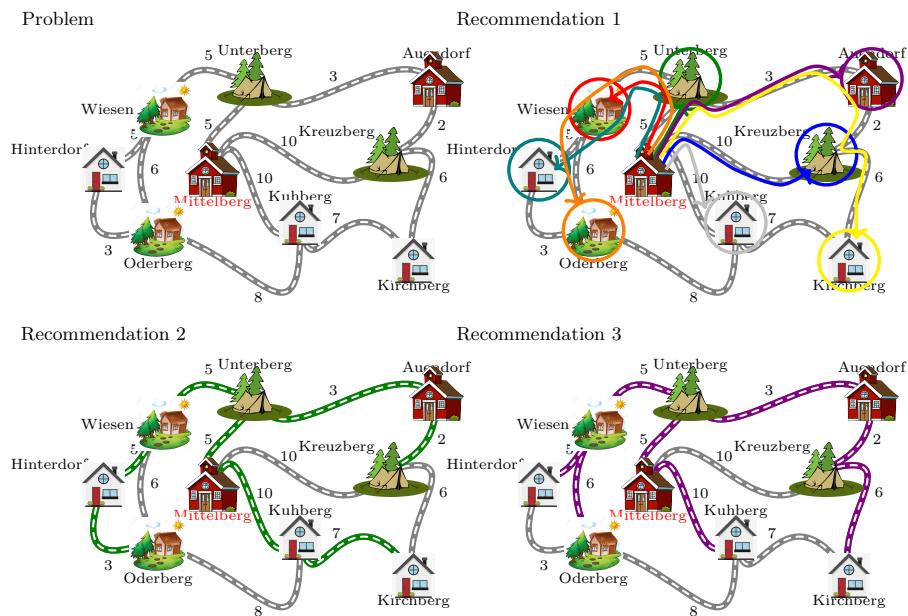


Fig. 4.1.: How to find and denote all shortest paths? (No. 807)

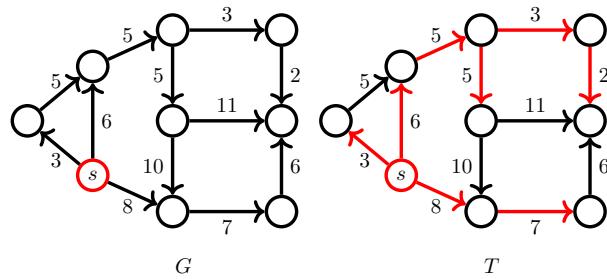
Definition 40. Shortest Path Tree Problem

Given: Digraph $G = (V, A)$, costs $c : A \rightarrow \mathbb{R}$ and a vertex $s \in V$

Find: A directed spanning tree T with root $s \in V$ such that for every $v \in V$ the (s, v) -path in T is a shortest path in G . A (s, v) -path p is a *shortest path* if no other path has lower cost, i.e., $\forall (s, v)$ -path p' it holds

$$c(p) := \sum_{a \in A(p)} c(a) \leq \sum_{a \in A(p')} c(a) = c(p').$$

Such a tree is called a *shortest path tree*.

Fig. 4.2.: Shortest Path tree with root s (No. 612)

Question 1: When does a shortest path between two vertices s, t exist?

- The definition of the cost function is quite general
- Negative cost may, e.g., model benefits for using a certain route
- However, with negative cycles, the definition of a shortest path is a little complicated

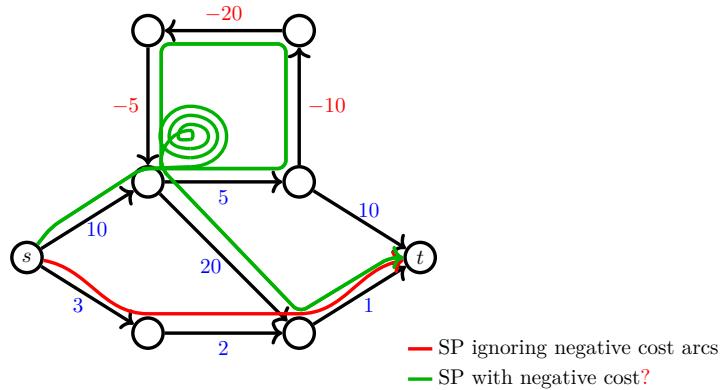


Fig. 4.3.: Negative cost (No. 610)

- We rule out negative cycles by defining conservative costs

Definition 41. Let $G = (V, A)$ be a digraph with costs $c : A \rightarrow \mathbb{R}$. The cost function c is called *conservative* if there is no cycle with negative total cost.

- With this restriction, we obtain the following nice property

Lemma 42. Let G be a digraph with conservative costs $c : A \rightarrow \mathbb{R}$.

1. If an (s, t) -path exists in G , a shortest (s, t) -path exists in G .
2. Let p be a shortest (s, t) -path. Then, every connected subpath of p is also a shortest path.

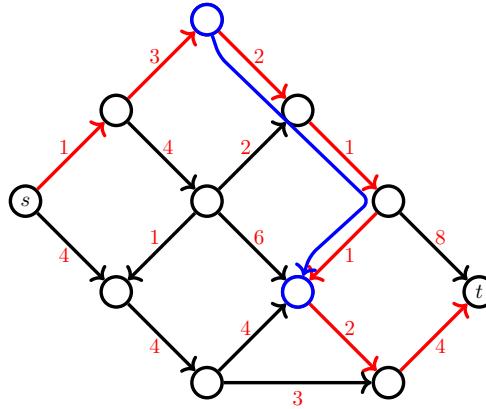


Fig. 4.4.: Parts of a shortest path are shortest paths (No. 611)

Proof. Exchange argument

- Property 1: A shortest (s, t) -path exists
 - Let $\mathcal{P}_{(s,t)}$ be the set of all (s, t) -paths in G
 - $\mathcal{P}_{(s,t)} \neq \emptyset$ due to the condition of the lemma
 - *Claim:* For every (s, t) -path p' with $|A(p')| \geq n$, \exists (s, t) -path p with $|A(p)| < |A(p')|$
- Proof of Claim*
- Let p' be an (s, t) -path with $|A(p')| \geq n$
 - $\Rightarrow p'$ contains a cycle C
 - Delete C from p' to obtain a path p
 - $\Rightarrow |A(p)| \leq |A(p')| - 2$ and

$$c(p) \leq c(p) + \underbrace{c(C)}_{\geq 0} = c(p')$$

□C

- \Rightarrow restrict $\mathcal{P}_{(s,t)}$ to all paths with less than n arcs
- There are only finite many \Rightarrow a minimum exists
- Property 2: Every subpath of a shortest path is a shortest path
- Let p be a shortest path between s and t , and $u, v \in V(p)$
- Let $p_{[u,v]}$ be the subpath of p connecting u and v
- Assume $p' = p_{[u,v]}$ is not a shortest path
- $\Rightarrow \exists p''$ connecting u and v with $c(p'') < c(p')$
- Define $\bar{p} := p_{[s,u]} \cup p'' \cup p_{[v,t]}$
- $\Rightarrow c(\bar{p}) = c(p_{[s,u]}) + c(p'') + c(p_{[v,t]}) < c(p_{[s,u]}) + c(p') + c(p_{[v,t]}) = c(p)$, contradiction

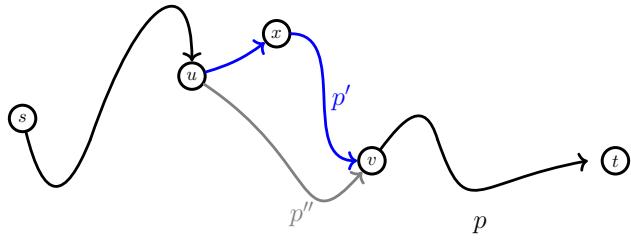


Fig. 4.5.: Subpaths of shortest paths (No. 550)

□

Question 2: Even if shortest paths exists, does a shortest path tree exist?

- Just because of the existance of a shortest path, not necessarily a shortest path tree exists

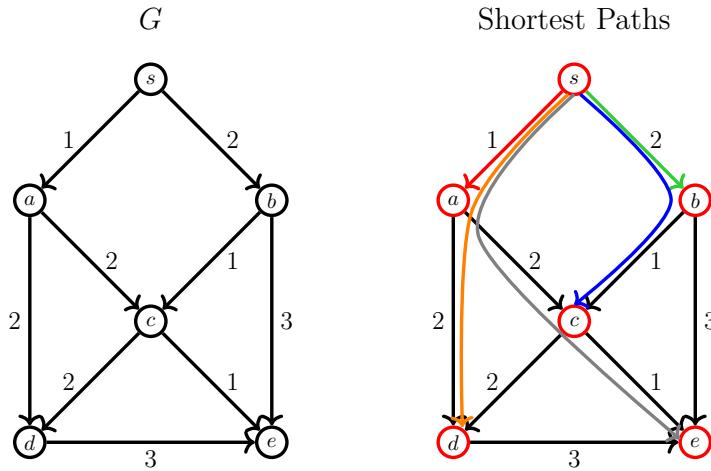


Fig. 4.6.: Does a shortest path tree exist? (No. 808)

Theorem 43 (Existance Shortest Path Trees). *Let $G = (V, A)$ be a digraph with conservative costs $c : A \rightarrow \mathbb{R}$. Then the following statements are equivalent:*

1. *There exists a shortest path tree with root s in G .*
2. *For every $v \in V$ there exists a (s, v) -path in G .*

Proof. Constructive algorithm

- (1) \Rightarrow (2): Clear
- (2) \Rightarrow (1): For every $v \in V$ there exists a path from s to v
 - Construct a shortest path tree T :

Step 1 Let $G' = (V, A')$ be a subgraph of $G = (V, A)$ with $A' = \emptyset$.
For every vertex $v \in V$ **do**
 Choose a shortest (s, v) -path p and add its arcs to G' , i.e., $A' = A' \cup A(p)$

Step 2 Compute a spanning tree in G'

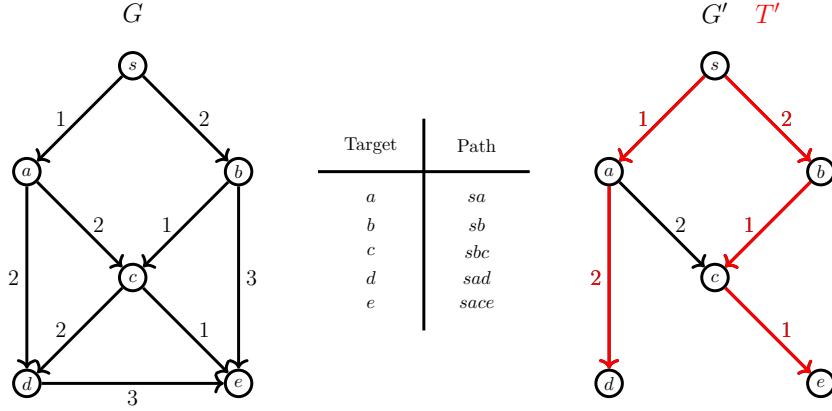


Fig. 4.7.: Construct shortest path tree (No. 551)

- *Claim:* T' is a shortest path tree.

Proof of Claim

- Assume p in T' from s to y is not a shortest (s, y) -path in G
- Let v be the last vertex on p such that $p_{[s, v]}$ is a shortest path
- Let w be the successor of vertex v on path p
- $\Rightarrow p_{[s, w]}$ is not a shortest path
- Since arc (v, w) is part of T' , it is also a part of G'
- By construction of G' , \exists shortest path \bar{p} from s to some vertex x such that $(v, w) \in A(\bar{p})$
- $\Rightarrow \bar{p}_{[s, v]}$ as well as $\bar{p}_{[s, w]}$ are shortest paths (Lemma 42)
- $\Rightarrow c(p_{[s, v]}) = c(\bar{p}_{[s, v]})$
- \Rightarrow

$$\begin{aligned} c(p_{[s, w]}) &= c(p_{[s, v]}) + c((v, w)) \\ &= c(\bar{p}_{[s, v]}) + c((v, w)) = c(\bar{p}_{[s, w]}) \end{aligned}$$

- \Rightarrow contradiction to $p_{[s, w]}$ is not a shortest path

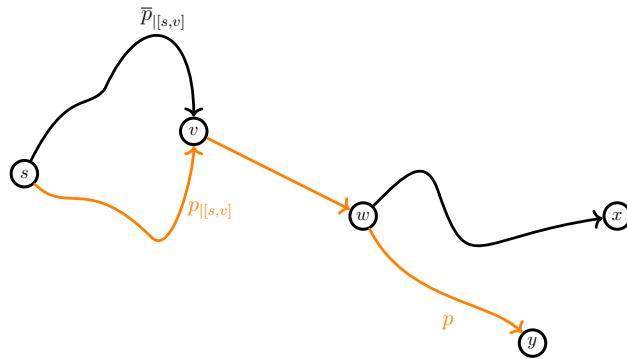


Fig. 4.8.: Construction of T' (No. 552)

□C

□

Question 3: Exists an optimality criteria for shortest path trees?

- Similar to the MST problem, we look for an optimality criterion:
- Given a tree, can we somehow decide whether this tree is a shortest path tree?

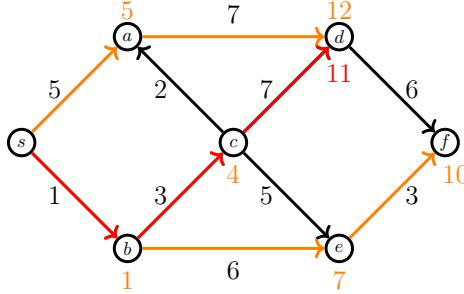


Fig. 4.9.: Is this a shortest path tree? (No. 554)

- Notation: Let T be a tree with a root s , then $\text{dist}(v) = \text{dist}_T(v) = c(p_{[s,v]})$ where $p_{[s,v]}$ is the path in T from s to v
- Since in every tree there exists a unique path between two vertices, $\text{dist}(v)$ is well defined

Theorem 44 (Optimality Criterion for Shortest Path Tree). *Let $G = (V, A)$ be a digraph with conservative costs $c : A \rightarrow \mathbb{R}$ and $s \in V$. Furthermore, let T be a directed spanning tree of G . Then the following statements are equivalent:*

1. *The tree T is a shortest path tree with root $s \in V$.*
2. *For all non-tree arcs $(v, w) \in A$ the triangle inequality holds:*

$$\text{dist}(v) + c(v, w) \geq \text{dist}(w).$$

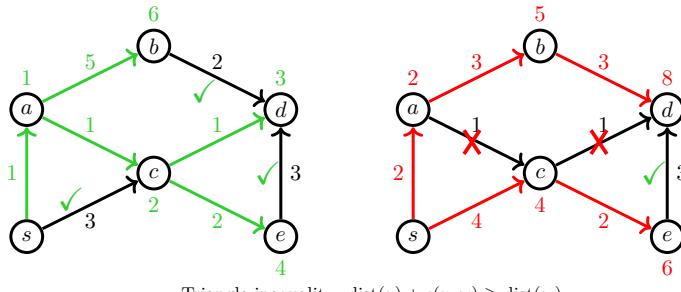
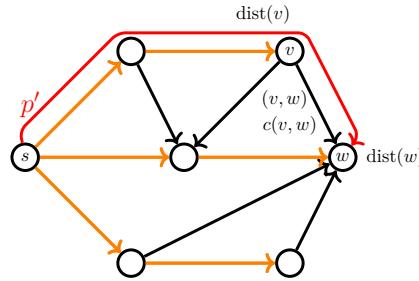


Fig. 4.10.: Using the theorem we can decide easily which tree is a shortest path tree (No. 816)

Proof. Induction on the length of the paths (number of arcs)

- (1) \Rightarrow (2): Let $(v, w) \in A$ be a non-tree arc
- Let $p_{[s,v]}$ and $p_{[s,w]}$ be the two paths in T from s to v, w respectively

Fig. 4.11.: Length of p' (No. 556)

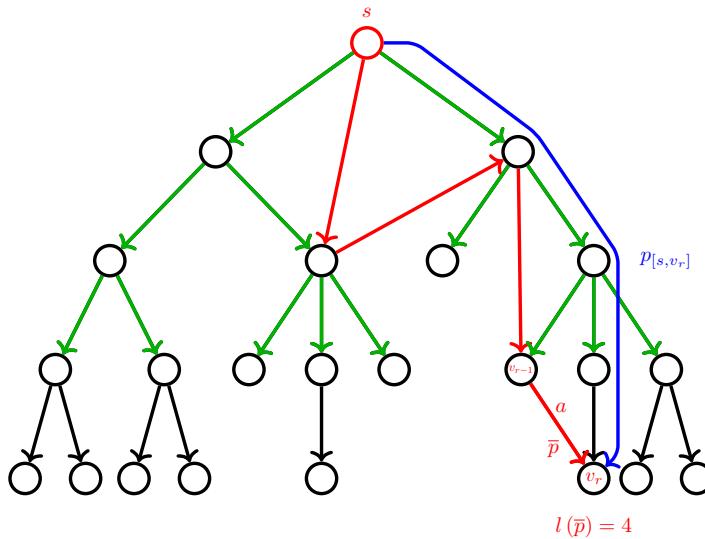
- Define $p' = p_{[s,v]} \cup (v,w)$
- Since T is a shortest path tree,

$$\text{dist}(w) \leq c(p') = \text{dist}(v) + c(v,w)$$

- (2) \Rightarrow (1): Let T be the considered tree
- Let $\bar{p} = sv_1 \dots v_r$ be a path in G
- We denote the length of the path with $\ell(\bar{p}) = r = |V(p)| - 1$
- Let $p_{[s,w]}$ denote the path in T from s to $w \in V$
- *Claim:* $c(\bar{p}) \geq c(p_{[s,v_r]})$ for $\bar{p} = sv_1 \dots v_r$

Proof of Claim: Induction on the length of \bar{p}

- I.B.: $r = 0$:
- Since $\bar{p} = s$ is the only path with length 0, $\Rightarrow c(\bar{p}) = 0 = c(p_{[s,s]}) = \text{dist}(s)$
- I.H.: For all paths \bar{p} of a fixed but arbitrary length $r - 1$ to vertex v , we have $c(\bar{p}) \geq c(p_{[s,v]})$
- I.S.: Let \bar{p} be a path in G from s to v_r with $\ell(p_{[s,v_r]}) = r$
- Let $a = (v_{r-1}, v_r)$ be the last arc of \bar{p}
- Then, $c(\bar{p}_{[s,v_{r-1}]}) \stackrel{\text{I.H.}}{\geq} c(p_{[s,v_{r-1}]}) = \text{dist}(v_{r-1})$

Fig. 4.12.: Induction on length of \bar{p} (No. 613)

• \Rightarrow

$$\begin{aligned}
 c(\bar{p}) &= c(\bar{p}_{[s, v_{r-1}]}) + c(a) \\
 &\stackrel{I.H.}{\geq} \text{dist}(v_{r-1}) + c(a) \\
 &\begin{cases} \geq \text{dist}(v_r) & \text{if } a \text{ non-tree edge (property)} \\ = \text{dist}(v_r) & \text{if } a \text{ tree edge} \end{cases} \\
 &\geq \text{dist}(v_r) = c(p_{[s, v_r]})
 \end{aligned}$$

□C

- By definition, T is a shortest path tree.

□

- Using this property, we can test any tree for optimality efficiently

4.2. Dijkstra's Algorithm

- Dijkstra¹ invented the following algorithm in 1956 and published it three years later to solve the shortest path problem
- The main idea is to keep a tree and update the tree if the triangle inequality is not satisfied
- In order to save the tree and test the triangle inequality, we use two labels for each vertex
 - $\text{dist}(v)$ “so far” shortest distance from s to a vertex v
 - $\text{pred}(v)$ the predecessor of v on a path p from s to v with $c(p) = \text{dist}(v)$

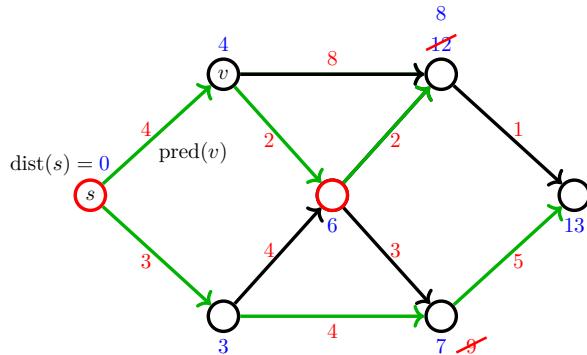


Fig. 4.13.: Idea Dijkstra (No. 614)

- Scan the vertices, to get the triangle inequality (optimality criteria) valid:
 - Chose a vertex v

¹What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, which I designed in about twenty minutes. One morning I was shopping in Amsterdam with my young fiancee, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities. Eventually, that algorithm became to my great amazement, one of the cornerstones of my fame. - Edsger Dijkstra, in an interview with Philip L. Frana, Communications of the ACM, 2001(6)

- Scan incident arcs (v, w)
- If $\text{dist}(w) > \text{dist}(v) + c(v, w)$, update $\text{dist}(w) = \text{dist}(v) + c(v, w)$ and set $\text{pred}(w) = v$
- The main question is in which order to do the scanning
- Dijkstra considered the problem with $c \geq 0$
- In that case, scanning according to $\text{dist}(v)$, i.e., choosing the next nearest vertex, leads to an efficient algorithm

Algo. 4.1 Dijkstra's Algorithm

Input: Digraph $G = (V, A)$ with positive costs $c : A \rightarrow \mathbb{R}_+$, a vertex $s \in V$ s.t. $\exists (s, v)$ -path in $G \forall v \in V$

Output: Shortest path tree T with root s

Method:

- Step 1
- Set $\text{dist}(v) = \infty$ for every $v \in V$ and $\text{dist}(s) = 0$
 - Set $\text{pred}(v) = \text{NULL}$ for every $v \in V$ and $\text{pred}(s) = s$
 - Set the set of not yet scanned vertices V' to V
- Step 2 **While** $\exists v \in V'$ with $\text{dist}(v) < \infty$ **do**
- Choose $v' \in V'$ with minimum $\text{dist}(v')$, i.e., $\text{dist}(v') \leq \text{dist}(v)$ for all $v \in V'$
 - **Forall** $w' \in V$ with $(v', w') \in A$ **do**
 - If $\text{dist}(v') + c((v', w')) < \text{dist}(w')$ **do**
 - Set $\text{dist}(w') = \text{dist}(v') + c((v', w'))$
 - Set $\text{pred}(w') = v'$ - Delete v' from V'
- Step 3 **Return** $T = (V, A')$ with $A' = \{(u, v) \in A \mid v \in V \setminus \{s\}, u = \text{pred}(v)\}$
-

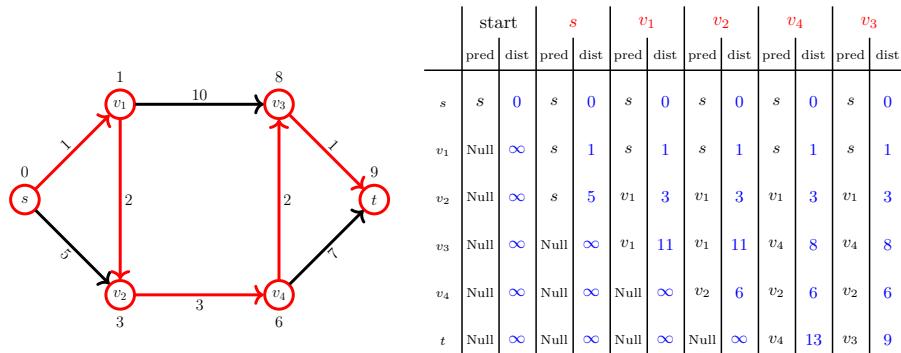
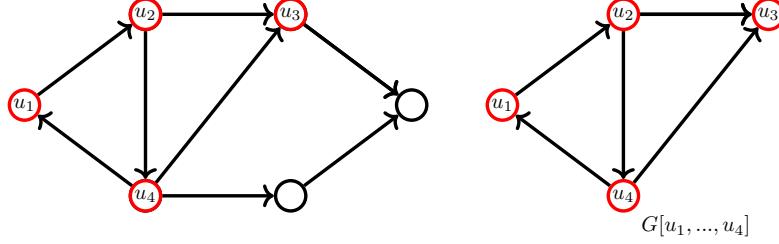


Fig. 4.14.: Dijkstra Algorithm (No. 557)

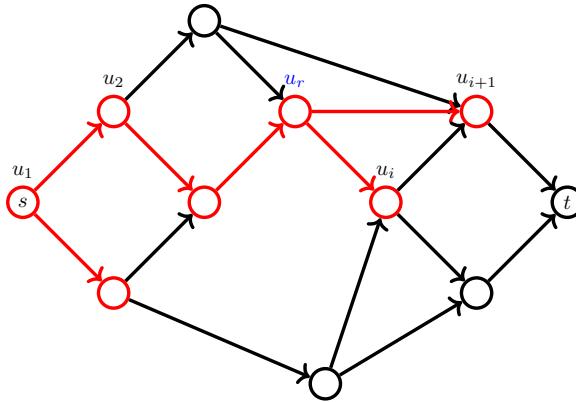
Theorem 45. Let $G = (V, A)$ be a digraph with positive costs and a root $s \in V$. The Dijkstra Algorithm computes a shortest path tree of G for all vertices which are reachable from the root s .

Proof. Induction on the number of visited vertices

- We denote with $G[V']$ the induced subgraph on $V' \subseteq V$ (see Definition IV)
- Let u_1, \dots, u_k be the order of scanned vertices
- We consider the different iterations and denote with $\text{dist}^i(v)$ the value of the label at the end of the i th iteration on vertex $v \in V$, i.e., after vertex v_i is scanned
- For each iteration i , we prove
 - A1 $\text{dist}^i(u_1) \leq \text{dist}^i(u_2) \leq \dots \leq \text{dist}^i(u_i)$
 - A2 $\text{dist}^i(u_i) \leq \text{dist}^i(v)$ for all $v \in V \setminus \{u_1, \dots, u_i\}$
 - A3 $\text{dist}^i(u_k)$ equals the cost of a shortest path from s to u_k in $G[u_1, \dots, u_i]$, $1 \leq k \leq i$

Fig. 4.15.: Notation $G[u_1, \dots, u_i]$ (No. 615)

- Note that $\text{dist}^r(u_i) = \text{dist}^i(u_i)$ for all $r \geq i$
- I.B.: $i = 1$
 - $u_1 = s$ and $\text{dist}^1(s) = 0$
 - Since we just consider positive cost, $\text{dist}^1(v) \geq 0$ for all $v \in V \setminus \{s\}$
 - \Rightarrow all assumptions A1, A2 and A3 hold true
- I.H.: Let A1, A2, A3 be true for $i \geq 1$
- I.S.: $i \rightarrow i + 1$
 - *Claim 1:* $\text{dist}^{i+1}(u_i) \leq \text{dist}^{i+1}(u_{i+1})$ (A1)
 - *Proof of Claim:*
 - Let $u_r = \text{pred}(u_{i+1})$
 - $\Rightarrow \text{dist}^{i+1}(u_r) + c(u_r, u_{i+1}) = \text{dist}^{i+1}(u_{i+1})$
 - Assume $\text{dist}^{i+1}(u_i) > \text{dist}^{i+1}(u_{i+1})$
 - $\Rightarrow i \neq r$, since $c \geq 0$
 - u_{i+1} is labeled with value $\text{dist}^{i+1}(u_{i+1})$ in the r th iteration

Fig. 4.16.: $\text{dist}(u_i) \leq \text{dist}(u_{i+1})$ (No. 616)

- $\Rightarrow u_{i+1}$ is chosen before u_i , contradiction
- \Rightarrow A1 holds true $\square C1$
- *Claim 2:* $\text{dist}^{i+1}(u_{i+1}) \leq \text{dist}^{i+1}(v)$ for all $v \in V \setminus \{u_1, \dots, u_i\}$ (A2)

Proof of Claim:

- Due to the choice of u_{i+1} , it holds $\text{dist}^i(u_{i+1}) \leq \text{dist}^i(v) \forall v \in V \setminus \{u_1, \dots, u_i\}$
- Furthermore,

$$\text{dist}^{i+1}(v) = \begin{cases} \min \{ \text{dist}^i(u_{i+1}) + c(u_{i+1}, v), \text{dist}^i(v) \} & \text{if } (u_{i+1}, v) \in A \\ \text{dist}^i(v) & \text{otherwise} \end{cases}$$

- Since $\text{dist}^{i+1}(u_{i+1}) = \text{dist}^i(u_{i+1})$ and $c \geq 0$, A2 holds true $\square C2$
- *Claim 3:* $\text{dist}^{i+1}(u_k)$ is the cost of a shortest path from s to u_k in $G[u_1, \dots, u_{i+1}]$, $1 \leq k \leq i+1$ (A3)

Proof of Claim

- We start by considering u_{i+1}
- Let p' be a shortest (s, u_{i+1}) -path in $G[u_1, \dots, u_{i+1}]$
- Let u_r be the predecessor of u_{i+1} on p'
- Since p' is a shortest path, a) $p'_{[u_1, u_r]}$ is a shortest path and b) $p'_{[u_1, u_r]}$ is a path in $G[u_1, \dots, u_i]$
- I.H.: $c(p'_{[u_1, u_r]}) = \text{dist}^i(u_r)$
- Thus,

$$\begin{aligned} c(p') &= c(p'_{[s, u_r]}) + c(u_r, u_{i+1}) \\ &\stackrel{I.H.}{=} \text{dist}^r(u_r) + c(u_r, u_{i+1}) \end{aligned}$$

- After scanning u_r , it holds true that

$$\text{dist}^r(u_{i+1}) \leq \text{dist}^r(u_r) + c(u_r, u_{i+1})$$

- \Rightarrow

$$\begin{aligned} \text{dist}^{i+1}(u_{i+1}) &\leq \text{dist}^r(u_{i+1}) \\ &\leq \text{dist}^r(u_r) + c(u_r, u_{i+1}) = c(p') \end{aligned}$$

- $\Rightarrow \text{dist}^{i+1}(u_{i+1})$ is the length of a shortest path in $G[u_1, \dots, u_{i+1}]$

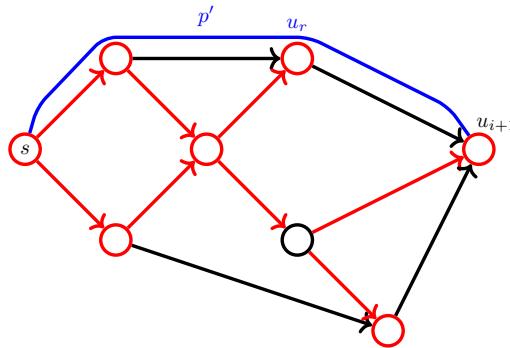


Fig. 4.17.: $\text{dist}^{i+1}(u_{i+1})$ is the length of a shortest path (No. 617)

- Consider any other vertex $u_k \in G[u_1, \dots, u_{i+1}]$

- Let p' be a shortest (s, u_k) -path in $G[u_1, \dots, u_{i+1}]$
- If p' is a path in $G[u_1, \dots, u_i]$, then $\text{dist}^{i+1}(u_k) = c(p')$ due to I.H.
- Otherwise, p' passes u_{i+1}
- Since $\text{dist}^{i+1}(u_{i+1}) \geq \text{dist}^{i+1}(u_k)$ and $\text{dist}^{i+1}(u_{i+1})$ equals the length of a shortest path in $G[u_1, \dots, u_{i+1}]$, this is a contradiction
- $\Rightarrow \text{dist}^{i+1}(u_k)$ equals the length of a shortest (s, u_k) -path in $G[u_1, \dots, u_{i+1}]$

□C3

- \Rightarrow after scanning all vertices we obtain a shortest path tree

□

- Note that we use at several points in the proof that $c \geq 0$

Theorem 46. *The Dijkstra Algorithm can be implemented with run-time $\mathcal{O}(m + n \log n)$*

Proof. Bound steps

- In total, at most m update steps are made
- Choosing the next vertex costs in total $n \cdot \log n$ by using Fibonacci Heaps
- \Rightarrow total run-time $\mathcal{O}(m + n \cdot \log n)$

□

- There are several methods to improve the run-time of Dijkstra's Algorithm like the A^* algorithm
- Often, there is a trade-off between speed and storage capacity
- With the storage capacity, certain shortest distances can be computed in a prerun

4.3. Moore-Bellman-Ford Algorithm

- There are cases, where the cost are negative
- Does that make a difference?

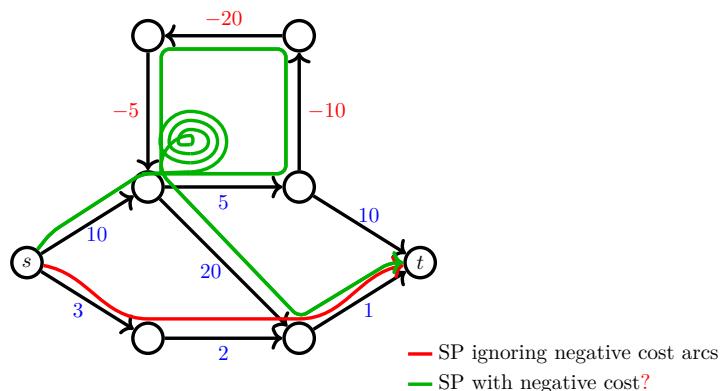


Fig. 4.18.: With negative costs a shortest path may not exist (No. 610)

- Dijkstra's Algorithm does not work for negative costs

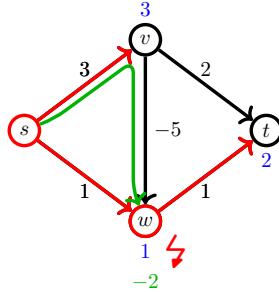


Fig. 4.19.: Dijkstra just for $c \geq 0$ (No. 618)

- The main problem is: although a vertex was scanned, the current distance may not be the shortest
- Richard Bellman, Lester Ford and Edward Moore independently published the following algorithm in 1958, 1956 and 1957 to compute a shortest path tree in a graph with negative costs or to compute a negative cycle
- **Note:** if a negative cycle in a graph exists, there is no shortest path tree
- The main extension to Dijkstra's Algorithm is that vertices are scanned several times
- However, after scanning each one n times, we either have a shortest path tree or a cycle with negative cost

Algo. 4.2 Moore-Bellman-Ford Algorithm

Input: Digraph $G = (V, A)$ with costs $c : A \rightarrow \mathbb{R}$ and vertex $s \in V$

Output: Shortest path tree T with root s or negative cycle

Method:

- Step 1 • For every $v \in V$ set $\text{dist}(v) = \infty$
 • Set $\text{dist}(s) = 0$
 • For every $v \in V$ set $\text{pred}(v) = \text{NULL}$ and $\text{pred}(s) = s$
- Step 2 **For** $i = 1, \dots, |V|$ **do**
 Forall $a = (v, w) \in A$ **do**
 If $\text{dist}(v) + c(a) < \text{dist}(w)$ **do**
 • Set $\text{dist}(w) = \text{dist}(v) + c(a)$
 • Set $\text{pred}(w) = v$
- Step 3 • Create the graph $T = (V, A')$ with $A' = \{(u, v) \in A \mid u = \text{pred}(v), v \in V \setminus \{s\}\}$
 • If T is a tree then T is a shortest path tree. Return T
 • Otherwise T contains a negative cycle. Return the cycle
-

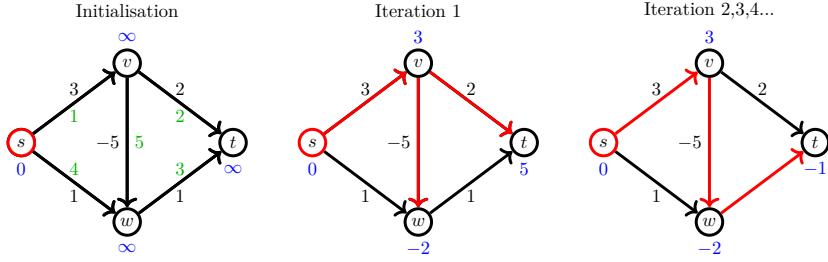


Fig. 4.20.: Moore-Bellman-Ford Algorithm, the green numbers show in which order we consider the arcs in the iterations (No. 619)

Theorem 47 (Moore-Bellman-Ford Algorithm). *The Moore-Bellman-Ford Algorithm computes a shortest path tree or a cycle with negative cost in $\mathcal{O}(m \cdot n)$ for a graph with m edges and n nodes.*

Proof. Induction on the number of iterations

- We denote with $\text{dist}^k(v)$ the label of v at the end of the k th iteration
- Let $\delta^\ell(v)$ denote the shortest path from s to v using at most ℓ edges
- For $\delta^\ell(v)$ it holds

$$\delta^\ell(v) = \min\{\delta^{\ell-1}(v), \min_{u \in V}\{\delta^{\ell-1}(u) + c(u, v)\}\}$$

- *Claim:* $\text{dist}^\ell(v) \leq \delta^\ell(v)$ for all $v \in V$ and $\ell \in \{0, 1, \dots, n\}$.

Proof of Claim:

- I.B.: for $\ell = 0$, i.e., the initialization, $\text{dist}^0(s) = 0$ and $\text{dist}^0(v) = \infty$
- I.S.: $k \rightarrow k + 1$
- Due to I.H. we have $\text{dist}^k(v) \leq \delta^k(v)$
- When updating the dist-values, we obtain

$$\begin{aligned} \text{dist}^{k+1}(v) &\leq \min\{\text{dist}^k(v), \min_{u \in V}\{\text{dist}^k(u) + c(u, v)\}\} \\ &\stackrel{\text{I.H.}}{\leq} \min\{\delta^k(v), \min_{u \in V}\{\delta^k(u) + c(u, v)\}\} \\ &= \delta^{k+1}(v). \end{aligned}$$

□C

- If T does not contain a cycle, then $\delta^n(v)$ equals the cost of a shortest path from s to v in G
- Let us assume T contains a cycle $C = v_1 v_2 \dots v_k$ with $v_1 = v_k$
- Since v_i is the predecessor of v_{i+1} , we have

$$\text{dist}^n(v_i) + c(v_i, v_{i+1}) \leq \text{dist}^n(v_{i+1})$$

- Let w.l.o.g. (v_{k-1}, v_k) be the last arc to be scanned in the last iteration on the cycle C and to update the value $\text{dist}(v_k)$
- $\Rightarrow (v_1, v_2)$ is scanned before (v_{k-1}, v_k)

- At that point

$$\text{dist}^{(v_1, v_2)}(v_1) + c(v_1, v_2) = \text{dist}^n(v_2)$$

- Scanning of (v_{k-1}, v_k) leads to set $\text{pred}(v_k) = v_{k-1}$ and to update the value $\text{dist}(v_k)$
- $\Rightarrow \text{dist}^n(v_1) < \text{dist}^{(v_1, v_2)}(v_1)$
- Consider the cost of C

$$\begin{aligned} c(C) &= \sum_{i=1}^{k-1} c(v_i, v_{i+1}) = c(v_1, v_2) + \sum_{i=2}^{k-1} c(v_i, v_{i+1}) \\ &\leq \text{dist}^n(v_2) - \text{dist}^{(v_1, v_2)}(v_1) + \sum_{i=2}^{k-1} \text{dist}^n(v_{i+1}) - \text{dist}^n(v_i) \\ &= \text{dist}^n(v_k) - \text{dist}^{(v_1, v_2)}(v_1) < 0 \end{aligned}$$

- $\Rightarrow C$ is a negative cycle

□

- The search of shortest paths is still a quite active field of studies
- Often further constraints, e.g., an extra length bound or non-linear costs, are added

4.4. All pair shortest paths

- Often not only the shortest paths from s to all vertices are interesting
- In many cases, one needs the distance between each pair of vertices

Cities	Jekaterinburg	Kaliningrad	Kasan	Moskau	Nischni Nowgorod	Rostow	Samara	St. Petersburg	Saransk	Sotschi	Wolgograd
Jekaterinburg		2484	717	1418	1016	1774	779	1783	1013	2078	1405
Kaliningrad	3036		1805	1088	1483	1574	1933	826	1589	1856	1773
Kasan	943	2085		719	323	1152	295	1200	309	1513	847
Moskau	1772	1261	821		402	959	856	635	512	1362	913
Nischni Nowgorod	1339	1686	389	417		1054	526	896	250	1449	848
Rostow	2230	2327	808	1076	407		926	1541	864	405	395
Samara	949	2327	368	1050	681	1096		1420	345	1315	636
St. Petersburg	2225	964	1520	709	1119	766	1768		1099	1926	1545
Saransk	1315	1926	396	649	283	682	518	1369		1244	610
Sotschi	2776	2874	2055	1622	1808	1829	1875	2343	1626		679

Distances in km by

plane

car

Fig. 4.21.: Distance between all WM locations 2018 in Russia (No. 780)

Definition 48. All pair shortest path (SP) problem

Given: Digraph $G = (V, A)$, costs $c : A \rightarrow \mathbb{R}$

Find: Between each pair of vertices $u, v \in V$ a shortest (u, v) -path, if one exists

- If we have arbitrary arc cost, we can compute these paths in $\mathcal{O}(m \cdot n^2)$ (use n times Moore-Bellman-Ford ??)

Adaption to use Dijkstra

- However, with a slight adaption of the instance, we can compute all pairs in $\mathcal{O}(n^3)$
- Idea: instead of using Moore-Bellman-Ford, we use Dijkstra
- To that end: use new cost c' instead of c in the directed graph G such that
 - $c'(a) \geq 0 \forall a \in A(G)$
 - if p is a shortest (u, v) -path according to c' , then p is a shortest (u, v) -path according to c

Algo. 4.3 All pair shortest path algorithm

Input: Digraph $G = (V, A)$ with costs $c : A \rightarrow \mathbb{R}$

Output: Shortest path tree T_v for every $v \in V$ or a negative cycle

Method:

- Step 1
 - Add vertex s and (s, v) arcs to G
 - Set $c(s, v) = 0 \forall v \in V(G)$
 - Step 2
 - Compute the shortest path length $\text{dist}_s(v)$ for all $v \in V(G)$ by Moore-Bellman-Ford (Alg. ??)
 - If a negative cycle is detected **return** negative cycle
 - Else set $c'(v, w) = c(v, w) + \text{dist}_s(v) - \text{dist}_s(w) \forall (v, w) \in A(G)$
 - Step 3
 - Compute for every $v \in V(G) \setminus \{s\}$ a shortest path tree T_v by Dijkstra's Algorithm (Alg. 4.1)
-

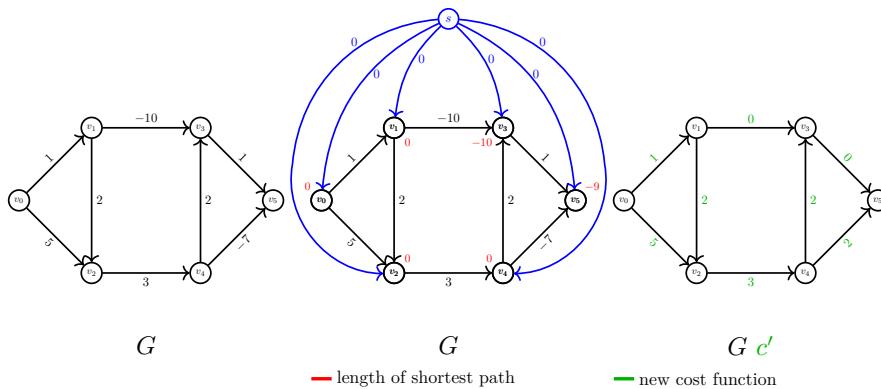


Fig. 4.22.: All pair shortest path algorithm (No. 781)

- We need to prove the correctness of the algorithm

Theorem 49. Let $G = (V, A)$ be a directed graph and $c : A \rightarrow \mathbb{R}$ a cost function. Then the all pair shortest path algorithm solves the all pair shortest path problem in $\mathcal{O}(n^3)$.

Proof. Definition of c'

- *Claim 1:* $c'(v, w) \geq 0$ for all $(v, w) \in A(G)$.
- *Proof of Claim:*

- $\text{dist}_s(v)$ represents the shortest path length from s to v in G where s was added

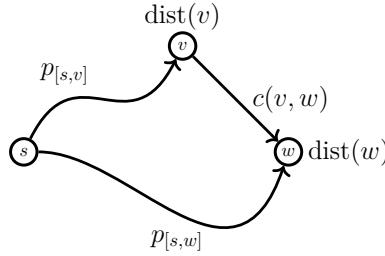


Fig. 4.23.: The detour over at least two edges to a vertex is greater than or equal to the direct connection (No. 555)

- Optimality criterion for shortest paths (Theorem 44):

$$c(v, w) + \text{dist}_s(v) \geq \text{dist}_s(w)$$

$$\bullet \Rightarrow c'(v, w) = c(v, w) + \text{dist}_s(v) - \text{dist}_s(w) \geq 0 \quad \square C1$$

- *Claim 2:* p is a shortest (u, w) -path according to c' if and only if p is a shortest (u, w) -path according to c .

Proof of Claim:

- Let $p = v_1 \dots v_k$ be a path in G
- Then,

$$\begin{aligned} c'(p) &= c'(v_1, v_2) + c'(v_2, v_3) + \dots + c'(v_{k-1}, v_k) \\ &\stackrel{(a)}{=} c(v_1, v_2) + \text{dist}_s(v_1) - \text{dist}_s(v_2) \\ &\quad + c(v_2, v_3) + \text{dist}_s(v_2) - \text{dist}_s(v_3) + \dots \\ &\quad + c(v_{k-1}, v_k) + \text{dist}_s(v_{k-1}) - \text{dist}_s(v_k) \\ &= c(p) + \text{dist}_s(v_1) - \text{dist}_s(v_k) \end{aligned}$$

(a): Definition of $c'(v, w) = c(v, w) + \text{dist}_s(v) - \text{dist}_s(w)$

- claim holds true

$\square C1$

- The run-time consists of
 - one run of the Moore-Bellman-Ford algorithm: $\mathcal{O}(m \cdot n)$
 - n runs of the Dijkstra algorithm: $\mathcal{O}(n \cdot n^2)$
- \Rightarrow total run-time is $\mathcal{O}(n^3)$

\square

Floyd-Warshall Algorithm

- A different algorithm was proposed by Floyd-Warshall based on a simple dynamic program
- Dynamic programming was developed by Richard Bellman in the 1950s
- The main idea is to
 - Step 1: simplify a complicated problem by breaking it down into simpler sub-problems in a recursive manner

- Step 2: Use an optimal solutions of the sub-problems to construct an optimal solution of the original problem
- Consider now the all-pair shortest path problem
- Let $V = \{1, \dots, n\}$, i.e., we assume the vertices are ordered
- The idea is the following:
 - Let $p^k(i, j)$ be a shortest path between i and j using only vertices $1, \dots, k$ and i and j
 - Then,

$$p^{k+1}(i, j) = \arg \min \{c(p^k(i, j)), c(p^k(i, k+1)) + c(p^k(k+1, j))\}$$

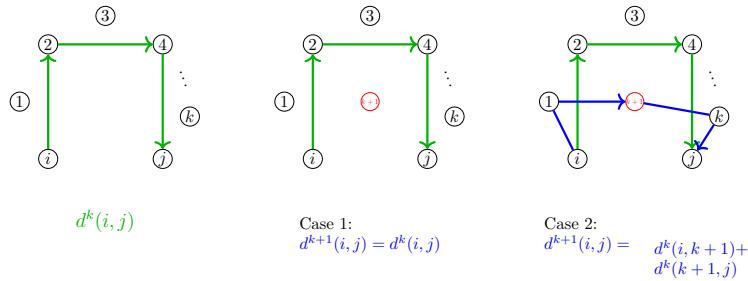


Fig. 4.24.: Recursion by Floyd-Warshall (No. 782)

- Instead of tracking all paths, we just save the corresponding costs

Algo. 4.4 Floyd-Warshall algorithm

Input: Digraph $G = (V, A)$ with costs $c : A \rightarrow \mathbb{R}$ with $V = \{1, \dots, n\}$

Output: shortest distances $\text{dist}_u(v)$ for all $u, v \in V$, if they exist

Method:

- Step 1 Set $d^0(i, j) = c(i, j)$ if $(i, j) \in A(G)$ and $d^0(i, j) = \infty$ otherwise
Set $d^0(i, i) = 0 \forall i \in V$
 - Step 2 **For** $k = 1, \dots, n$ **do**
Forall $i, j \in V$ **do**
Set $d^k(i, j) = \min \{d^{k-1}(i, j), d^{k-1}(i, k) + d^{k-1}(k, j)\}$
 - Step 3 **If** $d^n(i, i) = 0 \forall i \in V$ **return** $d^n(i, j)$ for all $i, j \in V$
Else return a negative cycle exists
-

- Using predecessors as in the Dijkstra algorithm, we can also compute the corresponding shortest paths

Theorem 50. *The Floyd-Warshall algorithm solves the all pair shortest path problem in $\mathcal{O}(n^3)$.*

Proof. Exercise (prove Claim 1 and 2)

- *Claim 1:* $d^k(i, i) = 0$ for all $i \in V$, $k = 1, \dots, n \Leftrightarrow$ there is no negative cycle in G

- *Claim 2:* $d^n(i, j)$ represents the shortest distance between i, j , if no negative cycle exists.

□



5. Maximum (s, t) -Flows

5.1. (s, t) -Flows and the Decomposition Theorem

- Classical problem setting: Transportation of products through a road-network with limited capacity
 - Roads with certain capacity: Modeled by a graph
 - Transportation of goods from one source to a target: Can be modeled by paths

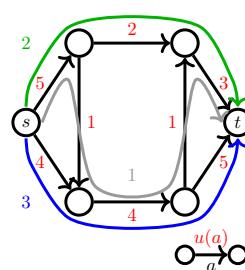
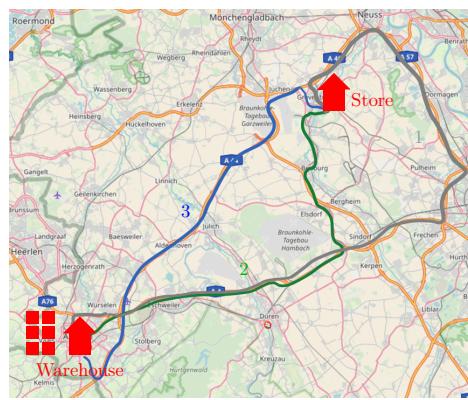


Fig. 5.1.: Flows model the transportation of goods in a network (No. 560)

- Several different questions turn up
 - What is the maximum amount that can be transported?
 - How can we transport everything with minimum cost?
 - How much time do we need to transport everything?
- Several variants: static vs. dynamic (i.e., time dependent flow and travel times), cost for transporting goods, different goods with different sources and targets, ...
- Start with the most basic part: How can we formally model the transportation of one good in a network?

Definition 51 ((Flow-) Network). A *(flow-) network* (G, u, s, t) consists of a

- directed graph $G = (V, A)$
- arc capacities $u(a) \geq 0$ for all $a \in A(G)$
- two specified vertices s (*source*) and t (*target/sink*)

- We assume in the following that $s \neq t$
- The capacities model how much may be sent over an arc

Definition 52 $((s, t)$ -Path Flow). Let (G, u, s, t) be a network. Let $\mathcal{P}_{(s,t)}$ be the set of all (s, t) -paths in G and $\mathcal{P}^a \subseteq \mathcal{P}_{(s,t)}$ the set of all (s, t) -paths traversing a . A function $x : \mathcal{P} \rightarrow \mathbb{R}_+$ is an (s, t) -path flow, if the *capacity constraint* is satisfied for all arcs, i.e.,

$$\sum_{p \in \mathcal{P}^a} x(p) \leq u(a) \quad \forall a \in A.$$

The *value* of x is the amount of goods transported, i.e., $\text{val}(x) = \sum_{p \in \mathcal{P}_{(s,t)}} x(p)$.

- An (s, t) -path flow models the transportation of our good in the network
- $x(p)$ units of the flow are sent along the path $p \in \mathcal{P}_{(s,t)}$ from s to t
- In total, $\text{val}(x)$ is transported from s to t

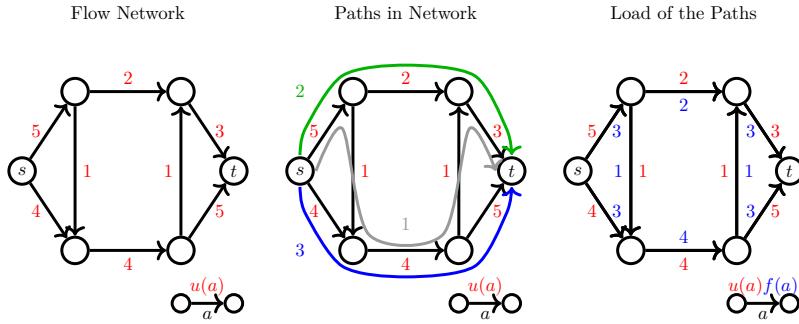


Fig. 5.2.: Flow-networks, paths and load (No. 1063)

- Problem: Too many paths
 - There may be exponentially many (s, t) -paths in G
 - Hence, a feasible (s, t) -path flow may assign to all of them positive flow
 - \Rightarrow not practical

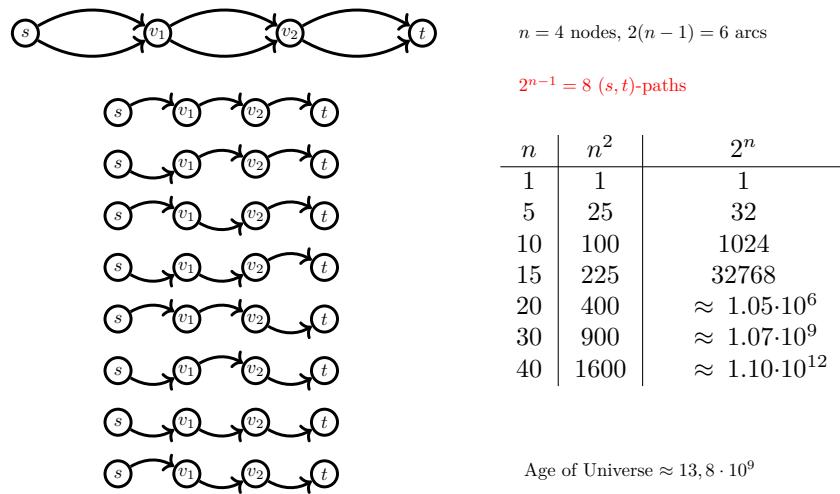


Fig. 5.3.: Paths flows may consist of exponentially many paths (No. 562)

- Consider an aggregation of the flow: Just store the load on every arc

Definition 53 $((s, t)$ -Flow). Given a flow network (G, u, s, t) . An (s, t) -flow is a function $f : A(G) \rightarrow \mathbb{R}_{\geq 0}$ which satisfies

- the *capacity constraint* at every arc, i.e., $f(a) \leq u(a) \forall a \in A(G)$
- the *flow conservation* at every vertex $v \in V \setminus \{s, t\}$, (in-flow = out-flow):

$$\sum_{a \in \delta^-(v)} f(a) = \sum_{a \in \delta^+(v)} f(a).$$

where $\delta^+(v)$ denotes all out-going arcs of $v \in V$ and $\delta^-(v)$ all in-coming arcs of $v \in V$

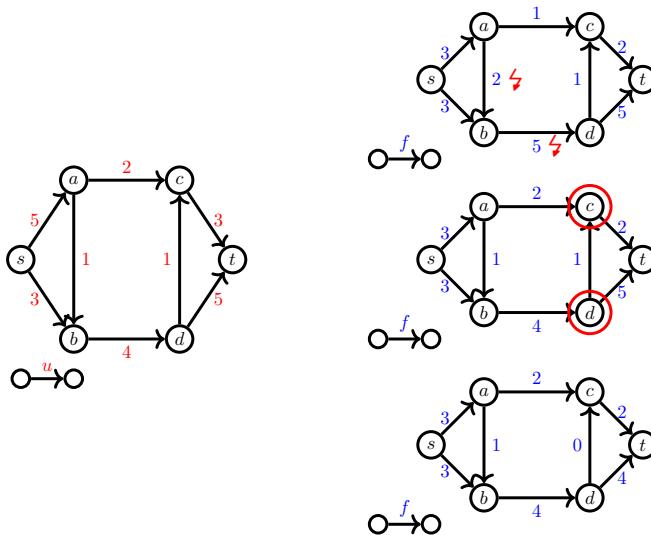


Fig. 5.4.: valid and invalid (s, t) -flows (No. 563)

- We have two concepts: (s, t) -path flows and (s, t) -flows
- (s, t) -path flows model more realistic the transportation
- What is the relation between the two?
- This is answered by the Flow Decomposition Theorem by Ford and Fulkerson

Theorem 54 (Flow Decomposition Theorem, Ford-Fulkerson 1962). *Let f be an (s, t) -flow in the network (G, u, s, t) . Then, f is a positive linear combination of directed (s, t) -paths and directed circles in G . Here, the number of paths and circles can be limited by $|A(G)|$. If f is integer, then the coefficients in the linear combination can be selected as integers.*

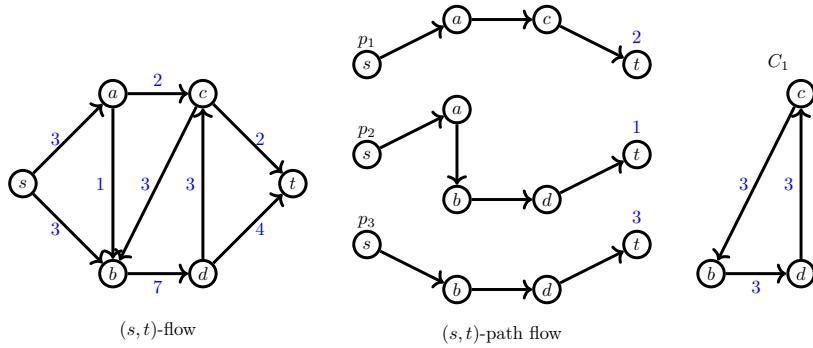


Fig. 5.5.: flow decomposition (No. 564)

Proof. Induction on number of arcs with positive flow

- Let f be an (s, t) -flow
- Let n' be the number of arcs with $f(a) > 0$, $a \in A(G)$
- Induction on n'
- I.B.: $n' = 1$:
 - Since f is an (s, t) -flow, (s, t) is the only arc with $f(a) > 0$
 - $\Rightarrow (s, t) \in A(G)$ and $p = (s, t) \in \mathcal{P}_{(s,t)}$
 - $\Rightarrow f$ is a linear combination of p , where $f(a)$ units of flow are sent along p

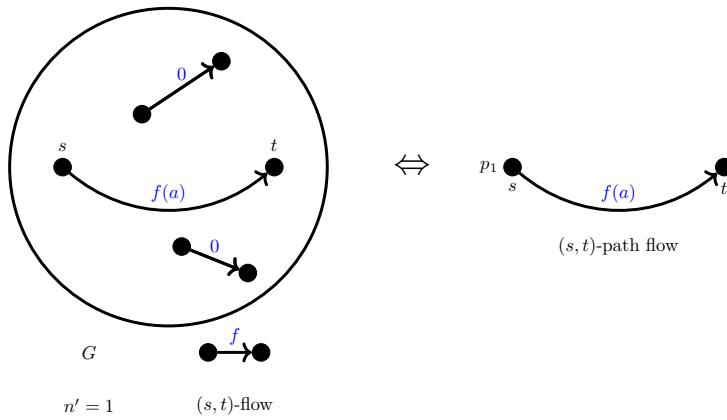


Fig. 5.6.: Induction basis (No. 565)

- I.H.: There exists an appropriate decomposition for an arbitrary, but fixed $n' \leq n$
- I.S.: Show the statement for $n + 1$
 - Let $a_1 = (v_1, u_1)$ be an arc with $f(a_1) > 0$

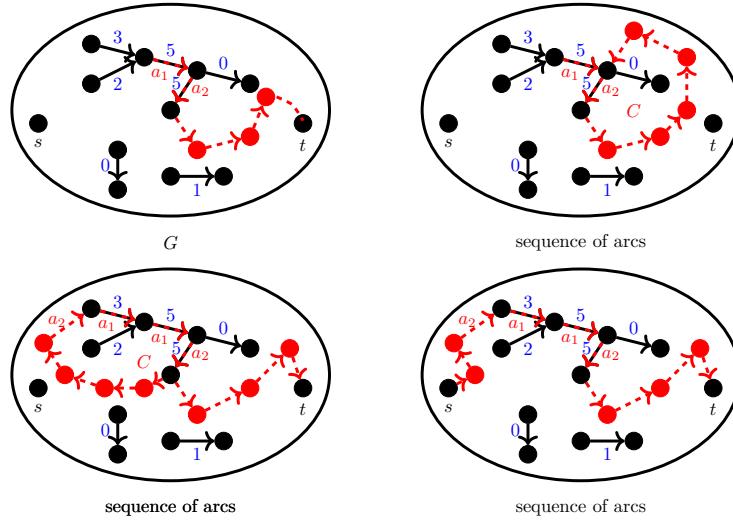


Fig. 5.7.: induction step (No. 566)

- Since f is a feasible (s, t) -flow, flow conservation is valid
 - Unless $u_1 \neq t$, \exists arc $a = (v_2, u_2) \in A(G)$ with $f(a) > 0$ and $u_1 = v_2$
 - Analogously, unless $v_1 \neq s$, \exists arc $a = (v_{-1}, u_{-1})$ with $f(a) > 0$ and $u_{-1} = v_1$
 - Let $a_{-k'}, \dots, a_{-1}, a_1, \dots, a_k$ be a sequence of consecutive arcs with $f(a_i) > 0$ such that $v_{-k'} = s$ or $v_{-k'} \in \{u_{-k'+1}, \dots, u_k\}$, and $u_k = t$ or $u_k \in \{v_{-k'}, \dots, v_{k-1}\}$
 - Case 1: $v_{-k'} = s$ and $u_k = t$
 - \Rightarrow the sequence of arcs is an (s, t) -path p'
 - Let $f_{\min} = \min_{a \in A(p')} f(a) > 0$
 - Define an arc-function $f' : A \rightarrow \mathbb{R}$ with
- $$f'(a) := \begin{cases} f(a) & \text{if } a \notin p' \\ f(a) - f_{\min} & \text{if } a \in p' \end{cases}$$
- f' is an (s, t) -flow since
 - $0 \leq f(a) \leq u(a)$
 - flow conservation holds
 - For f' there is at least one arc less in G with $f' > 0$
 - \Rightarrow Hypothesis holds and a linear combination of f' exists
 - Then, p' multiplied with f_{\min} plus the linear combination of f' is a linear combination of f
 - Case 2: Otherwise
 - Then $a_{-k'}, \dots, a_k$ contains a directed cycle C
 - Proceed as in case 1, but delete the cycle instead of the path
 - In every step of the induction, one path or cycle is added and n' is reduced by at least one
 - Since the maximum number of arcs with positive flow is $|A(G)|$, the maximum number of necessary paths and cycles is $|A(G)|$
 - Due to the construction, f_{\min} is integer if $f(a)$ is integer

□

- Hence, the problem of transporting goods from a source to a sink can be modeled by an (s, t) -flow
- We define the value of an (s, t) -flow as the net outflow of the source

Definition 55 (Flow value). The *value* $\text{val}(f)$ of an (s, t) -flow f is the net outflow from the source, i.e.,

$$\text{val}(f) := \sum_{a \in \delta^+(s)} f(a) - \sum_{a \in \delta^-(s)} f(a)$$

- Due to the Flow Decomposition Theorem we know, that $\text{val}(f)$ of an (s, t) -flow can be transformed into an (s, t) -path flow transporting $\text{val}(f)$ unites of flow

5.2. Ford-Fulkerson Algorithm

- A classical question in flow theory is how much flow can be sent through a given network

Definition 56 (Maximum Flow Problem (MFP)).

Given: Network (G, u, s, t)

Find: An (s, t) -flow of maximum value

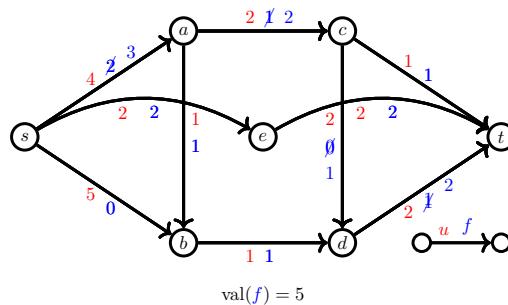


Fig. 5.8.: Maximum (s, t) -flow problem (No. 567)

- How can we solve the problem?
- Idea of an Algorithm:
 - Start sending flow along (s, t) -paths
 - Update the remaining capacity
 - Problem: if we send flow “the wrong way”, we may block a better flow
 - Solution: model sending flow back in the network by a reverse arc

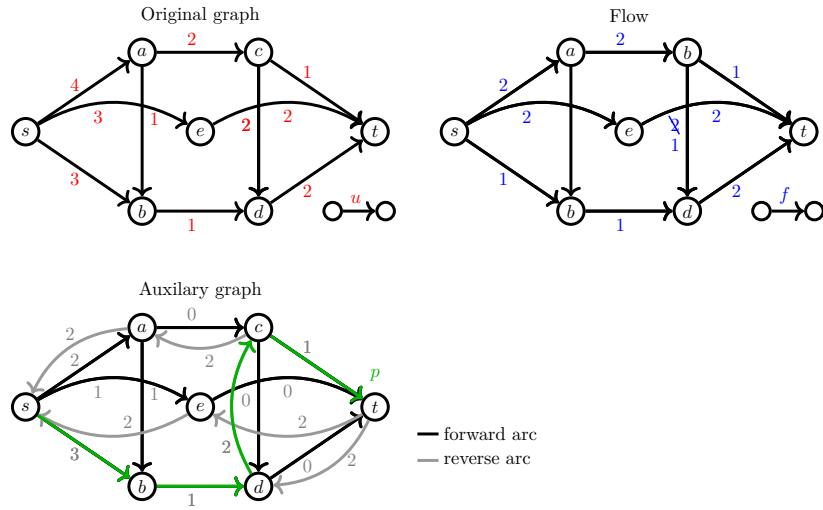


Fig. 5.9.: Idea Ford-Fulkerson Algorithm (No. 582)

- We now need to formalize this idea

Definition 57 (Reversed graph). For a digraph G , we define $\overset{\leftrightarrow}{G}$ by adding all arcs in reverse direction (parallel arcs possible)

- the initial arc $a \in A(G)$ is called a *forward arc*
 - the added arc \overleftarrow{a} in reverse direction is called a *reverse arc*

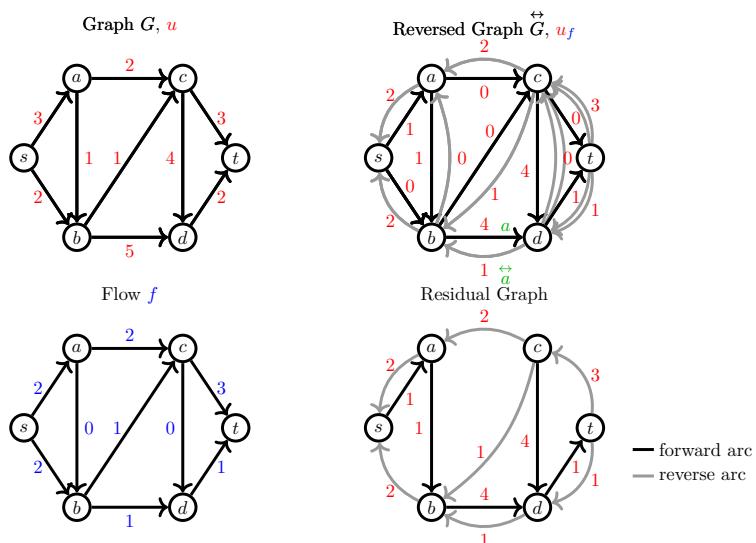


Fig. 5.10.: Reversed graph, forward and reverse arc (No. 568)

- There is always a unique assignment between an arc and its reverse arc
 - Note that $\overset{\leftrightarrow}{G}$ may contain parallel arcs
 - Given an (s, t) -flow, we need to keep track how much flow can be sent along these arcs

Definition 58 (Residual capacity). Let f be an (s, t) -flow in network (G, u, s, t) . We define the *residual capacities* u_f on $\overset{\leftrightarrow}{G}$ regarding u and f by

- $u_f(a) := u(a) - f(a)$ for all $a \in A(G)$
- $u_f(\overleftarrow{a}) := f(a)$ for all $a \in A(G)$

- Since many arcs of $\overset{\leftrightarrow}{G}$ with a flow f have capacity 0, we just consider a subgraph

Definition 59 (Residual graph). The *residual graph* belonging to G, u and f is the subgraph G_f of $\overset{\leftrightarrow}{G}$ with

- $V(G_f) := V(G)$
- $A(G_f) :=$ all arcs $a \in A(\overset{\leftrightarrow}{G})$ with $u_f(a) > 0$

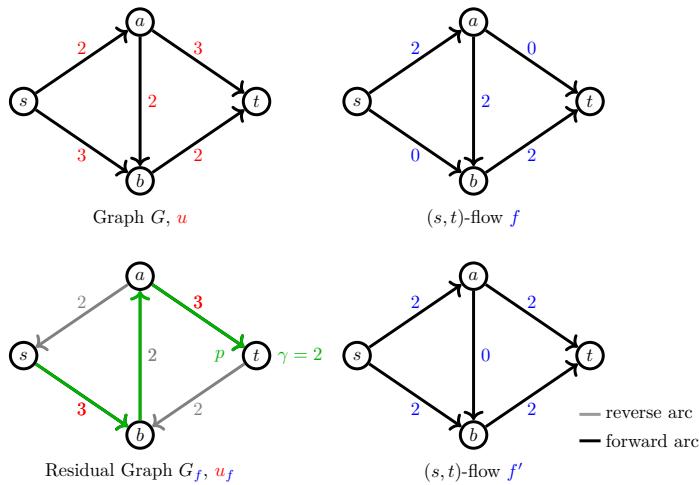


Fig. 5.11.: Residual graph (No. 583)

- We can now change a flow by sending flow along an (s, t) -path in the residual graph
 - sending along a forward arc leads to an increase of flow on that arc
 - sending along a reverse arc leads to a decrease of flow on that arc

Definition 60 (f -augmenting path). An f -augmenting path is an elementary (s, t) -path in the residual graph G_f . Augmenting a flow f along an (s, t) -path p by the value $\gamma \leq \min_{a \in A(p)} u_f(a)$ means to construct a flow f' in G via

- $f'(a) := f(a) + \gamma$, if $a \in A(p)$ and a is forward arc
- $f'(a) := f(a) - \gamma$, if $\overleftarrow{a} \in A(p)$ and a is the reverse arc
- $f'(a) := f(a)$, otherwise.

- Show that augmenting a flow is well defined and leads to a feasible (s, t) -flow

Lemma 61. Let f be a feasible (s, t) -flow in G and p be an (s, t) -path in G_f . Let $0 \leq \gamma \leq \min_{a \in p} \{u_f(a)\}$ and f' be the by γ augmented flow along p . Then f' is a feasible (s, t) -flow in G with $\text{val}(f') = \text{val}(f) + \gamma$.

Proof. Check constraints of a flow

- f' is a function assigning values to all arcs $a \in A$

- *Claim 1:* f' satisfies the capacity constraints.

Proof of Claim:

- Let $a \in A(p)$ be a forward arc
- Since $f(a) \geq 0$ and $f'(a) = f(a) + \gamma \Rightarrow f'(a) \geq 0$
- Furthermore,

$$f'(a) = f(a) + \gamma \leq f(a) + u_f(a) = f(a) + u(a) - f(a) = u(a)$$

- Let $a \in A(p)$ be a reverse arc
- Since $f(a) \leq u(a)$ and $f'(a) = f(a) - \gamma \Rightarrow f'(a) \leq u(a)$
- Furthermore,

$$f'(a) = f(a) - \gamma \geq f(a) - u_f(a) = f(a) - f(a) = 0$$

□C1

- *Claim 2:* f' satisfies the flow conservation at every vertex.

Proof of Claim:

- Let p be the sequence of arcs a_1, a_2, \dots, a_k with $a_i = (v_i, u_i)$, $u_i = v_{i+1}$, $v_1 = s$, $u_k = t$ and $v_i, u_i \in V \setminus \{s, t\}$
- If $a_i \in \delta_{G_f}^-(v_{i+1}) \Rightarrow a_{i+1} \in \delta_{G_f}^+(v_{i+1})$, $1 \leq i \leq k - 1$

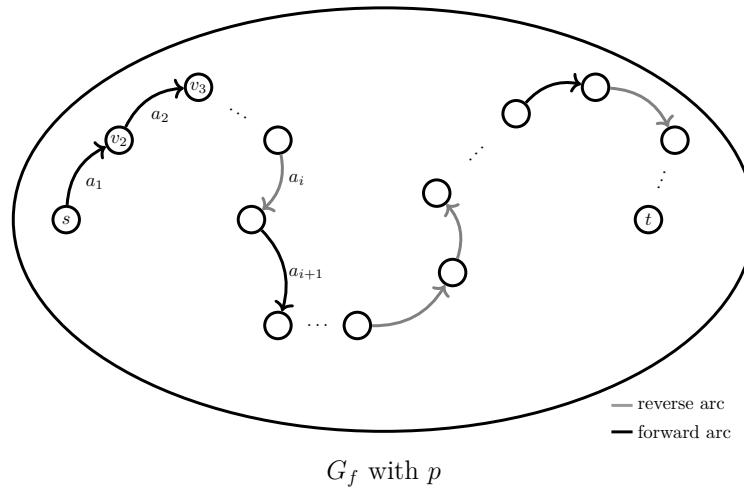


Fig. 5.12.: Flow Conservation (No. 584)

- Let $i \in \{1, \dots, k - 1\}$
- There are four combinations of $a_i, a_{i+1} \in \{\text{forward arc, reverse arc}\}$

- Case 1: a_i is forward arc, a_{i+1} is also a forward arc

- Definition of f' :

$$f'(a_i) = f(a_i) + \gamma \text{ and } f'(a_{i+1}) = f(a_{i+1}) + \gamma$$

- $\Rightarrow f'(a_i) - f'(a_{i+1}) = f(a_i) - f(a_{i+1})$
- \Rightarrow the change of the in-flow and out-flow remains the same

- Case 2: a_i is forward arc, a_{i+1} is reverse arc: analogous
- Case 3: a_i is reverse arc, a_{i+1} is forward arc: analogous
- Case 4: a_i is reverse arc, a_{i+1} is reverse arc: analogous
- Since f satisfies flow conservation, also f' satisfies flow conservation $\square C2$

- Consider the value of f'

- Let a_1 the first arc of p
- Since an f -augmenting path is always an elementary path, a_1 is a forward arc in G
- $\Rightarrow f'(a_1) = f(a_1) + \gamma$
- Due to the definition of f' ,

$$\begin{aligned} \text{val}(f') &= \sum_{a \in \delta^+(s)} f'(a) - \sum_{a \in \delta^-(s)} f'(a) \\ &= \sum_{a \in \delta^+(s) \setminus A(p)} f(a) - \sum_{a \in \delta^-(s) \setminus A(p)} f(a) + f'(a_1) \\ &= \sum_{a \in \delta^+(s) \setminus A(p)} f(a) - \sum_{a \in \delta^-(s) \setminus A(p)} f(a) + f(a_1) + \gamma \\ &= \text{val}(f) + \gamma \end{aligned}$$

\square

- Using this insight we obtain a necessary condition for an optimal flow

Lemma 62. *If f is a maximum flow, there exists no f -augmenting path in the residual graph G_f .*

- Combining all ideas and definitions lead to the following algorithm by Ford and Fulkerson

Algorithm 5.1 Ford-Fulkerson Algorithm

Input: Network (G, u, s, t)

Output: Maximum (s, t) -flow f

Initialization: Set $f(a) := 0$ for all $a \in A(G)$

Method: **While** there is an f -augmenting (s, t) -path **do**

- Determine an f -augmenting path p
- Set $\gamma := \min\{u_f(a) \mid a \in A(p)\}$
- Augment f along p by γ

Return f

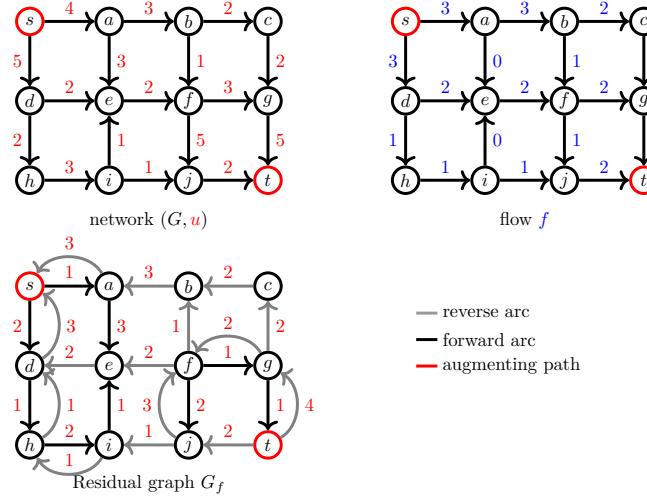


Fig. 5.13.: Ford-Fulkerson Algorithm (No. 600)

- Notes on the algorithm:

1. The construction of an f -augmenting (s, t) -paths can be done with a Breadth-First-Search or Depth-First-Search in $O(m)$ time
2. The increment value γ can be small and thus the algorithm may require many iterations
 - In the worst case: A sequence of $2U$ augmenting paths with value $\gamma = 1$, where U is an upper capacity (see example below)
 - But two paths are sufficient
 - Even worse: With irrational capacities there are examples where the algorithm does not terminate and the sequence of flow values does not converge towards the optimal flow value

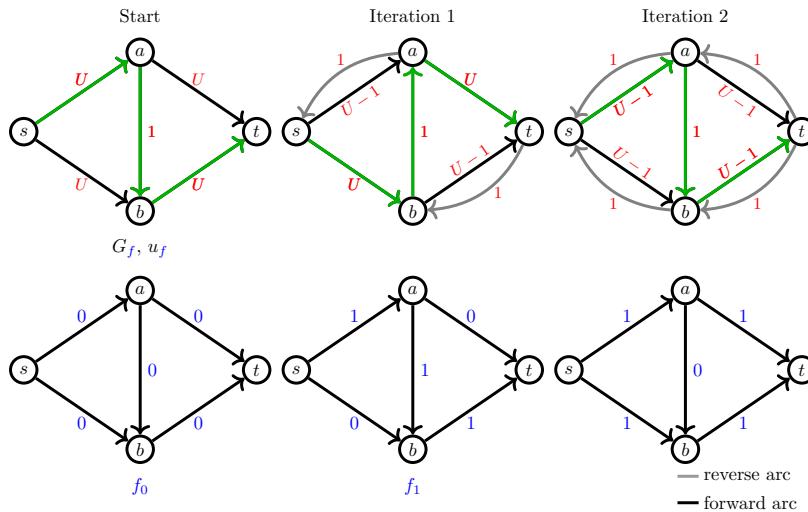


Fig. 5.14.: 2U iterations (No. 585)

- Does the algorithm terminate if we have integer capacities?

Theorem 63 (Integral Flow Theorem). *If the capacities of a network (G, u, s, t) are integers, then the Ford-Fulkerson Algorithm terminates in $\mathcal{O}(u_{\max} \cdot |V(G)| \cdot |A(G)|)$ time with a flow value $\text{val}(f) \leq |V(G)| \cdot u_{\max}$, where $u_{\max} := \max_{a \in A} u(a)$.*

Proof. Bound on maximum flow

- Let f be a feasible (s, t) -flow
- Then,

$$\begin{aligned} \text{val}(f) &:= \sum_{a \in \delta^+(s)} f(a) - \sum_{a \in \delta^-(s)} f(a) \\ &\leq \sum_{a \in \delta^+(s)} u(a) - \sum_{a \in \delta^-(s)} 0 \\ &\leq \sum_{a \in \delta^+(s)} u_{\max} \leq u_{\max} \cdot |V(G)| \end{aligned}$$

- In each iteration, $\gamma \geq 1$ due to the definition of u_f and $u(a) \in \mathbb{N}$
- \Rightarrow at most $u_{\max} \cdot |V(G)|$ iterations are necessary
- Each iteration needs the computation of one (s, t) -path: $\mathcal{O}(|A(G)|)$
- \Rightarrow in total: $\mathcal{O}(u_{\max} \cdot |V(G)| \cdot |A(G)|)$

□

- We now need to prove the maximality of the flow
- To that end, we use an important concept in optimization that is called duality

5.3. Max-Flow-Min-Cut Theorem

- The Max-Flow-Min-Cut Theorem is one of the most famous examples in duality theory

Duality

- Important concept in optimization
- Consider a maximization problem $I \in \mathcal{I}$:
Find a feasible solution $x \in X$ with maximum value $b(x)$, i.e., solve

$$\max \{b(x) \mid x \in X\}$$

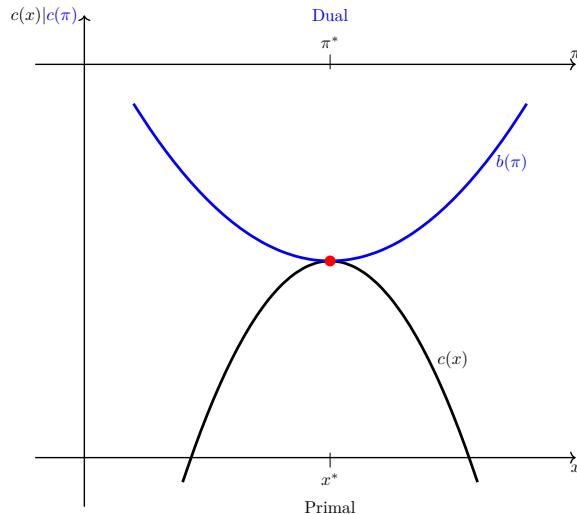


Fig. 5.15.: Primal vs. dual (No. 703)

- Drawbacks:
 - Some problems are difficult to solve
 - Often, it suffices to know the best value of the objective function, e.g., the analysis of run-time
- \Rightarrow it suffices to find upper bounds on $b_{\max} := \max\{b(x) \mid x \in X\}$
- Idea: Construct a different (sometimes easier optimization problem) $I' \in \mathcal{I}'$ with a set of feasible solutions Y and an objective function $c(y)$, $y \in Y$, such that $c(y)$ is an upper bound on b_{\max}
- If these problems I and I' are “related” as mentioned above, we call them dual pairs
- *Weak duality*: For all $x \in X$ and $y \in Y$, it holds that $b(x) \leq c(y)$
- *Strong duality*: It holds that

$$\max\{b(x) \mid x \in X\} = \min\{c(y) \mid y \in Y\}$$

- We will use that concept of duality to prove the optimality of the flow constructed by the Ford-Fulkerson Algorithm

Minimum (s, t) -Cuts

- Need to find a dual problem for the maximum flow problem
- When analyzing the run-time of the Ford-Fulkersons Algorithm, we already used some bound on the maximum amount of flow
- The question is: Can we improve this bound?

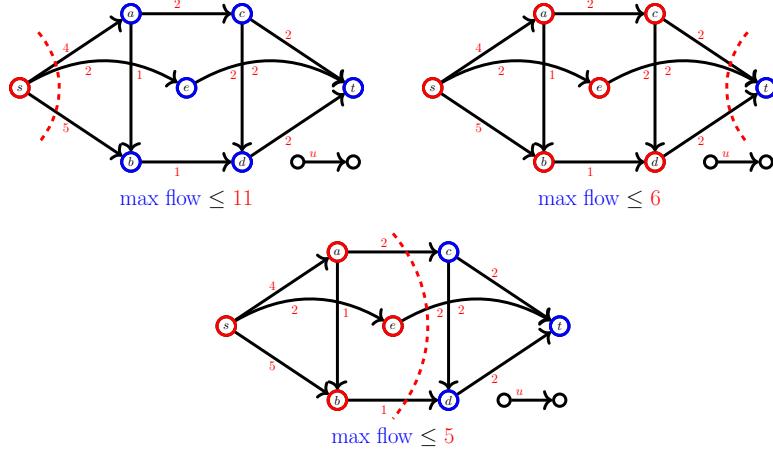
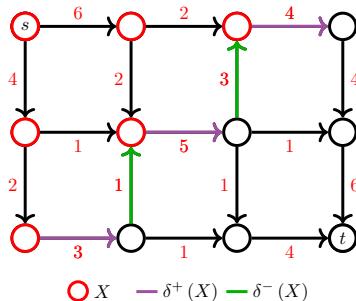


Fig. 5.16.: Bound the value on the maximum flow (No. 759)

Definition 64 $((s, t)\text{-Cut})$. Let (G, u, s, t) be a network. Let $X \subseteq V$, with $s \in X$ and $t \notin X$. Then X determines an (s, t) -cut $\delta(X)$, where $\delta(X)$ contains all arcs $(v, u) \in A(G)$ with $v \in X$ and $u \notin X$ or $v \notin X$ and $u \in X$. Let $\delta^+(X)$ be the set of all out-going arcs from X , i.e., $\delta^+(X) = \{(u, v) \in A(G) \mid u \in X, v \notin X\}$. Then, the *capacity of an (s, t) -cut* $\delta(X)$ is defined as the sum of the capacities of these arcs, i.e.,

$$\text{cap}(X) := \sum_{a \in \delta^+(X)} u(a).$$

Fig. 5.17.: (s, t) -cut (No. 586)

- An (s, t) -cut can be interpreted as disconnecting s from t by destroying some arcs
- The cost for destroying an arc is $u(a)$
- \Rightarrow the cost for disconnecting s from t is the sum over all destroyed arcs

Definition 65. Minimum (s, t) -Cut Problem

Given: Network (G, u, s, t)

Find: An (s, t) -cut of minimum capacity

- We will start by proving weak duality between the minimum (s, t) -cut problem and the maximum flow problem

Lemma 66 (Weak duality of max-flow and min-cut problem). *Let (G, u, s, t) be a network. For any (s, t) -flow f and any (s, t) -cut $X \subseteq V(G)$ it holds:*

$$1. \quad \text{val}(f) = \sum_{a \in \delta^+(X)} f(a) - \sum_{a \in \delta^-(X)} f(a)$$

$$2. \quad \text{val}(f) \leq \text{cap}(X) \quad (\text{weak duality})$$

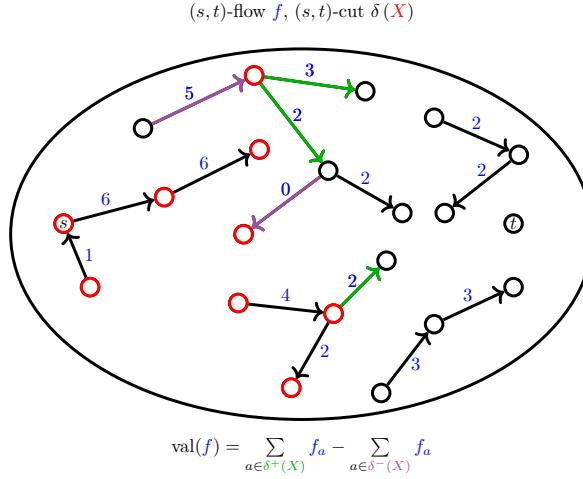


Fig. 5.18.: Weak duality (No. 587)

Proof. Definition of capacity

- *Property 1:* Let f be an (s, t) -flow and let X determine an (s, t) -cut
- Then,

$$\begin{aligned} \text{val}(f) &:= \sum_{a \in \delta^+(s)} f(a) - \sum_{a \in \delta^-(s)} f(a) \\ &\stackrel{(a)}{=} \sum_{a \in \delta^+(s)} f(a) - \sum_{a \in \delta^-(s)} f(a) + \sum_{v \in X \setminus \{s\}} \left(\sum_{a \in \delta^+(v)} f(a) - \sum_{a \in \delta^-(v)} f(a) \right) \\ &\stackrel{(b)}{=} \sum_{a \in \delta^+(X)} f(a) - \sum_{a \in \delta^-(X)} f(a) \end{aligned}$$

- (a) flow conservation holds at every vertex besides s and t
- (b) if $a = (u, v)$ with $u \in X$ and $v \in X$, then $a \in \delta^+(u)$ and $a \in \delta^-(v) \Rightarrow$ cancels out

- *Property 2:* f obeys the capacity constraint

- \Rightarrow

$$\begin{aligned} \text{val}(f) &= \sum_{a \in \delta^+(X)} f(a) - \sum_{a \in \delta^-(X)} f(a) \\ &\leq \sum_{a \in \delta^+(X)} u(a) - \sum_{a \in \delta^-(X)} 0 \\ &= \text{cap}(X) \end{aligned}$$

□

- However, we can even prove a stronger relation between the two problems

Theorem 67 (Max-Flow-Min-Cut Theorem, Ford-Fulkerson 1956, Elias, Feinstein, Shannon 1956). *In a network (G, u, s, t) the maximum value of an (s, t) -flow equals the minimum capacity of an (s, t) -cut.*

Proof. Definition of a minimum cut according to the Ford-Fulkerson Algorithm

- Let f be a maximum (s, t) -flow and X induces a minimum (s, t) -cut $\delta(X)$
- Due to Lemma 66, we have $\text{val}(f) \leq \text{cap}(X)$
- We will now prove: $\text{val}(f) \geq \text{cap}(X)$ by defining an (s, t) -cut
- Set $R = \{v \in V(G) \mid \text{there is directed } (s, v)\text{-path in } G_f\}$

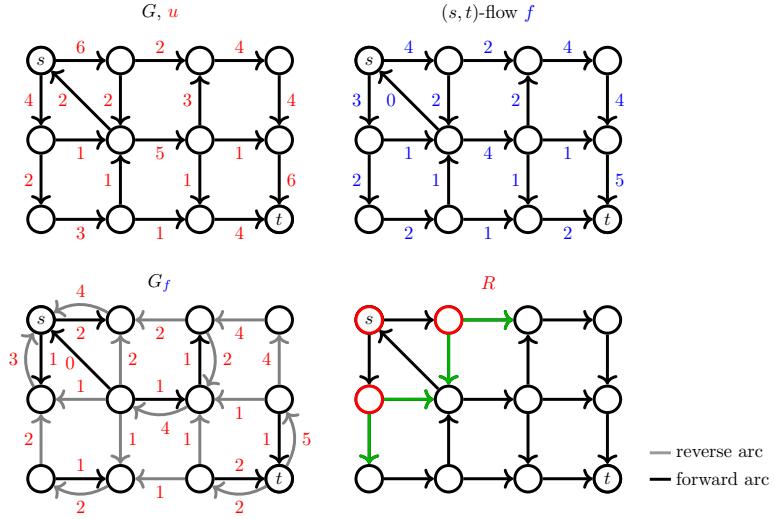


Fig. 5.19.: Construction minimum cut (No. 588)

- Since f a maximum (s, t) -flow, there exists no f -augmenting path in G_f (Lemma 62)
- $\Rightarrow s \in R$ and $t \notin R$
- For an arc $a = (x, y) \in \delta^+(R)$ in G , $f(a) = u(a)$ (otherwise $y \in R$)
- For arc $a = (u, v) \in \delta^-(R)$ in G , $f(a) = 0$ (otherwise $u \in R$)
- \Rightarrow

$$\begin{aligned} \text{val}(f) &\stackrel{(a)}{=} \sum_{a \in \delta^+(R)} f(a) - \sum_{a \in \delta^-(R)} f(a) \\ &= \sum_{a \in \delta^+(R)} u(a) - \sum_{a \in \delta^-(R)} 0 \\ &= \text{cap}(R) \stackrel{(b)}{\geq} \text{cap}(X) \end{aligned}$$

- (a) Lemma 66
- (b) $\delta(X)$ is a minimum cut

- $\Rightarrow \text{val}(f) = \text{cap}(X)$

□

- From this Theorem and the proof, we can deduce several important results

Theorem 68 (Optimality criterion for maximum flows). *Let (G, u, s, t) be a flow network. An (s, t) -flow is maximum if and only if there is no f -augmenting path.*

- This leads to the correctness of the Ford-Fulkersons Algorithm

Theorem 69 (Optimality of Ford-Fulkerson Algorithm). *The Ford-Fulkersons Algorithm computes a maximum (s, t) -flow.*

5.4. The Edmonds-Karp Algorithm and Dinic Algorithm with blocking flows

- Problem of Ford-Fulkerson Algorithms: Run-time depending on $u_{\max} = \max_{a \in A} \{u(a)\}$
- The Edmonds-Karp algorithm specifies the choice of f -augmenting paths in the Ford-Fulkerson Algorithm:
 - Choose shortest f -augmenting path w.r.t. the number of arcs

Algorithm 5.2 Edmonds-Karp Algorithm

Input: Network (G, u, s, t)

Output: Maximum (s, t) -flow f

Initialization: Set $f(a) := 0$ for all $a \in A(G)$

Method: **While** there is an f -augmenting (s, t) -path **do**

- Determine a **shortest** f -augmenting path p
- Set $\gamma := \min\{u_f(a) \mid a \in A(p)\}$
- Augment f along p by γ

Return f

-
- Easy to compute by BFS in the residual graph G_f
 - By this they can prove a run-time only depending on the number of arcs and vertices
 - We start by analyzing some properties on the sequence of augmentations if we always chose a shortest f -augmenting path

Lemma 70 (Monotonicity Lemma). *Let f_1, f_2, \dots be a sequence of (s, t) -flows in G , where f_{k+1} is created from f_k by augmenting along path p_k . The path p_k is a shortest path in the residual graph G_{f_k} of f_k w.r.t. the number of arcs. Then the following holds:*

1. The length of the paths is non-decreasing, i.e., $|A(p_k)| \leq |A(p_{k+1})|$ (weak monotony)
2. $|A(p_k)| + 2 \leq |A(p_\ell)|$ for all $k < \ell$, s.t. $p_k \cup p_\ell$ contains a pair of opposing arcs a, \bar{a} (strong monotony)

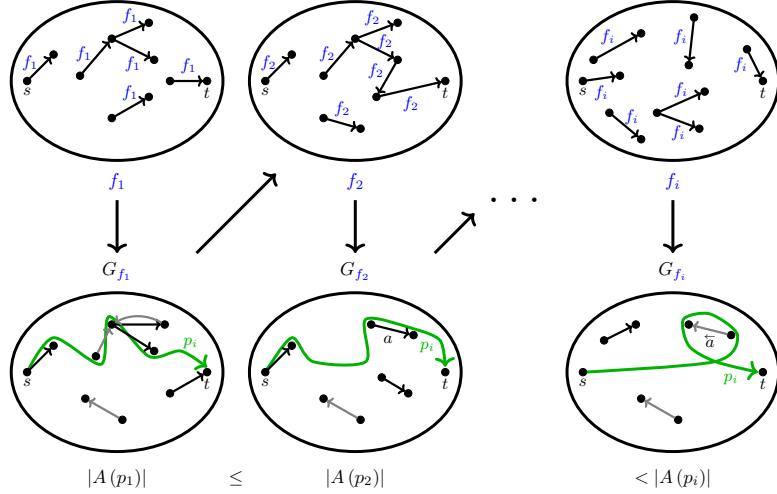
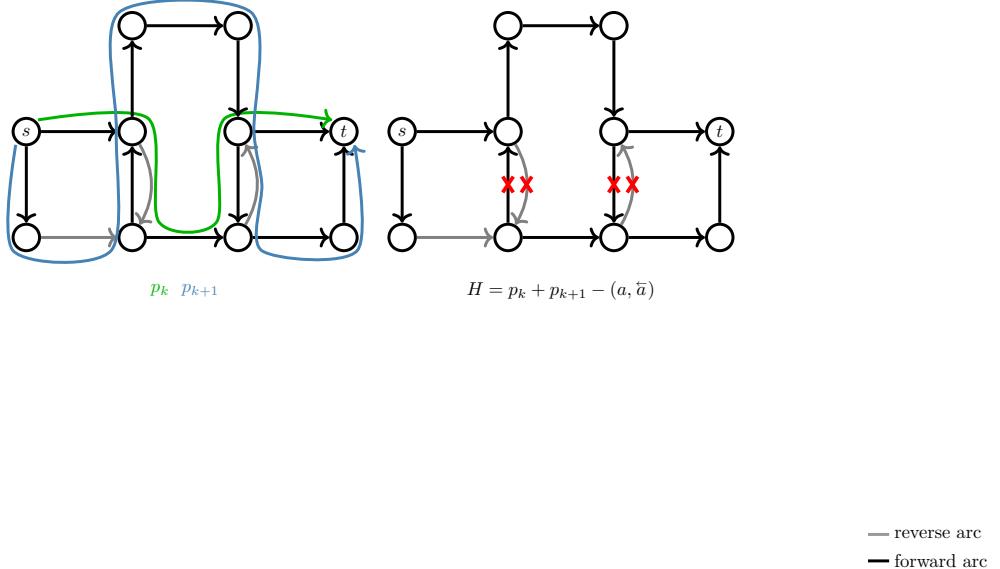


Fig. 5.20.: Path length increases (No. 589)

Proof. Construction of f_k via shortest paths

- *Property 1:* Consider f_k and f_{k+1}
- Let H be an auxiliary graph defined by $H := p_k + p_{k+1}$ and delete pairs of reverse arcs

Fig. 5.21.: Auxiliary Graph H (No. 590)

- *Claim 1:* $\forall (u, v) \in A(H) \exists (u, v) \in A(G_{f_k})$
Proof:

- Assume: $\exists a = (u, v) \in A(H)$ and $(u, v) \notin A(G_{f_k})$

- $\Rightarrow a \in p_{k+1}$, since $p_k \subseteq A(G_{f_k})$
 - a is added to $G_{f_{k+1}}$ due to augmenting along p_k
 - $\Rightarrow a$ is reverse arc of $a' \in A(p_k)$
 - \Rightarrow by construction, a and a' are deleted from H
 - \Rightarrow contradiction $\square C1$
 - *Claim 2:* H contains two arc-disjoint (s, t) -paths \bar{p}_1, \bar{p}_2 .
- Proof:*
- $H' := H + (t, s) + (t, s)$
 - Due to the construction, the in-degree equals the out-degree of every vertex in H'
 - $\Rightarrow H'$ is an arc-disjoint union of (directed) elementary cycles
 - \Rightarrow there are exactly two cycles C_1 and C_2 that contain the arcs (t, s) and (t, s)

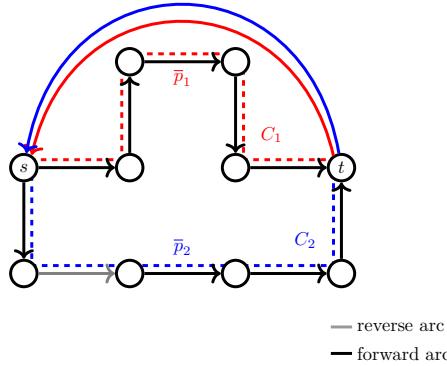


Fig. 5.22.: $H' := H + (t, s) + (t, s)$ (No. 591)

- Deleting these arcs, we obtain two disjoint (s, t) -paths in H $\square C2$
- Due to Claim 1 and 2: \bar{p}_1, \bar{p}_2 are f -augmenting
- Since p_k is a shortest f_k -augmenting path \Rightarrow

$$|A(p_k)| \leq |A(\bar{p}_1)| \text{ and } |A(p_k)| \leq |A(\bar{p}_2)|$$

- \Rightarrow

$$\begin{aligned} 2|A(p_k)| &\leq |A(\bar{p}_1)| + |A(\bar{p}_2)| \leq |A(H)| \\ &\leq |A(p_k)| + |A(p_{k+1})| \end{aligned}$$

- $\Rightarrow |A(p_k)| \leq |A(p_{k+1})|$
- *Property 2:* By Property 1 it suffices to show the statement for k, ℓ s.t. $p_k \cup p_\ell$ contain a reverse arc, but $\forall i, k < i < \ell: p_k \cup p_i$ contains no pair of reverse arcs
- Define $H = p_k + p_\ell$ and delete pairs of reverse arcs
- As before,
 1. $\forall (u, v) \in A(H) \exists (u, v) \in A(G_{f_k})$ (exercise)
 2. H contains two arc-disjoint (s, t) -paths \bar{p}_1, \bar{p}_2
- $\Rightarrow \bar{p}_1$ and \bar{p}_2 are f_k -augmenting with $|A(p_k)| \leq |A(\bar{p}_i)|, i = 1, 2$

- \Rightarrow

$$\begin{aligned} 2|A(p_k)| &\leq |A(\bar{p}_1)| + |A(\bar{p}_2)| \leq |A(H)| \\ &\stackrel{(a)}{\leq} |A(p_k)| + |A(p_\ell)| - 2 \end{aligned}$$

(a) since at least two arcs are deleted

- $\Rightarrow |A(p_k)| + 2 \leq |A(p_\ell)|$

□

- In the following, we want to limit the number of times an arc is
 1. part of an f -augmenting path and
 2. limits the potential increment due to its residual capacity
- Since these arcs play a crucial role, we define them more formally

Definition 71 (Bottleneck Arc). Let p be an f -augmenting path and $\gamma = \min_{a \in p} \{u_f(a)\}$. An arc $a \in p$ is a *bottleneck arc* if $\gamma = u_f(a)$.

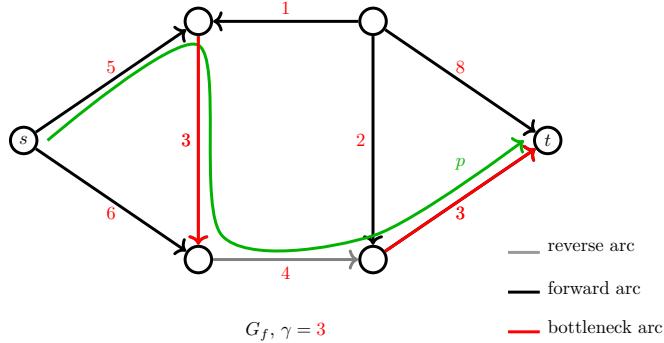


Fig. 5.23.: Bottleneck arc (No. 592)

- In other words: If the residual capacity $u_f(a)$ of all bottleneck arcs in a path are increased by 1, we can augment along p one more unit of flow
- We can bound the number of f -augmenting paths and thus the run-time of Edmonds-Karp Algorithm if the upper capacities are integer
- Recall that Edmonds-Karp Algorithm chooses a shortest path according to the number of arcs as f -augmenting path in each iteration of the Ford-Fulkerson algorithm

Theorem 72 (Edmonds-Karp 1972). Let (G, u, s, t) be a network. The Edmonds-Karp Algorithm stops after at most $\frac{mn}{2}$ augmentations, where $m = |A(G)|$ and $n = |V(G)|$. Thus, the algorithm determines a maximum flow in $\mathcal{O}(m^2n)$ time.

Proof. Bound on the number of iterations

- *Claim:* An arc $a \in A(G)$ can occur at most $\frac{n}{4}$ times as a bottleneck arc
- Proof:*

- Consider arc a and let \bar{a} be its reversed arc
- Let p_1, \dots, p_r be the sequence of f -augmenting paths in the algorithm
- Let p_{i_1}, p_{i_2}, \dots be the sequence of the paths in the algorithm which contain a as bottleneck arc
- Consider two successive paths $p_{i_j}, p_{i_{j+1}}$ of the sequence
- Since a is a bottleneck arc, \bar{a} is in $G_{f_{i_{j+1}}}$ but not a
- Since a is in $p_{i_{j+1}}$, there is an f -augmenting path p_k with $i_j < k < i_{j+1}$ which contains \bar{a}
- Due to the Monotony Lemma:

$$|A(p_{i_j})| + 4 \leq |A(p_k)| + 2 \leq |A(p_{i_{j+1}})| \stackrel{(a)}{\leq} n - 1$$

- (a) all paths are elementary
- \Rightarrow at most $\frac{n}{4}$ paths in the sequence contain a $\square C$
- \Rightarrow maximum $|A(G)| \cdot \frac{n}{4} = 2m \cdot \frac{n}{4} = m \cdot \frac{n}{2}$ augmenting paths
- Search of such a path is done with a BFS in G_f in $\mathcal{O}(m)$ time
- $\Rightarrow \mathcal{O}(m^2n)$ total run-time

\square

Dinic Algorithm

- Dinic independently proposed in 1970 an even better version
- Instead of choosing one single (s, t) -path to augment, he considered a set of (s, t) -paths simultaneously to augment along them

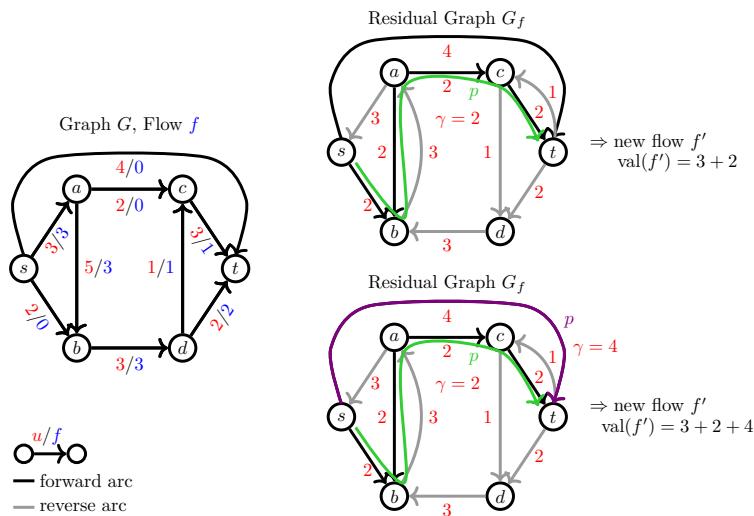


Fig. 5.24.: Augment along several paths (No. 805)

- \Rightarrow we augment along an (s, t) -flow within the residual graph

Definition 73 (f -augmenting (s, t) -flow f_R). Let (G, u, s, t) be a flow network and f an (s, t) -flow. An f -augmenting (s, t) -flow f_R is an (s, t) -flow in the residual graph G_f . Augmenting the flow f along f_R means to construct a flow f' in G via

$$f'(a) := f(a) + f_R(a) - f_R(\overleftarrow{a})$$

for all $a \in A(G)$, where \overleftarrow{a} is the reverse arc of a in G_f , and with $f_R(a) := 0$ if $u_f(a) = 0$, $a \in A(\overleftrightarrow{G})$.

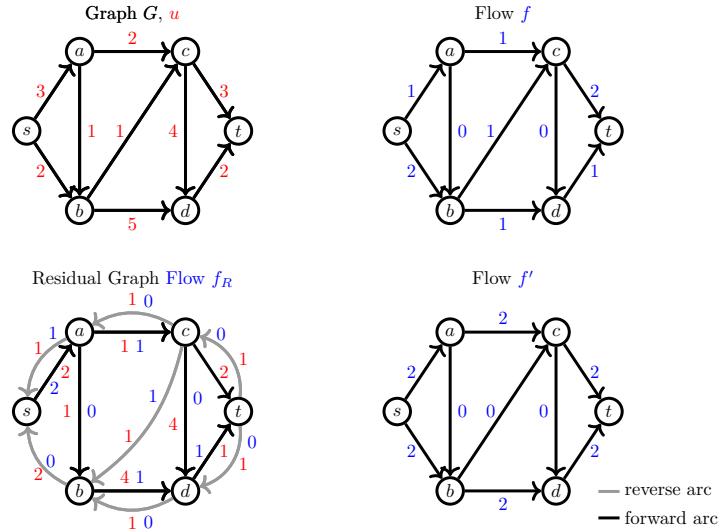


Fig. 5.25.: f -augmenting (s, t) -flow (No. 760)

- As for augmenting (s, t) -paths, we need to prove that f' is a feasible (s, t) -flow

Lemma 74. Let f be a feasible (s, t) -flow in G and f_R be an f -augmenting (s, t) -flow in G_f . Then, augmenting the flow f along f_R leads to a feasible (s, t) -flow f' in G with $\text{val}(f') = \text{val}(f) + \text{val}(f_R)$.

Proof. Exercise □

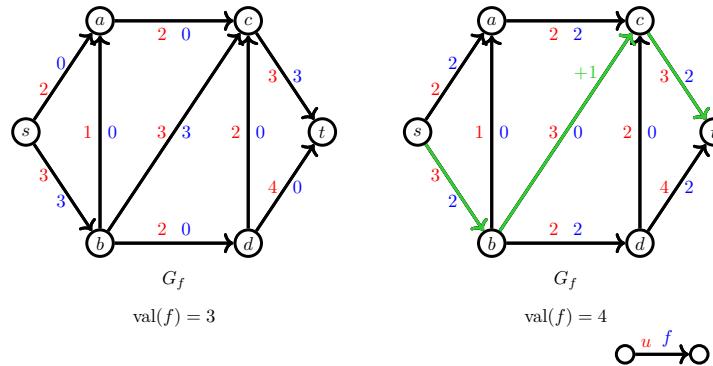


Fig. 5.26.: Which flow to choose? (No. 806)

- In order to get the most out of a residual graph without solving a maximum flow problem, Dinic used so called blocking flows

Definition 75 (Blocking (s, t) -flow). Given a network (G, u, s, t) , a *blocking* (s, t) -flow is an (s, t) -flow f , which saturates an arc on each (s, t) -path in G . An arc is *saturated*, if $f(a) = u(a)$.

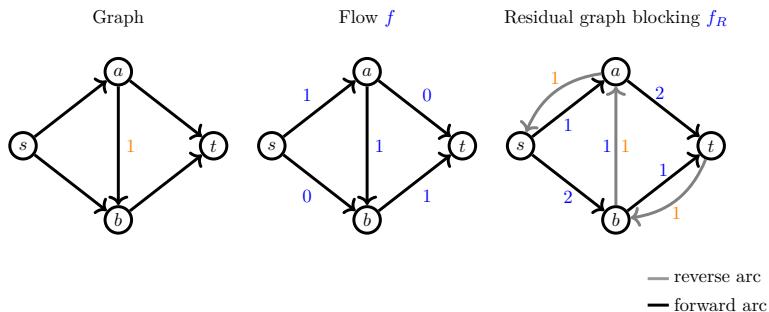


Fig. 5.27.: Blocking (s, t) -flow (No. 761)

- Blocking flows are generally not maximum flows
- Using blocking flows as f -augmenting (s, t) -flows in a residual network increases potentially the extra amount we send after one augmentation
- However, it still may lead to many unnecessary iterations
- Dinic's second idea was therefore, to use blocking flow in a subgraph of a residual graph
- This subgraph contains all shortest (s, t) -paths w.r.t. the number of arcs of the residual graph and is called level graph

Definition 76 (Level graph). Given a network (G, u, s, t) and an (s, t) -flow f . The level graph G_f^L is a subgraph of G_f , s.t. the following holds: Let p be a shortest (s, t) -path in G_f , then $p \in G_f^L$. Let p be an (s, t) -path in G_f^L , then p is a shortest (s, t) -path in G_f according to the number of arcs.

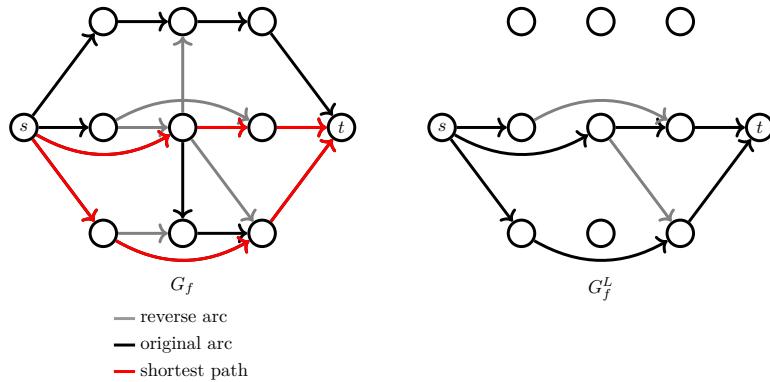


Fig. 5.28.: left: two shortest paths in G_f , right: level graph (No. 593)

- Level i is the set of all vertices with distance i to vertex s

Lemma 77. Given a network (G, u, s, t) and an (s, t) -flow f . Then the level graph G_f^L exists and can be computed in polynomial time.

Proof. Construction algorithm

- Consider the following procedure to compute a subgraph of G_f :
 - Perform a BFS in G_f until t is reached
 - Add all arcs that connect consecutive levels
 - Delete all vertices and arcs, that are not connected to t

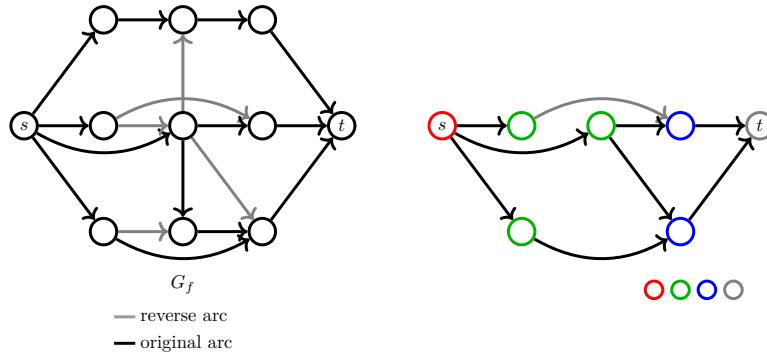


Fig. 5.29.: Constructing a level graph (No. 594)

- Let G_f^L be the constructed graph
- Exercise: prove that G_f^L is a level graph

□

- Using blocking flows in the level graph to augment a given flow leads to the following algorithm:

Algorithm 5.3 Dinic Algorithm

Input: Network (G, u, s, t)

Output: Maximum (s, t) -flow f

Initialization: Set $f(a) := 0$ for all $a \in A(G)$

Method: **While** there is an f -augmenting (s, t) -path **do**

- Construct level graph G_f^L
- Construct a blocking integer flow f_R in G_f^L with $u_f^L(a) = u_f(a)$ for all $a \in A(G_f^L)$
- Augment f along f_R

Return f

-
- The number of iterations in the Dinic Algorithm decreases drastically

Lemma 78. *In each iteration, the length of a shortest (s, t) -path in the corresponding level graph increases by at least 1.*

Proof. Definition of level graph and blocking flow

- Let f_{i+1} be the flow constructed in the i th iteration by augmenting f_i along f_R
- Then, f_R is a blocking flow in $G_{f_i}^L$
- Let ℓ_{f_i} ($\ell_{f_{i+1}}$) be the length of a shortest path in $G_{f_i}^L$ ($G_{f_{i+1}}^L$ respectively)
- Let p' be a shortest path in $G_{f_{i+1}}^L$, i.e., $\ell_{f_{i+1}} = |A(p')|$
- Case 1: $p' \in G_{f_i}^L$
 - f_R is a blocking flow
 - $\Rightarrow \exists a \in A(p')$ with $u_{f_{i+1}}(a) = 0$
 - Contradiction to $p' \in G_{f_{i+1}}^L$
- Case 2: $p' \in G_{f_i}$ and $p' \notin G_{f_i}^L$
 - $\Rightarrow |A(p')| \geq \ell_{f_i} + 1$ due to the definition of a level graph
- Case 3: $p' \notin G_{f_i}$
 - Consider the multi-graph $H(V, A_H)$ that consists of
 - $f_R(a)$ parallel arcs, $a \in G_f^L$ (since $f_R(a)$ is integral, that is possible)
 - all arcs of p'
 - minus pairs of reverse arcs
 - *Claim a:* If $a \in A_H \Rightarrow a \in A(G_{f_i})$ (Exercise)
 - Let $k = \text{val}(f_R)$
 - *Claim b:* H contains $k + 1$ arc-disjoint (s, t) -paths p_1, \dots, p_{k+1} (Exercise)
 - Since p_1, \dots, p_{k+1} are augmenting paths in G_{f_i} , $|A(p_r)| \geq \ell_{f_i}$, $r = 1, \dots, k + 1$
 - Thus,

$$(k+1)\ell_{f_i} \leq \sum_{r=1}^{k+1} |A(p_r)| \stackrel{(a)}{\leq} |A(H)| - 2 = k \cdot \ell_{f_i} + |A(p')| - 2$$

- (a) at least one pair of reverse arcs was deleted, since $p' \notin G_{f_i}$
- $\Rightarrow \ell_{f_i} + 2 \leq |A(p')|$
- In summary of all cases: $\ell_{f_i} + 1 \leq |A(p')| = \ell_{f_{i+1}}$

□

- Hence, the number of iterations is bounded by n
- Dinic 1970: Construction of the level graph and a blocking flow in $O(n \cdot m)$
- Karzanov 1974, Malhotra, Kumar & Maheshwari 1978: Construction of the level graph and a blocking flow in $O(n^2) \Rightarrow$ Total run-time $O(n^3)$



6. Minimum Cost Flow

6.1. Problem Formulation

- Max-flow problem:
 - Models transport through network
 - Costs are neglected
 - But: integration of costs necessary for applications

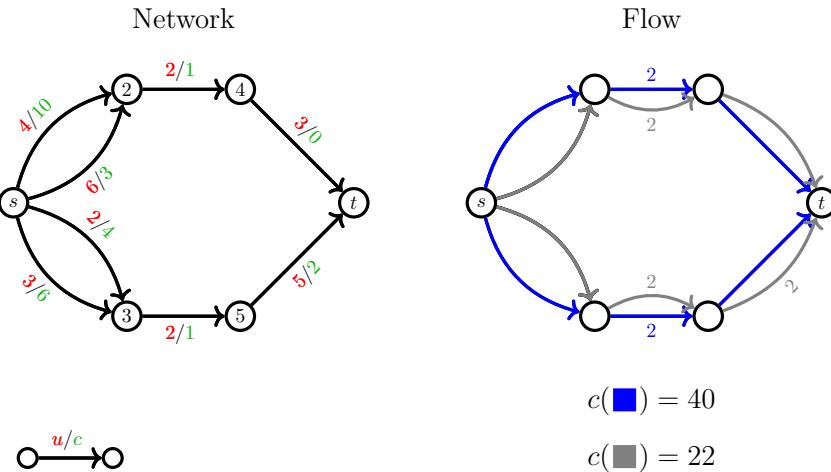


Fig. 6.1.: Different flows cause different costs (No. 43)

Definition 79 (Network). A *(Flow-)network with arc costs* (G, u, c) consists of a

- directed graph $G = (V, A)$
- upper capacities $u : A \rightarrow \mathbb{N}$
- arc cost $c : A \rightarrow \mathbb{Z}$

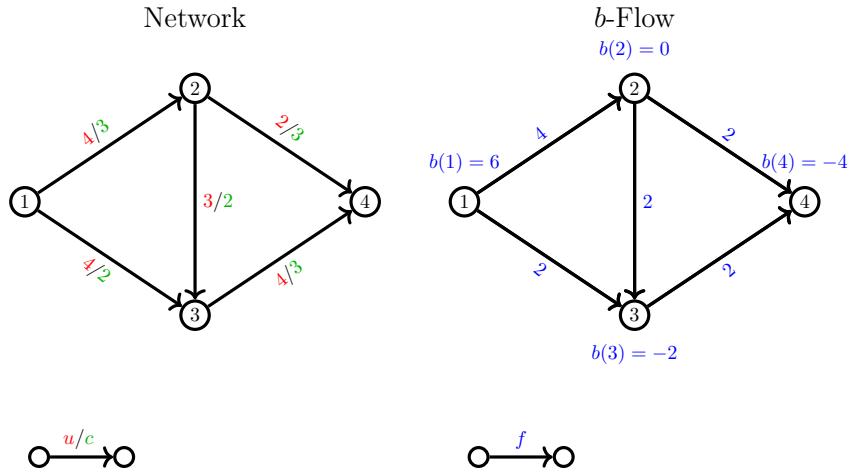
- Since in many applications several warehouses and stores exist, we model them with so-called b -flows

Definition 80 (b -flow). Given a network (G, u, c) and balances $b : V \rightarrow \mathbb{Z}$ with $\sum_{v \in V} b(v) = 0$, a b -flow in (G, u, c) is a mapping $f : A \rightarrow \mathbb{R}$ satisfying

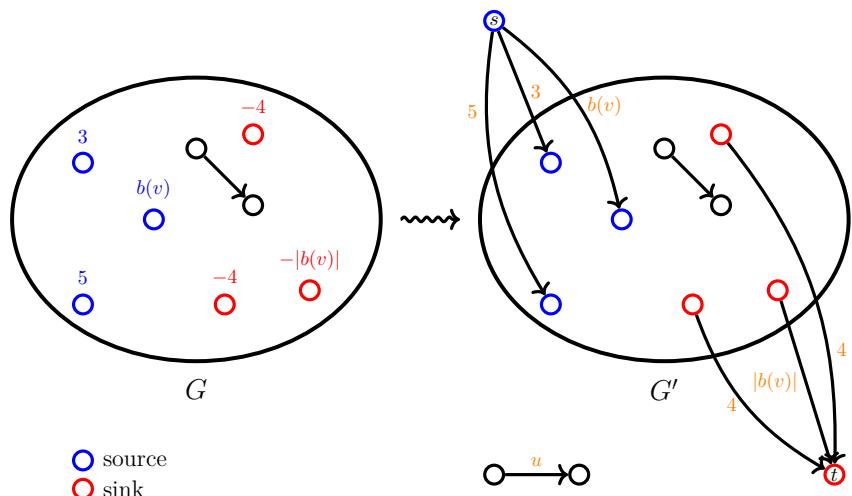
- *Capacity constraints*
 $0 \leq f(a) \leq u(a) \quad \forall a \in A$
- *Flow-balance constraints* [out-flow - in-flow = balance]

$$\sum_{a \in \delta^+(v)} f(a) - \sum_{a \in \delta^-(v)} f(a) = b(v) \quad \forall v \in V$$

with $\delta^+(v)$ denoting the outgoing arcs and $\delta^-(v)$ the incoming arcs of v

Fig. 6.2.: A simple network with a b -flow (No. 44)

- If $b(v) > 0$, v is called a *source* and $b(v)$ its *supply*
- If $b(v) < 0$, v is called a *sink* and $|b(v)|$ its *demand*
- If $b(v) = 0$, v is called a *transshipment vertex*
- (s, t) -flow: $b(v) = 0 \forall v \in V \setminus \{s, t\}$, $b(s) = -b(t)$, $b(s) \geq 0$
- Can we find a b -flow in a network (G, u, c) in polynomial time?
 - Transform G into G'
 - Add vertex s and t to G
 - Add arcs (s, v) with $u(s, v) = b(v)$ to G $\forall v \in V$ with $b(v) > 0$
 - Add arcs (v, t) with $u(v, t) = -b(v)$ to G $\forall v \in V$ with $b(v) < 0$
 - Compute a maximum (s, t) -flow f in (G', u) (e.g., using Ford-Fulkerson)
 - If f transports $0.5 \sum_{v \in V} |b(v)|$, f restricted to G is a b -flow in (G, u, c)

Fig. 6.3.: Finding a b -flow is easy (No. 45)

Definition 81. Minimum Cost Flow (MCF) Problem

Given: • flow network (G, u, c)
 • balances $b : V \rightarrow \mathbb{Z}$, $\sum_{v \in V} b(v) = 0$

Find: a b -flow f with minimum cost $c(f)$

$$c(f) := \sum_{a \in A} c(a) \cdot f(a)$$

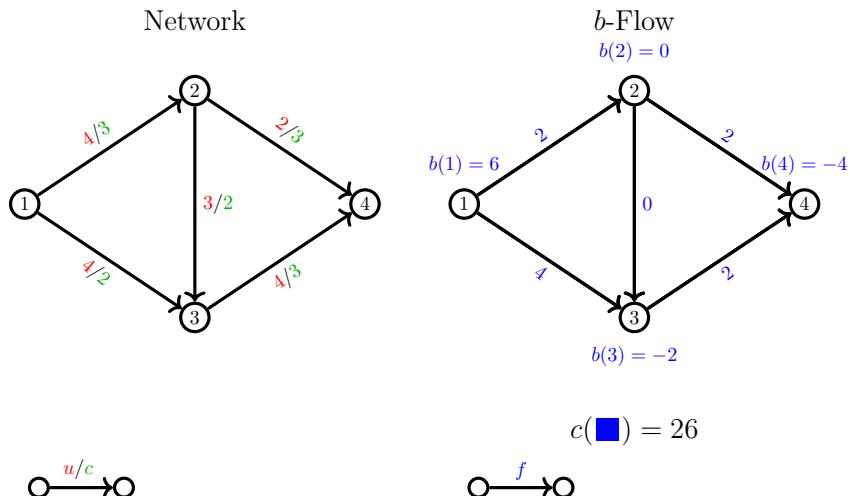


Fig. 6.4.: MCF problem (No. 46)

- The definition demands that the supply and demand are balanced

- In practice, this is not necessarily the case

- What if $\sum_{v \in V} b(v) \neq 0$?
 - If $\sum_{v \in V} b(v) < 0$:
 - Add dummy source s' with $b(s') = -\sum_{v \in V} b(v)$
 - Add arcs (s', v) for all $v \in V$ with $b(v) < 0$
 - Set $u(s', v) = b(s')$ and $c(s', v) = M > (n - 1) \cdot \max_{a \in A(G)} c(a)$

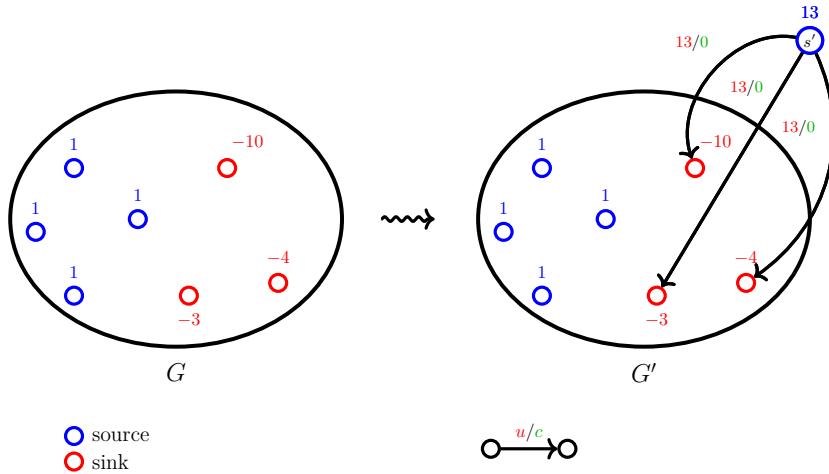
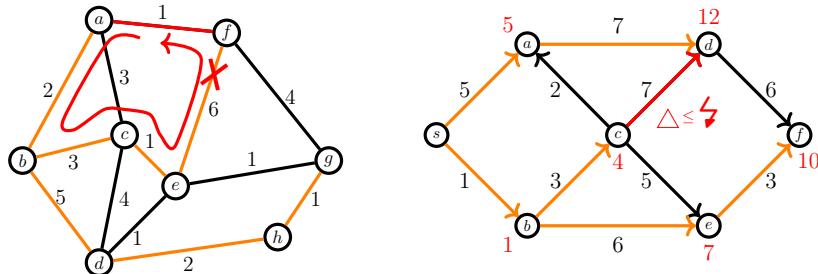


Fig. 6.5.: Unbalanced b -values can be easily transformed to balanced ones (No. 47)

- If $\sum_{v \in V} b(v) > 0$: add dummy target similar to the problem above
- Question: How can we compute an optimal minimum cost flow?

6.2. An Optimality Criterion

- Optimality Criterion: some criterion to determine whether a given solution is optimal



Minimum spanning tree?

Shortest path tree?

Fig. 6.6.: Optimality Criteria (No. 766)

- Approach to determine such a criterion:
 - Compare some solution with an optimal solution
 - What are the differences in the structure of the solutions?
 - Can we quantify the cost difference induced by the difference in the structure?
- Comparison of two b -flows: f_1 and f_2

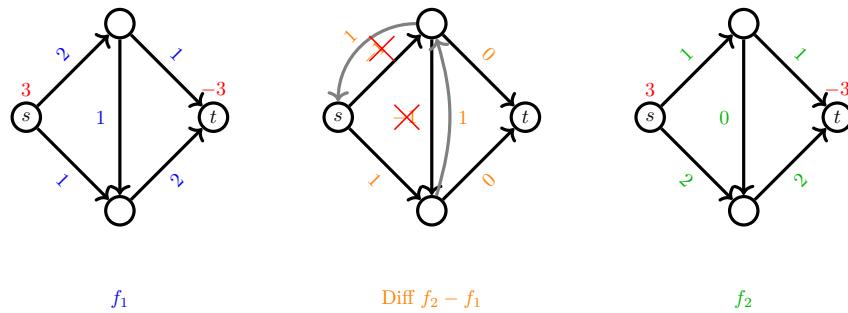


Fig. 6.7.: Difference of two flows (No. 80)

- Some edges have positive flow, some negative
- \Rightarrow definition residual network (to avoid negative flow)

Definition 82 (reverse graph, reverse arc). For a digraph G we define the *reverse graph* $\overset{\leftrightarrow}{G} := (V(G), A(G) \cup \{\overset{\leftarrow}{a} \mid a \in A(G)\})$, where for $a = (v, w) \in A(G)$ we define $\overset{\leftarrow}{a}$ to be a new arc from w to v . We call $\overset{\leftarrow}{a}$ the *reverse arc* of a and vice versa.

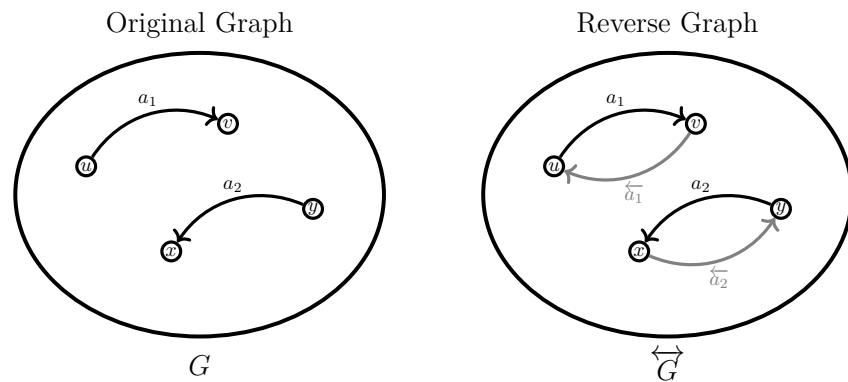


Fig. 6.8.: Reverse Graph (No. 53)

- Note: if $a = (v, w)$, $a' = (w, v) \in A(G)$, then $\overset{\leftarrow}{a}$ and a' are two parallel arcs in $\overset{\leftrightarrow}{G}$

Definition 83 (residual capacities, residual graph). Given a digraph G with capacities $u : A(G) \rightarrow \mathbb{R}_+$, and a flow f , we define *residual capacities* $u_f : A(\overset{\leftrightarrow}{G}) \rightarrow \mathbb{R}$ by $u_f(a) := u(a) - f(a)$ and $u_f(\overset{\leftarrow}{a}) := f(a)$ for all $a \in A(G)$. The *residual graph* G_f is the graph $(V(G), \{a \in A(\overset{\leftrightarrow}{G}) \mid u_f(a) > 0\})$.

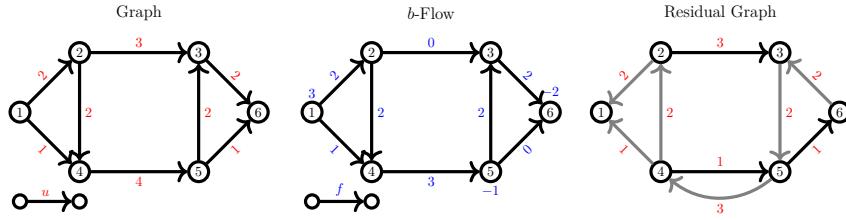


Fig. 6.9.: Residual graph (No. 767)

- The residual graph contains only the necessary arcs
- Consider the difference between two flows, we obtain nice structural results

Lemma 84. Let (G, u, c, b) be a MCF instance. Let f and f' be two b -flows in (G, u, c) . Then $g := f' \Delta f : A(G) \rightarrow \mathbb{R}_+$ defined by

$$g(a) := \max\{0, f'(a) - f(a)\} \text{ and } g(\overleftarrow{a}) := \max\{0, f(a) - f'(a)\}$$

for $a \in A$ has the following properties:

1. For every $v \in V$, we have flow conservation in $\overset{\leftrightarrow}{G}$:

$$\sum_{a \in \delta_{\overset{\leftrightarrow}{G}}^+(v)} g(a) - \sum_{a \in \delta_{\overset{\leftrightarrow}{G}}^-(v)} g(a) = 0$$

2. $g(a) = 0 \ \forall a \notin A(G_f)$

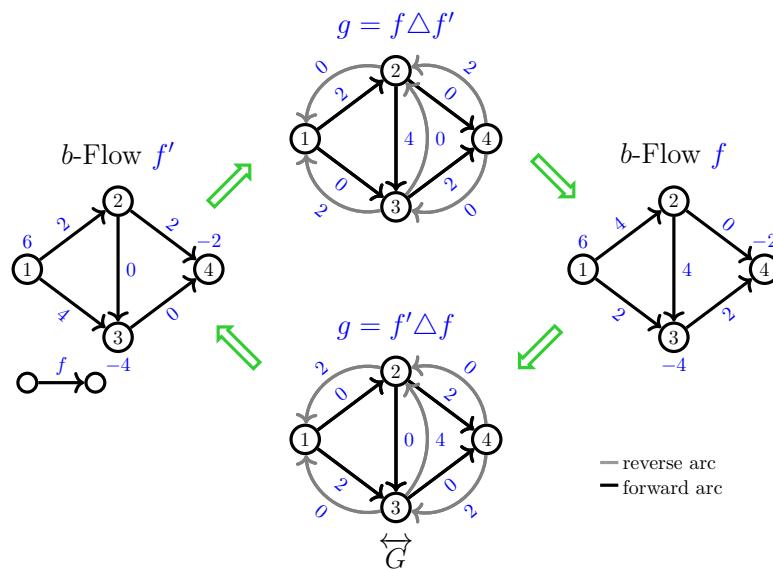


Fig. 6.10.: Difference between two flows (No. 56)

- Interpretation of $f' \Delta f$: in order to get from f to f' , augment along $f' \Delta f$

Proof. Definition of g

- Note, either $g(a) > 0$ or $g(\overleftarrow{a}) > 0 \ \forall a \in A(G)$, but not both

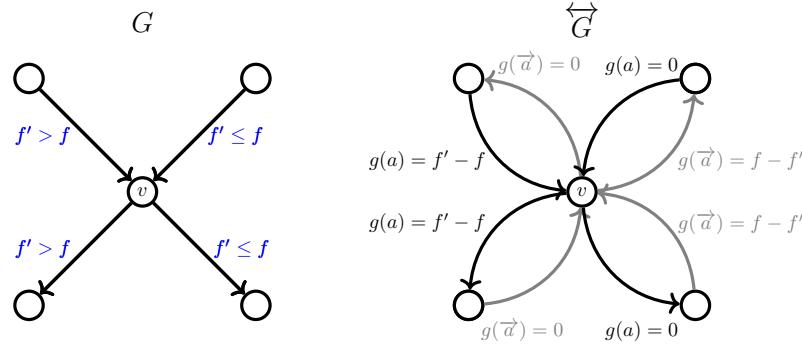


Fig. 6.11.: Flow conservation holds (No. 57)

- Consider $v \in V$ and show $\sum_{a \in \delta_G^+(v)} g(a) - \sum_{a \in \delta_G^-(v)} g(a) = 0$:

$$\begin{aligned}
 0 &= b(v) - b(v) \\
 &= \sum_{a \in \delta_G^+(v)} f'(a) - \sum_{a \in \delta_G^-(v)} f'(a) - \left(\sum_{a \in \delta_G^+(v)} f(a) - \sum_{a \in \delta_G^-(v)} f(a) \right) \\
 &= \sum_{a \in \delta_G^+(v)} (f'(a) - f(a)) - \sum_{a \in \delta_G^-(v)} (f'(a) - f(a)) \\
 &= \sum_{\substack{a \in \delta_G^+(v) \\ f'(a) > f(a)}} (f'(a) - f(a)) + \sum_{\substack{a \in \delta_G^+(v) \\ f'(a) \leq f(a)}} (f'(a) - f(a)) \\
 &\quad - \sum_{\substack{a \in \delta_G^-(v) \\ f'(a) > f(a)}} (f'(a) - f(a)) - \sum_{\substack{a \in \delta_G^-(v) \\ f'(a) \leq f(a)}} (f'(a) - f(a)) \\
 &= \sum_{\substack{a \in \delta_{\overleftarrow{G}}^+(v) \cap A(G) \\ f'(a) > f(a)}} (f'(a) - f(a)) + \sum_{\substack{a \in \delta_{\overleftarrow{G}}^+(v) \cap A(G) \\ f'(a) \leq f(a)}} (f'(a) - f(a)) \\
 &\quad - \sum_{\substack{a \in \delta_{\overleftarrow{G}}^-(v) \cap A(G) \\ f'(a) > f(a)}} (f'(a) - f(a)) - \sum_{\substack{a \in \delta_{\overleftarrow{G}}^-(v) \cap A(G) \\ f'(a) \leq f(a)}} (f'(a) - f(a)) \\
 &= \sum_{\substack{a \in \delta_{\overleftarrow{G}}^+(v) \cap A(G) \\ f'(a) > f(a)}} \left(g(a) + \underbrace{g(\overleftarrow{a})}_{=0} \right) + \sum_{\substack{a \in \delta_{\overleftarrow{G}}^+(v) \cap A(G) \\ f'(a) \leq f(a)}} \left(\underbrace{g(a)}_{=0} - \underbrace{g(\overleftarrow{a})}_{=f(a)-f'(a)} \right) \\
 &\quad - \sum_{\substack{a \in \delta_{\overleftarrow{G}}^-(v) \cap A(G) \\ f'(a) > f(a)}} \left(g(a) + \underbrace{g(\overleftarrow{a})}_{=0} \right) - \sum_{\substack{a \in \delta_{\overleftarrow{G}}^-(v) \cap A(G) \\ f'(a) \leq f(a)}} \left(\underbrace{g(a)}_{=0} - \underbrace{g(\overleftarrow{a})}_{=f(a)-f'(a)} \right) \\
 &= \sum_{a \in \delta_{\overleftarrow{G}}^+(v)} g(a) - \sum_{a \in \delta_{\overleftarrow{G}}^-(v)} g(a)
 \end{aligned}$$

- \Rightarrow flow conservation holds

- Show $g(a) = 0 \ \forall a \notin A(G_f)$

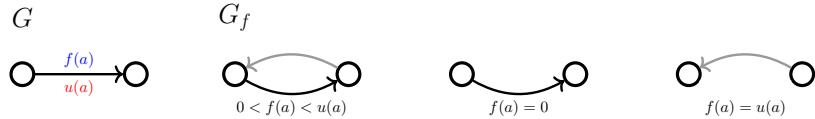


Fig. 6.12.: Arcs in the residual graph (No. 768)

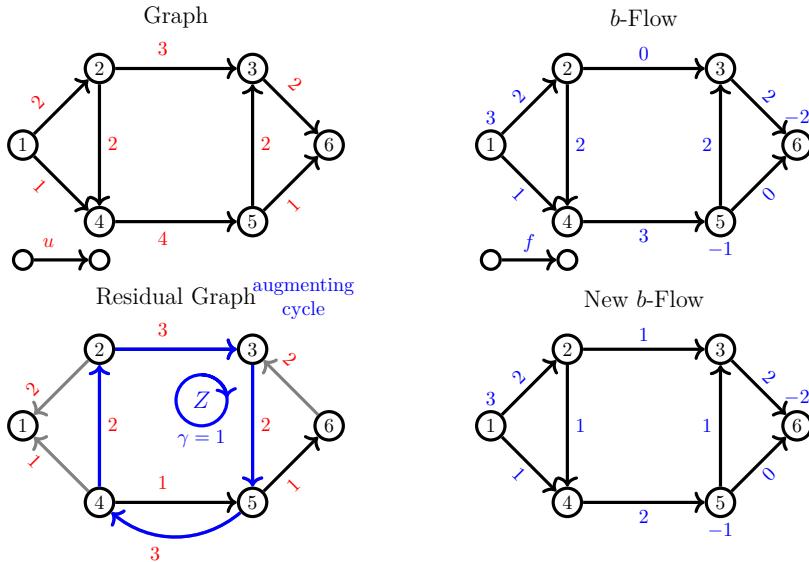
- Let $a \in A(G) \setminus A(G_f)$
- Case 1: $a \in A(G)$
 - $\Rightarrow f(a) = u(a)$
 - $\Rightarrow f'(a) \leq u(a) = f(a) \Rightarrow g(a) = 0$
- Case 2: $\exists a' \in A(G)$, s.t. $\overset{\leftarrow}{a'} = a$
 - $\Rightarrow f(a') = 0$
 - $\Rightarrow f'(a') \geq 0 = f(a') \Rightarrow g(a) = 0$

□

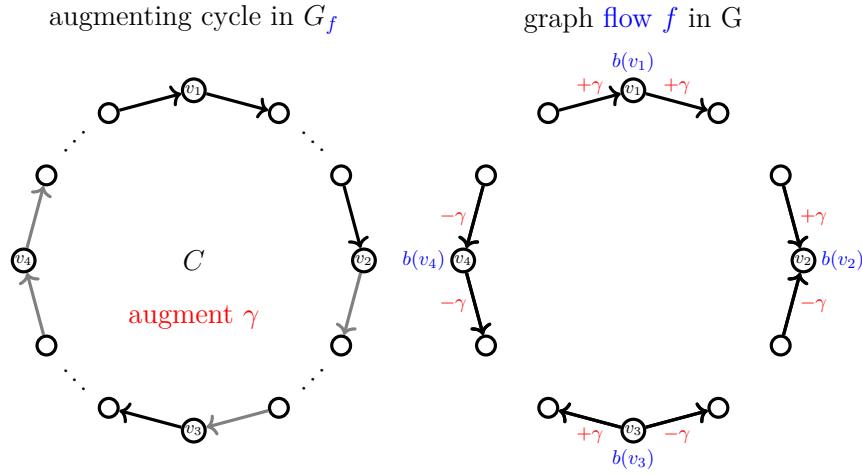
- Using this result, we can model how to get from one flow to another

Definition 85 (f -augmenting cycle). Given a digraph G with capacities u and a b -flow f , then an f -augmenting cycle is a directed cycle in G_f . Given a b -flow f and an f -augmenting cycle C in G_f , to *augment* f along C by $\gamma \leq \min_{a \in A(C)} u_f(a)$ means to define a new b -flow f' by:

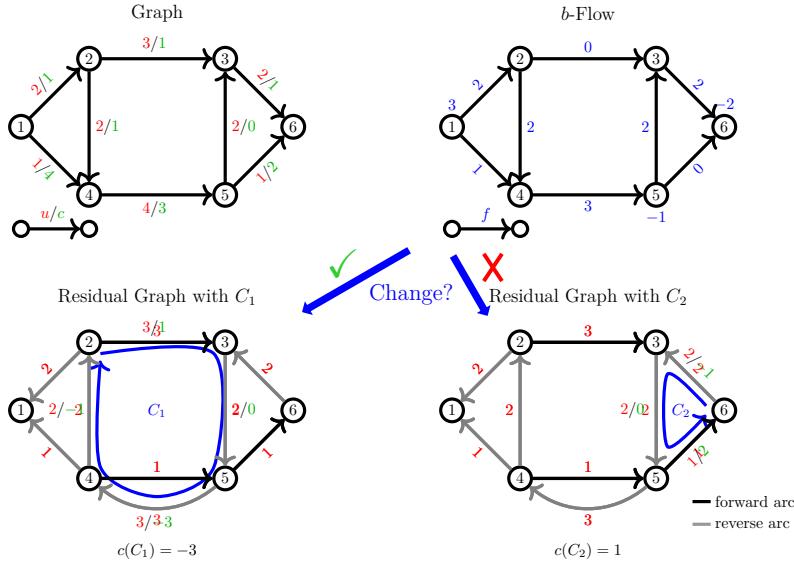
- $f'(a) := f(a) + \gamma$, if $a \in A(C)$ and a is forward arc
- $f'(a) := f(a) - \gamma$, if $\overleftarrow{a} \in A(C)$ and a is reverse arc of \overleftarrow{a}
- $f'(a) := f(a)$, otherwise

Fig. 6.13.: Residual graph and f -augmenting cycle (No. 54)

- Note, augmenting a b -flow f along an f -augmenting cycle leads to another b -flow

Fig. 6.14.: Augmenting leads to a new b -flow (No. 55)

- How can we decide based on these insights how to change a b -flow?

Fig. 6.15.: How should we change our b -flow? (No. 804)

- Extend residual network by costs c

Definition 86 (residual cost). Let (G, u, c, b) be a network. We define the *residual costs* $c : A(\overset{\leftrightarrow}{G}) \rightarrow \mathbb{R}$ by $c(a) = c(a)$ and $c(\overset{\leftarrow}{a}) = -c(a)$ for all $a \in A(G)$.

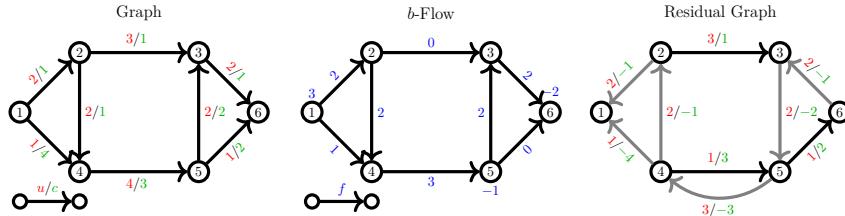
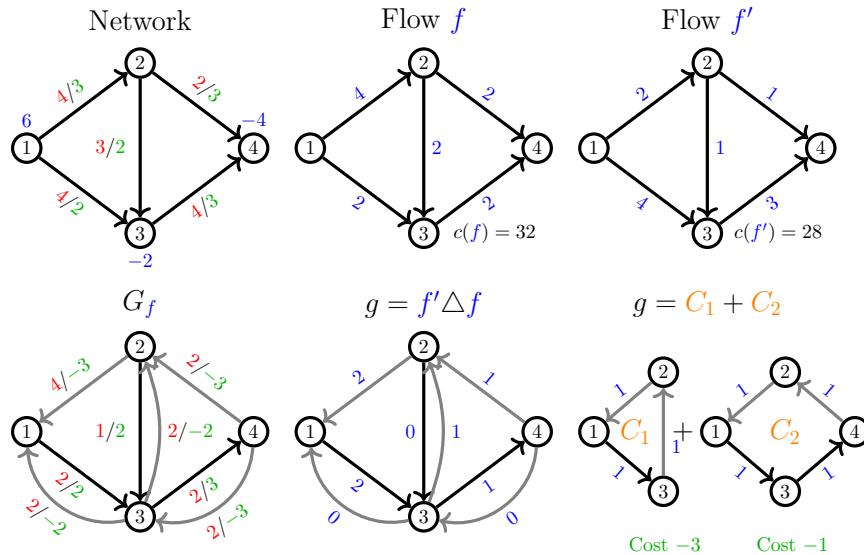


Fig. 6.16.: Residual cost (No. 769)

- Note, the residual costs are independent of a flow f
- Putting everything together, we obtain the following optimality criterion

Theorem 87 (Klein 1967). *Let (G, u, c, b) be an instance of the MCFP. A b -flow f is of minimum cost if and only if there is no f -augmenting cycle with negative total cost.*

Fig. 6.17.: If f is not optimal, a negative cycle exists (No. 58)

Proof. Properties of $f' \Delta f$

- (\Leftarrow): \exists no f -augmenting cycle C with negative cost
- Assume: f is not optimal
- \Rightarrow there exists b -flow f' with $c(f') < c(f)$
- Consider $g := f' \Delta f$
- *Claim:* $c(g) = c(f') - c(f)$.
Proof of Claim:
 - Consider $a \in A(G)$ and $c(a)g(a) + c(\overleftarrow{a})g(\overleftarrow{a})$
 - If $f'(a) > f(a) \Rightarrow c(a)g(a) + c(\overleftarrow{a})g(\overleftarrow{a}) = c(a)(f'(a) - f(a))$
 - If $f(a) > f'(a) \Rightarrow c(a)g(a) + c(\overleftarrow{a})g(\overleftarrow{a}) = c(\overleftarrow{a})(f(a) - f'(a)) = c(a)(f'(a) - f(a))$
 - If $f(a) = f'(a) \Rightarrow c(a)g(a) + c(\overleftarrow{a})g(\overleftarrow{a}) = 0 = c(a)(f'(a) - f(a))$

• \Rightarrow

$$\begin{aligned}
 c(g) &= \sum_{\substack{a \in A(G) \\ a \in A(G)}} c(a) \cdot g(a) \\
 &= \sum_{a \in A(G)} \left(c(a) \cdot g(a) + c(\bar{a}) \cdot g(\bar{a}) \right) \\
 &= \sum_{a \in A(G)} c(a) (f'(a) - f(a)) \\
 &= c(f') - c(f)
 \end{aligned}$$

□C

- By the Flow Decomposition Theorem, g is a linear combination of cycles C_1, \dots, C_k in G_f
- $\Rightarrow 0 \stackrel{\text{Claim}}{>} c(g) = \lambda_1 c(C_1) + \lambda_2 c(C_2) + \dots + \lambda_k c(C_k)$ with $\lambda_i > 0$
- $\Rightarrow \exists C_i$ with negative cost
- $\Rightarrow \exists f$ -augmenting cycle with negative cost
- (\Rightarrow) : f is optimal
- Assume: there exists an f -augmenting cycle C with $c(C) < 0$
- Augment f along C by $\gamma = \min_{a \in A(C)} u_f(a) > 0$
- \Rightarrow change in cost:

$$\sum_{a \in A(G) \cap A(C)} \gamma c(a) - \sum_{\substack{a \in A(G) \\ \bar{a} \in A(C)}} \gamma c(a) = \gamma c(C) < 0$$

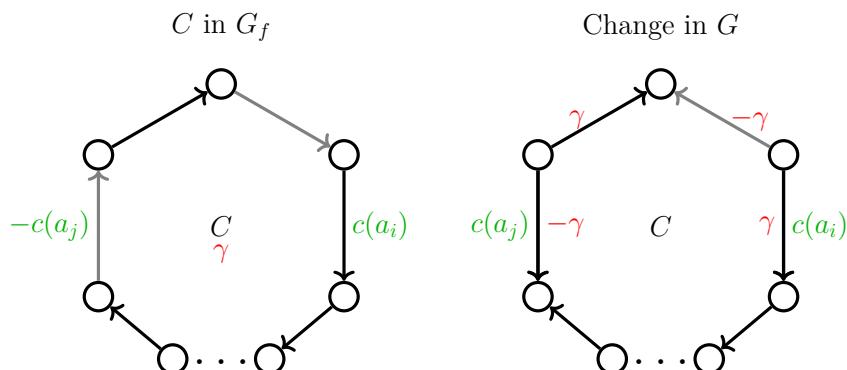


Fig. 6.18.: Change in cost (No. 59)

- Contradiction to the optimality of f

□

- Use this criterion to develop an algorithm for the MCF problem

Algo. 6.1 Minimum Cycle-Canceling Algorithm

Input: A digraph G , upper capacities $u : A(G) \rightarrow \mathbb{N}$, arc costs $c : A(G) \rightarrow \mathbb{Z}$, balances $b : V(G) \rightarrow \mathbb{Z}$

Output: A minimum cost b -flow f

Method:

Step 1: Find a feasible b -flow via a transformation into an maximum (s, t) -flow problem

Step 2: Find a directed cycle C in G_f with negative cost
If C does not exist **then stop**

Step 3: Compute $\gamma := \min_{a \in E(C)} u_f(a)$. Augment f along C by γ .
Go to Step 2

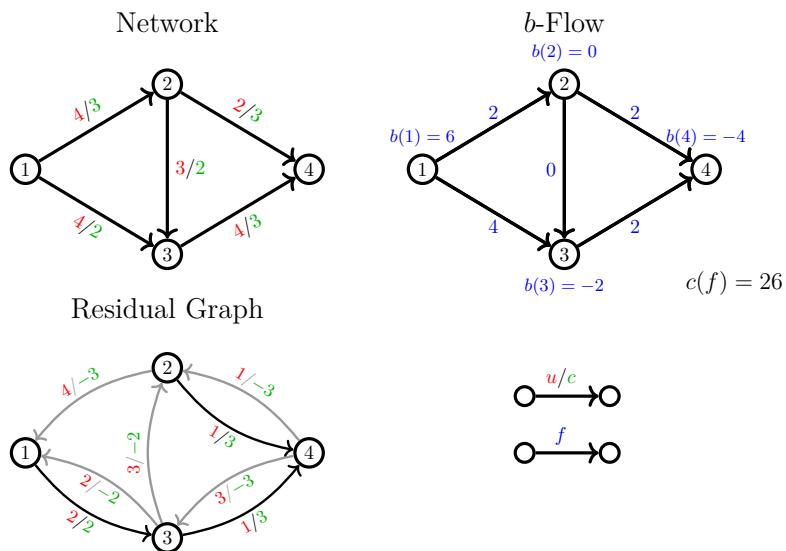


Fig. 6.19.: Minimum cycle-canceling algorithm (No. 770)

- Problem: Which negative cycle?

Any:

- change in cost possibly just 1
- $\Rightarrow \mathcal{O}(mc_{\max})$ iterations with $c_{\max} = \max_{a \in A} c(a)$

Min cost:

- best improvement in each iteration
- strongly **NP**-hard to find

Min mean cost:

- in general better improvement than 1
- construction in polynomial time
- guarantees polynomial run-time of MCF algorithm

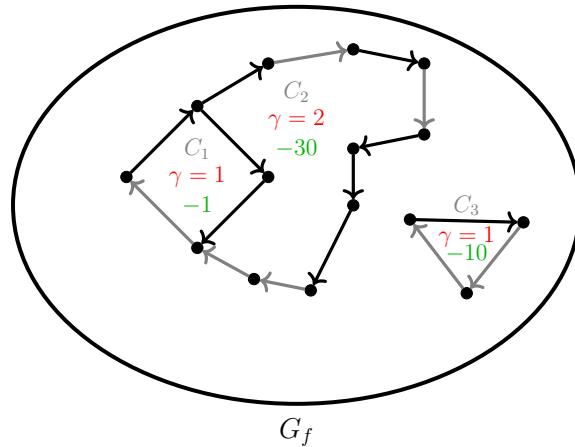


Fig. 6.20.: Different cycles: C_1 any (worst case of improvement), C_2 minimum cost \Rightarrow best improvement, C_3 minimum mean cost (No. 61)

6.3. Successive Shortest Path Algorithm

- Idea Successive Shortest Path
 - For $b(s) = 1, b(t) = -1, b(v) = 0 \forall v \in V \setminus \{s, t\}$: optimal solution is shortest path

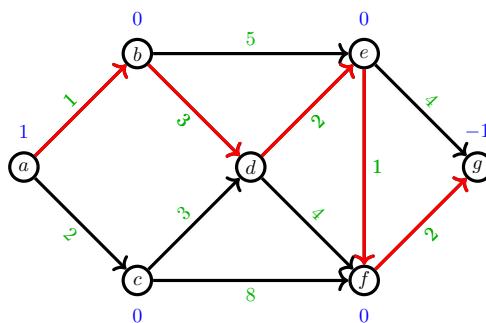


Fig. 6.21.: Shortest path approach (No. 70)

- Idea: always send flow along shortest path (unit per unit)
- Problem: infeasible
 - \Rightarrow compute shortest path in residual graph

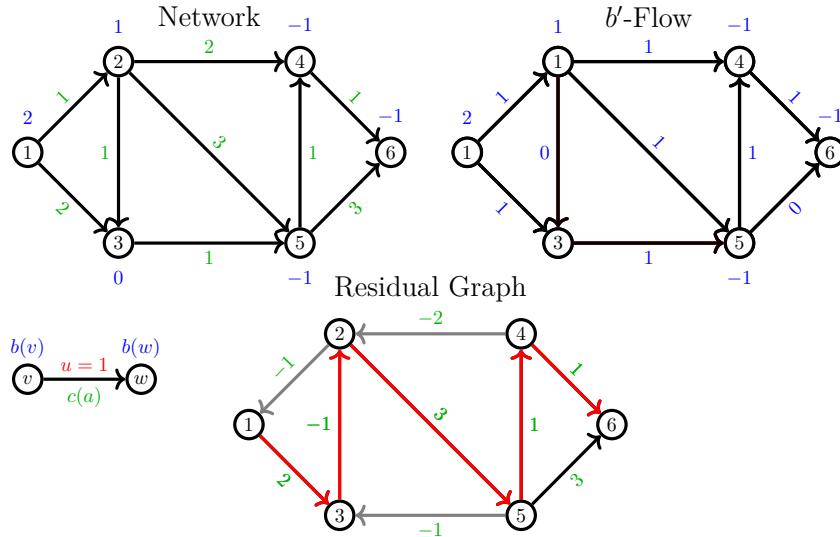


Fig. 6.22.: Successive shortest path algorithm (No. 71)

Theorem 88 (Jewell 1958, Iri 1969, Busacker & Gowen 1961). Let (G, u, c, b) be an instance of the MCF problem, and let f be a minimum cost b -flow. Let p be a shortest (s, t) -path in G_f w.r.t. c for some s and t . Let f' be a flow obtained when augmenting f along p by at most the minimum residual capacity γ on p . Then f' is a minimum cost b' -flow for $b'(s) = b(s) + \gamma$, $b'(t) = b(t) - \gamma$, and $b'(v) = b(v) \forall v \in V \setminus \{s, t\}$.

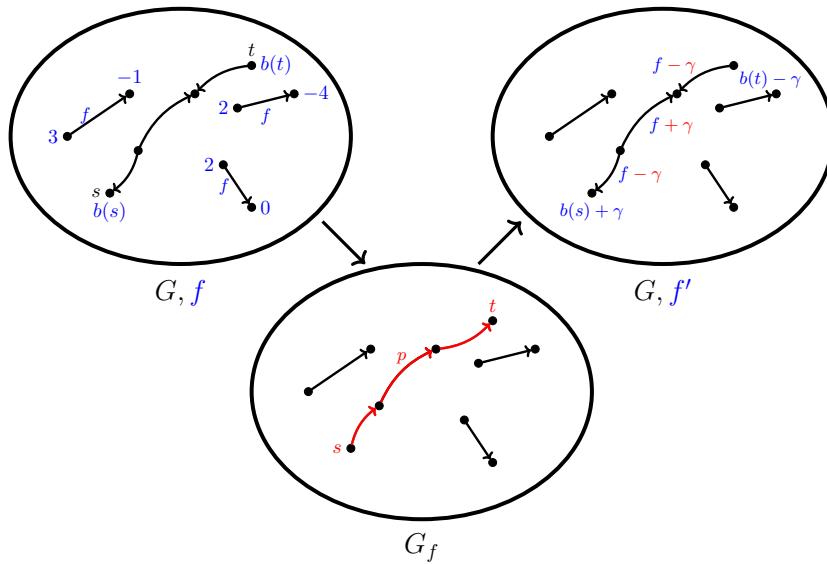
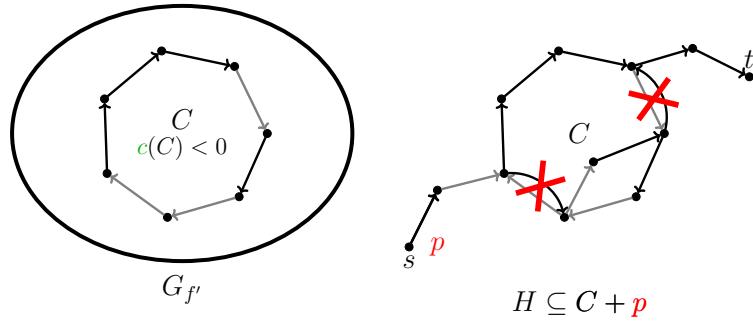


Fig. 6.23.: Obtaining a new MCF by using a shortest path (No. 72)

Proof. No negative cycle is constructed

- f' is a b' -flow
- Suppose: f' is not a minimum cost b' -flow
- $\Rightarrow \exists$ directed cycle C in $G_{f'}$ with negative cost

- Set $H = C + p$ and delete pairs of reverse arcs

Fig. 6.24.: Construction of H (No. 73)

- *Claim:* $A(H) \subseteq A(G_f)$

Proof:

- Assume: $\exists a \in A(H) \setminus A(G_f)$
- $\Rightarrow a \in A(C)$, since $A(p) \subseteq A(G_f)$
- a is added to G_f' due to augmentation along p
- $\Rightarrow a$ is reverse arc of $a' \in A(p)$
- \Rightarrow by construction, a and a' are deleted from H
- \Rightarrow contradiction

□C

- \Rightarrow

$$\begin{aligned} c(A(H)) &\stackrel{(a)}{=} c(A(C)) + c(p) \\ &\stackrel{(b)}{<} c(p) \end{aligned}$$

- (a) cost of reverse arcs add up to 0
- (b) $c(A(C)) < 0$

- *Claim:* H consists of: a) one (s, t) -path p' and b) several directed cycles C_1, \dots, C_k

Proof:

- Due to the construction of H , every vertex $v \in V(H) \setminus \{s, t\}$ has the same in- and out-degree
- Add one arc (t, s) to H to obtain H'
- \Rightarrow in-degree matches the out-degree of s and t
- Consider the connected component of H'
- In every component, we can construct an eulerian tour
- One tour contains the arc (t, s)
- Deleting this arc from the tour leads to an (s, t) -path p'

□C

- Since f has minimum cost $\Rightarrow c(A(C_i)) \geq 0$
- $\Rightarrow c(p') \leq c(A(H)) < c(p)$
- Contradiction to choice of p

□

- Use this property:

- Start with some minimum cost flow
- Increase flow send in the network unit per unit
- Send flow along shortest path in G_f between nodes with unsatisfied balances
- How can we obtain a minimum cost flow?
 - If $c(a) \geq 0 \forall a \in A(G) \Rightarrow f = 0$ is an optimum b -flow for $b = 0$
 - If some $c(a) < 0$ and all capacities bounded:
 - saturate all arcs of negative cost and bounded capacities
 - change the b -value accordingly
 - \Rightarrow just positive arc cost in residual graph
 - \Rightarrow flow is optimal
 - If some $c(a) < 0$:
 - Delete all arcs with bounded capacities
 - If augmenting cycle exists \Rightarrow instance is unbounded
 - Otherwise:
 - Set upper capacity of unbounded arcs in original graph to an upper bound, e.g.,

$$U = \sum_{\substack{a \in A(G) \\ u(a) < \infty}} u(a) + \sum_{v \in V} |b(v)|$$

- \Rightarrow case 2

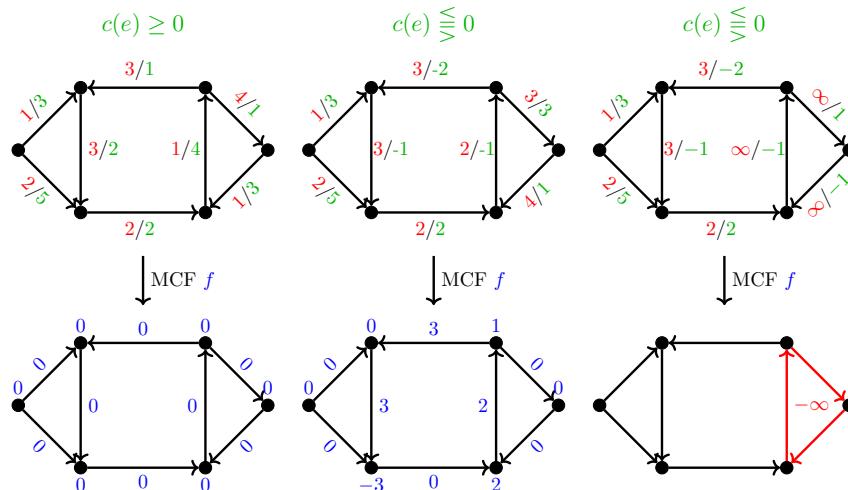


Fig. 6.25.: Finding any MCF (No. 74)

Algo. 6.2 Successive Shortest Path Algorithm

Input: A digraph G , upper capacities $u : A(G) \rightarrow \mathbb{N} \setminus \{\infty\}$, arc costs $c : A(G) \rightarrow \mathbb{Z}$, balances $b : V(G) \rightarrow \mathbb{Z}$

Output: A minimum cost b -flow f

Method:

Step 0: • Set $f(a) = 0 \forall a \in A(G)$ with $c(a) \geq 0$ and $f(a) = u(a) \forall a \in A(G)$ with $c(a) < 0$
• Set $b'(v) = \sum_{\substack{a \in \delta^+(v) \\ c(a) < 0}} u(a) - \sum_{\substack{a \in \delta^-(v) \\ c(a) < 0}} u(a)$

Step 1: Set $\Delta(v) = b(v) - b'(v) \forall v \in V(G)$

Step 2: If $\Delta(v) = 0 \forall v \in V$ then stop

Else choose a vertex s with $\Delta(s) > 0$ If non exists stop

• Choose a vertex t with $\Delta(t) < 0$ s.t. t is reachable from s in G_f
If there is no such t then stop (There exists no b -flow)

Step 3: Find an (s, t) -path p in G_f of minimum cost

Step 4: Compute $\gamma := \min\{\min_{a \in A(p)} u_f(a), \Delta(s), -\Delta(t)\}$

Augment f along p by γ

Set $\Delta(s) = \Delta(s) - \gamma$, $\Delta(t) = \Delta(t) + \gamma$

Go to Step 2

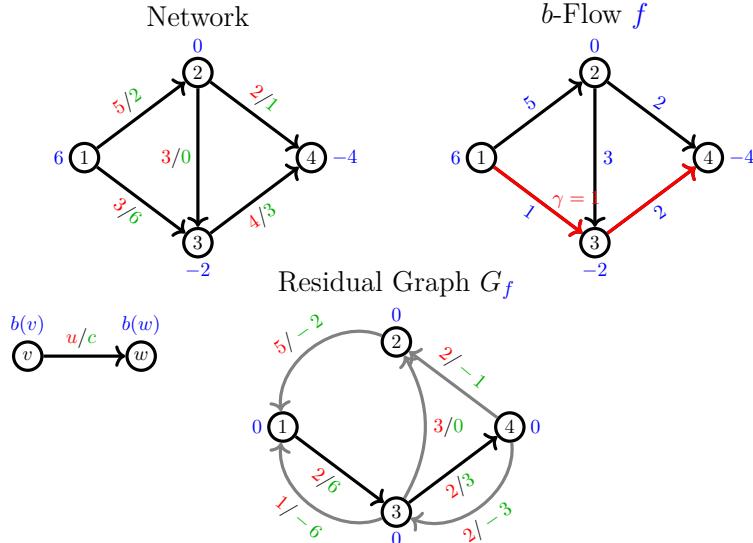


Fig. 6.26.: Successive shortest path (No. 75)

Theorem 89 (Edmonds and Karp 1972). For integral capacities and supplies, the Successive Shortest Path Algorithm can be implemented with a running time of $\mathcal{O}(nm + B(n^2 + m))$ with $B = 0.5 \cdot \sum_{v \in V(G)} |b(v)|$.

- Several modifications of the algorithm to improve the running time

- Basic idea:
 - Transformation of problem, s.t., $u(a) = \infty$
 - Choose paths with large augmenting value γ
 - Start with $\gamma \approx B$
 - Iteratively reduce γ by a factor of two



7. Maximum Matchings

7.1. Optimality Criteria and Edmonds Matching Algorithms

- In practice often one
 - needs to form pairs within a group (tandem-students)
 - or assign elements to another one (worker to jobs)

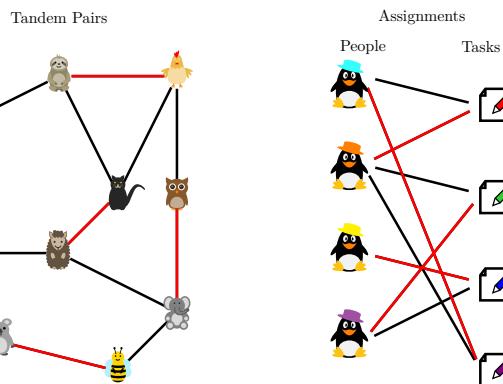


Fig. 7.1.: Pairs are everywhere (No. 817)

Definition 90 (Matching). Let G be an undirected graph. A *matching* in G is a set $M \subseteq E(G)$ of pairwise disjoint edges, i.e., no two edges from M are incident to the same node. A node is *covered* by M if $v \in e$ for some $e \in M$; otherwise v is *exposed* (by M). A matching M is *perfect* if every node in G is incident to an edge in M .

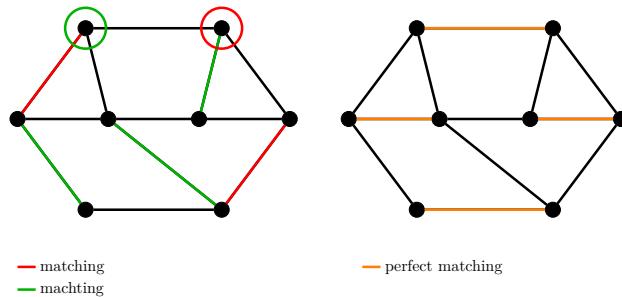


Fig. 7.2.: Matchings (No. 482)

- Often, the objective is to find a matching with maximum cardinality

Definition 91. Maximum matching problem

Given: Undirected graph $G = (V, E)$

Find: Matching $M \subseteq E(G)$ with maximum cardinality, i.e., there exists no matching M' with $|M'| > |M|$

- The best possible outcome of a maximum matching problem is to find a matching M with $|M| = \frac{|V|}{2}$, i.e., a perfect matching
- Question: Can we find a criterion when a matching is a maximum matching?

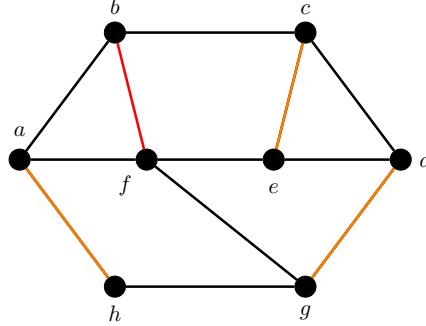
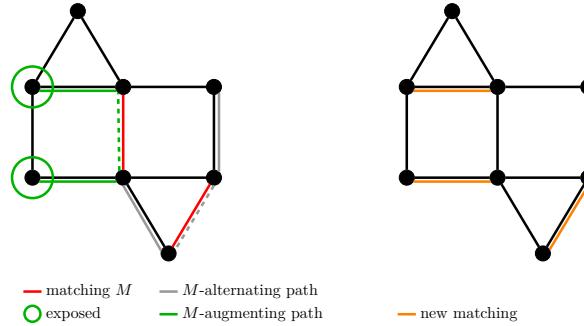


Fig. 7.3.: Finding a maximum matching (No. 773)

- Consider the following matching in G
- There are two nodes e and a that are exposed
- We find a (e, a) -path which alternating uses non-matching and matching edges
- Replace the matching edges by the non-matching edges on the path
- \Rightarrow new matching with one more edge
- These paths are called augmenting paths, as in the flow theory

Definition 92 (M -alternating path, M -augmenting path). Let M be a matching in a graph G . An M -alternating path in G is an elementary path W in G for which $E(W) - M$ is a matching (i.e. W contains alternating matching and non-matching edges). An M -alternating path W is M -augmenting if both end nodes are exposed.

Fig. 7.4.: M -augmenting path (No. 499)

- If M is a matching and W an M -augmenting path, then we obtain a matching M' with $|M'| = |M| + 1$ by exchanging matching and non-matching edges
- Note, that every M -augmenting path has an odd number of edges

Theorem 93 (Berge, 1957). A matching M in a graph is maximum if and only if no M -augmenting path exists.

Proof. Proof of Berge's Theorem

- (\Rightarrow): Let M be a maximum matching
- Assume, there exists an M -augmenting path W
- Replace edges from M along W with edges from $E(W) - M$
- \Rightarrow increase cardinality of M by 1
- Contradiction to M maximum
- (\Leftarrow): There exists no M -augmenting path
- Assume: M is not maximum
- \Rightarrow a matching M' with $|M'| > |M|$ exists
- Consider the symmetrical difference $M' \Delta M := (M' - M) \cup (M - M')$ and $G' := (V, M' \Delta M)$

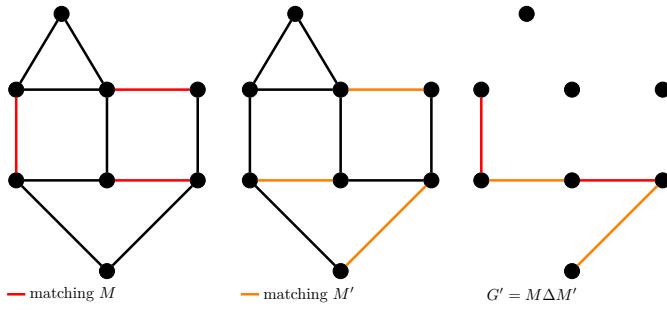


Fig. 7.5.: symmetrical difference (No. 500)

- \Rightarrow all nodes in G' have a degree smaller or equal to 2
- Consider the connected components of G'
- These are: isolated nodes, elementary cycles, and elementary paths

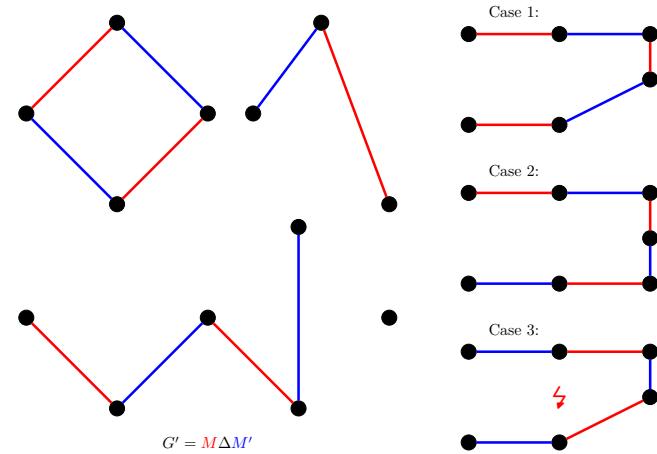


Fig. 7.6.: symmetrical difference (No. 501)

- In cycles: equal number of edges w.r.t. M' and M
- Consider an elementary path p and denote with $M(p) = M \cap E(p)$ and $M'(p) = M' \cap E(p)$
- We have three types of elementary paths

1. begins and ends with an edge from M
 - $\Rightarrow |M(p)| > |M'(p)|$ along the path
 2. begins with an edge from M , ends with an edge from M'
 - $\Rightarrow |M(p)| = |M'(p)|$ along the path
 3. begins and ends with edge from M'
 - $\Rightarrow M$ -augmenting \Rightarrow cannot occur
- Cases result in: $|M| \geq |M'|$, contradiction

□

- Berge's Theorem provides the idea for an algorithm for computing a maximum matching similar to flows:
 - Iteratively find augmenting paths until none exist

7.2. Edmonds Matching Algorithms

- Edmonds used the optimality criteria introduced by Berge to compute a maximum matching
- In order to efficiently find an M -alternating path, we use an M -alternating trees

Definition 94 (M -alternating tree). Let G be a graph and M be a matching. An M -alternating tree is a tree with an exposed root r , whose paths from the root to the leaves are M -alternating paths.

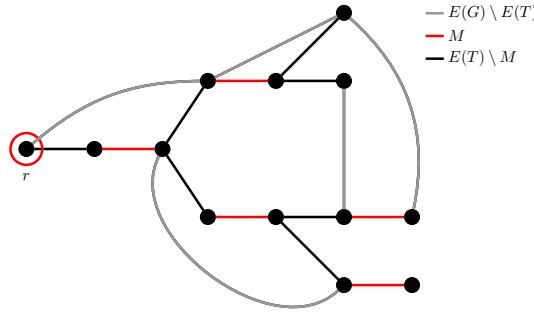


Fig. 7.7.: M -alternating tree (No. 504)

- M -alternating trees model search trees for M -augmenting paths

Lemma 95. If a leaf in an M -alternating tree is exposed, the path from the root to this leaf is an M -augmenting path in G .

Proof. Due to the definition of M -alternating trees

□

- However, such a tree does not always contain an M -augmenting path, even if one exists

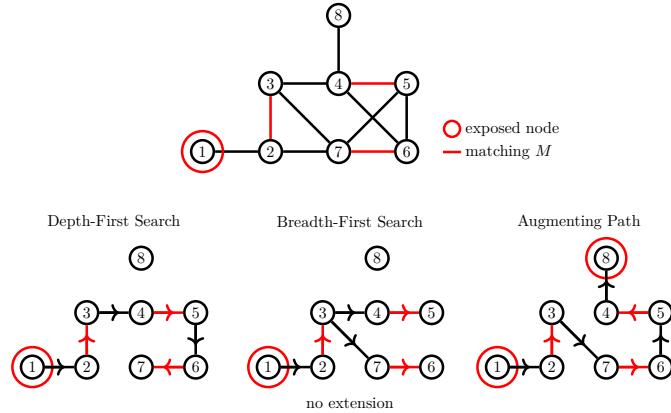


Fig. 7.8.: How to find an augmenting path? (No. 502)

- The problem is based on cycles of odd length
- More formally, we define these cycles as blossoms

Definition 96 (Blossom). Let G be a graph, let M be a matching and let T be an M -alternating tree with root r . Set $\ell(v)$ the length of the path from r to v . A *blossom* $B = v_0v_1 \dots v_k$ is a fundamental cycle (Def. ??) defined by a non-tree edge uv if one of the two cases holds true

1. uv is a matching edge and $\ell(v)$ and $\ell(u)$ are odd
2. uv is a non-matching edge and $\ell(v)$ and $\ell(u)$ are even

The vertex v_0 is called the *base* of the blossom. The *stem* of the blossom is the path from the root to base in the tree.

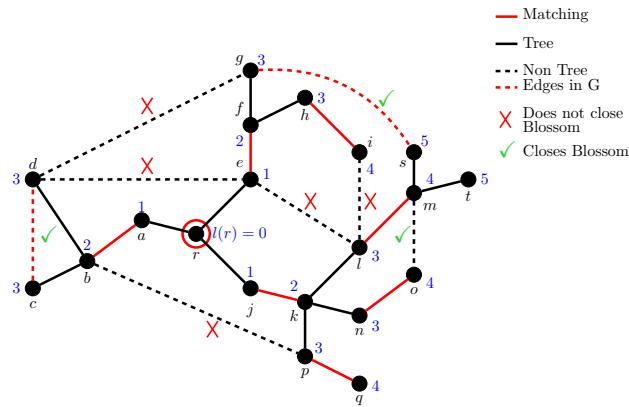


Fig. 7.9.: Blossoms (No. 820)

- Let us have a closer look at blossoms
- They have the following properties

Lemma 97. For a blossom $B = v_0v_1 \dots v_k$ it holds:

- v_0v_1 and v_0v_{k-1} are non-matching edges
 - Any edge in G incident to $B \setminus \{v_0\}$ is a non-matching edge
 - For $v_i \in B \setminus \{v_0\}$, there exists an M -alternating path (v_0, v_i) s.t. the last edge is a matching edge

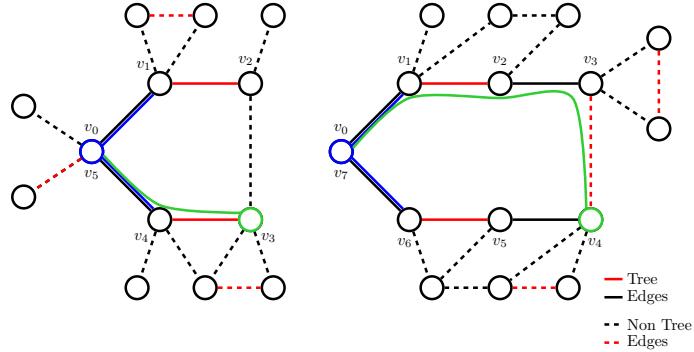


Fig. 7.10.: Blossoms and their properties (No. 821)

Proof. Exercise

- Hence, blossoms encode alternatives in the search for M -augmenting paths
 - An alternating path passing the base and then going through a blossom can choose two different alternatives to any vertex
 - We do not know which alternating path through the cycle to choose
 - Solution by Edmonds (1965): postpone the decision
 - To that end, contract parts of the graph
 - On the contracted parts, the decision on the direction is made only after more information is available

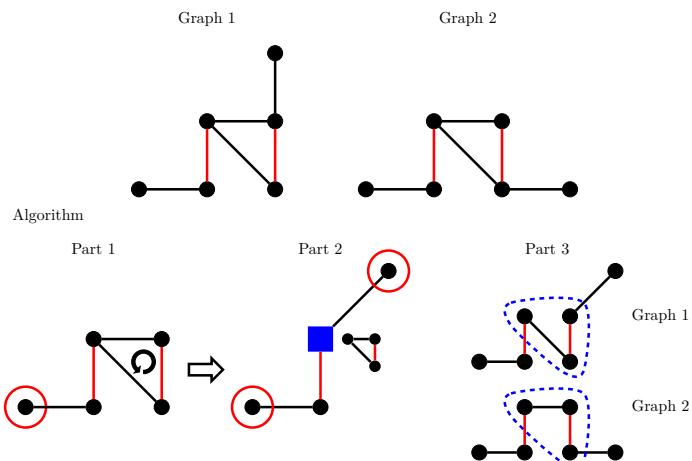


Fig. 7.11.: Odd cycles (No. 503)

Definition 98 (Shrinking, contraction). Let B be a blossom in a graph G . Then, *shrinking* or *contracting* the blossom B results in a new graph $G^c[V(B)]$. The graph $G^c[V(B)]$ is created by

- replacing $G[V(B)]$ by a pseudo-node v_B
- introducing edges v_Bx for all $x \in V(G) \setminus V(B)$ connected to a node $y \in V(B)$ in G .

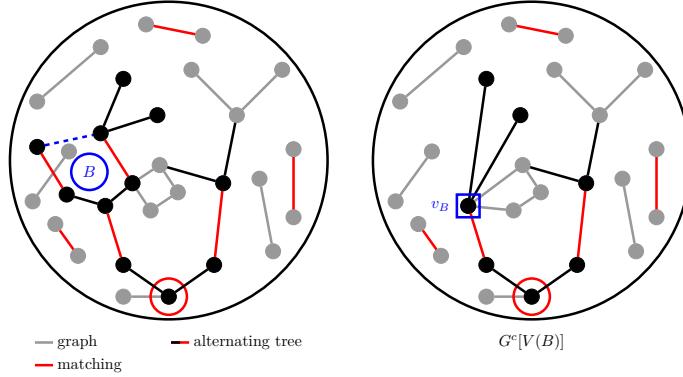


Fig. 7.12.: Contraction of a blossom (No. 507)

- After shrinking we can transfer the current matching

Definition 99 (corresponding matching). Let B be a blossom of G w.r.t. a matching M and an M -alternating tree T with base v_0 . If $\exists xv_0 \in M \cap E(T)$, set $M' = M - E(B) + xv_B$ where v_B represents the blossom B in $G^c[V(B)]$. Otherwise set $M' = M - E(B)$. Then M' is called the *corresponding matching by contraction*.

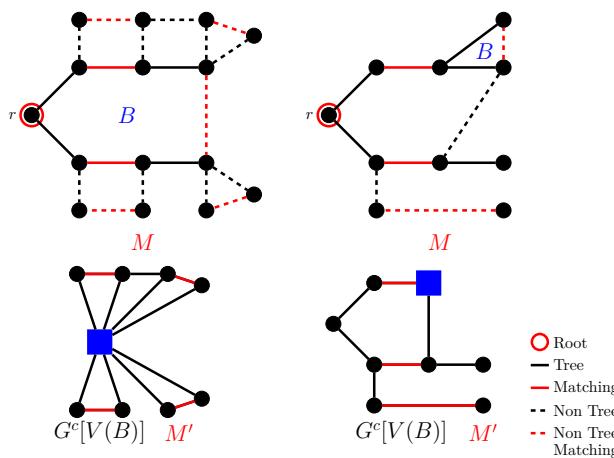


Fig. 7.13.: Corresponding matching by contraction (No. 822)

- We need to prove that M' really is a matching

Lemma 100. Let B be a blossom of G w.r.t. a matching M and an M -alternating tree T with base v_0 . Then the corresponding matching M' by contraction is a matching in $G^c[V(B)]$.

Proof. Exercise □

- Most important: a matching is optimal in G if and only if its corresponding matching is optimal in the shrunk graph

Lemma 101. Let B be a blossom of G w.r.t. a matching M and an M -alternating tree T with base v_0 and M' the corresponding matching by contraction. Then the following are equivalent:

- An M -augmenting path in G exists.
- An M' -augmenting path in $G^c[V(B)]$ exists.

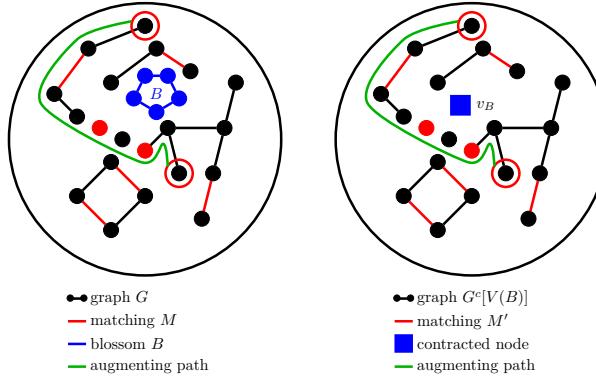


Fig. 7.14.: Optimality remains (No. 509)

- The difficulty is the distinction of v_B being exposed or not exposed

Proof. Use the two directions to traverse a blossom

- (\Leftarrow): Let W be an M' -augmenting path in $G^c[V(B)]$
 - Case (a): W does not pass through the pseudo-node v_B
 - $\Rightarrow W$ is M -augmenting in G , since only nodes and edges from G are used
 - Case (b): W contains the pseudo-node v_B
 - Case (b1): v_B is not exposed
 - $\Rightarrow W$ has the form $W = W_1 + xv_B + v_By + W_2$ with $xv_B \in M'$ and $v_By \in E(G^c[V(B)]) - M'$
 - Let $y' \in B$ with $y'y \in E(G)$
 - Blossom Lemma 97: an M -alternating path W' in B from base v to node y' in B exists, where the last edge in W' is a matching edge
 - $\Rightarrow W_1 + xv + W' + y'y + W_2$ is M -augmenting in G
 - Case (b2): v_B is exposed

- $\Rightarrow W$ has the form $W = v_B x + W_1$ with $xv_B \notin M$
- Let $x' \in B$ with $x'x \in E(G)$
- Blossom Lemma 97: Choose an alternating path W' in B that starts in the base v of B and ends in x' with a matching edge
- v_B exposed $\Rightarrow v$ exposed
- $\Rightarrow W' + x'x + W_1$ is an M -augmenting path in G

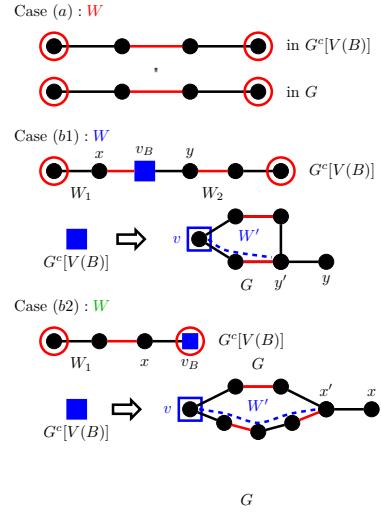


Fig. 7.15.: Optimal remains optimal (No. 510)

- (\Rightarrow): Exercise

□

- Idea of Edmonds Matching Algorithm:
 - construct a maximal M -alternating tree
 - if an M -augmenting path exists, augment and restart with new matching
 - if a blossom exists, shrink and restart in the resulting graph
 - if both do not apply, delete all vertices of the tree and restart

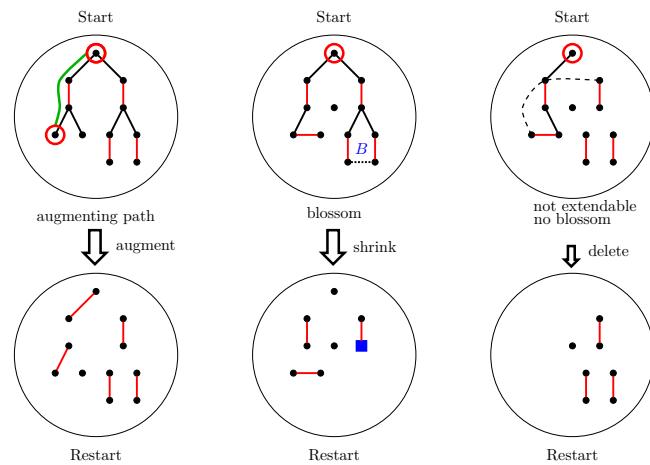


Fig. 7.16.: Idea Edmonds' Matching Algorithm (No. 569)

- The main question is: can we really ignore parts of the graph if neither a blossom nor an augmenting path exists
 - Define it more formally

Definition 102 (Hungarian Tree). A *Hungarian tree* w.r.t. M is a maximal M -alternating tree (i.e. not extendable), that has no blossoms and no exposed leaves (i.e. contains no M -augmenting path).

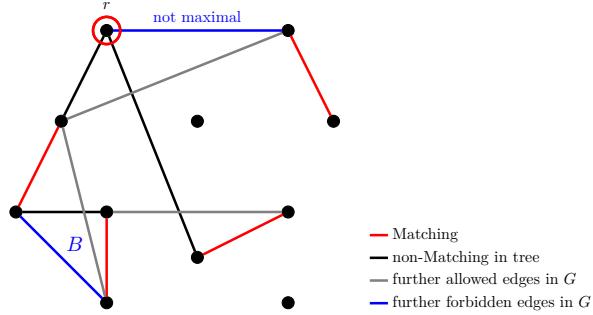


Fig. 7.17.: Hungarian tree (No. 570)

- The following theorem states, that this part of the graph can be ignored

Theorem 103. *Let G be a graph and let M be a matching in G . Let H be a Hungarian tree in G . Then the following two properties are equivalent:*

- There exists an M -augmenting path in G .
 - There exists an M -augmenting path in $G - H$, where $G - H$ is the induced subgraph of $V(G) \setminus V(H)$.

Proof. Definition of Hungarian tree

- Let r be the root of the Hungarian tree H
 - Let $\ell(v)$ be the length of the M -alternating path from r to v for all $v \in V(H)$

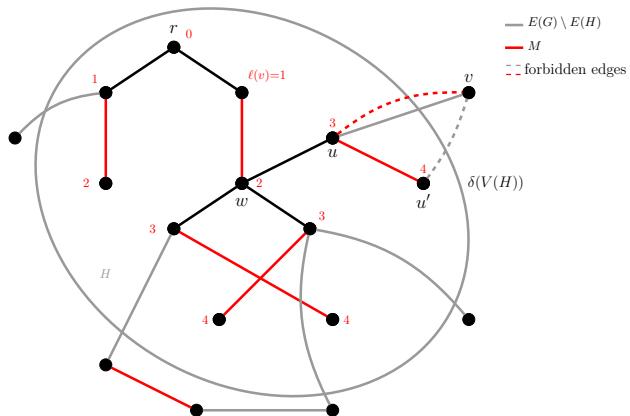


Fig. 7.18.: Hungarian tree in a graph (No. 762)

- *Claim 1:* Let $uv \in \delta(V(H)) \setminus \{r\}$ with $u \in V(H)$. Then, a) $uv \notin M$ and b) $\ell(u)$ is odd.

Proof of Claim:

- Any vertex $v \in V(H) \setminus \{r\}$ is incident to a matching edge
- $\Rightarrow uv \notin M$
- Assume $\ell(u)$ is even
- Consider the path $p_{[r,u]}$ from r to u in H
- Then, $p_{[r,u]} \cap \delta(u) = wu$ with $wu \in M$
- $\Rightarrow p_{[r,u]} + uv$ is an M -alternating path
- Since $v \notin V(H)$, H can be extended by uv (Contradiction) $\square C1$

- (\Leftarrow) : Consider M -augmenting path $p = v_0 \dots v_k$ in $G - H$
- $\Rightarrow v_0 \neq r \neq v_k$
- Since $\delta(V(H)) \cap M = \emptyset$ (Claim 1), $\Rightarrow p$ is M -augmenting path in G
- (\Rightarrow) : Denote with G' the induced subgraph of $V(H)$ in G (Definition IV)
- *Claim 2:* Let $uv \in E(G') \setminus E(H)$. Then, a) uv is a non-matching edge and b) if $\ell(u)$ is even $\Rightarrow \ell(v)$ is odd.

Proof of Claim:

- The node r is exposed and any vertex besides r is incident to a matching edge in H

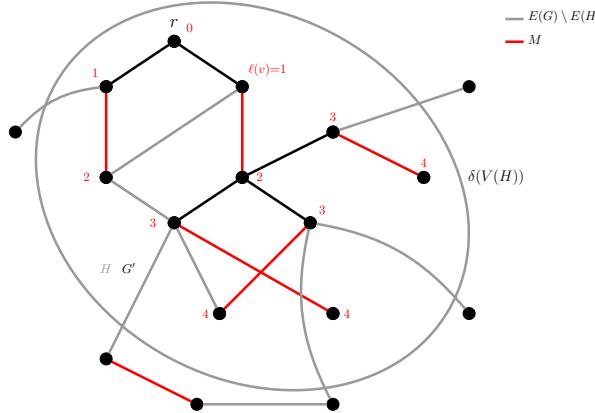


Fig. 7.19.: Properties of G' (No. 823)

- $\Rightarrow uv \notin M$
- If $\ell(u)$ and $\ell(v)$ are even $\Rightarrow uv$ defines a blossom (Contradiction) $\square C2$

- *Claim 3:* Let $p = v_0 \dots v_k$ be an M -alternating path in G' with $v_0v_1 \in M$, $v_{k-1}v_k \in M$ and $\ell(v_0)$ odd. Then, $\ell(v_k)$ is even.

Proof of Claim:

- Let $v_{i_1}v_{i_1+1}, \dots, v_{i_t}v_{i_t+1}$ be non-tree edges of p , $i_1 < i_2 < \dots < i_t$
- Consider $q_\ell = p|_{[v_{i_{\ell-1}+1}, v_{i_\ell}]}$ with $v_{i_0+1} = v_0$, $1 \leq \ell \leq t$
- Since $v_{i_\ell}v_{i_\ell+1}$ is a non-tree edge, $v_{i_\ell}v_{i_\ell+1} \notin M$
- Since q_ℓ is an M -alternating path, $v_{i_\ell-1}v_{i_\ell} \in M$ and $v_{i_\ell+1}v_{i_\ell+2} \in M$
- Since q_ℓ is an M -alternating path, $|E(q_\ell)|$ is odd
- Since $\ell(v_{i_0+1})$ is odd, $\ell(v_{i_1}) = \ell(v_{i_0+1}) + |E(q_1)|$ is even

- $\stackrel{C_2}{\Rightarrow} \ell(v_{i_1+1})$ is odd
 - \Rightarrow all nodes $v_{i_\ell+1}$ are odd and in particular $\ell(v_{i_\ell+1})$ is odd
 - Since $v_{i_\ell+1}v_{i_\ell+2} \in M$ and $v_{k-1}v_k \in M$, $\Rightarrow |E(p_{|[v_{i_\ell+1}, v_k]})|$ is odd
 - $\Rightarrow \ell(v_k) = \ell(v_{i_\ell+1}) + |E(p_{|[v_{i_\ell+1}, v_k]})|$ is even

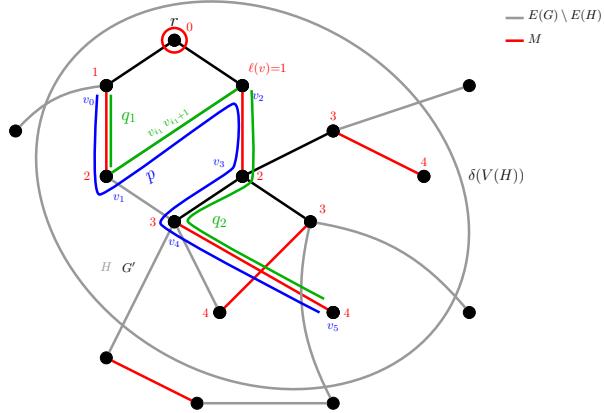


Fig. 7.20.: Properties of G' (No. 824)

□C3

- Let $W = w_0w_1 \dots w_k$ be an M -augmenting path in G

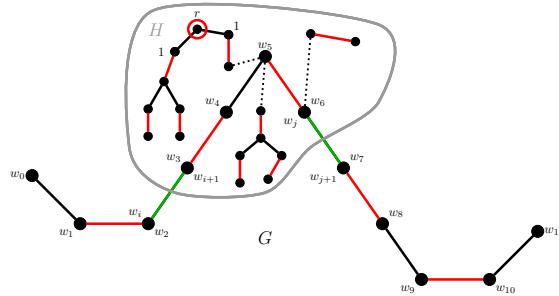


Fig. 7.21.: M -augmenting path (No. 763)

- Prove: W is part of $G - H$
 - Assume W is not part of $G - H$
 - W.l.o.g. $w_0 \in V(G) \setminus V(H)$, since in $V(H)$ only the root node is exposed and if W starts in w_0 , then $W' = v_0v_1 \dots v_k$ with $v_i = w_{k-i}$, $i = 0, \dots, k$ is also an M -augmenting path
 - Let $w_iw_{i+1} \in E(W) \cap \delta(V(H))$ with i minimal
 - Case 1: $w_k \neq r$, i.e., no endnode is r
 - $\Rightarrow \exists w_jw_{j+1} \in E(W) \cap \delta(V(H))$ with $i+1 < j$
 - Let w.l.o.g. be $W|_{[w_{i+1}, w_j]} \in G'$, otherwise choose w_j smaller
 - Since $w_iw_{i+1}, w_jw_{j+1} \in \delta(V(H))$, $\stackrel{C1}{\Rightarrow} w_iw_{i+1} \notin M$ and $w_jw_{j+1} \notin M$ and $\ell(w_{i+1})$ and $\ell(w_j)$ are odd
 - $W|_{[w_{i+1}w_j]}$ is an M -alternating path, $w_{i+1}w_{i+2} \in M$, $w_{j-1}w_j \in M$ and $\ell(w_{i+1})$ is odd

- $\stackrel{C3}{\Rightarrow} \ell(w_j)$ is even, Contradiction
- Case 2: $w_k = r$, i.e., endnode is r
 - Let $w_j w_{j+1} \in E(W) \cap \delta(V(H))$ s.t. $W_{|[w_{j+1}, w_k]} \in G'$
 - $\Rightarrow W_{|[w_{j+1}, w_{k-1}]} \in G'$ is an M -alternating path, $w_{j+1} w_{j+2} \in M$, $w_{k-2} w_{k-1} \in M$ and $\ell(w_{j+1})$ is odd
 - $\stackrel{C3}{\Rightarrow} \ell(w_{k-1})$ is even
 - \Rightarrow Contradiction to $\ell(w_{k-1}) = 1$

□

- Combining all ideas leads to the following algorithm

Algorithm 7.1 Edmonds' Matching Algorithm

Input: Undirected graph G Output: Matching M with maximum cardinalityInitialization: Set $G' = G$, $M = \emptyset$ and $M' = \emptyset$

Method:

- **While** $G' \neq \emptyset$ and an exposed node exists **do**
 - Construct a maximal M' -alternating tree T in G' starting in any exposed node (e.g. Breath-First-Search)
 - **Case 1:** T contains M' -augmenting path
 - Construct corresponding M -augmenting path W in G
 - Augment M along W
 - Set $G' := G$ and $M' := M$ // restart in G with bigger matching
 - **Case 2:** T contains blossom B with base v_0
 - Shrink B to a vertex v_B
 - **If** $xv_0 \in E(T) \cap M'$, set $M' = M' - E(B) + xv_B$
 - **Else** set $M' = M' - E(B)$
 - Set $G' := G'^c[V(B)]$ // restart in smaller graph
 - **Case 3:** T is Hungarian
 - Set $G' := G' - V(T)$ // restart in smaller graph
- **Return** M

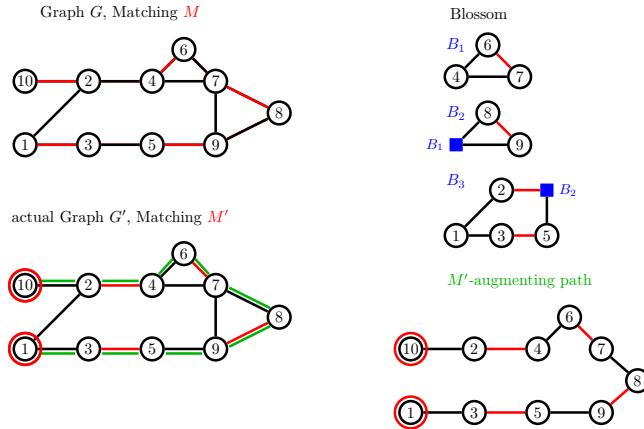


Fig. 7.22.: Edmonds' Matching Algorithm (No. 571)

- Instead of starting with $M_0 = \emptyset$, you can start with any matching

Theorem 104. *Edmonds' Matching Algorithm computes a maximum matching and has a run-time of $\mathcal{O}(n^2m)$.*

Proof. Bound number of times each case statements can occur

- At the termination of the algorithm no M -augmenting path exists (Lemma 101 and Theorem 103)
- $\Rightarrow M$ is a maximum matching (Theorem of Berge 93)
- Consider the run-time
- There are at most n exposed nodes
- \Rightarrow at most n augmentations of the matching are possible
- Per augmentation,
 - a BFS is performed: $\mathcal{O}(m)$
 - At most $\frac{n}{2}$ -times a blossom is shrunk (every contraction reduces the number of nodes by at least 2):
 - Shrinking/Expansion costs: $\mathcal{O}(m)$
 - At most $\frac{n}{3}$ -times a Hungarian tree is split off
 - finding an augmenting path: $\mathcal{O}(n)$
- In total: $\mathcal{O}(n(m + n \cdot m + n)) = \mathcal{O}(n^2m)$

□

- In the following, we will consider dual problems for the matching problem

7.3. Covering Problems and Upper Bounds

- Dual problems are pairs of optimization problems: $\max\{b(x) \mid x \in X\}$ and $\min\{c(y) \mid y \in Y\}$ where X and Y are the set of feasible solutions and b and c the corresponding objective functions
- *Weak duality:* For all $x \in X$ and $y \in Y$, it holds that $b(x) \leq c(y)$

- *Strong duality*: It holds that

$$\max\{b(x) \mid x \in X\} = \min\{c(y) \mid y \in Y\}$$

- How can we define a dual problem to the maximum matching problem?

Definition 105 (Vertex cover). A *vertex cover* in G is a set $S \subseteq V$ of vertices such that every edge of G is incident to at least one vertex in S .

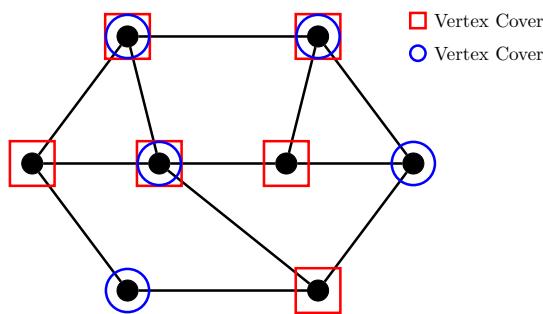


Fig. 7.23.: Vertex cover (No. 772)

- Obviously, $\tau(G) \leq |V|$

Definition 106. Minimum vertex cover problem

Given: Undirected graph $G = (V, E)$

Find: Vertex cover $S \subseteq V$ with minimum cardinality

- One can easily show: the minimum vertex cover problem and the maximum matching problem are weakly dual

Lemma 107. Let M be a matching in an undirected graph and S a vertex cover. Then, $|M| \leq |S|$.

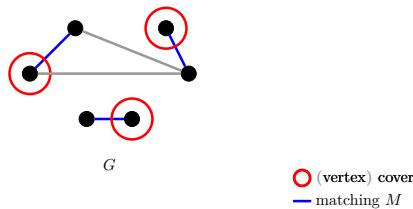


Fig. 7.24.: Bound on matching number by vertex cover (No. 485)

Proof. Matching needs to be covered

- Let M be any matching

- Any vertex cover S of G needs to contain at least one vertex of M
- $\Rightarrow |M| \leq |S|$

□

- The set cover problem just provides weak duality for general graphs

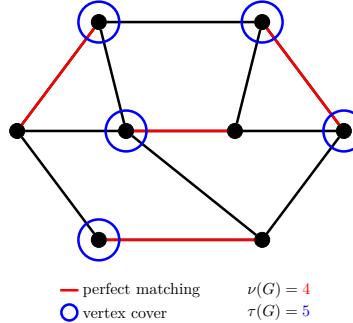


Fig. 7.25.: Vertex cover vs. maximum matching (No. 484)

- New idea: extend the idea of vertex cover to general vertex sets
- This leads to the so-called *odd set cover problem*

Definition 108 (Odd Set Cover (OSC)). Let G be an undirected graph. A family of sets $\mathcal{V} = \{V_1, \dots, V_p\}$ with $V_i \subseteq V(G)$ and $|V_i|$ odd, $1 \leq i \leq p$, is called an *odd set cover* (OSC), if all edges are covered. A set $V_i = \{v\}$ *covers* all edges incident to v ; all other sets V_i *cover* all edges $uv \in E(G)$ with $u, v \in V_i$. The *capacity* c of an OSC is given by $c(\mathcal{V}) = \sum_{i=1}^p c(V_i)$ with $c(V_i) = 1$, if $|V_i| = 1$ and $c(V_i) = r$ if $|V_i| = 2r + 1$ with $r \geq 1$.

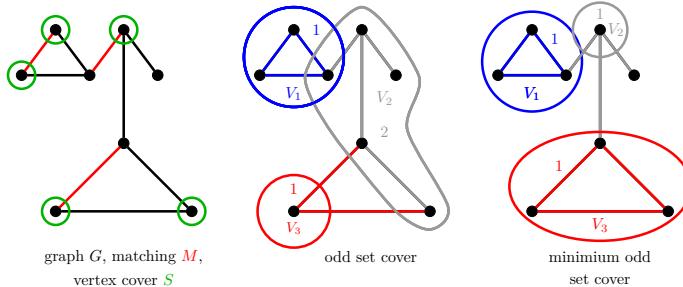


Fig. 7.26.: Odd Set Cover (No. 572)

- If all $|V_i| = 1$, then an OSC is a vertex cover and $c(\mathcal{V})$ equals the cardinality of the vertex cover.
- In general, the objective is to find an odd set cover with minimum capacity

Definition 109. *Minimum odd set cover problem*

Given: undirected graph G

Find: an odd set cover with minimum capacity

- The odd set cover problem is strongly dual to the maximum matching problem

Theorem 110 (Edmonds 1965). *Let G be an undirected graph. Then the cardinality of a maximum matching in G equals the minimum capacity of an odd set cover, i.e.,*

$$\max\{|M| \mid M \text{ is matching in } G\} = \min\{c(\mathcal{V}) \mid \mathcal{V} \text{ is an odd set cover in } G\}.$$

Proof. Proof of Edmonds' Theorem

- *Step 1:* $\max\{|M| \mid M \text{ is matching in } G\} \leq \min\{c(\mathcal{V}) \mid \mathcal{V} \text{ is OSC in } G\}$
 - Let M be a matching and \mathcal{V} an odd set cover
 - Then, M can contain at most $c(V_i)$ of the edges covered by $V_i \in \mathcal{V}$
 - Since all edges are covered, $|M| \leq c(\mathcal{V})$

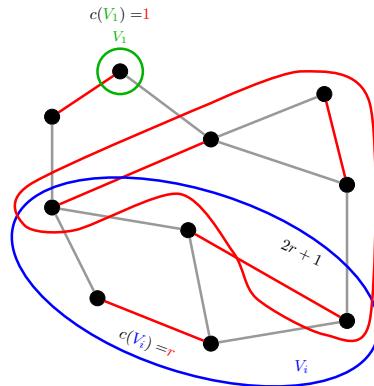


Fig. 7.27.: M smaller than OSC (No. 573)

- *Step 2:* $\max\{|M| \mid M \text{ is matching in } G\} \geq \min\{c(\mathcal{V}) \mid \mathcal{V} \text{ is OSC in } G\}$
 - Idea: construct an OSC with the same capacity as $|M|$
 - Let R be the graph at the termination of the algorithm, i.e., no exposed vertex exists in R
 - Furthermore, consider the Hungarian trees that were deleted in the different iterations

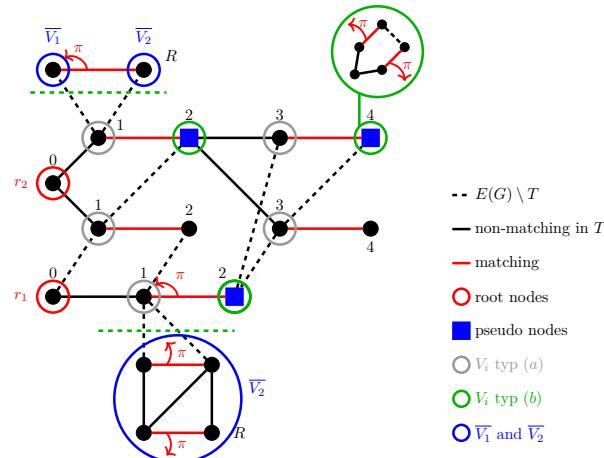


Fig. 7.28.: Construction of an OSC (No. 575)

- Denote with $\ell(v)$ the length of the path to a vertex in the corresponding Hungarian trees starting in its root vertex r
- *Claim:* Each edge $e \in E(G)$ is either
 1. incident to a node with odd length
 2. contained in a blossom
 3. is part of the rest graph R

Proof of Claim:

- In other words:
 1. \nexists edge ab with a has an even length and b has an even length
 2. \nexists edge with a has an even length and b is in R
- In case (1.):
 - ab is a non-matching edge
 - If a and b are in the same Hungarian tree $\Rightarrow ab$ closes a blossom
 - If a and b are in different Hungarian trees
 - \Rightarrow contradiction to definition of Hungarian tree (not extendable)
- In case (2.): $\Rightarrow M$ -alternating tree is not maximal (add ab). Contradiction to Hungarian tree $\square C$
- Use this insight to construct an OSC \mathcal{V}_0 as follows:

Type (a) choose all vertices with odd length $\ell(v)$ as singletons V_i , i.e. single-element vertex sets

Type (b) each blossom forms a multi-element set V_i

Type (c) Remainder set(s)

- R has $2k$ nodes, since no exposed node exists
- $\Rightarrow R$ has k matching edges
- If $k = 0$: \mathcal{V}_0 is finished according to (a) and (b)
- If $k = 1$: add one vertex as single-element set \bar{V} to \mathcal{V}_0
- If $k \geq 2$: add one vertex as single-element set \bar{V}_1 to \mathcal{V}_0 , combine the other $2k - 1$ vertices to one set \bar{V}_2 (we assume in the following $k \geq 2$)

- *Claim:* \mathcal{V}_0 is an odd set cover.

Proof of Claim:

- For each edge $e \in E(G)$, one of the following cases holds:
 1. e is incident to nodes with odd length
 \Rightarrow covered by a node of Type (a)
 2. e is contained in the (shrunk) blossom of G'
 \Rightarrow covered by a node of Type (b)
 3. e is in the rest R of the graph
 \Rightarrow covered by a node of Type (c) $\square C$
- Consider the following assignment $\pi : M \rightarrow \mathcal{V}_0$ of edges $e \in M$ to $V_i \in \mathcal{V}_0$
 - If $e \in M \setminus E(R)$ is incident to a vertex with odd length: assign it to the corresponding set V_i of Type (a)
 - If $e \in M \setminus E(R)$ is part of a blossom: assign it to the corresponding set V_i of Type (b)
 - If $e \in E(R) \cap M$: assign one edge to \bar{V}_1 ; assign the remainder to \bar{V}_2
- Due to the construction:

1. The assignment is well-defined (no matching edge can be assigned to two sets)
 2. $|\pi^{-1}(V_i)| \geq c(V_i)$ for all $V_i \in \mathcal{V}_0$
- $\Rightarrow |M| \geq c(\mathcal{V}_0)$

□

- Using the construction in the proof we can obtain a minimum odd set cover using Edmonds Algorithm

Lemma 111. *The odd set cover problem can be solved in polynomial time.*

- Odd set covers are strongly dual
- However, they are not intuitive to explain someone that no better matching exists

Tutte-Berge-Formula

- One of the first mathematicians to study graph theory in modern times was William Thomas Tutte
- During WW2, he made a brilliant and fundamental advance in crypt analysis helping to decode German codes
- In late 1945, Tutte resumed his studies at Cambridge and provided in his PhD thesis many fundamental results on matchings that are still used today
- One of them, give a characterization of graph that contain a perfect matching
- A necessary condition for a graph to have a perfect matching is that all connected components consist of an even number of nodes
- Is this condition sufficient?
 - If we delete the marked nodes, we have 5 connected components of odd degree in G_3

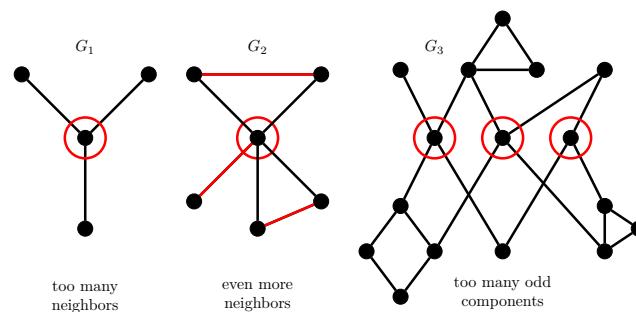


Fig. 7.29.: Sufficient condition (No. 488)

- If a perfect matching exists, at least one vertex of each odd component needs to be connected to one of the marked nodes
- There are only three \Rightarrow no perfect matching can exist

- Notation: Let $X \subseteq V(G)$. Then $q_G(X)$ denotes the *number of odd connected components* in $G - X$.

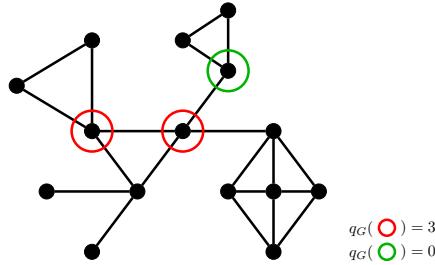


Fig. 7.30.: Number of odd components (No. 489)

- If $q_G(X) > |X|$ for some set $X \subseteq V(G)$, then no perfect matching exists
- Tutte showed that otherwise, i.e., if $q_G(X) \leq |X| \forall X \subseteq V(G)$, there always exists a perfect matching

Definition 112 (Tutte condition, barrier). A graph G satisfies the *Tutte condition* if $q_G(X) \leq |X|$ for all $X \subseteq V(G)$. A vertex set $\emptyset \neq X \subseteq V(G)$ is a *barrier* if $q_G(X) = |X|$. A set X with $q_G(X) > |X|$ is a *Tutte violator*.

- This condition is necessary and sufficient for a graph to have perfect matching

Theorem 113 (Tutte, 1947). *A graph G has a perfect matching if and only if it satisfies the Tutte condition:*

$$q_G(X) \leq |X| \quad \forall X \subseteq V(G).$$

Proof. Construction of Tutte violator

- (\Rightarrow) : Let $G = (V, E)$ be a graph with a perfect matching M

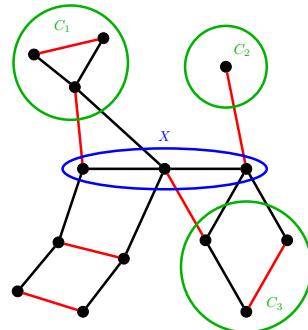
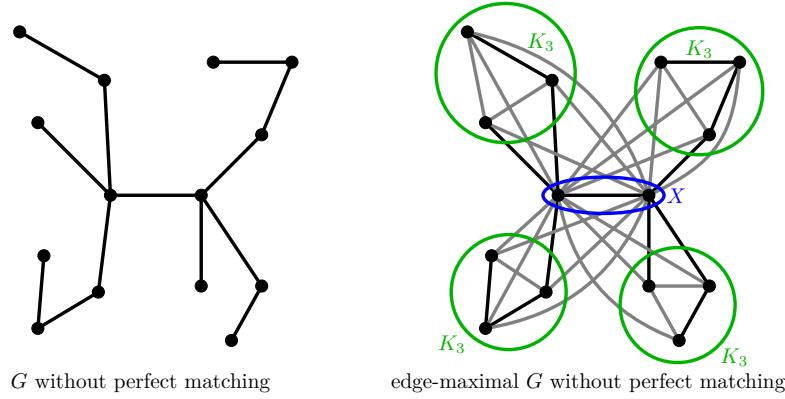


Fig. 7.31.: Perfect \Rightarrow Tutte condition holds true (No. 784)

- Let $X \subseteq V(G)$
- Consider C being an odd component in $G - X$
- Since G has a perfect matching, $\exists uv \in M$ with $u \in X$ and $v \in V(C)$

- Every $u \in X$ is at most connected to one odd component
- $\Rightarrow q_G(X) \leq |X|$
- (\Leftarrow): Let $G = (V, E)$ be a graph without a perfect matching
- Show: there exists a Tutte violator, i.e., $\exists X \subseteq V(G)$ with $q_G(X) > |X|$
- W.l.o.g. G is edge-maximal without a perfect matching

Fig. 7.32.: G is edge-maximal (No. 785)

- Assume G' is obtained from G by adding edges
- Let $X \subseteq V$
- Every odd component of $G' - X$ is composed of at least one odd component of $G - X$
- Every odd component of $G - X$ is at most part of one odd component of $G' - X$
- $\Rightarrow q_G(X) \geq q_{G'}(X)$
- Define $X = \{v \in V(G) \mid d_G(v) = |V| - 1\}$, i.e., v is connected to every node in G
- Case 1: Every connected component in $G - X$ is a complete graph
 - If $q_G(G - X) \leq |X|$, assign to every $v \in X$ one vertex of an odd component
 - $\Rightarrow G$ has a perfect matching unless $|V(G)|$ is odd
 - Since G has no perfect matching, $|V(G)|$ is odd
 - $\Rightarrow \emptyset$ is a Tutte violator
- Case 2: \exists connected component in $G - X$ that is not complete
 - Let K be a component of $G - X$ that is not complete

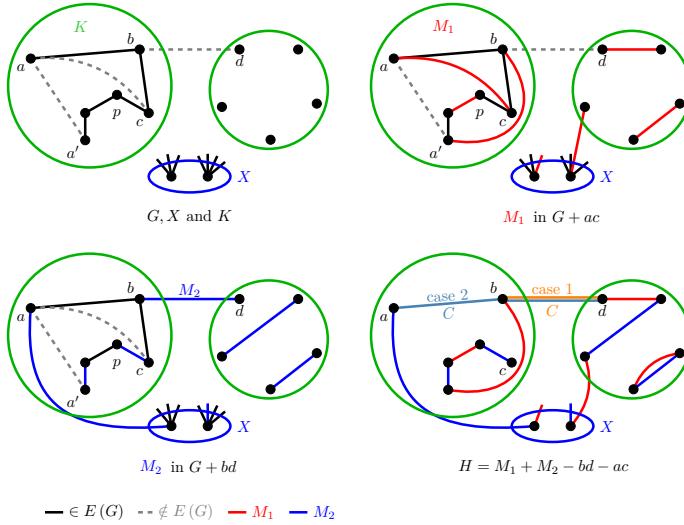


Fig. 7.33.: Construction of perfect matching (No. 786)

- $\Rightarrow \exists a, a' \in V(K)$ with $aa' \notin E(G)$
- Let $p = abc\dots a'$ be a shortest (a, a') -path according to the number of edges in K
- $\Rightarrow ab, ac \in E(G)$ but $ac \notin E$ (otherwise p is no shortest path)
- Since $b \notin X, \exists d \in V(G)$, s.t. $bd \notin E(G)$
- By the maximality of G , \exists a perfect matching M_1 in $G + ac$ and a perfect matching M_2 in $G + bd$
- Consider $H = (V, M_1 \cup M_2 - bd - ac)$
- Then, $\forall v \in V(H) \setminus \{a, c, d, b\}$ we have $d_H(v) = 2$ and $d_H(v) = 1 \forall v \in \{a, c, d, b\}$
- Denote with p_{xy} a path in H connecting x and y
- Due to construction, H contains either p_{db} , p_{da} or p_{dc} , i.e., a path starting in d and ending in K
- We will show, that there exists a cycle C in H
- Case a: \exists a path p_{db}
 - Define $C = p_{db} \cup bd$
 - Due to construction, C is a simple, even cycle
- Case b: \exists a path p_{da} (or p_{dc})
 - Consider $C = p_{da} \cup ab \cup bd$ (or $C = p_{dc} \cup cb \cup bd$)
 - Since $b \notin p_{da}$, C is a simple cycle
 - Furthermore, C is an even cycle since the last edge on p_{da} is an edge in M_2
- Define $M' = \{M_2 \setminus E(C)\} \cup \{E(C) \setminus M_2\} \subseteq E(G)$
- Since M_2 is a perfect matching, M' is a perfect matching in G
- \Rightarrow Contradiction

□

- If a graph G does not have a perfect matching, we can prove this by presenting a set X that violates the Tutte condition
- An important property follows from this Theorem

Lemma 114. For any graph G and any $X \subseteq V(G)$ we have

$$q_G(X) - |X| \equiv |V(G)| \pmod{2}.$$

Proof. Counting argument □

- The Tutte condition provides a good characterization of the perfect matching problem
- For general graphs, Berge and Tutte obtained the following formula

Theorem 115 (Tutte-Berge-Formula, 1958). For a graph $G = (V, E)$ it holds true that

$$\nu(G) = \min_{X \subseteq V} \left\{ \frac{1}{2}(|V| + |X| - q_G(X)) \right\}$$

where $\nu(G)$ denotes to matching number of graph. The matching number $\nu(G)$ is the cardinality of a maximum matching in a graph.

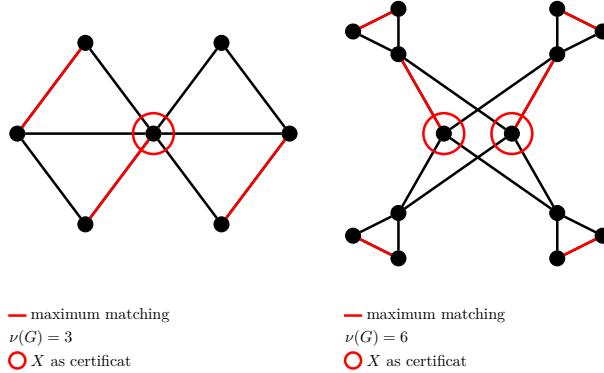


Fig. 7.34.: Tutte-Berge-Formula (No. 491)

Proof. Tutte Violator

- Consider $V(G)$ and a maximum matching M
- Denote with $\text{Cov}(M) = \{v \in V \mid \exists(u, v) \in M\}$ and $\text{UnCov}(M) = \{v \in V \mid \nexists(u, v) \in M\}$
- $\Rightarrow |V(G)| = |\text{Cov}(M)| + |\text{UnCov}(M)|$
- Let $X \subseteq V(G)$ and $q_G(X)$ the number of odd components when X is deleted from G
- $\Rightarrow |\text{UnCov}(M)| \geq q_G(X) - |X|$
- Thus,

$$\begin{aligned} |V(G)| &= |\text{Cov}(M)| + |\text{UnCov}(M)| \\ &= 2 \cdot \nu(G) + |\text{UnCov}(M)| \\ &\geq 2 \cdot \nu(G) + q_G(X) - |X| \end{aligned}$$

- *Claim:* $\exists X \subseteq V(G)$ with

$$2\nu(G) + \max_{X \subseteq V(G)} \{q_G(X) - |X|\} \geq |V(G)|.$$

Proof of claim:

- Let $k := \max_{X \subseteq V(G)} \{q_G(X) - |X|\}$
- Construct a new graph H by adding k vertices to G
- Connect each of the new vertices with all of the old vertices

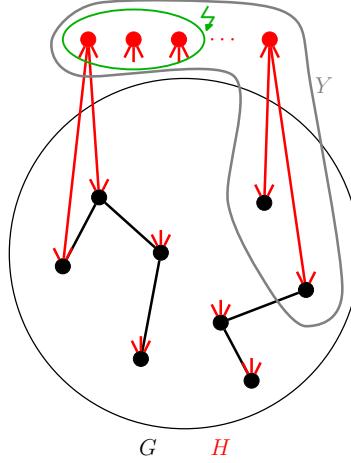


Fig. 7.35.: Construction of H (No. 1174)

- *Claim a:* H has a perfect matching

Proof of claim:

- Assume that H has no perfect matching
- \Rightarrow (Tutte condition, Theorem 113) There is a set $Y \subseteq V(H)$ such that

$$q_H(Y) > |Y| \quad (*)$$

- By Lemma 114, k has the same parity as $|V(G)|$
- Since $|V(H)| = |V(G)| + k$, it follows that $|V(H)|$ is even
- Therefore, H itself is no odd connected component
- To satisfy $(*)$, it must hold $Y \neq \emptyset$ and thus $q_H(Y) > 1$
- $\Rightarrow Y$ contains all new vertices (otherwise, there is only one connected component in $H - Y$)
- So, $q_G(Y \cap V(G)) = q_H(Y) \stackrel{(*)}{>} |Y| = |Y \cap V(G)| + k$
- Contradiction to definition of k \square Ca

- $\Rightarrow H$ has a perfect matching, so $2\nu(H) = |V(H)|$
- Furthermore, $\nu(G) \geq \nu(H) - k$ which is equivalent to $2\nu(G) \geq 2\nu(H) - 2k$
- $\Rightarrow 2\nu(G) + k \geq 2\nu(H) - k = |V(H)| - k = |V(G)|$ \square

\square

- X minimizing $\frac{1}{2}(|V| + |X| - q_G(X))$ is a certificate for the optimality of a matching
- Any set X provides with $\frac{1}{2}(|V| + |X| - q_G(X))$ an upper bound on $\nu(G)$

7.4. Matchings in Bipartite Graphs

- In many applications a matching represents pairs of different things:
 - workers and jobs
 - lectures and time slots in rooms

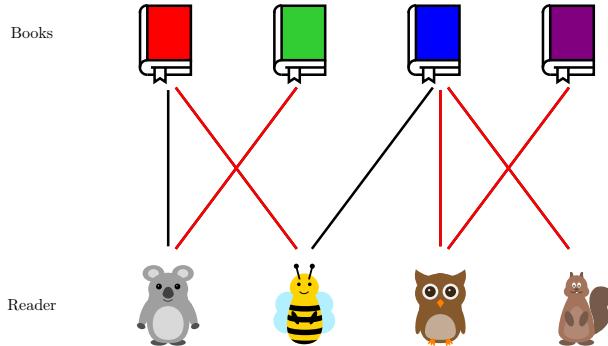


Fig. 7.36.: Bipartite graphs and matchings (No. 819)

- This leads to a bipartite graph
- Optimization problems are often more easy to solve on special graph classes
- In the following, we consider bipartite graphs
- The bipartition of a bipartite graph G is always denoted by A, B
- We will start by considering again the maximum matching and its dual vertex cover problem
- What is the relation between the two?

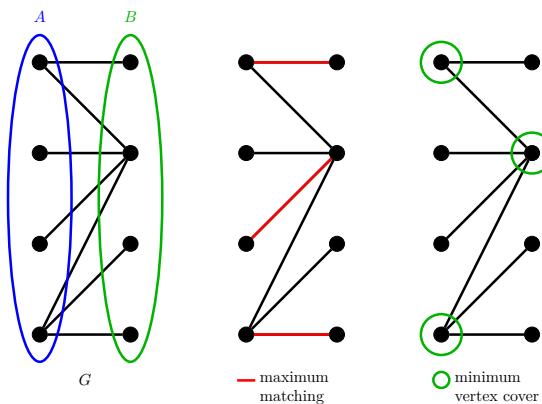


Fig. 7.37.: Bipartite graphs and Theorem of König (No. 492)

- To ease notation, we define
 - the *vertex cover number* $\tau(G)$ as the minimum size of a vertex cover in G
 - the *matching number* $\nu(G)$ is the cardinality of a maximum matching in G .
- Theorem of König relates the vertex cover number with the matching number of a graph

Theorem 116 (König, 1931). *If G is bipartite, then the vertex cover number and the matching number of G are equal, i.e., $\nu(G) = \tau(G)$.*

Proof. Max-Flow-Min-Cut Theorem 67

- Lemma 107: $\nu(G) \leq \tau(G)$
- Prove $\nu(G) \geq \tau(G)$ by
 - constructing a matching M and a vertex cover S
 - such that $|M| \geq |S|$
- Transform $G = (A \cup B, E)$ into a flow network $G' = (V', E')$ by
 - directing all edges from A to B
 - adding nodes s and t to G
 - add edges $(s, v) \forall v \in A$ and $(u, t) \forall u \in B$
 - set upper capacities $u(e) = 1 \forall e \in E'$

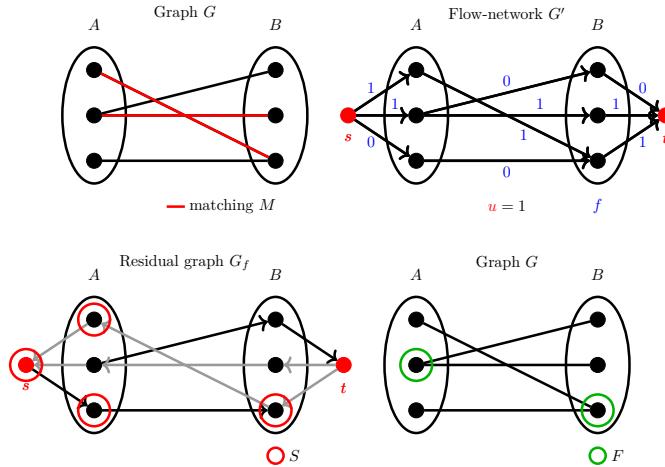


Fig. 7.38.: Matching and flow transformation (No. 493)

- Consider integer maximum (s, t) -flow f in G'
- $\Rightarrow f(e) \in \{0, 1\}$
- \Rightarrow edges from A to B with $f(e) = 1$ define a matching M in G and

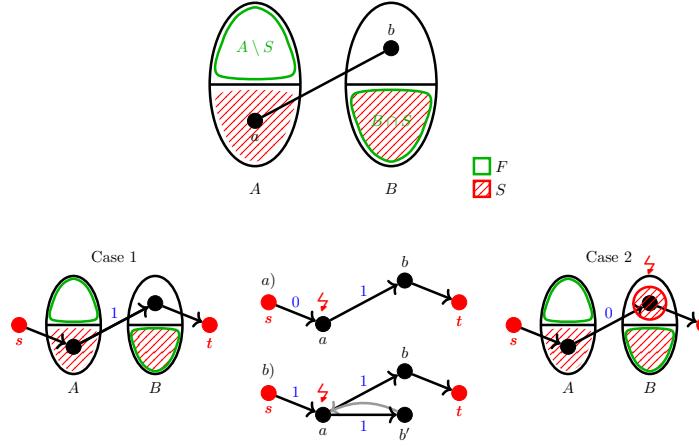
$$\sum_{e \in \delta^+(v)} f(e) = f(s, v) \leq 1 \quad \forall v \in A$$

and

$$\sum_{e \in \delta^-(v)} f(e) = f(v, t) \leq 1 \quad \forall v \in B$$

- Furthermore, $\text{value}(f) = |M|$
- Let $S := \{s\} \cup \{v \in V(G) \mid v \text{ reachable from } s \text{ in } G_f\}$
- $\Rightarrow S$ is cut with minimum capacity

- *Claim:* $F = (A - S) \cup (B \cap S)$ is a vertex cover in G .
- Proof of Claim:*

Fig. 7.39.: F is vertex cover (No. 494)

- Assumption: the edge $e = (a, b)$ with $a \in A$ and $b \in B$ is not incident to F
- $\Rightarrow a \in S$ and $b \notin S$
- *Case 1:* $f(a, b) = 1$
- $a \in S \Rightarrow a$ reachable from s in G_f
- *Case 1.a:* $(s, a) \in G_f$
 - $\Rightarrow f(s, a) = 0 \Rightarrow$ no flow conservation in a , contradiction
- *Case 1.b:* $(b', a) \in G_f$ with $b' \in B$
 - $\Rightarrow (b', a)$ is backward edge and $b' \neq b$ since $b \notin S$
 - $\Rightarrow f(a, b') = 1$
 - Since $f(a, b) = 1$, outflow from a is ≥ 2
 - Contradiction to only one incoming edge, i.e., capacity 1
- *Case 2:* $f(a, b) = 0$
 - $\Rightarrow (a, b)$ is in G_f
 - Since $a \in S \Rightarrow b$ is reachable from s in $G_f \Rightarrow b \in S$, contradiction $\square C$
- Furthermore,

$$\begin{aligned} \text{cap}(S) &= \sum_{a \in A - S} u(s, a) + \sum_{b \in B \cap S} u(b, t) + \underbrace{\sum_{\substack{(a, b) \in A(G') \\ a \in A \cap S, b \in B - S}} u(a, b)}_{=0} \\ &= |A - S| + |B \cap S| \end{aligned}$$

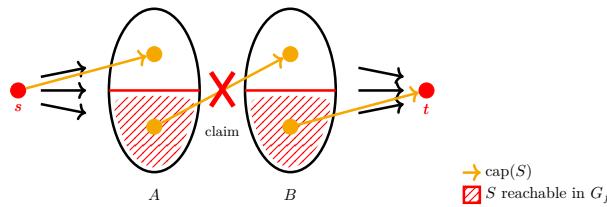


Fig. 7.40.: Capacity of the cut (No. 495)

$$\Rightarrow \tau(G) \leq |F| = \text{cap}(S) = \text{value}(f) = |M| \leq \nu(G)$$

- \Rightarrow in summary: $\nu(G) = \tau(G)$

□

- There are many equivalent versions of the Theorem by König
- Nice consequence of the proof: an algorithm to construct a maximum matching

Theorem 117. *A maximum matching and a minimum vertex cover in a bipartite graph can be calculated in $\mathcal{O}(nm)$.*

Proof. Max-Flow Algorithm

- Apply max-flow algorithm to the transformed graph
- Maximum flow value equals the matching number $\nu(G)$
- Edges $(a, b) \in G$ with $f(a, b) = 1$ form a matching
- Each augmentation in $\mathcal{O}(m)$ and at most $|A|$ augmentations $\Rightarrow \mathcal{O}(nm)$
- Let $S := \{s\} \cup \{v \in V(G) \mid v \text{ reachable from } s \text{ in } G_f\}$
- Then, $F = (A - S) \cup (B \cap S)$ is a minimum vertex cover in G

□

- Possible speed-up to $\mathcal{O}(n^{\frac{1}{2}}(m+n))$ by augmentation along shortest paths (Hopcroft & Karp 1973) - as for Edmonds-Karp algorithm for maximum (s, t) -flows

Theorem of Hall

- If no perfect matching exists, can we somehow decide why this is the case?

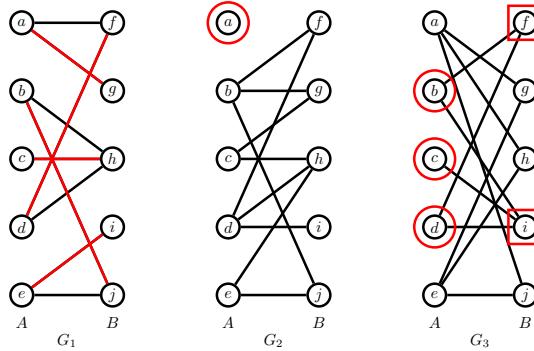


Fig. 7.41.: Why haven't we a perfect matching? (No. 511)

- Theorem of Hall: nice criterion, when a set of vertices is not “connected” enough

Theorem 118 (Hall 1935). *Let $G = (A \cup B, E)$ be bipartite. For any $X \subseteq A$ define*

$$N(X) := \{y \in B \mid \text{there exists an } x \in X \text{ with } xy \in E(G)\}.$$

Then the following statements are equivalent:

1. G has a matching M that covers A , i.e. every node in A is incident to an edge in M .
2. For every subset $X \subseteq A$ there are sufficiently many “partners” in B i.e. $|N(X)| \geq |X|$ for all $X \subseteq A$ [Hall’s condition].

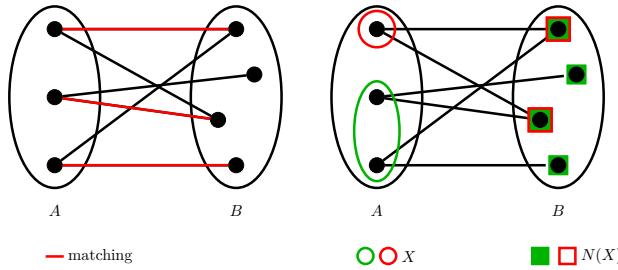


Fig. 7.42.: Neighbors (No. 497)

Proof. Properties of bipartite graphs and Theorem of König

- $((1) \Rightarrow (2))$: straightforward
- $((2) \Rightarrow (1))$: $\forall X \subseteq A$, we have $|N(X)| \geq |X|$
- Assumption: no matching exists that covers A
- $\Rightarrow \nu(G) < |A|$
- Theorem of König 116: $\tau(G) = \nu(G) < |A|$
- Let $A' \subseteq A$, $B' \subseteq B$, such that $A' \cup B'$ is a vertex cover of minimum cardinality

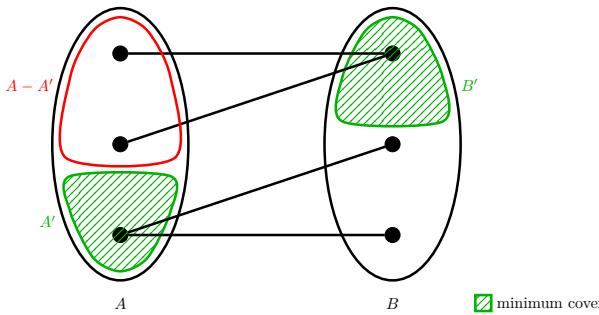


Fig. 7.43.: Minimum cover (No. 498)

- $\Rightarrow |A'| + |B'| = |A' \cup B'| = \tau(G) < |A| \Leftrightarrow |B'| < |A| - |A'|$.
- Consider, $A - A'$
- Any edge ab with $a \in A - A'$ needs to be covered by $b \in B'$
- $\Rightarrow N(A - A') \subseteq B'$
- $\Rightarrow |N(A - A')| \leq |B'| < |A| - |A'| = |A - A'|$
- Contradiction to $|N(A - A')| \geq |A - A'|$

□

- The special case $|A| = |B|$ of Hall's Theorem was already known to Frobenius in 1917
- According to Hall's Theorem, it is easy to obtain a certificate if no A covering matching exists:
 - Just compute a set $W \subseteq A$ with $|N(W)| < |W|$
 - A set $W \subseteq A$ with $|N(W)| < |W|$ is called *Hall Violator*.

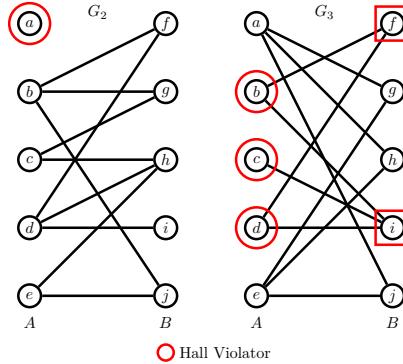


Fig. 7.44.: Hall Violator (No. 512)

- Let M be a maximum matching, that does not cover A
- How can we compute a Hall Violator?

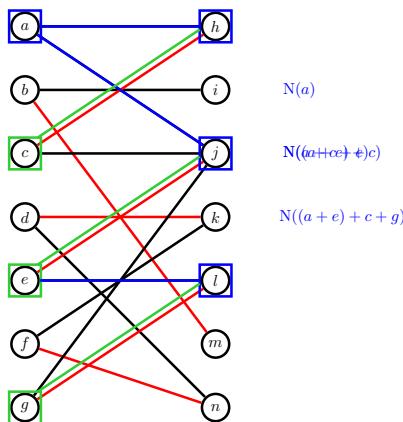


Fig. 7.45.: Computing a Hall Violator (No. 818)

- Observation: Every Hall Violater needs to contain an exposed node $a \in A$
- Idea:
 - Chose a and consider all neighbours $N(a)$ of a
 - These neighbours have neighbours in A
 - \Rightarrow add them to the prospective Hall Violater
 - Proceed until all M -reachable nodes are considered
 - \Rightarrow this set is a Hall Violator
- Let M be a matching and $a \in A$. A node $w \in A \cup B$ is called *M -reachable* (from a), if there exists an M -alternating path from a to w
- Using this, we obtain a nice algorithm

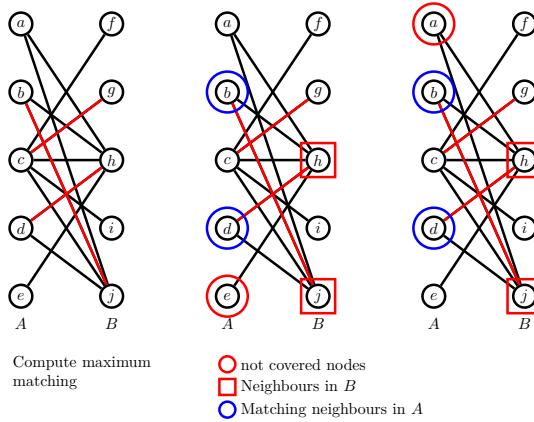
Algorithm 7.2 Hall Violator AlgorithmInput: Bipartite graph $G = (A \cup B, E)$ Output: Hall Violator W Step 1: Compute a maximum matching M in G Step 2: If $|M| = |A|$, **Return**: no Hall Violator existsStep 3: If $|M| < |A|$, chose $a \in A$ that is not covered by M Step 4: Compute all nodes S with BFS, that are M -reachable from a
Return: $W = S \cap A$ as Hall Violator

Fig. 7.46.: Finding a Hall Violator (No. 513)

- Remains to prove that W really is a Hall Violator

Theorem 119. *The Hall Violator Algorithm proves, that an A covering matching exists, or computes a Hall Violator.*

Proof. Definition of M -reachable

- Let a be the exposed node, with which W was computed in the Hall Violator Algorithm

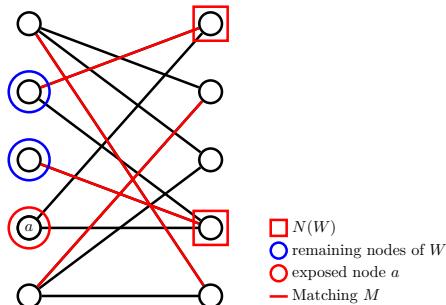


Fig. 7.47.: Algorithm correct (No. 514)

- *Claim:* All nodes $N(W)$ are matched with M

Proof of Claim:

- Assume, \exists an exposed node $y \in N(W)$
- Since y is M -reachable from a , an M -alternating path from a to y exists
- This path starts with a non-matching edge and ends with a non-matching edge
- Construct a new matching M' by using all non-matching edges of the path instead of the matching edges of the path
- $|M'| = |M| + 1$, contradiction to Step 1 $\square C$
- By construction, W contains all matching partners $M(N(W))$ of $N(W)$
- $\Rightarrow |W| = |\{a\}| + |M(N(W))| = |N(W)| + 1 > |N(W)|$
- $\Rightarrow W$ is a Hall Violator \square

- If no A covering matching exists, we can easily provide a nice certificate, e.g., a violator, for it

7.5. Postperson Problem

- Start with a classical real world problem

Postperson Problem (informal)

Given: a street network and a set of streets where letters need to be delivered to every house

Aim: find a tour that traverses every street in order to deliver all letters with minimum length

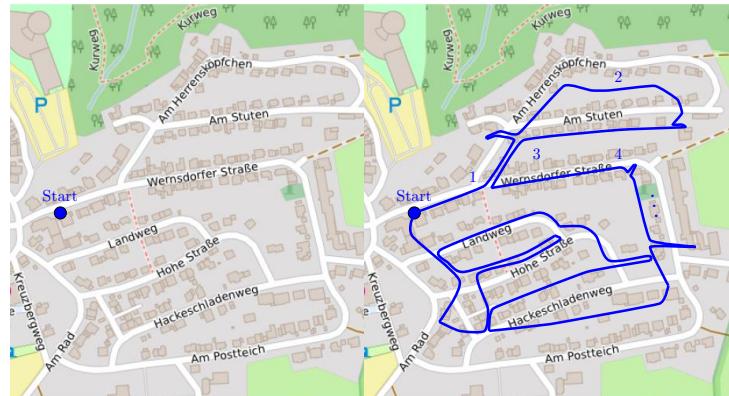


Fig. 7.48.: The postperson problem (No. 794)

- The postperson problem is named after the Chinese mathematician Guan Meigu, who studied the problem first and is often called the Chinese postman problem or “Briefträgerproblem”
- How can we formulate this problem in a mathematical way?
- Transform the street network into a graph
 - vertices represent crossings

- edges represent streets connecting the crossings

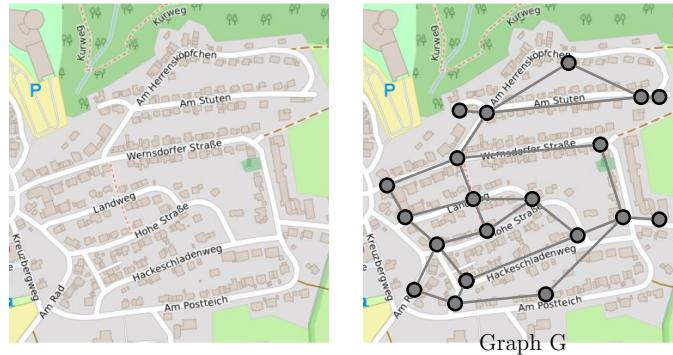


Fig. 7.49.: Maps to graphs (No. 795)

Definition 120. Postperson Problem

Given: Undirected graph $G = (V, E)$, edge weights $c(e) \geq 0$

Aim: Find a cycle C that runs along each edge at least once and has minimum length $c(C)$ with

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

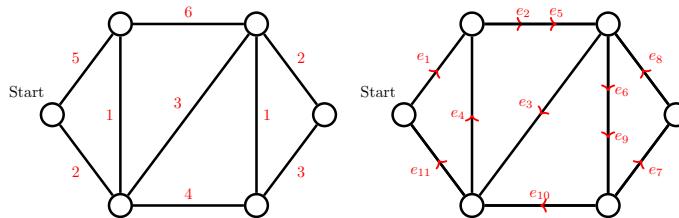


Fig. 7.50.: The Postperson Problem (No. 796)

- Note, $E(C)$ is a multi-set. Edges that occur several times are counted several times
- Question: How can we compute an optimal tour?

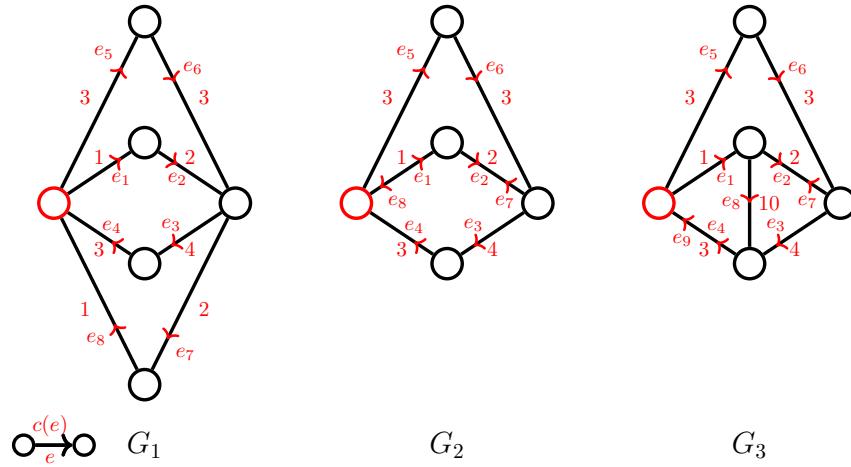


Fig. 7.51.: Sometimes it is easy to find the optimal solution (No. 797)

- **Best Option:** We don't need to traverse any edge twice (Eulerian Problem)

- **General Case:** We need to traverse some edges twice

- If we know that we need to get from one crossing to another, how do we get from one point to the other? (Shortest path problem)
- How do we pair crossings that need to be connected by extra paths? (Minimum perfect matching problem)

Definition 121. Minimum Perfect Matching Problem

Given: Undirected graph G , edge weights $c(e)$

Aim: Find a perfect matching M with minimum weight $c(M)$, where

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

- The minimum perfect matching problem can be solved in polynomial time by a combinatorial algorithm that uses the unweighted Edmond's algorithm as a subroutine
- Using these ideas and insights, we obtain the following algorithm

Algo. 7.1 Postperson Algorithm

Input: Undirected, connected graph G , edge weights $c(e) \geq 0$

Output: A tour of minimum length, using each edge at least once

Method:

Step 1: If G is Eulerian, compute an Euler-Tour. **Return:** Euler-Tour

Step 2: Determine all nodes with odd degree V_{odd} .

Step 3: Consider the complete graph H on V_{odd} and add weights $w(e) = \text{dist}(v, w)$ to all edges $e = vw \in E(H)$. Here, $\text{dist}(v, w)$ denotes the shortest path in G from v to w

Step 4: Compute a minimum perfect matching M in H .

Step 5: Add the corresponding shortest paths of all matching edges to G .

Step 6: Compute an Euler-Tour. **Return:** Euler-Tour

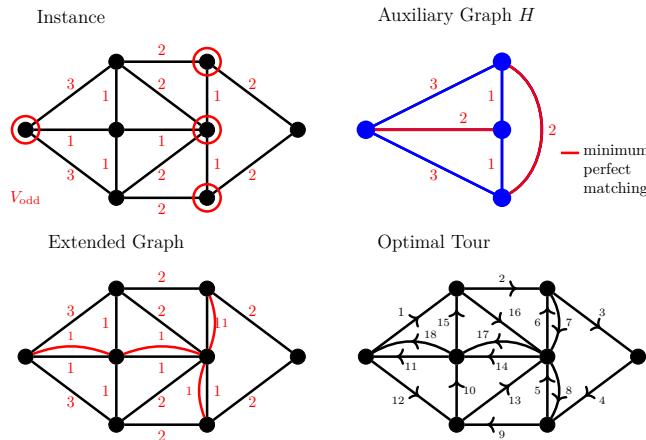


Fig. 7.52.: Postperson Algorithm (No. 578)

Theorem 122. *The Postperson Algorithm 7.1 computes an optimal solution.*

Proof. Exercise □



8. Linear Programming

- Linear programming is a very powerful modeling tool
- Linear (and integer) optimization have become indispensable in practice
- “From an economical point of view, linear optimization has been the most important mathematical development in the 20th century” (Prof. Dr. Martin Grötschel)

Literature

- Chvátal
Linear Programming. W.H. Freeman & Co., 1983
- Bertsimas, Tsitsiklis
Introduction to Linear Optimization. Athena Scientific, 1997

8.1. Stigler's Diet Problem and Basics

Stigler's Diet Problem

- 1947 invented Georg Dantzig the simplex algorithm to solve linear programmes
- When he was looking for a real world problem, he found an article published by Georg Stigler
- In 1945, George J. Stigler of the University of Minnesota published the paper “The Cost of Subsistence” in the Journal of Farm Economics, Vol. 27.
- *Stigler's Diet Problem*

Given:

- List of food with their nutritional values and costs
- Necessary nutritional intake of an adult for a healthy diet

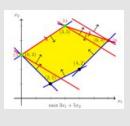
Find:

- A composition of food items and their quantities so that sufficient nutritional values are ingested and costs are minimized.

- How do we model the problem?



$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$



George B. Dantzig



THE COST OF SUBSISTENCE

GEORGE J. STIGLER
University of Minnesota

- This paper is organized under five headings, devoted to
1. The quantities of the various nutrients which should be contained in an average person's diet.
 2. The quantities of these nutrients which are found in certain common foods.
 3. The methodology of finding the minimum cost diet.
 4. The minimum cost diet in August 1939 and August 1944.
 5. Comparison with conventional low-cost diets.

TABLE A. NUTRITIONAL VALUES OF COMMON FOODS PER DOLLAR OF EXPENDITURE, AUGUST 15, 1939												
Commodity	Unit	Aug. 15, 1939 (cents)	Edible per \$1.00 (1,000)	Calories (grams)	Protein (grams)	Calcium (grams)	Iron (mg.)	Vitamin A (U.S.)	Vitamin C (mg.)	Riboflavin (mg.)	Niacin (mg.)	Ascorbic (mg.)
1. Wheat Flour (Enriched)	16 lb.	50.0	12,400	44.7	1,413	4.5	961	41.4	50.3	441	—	—
2. Wheat Flour (Unenriched)	9 lb.	52.1	12,294	41.4	1,351	4.1	735	38.2	51.5	124	—	—
3. White Bread	1 lb.	12.5	1,250	11.4	352	2.7	127	16.2	8.5	12.5	—	—
4. White Bread (Enriched)	1 lb.	12.5	1,250	11.4	352	2.7	127	16.2	8.5	12.5	—	—
5. Hamburger	1 lb.	5.5	5,000	49.6	280	4.5	86	50.9	16.5	1.6	110	—
6. Hamburger (Enriched)	1 lb.	5.5	5,000	49.6	280	4.5	86	50.9	16.5	1.6	110	—
7. Hamburger (Bacon)	1 lb.	5.5	5,000	49.6	280	4.5	86	50.9	16.5	1.6	110	—
8. White Bread (Enriched)	1 lb.	12.5	1,250	11.4	352	2.7	127	16.2	8.5	12.5	—	—
9. White Bread (Unenriched)	1 lb.	12.5	1,250	11.4	352	2.7	127	16.2	8.5	12.5	—	—
10. Eggs (dozen)	1 lb.	12.5	4,800	12.4	380	1.1	86	18.9	5.5	3.0	55	—
11. Eggs (dozen)	1 lb.	12.5	4,800	12.4	380	1.1	86	18.9	5.5	3.0	55	—
12. Eggs (dozen)	1 lb.	12.5	4,800	12.4	380	1.1	86	18.9	5.5	3.0	55	—
13. Eggs (dozen)	1 lb.	12.5	4,800	12.4	380	1.1	86	18.9	5.5	3.0	55	—
14. Eggs (dozen)	1 lb.	12.5	4,800	12.4	380	1.1	86	18.9	5.5	3.0	55	—
15. Milk (skim)	1 lb.	12.5	5,000	12.1	310	16.2	15	16.5	4.0	18.0	1.7	177
16. Milk (whole)	1 lb.	12.5	5,000	12.1	310	16.2	15	16.5	4.0	18.0	1.7	177
17. Milk (buttermilk)	1 lb.	12.5	5,000	12.1	310	16.2	15	16.5	4.0	18.0	1.7	177
18. Milk (cheese)	1 lb.	12.5	5,000	12.1	310	16.2	15	16.5	4.0	18.0	1.7	177
19. Butter	1 lb.	12.5	5,000	12.1	310	16.2	15	16.5	4.0	18.0	1.7	177
20. Margarine	1 lb.	12.5	5,000	12.1	310	16.2	15	16.5	4.0	18.0	1.7	177

TABLE 1. DAILY REQUIREMENTS OF NUTRIENTS FOR A MINIMUM-COST ANNUAL DIET (MEANING 100 PERSONS)			
Nutrient	Allowance	Quantity	Cost
Calories	3,000	calories	
Carbohydrates	120 g.	g. grams	
Protein	50 g.	g. grams	
Vitamin A	2,000	International Units	
Vitamin B	—	—	
Thiamine (B ₁)	—	—	
Niacin (B ₃)	—	—	
Riboflavin (B ₂)	—	—	
Ascorbic Acid (C)	—	—	

* National Research Council, Recommended Dietary Allowances, Reprint and Circular Series No. 114, January, 1944.

TABLE 2. MINIMUM COST ANNUAL DIETS, AUGUST 1939 AND 1944				
Commodity	August 1939		August 1944	
	Quantity	Cost	Quantity	Cost
Wheat Flour	370 lb.	\$13.38	535 lb.	\$34.55
Evaporated Milk	57 cans	3.34	—	—
Cabbage	111 lb.	4.11	107 lb.	5.28
Spinach	23 lb.	1.85	18 lb.	1.56
Dried Navy Beans	285 lb.	16.80	—	—
Pancake Flour	—	—	134 lb.	18.08
Pork Liver	—	—	25 lb.	5.48
Total Cost		\$89.93		\$59.88

Fig. 8.1.: Nutritional table based on values from 1937. With the help of a heuristic, Stigler calculates a diet for 39,93\$ per annum (Prices from 1937!). (No. 674)

Simplified Example

- Given:

- Nutritional values + prices: (all values in 100g)

Food item	Fat	Carbohydrates	Proteins	Calories	Price
Apple	0.4	11.0	0.3	50	0.30
Peanut butter	49.9	12.7	27.1	627	0.82
Skim milk	0.3	5.7	4.2	42	0.07
Wholegrain bread	1.0	37.2	6.9	188	0.29

- Criteria for a healthy diet

- No less than 2000 calories per day
- No more than 20g of fat per day
- At least $\frac{1}{3}$ of the intake consists of fruit and vegetables

- Find: a minimum-cost diet plan

- A model:

1. Variables: What is to be decided?

- For every food item, define one variable which indicates how much of it is to be eaten:

$$x_A \geq 0, x_E \geq 0, x_M \geq 0, x_V \geq 0$$

x_A amount of apples in 100g, x_E amount of peanut butter in 100g, x_M amount of skim milk in 100g, x_V amount of wholegrain bread in 100g

- e.g.: $x_A = 3, x_E = 0.6, x_M = 1.5, x_V = 4$ means: the diet should consist of 300g of apples, 60g of peanut butter, 150g of skim milk, 400g of wholegrain bread

2. Objective function: Minimize the total costs

- The costs depend on the amount x_A, \dots, x_V :

$$c(x_A, \dots, x_V) = 0.3x_A + 0.82x_E + 0.07x_M + 0.29x_V$$

- e.g.: for $x_A = 3, x_E = 0.6, x_M = 1.5, x_V = 4$ costs arise of

$$\begin{aligned} c(3, 0.6, 1.5, 4) &= 0.3 \cdot 3 + 0.82 \cdot 0.6 + 0.07 \cdot 1.5 + 0.29 \cdot 4 \\ &= 7.772 \end{aligned}$$

3. Restriction: What constraints must be met?

- Sufficient calories:

$$50x_A + 627x_E + 42x_M + 188x_V \geq 2000$$

- Not too much fat:

$$0.4x_A + 49.9x_E + 0.3x_M + 1.0x_V \leq 20$$

- Lots of fruit:

$$x_A \geq \frac{1}{3}(x_A + x_E + x_M + x_V)$$

- e.g.: for $x_A = 3, x_E = 0.6, x_M = 1.5, x_V = 4$ it holds

$$50 \cdot 3 + 627 \cdot 0.6 + 42 \cdot 1.5 + 188 \cdot 4 = 1514.3 < 2000$$

\Rightarrow this solution/diet does not provide enough calories.

- In summary, we want to solve:

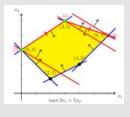
$$\begin{array}{llllllll} \min & 0.3x_A & + & 0.82x_E & + & 0.07x_M & + & 0.29x_V \\ \text{s.t.} & 50x_A & + & 627x_E & + & 42x_M & + & 188x_V \geq 2000 \\ & 0.4x_A & + & 49.9x_E & + & 0.3x_M & + & 1.0x_V \leq 20 \\ & \frac{2}{3}x_A & - & \frac{1}{3}x_E & - & \frac{1}{3}x_M & - & \frac{1}{3}x_V \geq 0 \\ & x_A, & x_E, & x_M, & x_V & \geq 0 \end{array}$$

- *s.t.:* “subject to”; from here on, the constraints are listed
- *min:* followed by the objective function that we want to minimize

- The minimum cost solution is $x_A = 4.501, x_E = 0.188, x_V = 8.814, x_M = 0$ (i.e. the minimum cost diet plan consists of 450g of apples, 18.8g of peanut butter, 881.4g of wholegrain bread); it costs 4.11\$, provides 2000 calories, contains 19.82g of fat and has a fruit content of $\frac{1}{3}$
- In the earliest 1950s, Georg Dantzig moved to Santa Monica
- His doctor advised him to go on a diet to lose some weight
- Dantzig decided to use his research to find a good diet
- The results were questionable at the beginning



$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$



Dantzig's personal diet plan

Day 1 "Not exactly," I replied, "AND 500 gallons of vinegar." She thought it funny and laughed.

Viniger is no food

Day 2 time to prepare supper. Again the diet seemed to be plausible except for calling for the consumption of 200 bouillon cubes per day. Anne made one of the great puns

Maximum of 3 cubes

Day 3+ The next day the above scene was repeated, except this time the diet called, among other things, for two pounds of bran per day. Anne said, "If you consume

exactly the same except this time it was two pounds of blackstrap molasses which substituted for the bran; apparently their nutritional contents were quite similar.

Further constraints



THE COST OF SUBSISTENCE

GEORGE J. STIGLER
University of Minnesota

This paper is organized under five headings, devoted to
 1. The quantities of the various nutrients which should be contained in an average person's diet.
 2. The quantities of these nutrients which are found in certain common foods.
 3. The methodology of finding the minimum cost diet.
 4. The minimum cost diet in August 1939 and August 1944.
 5. Comparison with conventional low-cost diets.

Fig. 8.2.: Dantzig's personal diet (No. 825)

1

Guidelines for Modeling Linear Optimization Problems

- For modeling an optimization problem as a linear program, one has to do the following:

Step 1: Identify decision variables <ul style="list-style-type: none"> What decisions have to be made? How are these represented best? 	Step 2: Identify constraints <ul style="list-style-type: none"> Which relations limit the solution space? How are these relations formulated in decision variables?
Step 3: Identify objective function <ul style="list-style-type: none"> Which quantity do we want to optimize? In which direction do we optimize? [maximization or minimization] 	Step 4: Identify parameters <ul style="list-style-type: none"> Which values are predefined and relevant?

- Sometimes, as in the optimization cycle, these steps need several iterations, e.g., if some parameters are not known, new variables need to be specified

Linear Programs in General

- Such a model is known as a *linear program* (LP)
- "Program" is a historic term and has nothing to do with programming a computer
- Linear programs can appear in different forms
 - As a minimization or maximization problem
 - Constraints restrict by $\leq, \geq, =$
e.g.: $x_1 + x_2 \geq 3, x_1 + x_2 \leq 25$ or $x_1 + x_2 = 10$

¹In his paper "The Diet Problem" George Dantzig describes how he came across the diet problem and used it for his own diet plan. The description clearly shows how model and practice can diverge.

- With positive, negative or arbitrary (i.e. not further restricted) variables
e.g.: $x \geq 0, x \leq 0, x \in \mathbb{R}$
- Therefore, some conventions make life much easier

Definition 123 (Canonical Form of LPs). The *canonical form of a linear program* is given by:

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

with the *objective function coefficients* $c \in \mathbb{R}^n$, the *coefficient matrix* $A \in \mathbb{R}^{m \times n}$, the *right-hand side* $b \in \mathbb{R}^m$, and the *variables* $x \in \mathbb{R}^n$.

- Example:** detailed and compact representation

In detail

Compact

$$\begin{array}{ll} \max & x_1 + 2x_2 \\ \text{s.t.} & x_1 + x_2 \leq 3 \\ & 2x_2 \leq 4 \\ & 4x_1 - x_2 \leq 8 \\ & x_1, x_2 \geq 0 \end{array} \quad \begin{array}{ll} \max & (1, 2)^\top x \\ \text{s.t.} & \begin{pmatrix} 1 & 1 \\ 0 & 2 \\ 4 & -1 \end{pmatrix} x \leq \begin{pmatrix} 3 \\ 4 \\ 8 \end{pmatrix} \\ & x \geq 0 \end{array}$$

- A solution that satisfies all constraints is called feasible

Definition 124 (feasible solution, optimal solution). Let $\max\{c^\top x \mid Ax \leq b, x \geq 0\}$ be a linear program. A vector $x \in \mathbb{R}^n$ is *feasible* if $x \in P$, i.e., $Ax \leq b$ and $x \geq 0$. A feasible solution x^* is called *optimal* for a maximization problem if there is no $x' \in P$ with $c^\top x' > c^\top x^*$.

- We will later see how such optimal solutions can be found

Geometrical Interpretation

- So far, we interpreted an LP in an algebraic version

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

- The set of feasible solutions is given by $\{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$
- What if we see every constraint as a hyperplane in \mathbb{R}^n defining a feasible region
- A *hyperplane* in \mathbb{R}^n is a set of solutions of an equation $a_1x_1 + \dots + a_nx_n = b$, $a_1, \dots, a_n, b \in \mathbb{R}$ (not all $a_j = 0$)
- A hyperplane defines two (closed) halfspaces

$$H^+ = \{x \mid a_1x_1 + \dots + a_nx_n \geq b\} \text{ and } H^- = \{x \mid a_1x_1 + \dots + a_nx_n \leq b\}$$

- Then, the set of feasible solutions equals the intersection of hyperplanes $a_i^\top x \leq b_i$

- In geometry, the intersection of hyperplanes in \mathbb{R}^n define a polyhedron:
 - A *polyhedron* in \mathbb{R}^n is the intersection of finitely many halfspaces (generated by hyperplanes)
 - A *polytope* is a bounded polyhedron
- **Example:** Consider a linear program with two variables x_1 and x_2

$$\begin{array}{lllll} \max & 2x_1 & + & x_2 & \\ \text{s.t.} & x_1 & + & x_2 & \leq 3 \\ & & & 2x_2 & \leq 4 \\ & 4x_1 & - & x_2 & \leq 8 \\ & x_1, & x_2 & \geq 0 & \end{array}$$

- How can we solve this problem graphically?
 - Variables represent the x - and y -axis, i.e. the x -axis represents the x_1 -values of a solution and the y -axis the x_2 -values of a solution
 - Each constraint “limits” the range of feasible solutions
 - Consider $x_1 + x_2 \leq 3$.
 - Draw the straight line with $x_2 = 3 - x_1$
 - All points below the straight line fulfill the condition $x_1 + x_2 \leq 3$ (e.g.: the point $(1, 1)$)
 - Draw all conditions in this way
 - The domain in which the points that meet all the conditions lie, is the domain in which our solution lies
 - Mapping of the objective function $2x_1 + x_2$
 - Consider the straight line $x_2 = c - 2x_1$ with the gradient -2 e.g. $x_2 = 2 - 2x_1$ for $c = 2$.
 - All points (x_1, x_2) on the straight line have the objective function value c (in our case a value of 2)
 - $x_2 = 2 - 2x_1 \Rightarrow 2x_1 + x_2 = 2x_1 + 2 - 2x_1 = 2$.
 - Shifting the straight line “upwards” increases the objective function value
 - \Rightarrow move the straight line until it only just cuts a feasible point.
 - The point corresponds to an optimal solution

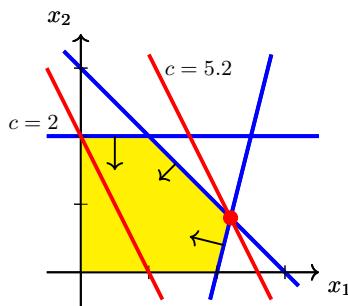


Fig. 8.3.: All feasible points are within the marked region. The optimal solution (drawn in red) is obtained by slowly moving the objective function upwards. (No. 691)

- The same principle applies to multiple variables but is difficult to draw

Optimal Solutions for Linear Programs via Geometry

- We start with an LP $\max\{c^\top x \mid Ax \leq b, x \geq 0\}$
- Consider the *polyhedron* given by $P = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$. Then, P is the *feasible region* of the LP
- A polyhedron is either *bounded*, *unbounded* or *inconsistent*

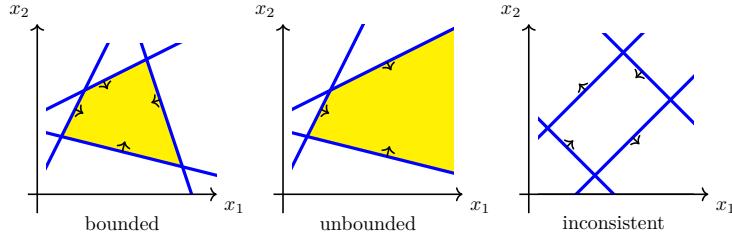


Fig. 8.4.: Polyhedron can be divided into three groups: bounded, unbounded and inconsistent. (No. 693)

- This motivates the following lemma:

Lemma 125. Consider a linear program given by $P := \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$ and a cost function $c : \mathbb{R}^n \rightarrow \mathbb{R}$. For every LP, exactly one of the alternatives applies

1. The LP is infeasible, i.e., $P = \emptyset$.
2. The objective function is unbounded.
3. A finite optimal solution exists.

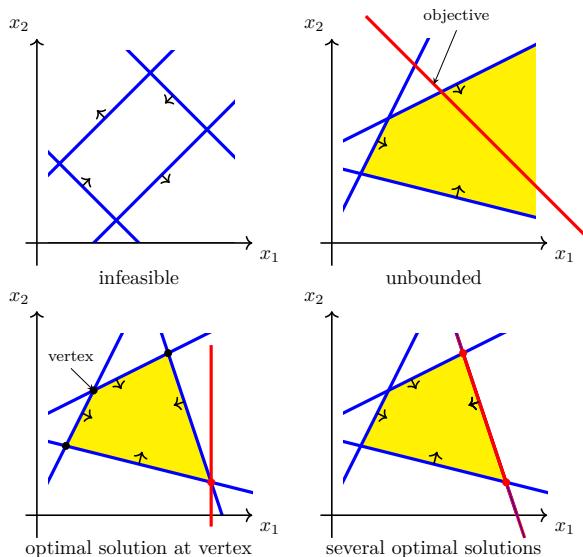


Fig. 8.5.: A linear optimization problem either has no, one, or infinitely many optimal solutions. (No. 694)

Proof. Exercise □

- An important observation (by picture): if an optimal solution exists, then one at a vertex of the polytop
- Let us assume in the following, that the considered polyhedron P is bounded, i.e., we have a polytope
- A vector $x \in P \subseteq \mathbb{R}^n$ is a *vertex* or an *extreme point* of P if there exists no $y \in \mathbb{R}^n$ with $x + y \in P$ and $x - y \in P$
- An equivalent definition is that x is a vertex if x is no convex combination of the other points in P
- A vector $x \in P$ is a *convex combination* of $x^1, \dots, x^p \in P$ if $x = \sum_{i=1}^p \lambda_i x^i$ with $\sum_{i=1}^p \lambda_i = 1$
- Roughly speaking, the *vertices* are special points of the polytope that are located at the “boundary” of P , where the sides intersect

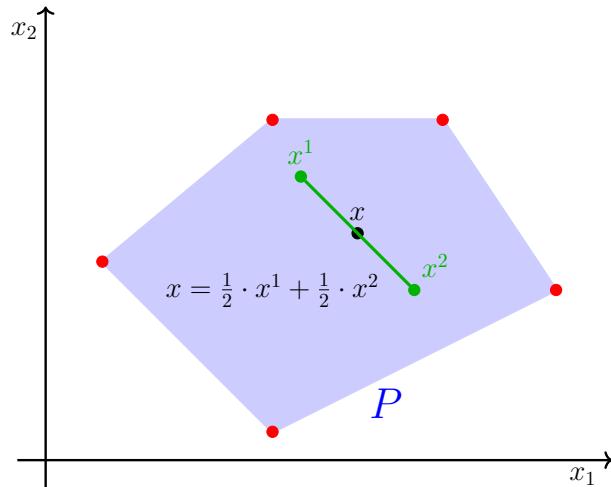


Fig. 8.6.: polytope, convex combination, and vertices (No. 697)

Theorem 126 (Minkowski, 1896). *Every polytope is the convex hull of its vertices, i.e., every point x of a polytope P can be represented as convex combination of a finite number of vertices.*

- The following consequence is essential to obtain an optimality criterion

Lemma 127. *Let $\max\{c^\top x \mid Ax \leq b, x \geq 0\}$ be an LP. If there exists an optimal solution \bar{x} for $c \in \mathbb{R}^n$, then there exists a vertex $x^* \in P := \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$ with $c^\top \bar{x} = c^\top x^*$.*

Proof. Contradiction to optimality

- Let $\bar{x} \in P$ be an optimal solution
- Assume all vertices are not optimal
- $\Rightarrow \bar{x}$ is a non-vertex
- $\Rightarrow \exists$ vertices $x^1, \dots, x^k \in P$ with $\lambda_i \geq 0$, $i = 1, \dots, k$ and $\sum_{i=1}^k \lambda_i = 1$ s.t.

$$\bar{x} = \sum_{i=1}^k \lambda_i x^i$$

- Due to the assumption, $c^\top x^i < c^\top \bar{x}$
- However,

$$\begin{aligned} c^\top \bar{x} &= c^\top \left(\sum_{i=1}^k \lambda_i x^i \right) = \sum_{i=1}^k \lambda_i (c^\top x^i) < \sum_{i=1}^k \lambda_i (c^\top \bar{x}) \\ &= c^\top \bar{x} \sum_{i=1}^k \lambda_i = c^\top \bar{x} \end{aligned}$$

- Contradiction. Hence, there must exist an optimal vertex

□

- Furthermore, the number of vertices of a polytope is bounded
- First idea for an algorithm: “Brute force” (not practical)
 - Consider all vertices
 - Return vertex with best objective value

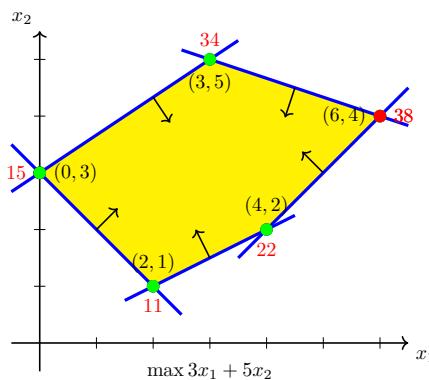


Fig. 8.7.: Considering all vertices usually takes too long. (No. 695)

- Better: search the vertices systematically via a walk along the border of P
 - find a vertex of P
 - go from one vertex to the next
 - s.t. the objective function value does not worsen

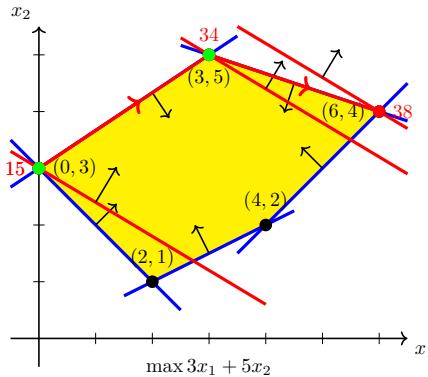


Fig. 8.8.: Instead of visiting all vertices, we only look at those that improve our objective function. (No. 675)

- In order to pursue this idea, we need to:
 1. Represent the vertices algebraically
 2. Define how to walk through the polytope algebraically
 3. Show that there is a path along the border between every vertex x and an optimal vertex x^* such that the objective function does not get worse

8.2. Algebraic Interpretation of Linear Programs

8.2.1. The Standard Form

- To solve LPs, we need some algebraic methods
- Thus, we define a new class of LPs

Definition 128 (Standard form). An LP is given in a *standard form*, if it is written as

$$\max\{c^\top x \mid Ax = b, x \geq 0\}$$

with an *objective function coefficients* $c \in \mathbb{R}^n$, the *coefficient matrix* $A \in \mathbb{R}^{m \times n}$, the *right-hand side* $b \in \mathbb{R}^m$, and the *variables* $x \in \mathbb{R}^n$.

- We denote with $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ the set of feasible solutions
- To ease the analysis of algorithms and structural properties, we assume the following:
 1. The LP is given in the standard form as any LP can be transformed into standard form
 2. $P \neq \emptyset$, i.e., there always exists a feasible solution
 3. $\text{rank}(A) = m < n$, i.e., there are no linear dependencies² between the rows of the matrix (full rank assumption)
- We will prove in the following that these three assumption can be made without loss of generality

²Vectors $a_1, a_2, \dots, a_k \in \mathbb{R}^n$ are *linearly independent* if $\sum_{i=1}^k \alpha_i a_i = 0^n$ only has the solution $\alpha_i = 0$, $\alpha_i \in \mathbb{R}$, $i = 1, \dots, k$.

Transformation into standard form

- Every LP can be modified to be in standard form
- These transformations are done in such a way, that an optimal solution of the modified LP can be transformed into an optimal solution of the original program

Lemma 129. *Every linear program can be transformed into standard form.*

Proof. Simple modifications

- Consider the cases, in which an LP is not given in standard form
- Given: objective is to be minimized
- Modification: multiplication with -1 leads to a maximization problem, i.e.,

$$\min\{c^\top x \mid Ax \leq b, x \geq 0\} = -\max\{-c^\top x \mid Ax \leq b, x \geq 0\}$$

- Given: constraint is $a_i^\top x \leq b_i$
- Modification: add slack variable s_i with

$$a_i^\top x + s_i = b_i \text{ with } s_i \geq 0$$

- Given: constraint is $a_i^\top x \geq b_i$
- Modification: add slack variable s_i with

$$a_i^\top x - s_i = b_i \text{ with } s_i \geq 0$$

- Given: an unbounded variable $x_j \in \mathbb{R}$
- Modification:
 - Replace x_j by two variables y_j and z_j
 - y_j represents the positive part of x_j
 - z_j represents the negative part of x_j
 - Add the constraints:

$$y_j - z_j = x_j, \quad y_j, z_j \geq 0.$$

□

General LP

Standard Form

$$\begin{array}{rcl}
& & \text{Transformation} \\
\min & 3x_1 & + \quad 2x_2 \\
& x_1 & + \quad x_2 \quad \geq \quad 5 \\
& 3x_1 & + \quad 5x_2 \quad \leq \quad 20 \\
& x_1 & \quad \quad \quad \geq \quad 0 \\
& x_2 & \in \quad \mathbb{R} \\
& & \\
& & \Leftrightarrow \quad -\max \quad - \quad 3x_1 \quad - 2(y_2 - z_2) \\
& & \quad x_1 \quad + \quad y_2 - z_2 \quad - \quad s_1 \quad = \quad 5 \\
& & \quad 3x_1 \quad + 5(y_2 - z_2) \quad + \quad s_2 \quad = \quad 20 \\
& & \quad x_1 \quad \geq \quad 0 \\
& & \quad s_1 \quad \geq \quad 0 \\
& & \quad s_2 \quad \geq \quad 0 \\
& & \quad y_2 \quad \geq \quad 0 \\
& & \quad z_2 \quad \geq \quad 0 \\
& & \\
& & x_2 = y_2 - z_2
\end{array}$$

Fig. 8.9.: Transformation of LP into standard form (No. 774)

There is at least one feasible solution

- For problem instances from practice, not necessarily a feasible solution exists
- Can we decide in advance, whether a feasible solution exists?

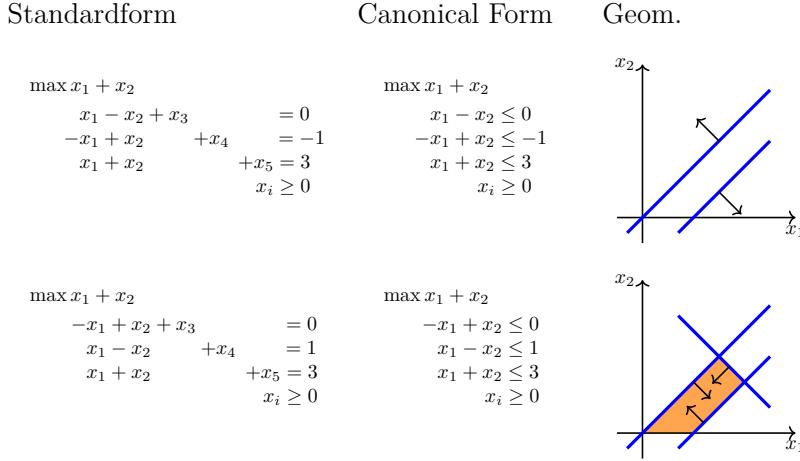


Fig. 8.10.: In general, we cannot see whether a feasible solution exists (No. 826)

- However, we can again eliminate these cases in a preprocessing step
- **Note:** For general optimization problems it is difficult to decide whether $P = \emptyset$ or not (i.e., SAT, Hamiltonian-Cycle,...)

Lemma 130. *If we can compute an optimal solution to any feasible LP, then we can decide for arbitrary LPs whether they are feasible or not.*

Proof. Definition of an auxiliary LP

- Let $S = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ be the set of feasible solutions for a given LP

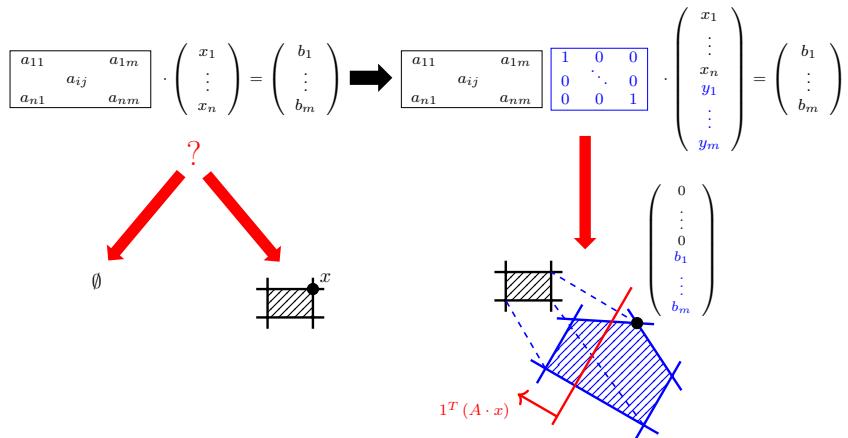


Fig. 8.11.: Transformation of S into optimization problem (No. 775)

- W.l.o.g. $b \geq 0$
- Set $S_{\text{aux}} = \{(x, y) \in \mathbb{R}^{n+m} \mid (A, I) \begin{pmatrix} x \\ y \end{pmatrix} = b, x \geq 0, y \geq 0\}$ with $I \in \mathbb{R}^m$ being the unity matrix
- Then, $\begin{pmatrix} 0 \\ b \end{pmatrix} \in S_{\text{aux}}$, i.e., $S_{\text{aux}} \neq \emptyset$
- And: $\exists x \geq 0$ with $Ax = b \Leftrightarrow (A, I) \begin{pmatrix} x \\ y \end{pmatrix} = b$ has a solution with $y = 0$
- To decide whether $S = \emptyset$ or not, we construct the following auxiliary LP:

$$\begin{aligned} (\text{LP}_{\text{aux}}) \quad & \max 1^\top (Ax) \\ \text{s.t. } & (A, I) \begin{pmatrix} x \\ y \end{pmatrix} = b \\ & x, y \geq 0 \end{aligned}$$

- Let $(x, y) \in S_{\text{aux}}$, then

$$1^\top (Ax) = 1^\top (b - y) = 1^\top b - 1^\top y$$

- Since $y \geq 0$, one of the two is true
 - (i) $1^\top (Ax) < 1^\top b$ or
 - (ii) $1^\top (Ax) = 1^\top b$
- (ii) holds if and only if $y = 0$
- Let (x^*, y^*) be an optimal solution of LP_{aux}
- If $1^\top (Ax^*) < 0$, then $S = \emptyset$
- Otherwise, $y^* = 0$ and thus x^* is a feasible solution for the original problem, i.e., $S \neq \emptyset$

□

- By solving LP_{aux} we can decide whether $S = \emptyset$ or not
- If $S = \emptyset$, we can stop our algorithm
- \Rightarrow we assume, that every given LP has a feasible solution, i.e.,

$$S = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\} \neq \emptyset$$

Full Rank Assumption

- We assume in the following $m \leq n$ and $S \neq \emptyset$
- It holds $\text{rank}(A) \leq \min\{m, n\}$
- We assume that $\text{rank}(A) = m$
 - If $\text{rank}(A) < m$, we can eliminate linearly dependent constraints while the problem remains feasible
- We prove: if $\text{rank}(A) = n \Rightarrow$ the LP is easy to solve
- From linear algebra, $Ax = b$ can be solved via the Gaussian elimination
- However, we need to guarantee $x \geq 0$
- \Rightarrow simple to test

Lemma 131. If $\text{rank}(A) = n$, then $Ax = b$ has one unique solution or no solution at all. If this solution is non negative, then the corresponding LP has exactly one solution, i.e., $|S| = 1$.

Proof. Gaussian elimination

- For a system of linear equations $Ax = b$ we have:
 1. $\det(A) = 0$: no solution exists, i.e., $S = \emptyset$
 2. $\det(A) \neq 0$: exactly one solution exists \bar{x}
- Let us assume $\det(A) \neq 0$.
- Then \bar{x} can be computed by Gaussian elimination
- If $\bar{x} \geq 0$, then \bar{x} is the optimal solution of the LP
- If $\bar{x} \not\geq 0$, then $S = \emptyset$

□

- Thus, if $\text{rank}(A) = n$, we can compute efficiently an optimal solution
- $\text{rank}(A) = n$ can easily be tested in a preprocessing step
- Thus, we can assume $\text{rank}(A) = m < n$
- In a next step, we will develop the theory around LPs that lead to algorithms solving LPs

8.2.2. Bases and an Optimality Criterion

- Consider the two linear programs. Which one is easier to solve?

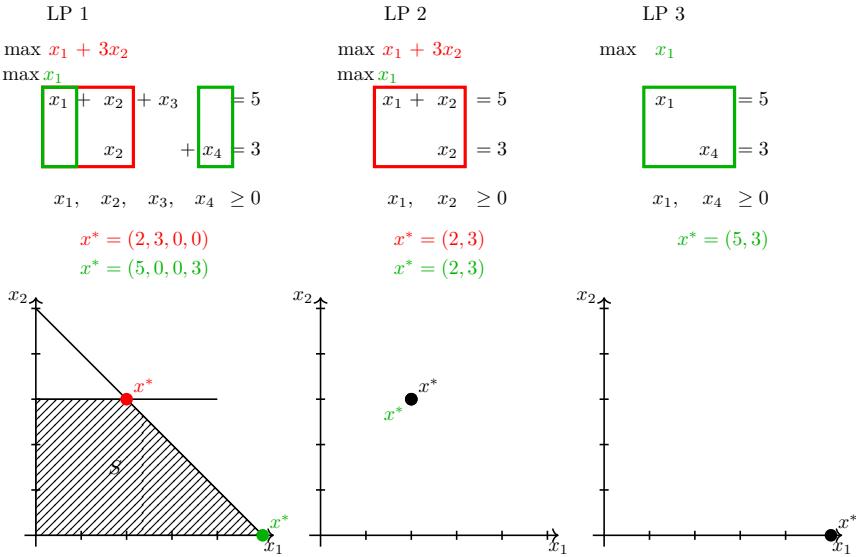


Fig. 8.12.: Which LP to solve? (No. 859)

- It seems, that instead of solving the complicated LP, it suffices to solve a simple subproblem

- We call the columns defining this subproblem a *basis*
- More formally, denote the columns of A by: A_1, \dots, A_n
- For an ordered set $J \subseteq \{1, \dots, n\}$ denote by A_J the matrix consisting of the columns A_j with $j \in J$
- A *basis* $B = (B_1, \dots, B_m) \subseteq \{1, \dots, n\}$ is an ordered subset of the column indices for m linearly independent columns A_{B_1}, \dots, A_{B_m} of A
- The matrix A_B to a basis is called *base matrix* or *base*
- $N = (1, \dots, n) \setminus B$ is the set of *non-basis* elements
- **Example:** Bases and non-bases

- Given the following problem

$$\begin{array}{rccccccl} \max & x_1 & + & x_2 & + & x_3 & & \\ & 3x_1 & + & 7x_2 & & & + & x_4 & = 4 \\ & -1x_1 & - & 1x_2 & - & 2x_3 & & + & x_5 = 3 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 \geq 0 \end{array}$$

- Matrix A in standard form

$$A = \begin{pmatrix} 3 & 7 & 0 & 1 & 0 \\ -1 & -1 & -2 & 0 & 1 \end{pmatrix}$$

with

$$A_1 = \begin{pmatrix} 3 \\ -1 \end{pmatrix}, A_2 = \begin{pmatrix} 7 \\ -1 \end{pmatrix}, A_3 = \begin{pmatrix} 0 \\ -2 \end{pmatrix}, A_4 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, A_5 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

- For $J = (4, 2)$:

$$A_J = \begin{pmatrix} 1 & 7 \\ 0 & -1 \end{pmatrix}$$

- $B' = (4, 5)$ is a basis of the form

$$A_{B'} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

or $B'' = (3, 1)$ is a basis of the form

$$A_{B''} = \begin{pmatrix} 0 & 3 \\ -2 & -1 \end{pmatrix}, \text{ caution not } \bar{A} = \begin{pmatrix} 3 & 0 \\ -1 & -2 \end{pmatrix}.$$

- $N' = (1, 2, 3, 4, 5) \setminus B' = (1, 2, 3)$ and $N'' = (1, 2, 3, 4, 5) \setminus B'' = (2, 4, 5)$
- Special solutions x of an LP are those that are obtained by $x_i = ((A_J^{-1}) \cdot b)_i$ with $J \subseteq \{1, \dots, n\}$, $i \in J$ and $x_i = 0$ otherwise

Definition 132 ((feasible) basic solution). Let B be a basis. Then, $x \in \mathbb{R}^n$ is a *basic solution* if

$$\begin{array}{rcl} A_B x_B & = & b \\ x_N & = & 0. \end{array}$$

A basic solution with $x_B \geq 0$ is called a *feasible basic solution* for the LP with $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$. The variables x_j , $j \in B$, are *basic variables* the variables x_j , $j \in N$ are *non-basic variables*.

- In other words, a basic feasible solution has many zero components
- Furthermore, the definition of a basic solution is independent of the cost vector and only depends on A and b
- **Example** (continuation): Consider again

$$\begin{array}{rclclclclclcl} \max & x_1 & + & x_2 & + & x_3 & & & & & & = & 4 \\ & 3x_1 & + & 7x_2 & & & + & x_4 & & & & & & \\ & -1x_1 & - & 1x_2 & - & 2x_3 & & & + & x_5 & = & 3 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & \geq & 0 \end{array}$$

then $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 4$ and $x_5 = 3$ is a basic solution to the basis $B = (4, 5)$ with $x_B = (x_4, x_5) = (4, 3)$.

- Under our full rank assumption, i.e., $\text{rank}(A) = m$, there always exists a basic solution

Theorem 133. *Let an LP be given in standard form*

$$\max\{c^\top x | Ax = b, x \geq 0\}$$

with $\text{rank}(A) = m < n$. Then there always exists an $x \in \mathbb{R}^n$ such that $Ax = b$.

Proof. Definition of $\text{rank}(A)$

- Since $\text{rank}(A) = m$, we can choose m linearly independent columns from matrix A
 - Let B be a set of m indices of such columns (i.e. $\{A_j | j \in B\}$ are linearly independent) and let $N := \{1, \dots, n\} \setminus B$
 - We write $Ax = b$ as
- $$\sum_{k \in B} x_k A_k + \sum_{k \in N} x_k A_k = b.$$
- Set $x_k = 0$ for all $k \in N$
 - \Rightarrow all x_k for $k \in B$ are uniquely determined, because the $(m \times m)$ sub-matrix A_B consisting of columns of B is non-singular
 - A solution is given by $x = (x_B, x_N)$

□

- Let us assume, we have a feasible solution. How can we determine whether it is a basic solution?

$$A = \begin{pmatrix} 3 & 7 & 0 & 1 & 0 & 6 \\ -1 & -1 & -2 & 0 & 1 & -2 \end{pmatrix}$$

$$x_1 = (1 \ 0 \ 3 \ 2 \ 0 \ 0) \quad \text{X}$$

$$x_1 = (0 \ 0 \ 3 \ 1 \ 0 \ 0) \quad \checkmark$$

$$x_1 = (1 \ 0 \ 0 \ 0 \ 0 \ 1) \quad \text{X}$$

Fig. 8.13.: Which one is a basic solution? (No. 1110)

- Furthermore, given a vector $x \in \mathbb{R}^n$ we can easily decide whether it is a basic solution or not

Theorem 134. *A feasible solution x of a linear program in standard form is basic if and only if the columns of the matrix A_B are linearly independent, where $B = \{j \in [n] \mid x_j > 0\}$.*

Proof. Extension of a linear independent vectors

- (\Rightarrow): Let x be a feasible basic solution with B' being the basis, i.e. $A_{B'}x_{B'} = b$ and $x_N = 0$ where $N = \{1, \dots, n\} \setminus B'$
- The columns $\{A_i \mid i \in B'\}$ are linearly independent, by definition of basis B'
- It holds that $B \subseteq B'$ since $x_{B'} \geq 0$
- \Rightarrow the columns of A_B are linearly independent
- (\Leftarrow): Suppose x is feasible and the columns of A_B are linearly independent
- If $|B| = m =: \text{rank}(A)$, then B already is a basis
 - $\Rightarrow x$ is a basic solution
- If $|B| < m$, then collect $n - |B|$ additional indices in a set K such that the columns $\{A_i \mid i \in B \cup K\}$ are linearly independent
 - It is always possible to find such indices since $\text{rank}(A) = m > |B|$ (Linear Algebra I)
 - Set $B' := B \cup K$ is a basis, and $|B'| = m$
 - Since $x_N = 0$ for $N = [n] \setminus B$ and $x_B > 0$, $x_{B'} \geq 0$ is satisfied
 - $\Rightarrow x$ is a basic feasible solution

□

- In the proof, we distinguished between the number of positive values of x
- If there are less than m , we were able to freely chose the basis
- These kind of basic solution pose some difficulties later on, so we define them as degenerated basic solutions

Definition 135. If one or more of the basic variables in a basic solution have value zero, that solution is said to be a *degenerated basic solution*.

- The most important properties of basic solutions are the following

Theorem 136 (Fundamental Theorem of Linear Programming). *Given a linear program in standard form where A is an $m \times n$ matrix of rank m . Then the following two properties hold:*

1. *If there is a feasible solution, there is a basic feasible solution.*
2. *If there is an optimal feasible solution, there is an optimal basic feasible solution.*

Proof. Extension of a linear independent vectors

- *Property 1:* There exists a basic feasible solution
- Let x be a feasible solution, i.e.

$$\sum_{i=1}^n A_i x_i = b$$

- Assume w.l.o.g. that exactly the first p entries of x are greater than 0
 - Thus,
- $$\sum_{i=1}^p A_i x_i = b$$
- Case 1: The columns A_1, \dots, A_p are linearly independent
 - $\Rightarrow p \leq m = \text{rank}(A)$
 - A basic solution can be obtained as in Theorem 134
 - Case 2: The columns A_1, \dots, A_p are linearly dependent
 - \Rightarrow there exist constants y_1, \dots, y_p of which at least one is positive, such that

$$\sum_{i=1}^p A_i y_i = 0$$

- For all $\varepsilon \in \mathbb{R}$ it holds

$$\sum_{i=1}^p A_i (x_i - \varepsilon \cdot y_i) = b$$

- Set $y = (y_1, \dots, y_p, 0, \dots, 0)$
- $\Rightarrow x' := x - \varepsilon y$ solves $Ax' = b$
- Set $\varepsilon' := \min_{i \in \{1, \dots, p\}} \{ \frac{x_i}{y_i} \mid y_i > 0 \}$
- $A(x - \varepsilon' y)$ is feasible and at least one of the new variables is equal to zero
- \Rightarrow at most $p - 1$ positive variables in solution $x - \varepsilon' y$
- Repeat until all corresponding columns of A are linearly independent
- \Rightarrow Case 1 applies

- *Property 2:* There exists an optimal basic feasible solution
- Let $x = (x_1, \dots, x_n)$ be an optimal feasible solution
- W.l.o.g., assume that the first p entries of x are greater than 0
- Case 1: Columns A_1, \dots, A_p are linearly independent
 - $\Rightarrow x$ is a optimal basic feasible solution (Theorem 134)
- Case 2: Columns A_1, \dots, A_p are linearly dependent
 - We choose y_1, \dots, y_p in case 2 of property 1
 - *Claim:* $c^\top y = 0$.
 - *Proof of Claim*
 - Suppose $c^\top y \neq 0$
 - Set $\varepsilon^+ = \min_{i \in \{1, \dots, p\}} \{ \frac{x_i}{y_i} \mid y_i > 0 \}$ and $\varepsilon^- = \min_{i \in \{1, \dots, p\}} \{ \frac{x_i}{y_i} \mid y_i < 0 \}$
 - Choose $0 \leq \varepsilon \leq \min\{\varepsilon^+, \varepsilon^-\}$

- $\Rightarrow x^+ = x + \varepsilon y$ and $x^- = x - \varepsilon y$ are feasible
- Then, either $c^\top x^+ = c^\top x + c^\top \varepsilon y > c^\top x$ or $c^\top x^- = c^\top x - (-\varepsilon)c^\top y > c^\top x$
- \Rightarrow Contradiction to optimality of x $\square C$
- Set $\varepsilon' := \min_{i \in \{1, \dots, p\}} \left\{ \frac{x_i}{y_i} \mid y_i > 0 \right\}$
- Then, $x' = x - \varepsilon' y$ is a feasible solution and

$$c^\top x = c^\top (x - \varepsilon' y) = c^\top x'$$

- $\Rightarrow x'$ is an optimal feasible solution with fewer positive components
- Proceed as in property 1 (repeat until basic optimal solution is obtained)

\square

- The first part of the Theorem is often referred to Carathéodory's Theorem
- It furthermore provides a first idea to compute an optimal solution by enumeration
 - Step 1 Derive all subsets of $\{1, \dots, n\}$ of size m and denote this list with N
 - Step 2 For every set $B \in N$
 - test if $\text{rank}(A_B) = m$
 - If this is the case, compute $x_B = A^{-1}b$ and $c_B^\top x_B$
 - Step 3 Choose the solution x_B maximum cost
- The run-time of the algorithm is determined by N , the number of subsets of $\{1, \dots, n\}$
- Here, we obtain

$$|N| = \binom{m}{n} = \frac{n!}{(n-m)!m!}$$

- In the following, we will try to reduce the number of tested bases
- To that end, look for an optimality criterion

Theorem 137 (Optimality criterion for LPs). *Let an LP be given in standard form*

$$\max\{c^\top x \mid Ax = b, x \geq 0\}$$

with $\text{rank}(A) = m < n$. Consider a feasible, non-degenerated basic solution x_B^ according to a basis B . Set the reduced cost according to B as $\bar{c}^\top = (c^\top - c_B^\top A_B^{-1}A)$. Then, x_B^* is an optimal solution if $\bar{c}^\top \leq 0$.*

Proof. Definition of basic solution

- Let $B \subseteq \{1, \dots, n\}$ be a basis
- Let $x^* = (x_B^*, x_N^*)$ be the feasible basic solution of the basis B with $x_B^* = A_B^{-1}b$ and $x_N^* = 0$
- Let $x = (x_B, x_N)$ be a any feasible solution
- Then, $Ax = b \Leftrightarrow A_B x_B + A_N x_N = b$
- $\Leftrightarrow x_B = A_B^{-1}b - A_B^{-1}A_N x_N$, i.e., the basic variables are represented by the non-basic variables

- Consider the objective function $c^\top x$:

$$\begin{aligned} c^\top x &= c_B^\top A_B^{-1}b + (c_N^\top - c_B^\top A_B^{-1}A_N)x_N \\ &= c_B^\top x_B^* + (c_N^\top - c_B^\top A_B^{-1}A_N)x_N \end{aligned}$$

- (\Leftarrow): $\bar{c}_N^\top \leq 0$

- \Rightarrow

$$\begin{aligned} c^\top x &= c_B^\top x_B^* + (c_N^\top - c_B^\top A_B^{-1}A_N)x_N \\ &\leq c_B^\top x_B^* \end{aligned}$$

- $\Rightarrow x^*$ is optimal

- (\Rightarrow): x_B^* is an optimal solution

- Assume: there exists $s \in N$ with $\bar{c}_s > 0$

- Define $x_B(\gamma) = A_B^{-1}b - A_B^{-1}A_s\gamma$ for $\gamma \in \mathbb{R}_{>0}$, $x_i(\gamma) = 0$ for $i \in N \setminus (B \cup \{s\})$ and $x_s(\gamma) = \gamma$

- Set $\bar{A} = A_B^{-1}A$ and $\bar{b} = A_B^{-1}b$

- *Claim:* If $\gamma \leq \min_{i \in B} \left\{ \frac{\bar{b}_i}{\bar{a}_{is}} \mid \bar{a}_{is} > 0 \right\}$, $x'(\gamma)$ is a feasible solution.

Proof of Claim

- $x'_N(\gamma)$ is positive
- Furthermore,

$$\begin{aligned} x_B(\gamma) &= A_B^{-1}b - A_B^{-1}A_N x_N(\gamma) \\ &= \bar{b} - \gamma \bar{A}_s \geq 0 \end{aligned}$$

due to the choice of γ

- Furthermore,

$$\begin{aligned} Ax(\gamma) &= A_B(A_B^{-1}b - A_B^{-1}A_s\gamma) + A_s \cdot \gamma \\ &= b - A_s\gamma + A_s\gamma. \end{aligned}$$

□C

- Set $\gamma = \min_{i \in B} \left\{ \frac{\bar{b}_i}{\bar{a}_{is}} \mid \bar{a}_{is} > 0 \right\}$

- Since x^* is non-degenerated, $\gamma > 0$

- Thus,

$$\begin{aligned} c^\top x(\gamma) &= c_B^\top x_B^* + (c_N^\top - c_B^\top A_B^{-1}A_N)x_N \\ &= c_B^\top x_B^* + \bar{c}_s\gamma > c_B^\top x_B^* \end{aligned}$$

- Contradiction to the optimality of x^*

□

- Unfortunately, the condition is not necessary if we have a degenerated base

- Consider

$$\begin{array}{rclcl} \max & x_1 & + & x_2 & \\ & & & x_2 & + x_3 = 0 \\ & x_1 & & & + x_4 = 3 \\ & x_1, & x_2, & x_3, & x_4 \geq 0 \end{array}$$

- $B = (1, 3)$ is basis with an optimal solution $x^* = (3, 0, 0, 0)$
- However, $\bar{c}_2 = 1$
- Still, the theorem provides good insights into improving a given solution
 - Let B be a basis
 - If we increase the value x_s with $\bar{c}_s > 0$, $s \in N$, we improve the objective for the new solution x
 - That means, we want to add s to the basis B
 - Furthermore, we need to delete an index from B in order to obtain a new feasible basic solution
 - Question: Do we need to test all $r \in B$ in order to decide which index to remove from the basis?
 - It holds $x_B = A_B^{-1}b - A_B^{-1}A_N x_N$ and we need to guarantee $x_B \geq 0$

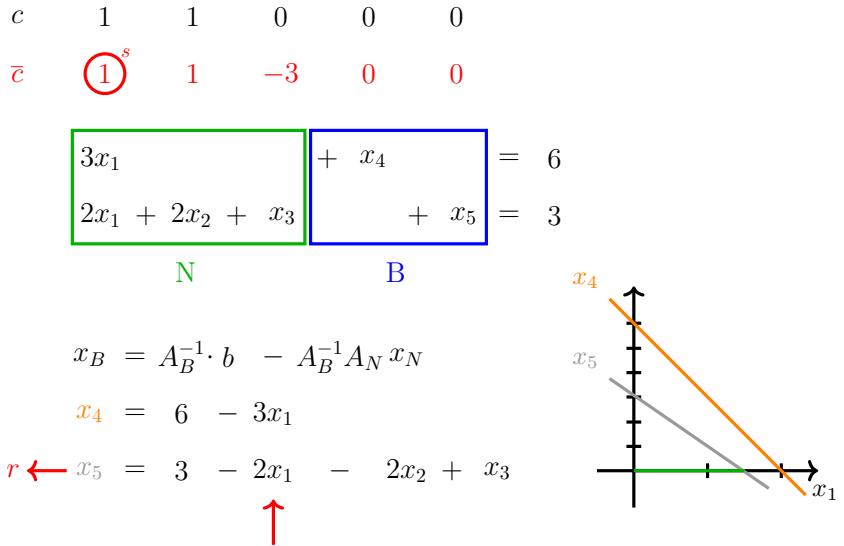


Fig. 8.14.: Obtaining a new basis (No. 776)

- \Rightarrow Increase the value of x_s until the first basic variable x_r decreases to 0
Formally³:

$$r \in \arg \min_{i \in B} \left\{ \frac{\bar{b}_i}{\bar{a}_{is}} \mid \bar{a}_{is} > 0 \right\}$$

with $\bar{b} = A_B^{-1}b$ and $\bar{A}_s = A_B^{-1} \cdot A_s$

- Remove r from bases
- We need to prove, that the intuition is right:
 - Changing our basis leads to a better solution
 - If no change is possible, stop
- In summary, we obtain the following theorem:

³Note, we assume here that $\arg \min$ exists and $\bar{b}_i > 0$. If this is not the case, the problem is either unbounded or the basis is degenerated (see Simplex Algorithm 8.1)

Theorem 138. Let an LP be given in standard form

$$\max\{c^\top x \mid Ax = b, x \geq 0\}$$

with $\text{rank}(A) = m < n$. Let x_B be a basic feasible solution, $\bar{A} = A_B^{-1}A$, $\bar{b} = A_B^{-1}b$ and $\bar{c}^\top = (c^\top - c_B^\top A_B^{-1}A)$. Let s be some index in $N \setminus B$ with $\bar{c}_s > 0$. Set

$$r \in \arg \min_{i \in B} \left\{ \frac{\bar{b}_i}{\bar{a}_{is}} \mid \bar{a}_{is} > 0 \right\}$$

with $\bar{b} = A_B^{-1}b$ and $\bar{A}_s = A_B^{-1} \cdot A_s$.

1. If r does not exist, the optimal function is unbounded.

2. If r exists, $B' = B \setminus \{r\} \cup \{s\}$ is a basis with $c(x_{B'}) \geq c(x_B)$.

$$\begin{array}{ccc} & \begin{array}{c} \text{a better} \\ \text{solution} \\ \text{exists} \end{array} & \begin{array}{c} \text{a better} \\ \text{solution may} \\ \text{exists} \end{array} \\ \uparrow & & \uparrow \\ \bar{A} = \begin{pmatrix} 3 & -1 & 1 & 0 & 0 \\ 2 & -3 & 0 & 1 & 0 \\ 1 & -5 & 0 & 0 & 1 \end{pmatrix} & \bar{b}_1 = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \quad \bar{b}_2 = \begin{pmatrix} 3 \\ 2 \\ 0 \end{pmatrix} \\ \bar{c} = (\boxed{2} \ -3 \ 0 \ 0 \ 0) & \begin{array}{c} \uparrow \\ \rightarrow \text{a better solution may exist} \end{array} & \begin{array}{c} \uparrow \\ \rightarrow \text{the solution is optimal} \end{array} \\ \bar{c} = (\boxed{-1} \ 4 \ 0 \ 0 \ 0) & & \\ \bar{c} = (-1 \ \boxed{2} \ 0 \ 0 \ 0) & \begin{array}{c} \uparrow \\ \rightarrow \text{the objective is unbounded} \end{array} & \end{array}$$

Fig. 8.15.: Interpretation of \bar{A} (No. 1133)

Proof. Definition of feasible solutions

- *Case 1:* The r does not exist, i.e., $\bar{a}_{is} \leq 0$ for all $i \in B$
- Define $x_B(\gamma) = A_B^{-1}b - A_B^{-1}A_s\gamma$ for $\gamma \in \mathbb{R}_{>0}$, $x_i(\gamma) = 0$ for $i \in N \setminus (B \cup \{s\})$ and $x_s(\gamma) = \gamma$

$$\begin{array}{ccc} \bar{A} = \begin{pmatrix} 3 & \boxed{-1} & 1 & 0 & 0 \\ 2 & \boxed{-3} & 0 & 1 & 0 \\ 1 & \boxed{-5} & 0 & 0 & 1 \end{pmatrix} & & \bar{b} = \begin{pmatrix} 3 \\ 5 \\ 6 \end{pmatrix} \\ \bar{c} = (-1 \ \boxed{2} \ 0 \ 0 \ 0) & & \end{array}$$

$$x_B(\gamma) = \begin{pmatrix} 3 \\ 5 \\ 6 \end{pmatrix} - \gamma \cdot \begin{pmatrix} -1 \\ -3 \\ -5 \end{pmatrix} \quad x_1 = 0 \quad x_2 = y$$

Fig. 8.16.: Define $x_B(\gamma)$ (No. 1134)

- *Claim:* $x(\gamma)$ is a feasible solution.

Proof of Claim

- $x_N(\gamma)$ is positive
- Furthermore,

$$\begin{aligned} x_B(\gamma) &= A_B^{-1}b - A_B^{-1}A_N x_N(\gamma) \\ &= \bar{b} - \gamma \bar{A}_s \geq 0 \end{aligned}$$

since $-\gamma \bar{A}_s \geq 0$

- Furthermore,

$$\begin{aligned} Ax(\gamma) &= A_B(A_B^{-1}b - A_B^{-1}A_s\gamma) + A_s \cdot \gamma \\ &= b - A_s\gamma + A_s\gamma. \end{aligned}$$

□C

- The cost are

$$\begin{aligned} c^\top x(\gamma) &= c_B^\top x_B - c_B^\top A_B^{-1}A_s\gamma + c_s\gamma \\ &= c_B^\top x_B + (c_s - c_B^\top A_B^{-1}A_s)\gamma \\ &= c_B^\top x_B + \bar{c}_s\gamma \rightarrow \infty \text{ as } \gamma \rightarrow \infty \end{aligned}$$

- \Rightarrow the set of feasible solutions is unbounded and objective can be made arbitrarily large
- *Case 2: r exists*
- *Claim: $B' = B \setminus \{r\} \cup \{s\}$ is a basis.*

Proof of Claim

- Note that $A_{B'} = A_B G$ where G is the matrix

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & \bar{a}_{1s} & 0 & \cdots & 0 \\ 0 & 1 & & 0 & & \vdots & & \\ 0 & \ddots & 1 & \bar{a}_{(r-1)s} & & & & \\ \vdots & \ddots & 0 & \bar{a}_{rs} & 0 & & \vdots & \\ & & & \bar{a}_{(r+1)s} & 1 & \ddots & & \\ & & \vdots & \vdots & 0 & \ddots & 0 & \\ 0 & \cdots & 0 & \bar{a}_{ms} & 0 & 0 & 0 & 1 \end{bmatrix}$$

- This is because multiplying the r th column of G with A_B yields

$$A_B A_B^{-1} A_s = A_s$$

while all other columns remain the same

- Matrix G is non-singular because $\bar{a}_{rs} \neq 0$ (by assumption)
- $\Rightarrow A_{B'}$ is non-singular, since it is a product of two non-singular matrices

- Define matrix

$$H := \begin{bmatrix} 1 & 0 & \cdots & 0 & \eta_1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 & & \vdots & & \\ 0 & \ddots & 1 & \eta_{r-1} & & & & \\ \vdots & \ddots & 0 & \eta_r & 0 & & \vdots & \\ & & & \eta_{r+1} & 1 & \ddots & & \\ & & \vdots & \vdots & 0 & \ddots & 0 & \\ 0 & \cdots & 0 & \eta_m & 0 & 0 & 0 & 1 \end{bmatrix}$$

with $\eta_i := -\bar{a}_{is}/\bar{a}_{rs}$ for $i \neq r$ and $\eta_r := 1/\bar{a}_{rs}$

- H is inverse to G , i.e. $G^{-1} = H$ (this can easily be verified by multiplying the two)
- It follows from $A_{B'} = A_B G$ that $A_{B'}^{-1} = H A_B^{-1}$

- Therefore, the new basic solution x' is determined as follows:

$$\begin{aligned} x'_{B'} &= HA_B^{-1}b - HA_B^{-1}A_{N'}x'_{N'} \\ &= H(A_B^{-1}b - A_B^{-1}A_{N'}x'_{N'}) \\ &= Hx_B \end{aligned}$$

- Consider now \bar{c}_s :

$$\begin{aligned} 0 < \bar{c}_s &= c_s - c_B^\top \bar{A}_s \\ &= \bar{a}_{rs} \left(\frac{1}{\bar{a}_{rs}} c_s - \sum_{\substack{i \in B: \\ i \neq r}} \frac{\bar{a}_{is}}{\bar{a}_{rs}} - c_r \right) \\ &= \bar{a}_{rs} (c_{B'}^\top \eta - c_r) \end{aligned}$$

- Thus

$$\begin{aligned} c^\top x' &= c_{B'}^\top x'_{B'} \\ &= c_{B'}^\top Hx_B \\ &= c_B^\top x_B + \frac{\bar{a}_{rs}}{\bar{a}_{rs}} (c_{B'}^\top \eta - c_r) x_r \\ &\geq c^\top x \end{aligned}$$

□

- All this leads to the Simplex Algorithm

8.3. The Simplex Algorithm

- The Simplex Algorithm is based on the theory developed above

Algo. 8.1 The Simplex algorithm

Input: Maximization problem $\max\{c^\top x \mid Ax = b, x \geq 0\}$

Output: Optimal basic solution x_B if one exists

Method:

Step 1 Find an initial feasible basic solution x_B to the basis B

Step 2 Compute the reduced cost $\bar{c}_N^\top = (c_N^\top - c_B^\top A_B^{-1} A_N)$ for all non-basis variables $N = \{1, \dots, n\} \setminus B$

If $\bar{c}_N^\top \leq 0$: Stop and **Return** x_B // x_B is an optimal solution

Else chose $s \in N$ with $\bar{c}_s > 0$ // *Pricing step*

Step 3 Compute $\bar{b} = A_B^{-1}b$ and $\bar{A}_s = A_B^{-1} \cdot A_s$

If $\bar{A}_s \leq 0$: Stop and **Return** Problem unbounded

Else Select index r with

$$r \in \arg \min_{i=1, \dots, m} \left\{ \frac{\bar{b}_i}{\bar{a}_{B_i s}} \mid \bar{a}_{B_i s} > 0 \right\}$$

Set $B = B \cup \{s\} \setminus \{r\}$ **Goto** Step 1

$$Ax \leq b, b \geq 0$$

$$\begin{array}{l} \max 3x_1 + 4x_2 \\ x_1 + 2x_2 \leq 3 \\ 2x_1 - x_2 \leq 5 \\ x_1, x_2 \geq 0 \end{array} \xrightarrow{\text{transform}} \begin{array}{l} \max 3x_1 + 4x_2 \\ x_1 + 2x_2 + x_3 = 3 \\ 2x_1 - x_2 + x_4 = 5 \\ x_1, x_2, x_3, x_4 \geq 0 \end{array}$$

$$Ax \leq b, b \geq 0$$

$$\begin{array}{l} \max 3x_1 + 4x_2 \\ x_1 + 2x_2 \leq -3 \\ 2x_1 - x_2 \leq 5 \\ x_1, x_2 \geq 0 \end{array} \xrightarrow{\text{transform}} \begin{array}{l} \max 3x_1 + 4x_2 \\ x_1 + 2x_2 + x_3 = -3 \\ 2x_1 - x_2 + x_4 = 5 \\ x_1, x_2, x_3, x_4 \geq 0 \end{array}$$

Fig. 8.17.: Simplex algorithm (No. 1140)

- There are several points to be specified in order to use the algorithm

Calculating an optimal solution using the Simplex tableau

- If one needs to calculate an optimal solution by hand, the most convenient way is using a Simplex tableau
- The start tableau for the problem

$$\max\{c^\top x | Ax \leq b, x \geq 0\}$$

with $b \geq 0$ is given by

	c_1	c_2	\dots	0	\dots	0	-0
x_{n+1}	a_{11}	a_{12}	\dots	0	\dots	0	b_1
\vdots	\vdots			1		\vdots	\vdots
x_{n+m}	a_{12}	a_{m2}	\dots	0	\dots	1	b_m
	x_1	x_2	\dots	x_{n+j}	\dots	x_{n+m}	

- In the tableau, the coefficients to a given basis B are summarized:

	\bar{c}_1	\bar{c}_2	\dots	\bar{c}_{n+j}	\dots	\bar{c}_{n+m}	$-\bar{z}$
x_{B_1}	\bar{a}_{11}	\bar{a}_{12}	\dots	\bar{a}_{1n+j}	\dots	\bar{a}_{1n+m}	\bar{b}_1
\vdots	\vdots					\vdots	\vdots
x_{B_n}	\bar{a}_{m1}	\bar{a}_{m2}	\dots	\bar{a}_{mn+j}	\dots	\bar{a}_{mn+m}	\bar{b}_m
	x_1	x_2	\dots	x_{n+j}	\dots	x_{n+m}	

- With $\bar{A} = A_B^{-1}A$, $\bar{b} = A_B^{-1}b$, and \bar{c}_i the reduced cost
- From the left and right column, x_B can be directly seen
- $-\bar{z}$ denotes the current negated objective value
- Doing one step of the Simplex algorithm:
 - Chose $s \in \{1, \dots, n+m\}$ with $\bar{c}_s > 0$
 - Compute $r \in \arg \min_{j \in B} \left\{ \frac{\bar{b}_j}{\bar{a}_{js}} \right\}$ for all $\bar{a}_{rs} > 0$
 - Update \bar{A} , \bar{b} , and \bar{c} by Gaussian elimination, i.e., transform column \bar{A}_r to unit vector with $\bar{a}_{rs} = 1$
 - Note: the element \bar{a}_{rs} is called the *pivot element*

- **Example:** Simplex Algorithm using the Simplex tableau

- Consider the following problem:

$$\begin{array}{rclcl}
 \max & 2x_1 & + & x_2 & \\
 & x_1 & & + & x_3 = 4 \\
 & 2x_1 & + & x_2 & + x_4 = 10 \\
 & x_1, & x_2, & x_3, & x_4 \geq 0
 \end{array}$$

- Starting with the basis $B = (3, 4)$ and by selecting $s = 1$ and $r = 3$ in the first step and $s = 2$ and $r = 4$ in the second step, one obtains the optimal tableau, which represents an optimal solution (details on the tableau, see below)

Simplex Tableau

					s						s					
					↓						↓					
					$\frac{b_i}{a_{s_i}}$						$\frac{b_i}{a_{s_i}}$					
x_3	2	1	0	0	0	$\frac{b_i}{a_{s_i}}$	0	1	-2	0	$\frac{b_i}{a_{s_i}}$	0	-1	-1	-10	
	1	0	1	0	4	$\frac{b_i}{a_{s_i}}$	1	0	1	0	$\frac{b_i}{a_{s_i}}$	1	0	1	4	
x_4	2	1	0	1	10	$\frac{b_i}{a_{s_i}}$	0	1	-2	1	$\frac{b_i}{a_{s_i}}$	2	2	1	2	
	x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4		

Initial Tableau

First Tableau

Optimal Tableau

Fig. 8.18.: The tableau reflects the optimal basic solution $B = (1, 2)$ with $x_B = (4, 2)$ since all reduced costs are non-positive. (No. 677)

- Given a simplex tableau, different parameters are easy to detect

Tableau 1						
	-3	0	-2	-1	0	0
x_5	5	0	7	4	1	0
x_2	3	1	8	1	0	0
x_6	2	0	3	0	0	1
	x_1	x_2	x_3	x_4	x_5	x_6

Tableau 2						
	0	0	0	0	2	-1
x_1	1	0	0	3	-1	3
x_2	0	1	0	-2	-1	2
x_3	0	0	1	1	-1	6
	x_1	x_2	x_3	x_4	x_5	x_6

Tableau 3						
	0	2	0	-1	3	0
x_1	1	4	0	2	0	0
x_3	0	2	1	3	2	0
x_6	0	3	0	6	-1	1
	x_1	x_2	x_3	x_4	x_5	x_6

- Current base?
 - Current basic solution?
 - Current objective?
 - Optimal?

Fig. 8.19.: Different Tableaus (No. 1141)

Find a feasible basic solution (Step 1)

- Can be done by following the construction in Lemma 130, however, we will do a short recap here
 - Main idea: add columns/variables to the problem s.t. a basis with a feasible solution can easily be determined (e.g., the identity matrix) and the variables will never be part of an optimal solution
 - **Case 1:** an LP is given as $Ax \leq b$
 - Add so called *slack variables* to every row, i.e.,
 - Consider new matrix $(A \mid I_m)$ where I_m represents the identity matrix

- \Rightarrow new columns $\{n+1, \dots, n+m\}$, set cost $c_j = 0$ for $j \in \{n+1, \dots, n+m\}$
- Chose $B = (n+1, \dots, n+m)$ as basis with $A_B = I_m = A_B^{-1}$
- If $b \geq 0 \Rightarrow x_B = b$ is a feasible solution \Rightarrow just solve the new problem
- Let $b \not\geq 0$
- Add negated identity matrix $-I_m$ to the problem (or just the parts with $b_i < 0$)
- The new variables are known as *artificial variables*
- Solve the linear program $\text{LP}_{\text{aux}} \max \{p^\top x \mid (A \mid I_m \mid -I_m)x = b, x \geq 0\}$ with $p_j = -1, j \in \{n+m+1, \dots, n+2m\}$ and $p_j = 0$ otherwise
- For $j = 1, \dots, m$, add j' to B with $j' = n+j$ if $b_j \geq 0$ and $j' = n+m+j$ otherwise
- Then, B is a base and determines a basic feasible solution to LP_{aux}
- Let x_{aux}^* be an optimal solution to LP_{aux}
- If $c(x_{\text{aux}}^*) < 0$, then no feasible solution exists
- Otherwise, use the basis of the final solution as starting basis for the original problem

Original problem in standard form

$$c = (2, -3, 1, 0, 0, 0)$$

$$A = \begin{pmatrix} 3 & 6 & 1 & 1 & 0 & 0 \\ 4 & 2 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 6 \\ 4 \\ 3 \end{pmatrix}$$

Reduced cost (Step 2)

$$\bar{c}^T = (0, -3, 0, 0, -\frac{1}{3}, -\frac{2}{3})$$

STOP

Current solution (Step 1)

$$B = (4, 1, 3)$$

$$A_B = \begin{pmatrix} 1 & 3 & 1 \\ 0 & 4 & 1 \\ 0 & 1 & 1 \end{pmatrix} \quad A_B^{-1} = \begin{pmatrix} 1 & -\frac{2}{3} & -\frac{1}{3} \\ 0 & \frac{1}{3} & -\frac{1}{3} \\ 0 & -\frac{1}{3} & \frac{4}{3} \end{pmatrix}$$

$$x_B = \begin{pmatrix} \frac{7}{3} \\ \frac{1}{3} \\ \frac{8}{3} \end{pmatrix}$$

Select leaving element (Step 3)

Fig. 8.20.: Find a feasible basic solution (No. 1142)

- **Case 2:** there are constraints $a_j^\top x = b_j$
 - If $b_j \geq 0$: add a slack variable with penalty cost $-M$ and unit vector e_j to the problem
 - If $b_j < 0$: add artificial variables with penalty cost $-M$ and the negated unit vector $-e_j$ to the problem
 - Easy to obtain a basis to start the Simplex algorithm (see above)

Pricing and Pivot Rules (Step 2+3)

- There are different selection rules for s (pricing rules and pivot rules)
- *Dantzig rule*: described according to G.B. Dantzig (1947)
 - Chose $s \in \arg \max_{j \in N} \{\bar{c}_j\}$
- *Steepest-edge pricing*: Combination of column and row selection, which collectively provide the greatest improvement for the objective function, i.e.,

$$s \in \arg \max_{j \in N} \left\{ \bar{c}_j \cdot \min_{i \in B} \left\{ \frac{\bar{b}_i}{\bar{a}_{ij}} \mid \bar{a}_{ij} > 0 \right\} \right\}$$

Degenerate Vertices and Cycling

- Theoretically, a sequence of non-improving pivot steps can be arbitrarily repeated (*cycling*)

- Example (Beale, 1955)**

- Problem

$$\begin{array}{llllllll}
 \max & \frac{3}{4}x_1 & - & 150x_2 & + & \frac{1}{50}x_3 & - & 6x_4 \\
 \text{s.t.} & \frac{1}{4}x_1 & - & 60x_2 & - & \frac{1}{25}x_3 & + & 9x_4 \leq 0 \\
 & \frac{1}{2}x_1 & - & 90x_2 & - & \frac{1}{50}x_3 & + & 3x_4 \leq 0 \\
 & & & & & x_3 & \leq & 1 \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 \geq 0
 \end{array}$$

- If you choose a column according to Dantzig and a row "if in doubt the top one" you get the initial basis again after some pivot steps.

	$\frac{3}{4}$	-150	$\frac{1}{50}$	-6	0	0	0	0
x_5	$\frac{1}{4}$	-60	$-\frac{1}{25}$	9	1	0	0	0
x_6	$\frac{1}{2}$	-90	$-\frac{1}{50}$	3	0	1	0	0
x_7	0	0	1	0	0	0	1	1
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	

Next simplex step = Start table

Previous base variables: $2 \times \{x_5, x_6, x_7\}, \{x_1, x_6, x_7\}, \{x_1, x_2, x_7\}, \{x_2, x_3, x_7\}, \{x_3, x_4, x_7\}, \{x_4, x_5, x_7\}$

Fig. 8.21.: The shown tableau corresponds to the initial tableau but is also recreated after 6 simplex iterations. (No. 682)

- Cycling can only occur in degenerate vertices
 - A vertex is denoted as *degenerate* if the corresponding basic variables have a value of 0

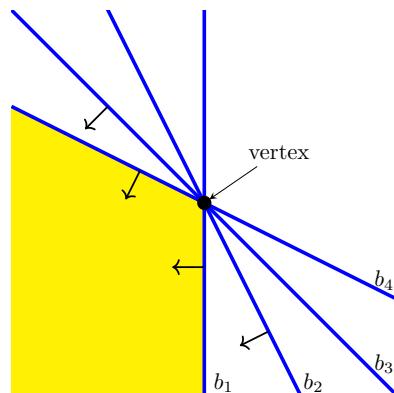


Fig. 8.22.: Degenerate vertices only occur if more than n constraints define the vertex (here: 2 would be sufficient, but there are 4). (No. 685)

- Cycling is a theoretical problem but in practice rather irrelevant
- Degeneracy, i.e. basis values take on 0, however, is a serious issue in practice, as the number of pivot steps can increase considerably.
- Solution for degeneracy: perturbation of the right-hand side

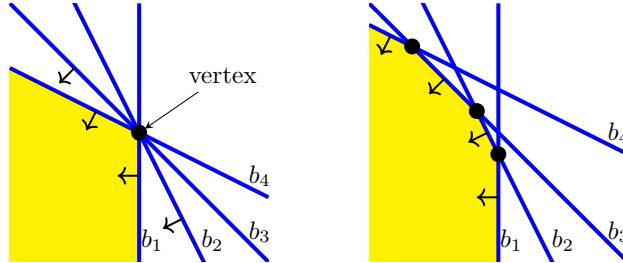


Fig. 8.23.: By changing the right-hand side, new non-degenerate vertices are created. (No. 686)

- Symbolically, $b_i \leftarrow b_i + \varepsilon_i$ with $0 < \varepsilon_1 \ll \varepsilon_2 \ll \dots \ll \varepsilon_m \ll 1$
- Lexicographical row selection is the implementation of the symbolic ε -disturbance (but not practicable)
- Basically, we disturb the right hand side numerically as soon as degeneracy occurs
- A method to prevent cycling: lexicographical pivot row selection
 - So far: two rows v_1, v_2 are equally “good” \Rightarrow roll dice, to choose which we will use as pivot row
 - Idea: seek criterion such that all rows are different
 - Lexicographical differentiation: compare two vectors in the first entry in which they differ.
 - Let $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^n$. Then u is *lexicographically greater* than v if $v_i = u_i$, for $i = 1, \dots, j-1$ and $u_j > v_j$.
Notation: $u \succ v$.
 - Lexicographical row selection (Dantzig, Orden, Wolfe, 1955)
 - Extended ratio test: select pivot row r with

$$r = \arg \operatorname{lex} \min_{i \in B} \left\{ \frac{(\bar{b}_i, \bar{a}_i^\top)^\top}{\bar{a}_{is}} \mid \bar{a}_{is} > 0 \right\}$$

- Example by Beale:
 - For the tableau calculation (slightly different than normal): write \bar{b} “to the left”, then I_m , A_N

	x_5	x_6	x_7	x_1	x_2	x_3	x_4
x_5	0	1	0	$\frac{3}{4}$	-150	$\frac{1}{50}$	-6
x_6	0	0	1	$\frac{1}{4}$	-60	$-\frac{1}{25}$	9
x_7	1	0	0	0	-90	$-\frac{1}{50}$	3
					0	1	0

- Select x_1 as pivot column
- In ratio test divide whole row by possible pivot element:

- a) divide $(0, 1, 0, 0, \frac{1}{4}, -60, -\frac{1}{25}, 9)$ by $\frac{1}{4} \Rightarrow (0, 4, 0, 0, 1, -240, -\frac{4}{25}, 36)$
- b) divide $(0, 0, 1, 0, \frac{1}{2}, -90, -\frac{1}{50}, 3)$ by $\frac{1}{2} \Rightarrow (0, 0, 2, 0, 1, -180, -\frac{1}{25}, 6)$

- Lexicographical minimum is unique
 $(0, 4, 0, 0, 1, -240, -\frac{4}{25}, 36) \succ (0, 0, 2, 0, 1, -180, -\frac{1}{25}, 6)$
- \Rightarrow choose b)

- Objective function row falls strictly lexicographically with every pivot step
 \Rightarrow cycling precluded

- Pivot rules that prevent cycling are referred to as *finite*
- Further finite pivot rules exist, e.g. Bland's rule (1977): [Gärtner p. 72: Blands rule]
 - Among the possible pivot columns or rows, select the one with the lowest subscript (least subscript rule)

Theorem 139 (Termination of the Simplex algorithm). *When using a finite pivot rule, the Simplex algorithm terminates with exactly one of the three alternatives:*

1. *LP is infeasible*
2. *Objective function is unbounded*
3. *Output of a finite optimal solution.*

- The proof can be found in “Understanding and Using Linear Programming” by Matousek and Gärtner
- Unfortunately, Bland's rule is one of the slowest pivot rules and it is almost never used in practice

Run time

- In the worst case, exponentially many iterations are needed
- **Example:** Klee-Minty-Cube (1972)
 - Consider the problem

$$\begin{aligned}
 & \max \sum_{j=1}^n 10^{n-j} x_j \\
 \text{s.t. } & \left(2 \sum_{j=1}^{i-1} 10^{i-j} x_j \right) + x_i \leq 100^{i-1} \quad \forall i = 1, \dots, n \\
 & x_j \geq 0 \quad \forall j = 1, \dots, n
 \end{aligned}$$

- The slightly disturbed cube has 2^n vertices (n = number of variables)
- Start in $(0, \dots, 0)$
- In the worst case, an ascending path can visit all vertices
- \Rightarrow we have 2^n pivot steps

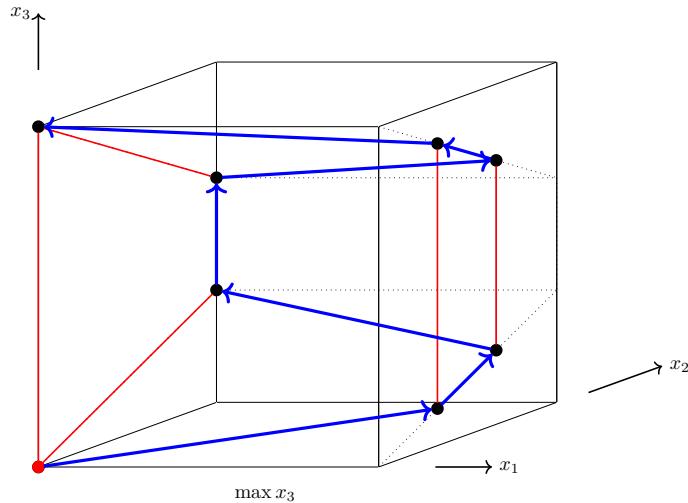


Fig. 8.24.: For $n = 3$ the Klee-Minty-Cube has the blue shape on the “normal cube”. In the worst case, starting in the red vertex, we can find a path where the objective function value improves in each vertex and a total of 8 vertices are visited. (No. 681)

- No polynomial pivot rule is known
- In practice: $\mathcal{O}(m)$ iterations often suffice
- For further reading: *smoothed analysis* of the Simplex algorithm (see Spielman, Teng [?])

8.4. Duality for Linear Programs

Motivation and Definition

- Bounds on the objective are essential for the efficiency of algorithms but also for their theoretical analysis
- Lower bounds on a maximization problem
 - Value at most as large as optimum objective function value
 - It is sufficient to evaluate any feasible solution
- Upper bound on a maximization problem
 - Value at least as large as optimum objective function value
 - Interesting in theory and practice, e.g. to assess the improvement potential of an existing solution

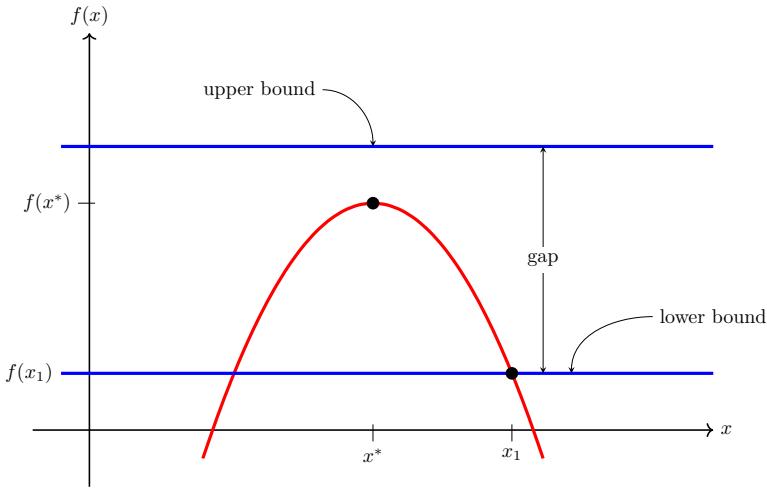


Fig. 8.25.: The maximum of the function f is assumed in x^* . The objective function value of any feasible point (e.g. x_1) always represents a lower bound. Upper bounds are more difficult to obtain. (No. 701)

- Question: How can we obtain an upper bound for a linear program?

- **Example:**

$$\begin{aligned}
 \max \quad & 2x_1 + 4x_2 - 3x_3 \\
 \text{s.t.} \quad & x_1 + 3x_2 - x_3 \leq 4 \quad (I) \\
 & x_1 + 2x_2 - 2x_3 \leq 3 \quad (II) \\
 & x_1, x_2, x_3 \geq 0
 \end{aligned}$$

- Consider objective function value: $2x_1 + 4x_2 - 3x_3$
- For each feasible assignment of x_1, x_2, x_3 the value is smaller than an objective function value with “larger” coefficients:

$$\begin{aligned}
 2x_1 + 4x_2 - 3x_3 & \leq 4x_1 + 10x_2 - 1x_3 \\
 2x_1 + 4x_2 - 3x_3 & \leq 2x_1 + 5x_2 - 3x_3
 \end{aligned}$$

- Idea: Attempt to choose “larger” coefficients in such a way that we obtain a restriction

$$\begin{aligned}
 2x_1 + 4x_2 - 3x_3 & \leq 2x_1 + 6x_2 - 2x_3 \\
 & = 2 \cdot (x_1 + 3x_2 - x_3) \\
 & \stackrel{(I)}{\leq} 2 \cdot 4
 \end{aligned}$$

- 1. bound through (I). restriction

$$2x_1 + 4x_2 - 3x_3 \leq 8$$

- 2. bound through combination of (I). and (II). restriction

$$\begin{aligned}
 2x_1 + 4x_2 - 3x_3 & \leq 2x_1 + 5x_2 - 3x_3 \\
 & = (x_1 + 3x_2 - x_3) + (x_1 + 2x_2 - 2x_3) \\
 & \stackrel{(I)+(II)}{\leq} 4 + 3 = 7.
 \end{aligned}$$

- More general: Linear combination of (I). and (II).

$$2x_1 + 4x_2 - 3x_3 \leq \pi_1 \cdot (x_1 + 3x_2 - x_3) + \pi_2 (x_1 + 2x_2 - 2x_3) \leq 4\pi_1 + 3\pi_2$$

- Whereby the combination of coefficients must meet the following condition:

$$\begin{aligned} & \pi_1(x_1 + 3x_2 - x_3) + \pi_2(x_1 + 2x_2 - 2x_3) \\ = & (\pi_1 + \pi_2)x_1 + (3\pi_1 + 2\pi_2)x_2 + (-\pi_1 - 2\pi_2)x_3 \end{aligned}$$

\Rightarrow

$$\begin{array}{rcl} \pi_1 & + & \pi_2 \geq 2 \\ 3\pi_1 & + & 2\pi_2 \geq 4 \\ -\pi_1 & - & 2\pi_2 \geq -3 \\ \pi_1, \pi_2 & \geq & 0 \end{array}$$

- Best (smallest) upper bound: minimize $4\pi_1 + 3\pi_2$, so that the other restrictions are satisfied
- In other words: solve the LP

$$\begin{array}{ll} \text{min} & 4\pi_1 + 3\pi_2 \\ \text{s.t.} & \pi_1 + \pi_2 \geq 2 \\ & 3\pi_1 + 2\pi_2 \geq 4 \\ & -\pi_1 - 2\pi_2 \geq -3 \\ & \pi_1, \pi_2 \geq 0 \end{array}$$

- In summary

- If an LP is given:

$$\begin{array}{ll} \text{max} & c^\top x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{array}$$

- Search for best upper bound leads to

$$\begin{array}{ll} \text{min} & \pi^\top b \\ \text{s.t.} & \pi^\top A \geq c^\top \\ & \pi \geq 0. \end{array}$$

- This concept is called duality

Definition 140 (Dual Linear Program). The *dual LP* to the maximization problem $\max\{c^\top x \mid \underbrace{Ax \leq b, x \geq 0}_{:=P}\}$ corresponds to the minimization problem $\min\{b^\top \pi \mid \underbrace{A^\top \pi \geq c, \pi \geq 0}_{:=D}\}$. The maximization problem is also referred to as the *primal LP*.

- The dual program can also be set up for LPs in general form

- A dual variable belongs to each primal restriction
- Each primal variable has a dual restriction

- The following conversion rules apply

	primal (P)	dual (D)
Objective	$\max c^\top x$	$\min b^\top \pi$
Constraint (P)/Variable (D)	$a_i^\top x \leq b_i$ $a_i^\top x = b_i$ $a_i^\top x \geq b_i$	$\pi_i \geq 0$ π_i unrestricted $\pi_i \leq 0$
Variable (P)/Constraint (D)	$x_j \geq 0$ x_j unrestricted $x_j \leq 0$	$(A^\top)_j \pi \geq c_j$ $(A^\top)_j \pi = c_j$ $(A^\top)_j \pi \leq c_j$

- **Example:** Dual transformation of an LP

$$\begin{array}{llllllll}
 \text{primal} & & & \text{dual} & & & & \\
 \max & -4x_1 & + & x_2 & \min & 4\pi_1 & + & 3\pi_2 & + & 9\pi_3 \\
 \text{s.t.} & x_1 & - & x_2 & \geq & 4 & & & & = & -4 \\
 & & & & = & 3 & & & & \leq & 1 \\
 & & 3x_1 & + & x_2 & \leq & 9 & & & & \leq & 0 \\
 & & & & x_2 & \leq & 0 & & & & \in & \mathbb{R} \\
 & & & & x_1 & \in & \mathbb{R} & & & & \pi_3 & \geq & 0
 \end{array}$$

- One often speaks of *dual-pairs*, due to the following fact:

Theorem 141. *The dual of the dual is again the primal.*

Proof. Exercise □

- Often dual LPs have a natural interpretation in practice
- **Example:** Diet Problem

Primal

Dual

$$\begin{array}{llllllll}
 \min & 0.19x_A & + & 0.17x_B & \max & 75\pi_K & + & 375\pi_M \\
 \text{s.t.} & 3x_A & + & 0.3x_B & \geq & 75 & \text{V K} & \text{s.t.} & 3\pi_K & + & 5\pi_M & \leq & 0.19 \\
 & 5x_A & + & 30x_B & \geq & 375 & \text{Mag} & & 0.3\pi_K & + & 30\pi_M & \leq & 0.17 \\
 & x_A, x_B & \geq & 0 & & & & & \pi_K, \pi_M & \geq & 0
 \end{array}$$

x_A : apple, x_B : Banana

π_K : Vitamin K, π_M : Magnesium

- Dual: production of m kind of pills each containing one nutrient (Vitamin K, Magnesium, ...)
- Variables π_i : price for nutrient i
- Constraint: the total price of all pills substituting one unit of food j must not exceed the price c_j of one unit of food j
- Objective: maximize the total profit

Basic Properties and Complementary Slackness

- The definition of a dual LP was motivated by obtaining bounds

- Question: Is that true? And are there further relations between primal and dual LP pairs?

Theorem 142 (Weak Duality Theorem). *Let $\max\{c^\top x \mid x \in P\}$ be a primal and $\min\{b^\top \pi \mid \pi \in D\}$ be the corresponding dual problem with $P = \{x \geq 0 \mid Ax \leq b\}$ and $D = \{\pi \geq 0 \mid A^\top \pi \geq c\}$. Then, for $x \in P$ and $\pi \in D$ we obtain $b^\top \pi \geq c^\top x$.*

Proof. Definition of duality

- Let x be a feasible primal solution and π a feasible dual solution for the problems

$$\begin{array}{ll} \max & c^\top x \\ \text{and} & \min \\ Ax & \leq b \\ x & \geq 0 \end{array} \quad \begin{array}{ll} b^\top \pi \\ A^\top \pi & \geq c \\ \pi & \geq 0 \end{array}$$

- Then,

$$\begin{aligned} c^\top x &\stackrel{(a)}{\leq} (A^\top \pi)^\top x = (\pi^\top A) x \\ &= \pi^\top (Ax) \\ &\stackrel{(b)}{\leq} \pi^\top b = b^\top \pi \end{aligned}$$

- (a): $c^\top \leq A^\top \pi$
- (b): $Ax \leq b$.

□

- Duality yields certificates for optimal solutions:

- Let $x \in P, \pi \in D$ with $c^\top x = b^\top \pi$
- $\Rightarrow x$ is optimal for primal LP and π is optimal for dual LP

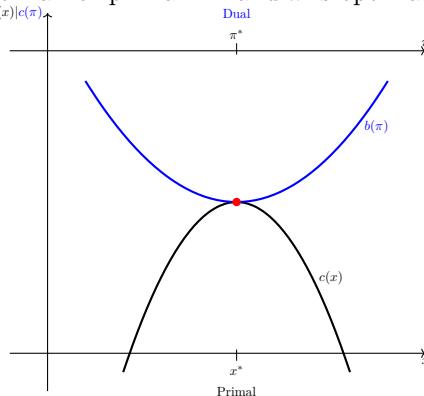


Fig. 8.26.: Considering the objective functions of the primal and the dual problem, they meet at one point: the optimal value of both functions. (No. 703)

- The reverse holds true as well!

Theorem 143 (Strong Duality Theorem). *If the primal LP has a finite optimal solution x^* , then the dual LP also has a finite optimal solution π^* and $c^\top x^* = b^\top \pi^*$.*

Proof. Basic solutions

- Let the primal problem be given in standard form, i.e., $\max \{c^\top x \mid Ax = b, x \geq 0\}$
- Let x be an optimal non-degenerated basic solution to the basis B (we can assume that w.l.o.g.)
- Show: $\pi^\top = c_B^\top A_B^{-1}$ is an optimal solution of the dual problem

1. Feasibility

- Consider the reduced cost $\bar{c}^\top = c^\top - c_B^\top A_B^{-1} A$ (Theorem ??)
- For the reduced costs, we have $\bar{c} \leq 0$ with $\bar{c}_j = 0$ for $j \in B$ (Theorem ??)
- \Rightarrow

$$0 \geq \bar{c}^\top = c^\top - c_B^\top A_B^{-1} A = c^\top - \pi^\top A$$

- $\Rightarrow A^\top \pi \geq c^\top$, i.e., π is feasible solution

2. Objective function values:

$$b^\top \pi = \pi^\top b \stackrel{(a)}{=} \pi^\top A_B x_B \stackrel{(b)}{=} c_B^\top x_B = c^\top x$$

- (a): $A_B x_B = b$ per definition of x
 (b): $\pi^\top A_B = c_B^\top$ per definition of π

□

• **Example:**

Primal	Dual
$\max \quad 3x_1 + 2x_2$ $2x_1 + 1x_2 \leq 22$ $1x_1 + 2x_2 \leq 23$ $4x_1 + x_2 \leq 40$ $x_1, x_2 \geq 0$	$\min \quad 22\pi_1 + 23\pi_2 + 40\pi_3 \geq 3$ $2\pi_1 + 1\pi_2 + 4\pi_3 \geq 2$ $1\pi_1 + 2\pi_2 + \pi_3 \geq 0$ $\pi_1, \pi_2, \pi_3 \geq 0$

- Solution for primal $x = (7, 8, 0, 0, 0)$ and for dual $\pi = (\frac{4}{3}, \frac{1}{3}, 0, 0)$.
- Optimal?
- Remember: every LP has three options
 1. to terminate with an optimal solution
 2. to be infeasible
 3. to have an unbounded objective
- For primal-dual pairs not all combinations are possible
- Using this theorem, we can easily deduce criteria when no feasible solution exists for either the primal or the dual program

Corollary 144. Let $\max\{c^\top x \mid x \in P\}$ be a primal and $\min\{b^\top \pi \mid \pi \in D\}$ be the corresponding dual problem with $P = \{x \geq 0 \mid Ax \leq b\}$ and $D = \{\pi \geq 0 \mid A^\top \pi \geq c\}$.

1. Let $P \neq \emptyset$. If the value $c^\top x$ is unbounded then $D = \emptyset$.
2. Let $D \neq \emptyset$. If value $b^\top \pi$ is unbounded then $P = \emptyset$.

Proof. Consequence of Theorem 142

□

- **Remark:** Both, the primal and dual LP, can be infeasible.

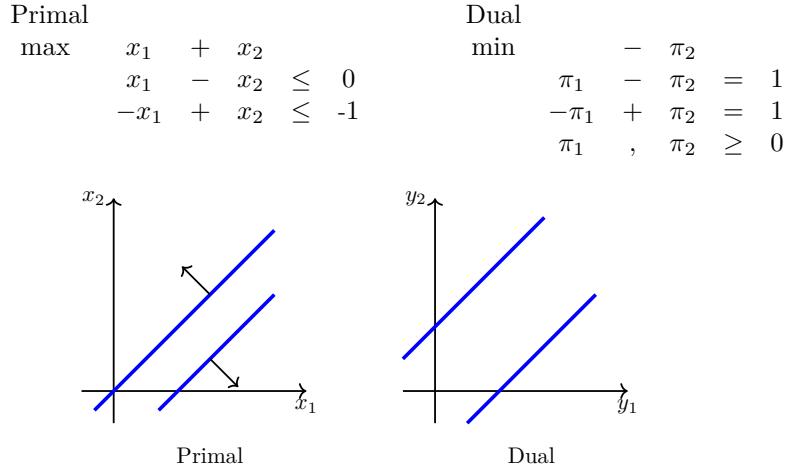


Fig. 8.27.: The feasible regions for both problems are empty. (No. 702)

- In summary, one obtains the following relationship between primal and dual programs

Primal\Dual	optimal solution	unbounded	infeasible
optimal solution	$c^\top x^* = b^\top \pi^*$	no	no
unbounded	no	no	yes
infeasible	no	yes	yes

- Question: Can we decide besides evaluating the objective whether a solution is optimal?

Theorem 145 (Complementary Slackness Theorem). *Let x be a feasible primal solution and π a feasible dual solution. Then x and π are optimal if and only if*

$$\begin{aligned} \pi_i(b_i - a_i^\top x) &= 0 & \forall i \in \{1, \dots, m\} \\ (c_j - A_j^\top \pi)x_j &= 0 & \forall j \in \{1, \dots, n\}. \end{aligned}$$

Proof. Use strong duality

- Set $u_i := \pi_i(a_i^\top x - b_i)$ $\forall i = 1, \dots, m$ and $v_j = (c_j - A_j^\top \pi)x_j$ $\forall j = 1, \dots, n$
- *Claim:* $u_i \geq 0$ and $v_j \geq 0$
Proof of Claim
 - If $a_i^\top x - b_i = 0$ in the primal $\Rightarrow u_i = 0$
 - If $a_i^\top x - b_i \geq 0$ in the primal \Rightarrow the dual variable $\pi_i \geq 0 \Rightarrow u_i \geq 0$
 - If $a_i^\top x - b_i \leq 0$ in the primal \Rightarrow the dual variable $\pi_i \leq 0 \Rightarrow u_i \geq 0$
 - If x_j is unconstrained $\Rightarrow \pi^\top A_j = c_j$ in the primal $\Rightarrow v_j = 0$
 - If $x_j \geq 0 \Rightarrow \pi^\top A_j \geq c_j$ in the dual $\Rightarrow v_j \geq 0$

- If $x_j \leq 0 \Rightarrow \pi^\top A_j \leq c_j$ in the dual $\Rightarrow v_j \leq 0$ □C

- Consider

$$\begin{aligned}
 \sum_{j=1}^n v_j + \sum_{i=1}^m u_i &= \sum_{j=1}^n (c_j - A_j^\top \pi) x_j + \sum_{i=1}^m \pi_i (a_i^\top x - b_i) \\
 &= \sum_{j=1}^n c_j x_j - \sum_{i=1}^m \pi_i b_i - \sum_{j=1}^n \pi^\top A_j x_j + \sum_{i=1}^m \pi_i a_i^\top x \\
 &= c^\top x - \pi^\top b - \pi^\top (Ax) + (\pi^\top A)x \\
 &= c^\top x - \pi^\top b
 \end{aligned}$$

- (\Rightarrow): If x and π are optimal
- $\Rightarrow \pi^\top b = c^\top x$ (strong duality)
- $\Rightarrow \sum_{i=1}^m u_i + \sum_{j=1}^n v_j = 0$
- Since $u_i \geq 0$ and $v_j \geq 0 \Rightarrow u_i = 0$ and $v_j = 0$, i.e., the condition holds
- (\Leftarrow): If $u_i = 0 = v_j \forall i = 1, \dots, n, j = 1, \dots, m$
- $\Rightarrow -\pi^\top b + c^\top x = 0$
- $\Rightarrow x$ and π are optimal (weak duality)

□

- **Example:** Optimality?
- **Note:** The complementary slackness conditions are more relevant if we consider $\max \{c^\top x \mid Ax \leq b, x \geq 0\}$ and the dual $\min \{b^\top \pi \mid A^\top \pi \geq c, \pi \geq 0\}$
- Then it follows:
 - If in the dual optimum solution $\pi_i \neq 0 \Rightarrow b_i - a_i^\top x = 0 \Rightarrow a_i^\top x = b$
 - If in the primal optimum solution $x_j \neq 0 \Rightarrow \pi^\top A_j - c_j = 0 \Rightarrow \pi^\top A_j = c_j \Rightarrow j$ -th restriction in dual LP is active
- Complementary slackness provides simple necessary and sufficient conditions for optimality of pairs of primal feasible and dual feasible solutions
- They are often used in the design of algorithms (primal-dual algorithms, primal-dual approximation algorithms)



9. Basics of Integer Programming

Literature

- Laurence A. Wolsey Chvátal
Integer Programming. Wiley, 1998

9.1. Motivation, Definitions and Modeling Power

- In linear programming, every decision can be fractional, i.e., can take an arbitrary amount as $x_A = 0.5$ or $x_A = 0.2321$
- In practice, decisions often have to take integer values
- **Example:** 0-1 Knapsack Problem

Definition 146. 0-1 Knapsack Problem

Given:

- n items with profit $p_i \in \mathbb{N}$ and weight $w_i \in \mathbb{N}$, $i \in \{1, \dots, n\}$
- capacity of the knapsack c

Find:

- choose items without exceeding the capacity
- maximize the total profit

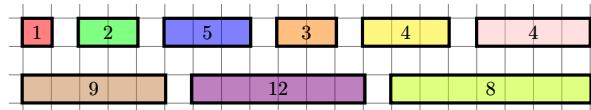


Fig. 9.1.: Knapsack Problem (No. 255)

- Formulation as linear program with integer variables
- Variables
 - $x_i \in \{0, 1\}$ with $x_i = 1$ if item i is taken
- Constraint
 - The knapsack capacity is not exceeded

$$\sum_{i=1}^n w_i x_i \leq c$$

- Objective function
 - maximize the profit

$$\max \sum_{i=1}^n p_i x_i$$

- To include these kind of decisions, we use linear constraints but require some variables to be integer

Definition 147 ((Linear) Mixed Integer Program (MIP)). Let $A^1 \in \mathbb{R}^{m \times n^1}$, $A^2 \in \mathbb{R}^{m \times n^2}$, $b \in \mathbb{R}^m$, $c^1 \in \mathbb{R}^{n^1}$, $c^2 \in \mathbb{R}^{n^2}$ and $x^1 \in \mathbb{R}^{n^1}$ and $x^2 \in \mathbb{R}^{n^2}$. Then

$$\begin{aligned} \max \quad & c^{1\top} x^1 + c^{2\top} x^2 \\ \text{subject to} \quad & A^1 x^1 + A^2 x^2 \leq b \\ & x^1 \geq 0 \\ & x^2 \geq 0 \text{ and integer} \end{aligned}$$

is a *mixed integer program*.

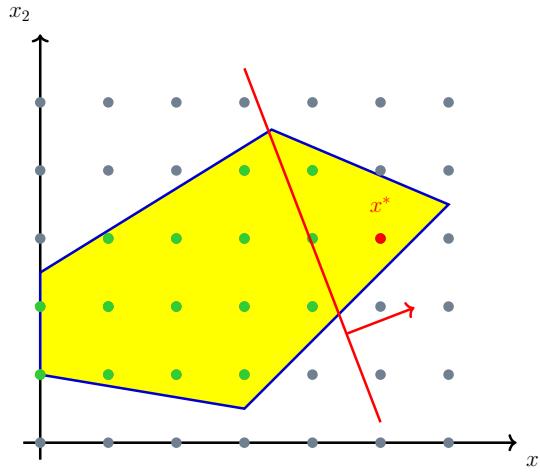


Fig. 9.2.: Mixed integer program (No. 708)

- Special cases:
 - *Integer Program (IP)*: all variables are integer
 - *Binary Integer Program (BIP)*: all variables are restricted to 0/1 values; a variable $x \in \{0, 1\}$ is called a *binary variable*
- For simplicity, we only consider integer programs
- MIPs enable us to model a huge variety of practical problems

Assignment Problem

Definition 148. Assignment Problem

Given:

- n people
- n jobs
- everyone is assigned exactly to one job
- c_{ij} if person i is assigned to job j

Find:

- minimum cost assignment

worker i cost c_{ij} job j

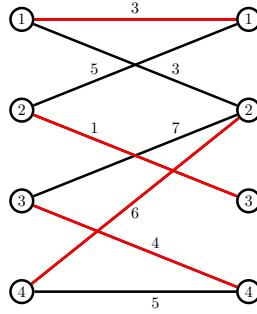


Fig. 9.3.: Assignment Problem (No. 712)

- Formulation as BIP

- Variables

- $x_{ij} = 1$ if person i does job j , $x_{ij} = 0$ otherwise, $i = 1, \dots, n$, $j = 1, \dots, n$

- Constraints

- Each person i does one job

$$\sum_{j=1}^n x_{ij} = 1 \quad i = 1, \dots, n$$

- Each job j is done by one person

$$\sum_{i=1}^n x_{ij} = 1 \quad j = 1, \dots, n$$

- Objective function

- minimize the cost

$$\min \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij}$$

Stable Set Problem

Definition 149. Weighted Stable Set Problem

Given:

- graph $G = (V, E)$
- weights $c_v \forall v \in V$

Find: Choose a stable set $S \subseteq V$ with maximum weight

$$c(S) := \sum_{v \in S} c_v.$$

A *stable set* is a set of vertices no two of which are adjacent.

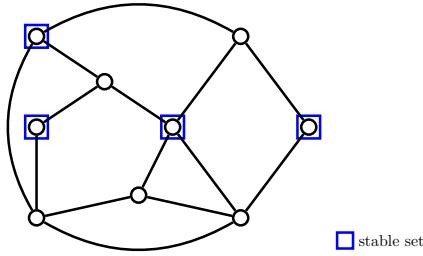


Fig. 9.4.: Stable Set Problem (No. 636)

- Formulation as BIP
 - Variables
 - $x_v = 1$ if v is chosen, $x_v = 0$ otherwise $\forall v \in V$
 - Constraints
 - No two vertices are adjacent

$$x_v + x_w \leq 1 \quad \forall vw \in E$$

- Objective
 - maximize the weight

$$\max \sum_{v \in V} c_v x_j$$

The Traveling Salesperson Problem (TSP)

Definition 150. Traveling Salesperson Problem

Given:

- salesperson must visit each of n cities exactly once
- return to the starting point
- c_{ij} time for traveling from city i to city j

Find:

- choose the order of cities visited
- finish tour as quickly as possible

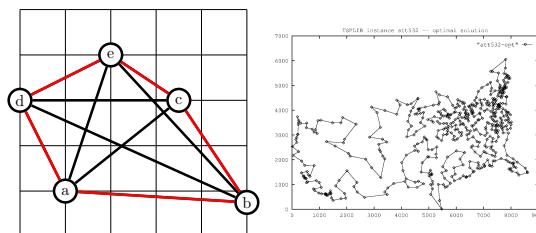


Fig. 9.5.: Traveling Salesperson Problem (No. 529)

- Further applications
 - truck driver, with list of clients
 - machine place modules on printed circuits boards
 - crane picks up and depose crates
- Formulation as BIP (first attempt)

- Variables
 - $x_{ij} = 1$ if salesperson travels directly from city i to city j , $x_{ij} = 0$ otherwise
- Constraints
 - Leave every city i exactly once

$$\sum_{\substack{j=1 \\ j \neq i}}^n x_{ij} = 1 \quad \forall i = 1, \dots, n$$

- Arrive at every city j exactly once

$$\sum_{\substack{i=1 \\ i \neq j}}^n x_{ij} = 1 \quad \forall j = 1, \dots, n$$

- Objective
 - minimize total travel time

$$\min \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij}$$

- Problem
 - Subtours
 - \Rightarrow need to add additional constraints

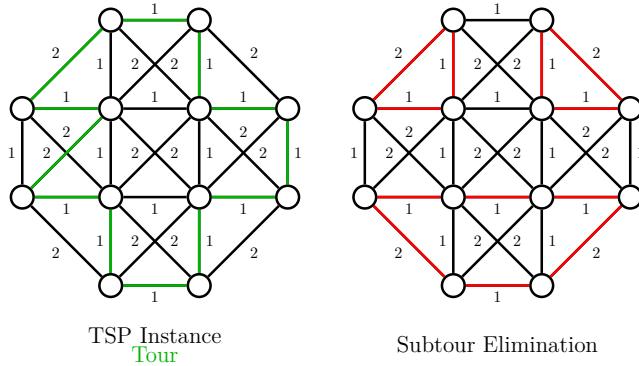


Fig. 9.6.: Subtours (No. 799)

1. Cut-set constraints

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \geq 1 \quad \forall S \subsetneq \{1, \dots, n\}, S \neq \emptyset$$

2. Subtour elimination constraints

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subsetneq \{1, \dots, n\}, 2 \leq |S| \leq n - 1$$

- One of these types of constraints need to be added
- Besides these many applications, some aspects can easily be integrated

Simple Selections

- Variables $x_i \in \{0, 1\}$: $x_i = 1$ chose element $i \in \{1, \dots, n\}$

1. Upper bound k on selection

- Add the constraint

$$\sum_{i=1}^n x_i \leq k$$

2. Lower bound k on selection

- Add the constraint

$$\sum_{i=1}^n x_i \geq k$$

3. Exactly one element is chosen

- Add the constraint

$$\sum_{i=1}^n x_i = 1$$

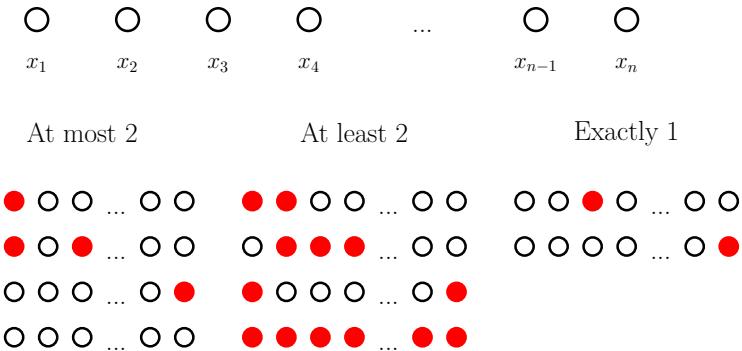


Fig. 9.7.: Simple Selections (No. 800)

Modeling Fixed Costs

- Typical nonlinear fixed charge cost function:

$$h(x) = \begin{cases} f + px & \text{if } 0 < x \leq C \\ 0 & \text{if } x = 0 \end{cases}$$

$$f > 0, p > 0$$

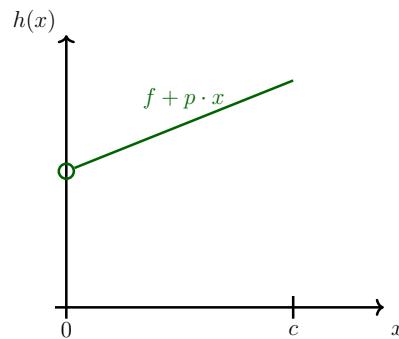


Fig. 9.8.: Fixed Cost (No. 801)

- Formulation in MIP
 - Variable
 - $y = 1$ if $x > 0$ and $y = 0$ otherwise
 - Constraints
 - If $x > 0$, pay the fix price, i.e., $y = 1$

$$x \leq C \cdot y$$

- Objective
 - Replace $h(x)$ by

$$h(x) = f \cdot y + p \cdot x$$

Dependencies between variables

- Let $y_1, y_2, y_3 \in \{0, 1\}$
 1. If $y_1 = 1 \Rightarrow y_2 = 1$
 - linear formulation: $y_2 \geq y_1$
 2. $y_3 = 1$ if and only if $y_1 = 1$ and $y_2 = 1$
 - nonlinear formulation: $y_3 = y_1 \cdot y_2$
 - linear formulation:

$$\begin{aligned} -y_1 + y_3 &\leq 0 \\ -y_2 + y_3 &\leq 0 \\ y_1 + y_2 - y_3 &\leq 1 \end{aligned}$$
 3. $y_3 = 1$ if and only if $y_1 = 1$ or $y_2 = 1$ (but not both)
 - linear formulation:

$$\begin{aligned} y_1 + y_2 + y_3 &\leq 2 \\ y_1 - y_2 - y_3 &\leq 0 \\ -y_1 + y_2 - y_3 &\leq 0 \\ -y_1 - y_2 + y_3 &\leq 0 \end{aligned}$$

Discrete Alternatives or Disjunctions

- Let $x \in \mathbb{R}^n$, $0 \leq x \leq u$.
- Model: either $a^1 x \leq b_1$ or $a^2 x \leq b_2$ hold

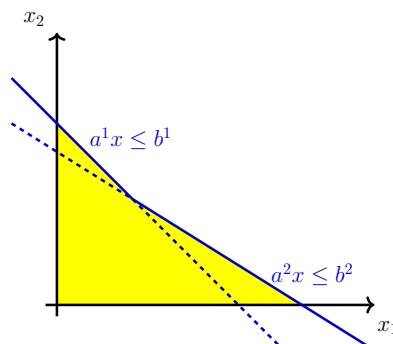


Fig. 9.9.: Either/or constraints (No. 802)

- Formulation in MIP

- Variables

- $y_i = 1$, if $a^i x \leq b_i$ holds, $i = 1, 2$; $y_i = 0$ otherwise

- Constraint

- One of the two options is chosen

$$y_1 + y_2 = 1$$

- The corresponding constraint holds

$$a^i x - b_i \leq M(1 - y_i) \quad \forall i = 1, 2$$

with $M = \max\{a^i x - b_i \mid 0 \leq x \leq u\}$

- Typical application

- Scheduling: job 1 before job 2 or the other way around

9.2. IPs, LPs and integer Polytopes

- In the last sections, we learned a lot about linear programs
- Can we use the knowledge about LPs to solve IPs? Maybe, IPs and LPs are the same?
- What if we just relax the integrality condition

Definition 151 (Linear Programming (LP) Relaxation). For the integer program $\max\{c^\top x \mid x \in P \cap \mathbb{Z}^n\}$ with $P = \{x \in \mathbb{R}_+^n \mid Ax \leq b, x \geq 0\}$, the *linear programming relaxation* is the linear program $z^{LP} = \max\{c^\top x \mid x \in P\}$.

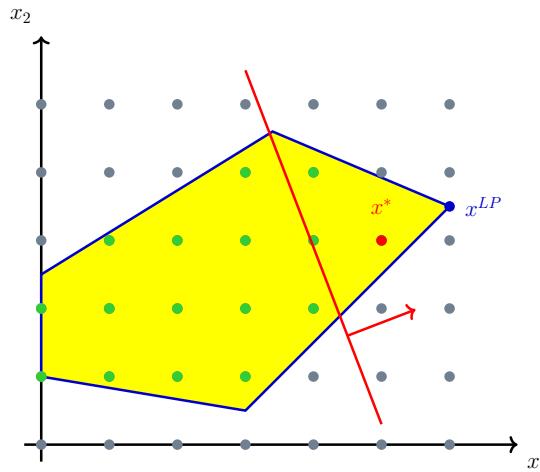


Fig. 9.10.: LP provides an upper bound (No. 709)

- An optimal solution of an LP relaxation always provides an upper bound for the IP

Lemma 152. Let

$$z^{IP} = \max\{c^\top x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}\} \text{ and } z^{LP} = \max\{c^\top x \mid Ax \leq b, x \geq 0\}.$$

Then $z^{LP} \geq z^{IP}$.

Proof.

- Let x^* be an optimal solution for z^{IP}

- $\Rightarrow x^*$ is a feasible solution for z^{LP}

- $\Rightarrow z^{LP} \geq c(x^*) = z^{IP}$

□

- Idea: Relax integrality and round

- **Example:** Relaxing and rounding is in general a bad idea

- Consider

$$\begin{array}{llll} \max & 1x_1 & + & 0.64x_2 \\ & 50x_1 & + & 31x_2 & \leq & 250 \\ & 3x_1 & - & 2x_2 & \geq & -4 \\ & x_1, & x_2 & \geq & 0 & \text{and integer} \end{array}$$

- optimal LP solution: $x_1^{LP} = 376/193, x_2^{LP} = 950/193$

- optimal IP solution: $x_1^* = 5, x_2^* = 0$

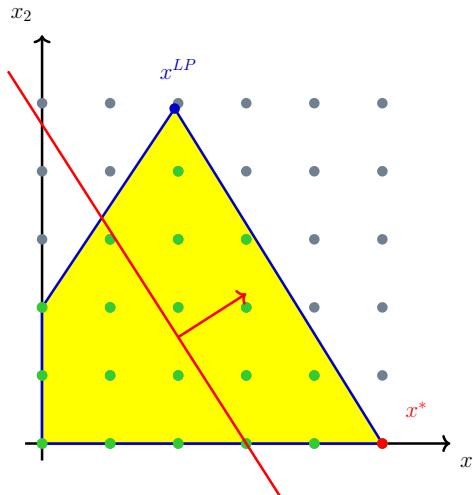


Fig. 9.11.: LP solution far away from IP solution (No. 710)

- Consider the *maximum matching problem*:

Problem:

BIP:

Given: graph $G = (V, E)$

$$\max \sum_{e \in E} x_e$$

Find: $M \subseteq E$ with $|M \cap \delta(v)| \leq 1$
 $\forall v \in V$ and $|M|$ maximum

$$\sum_{e \in \delta(v)} x_e \leq 1 \quad \forall v \in V$$

$$x_e \in \{0, 1\} \quad \forall e \in E$$

- $(0.5, 0.5, \dots, 0.5)$ is an optimal LP solution
- \Rightarrow no information for the original problem
- Rounding leads to infeasible solutions

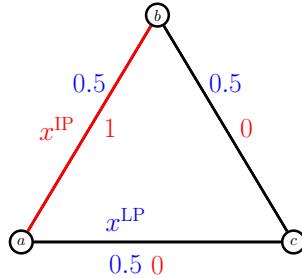
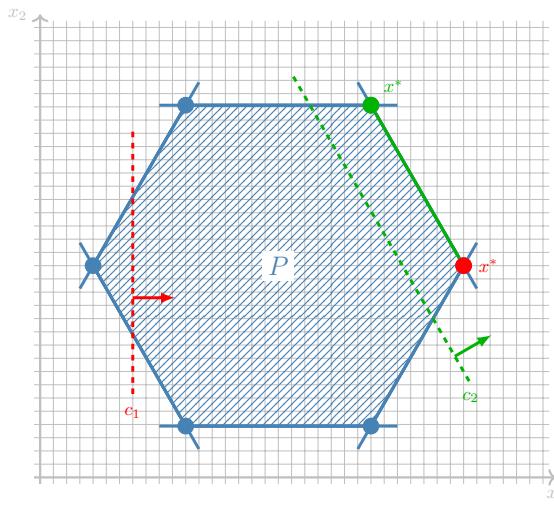


Fig. 9.12.: Maximum Matching (No. 711)

- There are several methods - heuristics and approximation algorithms - that use iterative rounding to obtain good solutions
- Furthermore, there are special cases, in which we can solve the linear programming relaxation instead of the integer program
- LPs like $\max\{c^\top x \mid Ax \leq b, x \geq 0\}$ are great and we can solve them easily
- In order to use that we want to consider polytopes that always guarantee to have an integer optimal solution independent of the cost vector

Definition 153. A polyhedron P is called *integer* if, for all $c \in \mathbb{R}^n$ with $\max\{c^\top x \mid x \in P\}$ finite, the maximum is attained by some integer point x .

Fig. 9.13.: P is integer (No. 728)

- We know that an optimal solution is always attained at a vertex
- \Rightarrow if all vertices are integer, we can solve the LP instead of the IP
- The following we show that integer vertices and integer polytopes are the same

Lemma 154. Let $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ be a polytope with $A \in \mathbb{R}^{m \times n}$ of rank m . Then P is integer if and only if all vertices of P are integer.

Proof. Properties of LPs

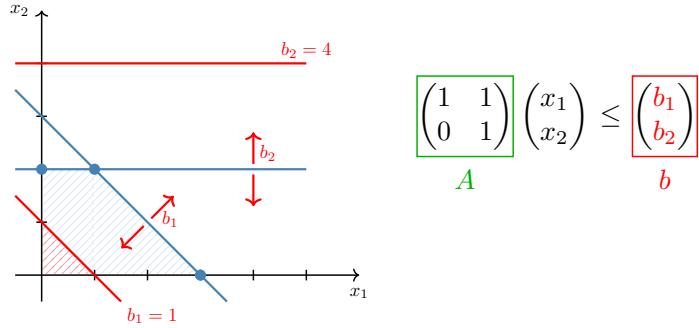
- (\Rightarrow): Let x^* be a vertex of P
- $\Leftrightarrow \exists$ basis $B \subseteq \{1, \dots, n\}$ of the induced vector space by A such that $A_B x_B^* = b$ and $x_N^* = 0$ for $N = \{1, \dots, n\} \setminus B$, i.e., a basis of A
- Define $c_B = 0$ and $c_N = -1$
- $\Rightarrow x_B^*$ is an optimal solution of $\max\{c^\top x \mid x \in P\}$
- Let y be another solution of $\max\{c^\top x \mid x \in P\}$
- $\Rightarrow \exists j \in N$ with $y_j > 0$
- $\Rightarrow c^\top y < 0 = c^\top x^*$
- $\Rightarrow x^*$ is the unique optimal solution of $\max\{c^\top x \mid x \in P\}$
- If P is integer $\Rightarrow x^*$ is integer
- (\Leftarrow): Let $\max\{c^\top x \mid x \in P\}$ be an LP
- Every LP obtains an optimal solution in a vertex
- Since all vertices of P are integer \Rightarrow the maximum is attained by some integer vector x

□

- Thus, if we need to solve an IP over an integer polytope, we can instead solve its linear relaxation

Corollary 155. Let $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ be an integer polytope and let $c \in \mathbb{R}^n$ be some cost vector. Let x^* be an optimal basic solution of the linear program $\max\{c^\top x \mid x \in P\}$, then x^* is an optimal solution of the integer program $\max\{c^\top x \mid x \in P \cap \mathbb{Z}^n\}$.

- Thus, integer polytopes are exactly what we are looking for
- A polytope is always defined by two components:
 - The matrix A defining the constraints
 - The right hand side b
- We will set the focus in the following on matrices that always define an integer polytope as long as b is integer
- Can we find some properties for the matrix A that guarantees that only integer basic solutions exist?

Fig. 9.14.: Polytopes characterized by matrix A (No. 1177)

9.3. Totally unimodular Matrices

- Let us start with $A \in \mathbb{Z}^{n \times n}$ and let us assume A is non-singular
- Let $\bar{x} = A^{-1}b$ be the unique solution solving the linear equations
- Then, due to Cramer's Rule:

$$|\bar{x}_j| = \frac{|\det(A_b^j)|}{|\det(A)|}$$

with A_b^j is the matrix formed by replacing the j th column A_j of A by the column vector b

$$\begin{array}{c} j \\ \boxed{\begin{array}{cccc} a_{1,1} & \dots & a_{1,j} & \dots & a_{1,n} \\ \vdots & & \vdots & & \vdots \\ a_{i,1} & \ddots & a_{i,j} & \ddots & a_{i,n} \\ \vdots & & \vdots & & \vdots \\ a_{n,1} & \dots & a_{n,j} & \dots & a_{n,n} \end{array}} \\ A \end{array} \quad \begin{array}{c} j \\ \boxed{\begin{array}{ccccc} a_{1,1} & \dots & b_1 & \dots & a_{1,n} \\ \vdots & & b_2 & & \vdots \\ a_{i,1} & \ddots & \vdots & \ddots & a_{i,n} \\ \vdots & & \vdots & & \vdots \\ a_{n,1} & \dots & b_n & \dots & a_{n,n} \end{array}} \\ A_b^j \end{array}$$

Fig. 9.15.: Replace A_j by b (No. 731)

- Our goal: guarantee $\bar{x}_j \in \mathbb{Z}$
- $\det(A_b^j) \in \mathbb{Z}$ if $b \in \mathbb{Z}$
- $\Rightarrow \text{require } \det(A) \in \{-1, 1\}$

Definition 156. An $n \times n$ matrix A is called *unimodular* if all entries are integer and its determinant equals $+1$ or -1 .

$$\begin{array}{ll} \det \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right) = 1 & \det \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 1 & 1 \end{pmatrix} \right) = 2 \\ \det \left(\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \end{pmatrix} \right) = 0 & \det \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 1 & -1 \end{pmatrix} \right) = -1 \end{array}$$

- For these matrices, our goal is obtained: we always get an integer solution for any $b \in \mathbb{Z}^n$

Lemma 157. Let $A \in \mathbb{Z}^{n \times n}$ be an integer non-singular matrix. Then $A^{-1}b$ is integer for each integer vector $b \in \mathbb{Z}^n$ if and only if A is unimodular.

Proof. Definition of unimodular and Cramer's Rule

- (\Leftarrow): Since A is non-singular, there exists a unique solution \bar{x} solving $A\bar{x} = b$
- Due to Cramer's rule,

$$|\bar{x}_j| = \frac{|\det(A_b^j)|}{|\det(A)|} = |\det(A_b^j)|$$

with A_b^j is the matrix formed by replacing the j th column A_j of A by the column vector b

$$A \quad A_b^j$$

Fig. 9.16.: Replace A_j by b (No. 731)

- Since the determinant of an integer matrix is integer, $\det(A_b^j) \in \mathbb{Z}$
- $\Rightarrow \bar{x} \in \mathbb{Z}^n$
- (\Rightarrow): Consider $A^{-1}e_j$, where e_j is the unit vector with 1 at the j th position and 0 otherwise, $j \in \{1, \dots, n\}$
- Due to the assumption, $A^{-1}e_j = (A^{-1})_j \in \mathbb{Z}^n$
- $\Rightarrow A^{-1}$ is integer
- $\Rightarrow \det(A^{-1})$ is integer
- Since $A \in \mathbb{Z}^{n \times n}$, $\det(A)$ is integer
- Furthermore, $\det(A^{-1}) \cdot \det(A) = \det(A^{-1} \cdot A) = 1$
- $\Rightarrow \det(A) \in \{-1, 1\}$, i.e., A is unimodular

□

- However, in general we have $A \in \mathbb{Z}^{m \times n}$
- Extend the concept in a first basic way

Definition 158. A matrix $A \in \mathbb{Z}^{m \times n}$ with $\text{rank}(A) = m$ is *unimodular* if for each basis B of A it holds $\det(B) \in \{-1, 1\}$.

- For this kind of matrices, we can prove that LPs in standard form with a unimodular matrix always have integer solutions

Theorem 159 (Veinott/Dantzig, 1968). *Let $A \in \mathbb{Z}^{m \times n}$ be a matrix with $\text{rank}(A) = m$. Then $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ is integer for every integer vector $b \in \mathbb{Z}^m$ if and only if A is unimodular.*

Proof. Definition unimodular and Lemma 157

- (\Leftarrow): Let A be unimodular
- Let $b \in \mathbb{Z}^m$ and \bar{x} a vertex of P
- Since \bar{x} is a vertex, \exists basis B with $\bar{x}_B = B^{-1}b$
- Since A is unimodular, $\det(B) \in \{-1, 1\}$ and thus B is unimodular
- $\Rightarrow \bar{x}_B = B^{-1}b$ is integer (Lemma 157)
- (\Rightarrow): Let B be a basis of A
- W.l.o.g. $B = \{1, \dots, m\}$
- Prove: $\det(B) \in \{-1, 1\}$
- Let $v \in \mathbb{Z}^m$ and let y be integer with $y + B^{-1}v \geq 0$, $y \in \mathbb{Z}^m$
- Set $\bar{z} = y + B^{-1}v$ and $b := B \cdot \bar{z} = B(y + B^{-1}v) = \underbrace{By}_{\text{integer}} + \underbrace{v}_{\text{integer}} \Rightarrow b \in \mathbb{Z}^m$
- Now extend \bar{z} to a vector $\bar{x} = \begin{pmatrix} \bar{z} \\ 0 \end{pmatrix} \in \mathbb{R}^n$
- $\Rightarrow A\bar{x} = B(y + B^{-1}v) = b$, $\bar{x} \geq 0$
- $\Rightarrow \bar{x} \in P$ and basic solution of B
- $\Rightarrow \bar{x}$ is a vertex of P
- Due to our assumption, \bar{x} is integer
- and $\bar{x}_B = y + B^{-1}v$
- Since y is integer, $B^{-1}v$ is integer
- $\Rightarrow \det(B) \in \{-1, 1\}$ (Lemma 157)

□

- Thus, for unimodular matrices, we can solve the LP relaxation instead of the IP

Corollary 160. *Let $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ be defined by a unimodular matrix $A \in \mathbb{Z}^{m \times n}$ and let $c \in \mathbb{R}^n$ be some cost vector. Let x^* be an optimal basic solution of the linear program $\max\{c^\top x \mid x \in P\}$, then x^* is an optimal solution of the integer program $\max\{c^\top x \mid x \in P \cap \mathbb{Z}^n\}$.*

- In order to extend this theorem to IPs in canonical form, i.e., $\max\{c^\top x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\}$, we need to extend the definition of unimodularity

Definition 161 (totally unimodular (TUM)). An $m \times n$ matrix A is called *totally unimodular (TUM)* if every square, non-singular submatrix is unimodular.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ TUM} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 1 & 1 \end{pmatrix} \text{ not TUM} \quad \begin{pmatrix} -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & -1 & 1 \end{pmatrix} \text{ not TUM}$$

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 0 & 1 \end{pmatrix} \text{ TUM} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & 1 & -1 \end{pmatrix} \text{ not TUM}$$

- Alternative definition: every square submatrix has determinant -1 , 0 , or 1 .
- First properties of totally unimodular matrices are the following:

Corollary 162. Let A be an $m \times n$ matrix that is TUM. Then:

- A has only entries $a_{ij} \in \{-1, 0, 1\}$, $i = 1, \dots, m$, $j = 1, \dots, n$.
- Every submatrix of A is TUM.
- A^\top is TUM.
- $-A$ is TUM.

- Since a matrix that is TUM is also unimodular, all properties for LPs in standard form transfer

Lemma 163. Let $A \in \mathbb{Z}^{m \times n}$ be TUM, $b \in \mathbb{Z}^m$ and $c \in \mathbb{R}^n$. Then,

- Every vertex of the polyhedron $P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ is integer.
- Let x^* be an optimal basic solution of the linear program $\max\{c^\top x \mid x \in P\}$. Then, x^* is an optimal solution of the integer program $\max\{c^\top x \mid x \in P \cap \mathbb{Z}^n\}$.

Proof. If A is TUM, then A is unimodular □

- However, we started to consider linear programs in canonical form, i.e., $P = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$
- Similar to Veinott/Dantzig, Hoffmann and Kruskal proved the relation between TUM matrices and integer polytopes

Theorem 164 (Hoffmann-Kruskal, 1956). Let $A \in \mathbb{Z}^{m \times n}$. Then A is totally unimodular if and only if for all $b \in \mathbb{Z}^m$ the polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0\}$ is integer.

Sketch of proof. Theorem of Veinott/Dantzig 159

P is integer for every vector $b \in \mathbb{Z}^m$

- $\Leftrightarrow \bar{P} := \{z \in \mathbb{R}^{n+m} \mid (A, I_m)z = b, z \geq 0\}$ is integer for every vector $b \in \mathbb{Z}^m$ where I_m being the $m \times m$ identity matrix (Exercise)
- $\Leftrightarrow (A, I_m)$ is unimodular (Theorem Veinott/Dantzig 159)
- $\Leftrightarrow A \in \mathbb{Z}^{m \times n}$ is totally unimodular (Exercise, Laplacian expansion)

□

- The complexity of recognizing TUM matrices has been open for a long time
- In 1980, Seymour solved the problem by proving a “Decomposition Theorem”
- This theorem states that every TUM matrix can be constructed from simple TUM matrices
- The corresponding algorithm has a runtime of $\mathcal{O}((n+m)^4m)$

Sufficient criteria for totally unimodular Matrices

- The algorithm of Seymour is complicated
- Often simple criteria suffice to show that a matrix is TUM

Theorem 165. Let $A \in \mathbb{Z}^{m \times n}$ with $a_{ij} \in \{-1, 0, 1\}$ and the following two properties:

1. Every column A_j contains at most two coefficients $a_{ij} \neq 0$
2. For every column $j = 1, \dots, n$, it holds

$$-1 \leq \sum_{i=1}^m a_{ij} \leq 1.$$

Then A is TUM.

- According to the first property: $\sum_{i=1}^m a_{ij} \in \{-2, -1, 0, 1, 2\}$

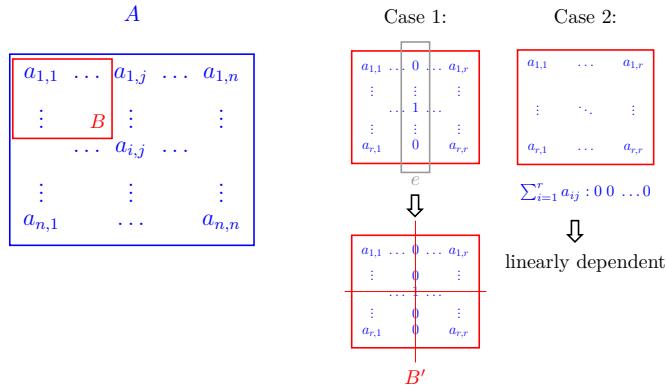
- The second property rules out -2 and 2

- **Example:** TUM?

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 & 1 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & -1 & -1 \\ 0 & 0 & -1 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \end{pmatrix}$$

Proof. Induction on the size of submatrices

- I.B.: Let B be a 1×1 submatrix of $A \Rightarrow \det(B) \in \{-1, 0, 1\}$
- I.H.: For every $\ell \times \ell$ submatrix B of A with $\ell \leq r-1$, $\det(B) \in \{-1, 0, 1\}$
- I.S.: Let B be an $r \times r$ submatrix of A , w.l.o.g., $B = (a_{ij})_{1 \leq i, j \leq r}$

Fig. 9.17.: $r \times r$ submatrix B (No. 729)

- *Case 1:* B contains a unit vector (or its negative)
- Let w.l.o.g. e be a column of B and a unit vector (or its negative)
- Laplacian expansion along e :

$$\det(B) = \pm 1 \cdot \det(B')$$

with B' being the corresponding submatrix of B

- B' is a $(r-1) \times (r-1)$ submatrix of A
- $\stackrel{I.H.}{\Rightarrow} \det(B') \in \{-1, 0, 1\}$
- $\Rightarrow \det(B) \in \{-1, 0, 1\}$
- *Case 2:* B does not contain a unit vector (or its negative)
- $\Rightarrow \sum_{i=1}^r a_{ij} = 0$ for all $j = 1, \dots, r$
- \Rightarrow the rows are linearly dependent $\Rightarrow \det(B) = 0$

□

- From this theorem, it is easy to conclude

Corollary 166. Let A be a matrix with entries $a_{ij} \in \{-1, 0, 1\}$.

1. If A is TUM, then adding a unit vector (or its negative) as new column leads again to a matrix that is TUM.
2. If A contains at most one $+1$ or -1 in every column, then A is TUM.

- A little more complicated, but a more powerful criterion is the following

Theorem 167 (Hoffman, 1956). Let A be a matrix with entries $a_{ij} \in \{-1, 0, 1\}$. Then A is TUM if both conditions hold

1. Every column A_j contains at most two coefficients $a_{ij} \neq 0$

2. The rows of A can be partitioned into two disjoint sets R_1 and R_2 s.t.
- If two non-zero entries in a column of A have the same sign, then the one row is in R_1 and the other in R_2 .
 - If two non-zero entries in a column of A have opposite signs, then the rows are both in R_1 or both in R_2 .

Proof. Exercise □

- **Example:** Theorem of Hoffman

$$\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \left(\begin{array}{ccc} * & * & 1 \\ * & * & -1 \\ & -1 & \\ * & -1 & \end{array} \right) \Rightarrow R_1 = \{1, 2, 3\}, R_2 = \{4\}$$

- For TUM matrices, the polytope is integer independent of the right hand side b
- However, a polytope might be integer but the corresponding matrices not TUM

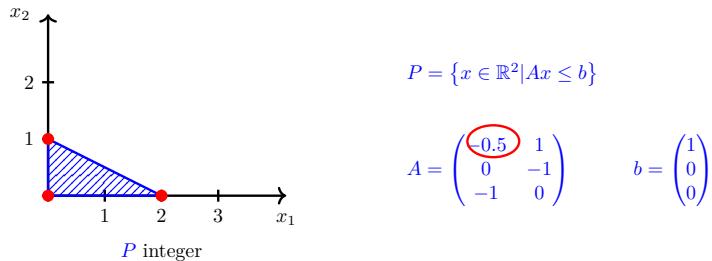


Fig. 9.18.: P integer but A not TUM (No. 727)

- ⇒ lot more research to do

TUM matrices and combinatorial optimization

- So far, we studied several different combinatorial optimization problems like the shortest path problem, the minimum spanning tree problem or the maximum matching problem

Shortest Path Problem Minimum Spanning Tree Maximum Matching

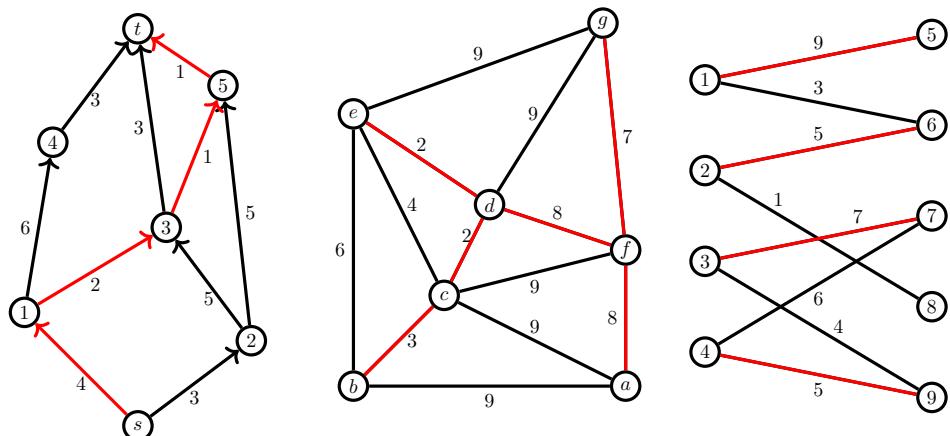


Fig. 9.19.: Classical combinatorial optimization problems (No. 721)

- Often, we can solve combinatorial optimization problems via combinatorial algorithms
 - However, a different approach is:
 - Step 1: Model the COP problem as IP
 - Step 2: Solve the IP to optimality
 - Consider the **shortest path problem**
- Given: Digraph $G = (V, A)$, costs $c : A \rightarrow \mathbb{R}_+$, a starting vertex $s \in V$ and a target vertex $t \in V$
- Find: a shortest (s, v) -path p
- Model the problem as IP

Shortest Path Problem Minimum Spanning Tree Maximum Matching

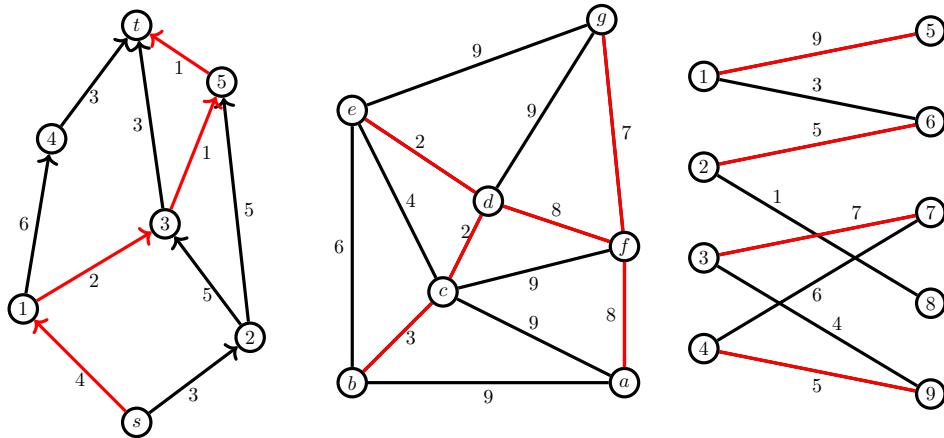


Fig. 9.20.: SP as MIP (No. 1136)

- Variables $x_a = \{0, 1\}$: 1 if the arc $a \in A$ is used in the path
- Constraints:
 - for every vertex besides s and t : if the path enters the vertex, then it also has to leave the vertex

$$\sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = 0 \quad \forall v \in V \setminus \{s, t\}$$

- At vertex s/t : one outgoing-arc (incoming-arc) needs to be used

$$\sum_{a \in \delta^+(s)} x_a = 1 \quad \text{and} \quad \sum_{a \in \delta^-(t)} x_a = 1$$

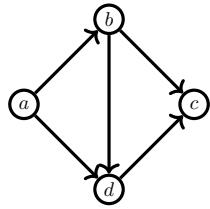
- In summary

$$\begin{aligned} \min \sum_{a \in A} c_a x_a \\ \sum_{a \in \delta^+(v)} x_a - \sum_{a \in \delta^-(v)} x_a = & \begin{cases} 1 & \text{for } v = s \\ 0 & \forall v \in V \setminus \{s, t\} \\ -1 & \text{for } v = t \end{cases} \\ x_a \in \{0, 1\} \quad \forall a \in A \end{aligned}$$

- The matrix defined by the constraints above is called the incidence matrix of a graph

Definition 168 (Incidence matrix). Let $G = (V, A)$ be a digraph. The $|V| \times |A|$ incidence matrix M of G is defined as

$$M_{v,a} := \begin{cases} +1 & \text{if } a = (v, w) \text{ for some } w \in V \\ -1 & \text{if } a = (w, v) \text{ for some } w \in V \\ 0 & \text{otherwise.} \end{cases}$$



$$M = \begin{matrix} & ab & ad & bd & bc & dc \\ a & 1 & 1 & 0 & 0 & 0 \\ b & -1 & 0 & 1 & 1 & 0 \\ c & 0 & 0 & 0 & -1 & -1 \\ d & 0 & -1 & -1 & 0 & 1 \end{matrix}$$

Fig. 9.21.: incidence matrix (No. 725)

- Nice property of the incidence matrix: it is totally unimodular

Theorem 169. The incidence matrix M of a digraph G is totally unimodular.

Proof. Theorem 165

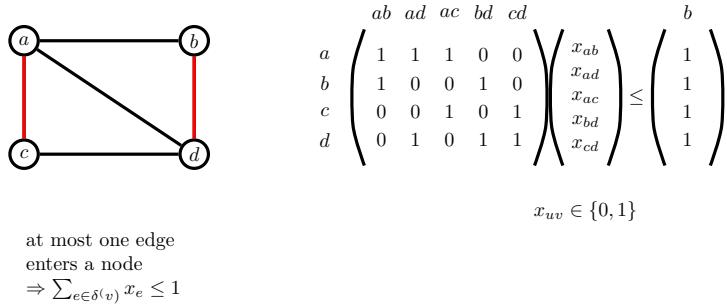
- M satisfies:
 1. Every column contains two non-zero entries
 2. The sum of every column equals 0
- $\Rightarrow M$ is TUM (Theorem 165)

□

- It follows immediately:

Corollary 170. The shortest path problem on a graph $G = (V, A)$ with cost c can be solved by computing an optimal basic solution x^* of $\min\{c^\top x \mid Mx = b, x \geq 0\}$ where $M \in \mathbb{Z}^{|V(G)| \times |A(G)|}$ is the incidence matrix of G and $b \in \{0, 1\}^{|V(G)|}$, $b(s) = 1$, $b(t) = -1$, $b(v) = 0$ otherwise.

- Using a similar approach, we can show that the maximum flow problem can be solved via the LP relaxation of the IP formulation for the problem (Exercise)
- Let us consider the **matching problem**
Given: Graph $G = (V, E)$
Find: $M \subseteq E$, such that $|M \cap \delta(v)| \leq 1$ for all $v \in V$



$$\begin{array}{l}
 ab \ ad \ ac \ bd \ cd \\
 a \left(\begin{array}{ccccc} 1 & 1 & 1 & 0 & 0 \end{array} \right) \left(\begin{array}{c} x_{ab} \\ x_{ad} \\ x_{ac} \\ x_{bd} \\ x_{cd} \end{array} \right) \leq \left(\begin{array}{c} b \\ 1 \\ 1 \\ 1 \\ 1 \end{array} \right) \\
 b \left(\begin{array}{ccccc} 1 & 0 & 0 & 1 & 0 \end{array} \right) \\
 c \left(\begin{array}{ccccc} 0 & 0 & 1 & 0 & 1 \end{array} \right) \\
 d \left(\begin{array}{ccccc} 0 & 1 & 0 & 1 & 1 \end{array} \right)
 \end{array}$$

$$x_{uv} \in \{0, 1\}$$

- Model the problem as IP
 - Variables $x_a = \{0, 1\}$: 1 if the arc $a \in A$ is used in the path
 - Constraints:
 - At most one arc incident to every v is selected

$$\sum_{a \in \delta(v)} x_a \leq 1 \quad \forall v \in V$$

- In summary

$$\begin{array}{ll}
 (IP_{\max M}) & \max \sum_{a \in A} x_a \\
 & \sum_{a \in \delta(v)} x_a \leq 1 \quad \forall v \in V \\
 & x_a \in \{0, 1\} \quad \forall a \in A
 \end{array}$$

- Can we also solve this problem by relaxation?
- Unfortunately, the matrix corresponding to an undirected graph is not TUM

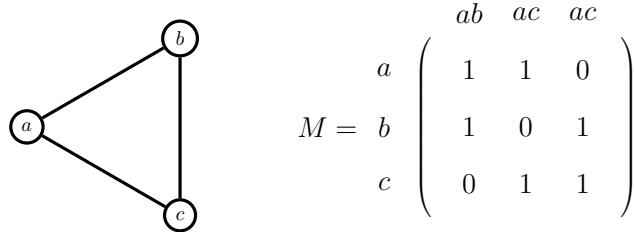


Fig. 9.23.: incidence matrix of undirected graph (No. 730)

- \Rightarrow Many considered problems on graphs will probably not be solved by this simple LP-relaxation technique
- What if we consider special graph classes?

Theorem 171. Let A be the vertex-edge incidence matrix of an undirected graph $G = (V, E)$, i.e.,

$$A_{v,e} = \begin{cases} 1 & \text{if } v \in e \\ 0 & \text{otherwise.} \end{cases}$$

Then A is totally unimodular if and only if G is bipartite.

Proof. Exercise □

- Thus, for bipartite graphs we can use the relaxation again and then also add weights to the matching problem

Corollary 172. *The maximum weighted matching problem in bipartite graphs can be solved in polynomial time by solving the LP relaxation of its integer linear programming formulation IP_{MaxM} .*

Proof. Vertex-edge incidence matrix is TUM □

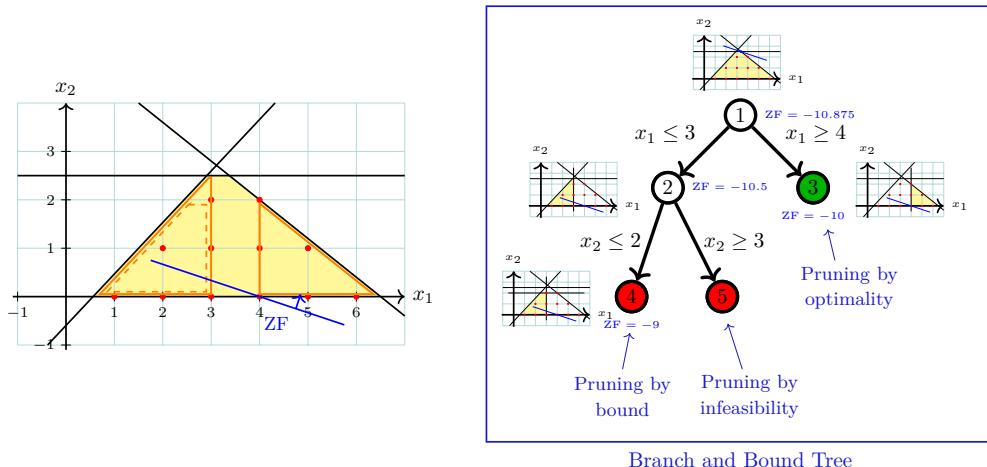
- The matrix A used in IP_{MaxM} corresponds to the vertex-edge incidence matrix of the given undirected graph
- G is bipartite $\Leftrightarrow A$ is TUM $\Leftrightarrow P := \{x \in \mathbb{R}^m \mid Ax \leq b, x \geq 0\}$ is integer
- \Rightarrow computing an optimal basic solution $\max\{c^\top x \mid x \in P\}$ solved the maximum matching problem

□

- Note, again this only holds for bipartite graphs
- In general, the matching problem cannot be solved as an LP

9.4. Branch and Bound

- Land and Doig introduced in 1960 branch and bound to solve IPs [?]
- It is one of the most common approaches to solve IPs
- **Example:** The Branch and Bound procedure based on LP-Relaxation works as follows:



$$\mathcal{L} = \left\{ \begin{array}{c} \text{Figure 1} \\ \text{Figure 2} \\ \text{Figure 3} \\ \text{Figure 4} \\ \text{Figure 5} \end{array} \right\}$$

Fig. 9.24.: Left: constrained LP-Relaxation, right: Branch & Bound Tree (No. 1138)

- The most important components for a maximization problem are

Upper Bounds	to determine the potential of the subproblem (e.g., LP relaxation, Lagrangian relaxation, Danzig-Wolfe Decomposition, Benders' Decomposition)
Lower Bounds	to prune unnecessary subproblems, i.e., find a feasible solution
Branching Rules	to define the subproblems
Selection Rules	to chose the next subproblem

- We obtain the following algorithm

Algo. 9.1 Branch&Bound for IPs

Input: Integer program $I = \max\{c^\top x \mid Ax \leq b, x \geq 0, x \in \mathbb{Z}\}$, $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{n \times m}$ and $b \in \mathbb{Z}^m$

Output: Optimal solution x^* or that no feasible solution exists

Method:

Step 1 // Initialization

Set $\mathcal{L} = \{I\}$ // list of all problems
 $\bar{z} = \infty$, $\bar{x} = \emptyset$ // upper bound an feasible solution

Step 2 // Stop criteria

If $\mathcal{L} = \emptyset$ Return x^* and z^* if they exists and \emptyset otherwise

Step 3 // Compute lower bound

- Chose $P \in \mathcal{L}$ and delete P from \mathcal{L}
- Compute the LP relaxation x_{LP} for P
- If $x_{LP} = \emptyset$, goto Step 2

Step 4 // Prune Bound: no potential in P

If $c(x_{LP}) \leq \bar{z}$, goto Step 2

Step 5 // Prune by optimality: Integer solution?

Elseif x_{LP} is an integer solution do

Set $\bar{z} = c(x_{LP})$ and $\bar{x} = x_{LP}$

Step 6 // Branch: partition in subproblems

Else //i.e. x_{LP} is no feasible solution

- Divide P in subproblems $P = P_1 \cup \dots \cup P_k$
 - Add them to \mathcal{L}
-

- In order to keep track of the procedure, *Branch&Bound trees* are used

- Branch&Bound trees are directed trees starting in the root node with the original problem
- all nodes represent different subproblems
- an edge (v, u) leads from a node v representing problem P to node u representing a subproblems P_i defined in Step 6

- Rule of the thumb: small Branch&Bound trees lead to fast computations
- In the worst case, the Branch&Bound tree enumerates all solution
- \Rightarrow the goal is, to prune as fast as possible, i.e., to stop splitting a problem and analyzing it further
- Three reasons for pruning a problem P after computing the optimal LP relaxation x_{LP}
 1. **Pruning by optimality:** x_{LP} is integer
 2. **Pruning by bound:** $c(x_{LP})$ is lower than the best lower bound, i.e., the objective of the best integer solution we have so far
 3. **Pruning by infeasibility:** no feasible solution for the LP relaxation exists
- Each Branch&Bound algorithm needs to specify the
 - Branching rules
 - Selection rules

Branching rules

- Branching rules can be distinguished by the number of contained subproblems
 - binary branching* P is divided into two subproblems with $P = P_1 \cup P_2$ and $P_1 \cap P_2 = \emptyset$
 - wide branching* P is divided into several subproblems with $P = P_1 \cup \dots \cup P_k$ with $P_j \cap P_i = \emptyset, i \neq j$
- Classical binary rule for integer problems is the following:
 - Let x_{LP} be an optimal solution for the LP relaxation of P
 - Let j be an index with $x_{LP,j} \notin \mathbb{Z}$
 - Then, define

$$P_1 = P \cap \{x \in \mathbb{R}^n \mid x_j \leq \lfloor x_{LP,j} \rfloor\}$$

and

$$P_2 = P \cap \{x \in \mathbb{R}^n \mid x_j \geq \lceil x_{LP,j} \rceil\}$$

- However, we still have several options since in general there are several indices of an LP relaxation that are not integer
 - To solve that problem, a score function is defined
 - This score function should rate the change in the objective considering the new subproblem
 - This leads to the following approach

Step 1 For any index $j \in F = \{j \in [n] \mid x_j \notin \mathbb{Z}\}$, we define a value s_j

Step 2 Choose $j = \arg \max_{k \in F} s_k$

- Classical ideas among others like pseudo cost branching, strong branching, ...

Most Infeasible $s_j = \min \{x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j\}$

Least Infeasible $s_j = \max \{x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j\}$

Selection rules

- Selection rules define, which subproblem out of the list \mathcal{L} to analyze next
- Classical approaches are
 - Depth First Search: corresponds to the last-in-first-out rule
 - Breath First Search: corresponds to first-in-first-out rule
 - Best First Search: chose the problem with the smallest upper bound

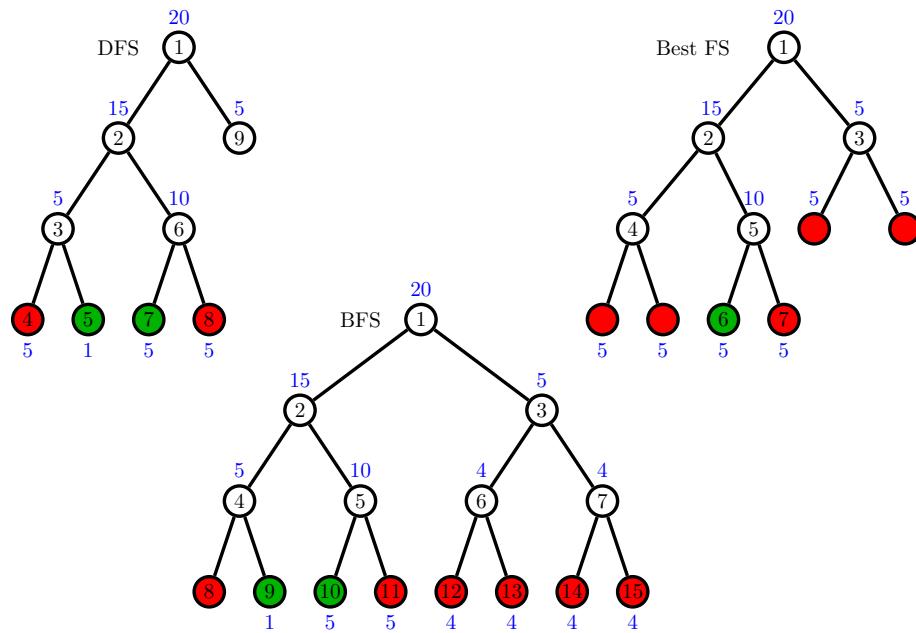


Fig. 9.25.: Selection rules (No. 1139)



10. Computational Complexity Theory: P vs. NP

Reading Material

- Computational complexity theory is an important field in theoretical computer science
- Goal: classify computational problems, e.g., optimization problems or decision problems, according to their resource usage, when using an algorithm to solve it

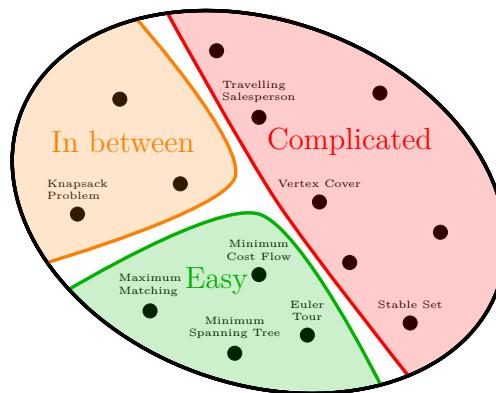


Fig. 10.1.: Classification of problems (No. 620)

- Resource usage is in most cases: time and storage
- A problem is inherently difficult if its solution requires significant resources, whatever algorithm used
- Thus, complexity theory determines the practical limits on what computers can and cannot do
- To formalize the theory, we need to define mathematical models of computation, problems ...
- We will just give a very rough introduction to these models
- Here we need to define:
 - The size of an instance for a problem
 - The resource consumption based on simple steps

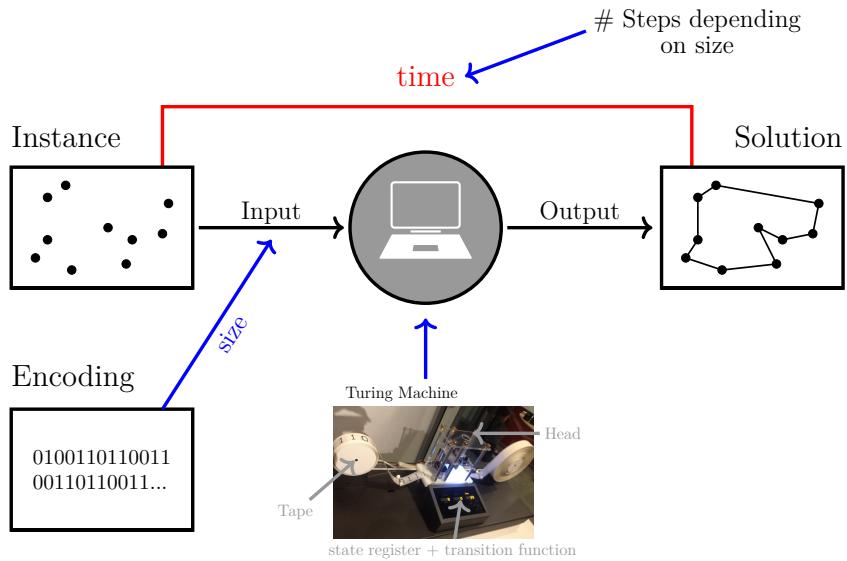


Fig. 10.2.: Model of computation (No. 840)

10.1. Encodings and Models of Computation

Problem, Instances and Encodings

- A computational *problem* is an abstract question to be solved
 - Shortest Path Problem: What is the shortest path from s to t in a graph G ?
- A *problem instance* is a concrete realization of a problem
 - Shortest Path Instance: What is the shortest path from s to t in the given graph G below?

Problem: What is the shortest path? What is a minimum spanning tree?

Instance:

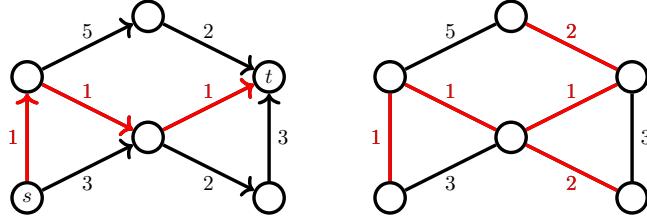


Fig. 10.3.: Problem vs. instance (No. 621)

- We will focus in the following on decision problems

Definition VII. *Decision problem* is a problem with a yes/no answer.

- **Example:** Decision problems

Bipartite graph	Max Flow
Instance: Undirected graph G	Instance: Network (G, s, t, u) ,
Question: Is G bipartite?	number v Question: Does an integer (s, t) -flow with $\text{value}(f) \geq v$ exist?

- In computer science, problem *instances* are defined as a string over an alphabet
 - alphabet: a set, usually $\{0, 1, \varepsilon\}$ where ε denotes something like a break between two different parts of the input
 - string: sequence of values of the alphabet, e.g., 001000011000
- To measure the difficulty of solving a problem, one tries to estimate how much time the best algorithm requires to solve the problem
- Obviously, the running time depends on the instance and its “size”

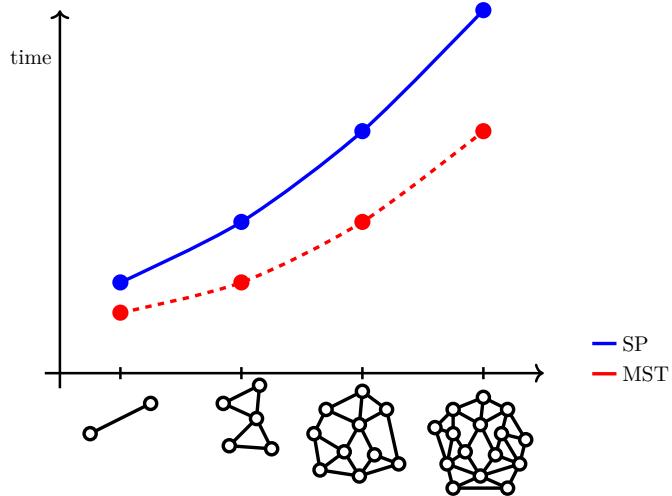


Fig. 10.4.: Size and run-time (No. 623)

- The *size* denotes the length (# bits) of the instance’s string and depends strongly on how big the real-world problem is and how it is encoded, i.e., translation into the language of computers

- **Example:** Integer numbers

- Input: number 5127
- Size in encoding

Encoding	String	Size
decimal	5127	4
binary	1010000000111	13
unary	...	5127

- **Example:** Simple graphs

- Input: Graph $G = (V, E)$ with $|V| = n$ and $|E| = m$
- Size in encoding

Encoding	Size
adjacency matrix	n^2
adjacency lists	$n + 2m$

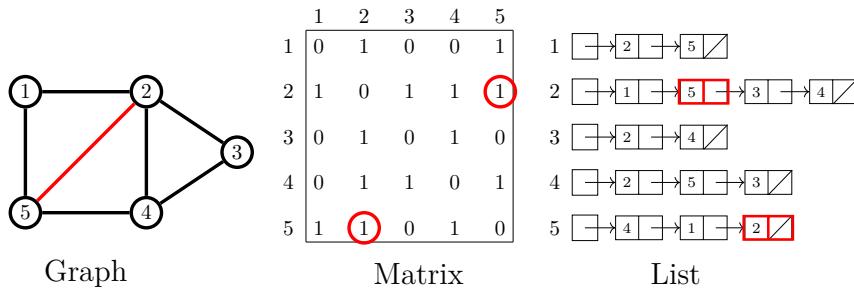


Fig. 10.5.: Encoding of a graph (No. 622)

- In general:

I instance of a computational problem

$C(I)$ encoding of I

$\langle C(I) \rangle$ length of encoding of I using C

- Often we simply use $\langle I \rangle$

- In many cases, it does not matter in theory which encoding is used

Definition VIII. Encodings C_1, C_2 are *polynomial equivalent* (w.r.t. a problem class) if and only if there exist polynomials $p_1, p_2 : N \rightarrow N$, such that for all instances I of the problem class it holds,

- $\langle C_2(I) \rangle \leq p_1(\langle C_1(I) \rangle)$,
- $\langle C_1(I) \rangle \leq p_2(\langle C_2(I) \rangle)$.

- **Example:** polynomial equivalent encodings

- Simple graphs

- Encoding as adjacency matrix and adjacency lists are polynomial equivalent

- Integer numbers

- Binary and k -nary (e.g. decimal) representation of integer numbers are polynomial equivalent

Binary: $\langle n \rangle = \lceil \log_2(|n|+1) \rceil + 1$

k -nary: $\langle n \rangle = \lceil \log_k(|n|+1) \rceil + 1 = \lceil (\log_2(|n|+1)) / \log_2 k \rceil + 1$

- BUT: binary and unary representations of integer numbers are not equivalent

Binary: $\langle n \rangle = \lceil \log_2(|n|+1) \rceil + 1$

Unary: $\langle n \rangle = n \geq 2^{\lceil \log_2(|n|+1) \rceil + 1} / 4$, i.e. $\langle n \rangle$ is exponential in $\lceil \log_2(|n|+1) \rceil + 1$

- Standard encodings

- Integer numbers: binary, $\langle n \rangle = \lceil \log_2(|n|+1) \rceil + 1$

- Vector $x = (x_1, \dots, x_n)$ as n integer numbers, $\langle x \rangle = \langle x_1 \rangle + \dots + \langle x_n \rangle$

- Graph $G = (V, E)$ as an array of adjacency lists, $\langle G \rangle = |V| + |E|$

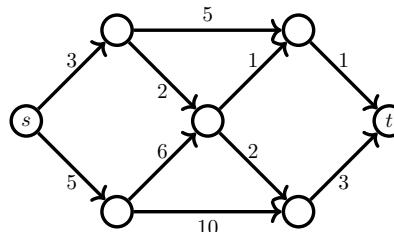
- Edge coefficients as integer numbers, i.e. with length $\langle u(e) \rangle = \lceil \log_2(|u(e)|+1) \rceil + 1$ for an edge e

Models of Computation

- Next to modeling instances and problems, we need a mathematical model of computation to quantify the computational resource consumption
- Such models lead to the definition of machines, which are similar to a physical setup of an algorithm in our sense

Problem:
Shortest Path

Instance



Machine



Turing Machine

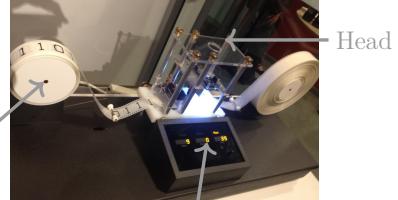


Fig. 10.6.: machines (No. 624)

- Most famous are Turing machines introduced in 1936 by Alan Turing
- *Turing machine* consists of:
 - A *tape* divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special blank symbol (here written as ' ϵ ') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, so that the Turing machine is always supplied with as much tape as it needs for its computation.
 - A *head* that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time.
 - A *state register* that stores the state of the Turing machine, one of finitely many. Among these is the special start state with which the state register is initialized, and a special accepting end state (to identify “yes” answers)
 - A *finite table/transition function* of instructions that, given the state the machine is currently in and the symbol it is reading on the tape (symbol currently under the head), tells the machine to do the following in sequence:
 - Either erase or write a symbol.
 - Move the head: 'L' one step left, 'R' one step right, or 'N' for staying.
 - Go to new state

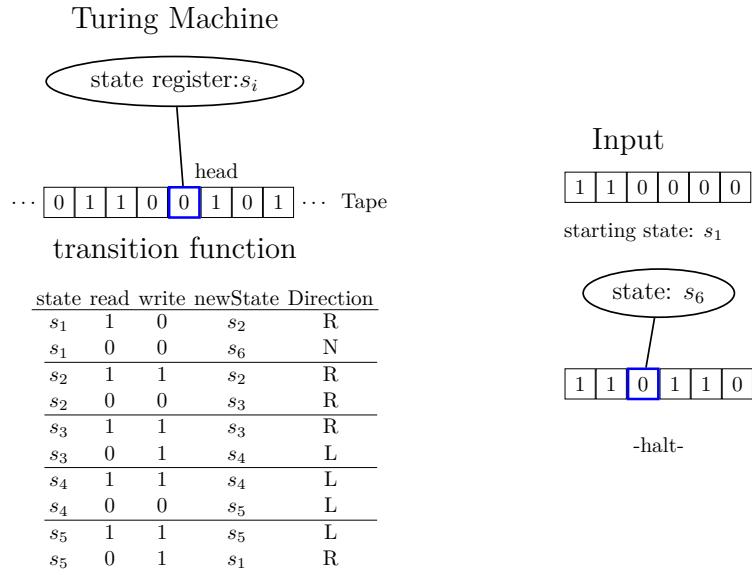


Fig. 10.7.: Turing machine (No. 625)

- **Example:** Turing Machine on $\{0, 1\}$, End-state s_6

State	read	\rightarrow	write	new State	Direction
s_1	1	\rightarrow	0	s_2	R
s_1	0	\rightarrow	0	s_6	N
s_2	1	\rightarrow	1	s_2	R
s_2	0	\rightarrow	0	s_3	R
s_3	1	\rightarrow	1	s_3	R
s_3	0	\rightarrow	1	s_4	L
s_4	1	\rightarrow	1	s_4	L
s_4	0	\rightarrow	0	s_5	L
s_5	1	\rightarrow	1	s_5	L
s_5	0	\rightarrow	1	s_1	R

- The *time required/runtime* by a Turing machine M on input x is the total number of state transitions, or steps, the machine makes before it halts and outputs the answer (“yes” or “no”)
- Turing machines are quite simple and relatively easy to analyze
- There are much more powerful theoretical machines, e.g., RAM (Random Access Machines), Conway’s Game of Life, ...
- **BUT: Church-Turing thesis:** it is believed that if a problem can be solved by an algorithm (= transition function), there exists a Turing machine that solves the problem
- In other words: Each of these (and other) machine types can be simulated by any other type, whereby the times for arithmetic operations, dependent on the same input length, only *differ by one polynomial* and the required memory differs only by a constant factor
- To *differ by only one polynomial* means: if T_i is the runtime for the model of computation i , $i \in \{1, 2\}$, then polynomials p_1, p_2 exist with
 - $T_1(\langle I \rangle) \leq p_2(T_2(\langle I \rangle))$ for all inputs I

- $T_2(\langle I \rangle) \leq p_1(T_1(\langle I \rangle))$ for all inputs I
 - \Rightarrow the resource consumption depends on the defined machine, but the influence is not too big (only a polynomial)
 - Why is a polynomial runtime so important?
 - Consider an algorithm A to solve a problem Π
 - Assume, we have 1 hour of computational time and two computers C_1 and C_2
 - C_2 has 10-fold speed of C_1 , i.e., for 1 step of C_1 the computer C_2 can perform 10 steps
 - Let I_i be the biggest instance of Π solvable within 1 hour by C_i
 - How much bigger is I_2 than I_1 ?
 - Case 1: Runtime of A is polynomial, i.e., $T_A(\langle I \rangle) = \langle I \rangle^k$
 - C_1 can perform $\langle I_1 \rangle^k$ steps in one hour
 - $\Rightarrow C_2$ can perform $\langle I_1 \rangle^k \cdot 10$ steps
 - $\Rightarrow \langle I_2 \rangle^k = \langle I_1 \rangle^k \cdot 10 \Rightarrow \langle I_2 \rangle = \langle I_1 \rangle \cdot 10^{\frac{1}{k}}$ (multiplicative improvement)
 - Case 2: Runtime of A is exponential, i.e., $T_A(\langle I \rangle) = 2^{\langle I \rangle}$
 - $\Rightarrow 2^{\langle I_2 \rangle} = 2^{\langle I_1 \rangle} \cdot 10 \Rightarrow \langle I_2 \rangle = \langle I_1 \rangle + \log(10)$ (additive improvement)

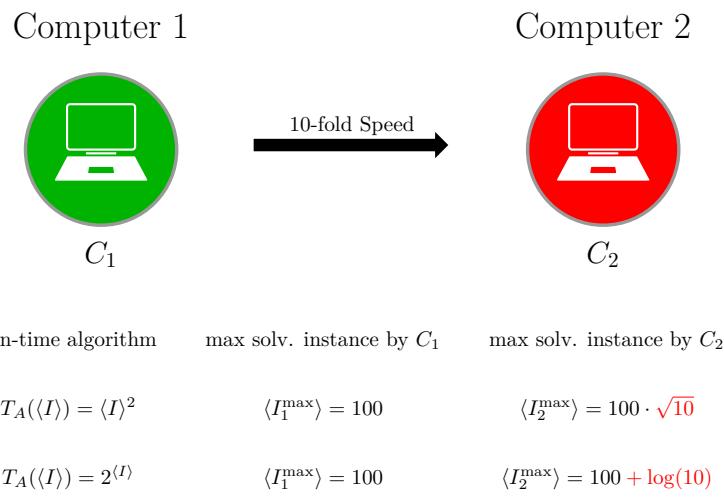


Fig. 10.8.: Speed up (No. 627)

- Turing machines, as described above, are quite simple, however, they can be equipped with more “power”
 - Types of Turing machines:
 - *deterministic*: as described above with a fixed transition function to determine future actions
 - *probabilistic*: randomly decides between two transition functions according to some probability distribution, which transition function to use
 - *non-deterministic*: allows a Turing machine to have multiple possible future actions from a given state
 - many more...
 - **Example:** 3SAT and deterministic and non-deterministic Turing machines

- 3SAT Problem:

- Given:
- Set of boolean variables x_1, \dots, x_n
 - Set of clauses C_1, \dots, C_m with $C_j = y_{1j} \vee y_{2j} \vee y_{3j}$, $y_{ij} \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$

Question: Is there an assignment of TRUE (= 1) and FALSE (= 0) to the variables, such that all clauses are satisfied?

- Deterministic machine:

- Enumerate all 2^n possible solutions
- If one solves the instance, return true

- Non-deterministic machine:

- Start with x_1 and guess the “right” answer for a feasible solution
- Proceed with all other variables
- If the obtained solution solves the instance, return true

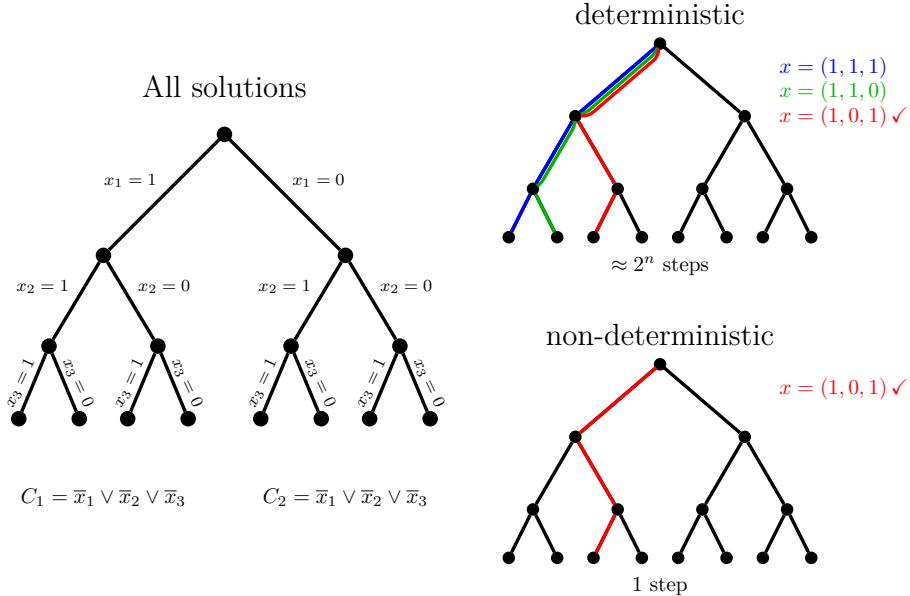


Fig. 10.9.: deterministic vs. non-deterministic example for 3SAT instances $(x_1 \vee x_2, \bar{x}_3)$, $(x_3 \vee \bar{x}_4 \vee \bar{x}_5)$, $(\bar{x}_1 \vee \bar{x}_4 \vee x_5)$ and $(\bar{x}_2 \vee x_4 \vee x_5)$ (No. 626)

- Non-deterministic machines have the power to efficiently find a solution, if one exists, by exploring the “right” path in the decision tree

10.2. Complexity Classes

- Complexity classes cluster problems of related resource consumption in e.g. time or space
- For us interesting complexity classes are defined by:
 - Type of computational problem: here decision problems (in contrast to optimization problems)
 - Model of computation: different types of Turing machines
 - The resource and its bound: here time which is polynomial bounded by the size

- The classes we consider are

Complexity Class	Model of computation	Resource constraint
P	deterministic Turing machine	Time $\mathcal{O}(\text{poly}(n))$
NP	non-deterministic Turing machine	Time $\mathcal{O}(\text{poly}(n))$

$\text{poly}(n)$ means polynomial according to the input of size n

- We start by characterizing algorithms and their run-time

Definition 173. The *run-time* of an algorithm A w.r.t. a deterministic model of computation (and an encoding) is a function $T_A : \mathbb{N} \rightarrow \mathbb{N}$ with $T_A(\langle I \rangle)$ equals the sum of the costs of all steps in A for an input I .

- The function T_A can be quite different
- Due to the Church-Turing thesis and the speed-up behavior, we classify algorithms quite rough into polynomial and not-polynomial algorithms

Definition 174. An algorithm A is *efficient (polynomial, has polynomial worst case complexity)* for a problem if a polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ exists such that A has a run-time $T_A(\langle I \rangle) \leq p(\langle I \rangle)$ for all instances I of the problem with $\langle I \rangle \geq n_0$, $0 \leq n_0 \in \mathbb{N}$.

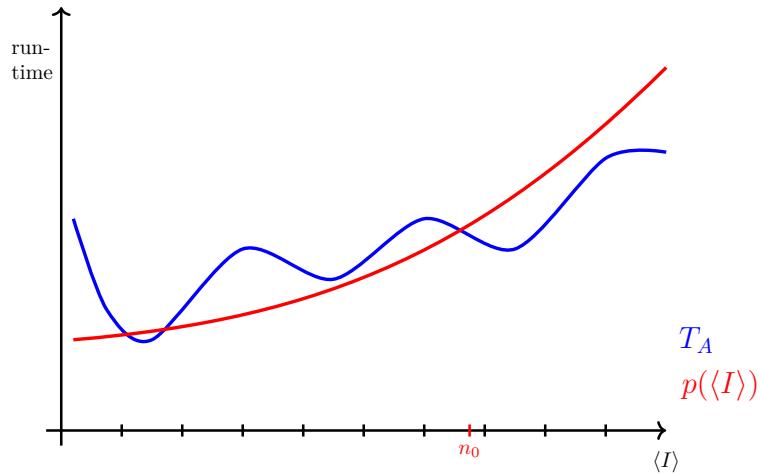


Fig. 10.10.: Polynomial solvable (No. 628)

- Using this, we can start classifying our problems

Complexity class P

- This class is often seen as the set of problems that can “in practice” be solved efficiently
- More formally,

Definition 175. A problem is *polynomial solvable* if there exists a polynomial algorithm for it.

- **Examples:** Some problems considered so far

Problem	Algorithm	Runtime
Euler-Tours	Algorithm of Hierholzer	$\mathcal{O}(m)$
Shortest-Path	Dijkstra Algorithm	$\mathcal{O}(n^2)$
Maximum Flow	Dinic's Algorithm	$\mathcal{O}(m \cdot n^2)$

Definition 176. The class **P** of decision problems contains all decision problems that can be solved in polynomial time by a deterministic model of computation, e.g., a deterministic Turing machine.

- **Note:** The definition is independent of machine model due to Church-Turing equivalence thesis

Complexity Class NP

- Idea: we consider a more powerful machine to solve our problems

Definition 177. The class **NP** of decision problems contains all decision problems that can be solved in polynomial time by a non-deterministic model of computation, e.g., a non-deterministic Turing machine.

- At first glance rather unintuitive
- Better intuition by the following theorem or alternative definition

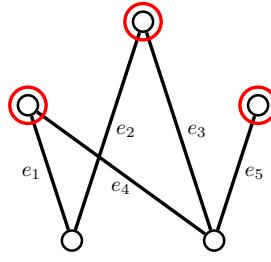
Theorem 178. A decision problem belongs to the problem class **NP** \Leftrightarrow a polynomial $p(n)$ and a (certificate-checking) algorithm A exist such that for every yes-instance I there is a certificate $Z(I)$ with length $\langle Z(I) \rangle \leq p(\langle I \rangle)$ that A verifies the answer “yes” after at most $p(\langle I \rangle)$ steps when inserting I and $Z(I)$.

- **Examples:**

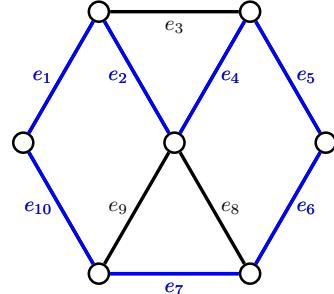
	Bipartite graph	Hamilton Cycle
Problem	Instance: Undirected graph G Question: Is G bipartite?	Instance: Undirected graph G Question: \exists a cycle visiting all vertices? (Hamiltonian Cycle)

Certificate	Set $A \subseteq V$	Series of edges $e_1 e_2 \dots e_n$
Algorithm	<ul style="list-style-type: none"> For all $e = \{a, b\} \in E$: test if $a \in A$ and $b \in V \setminus A$ or $b \in A$ and $a \in V \setminus A$ 	<ul style="list-style-type: none"> Test, if $e_1 \dots e_n$ forms a cycle Test if every vertex is part of $e_1 \dots e_n$

Problem: Bipartite?



Hamiltonian?



Certificate: vertex set A

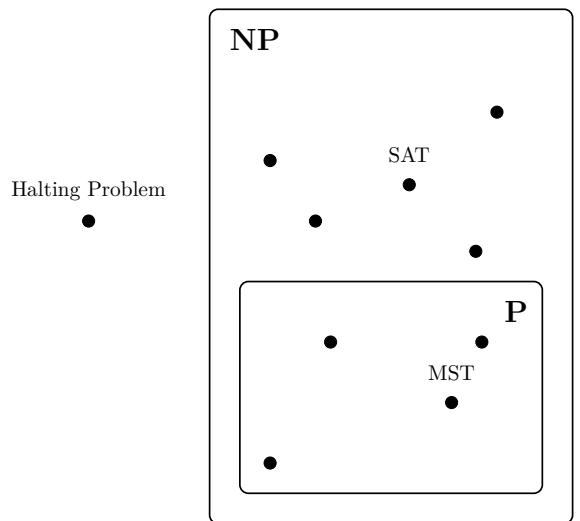
Test: $|e_i \cap A| = 1$
for $i = 1, \dots, 5$

Tour $C = e_1 e_2 e_4 e_5 e_6 e_7 e_{10}$

- $e_1 \dots e_n$ is a tour
- $\forall v \in V : v \in C$

Fig. 10.11.: Certificate checking for NP (No. 629)

- The class **NP** contains many practical problems that need to be solved
- What is the relation between **P** and **NP**?
- First observation: $\mathbf{P} \subseteq \mathbf{NP}$
- BIG QUESTION: $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$?
 - One of the biggest open questions of theoretical computer science
 - $\mathbf{P} = \mathbf{NP} \Rightarrow$ all problems in **NP** are polynomial solvable (but the efficient algorithms are not known yet)
 - $\mathbf{P} \neq \mathbf{NP} \Rightarrow$ transition from **P** to **NP** means a rise in complexity
 - The prize money is set at \$1,000,000 (Clay Mathematical Institute, Millennium Problems)
- In order to tackle this question, research focuses on identifying the most difficult problems in **NP**
- All these difficult problems are clustered in a set called **NP**-complete problems

Fig. 10.12.: Classification in **P** and **NP** (No. 630)

10.3. NP-complete problems

- First, we need to define when a problem Π_1 is more difficult than another problem Π_2

Definition 179. Let Π_1 and Π_2 be decision problems. Then, Π_1 *polynomially transforms* to Π_2 (i.e. $\Pi_1 \propto \Pi_2$) if a transformation $f : \Pi_1 \rightarrow \Pi_2$ exists such that

- for every instance $I \in \Pi_1$, an instance $f(I) \in \Pi_2$ is constructed in polynomial time (polynomial in $\langle I \rangle$),
- I is a yes-instance of Π_1 if and only if $f(I)$ is a yes-instance of Π_2 .

- In other words, Π_2 is more difficult than Π_1 since Π_1 is a “special case” of Π_2
- Example:** polynomial transformation of perfect matching problem in bipartite graph to a maximum flow problem

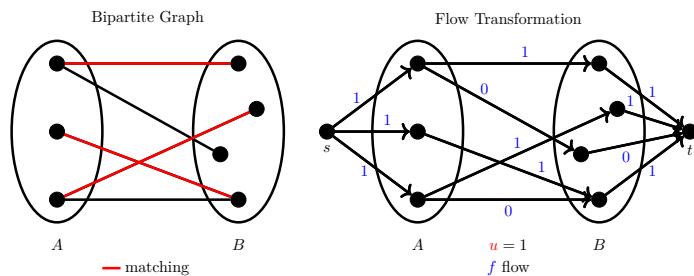


Fig. 10.13.: Max flow and bipartite matchings (No. 496)

	Perfect matching in bipartite graph	maximum flow problems
Problem:	Given: bipartite graph $G = (U \cup W, E)$	Given: Graph $G = (V, A)$, $s, t \in V$, $u : A \rightarrow \mathbb{Z}, k$

	Question: Is there a perfect matching in G ?	Question: Is there an (s, t) -flow f with $\text{value}(f) = k$?
Transformation f :	<p>Matching instance $I \rightarrow$ Maximum flow instance I'</p> <ul style="list-style-type: none"> • Direct all edges from U to W • Add vertex s and t to G • Connect s with all vertices in U • Connect all vertices in W with t • Set $u(a) = 1$ in the new graph • Set $k = U$ 	

- If Π_1 transforms to Π_2 , then \Rightarrow if A is a polynomial algorithm for Π_2 , then A can also solve Π_1 in polynomial time

Lemma 180. *It holds true that*

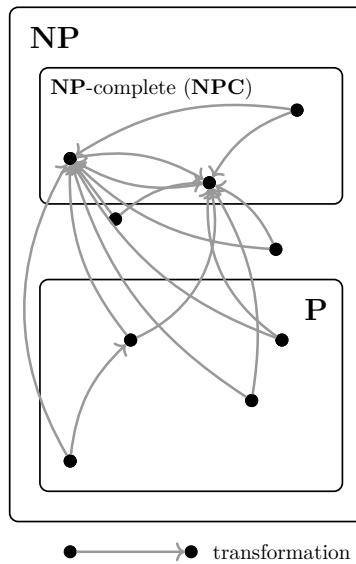
1. \propto is transitive.
2. If $\Pi_2 \in \mathbf{P}$ and $\Pi_1 \propto \Pi_2$, it follows that $\Pi_1 \in \mathbf{P}$.

- Nice visualization of the problems' relation is given via a graph
 - Each vertex represents a problem Π
 - An arc (Π_1, Π_2) exists if $\Pi_1 \propto \Pi_2$
- Using this idea, we see that the most difficult problems in **NP** are those that include all other problems in **NP** as special cases

Definition 181 (NP-complete). A decision problem Π is **NP-complete** if

1. $\Pi \in \mathbf{NP}$ and
2. $\Pi' \propto \Pi$ for all $\Pi' \in \mathbf{NP}$

- Collect them in a special class

Fig. 10.14.: **NP** complete (No. 633)

Definition 182. The set **NPC** consists of all of **NP**-complete decision problems.

Reading Material

- What do we know about **P**, **NP** and **NPC**?

Theorem IX. *It holds true that*

1. $\text{NPC} \cap \text{P} \neq \emptyset \Rightarrow \text{P} = \text{NP}$,
2. $\Pi_1 \in \text{NPC}, \Pi_2 \in \text{NP}, \Pi_1 \propto \Pi_2 \Rightarrow \Pi_2 \in \text{NPC}$.

Proof. Transitivity of \propto and per definition of classes **P**, **NP** and **NPC**. \square

- So far, almost all problems that are not in **P** but in **NP** are in **NPC**

Theorem X. [Ladner, 1975] If $\text{P} \neq \text{NP}$, then there exist problems $\Pi \in \text{NP} \setminus \text{P}$ with $\Pi \notin \text{NPC}$.

- These problems are also called **NP**-intermediate (**NPI**)
- There are only a few candidates for **NPI**, e.g., factorization of a number
- Long time graph-isomorphism was believed to be **NPI**
- By now, “most people believe that graph-isomorphism is in **P**” (Martin Grohe, talk)

- Main Question: Are there **NP**-complete problems? What do they look like?
 - Meanwhile several thousand

- Garey & Johnson started a systematic analysis in 1979 [?]
- A polynomial algorithm is not known for any of these problems (despite great research effort)
- \Rightarrow general presumption: $\mathbf{P} \neq \mathbf{NP}$
- The first problem proven to be \mathbf{NP} -complete is the satisfiability (SAT) problem
- This was a breakthrough and the start for many new fields of study: e.g.
 - Complexity theory in combinatorial optimization
 - Approximation algorithms

SAT and its Variations

- Most classical problem in computer science

		Definition 183. Satisfiability (SAT)					
Instance:	<ul style="list-style-type: none"> • n boolean variables x_1, \dots, x_n • m clauses C_1, \dots, C_m with $C_i = y_{i1} \vee y_{i2} \vee \dots \vee y_{i\ell_i}$ with literals $y_{ij} \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$ 						
Question:	Is there an assignment of TRUE (= 1) and FALSE (= 0) to the variables, so that all clauses are satisfied? (satisfiable truth assignment)						

$C_1 = \bar{x}_3$	$C_2 = x_1 \vee \bar{x}_2 \vee x_3$	$C_3 = x_2 \vee \bar{x}_3$	$C_4 = x_3 \vee \bar{x}_1$	$C_5 = \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$	
C_1				X	
C_2	X	X	X		
	x_1	x_2	x_3		
C_1	X		X		\checkmark
C_2	X		X		\checkmark
C_3		X		X	
C_4	X			X	
C_5	X	X	X		\checkmark
	x_1	x_2	x_3		

Fig. 10.15.: SAT instances (No. 634)

- **Example:** SAT instances

- $C_1 = \bar{x}_3, C_2 = x_1 \vee \bar{x}_2 \vee x_3$
 - is satisfiable with $x_3 = 0$ and $x_1 = 1$ (x_2 arbitrary)
- $C_1 = x_1 \vee x_2 \vee x_3, C_2 = x_1 \vee \bar{x}_2, C_3 = x_2 \vee \bar{x}_3, C_4 = x_3 \vee \bar{x}_1, C_5 = \bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$
 - not satisfiable, since
 1. C_1 true \Rightarrow one x_j must be assigned to TRUE
 2. C_2, C_3, C_4 true \Rightarrow all x_j are set to TRUE or all x_j are set to FALSE
 3. C_5 true \Rightarrow one x_j must be FALSE

- The **NP**-completeness of SAT was proven independently by Stephen Cook (USA) in 1971 and Leonid Levin (UdSSR) in 1973

Theorem 184 (Cook-Levin, 1971, 1973). *SAT is **NP**-complete.*

- The proof is rather technical
- But now, it is much easier to prove **NP**-completeness for a problem Π' :
 - Step 1: Show Π' is in **NP**
 - Step 2: Reduce some problem $\Pi \in \mathbf{NPC}$ to Π'

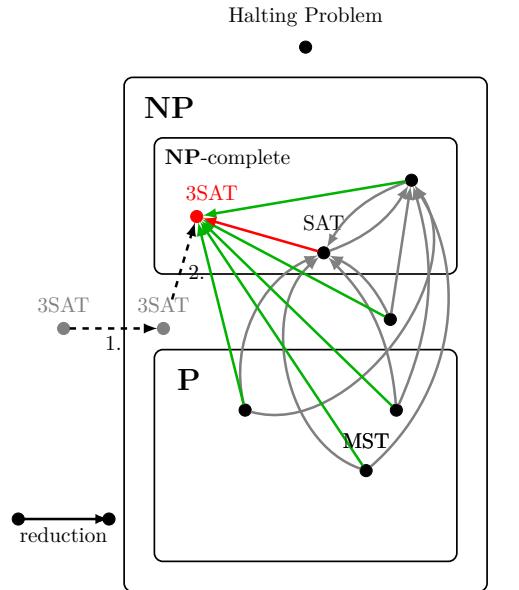


Fig. 10.16.: **P** and **NP** (No. 34)

- We will proceed by considering several different **NP**-completeness proofs
- Nice restriction (for further reductions) is 3SAT
- Here, each clause contains exactly 3 literals

Definition 185. *3-Satisfiability (3SAT)*

Instance:

- n Boolean variables x_1, \dots, x_n
- m clauses C_1, \dots, C_m with $C_i = y_{i1} \vee \dots \vee y_{i\ell_i}$, $\ell_i \leq 3$, with literals $y_{ij} \in \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$

Question: Is there an assignment of TRUE (= 1) and FALSE (= 0) to the variables, such that all clauses are satisfied?

- Nevertheless, 3SAT is **NP**-complete

Theorem 186. *3SAT is **NP**-complete.*

Proof. Simple transformation of SAT

- Step 1: 3SAT $\in \mathbf{NP}$ (special case of SAT)
- Step 2: Polynomial reduction from SAT to 3SAT
 - Let I be a SAT instance with n variables x_1, \dots, x_n and m clauses C_1, \dots, C_m
 - Construct an instance I' of 3SAT
 - I' contains the variables x_1, \dots, x_n
 - Furthermore, add all clauses C_j with $\ell_j \leq 3$
 - For every clause $C_j = y_{1j} \vee y_{2j} \vee \dots \vee y_{\ell_j j}$ with $\ell_j \geq 4$
 - Add variables $z_{1j}, \dots, z_{(\ell_j-1)j}$ to I'
 - Add the following clauses to I' :

$$C'_{j1} = y_{1j} \vee \bar{z}_{1j}, C'_{j2} = z_{1j} \vee y_{2j} \vee \bar{z}_{2j}, C'_{j3} = z_{2j} \vee y_{3j} \vee \bar{z}_{3j}, \dots, C'_{j(\ell_j-1)} = z_{(\ell_j-1)j} \vee y_{(\ell_j-1)j} \vee \bar{z}_{\ell_j j}, C'_{j\ell_j} = z_{(\ell_j-1)j} \vee y_{\ell_j j}$$

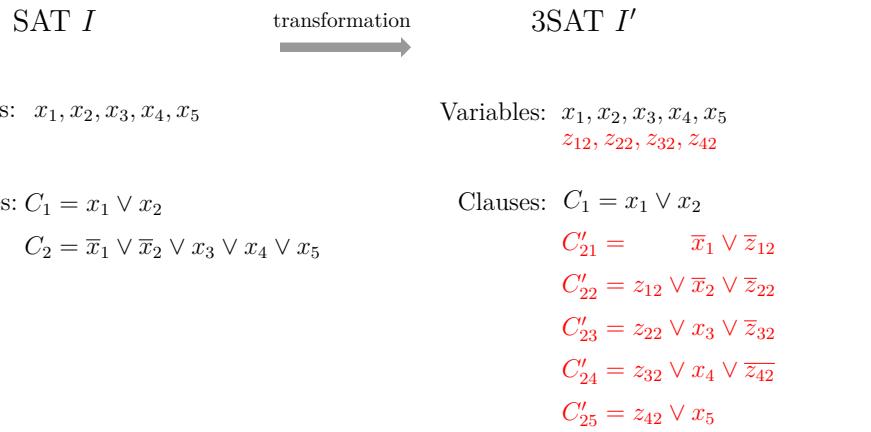


Fig. 10.17.: SAT to 3SAT (No. 635)

- *Claim:* The size of I' is polynomial in the size of I .

Proof:

- Let $\ell = \sum_{j=1}^m \ell_j$
- Then, I' contains at most $n + \ell \leq n + m \cdot n$ variables, since w.l.o.g. $\ell_j \leq n$
- I' contains at most $m + \ell \leq m + m \cdot n$ clauses
- \Rightarrow the size of I' is polynomial bounded by the size of I $\square C$

- *Claim:* If I is a yes-instance, then I' is a yes-instance.

Proof:

- Let x^* be an assignment satisfying I
- Set $x' = x^*$
- \Rightarrow all clauses with $\ell_j \leq 3$ are satisfied
- Let C_j be a clause with $\ell_j \geq 4$
- Let y_{ij} be the literal set to TRUE by x^*
- For all $i' \leq i$, set $z_{i'j} = \text{FALSE} \Rightarrow C'_{j'i'}$ is satisfied
- For all $i' > i$, set $z_{i'j} = \text{TRUE} \Rightarrow C'_{j'i'}$ is satisfied
- $\Rightarrow I'$ is a yes-instance $\square C$

- *Claim:* If I' is a yes-instance, then I is a yes-instance.
Proof:

- Let (x', z') be an assignment satisfying I'
- Set $x^* = x'$
- \Rightarrow all clauses with $\ell_j \leq 3$ are satisfied
- Let C_j be a clause with $\ell_j \geq 4$, $j \in \{1, \dots, m\}$
- Let C'_{ji} be the clauses identified with C_j , $i = 1, \dots, \ell_j$
- At least one clause C'_{ji} is satisfied by the corresponding literal y_{ji}
 - Let us assume, that is not the case
 - Every variable z_{ij} , $i = 1, \dots, \ell_j - 1$ can satisfy at most one clause
 - \Rightarrow in total ℓ_j clauses are satisfied \Rightarrow contradiction
- \Rightarrow clause C_j is also satisfied by x^*
- $\Rightarrow I$ is a yes-instance

□C

□

- Surprisingly, 2SAT can be solved in polynomial time
- Further important (or helpful) SAT variations:
 - Exactly one in 3SAT: every clause consists of three literals and a clause is satisfied if exactly one literal is satisfied
 - (3, B2)-SAT: every clause consists of three literals and every literal occurs exactly twice in the formula [?]
- SAT is a classical problem from computer science
- However, many classical combinatorial optimization problems are also **NP**-complete

Some of Karp's 21 NP-complete problems

- In 1972 Karp published a paper "Reducibility Among Combinatorial Problems" [?]
- In his work, he showed **NP**-completeness for 21 combinatorial and graph theoretical problems
- This was one of the first demonstrations that many natural combinatorial problems occurring in computer science are computationally intractable

Definition 187. Stable Set Problem

Instance:

- Undirected graph G
- natural number k

Question: Does G contain a stable set with k vertices? A *stable set* is a set of pairwise non-adjacent nodes

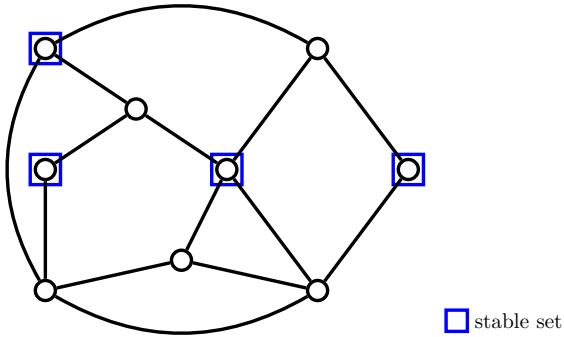


Fig. 10.18.: Stable set problem (No. 636)

Theorem 188 (Karp, 1972). *The stable set problem is NP-complete.*

Proof. Proof of Karp's Theorem

- Step 1: The stable set problem is in **NP**
 - Certificate is a set of vertices S of a graph $G = (V, E)$
 - Check that $|S| \geq k$ and $|\{u, v\} \cap S| \leq 1 \forall uv \in E(G)$
- Step 2: Transformation of SAT to the stable set problem
 - Let I be a SAT instance with
 - n variables x_1, \dots, x_n and
 - m clauses C_1, \dots, C_m , $C_j = y_{1j} \vee \dots \vee y_{\ell_j j}$, $y_{tj} \in \cup_{i=1}^n \{x_i, \bar{x}_i\}$
 - Construction of an instance I' of the stable set problem:
 - Define a graph $G = (V, E)$, which contains one node v_{ij} for every literal y_{ij} , $j = 1, \dots, m$, $i = 1, \dots, \ell_j$,
 - For every clause C_j add all edges between the corresponding nodes, i.e., for $y_{ij}, y_{i'j} \in C_j$ add $v_{ij}v_{i'j}$ to E
 - Let y_{ij} and $y_{i'j'}$ correspond to x_ℓ and \bar{x}_ℓ for some $\ell \in \{1, \dots, n\}$, then add $v_{ij}v_{i'j'}$ to E
 - Set $k = m$

$$C_1 = \bar{x}_1 \vee \textcircled{x}_2 \vee x_3 \quad , \quad C_2 = x_1 \vee \bar{x}_3 \quad , \quad C_3 = \textcircled{x}_2 \vee \bar{x}_3 \quad , \quad C_4 = \bar{x}_1 \vee \bar{x}_2 \vee \textcircled{\bar{x}}_3$$

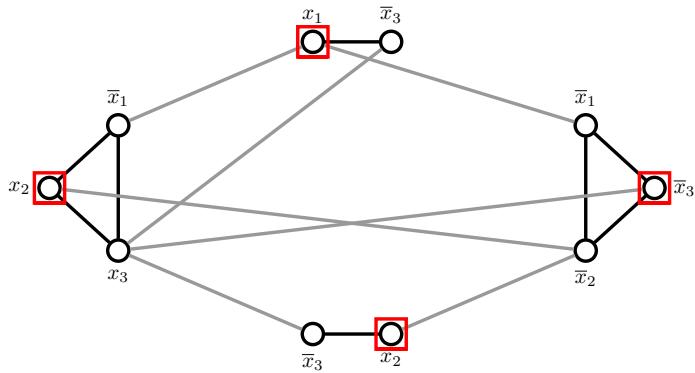


Fig. 10.19.: Reduction 3SAT to stable set (No. 637)

- Construction is obviously possible in polynomial time:

$$|V| = \sum_{i=1}^m \ell_j \leq n \cdot m \text{ and } |E| \leq 2n \cdot m + n^2 \cdot m$$

- *Claim 1:* If G has a stable set of size $k = m$, then a satisfiable assignment for I exists

Proof of Claim:

- G has a stable set S with m vertices
 - $\Rightarrow S$ contains exactly one vertex from each complete graph on the vertices representing C_j
 - Set the corresponding literals to TRUE \Rightarrow no contradiction, since contained in stable set
 - Set all other variables arbitrarily
 - \Rightarrow This provides a satisfiable assignment $\square C1$

Claim 2: If a satisfiable assignment for I exists, then G has a stable set of size $k = m$.

Proof of Claim:

- Consider satisfiable assignment x^* of C_1, \dots, C_m
 - Choose one literal per clause with value TRUE
 - Choose corresponding vertex to be contained in S
 - $\Rightarrow S$ is a stable set, since there are no contradicting edges in S (because of satisfiable assignment)
 - $\Rightarrow |S| \geq k = m$ by construction $\square C2$

- Two related problems are:

Definition 189. *Vertex Cover Decision Problem*

Instance: • Undirected graph G
 • natural number k

Question: Does G have a vertex cover S of cardinality k ? A *vertex cover* covers all edges of E . An edge $e = uv \in E$ is covered, if either u or v are contained in S .

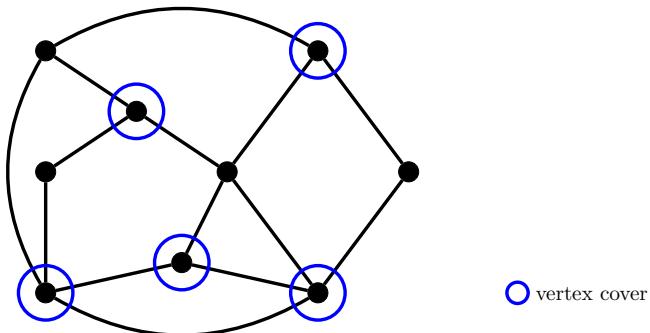


Fig. 10.20.: Vertex Cover (No. 487)

Definition 190. *Clique Decision Problem*

Instance: • Undirected graph G
 • natural number k

Question: Does G have a clique C of cardinality k ? A set of vertices $C \subseteq V(G)$ is a *clique* if $uv \in E(G) \forall u, v \in C$.

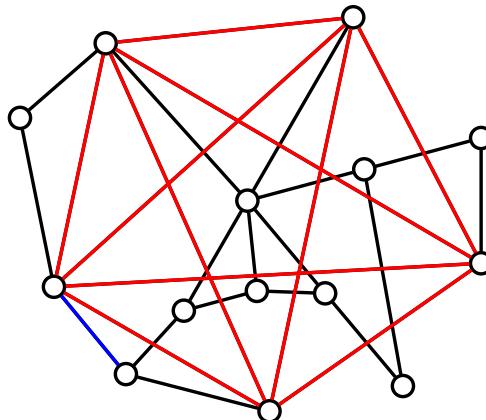


Fig. 10.21.: Clique (No. 638)

Corollary 191 (Karp 1972). *The Vertex cover decision problem and the clique decision problem are **NP**-complete.*

Proof Idea. Equivalence of problem solutions

- The following statements are equivalent:

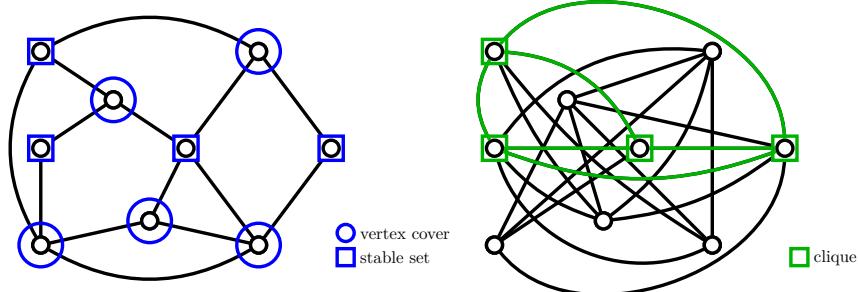


Fig. 10.22.: Vertex cover, stable set, clique (No. 639)

1. X is a vertex cover in G ,
 2. $V(G) - X$ is a stable set in G ,
 3. $V(G) - X$ is a clique in the complement of G .
- From this, polynomial transformations can immediately be specified

□

- One of the most basic problems concerning paths is the Hamilton Cycle problem

Definition 192. Hamilton Cycle Problem

Instance: Undirected graph G

Question: Does G contain a Hamilton cycle? A *Hamilton cycle* visits every node in G exactly once.

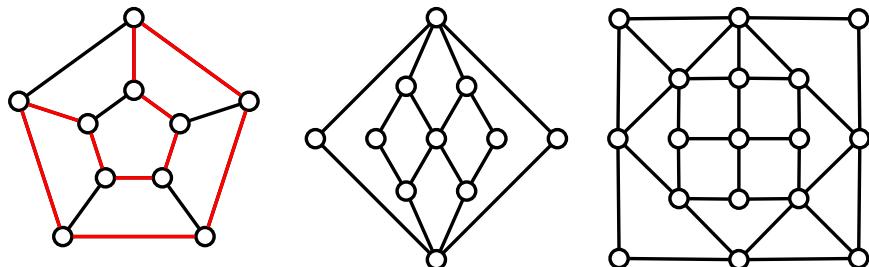


Fig. 10.23.: Hamiltonian Cycle (No. 640)

- The problem is named after Sir William Hamilton
- He invented 1857 “The Icosian Game”, in which the goal is to find a path in the dodecahedron



Sir William Hamilton
(1805-1865)



Icosian Game, 1857

Fig. 10.24.: Sir William Hamilton (No. 787)

Theorem 193 (Karp 1972). *Hamilton cycle problem is NP-complete.*

- The reduction is from 3SAT and uses small subgraphs to represent the variables and clauses

Vertex Coloring

- A classical graph theoretical problem is vertex coloring

Definition 194. *Vertex Coloring Problem*

Instance: Undirected graph G

Task: Find a vertex proper k -coloring for G with a minimum number of colors. A *proper k -coloring* is a mapping $f : V(G) \rightarrow \{1, \dots, k\}$ with $uv \in E(G) \Rightarrow f(u) \neq f(v)$.

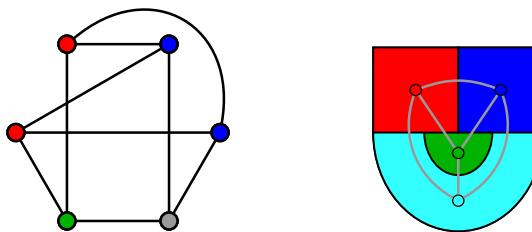


Fig. 10.25.: Vertex coloring (No. 748)

- Vertex coloring has many applications
- Color groups often represent items that should be together
- In these cases, edges represent conflicts
- In the decision version, we ask whether a coloring with k colors exists (i.e., a k -coloring)

- In the following, we want to prove the **NP**-completeness of the coloring problems
- We start by considering the following auxiliary graph

Definition 195 (auxiliary graph for coloring). The following graph H depicted below is called the *auxiliary graph for coloring*, where x_1, x_2 and x_3 are called the input vertices and v is called the output vertex.

- This auxiliary graph has the following properties

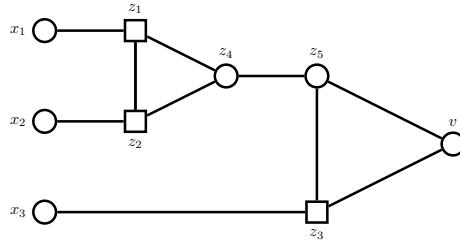
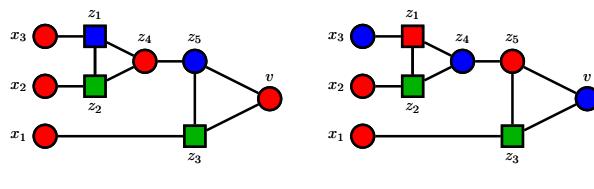


Fig. 10.26.: Auxiliary graph for coloring (No. 867)

Lemma 196. Let H be an auxiliary graph for coloring. Then

- (P1) Every proper 3-coloring of H in which the input vertices all have color 0 also assigns color 0 to the output vertex.
- (P2) If the input vertices receive colors that are not all 0, then this coloring extends to a proper 3-coloring of H in which the output vertex does not have color 0.

Proof. Sketch of proof: Case distinction



• 0 • 1 • 2

Fig. 10.27.: Properties of auxiliary graph for coloring (No. 868)

□

- Unfortunately, the problem is already **NP**-complete if we ask for a 3-coloring
- The proof is mainly based on the properties of the auxiliary graph
- **Note:** the most difficult part in such reduction is to find the right subgraph

Theorem 197 (Stockmeyer 1973 [?]). 3-coloring is **NP**-complete.

Proof. Reduction from 3SAT

- Step 1: 3-Coloring $\in \mathbf{NP}$
 - Let $G = (V, E)$ be an instance of the 3-coloring problem
 - Certificate: function $f : V \rightarrow \{1, 2, 3\}$
 - Check that $f(v) \neq f(u)$ for all $\{v, u\} \in E$, i.e., f is a proper 3-coloring
- Step 2: Polynomial reduction from 3SAT to Vertex Coloring
 - Let I be a 3SAT instance with n variables x_1, \dots, x_n and m clauses C_1, \dots, C_m
 - W.l.o.g. every clause contains exactly three literals
 - Construct an instance I' of the 3-coloring problem
 - Add vertices a, b and for every variable x_1, \dots, x_n two vertices x_i, \bar{x}_i
 - Connect the vertices a, x_i and \bar{x}_i such that they form a triangle ($1 \leq i \leq n$)
 - For each clause $C_1 \dots C_m$, add a copy H_1, \dots, H_m of the auxiliary graph for coloring such that the input vertices correspond to the literals
 - Connect vertex b with every output vertex v_1, \dots, v_m and add the edge ab
 - Connect for every clause C_j the literal y_{1j} with z_{1j} , y_{2j} with z_{2j} and y_{3j} with z_{3j}

$$\begin{array}{lll} C_1 = x_1 \vee x_2 \vee x_3 & C_2 = \bar{x}_1 \vee \bar{x}_2 \vee x_4 & C_3 = \bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4 \\ x_1 = \text{TRUE}, x_2 = \text{FALSE}, x_3 = \text{TRUE}, x_4 = \text{TRUE} & & \end{array}$$

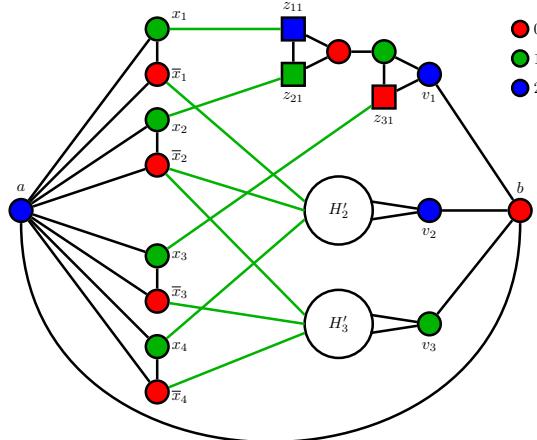


Fig. 10.28.: Construction of the graph (No. 874)

- *Claim:* The size of I' is polynomial in the size of I .

Proof of claim.

- I' contains $2n + m + 2$ vertices and $3n + 11m + 1$ edges
- \Rightarrow the size of I' is polynomial bounded by the size of I $\square C$
- The auxiliary graph for coloring has the following properties
 - (P1) Every proper 3-coloring of H in which the input vertices all have color 0 also assigns color 0 to the output vertex.

- (P2) If the input vertices receive colors that are not all 0, then this coloring extends to a proper 3-coloring of H in which the output vertex does not have color 0.

- Identify the color 0 (red) with the Boolean FALSE, color 1 (green) with TRUE
- *Claim:* If I is a yes-instance, then I' is a yes-instance.

Proof of claim:

- Consider a satisfiable assignment x^* of C_1, \dots, C_m
- Define the following coloring

$$x_i^* = \text{TRUE} \Rightarrow \begin{cases} f(x_i) = 1 \\ f(\bar{x}_i) = 0 \end{cases} \quad \text{and} \quad x_i^* = \text{FALSE} \Rightarrow \begin{cases} f(x_i) = 0 \\ f(\bar{x}_i) = 1 \end{cases} .$$

- Since the assignment x^* is feasible, every clause contains a literal with the boolean TRUE
- \Rightarrow Every auxiliary graph H_j contains an input vertex which has color 1
- \Rightarrow (P2) One can extend the coloring of H_j such that the output vertex does not have color 0
- Set $f(b) = 0$ and $f(a) = 2$

□C

- *Claim:* If I' is a yes-instance, then I is a yes-instance

Proof of claim:

- Consider a feasible 3-coloring of the graph
- W.l.o.g. $f(a) = 2$ and $f(b) = 0$, otherwise change color classes
- Since $f(a) = 2$, the vertices corresponding to the literals must have the color 0 or 1
- Furthermore, $f(x_i) \neq f(\bar{x}_i)$ for $i = 1, \dots, n$
- Define the assignment as follows: $x_i = \begin{cases} \text{FALSE}, & \text{if } f(x_i) = 0 \\ \text{TRUE}, & \text{if } f(x_i) = 1 \end{cases}$
- Since $f(b) = 0$, the output vertex is not colored in 0
- \Rightarrow (P1) and $f(a) = 2$: Every auxiliary graph has an input vertex that has color 1
- \Rightarrow Every clause contains at least one literal with the Boolean TRUE, the assignment is satisfying

□C

□

- The considered problems are all based on graphs
- The following problem is independent of graphs but often used for further reductions

The Subset-Sum Problem

- Subset Sum is the question to find a subset of elements such that their sum match a certain value
- We will see later that this is a quite special **NP**-complete problem and that it enables us to further differentiate the class **NPC**

Definition 198. *Subset Sum*

Instance: • Set of numbers $A = \{a_1, \dots, a_n\}$, $a_i \in \mathbb{N}$
 • $b \in \mathbb{N}$

Question: Does a set $S \subseteq A$ with $\sum_{a \in S} a = b$ exist?

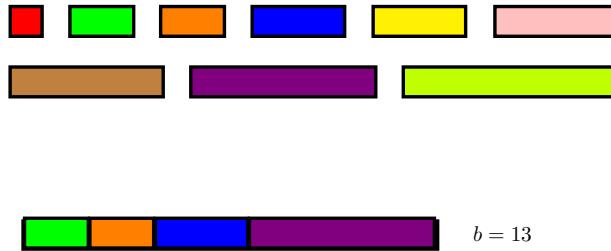


Fig. 10.29.: Subset Sum (No. 652)

Theorem 199. *Subset Sum is NP-complete.*

Proof. Reduction from 3SAT

- Step 1: Subset Sum is in NP
 - Given a set S , just compute the sum and compare it to b
- Step 2: Reduction from 3SAT
 - Let I be an instance with
 - m clauses C_1, \dots, C_m over
 - n variables x_1, \dots, x_n
 - Construct an instance of Subset Sum I' :
 - $A = \{a_1, \bar{a}_1, \dots, a_n, \bar{a}_n, c_1, d_1, \dots, c_m, d_m\}$
 - Every number $a \in A$ contains $m + n$ digits: out of the set $\{0, 1, 2, 4\}$
 - Denote with a_j the j^{th} digit of a , and c_j, d_j respectively
 - For every variable x_i define a_i and \bar{a}_i , $i \in \{1, \dots, n\}$
 - $\bar{a}_{ii} = a_{ii} = 1, \bar{a}_{ij} = a_{ij} = 0$ for $1 \leq i, j \leq n$
 - $a_{i(n+j)} = 1$, if $x_i \in C_j, 1 \leq j \leq m$ and
 - $\bar{a}_{i(n+j)} = 1$, if $\bar{x}_i \in C_j, 1 \leq j \leq m$
 - For every clause C_j define c_j and d_j , $j \in \{1, \dots, m\}$
 - $c_{j(n+j)} = 1, c_{ji} = 0$ for $1 \leq i \leq m + n, i \neq j$
 - $d_{j(n+j)} = 2, d_{ji} = 0$ for $1 \leq i \leq m + n, i \neq j$
 - Set $b := \underbrace{4444 \dots 4}_{m \text{ digits}} \underbrace{111 \dots 1}_{n \text{ digits}}$

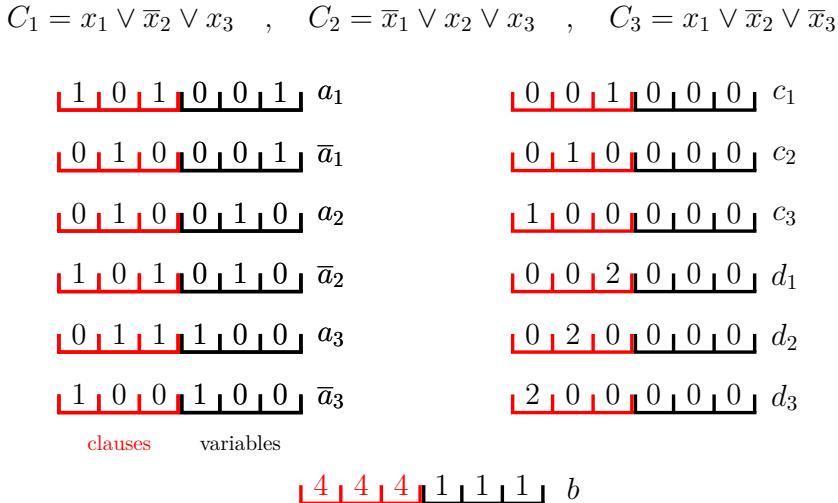


Fig. 10.30.: Construction of Subset sum Instance (No. 646)

- If you add all numbers, there is no carry over
- If b is obtained by a set S , then $|S \cap \{a_i, \bar{a}_i\}| = 1 \ \forall i = 1, \dots, n$
- \Rightarrow if $a_i \in S \Leftrightarrow$ assign x_i to TRUE (and if $\bar{a}_i \in S \Leftrightarrow$ assign x_i to FALSE)
- Consider a digit $j \in \{n+1, \dots, n+m\}$
- For any subset $A' \subseteq A \setminus \{c_1, d_1, \dots, c_m, d_m\}$, $0 \leq \sum_{a \in A'} a_j \leq 3$
- \Rightarrow If $\sum_{a \in A'} a_j \geq 1$, the digit $b_j = 4$ is obtained by adding c_j or d_j or both to A'
- $b_j = 4$ is obtained by a set $A' \Leftrightarrow$ at least one literal is set to TRUE by the correspondence of A' and an assignment x

□

- **Note:** the numbers we are using are quite big compared to the size of the SAT instance (e.g., four variables lead to numbers with a value > 999)

10.4. Further Complexity Results

NP-Hardness

- Until now, we considered decision problems (“yes”/“no”)
- However, we often deal with minimization problems

Definition 200 (Minimization problem). A minimization problem is defined by a pair (\mathcal{X}, c) , where \mathcal{X} represents the set of *feasible solution*, and $c : \mathcal{X} \rightarrow \mathbb{R}$ the *objective function* whose value is to be minimized. A solution x^* is an optimal solution, if there exists no feasible solution with a lower objective value, i.e., $c(x^*) \leq c(x) \ \forall x \in \mathcal{X}$.

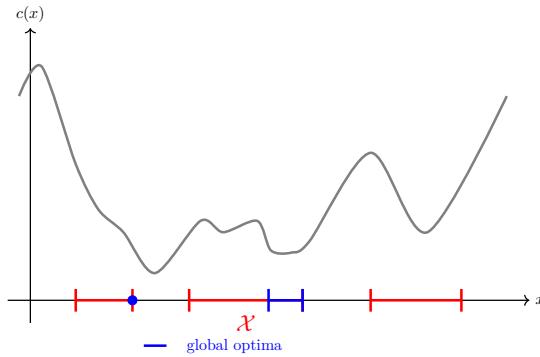


Fig. 10.31.: A general minimization problem (No. 250)

- We can easily deduce decision problems from minimization problems:

Definition 201 (Corresponding decision problem). Let (\mathcal{X}, c) be an minimization problem. Then the *corresponding decision problem* is defined as (\mathcal{X}, c, k) with $k \in \mathbb{R}$ and the question: does there exist a feasible solution $x \in \mathcal{X}$ with $c(x) \leq k$?

- An algorithm solving (\mathcal{X}, c) also solves (\mathcal{X}, c, k) :
 - compute an optimal solution x^*
 - if $c(x^*) \leq k$, return “yes”, otherwise “no”
- An algorithm solving (\mathcal{X}, c, k) also solves (\mathcal{X}, c)
 - Find smallest k' such that (\mathcal{X}, c, k') is a “yes”-instance
 - Use binary search to determine k'
 - This solution represents an optimal solution to (\mathcal{X}, c)
- Complexity classes can now be transformed to minimization problems (in a slightly simplified way)

Definition 202 (NP-hard). A minimization problem is **NP-hard**, if the corresponding decision problem is **NP-complete** for some k .

- One classical minimization problems is the Traveling Salesperson Problem

Definition 203. Traveling Salesperson Problem (TSP)

Given: Complete graph $K_n, n \geq 3, n \in \mathbb{N}$, edge coefficients $c(e) \geq 0$

Task: Determine a Hamilton cycle C with minimal length $c(C) = \sum_{e \in E} c(e)$

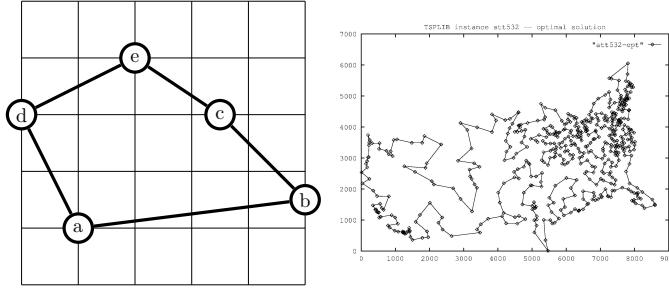


Fig. 10.32.: TSP (No. 247)

- The corresponding decision problem adds a value k and asks: is there a Hamilton cycle C with $c(C) \leq k$.

Theorem 204. *The Traveling Salesperson Problem is **NP-hard**.*

Proof. Reduction of Hamilton cycle

- Consider an instance I of the Hamilton cycle problem with $G = (V, E)$

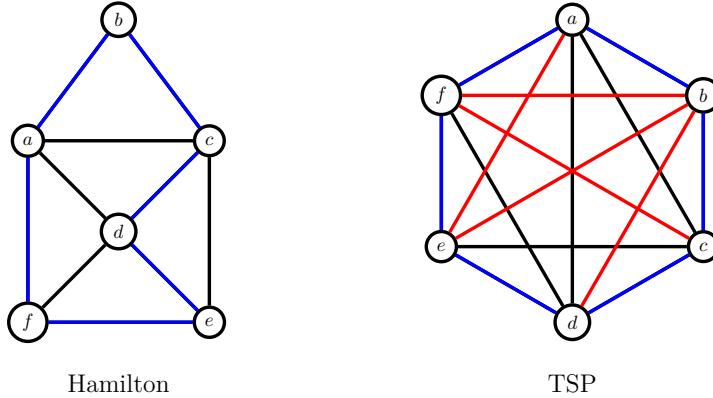


Fig. 10.33.: Hamilton vs. TSP (No. 648)

- Define an instance I' of the optimization problem (the traveling salesperson problem):
 - $G' = K_n$ with $n = |V|$
 - Set $c(e) := 1$ if $e \in E(G)$ and $c(e) = 2$ otherwise
- Let C^* be an optimal TSP tour in G'
- $\Rightarrow G$ has a Hamilton cycle $\Leftrightarrow c(C^*) = n$
- Hence, Hamilton cycle can be solved with one inquiry of an oracle for TSP.

□

- Consequence: If $\mathbf{P} \neq \mathbf{NP}$, no polynomial exact algorithm for **NP-hard** minimization problems exist

Strongly and weakly NP-complete/NP-hard Problems

- Complexity always depends on the encoding length of the instance
- However, let a be some (binary encoded) number
- \Rightarrow the encoding of a needs $\log_2(a)$ space
- Turning this around, we obtain the following:
- If we have an encoding of length n , we can use numbers of value $2^{\mathcal{O}(n)}$

Definition 205 (weakly and strongly NP-complete). Let Π be an NP-complete problem and Π_p the same problem restricted to all instances, where all values are polynomial restricted by the instance size. Then Π is *strongly NP-complete*, if Π_p is NP-complete, and *weakly NP-complete* otherwise.

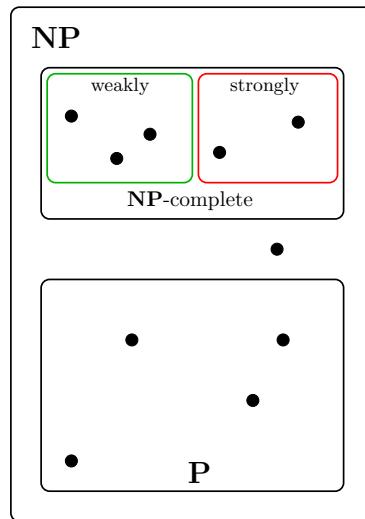


Fig. 10.34.: Strongly and weakly NP-complete (No. 649)

- Strongly NP-complete: even in case of instances with small numbers, the problem remains difficult
- Weakly NP-complete: these problems are only difficult, if the used numbers are “big” (as in the reduction of 3SAT to Subset Sum)
- All decision problems that don’t use numbers (e.g., SAT or Hamilton cycle) are strongly NP-complete
- This concept also works for minimization problems

Theorem 206. *The Traveling Salesperson Problem is strongly NP-hard.*

Proof. Proof of NP-completeness only uses 1 and 2 as numbers □

- For weakly NP-complete problems there often exist so-called pseudo-polynomial algorithms

- A pseudo-polynomial algorithm runs in polynomial time for small numbers

Definition 207 (Pseudo-polynomial algorithm). A *pseudo-polynomial algorithm* is an algorithm whose runtime, at the input of instance I , is polynomial in $\langle I \rangle$ and $\text{num}(I)$, where $\text{num}(I)$ is the largest value appearing in I .

- Consider the Subset Sum problem
- We already proved that Subset Sum is **NP**-complete

Theorem 208. A pseudo-polynomial algorithm for subset sum exists.

Proof. Transformation to a shortest path problem

- Let I be a Subset Sum instance with $A = \{a_1, \dots, a_n\} \subseteq \mathbb{N}$ and $b \in \mathbb{N}$
- Construct a digraph $G = (V, E)$ with
 - Vertices (j, r) , $j = 0, 1, \dots, n$ and $r = 0, 1, \dots, b$
 - Arcs $((i-1, r), (i, r))$ and $((i-1, r), (i, r+a_i))$, $i = 1, \dots, n$, $r = 0, \dots, b$

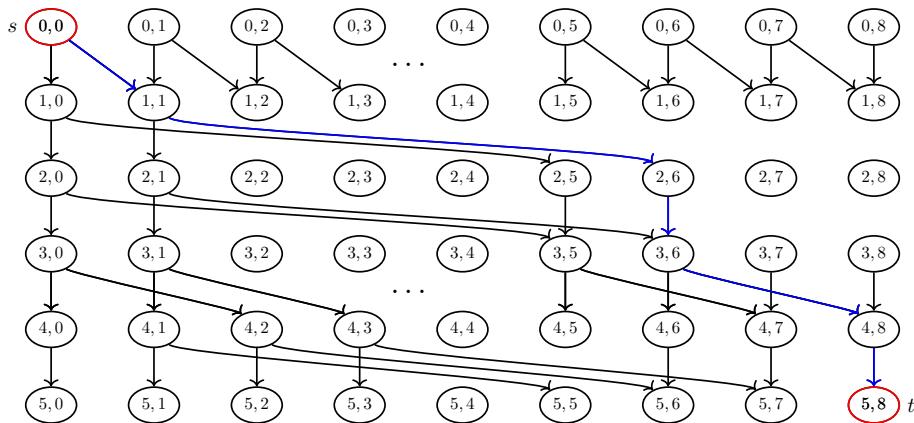
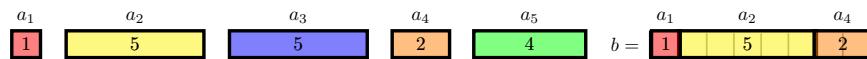


Fig. 10.35.: Pseudo-polynomial algorithm for Subset Sum (No. 650)

- A path p from $(0, 0)$ to (n, b) can be interpreted as:
 - If $((i-1, r), (i, r)) \in p$, $r \in \{0, \dots, b\}$: the element a_i is not part of A'
 - If $((i-1, r), (i, r+a_i)) \in p$, $r \in \{0, \dots, b\}$: the element a_i is part of A'
- Due to construction: if p exists, it defines a set $A' \subseteq A$ as described above with $\sum_{a \in A'} a = b$
- \Rightarrow instance I of subset sum is yes-instance \Leftrightarrow a path from $(0, 0)$ to (n, K) exists
- Such a path can be found by breadth-first-search in G in $O(n \cdot b)$

□

- In general, pseudo-polynomial algorithms just exist for weakly NP-hard problems

Theorem 209. *If $\mathbf{P} \neq \mathbf{NP}$, then no strongly NP-complete problem can be solved by a pseudo-polynomial algorithm.*

Proof. Contradiction

- Assume Π is solved by a pseudo-polynomial algorithm A
- $\Rightarrow A$ is polynomial in $\langle I \rangle$ and $\text{num}(I)$, i.e. a polynomial p exists with $T_A(I) \leq p(\langle I \rangle, \text{num}(I))$
- Now, consider the restriction Π_q of Π for a polynomial q
- For an instance $I \in \Pi_q$ it then holds $\text{num}(I) \leq q(\langle I \rangle)$
- $\Rightarrow T_A(I) \leq p(\langle I \rangle, \text{num}(I)) \leq p(\langle I \rangle, q(\langle I \rangle))$
- $\Rightarrow A$ solves Π in polynomial time $\Rightarrow \mathbf{P} = \mathbf{NP}$, contradiction

□

- Hence, there is no pseudo-polynomial algorithm for the Traveling Salesperson Problem, unless $\mathbf{P} = \mathbf{NP}$.

The Class **coNP**

- The class **NP** is not symmetrical w.r.t. yes-instances and no-instances
- The class **coNP** corresponds to the class **NP**, but for no-instances
- **coNP** = class of decision problem for which (as in the definition of **NP**) certificate-checking algorithm for no-instances exists

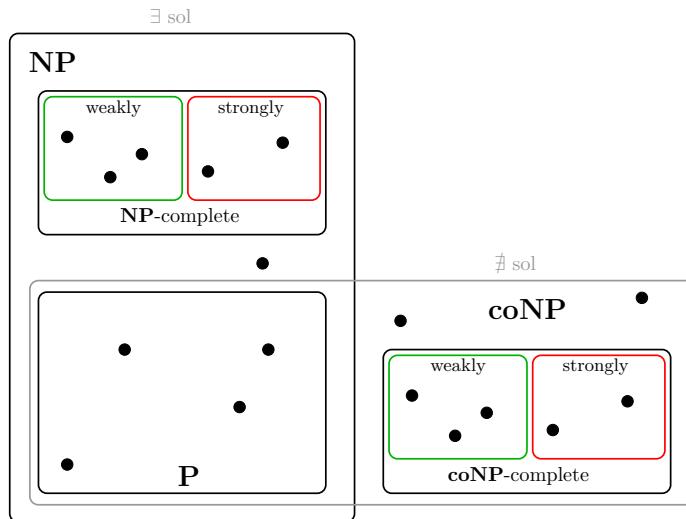


Fig. 10.36.: **NP**, **coNP**, **P** (No. 651)

Definition 210. A decision problem Π is **coNP**-complete if and only if

1. $\Pi \in \mathbf{coNP}$,
2. $\Pi' \propto \Pi$ for all $\Pi' \in \mathbf{coNP}$.

- **coNPC** := class of **coNP**-complete problems
- The *complement* $\mathbf{co}(\Pi)$ of a decision problem Π has the same instances as Π , however, the question w.r.t. $\mathbf{co}(\Pi)$ is the negation of the question w.r.t. Π

Definition 211. $\mathbf{co}(\text{Hamilton cycle})$

Instance: Undirected graph G

Question: Does G contain no Hamilton cycle?

- Obviously, $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$

Theorem 212. For a decision problem Π with $\Pi \in \mathbf{NP}$ and $\mathbf{co}(\Pi) \in \mathbf{coNP}$, the following are equivalent,

- Π is **NP**-complete
- $\mathbf{co}(\Pi)$ is **coNP**-complete.

Proof. Straightforward, since $\Pi \propto \Pi' \Leftrightarrow \mathbf{co}(\Pi) \propto \mathbf{co}(\Pi')$. □

- Question: Is $\mathbf{NP} \cap \mathbf{coNP} = \mathbf{P}$?
- In general one assumes:
 - $\mathbf{P} \neq \mathbf{coNP}$
 - $\mathbf{NP} \neq \mathbf{coNP}$
- The following is known:
 - If $\mathbf{P} = \mathbf{NP}$, then $\mathbf{NP} = \mathbf{coNP} = \mathbf{P}$
 - If $\mathbf{coNP} = \mathbf{NP}$, then $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ remains open



11. Approximation Algorithms

- Three alternatives to dealing with **NP**-hard problems
 1. *Exact Algorithms*: Algorithm with potentially exponential worst-case run-time that computes an optimal solution
 2. *Approximation algorithms*: Abstain from exact, but remain polynomial, provide worst-case quality guarantee
 3. *Heuristic algorithms*: Design a fast algorithm that computes a solution without any guarantee on the quality

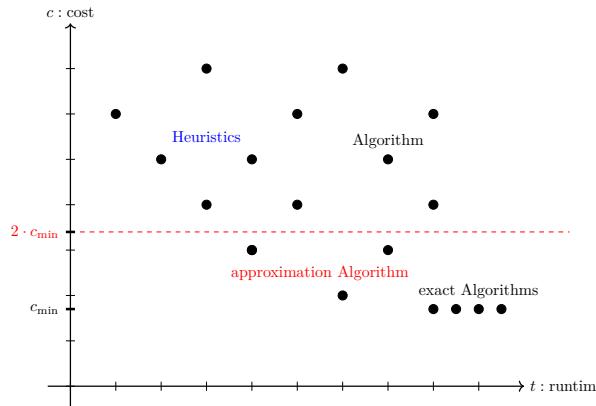


Fig. 11.1.: Classification of algorithms (No. 732)

- Let I be an instance of a minimization problem and $\text{OPT}(I)$ be the optimal solution value
- Approximation algorithms A are classified by “how well they perform”:
 - constant approximation ratio α : $A(I) \leq \alpha \cdot \text{OPT}(I)$ (for minimization problems)
 - absolute approximation α : $A(I) - \text{OPT}(I) \leq \alpha$ (for minimization problems)
 - approximation by a function $\alpha : \mathbb{N} \rightarrow \mathbb{R}$: $A(I) \leq \alpha(\text{size}(I)) \cdot \text{OPT}(I)$
- But caution: not all problems can have an approximation algorithm unless $\mathbf{P} = \mathbf{NP}$!

11.1. Approximation Algorithms with (Constant) Approximation Ratio

- We start with constant approximation ratios

Definition 213. An *approximation algorithm* A calculates for each instance $I \in \Pi$ of an minimization problem with cost c a feasible solution $x_{\text{ALG}}(I)$ in polynomial time s.t. $c(x_{\text{ALG}}(I)) \leq \alpha \cdot c(x^*(I))$ where $x^*(I)$ is an optimal solution. The value α is the *approximation ratio* of A .

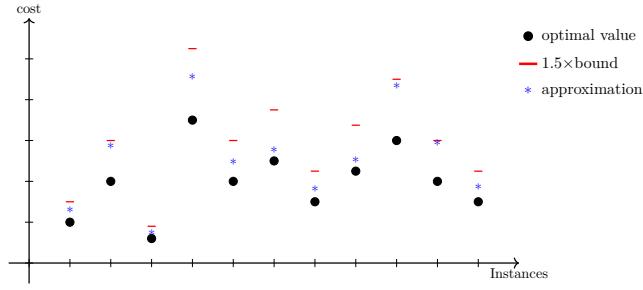


Fig. 11.2.: Approximation (No. 733)

- Constant approximation algorithms are known for many different classical optimization problems

Metric TSP Problem

- Consider the metric TSP problem

Definition 214. Metric Traveling Salesperson Problem (TSP)

Given:

- complete undirected graph K_n , $n \geq 3$
- edge cost $c_{ij} \in \mathbb{R}_+$ $\forall i, j \in [n]$, s.t.
 - $c_{ij} \leq c_{ik} + c_{ki} \forall i, j, k \in [n]$ (triangle inequality)
 - $c_{ij} = c_{ji} \forall i, j \in [n]$ (symmetry)
 - $c_{ii} = 0 \forall i \in [n]$

Task: Determine a Hamilton cycle C with minimum cost.

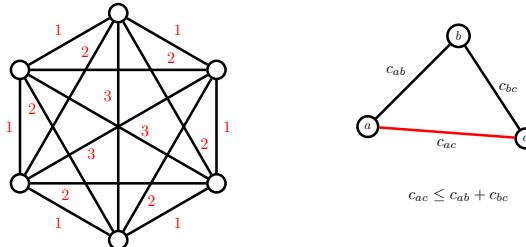


Fig. 11.3.: metric TSP (No. 734)

- In a metric TSP instance, $c_{ij} \geq 0$ holds
- Like the classical TSP problem, the metric TSP is **NP-hard**

Theorem 215. *The metric TSP is NP-hard.*

Proof. In the proof of Theorem 204 the costs are metric □

- Finding an optimal cycle through all vertices with minimum cost is difficult
- However, if we leave out one arc, we obtain a spanning tree
- Finding a minimum spanning tree is easy

- Can we somehow like these two problems?
- Consider the following algorithm

Algo. 11.1 Double-Tree Algorithm (DTA) by Rosenkrantz, Stearns, Lewis (1977)

Input: Complete undirected graph K_n and metric edge costs $c : E \rightarrow \mathbb{R}_+$

Output: Hamilton cycle (a *tour*)

Method:

- Step 1: Construct a minimum spanning tree T of K_n w.r.t. the c_{ij}
- Step 2: Duplicate each edge of T , resulting in an Eulerian graph T_d
- Step 3: Calculate an Euler tour in T_d
- Step 4: Construct tour $DTA(I)$ by traversing the Euler tour starting at vertex 1. Use the edge ik as shortcut if all vertices between i and k have already been traversed

return tour $DTA(I)$

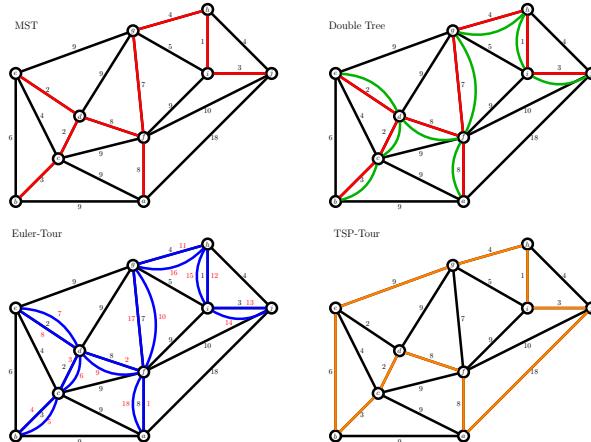


Fig. 11.4.: Double-Tree Algorithm; the missing edges in the graph have cost according to the shortest path between all vertices (No. 735)

- The algorithm has clearly a polynomial run-time

Theorem 216. *The Double-Tree Algorithm is a 2-approximation algorithm for the metric TSP, i.e.,*

$$DTA(I) \leq 2 \cdot OPT(I)$$

for all instances I of the metric TSP with $OPT(I)$ being the value of an optimal solution and $DTA(I)$ the objective value of the solution computed by the algorithm.

Proof. Triangle inequality

- Let H be an optimal tour of the TSP instance I
- Let T be the MST constructed in Step 1

- *Claim:* $c(T) \leq c(H)$.

Proof of Claim:

- Define $H^- = H - e$ where e is some edge on H
- Then H^- is a spanning tree in K_n

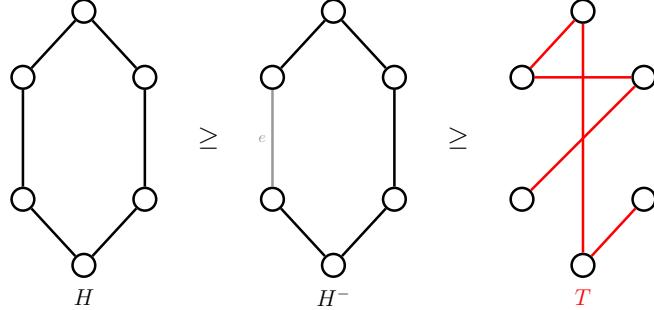


Fig. 11.5.: $c(T) \leq c(H)$ (No. 737)

- Since T is a minimum spanning tree,

$$c(T) \leq c(H^-) \stackrel{(a)}{\leq} c(H^-) + c(e) = c(H)$$

(a): $c(e) \geq 0 \ \forall e \in K_n$

□C

- Let $DTA(I)$ be the cost of the tour constructed by the algorithm

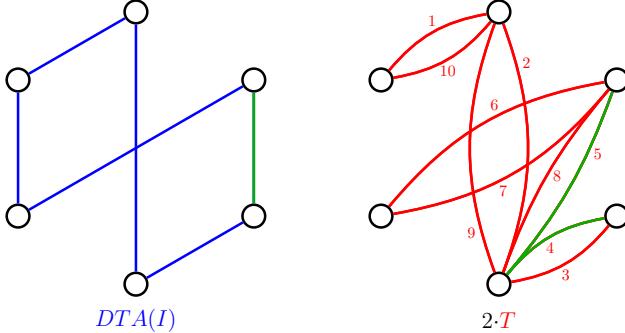


Fig. 11.6.: $DTA(I) \leq 2c(T)$ (No. 738)

- \Rightarrow

$$\begin{aligned} DTA(I) &\stackrel{(1)}{\leq} 2 \cdot c(T) \\ &\leq 2 \cdot c(H) = 2 \cdot OPT(I) \end{aligned}$$

- (1) due to the triangle inequality

□

- Classical Question: Is this approximation ratio sharp?

Case 1: Our analysis is not good, i.e., the algorithm performs even better

Case 2: For every $\alpha < 2$, there is an example, in which the algorithm does not obtain an α -approximation

- For the DTA the second case is true

Lemma 217. *The Double-Tree Algorithm can not obtain a better approximation ratio than 2.*

Proof. Worst case example

- Consider an $n \times 2$ -grid in the Euclidean plane with edge length 1

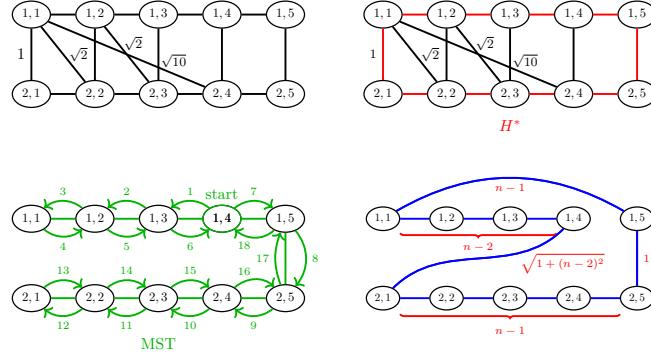


Fig. 11.7.: Worst Case DTA (No. 739)

- Denote the nodes with $(1, i)$ and $(2, i)$, $i = 1, \dots, n$
- $\Rightarrow c(\{(\ell, i), (t, j)\}) = \sqrt{(\ell - t)^2 + (i - j)^2}$
- Optimal tour: $H^* = (1, 1), (1, 2), \dots, (1, n), (2, n), (2, n-1), \dots, (2, 1), (1, 2), (1, 1)$
- Cost of H^* : $c(H^*) = 2n$
- Consider the MST $T = H^* - \{(2, 1), (1, 1)\}$
- Consider Euler-Tour E with start in $(1, n-1)$, the path to $(1, 1)$, the path back to $(1, n-1)$, the path to $(2, 1)$ and the path back to $(1, n-1)$
- Consider the Hamiltonian cycle H obtained from E :
 - Use all edges from $(1, n-1)$ to $(1, 1)$, then connect $(1, 1)$ with $(n, 1)$, use all edges to $(2, 1)$ and the edge $\{(2, 1), (1, n-1)\}$
- Cost of H :

$$\begin{aligned}
 c(H) &= n-2 + n-1 + 1 + n-1 + \sqrt{1 + (n-2)^2} \\
 &= 3n-3 + \sqrt{1 + (n-2)^2} \\
 &> 3n-3 + \sqrt{(n-2)^2} = 4n-5
 \end{aligned}$$

- Then

$$\frac{c(H)}{c(H^*)} = \frac{4n-5}{2n} = 2 - \frac{5}{2n} \xrightarrow{n \rightarrow \infty} 2$$

□

- The main idea of the double-tree algorithm is:

- Find a cheap subgraph that is Eulerian

- Take the order of vertices of an Eulerian-Tour in that graph to compute the traveling salesperson tour

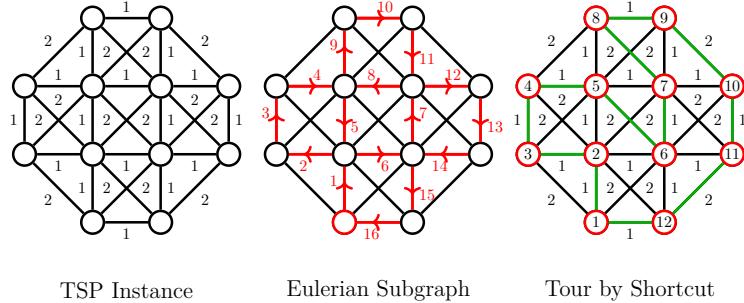


Fig. 11.8.: Structure of approximation algorithm (No. 1127)

- Eulerian-Subgraph:

Double-Tree: take a MST and double it

Christofides: take a MST, add edges with minimum cost s.t. all vertices have an even degree

- Christofides had this idea in 1976 and used a matching on all vertices with odd degree

Algo. 11.2 Christofides' Algorithm

Input: Complete Graph $G = (V, E)$, cost $c : E \rightarrow \mathbb{R}_+$ s.t. for every triple $i, j, k \in V$, the triangle inequality holds $c(i, k) \leq c(i, j) + c(j, k)$.

Output: Hamilton cycle H

Method:

Step 1: Compute a minimum spanning tree $T \subseteq G$

Step 2: Compute a minimum perfect matching M on $V_{\text{odd}} \subseteq V$. The set V_{odd} contains all vertices in T with odd degree $d_T(v)$, i.e., $V_{\text{odd}} = \{v \in T \mid d_T(v) \text{ odd}\}$

Step 3: Add M to T and compute an Euler-Tour Eu

Step 4: Skip over previously visited vertices to obtain a Hamilton cycle H from Eu
Return H

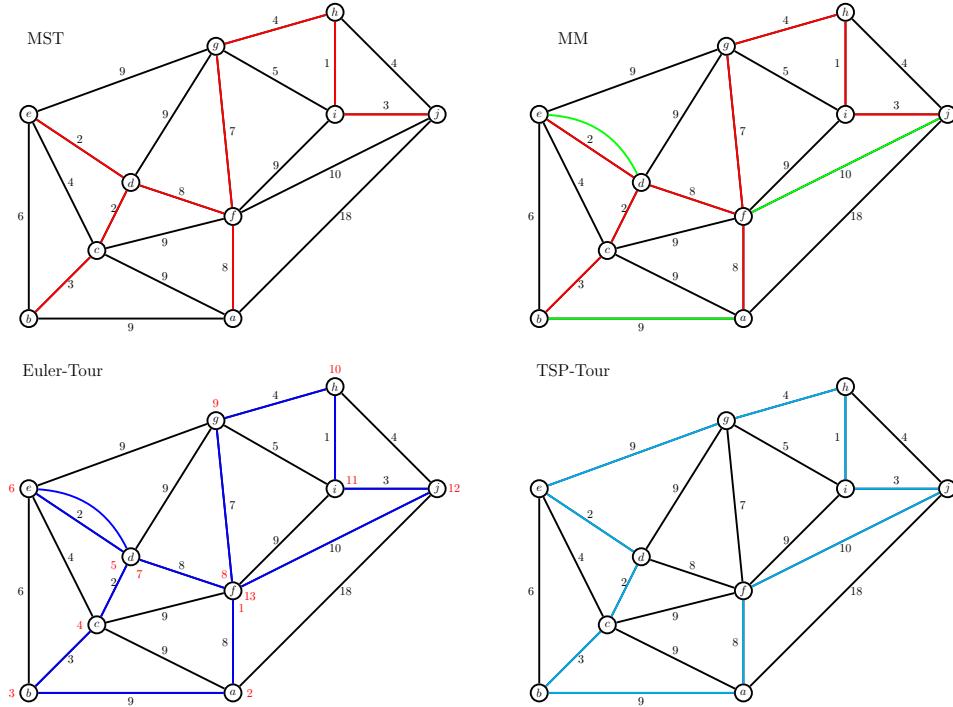


Fig. 11.9.: Christofides' Algorithm; the missing edges in the graph have cost according to the shortest path between all vertices (No. 263)

Theorem 218. *Christofides' algorithm for the metric traveling salesperson problem is a $3/2$ -approximation algorithm.*

Proof. Bound cost of T and M that form E by OPT

- *Claim:* Christofides' Algorithm computes a feasible tour H (Exercise)
- Let $H^* = v_1 \dots v_n$ be an optimal solution of the TSP problem
- Bound $c(T)$:
 - $H^* \setminus \{e\}$ is a tree $\forall e \in H^*$ and $c(e) \geq 0 \ \forall e \in E(G)$
 - $\Rightarrow c(H^*) \geq c(T)$ since T is a MST in G
- Bound $c(M)$:
 - The number of vertices with odd degree is even
 - \Rightarrow a minimum perfect matching M on V_{odd} exists
 - Reduce H^* to cycle H_{odd}^* on V_{odd}
 - Let $v_{i_1}, \dots, v_{i_{|V_{\text{odd}}|}}$ be the vertices with odd degree, ordered s.t. $v_{i_j} \leq v_{i_{j+1}}$ with $j = 1, \dots, |V_{\text{odd}}| - 1$
 - Then, $H_{\text{odd}}^* = v_{i_1} \dots v_{i_{|V_{\text{odd}}|}}$
 - Due to the triangle inequality, it holds $c(H_{\text{odd}}^*) \leq c(H^*)$
 - Since $|V_{\text{odd}}|$ is even, H_{odd}^* can be partitioned into two matchings $H_{\text{odd}}^* = M_1 \cup M_2$ with M_1 contains the odd edges, M_2 the even edges
 - Then

$$\begin{aligned} c(M) &\leq \min\{c(M_1), c(M_2)\} \\ &\leq 0.5 \cdot c(H_{\text{odd}}^*) \leq 0.5 \cdot c(H^*). \end{aligned}$$

- Let Eu be the Eulerian Tour computed in Step 3 according to the tree T and the matching M

- Then,

$$c(H) \leq c(Eu) = c(T) + c(M) \leq 1.5 \cdot c(H^*)$$

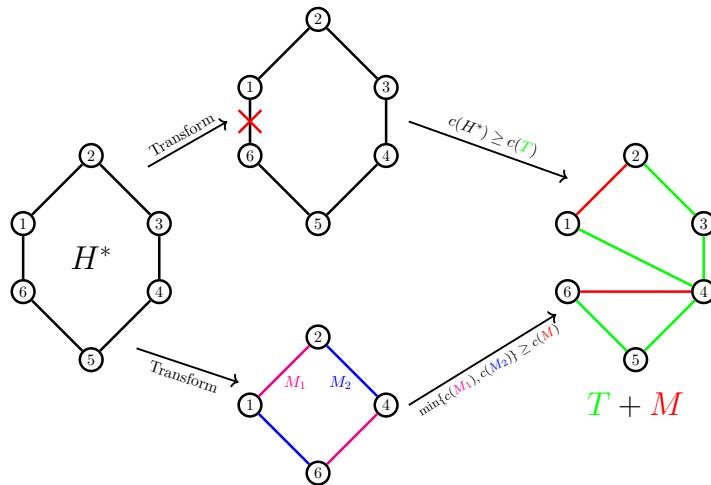


Fig. 11.10.: Bounding costs of MST and MM by the cost $c(H^*)$ of an optimal solution H^* . (No. 257)

□

- Also for this algorithm no better approximation ratio can be obtained

Theorem 219 (Cornuejols, Nemhauser, 1978 [?]). *Christofides' algorithm can not obtain a better approximation ratio than $3/2$.*

- In the new book [?] by Vera Traub and Jens Vygen, you find a detailed description of the TSP problem

Vertex Cover Problem

Definition 220. Minimum vertex cover problem

Given: Undirected graph $G = (V, E)$

Find: Vertex cover $S \subseteq V$ with minimum cardinality

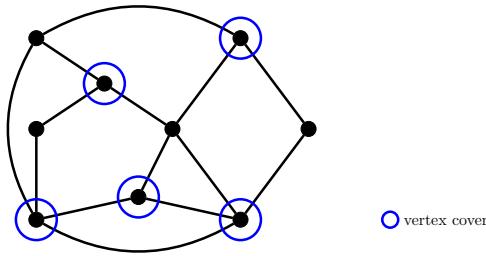


Fig. 11.11.: Vertex Cover (No. 487)

- The following algorithm was designed by Garvil in 1974 and uses matchings to obtain a constant approximation ratio

Algo. 11.3 Algorithm by GarvilInput: Undirected graph $G = (V, E)$ Output: A set of vertices $V' \subseteq V$

Method:

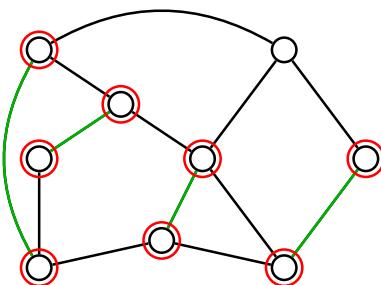
Step 1: Calculate a maximal matching M in G Step 2: Set $V' := \{v \in V \mid \exists uv \in M\}$ **return** V' 

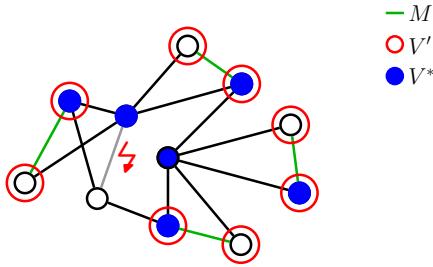
Fig. 11.12.: Algorithm by Garvil (No. 740)

- Note, the smaller the matching the smaller the vertex cover
- The runtime is clearly polynomial

Theorem 221 (Garvil 1974). *The algorithm by Garvil is a 2-approximation algorithm for the vertex cover problem.*

Proof. Maximality of M

- Let V' be a set of vertices computed by the algorithm and V^* be an optimal vertex cover

Fig. 11.13.: V' is a cover (No. 741)

- *Claim:* V' is a vertex cover.

Proof of Claim:

- Assume, $\exists uv \in E$ with $\{u, v\} \cap V' = \emptyset$
 - $\Rightarrow uv$ can be added to M , contradiction to the maximality of M $\square C$
 - Since V^* is a vertex cover, $\forall uv \in M$, we have $|\{u, v\} \cap V^*| \geq 1$
 - Since M is a matching, no vertex can cover two different edges
 - $\Rightarrow |V^*| \geq |M|$
 - Since $|V'| = 2 \cdot |M|$, Garvil's algorithm computes a 2 approximation
- \square
- Other, more “intelligent” algorithms have not yet led to approximation algorithms with a better approximation ratio for the vertex cover problem
 - In 2005, Dinur and Safra proved that no better approximation ratio than 1.3606 is possible unless $\mathbf{P} = \mathbf{NP}$ [?]

11.2. Absolute Approximation Algorithms

- Even better, in some cases, there is only a constant difference between the optimal solution and the approximation

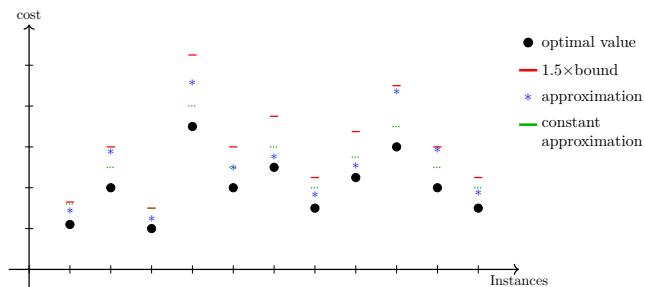


Fig. 11.14.: Constant approximation (No. 742)

Definition 222. An approximation algorithm A has an *absolute approximation ratio* with a constant d for a problem class Π if

$$c(x_{\text{ALG}}(I)) \leq c(x^*(I)) + d$$

for all instances $I \in \Pi$.

- There are only a few problems with a known absolute approximation ratio

Edge Coloring Problem

- One of the most prominent ones is one for the edge coloring problem

Definition 223. Edge Coloring Problem

Given: Undirected graph G

Task: Find an edge coloring of G with a minimum number of colors. An *edge coloring* with k colors is a mapping $f : E(G) \rightarrow \{1, \dots, k\}$, s.t. $f(e) \neq f(e')$ if e, e' have a common end point.

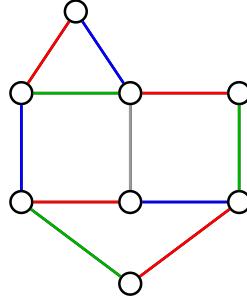


Fig. 11.15.: Edge Coloring (No. 743)

- In general the problem is hard to solve as proven by Holyer [?]

Theorem 224 (Holyer, 1981). *Edge coloring is NP-hard.*

Proof. Reduction from 3SAT and gadgets □

- Yet, edge coloring is one of the few problems that can be approximated with a constant factor
- Let G be an undirected graph. Then, $d_{\max}(G) = \max_{v \in V(G)} d(v)$ denotes the maximum degree of a node $v \in V(G)$.

Theorem 225 (Vizing 1964). *A simple, undirected graph G with maximum degree $d_{\max}(G)$ has an edge coloring with at most $d_{\max}(G) + 1$ colors and such can be constructed in polynomial time.*

Proof. Induction on number of edges

- *Base case:* Trivial for $|E(G)| = 0$
- *Induction Hypothesis:* For any graph G with $|E(G)| = m$ edges, $m \in \mathbb{N}$, there exists a coloring using $d_{\max}(G) + 1$ colors.
- *Inductive step:* Show property for graph G with $|E(G)| = m + 1$ edges
- Let $d_{\max}(G)$ be the maximum degree of G

- Consider an edge $\bar{e} \in G$ and define $G' := G - \bar{e}$

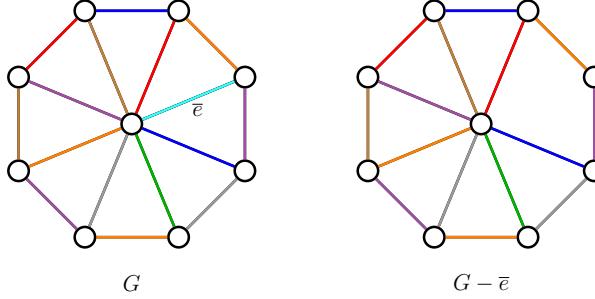


Fig. 11.16.: Remove, color, add (No. 744)

- *Case 1:* $d_{\max}(G') < d_{\max}(G)$.
 - $\Rightarrow d_{\max}(G') = d_{\max}(G) - 1$
 - $\stackrel{\text{I.H.}}{\Rightarrow}$ an edge coloring using colors $\{1, \dots, d_{\max}(G)\}$ for G' exists
 - Use color $d_{\max}(G) + 1$ for \bar{e}
- *Case 2:* $d_{\max}(G') = d_{\max}(G)$
 - \Rightarrow I.H.: \exists an edge coloring $c : E(G') \rightarrow \{1, \dots, d_{\max}(G') + 1\}$ for G'
 - Idea: recolor G' to obtain a “free” color for \bar{e} , if there not already exists one
 - Define $F_v(G')$ as the set of colors not used at v in G' according to the coloring c , i.e.,

$$F_v(G') = \{1, \dots, d_{\max}(G) + 1\} \setminus \cup_{e \in \delta(v) \cap E(G')} \{c(e)\}$$
 - Since $d_{\max}(G) + 1$ colors are available and $d(v) \leq d_{\max}(G)$, $|F_v(G')| \geq 1 \forall v \in V(G)$
 - Let $\bar{e} = x\bar{y}$ be the edge deleted
 - To \bar{e} we construct a series of edges xy_0, xy_1, \dots and a series of colors c_0, c_1, \dots as follows
 - Step 0: Set $i = 0$, $y_0 = \bar{y}$ with $\bar{e} = x\bar{y}$ and $G_{\text{aux}} = G'$
 - Step 1: Let $c_i \in F_{y_i}(G_{\text{aux}})$ be a missing color for y_i in G_{aux}
 - Step 2: If $\exists xy' \in \delta(x) \cap E(G_{\text{aux}})$ with $c(xy') = c_i$ do
 - Set $y_{i+1} = y'$, $i = i + 1$
 - Delete xy' from G_{aux}
 - Goto Step 1
 - Else Stop
 - Let xy_k be the last edge that was considered

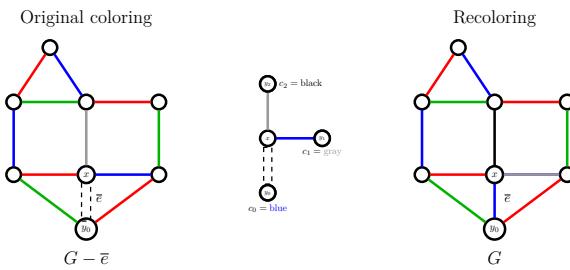


Fig. 11.17.: Sequence of edges (No. 745)

- Termination in two cases:

- Case a: $\nexists xy' \in \delta(x)$ with $c(xy') = c_k$
 - Define a new coloring $c' : E(G) \rightarrow \{1, \dots, d_{\max}(G) + 1\}$ with
$$c'(e) = \begin{cases} c_i & \text{if } e = xy, y \in \{y_0, \dots, y_k\} \\ c(e) & \text{otherwise} \end{cases}$$
 - Note, that $\bar{e} = xy_0$ is also colored by c'
 - Due to construction, c' is a feasible coloring of G
- Case b: $\exists xy_j \in \delta(x)$ with $c(xy_j) = c_k, j < k$
 - $\Rightarrow c_k \in F_{y_j}(G') \cap F_{y_k}(G')$ (e.g., c_k is red)
 - Let $\bar{c} \in F_x$ (e.g., \bar{c} is blue)
 - Due to construction $\bar{c} \neq c_k$
 1. If $\bar{c} \in F_{y_j}(G')$, set $c_j = \bar{c}$
 - Define a new coloring $c' : E(G) \rightarrow \{1, \dots, d_{\max}(G) + 1\}$ with
$$c'(e) = \begin{cases} c_i & \text{if } e = xy, y \in \{y_0, \dots, y_j\} \\ c(e) & \text{otherwise} \end{cases}$$
 - Due to construction, c' is a feasible coloring of G
 - 2. If $\bar{c} \in F_{y_k}(G')$, set $c_k = \bar{c}$
 - Define a new coloring $c' : E(G) \rightarrow \{1, \dots, d_{\max}(G) + 1\}$ with
$$c'(e) = \begin{cases} c_i & \text{if } e = xy, y \in \{y_0, \dots, y_k\} \\ c(e) & \text{otherwise} \end{cases}$$
 - Due to construction, c' is a feasible coloring of G
 - 3. If $\bar{c} \notin F_{y_k}(G') \cup F_{y_j}(G')$
 - Consider $G_{c_k \bar{c}} = (V, E_{c_k \bar{c}})$ with $E_{c_k \bar{c}} = \{e \in E(G') \mid c(e) \in \{c_k, \bar{c}\}\}$

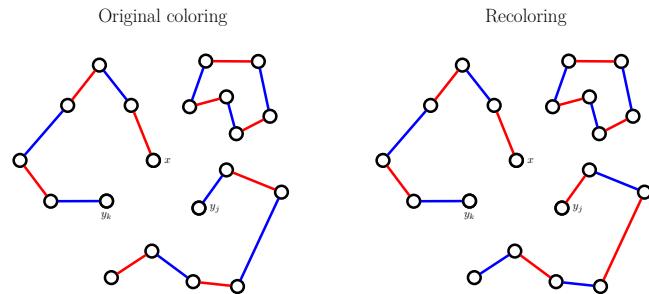


Fig. 11.18.: Red-blue components after the exchange of colors for the component with y_j (No. 747)

- Due to edge coloring, every vertex has degree ≤ 2
- \Rightarrow the connecting components are elementary cycles and elementary paths
- The vertices x, y_j and y_k are end nodes of paths, since $\bar{c} \notin F_{y_j}(G') \cup F_{y_k}(G')$ and $c_k \notin F_x(G')$
- \Rightarrow there exists one connected component containing only y_j, y_k or x (but not two of these nodes)
- Case I: Let y_j not be connected in $G_{c_k \bar{c}}$ with x or y_k

- Let p' be the path in $G_{c_k \bar{c}}$ connected with y_j
- Set $c_j = \bar{c}$
- Define a new coloring $c' : E(G) \rightarrow \{1, \dots, d_{\max}(G) + 1\}$ with

$$c'(e) = \begin{cases} c_i & \text{if } e = xy, y \in \{y_0, \dots, y_j\} \\ c_k & \text{if } e \in E(p'), c(e) = \bar{c} \\ \bar{c} & \text{if } e \in E(p'), c(e) = c_k \\ c(e) & \text{otherwise,} \end{cases}$$

- c' is a feasible coloring of G
- Case II: Let y_k not be connected in $G_{c_k \bar{c}}$ with x or y_j
 - As in Case I, just replace j by k
- Case III: Let x not be connected in $G_{c_k \bar{c}}$ with y_j or y_k
 - Let p' be the path in $G_{c_k \bar{c}}$ connected with x
 - Set $c_j = c_k$
 - Define a new coloring $c' : E(G) \rightarrow \{1, \dots, d_{\max}(G) + 1\}$ with

$$c'(e) = \begin{cases} c_i & \text{if } e = xy, y \in \{y_0, \dots, y_k\} \\ c_k & \text{if } e \in E(p'), c(e) = \bar{c} \\ \bar{c} & \text{if } e \in E(p'), c(e) = c_k \\ c(e) & \text{otherwise} \end{cases}$$

□

- The proof is constructive: we can obtain an algorithm from it to color every graph with a maximum degree of $d_{\max}(G)$ with $d_{\max}(G) + 1$ colors

Corollary 226. *For edge coloring, an absolute approximation algorithm exists with an absolute approximation ratio of 1.*

Proof. Construction in the Theorem 225

- Let $d_{\max}(G)$ be the maximum degree of a graph G
- Vizing's Algorithm: the number of colors needed is small or equal to $d_{\max}(G) + 1$
- Since every edges coloring of G needs at least $d_{\max}(G)$ colors
- \Rightarrow Vinzig's Algorithm is an approximation algorithm with an absolute approximation ratio of 1

□

- There are special cases where edge coloring is easy to solve

Theorem 227 (Gabow, 1976). *Edge coloring is polynomial solvable for bipartite graphs.*

The four color problem

- 23. Oct. 1852 wrote De Morgan to Hamilton: “A student of mine [Francis Guthrie] asked me today to give him a reason for a fact which I did not know was a fact - and do not yet. He says that if a figure be anyhow divided and the compartments differently colored so that figures with any portion of common boundary line are differently colored - four colors may be wanted, but not more - the following is the case in which four colors are wanted.”
- 26. Oct. 1852 Hamilton replied to De Morgan: “I am not likely to attempt your quaternion of color very soon.”
- Many solution attempts related the four-color problem to the “Problem of the five princes” which Möbuis (1790 - 1868) asked within his lecture on geometry
- An equivalent formulation of this problem “of the five princesses” is the following: “Once upon a time there was a queen reigning over a large kingdom. She had five wonderful children. In her last will, the queen said that after her death her children should divide the kingdom among themselves in such a way that the region belonging to each one should have a borderline (not just a point) in common with the remaining four regions. How should the kingdom be divided?”

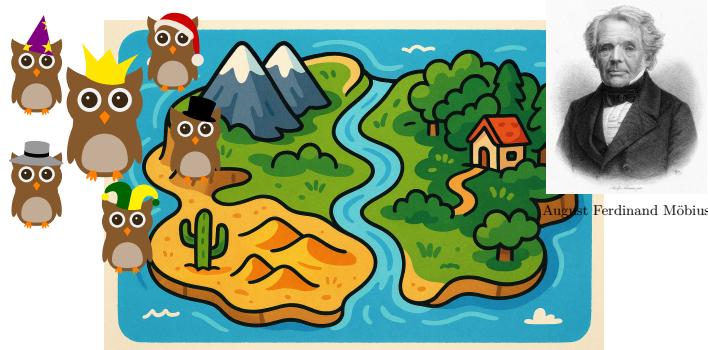


Fig. 11.19.: The problem of the five princesses (No. 836)

- However, unfortunately from a mathematical point of view no solution exists

Theorem 228 (Möbuis). *There exist no five regions which have common boundaries with one another.*

- If you use some creativity, the children can divide the kingdom
- However, this does not solve the four color problem (Exercise)
- Although the problem comes from geometry there is a close relation to a classical problem in graph theory

Definition 229. *Vertex Coloring Problem*

Instance: Undirected graph G

Task: Find a vertex coloring for G with a minimum number of colors. A *vertex coloring* with k colors is a mapping $f : V(G) \rightarrow \{1, \dots, k\}$ with $uv \in E(G) \Rightarrow f(u) \neq f(v)$.

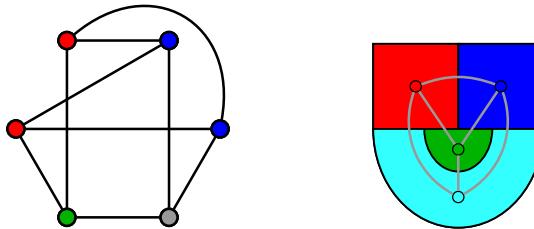


Fig. 11.20.: Vertex coloring (No. 748)

- The four color problem can easily be transformed into a vertex color problem:
 - Every region corresponds to a vertex
 - Two vertices are connected if the corresponding regions share a border
- Graphs obtained from such constructions are planar
- A graph is *planar*, if there is an embedding in the plane of the graph s.t. no two edges cross
- For general graphs, we know that the 3-coloring problem is **NP**-complete
- Unlike in other cases, considering planar graph does not change anything on the complexity
- First, we need again an auxiliary graph

Lemma 230. *The so called 'cross over' graph depicted below has the following properties:*

1. *In every proper 3-coloring of the 'cross over' graph, u and v have the same color, as do x and y*
2. *There exists both a proper 3-coloring in which u, v, x, y all get the same color and a proper 3-coloring in which x and y have a different color from u and v*

Proof. Case distinction

- Let f be a proper 3-coloring of the 'cross over' graph
- W.l.o.g. $f(z_1) = 0$
- $\Rightarrow f(z_i) \in \{1, 2\}$ for $i = 1, \dots, 5$
- W.l.o.g. $f(z_2) = f(z_4) = 1$ and $f(z_3) = f(z_5) = 2$
- *Case a:* $f(x) = 0$
 - Then, each vertex of the outer square uniquely determines the color of its clockwise next neighbor

- So, $f(x) = 0 \Rightarrow f(z_8) = 1 \Rightarrow f(u) = 0 \Rightarrow f(z_7) = 2 \Rightarrow f(y) = 0 \Rightarrow f(z_6) = 1 \Rightarrow f(v) = 0 \Rightarrow f(z_9) = 2$
- *Case b: $f(x) = 2$*
 - Then, each vertex of the outer square uniquely determines the color of its counterclockwise next neighbor
 - So, $f(x) = 2 \Rightarrow f(z_9) = 0 \Rightarrow f(v) = 1 \Rightarrow f(z_6) = 0 \Rightarrow f(y) = 2 \Rightarrow f(z_7) = 0 \Rightarrow f(u) = 1 \Rightarrow f(z_8) = 0$
- In both cases, u and v have the same color, as do x and y which proofs (i)
- In case *a*, u, v, x, y all have the same color and in case *b*, one obtains a coloring in which x and y have a different color from u and v which proofs (ii)

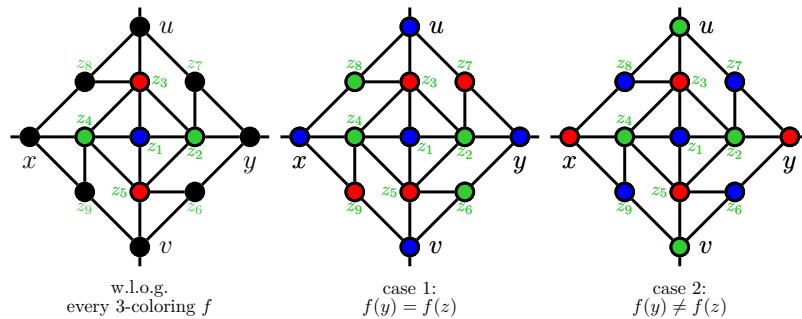


Fig. 11.21.: The 'cross over' graph and its proper 3-coloring (No. 864)

□

- With this 'cross over' graph, we can replace any crossing and transference colorings

Theorem 231 (Stockmeyer 1973 [?]). *3-coloring is NP-complete even on planar graphs.*

Proof. Reduction from 3-Coloring, 'cross-over' graph

- Step 1: 3-Coloring in planar graphs is in **NP**
 - Since 3-Coloring is in **NP**, also the problem on planar graphs is in **NP** (Theorem 197)
- Step 2: Polynomial reduction from 3-Coloring to 3-Coloring on planar graphs
 - Let I be a 3-Coloring instance
 - Construct an instance I' of the planar 3-coloring problem
 - Idea: Replace each crossing by a planar 'cross over' graph
 - Each edge involved in k crossings is cut into $k + 1$ segments by crossing
 - Add a new vertex on each segment
 - Replace crossing by a copy of 'cross over' graph attached by its terminals to the new vertices in the four segments incident to the crossing
 - For each original edge $\{w, z\}$, choose one endpoint and contract the edge between it and the vertex on the segment of $\{w, z\}$ incident to it

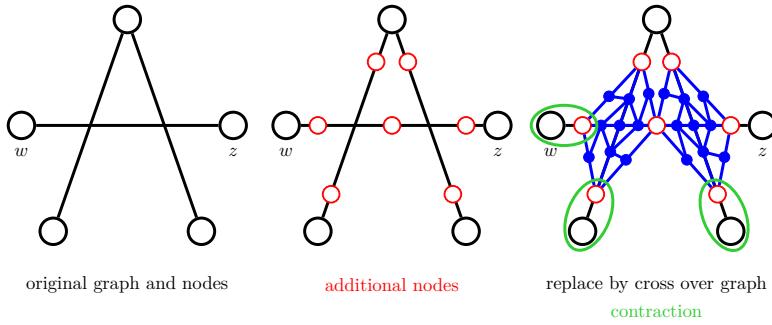


Fig. 11.22.: Replace crossings to obtain a planar graph (No. 863)

- An edge involved in no crossing returns to its original state
- *Claim:* The size of I' is polynomial in the size of I .
Proof of Claim
 - Number of crossings in a graph is polynomial bounded by the number of edges
$$\text{max number of crossings} \leq \binom{|E|}{2}$$
 - For each crossing, one adds at most 13 vertices and 24 edges
 - \Rightarrow the size of I' is polynomial bounded by the size of I $\square C$
- We now need to prove the equivalence
- *Claim:* If I is a yes-instance, then I' is a yes-instance.
Proof of claim
 - Consider a proper 3-coloring of the graph G

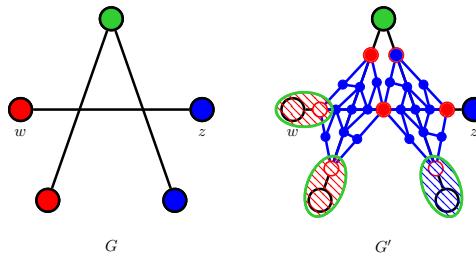


Fig. 11.23.: Transfere of coloring (No. 865)

- Start along each edge from the endpoint used in a copy of the 'cross over' graph
- Due to property (ii), one can extend this coloring to a proper coloring of G' $\square C$
- *Claim:* If I' is a yes-instance, then I is a yes-instance.
Proof of claim
 - Consider a proper 3-coloring of the new graph G'

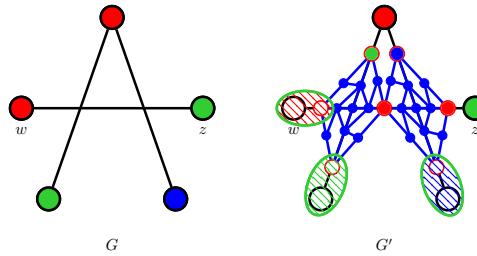


Fig. 11.24.: Transfere of coloring (No. 866)

- Due to property (i), the endpoints of each original edge to have different colors
- Restricting the coloring of G' to the vertices in G yields a proper 3-coloring of G $\square C$

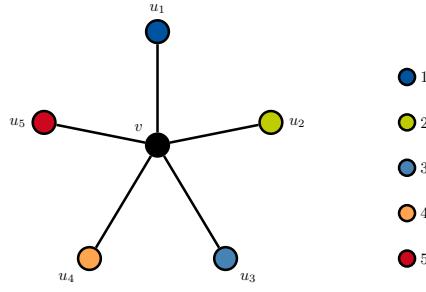
 \square

- Historically, in 1879 Alfred Kempe provided (among others) a proof for the four color problem
- In 1890 Heawood discovered a flaw in Kempe's proof
- Using Kempe's technique he was however able to prove the five color theorem

Theorem 232 (Heawood 1890). *Every planar graph can be colored with five colors.*

Proof. Contradiction, Kempe-chains

- Assume, there exists a planar graph that can not be colored with five colors
- Let $G = (V, E)$ be a graph with minimum number of vertices that has this property
- *Lemma:* Every planar graph contains a vertex v with $\delta(v) \leq 5$.
Proof: we omit the proof here, see the Appendix on planar graphs
- Let $v \in V$ be a vertex with $\delta(v) \leq 5$
- Consider $G' := G[V \setminus \{v\}]$, $G' = (V', E')$
- By choice of G , there exists a feasible coloring f' of G' with five colors
- *Claim 2:* It holds $\delta(v) = 5$ and all neighbors of v have a different color.
Proof of claim.
 - Assume, there are less than five colors in the neighborhood of v
 - Assign v a color that is not in its neighborhood
 - \Rightarrow feasible coloring of G with five colors (Contradiction) $\square C2$
- Let u_1, \dots, u_5 be the neighbors of v
- W.l.o.g. $f'(u_i) = i$ for $i = 1, \dots, 5$ and the vertices are ordered clockwise around v

Fig. 11.25.: v and its five neighbors (No. 875)

- Consider the subgraph $G^K(1, 3)$ that is induced by the vertices with color 1 and 3
- The connected components of $G^K(1, 3)$ are called Kempe-chains

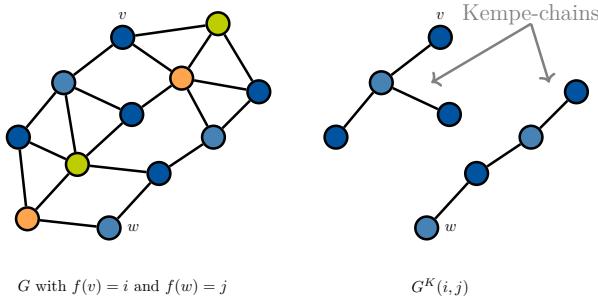
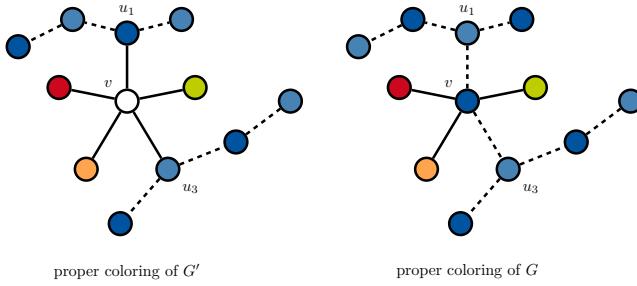


Fig. 11.26.: Kempe chains (No. 876)

- *Claim 3:* u_1 and u_3 lie in the same Kempe-chain.
Proof of claim.
 - Assume, u_1 and u_3 belong to the different Kempe-chains of $G'^K(1, 3)$

Fig. 11.27.: Proper coloring if u_1, u_3 in different Kempe-chains (No. 877)

- Swapping the colors along one of Kempe-chains leads to a feasible coloring f'' of G'
- It holds $f''(u_1) = f''(u_3)$
- \Rightarrow less than five colors in the neighborhood of v (Contradiction) $\square C3$
- Analogously, u_2 and u_4 lie in the same Kempe-chain
- Let $P_{1,3}$ be a (u_1, u_3) -path in G' that only consists of vertices colored in 1 or 3
- Let $P_{2,4}$ be a (u_2, u_4) -path in G' that only consists of vertices colored in 2 or 4

- *Claim 4:* G' can not contain both paths $P_{1,3}$ and $P_{2,4}$.

Proof of Claim 4

- Consider the path $P_{1,3}$
- Adding the edges $\{v, u_1\}$ and $\{v, u_3\}$ leads to a cycle C in G
- By construction, one of the vertices u_2 and u_4 lies inside of C and one outside of C

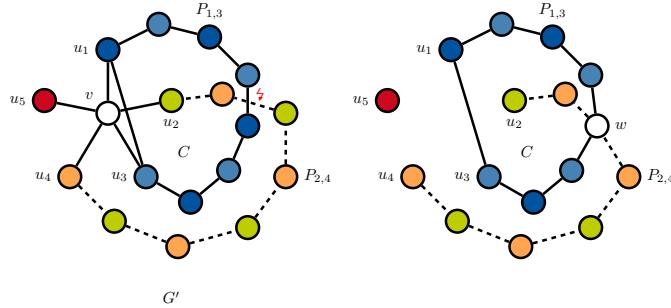


Fig. 11.28.: $P_{1,3}$ and $P_{2,4}$ (No. 878)

- \Rightarrow Every (u_2, u_4) -path has to cross C , especially $P_{2,4}$

- *Case a:* An edge of $P_{2,4}$ crosses an edge of C

- \Rightarrow Contradiction to G planar

- *Case b:* $P_{2,4}$ and C share a vertex w

- It holds $w \neq v$ since v is not contained in $P_{2,4}$

- $\Rightarrow v$ is contained in $P_{2,4}$ and $P_{1,3}$

- \Rightarrow Contradiction $\square C4$

- It follows that u_1, u_3 or u_2, u_4 lie in different Kempe-chains

- Swap the colors along one of these Kempe-chains

- \Rightarrow There are less than five colors in the neighborhood of v

- \Rightarrow There exists a feasible coloring of G with five colors. Contradiction

\square

- 1922 Philip Franklin proved that every planar graph with 25 vertices can be colored with 4 colors
- The size of colorable graphs increased to 27, 39, and 96 vertices
- During the 1960s and 1970s, German mathematician Heinrich Heesch developed methods of using computers to search for a proof
- Unfortunately, his project proposal got not funded by the DFG (Deutsche Forschungsgemeinschaft)
- The problem became now and then visible in public
- On April 1, 1975, Martin Gardner wrote in “Scientific American” an article about “Six sensational discoveries that somehow or another have escaped public attention”
- He wrote: “The most sensational of last year’s discoveries in pure mathematics was surely the finding of a counterexample to the notorious four-color-map conjecture... Last November, William McGregor, a graph theorist of Wappingers Fall, N.Y., constructed a map of 110 regions that cannot be colored with fewer than five colors.”

- It took one more year, to until the search for a proof ended with an happy end
- On June 21, 1976, Kenneth Appel and Wolfgang Haken from the University of Illinois used Heesch's approach and proved the four color theorem

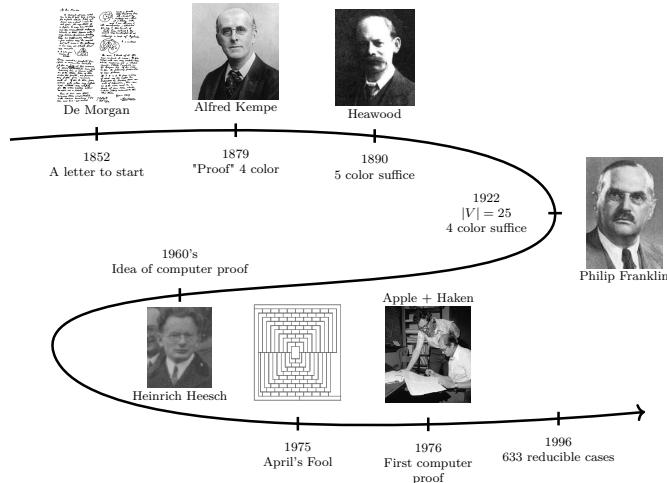


Fig. 11.29.: 4-color suffice (No. 838)

Theorem 233 (Appel & Haken 1976). *Every planar graph can be colored in polynomial time with at most four colors.*

- The proof is based on the coloring of 1.936 difficult cases
- It was the first major theorem to be proven by using a computer and was therefore not accepted by all mathematicians
- In 1996, Neil Robertson, Daniel P. Sanders, Paul Seymour, and Robin Thomas created a quadratic-time algorithm
- Here only 633 reducible configurations need to be checked

11.3. Approximation Schemes

- In general, there is a trade off between run-time vs. precision

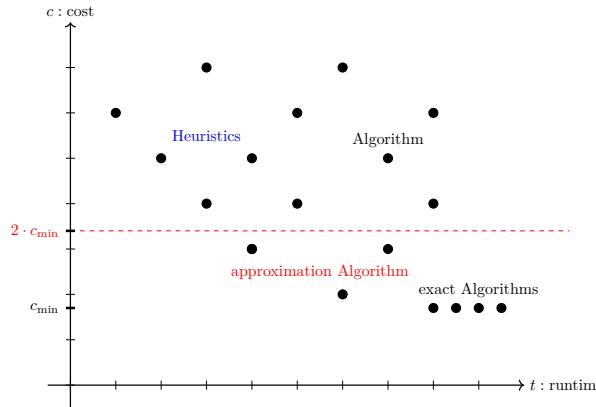


Fig. 11.30.: Classification of algorithms (No. 732)

- Approximation schemes allow to explicitly decide what is more important

Definition 234 (Polynomial Time Approximation Scheme (PTAS)). A *polynomial time approximation scheme* for a problem class Π (*with cost* $c \geq 0$) is an algorithm A that receives an instance $I \in \Pi$ and a number $\varepsilon > 0$ as input, and yields an $(1 + \varepsilon)$ -approximation algorithm, i.e., the runtime is polynomial in $\langle I \rangle$ and it has an approximation ratio of $(1 + \varepsilon)$.

- Note, that ε and $\frac{1}{\varepsilon}$ are considered as constant in the runtime analysis of a PTAS
- **Example:** A $(1 + \varepsilon)$ -algorithm A with runtime $\mathcal{O}(n^{\frac{1}{\varepsilon}})$ is considered as polynomial
- However, for small ε , the resulting runtime is often impractical
- A better scheme w.r.t. the runtime is the following

Definition 235 (Fully Polynomial Time Approximation Scheme (FPTAS)). A polynomial time approximation scheme is *fully polynomial* if its runtime is polynomial in $\langle I \rangle$ and $\frac{1}{\varepsilon}$.

- Difference between PTAS and FPTAS
 - The approximation ratio $(1 + \varepsilon)$ is the same in both cases
 - The runtime is different
 - A PTAS is polynomial in $\langle I \rangle$ for each fixed ε , however, can be exponential in $\frac{1}{\varepsilon}$, e.g., $\mathcal{O}(n^{\frac{2}{\varepsilon}})$
 - An FPTAS is polynomial in $\langle I \rangle$ and $\frac{1}{\varepsilon}$, e.g., $\mathcal{O}(\frac{n^2}{\varepsilon})$
- Only few **NP**-hard problems have an (F)PTAS, they belong to the easier **NP**-hard problem classes
- Note, any PTAS yields a constant approximation algorithm

Knapsack Problem

- We already considered the knapsack problem

Definition 236. Knapsack problem

Given:

- n items with integer weight w_i and integer profit p_i
- a knapsack with integer capacity c

Find: A subset $X \subseteq N$ with maximum profit $p(X) = \sum_{i \in X} p_i$ such that $w(X) = \sum_{i \in X} w_i \leq c$ holds true

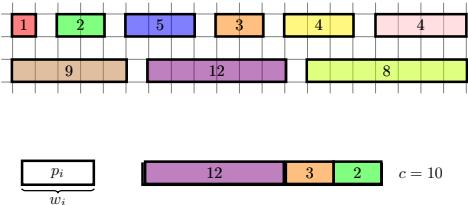


Fig. 11.31.: The Knapsack Problem and a solution (not optimal) (No. 255)

- In general, we assume $w_i \leq c$ for all $i = \{1, \dots, n\}$ in a given knapsack instance
- Next to subset sum the knapsack problem is one of the most basic weakly **NP-hard** problems

Theorem 237. *The decision version of knapsack is **NP**-complete.*

Proof. Reduction from subset sum to knapsack □

- For the construction of an FPTAS for the knapsack problem, we need two components:
 - a constant approximation algorithm
 - a pseudo-polynomial algorithm
- We start with the following simple greedy algorithm

Algorithm 11.1 Greedy approximation algorithm

Input: Set of elements $N = \{1, \dots, n\}$, weights $w_i \in \mathbb{N}$ and profit $p_i \in \mathbb{N}$ for all $i \in N$; capacity c

Output: Set of elements $X \subseteq N$

Method:

begin Set $X = \emptyset$

 Compute $u_i := p_i/w_i$ \ utility of each element

 Define an order i_1, \dots, i_n of elements, such that $u_{i_j} \geq u_{i_{j'}}$, $j \leq j'$

For $j = 1, \dots, n$ **do**

If $w(X) \cup w_{i_j} \leq c$ **do** Add i_j to X

 Set $k = \min \{j \in \{1, \dots, n\} \mid i_j \notin X\}$

If $p(X) < p(i_k)$ **do** Set $X = \{i_k\}$

Return X

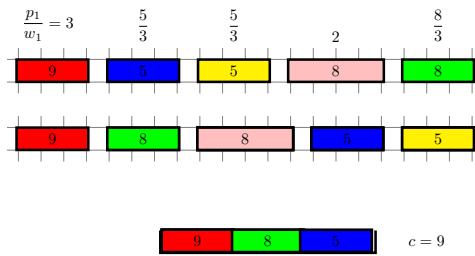


Fig. 11.32.: Greedy-Algorithm for the Knapsack Problem (No. 284)

- **Note**, the last step, i.e., comparison of $p(X)$ and $p(i_k)$, is important. Without that, we can't guarantee any approximation ratio. (Exercise)

Theorem 238. The Greedy approximation algorithm 11.1 is a 2-approximation algorithm for knapsack.

Proof. Bound of OPT by fractional solution

- Assume, all elements are ordered descending according to $u_i := \frac{p_i}{w_i}$
- Let $i < j \Rightarrow$ per unit of capacity, we obtain at least as much profit for i as for j
- Let X be the solution computed by the Greedy algorithm 11.1
- Let $k = \min \{j \in \{1, \dots, n\} \mid j \notin X\}$, i.e., first element not added to the knapsack
- Let us assume, we can add a fraction θ_k of element k s.t.

$$\sum_{i=1}^{k-1} w_i + \theta_k w_k = c$$

$$\bullet \Rightarrow \theta_k = \frac{c - \sum_{i=1}^{k-1} w_i}{w_k}$$

- Then, this is an optimal fractional solution X' (exchange argument)

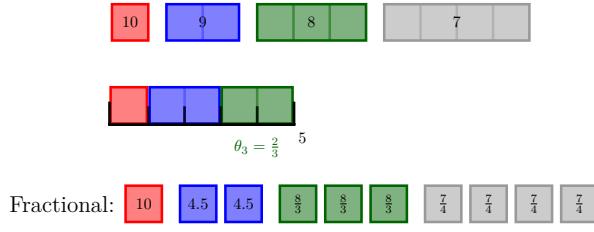


Fig. 11.33.: Greedy solution and fractional solution (No. 751)

- Let $\text{OPT}(I)$ be an optimal solution of the instance

$$\begin{aligned}
 \bullet & \Rightarrow \sum_{i=1}^{k-1} p_i + p_k \geq \sum_{i=1}^{k-1} p_i + \theta_k p_k \geq \text{OPT}(I) \\
 \bullet & \Rightarrow p(X) \geq \max \left\{ \sum_{i=1}^{k-1} p_i, p_k \right\} \geq \frac{1}{2} \left(\sum_{i=1}^{k-1} p_i + p_k \right) \geq \frac{1}{2} \text{OPT}(I)
 \end{aligned}$$

□

- Using this approach, we can obtain a simple upper bound on the maximum profit

Corollary 239. Let $\text{Greedy}(I)$ be the value of the solution computed by the greedy approximation algorithm 11.1. Then $P := 2 \cdot \text{Greedy}(I)$ is an upper bound $\text{OPT}(I)$.

Proof. Definition of approximation algorithm

□

- In a next step, we develop a pseudo-polynomial algorithm
- The algorithm is in principle similar to that of subset sum and based on dynamic programming
- Dynamic programming is a powerful technique to develop exact algorithms
- The idea is to break the “big” problem down to smaller problems and design out of the optimal solutions for the small problems an optimal solution of the big problem
- Bellman (1956) [?, ?] and Dantzig (1957) [?] introduced this dynamic program for the knapsack problem independently
- Instead of the classical setting they considered the following version
- *Fixed profit knapsack problem*

Given: • n items with integer weight w_i and integer profit p_i
 • a profit P

Find: A subset $X \subseteq N$ with minimum weight $w(X) = \sum_{i \in X} w_i$ such that $p(X) = \sum_{i \in X} p_i \geq P$ holds true

- Using binary search, we can solve the knapsack problem by the fixed profit knapsack problem

- Set $\bar{P} = \sum_{i=1}^n p_i$ and $\underline{P} = 0$

- Set $P = \left\lfloor \frac{\bar{P}+P}{2} \right\rfloor$
- Let X^P be an optimal solution for the fixed profit knapsack problem
- If $w(X^P) \leq c$, set $\underline{P} = P$ and $P = \left\lfloor \frac{\bar{P}+P}{2} \right\rfloor$
- If $w(X^P) > c$, set $\bar{P} = P$ and $P = \left\lfloor \frac{\bar{P}+P}{2} \right\rfloor$
- Repeat until $\bar{P} = \underline{P}$

i	1	2	3	4
p_i	7	5	2	1
w_i	10	8	3	1

$$c = 12$$

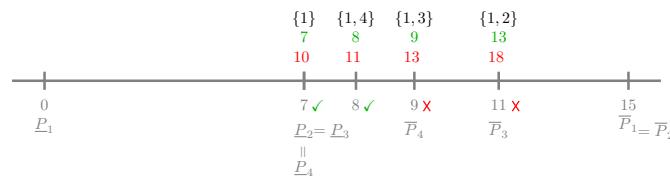
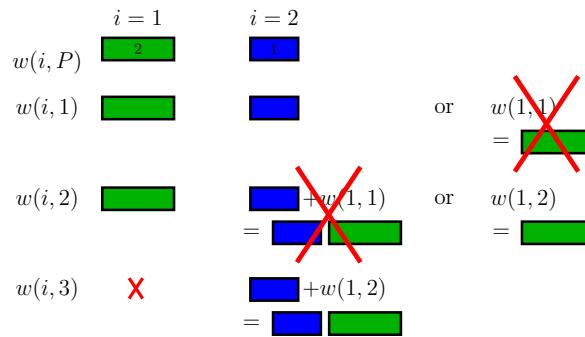


Fig. 11.34.: Bineary Search from fixed profit to the classical knapsack problem (No. 788)

- The following dynamic programming algorithms is basically due to Bellmann (1956) [?, ?] and Dantzig (1957) [?].
- Idea for the fixed profit knapsack problem
 - For $n = 1$ and some P , the problem is easy to solve
 - For $n = 2$ and some P : Can we use the case of $n = 1$, to solve it?
 - Option 1: take item 2 and cover $P - p_2$ with $\{1\}$
 - Option 2: don't take item 2 and cover P with $\{1\}$
 - Best of Option 1 and 2 is an optimal solution



$$\text{General: } w(i, P) = \min\{w(i-1, P), w(i-1, P - p_i) + w_i\}$$

Fig. 11.35.: Dynamic program for the fixed profit knapsack problem (No. 832)

- More general: $w(i, P)$: denotes the minimum total weight of a subset $S \subseteq \{1, \dots, i\}$ with profit $\sum_{i \in S} p_i \geq P$
- Compute $w(i, P)$ by
 - Option 1: take item i and cover $P - p_i$ with $\{1, \dots, i-1\}$ with minimum weight
 - Option 2: do not take item i and cover P with $\{1, \dots, i-1\}$ with minimum weight

- Out of the two, chose the better one
- Leads to a recursion formula

$$w(i, P) = \begin{cases} w(i-1, \max\{P - p_i, 0\}) + w_i & \text{if } w(i-1, \max\{P - p_i, 0\}) + w_i \leq w(i-1, P) \\ w(i-1, P) & \text{otherwise} \end{cases}$$

Algorithm 11.2 Dynamic Programming Knapsack (DPK) Algorithm

Input: Set of elements $N = \{1, \dots, n\}$, weights $w_i \in \mathbb{N}$ and profit $p_i \in \mathbb{N}$ for all $i \in N$;
 capacity c ; upper bound on maximum profit \bar{P} , e.g., $\bar{P} = \sum_{i=1}^n p_i$

Output: $X \subseteq N$ with $w(X) \leq c$ and $p(X)$ maximal

Method:

Step 1: • Set $X = \emptyset$

• Set $w(0, 0) := 0$ and $w(0, P) := \infty$ for $P = 1, \dots, \bar{P}$

// $w(i, P) = \text{minimum total weight } S \subseteq \{1, \dots, i\} \text{ with } \sum_{j \in S} p_j \geq P$

Step 2: **For** $i = 1, \dots, n$ **do**

For $P = 0, \dots, \bar{P}$ **do**

 • Set $x(i, P) := 0$ and $w(i, P) := w(i-1, P)$

For $P = p_i, \dots, \bar{P}$ **do**

If $w(i-1, P - p_i) + w_i \leq w(i-1, P)$ **then**

 • Set $w(i, P) := w(i-1, P - p_i) + w_i$ and $x(i, P) := 1$ // take item i

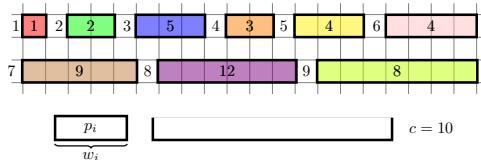
Step 3: Let $P = \max \{i \in \{0, \dots, \bar{P}\} \mid w(n, i) \leq c\}$. Set $X := \emptyset$

For $i := n, \dots, 1$ **do**

If $x(i, P) = 1$ **then**

 • Set $X := X \cup \{i\}$ and $P := P - p_i$

Return X



Profit\Item	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0				
1	∞	1	1	1	1	1				
2	∞	∞	2	2	2	2				
3	∞	∞	3	3	3	3				
4	∞	∞	∞	∞	2	2				
5	∞	∞	∞	3	3	3				
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots				
$P = 48$	∞	∞	∞	∞	∞	∞				

$w(i, P)$

Fig. 11.36.: The dynamic programming knapsack algorithm (No. 256)

Theorem 240. *The Dynamic Programming Knapsack Algorithm finds an optimal solution in $\mathcal{O}(n\bar{P})$ time.*

Proof. Recursion formula and induction (Exercise) □

- The DPK algorithm provides an optimal solution however the runtime is $\mathcal{O}(n\bar{P})$

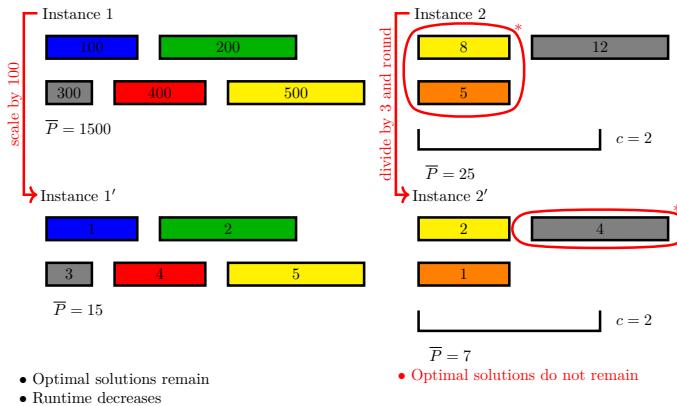


Fig. 11.37.: Runtime reduction by scaling (No. 1143)

- If we can “restrict” the upper bound \bar{P} to be polynomial in $\langle I \rangle$, the runtime is polynomial
- \bar{P} is small enough, if all p_i are polynomial in $\langle I \rangle$
- To that end, scale p_j to $\lfloor p_j/t \rfloor$ with some adequate t
- \Rightarrow reduces the runtime by a factor t at the cost of precision
- Rounding to integers is necessary since the pseudo-polynomial algorithm requires integer numbers
- Trading precision for runtime is a “standard trick” when developing approximation schemes

Algo. 11.4 Knapsack Approximation Scheme

Input: Set of elements $N = \{1, \dots, n\}$, weights $w_i \in \mathbb{N}$ and profit $p_i \in \mathbb{N}$ for all $i \in N$; capacity c ; $\varepsilon > 0$

Output: Set of elements $X \subseteq N$

Method:

Step 1:

- Apply the Greedy approximation algorithm 11.1
- Let X_1 be the resulting solution
- if $p(X_1) = 0$ then **return** $X := X_1$

Step 2: Define a new knapsack instance I'

- Set $t := \max\{1, \varepsilon p(X_1)/n\}$
- Set $p'_j := \lfloor p_j/t \rfloor$, $w'_j = w_j$ and $\bar{P} := \lfloor 2 \cdot p(X_1)/t \rfloor$

Step 3: Apply the DPK algorithm 11.2 to I' with upper bound \bar{P} and let X_2 be the resulting solution

return $X = \arg \max\{p(X_1), p(X_2)\}$

Theorem 241 (Ibarra & Kim 1975, Sahni 1976, Gens & Levner 1979). *The knapsack approximation scheme is a fully polynomial approximation scheme (FPTAS) for knapsack with runtime $\mathcal{O}(\frac{n^2}{\varepsilon})$.*

Proof. Scaling and rounding

- Let X^* be an optimal solution of the knapsack instance I
- Let X_{ALG} be the solution computed by Algorithm 11.4
- If the algorithm terminates in Step 1, then X_1 is optimal
- Hence, let $p(X_1) > 0$
- *Claim 1:* Set $\bar{P} := \lfloor \frac{2}{t} p(X_1) \rfloor$ with $t = \max \left\{ 1, \varepsilon \frac{p(X_1)}{n} \right\}$. Then, $\text{OPT}(I') \leq \bar{P}$.

Proof of Claim:

- According to the greedy algorithm, $p(X^*) \leq 2p(X_1)$
- Let X' be an optimal solution for I'
- \Rightarrow

$$\sum_{i \in X'} p'_i \stackrel{(a)}{\leq} \left\lfloor \sum_{i \in X'} \frac{p_i}{t} \right\rfloor \stackrel{(b)}{\leq} \left\lfloor \sum_{i \in X^*} \frac{p_i}{t} \right\rfloor = \left\lfloor \frac{1}{t} p(X^*) \right\rfloor \leq \left\lfloor \frac{2}{t} p(X_1) \right\rfloor = \bar{P}$$

(a) definition of p' (b) X^* is optimal solution of I

□C1

- \Rightarrow the DPK algorithm 11.2 computes an optimal solution for I'

- Let X_2 be this optimal solution for I'

- *Claim 2:* $p(X_2) \geq p(X^*) - n \cdot t$

Proof of Claim:

$$\begin{aligned}
p(X_2) &= \sum_{j \in X_2} p_j \geq \sum_{j \in X_2} t \left\lfloor \frac{p_j}{t} \right\rfloor = t \sum_{j \in X_2} p'_j = t \cdot p'(X_2) \\
&\stackrel{(a)}{\geq} t \cdot p'(X^*) = \sum_{j \in X^*} t \cdot p'_j \geq \sum_{j \in X^*} t \cdot \left(\frac{p_j}{t} - 1 \right) \\
&= \sum_{j \in X^*} (p_j - t) \geq p(X^*) - n \cdot t
\end{aligned}$$

(a) X_2 is optimal solution of I'

□C2

- If $t = 1$, then $I' = I$ and hence X_2 optimal for I

$$\bullet \text{ If } t > 1, \Rightarrow t = \varepsilon \frac{p(X_1)}{n}$$

- Claim 2 \Rightarrow

$$p(X_2) \geq p(X^*) - n \cdot t = p(X^*) - \varepsilon \cdot p(X_1)$$

- \Rightarrow

$$(1 + \varepsilon) \cdot p(X_{\text{ALG}}) = (1 + \varepsilon) \cdot \max \{p(X_1), p(X_2)\} \geq p(X_2) + \varepsilon \cdot p(X_1) \geq p(X^*)$$

- \Rightarrow for fixed ε , the algorithm is a $(1 + \varepsilon)$ -approximation algorithm

- Regarding the runtime:

- Step 1 needs $\mathcal{O}(n \log n)$
- Step 2 (dominating term): $\mathcal{O}(n \cdot \bar{P}) = \mathcal{O}\left(\frac{n}{t} \cdot p(X_1)\right) = \mathcal{O}\left(\frac{n^2}{\varepsilon}\right)$ with $t := \max\{1, \frac{\varepsilon}{n} \cdot p(X_1)\}$
- \Rightarrow the runtime is polynomial in n and $\frac{1}{\varepsilon}$

□

- Hence, knapsack can be approximated well
- Can we do even better, i.e., design an absolute approximation algorithm for the knapsack problem?

Theorem 242. *If $\mathbf{P} \neq \mathbf{NP}$, then no absolute approximation algorithm for the knapsack problem exists.*

Proof. Uses scaling

- Assume a polynomial algorithm A and a constant $k \in \mathbb{N}$ exist with $|A(I) - OPT(I)| \leq k$ for all instances I of knapsack
- Show: a polynomial algorithm exists which exactly solves knapsack - a contradiction to $\mathbf{P} \neq \mathbf{NP}$
- Let I be an instance of the knapsack problem
- Construct a new instance I' via $p'_j := (k + 1) \cdot p_j$, $w'_j = w_j$, $c' = c$
- I and I' have the same optimal solutions, since $(k+1)p(X) = p'(X) \forall X \subseteq \{1, \dots, n\}$.
- *Claim:* Algorithm A computes an optimal solution of I' .
Proof of Claim:
 - Let X_A be the solution computed by A and X^* be an optimal solution of I'

- Assume, $p'(X_A) < p'(X^*)$
- $\Rightarrow p(X_A) < p(X^*)$
- Since $p_i \in \mathbb{N} \Rightarrow p(X_A) + 1 \leq p(X^*)$
- \Rightarrow

$$\begin{aligned} |p'(X_A) - p'(X^*)| &= |(k+1)p(X_A) - (k+1)p(X^*)| \\ &= (k+1) \cdot |p(X_A) - p(X^*)| \\ &\geq (k+1) \end{aligned}$$

- Contradiction to $|p'(X_A) - p'(X^*)| \leq k$ $\square C$
- $\Rightarrow A(I') = OPT(I')$, i.e. A can solve I' and hence solves I exactly \square

Strongly NP-hard Problems and FPTAS

- The following negative result shows that only few NP-hard optimization problems can have an FPTAS
- Furthermore, it strengthens the difference between weakly and strongly NP-hardness

Theorem 243 (Garey & Johnson 1978). *Let Π be a strongly NP-hard optimization problem with the following characteristics*

- *the solutions have integer objective function values*
- *$OPT(I) \leq p(\langle I \rangle, \text{num}(I))$ for a polynomial p and all instances I .*

Then, Π has an FPTAS if and only if $\mathbf{P} = \mathbf{NP}$.

Proof. Scaling

- W.l.o.g. let Π be a minimization problem
- Assume an FPTAS A exists, i.e., for any instance $I \in \Pi$ and $\varepsilon > 0$ the algorithm A computes an $(1 + \varepsilon)$ -approximation
- Apply A to instance I with

$$\varepsilon := \frac{1}{p(\langle I \rangle, \text{num}(I)) + 1}$$

- Let $A(I, \varepsilon)$ be the value of the computed solution
- Since A is an $(1 + \varepsilon)$ -approximation algorithm,

$$\begin{aligned} A(I, \varepsilon) - OPT(I) &\leq (1 + \varepsilon)OPT(I) - OPT(I) \\ &= \varepsilon \cdot OPT(I) \\ &= \frac{OPT(I)}{p(\langle I \rangle, \text{num}(I)) + 1} \\ &\stackrel{(a)}{<} \frac{OPT(I)}{OPT(I)} = 1 \end{aligned}$$

- (a) due to the conditions within the theorem $OPT(I) \leq p(\langle I \rangle, \text{num}(I))$
- Since the objective is integer, $A(I, \varepsilon) - OPT(I) = 0$

- \Rightarrow each $(1 + \varepsilon)$ -optimal solution is optimal and A an exact algorithm for ε
- The runtime is polynomial in $\langle I \rangle$ and $\frac{1}{\varepsilon} = p(\langle I \rangle, \text{num}(I)) + 1$
- \Rightarrow it is pseudo-polynomial in $\langle I \rangle$
- \Rightarrow the strongly **NP**-hard problem Π has a pseudo-polynomial algorithm
- $\Rightarrow \mathbf{P} = \mathbf{NP}$

□

- There are several classical problems to which this result applies
 - Let n be the number of nodes, m be the number of edges, and c_{\max} be the maximum cost values of an instance for different problems
 - Then,

TSP	$\text{OPT}(I) \leq c_{\max} \cdot n$
Vertex Cover	$\text{OPT}(I) \leq n$
Edge Coloring	$\text{OPT}(I) \leq m + 1$
 - \Rightarrow for these problems no FPTAS exists unless $\mathbf{P} = \mathbf{NP}$



A. Mathematik lesen, lernen und schreiben

- Mathematik ist eine Sprache, um komplexe Zusammenhänge präzise zu formulieren und zu analysieren¹
- Mathematik ist für sich selbst eine Wissenschaft (reine Mathematik) und gleichzeitig die Grundlage für vielen Anwendungen wie in der Physik, Chemie, Wirtschaftswissenschaften oder Informatik (angewandte Mathematik)
- Ziel der Mathematik ist es Aussagen zu beweisen oder zu widerlegen
- Fantastisch an Mathematik ist, dass wir nicht sie glauben müssen
- Ohne Mathematik wären kürzeste Wege Berechnungen, Flüge zum Mond, der Bau von Energienetzwerken nicht denkbar
- Mathematiker*innen haben eine eigene Art zu denken
- Das ist am Anfang sehr verwirrend und es dauert eine Weile, bis man das versteht
- Ziel dieses Textes ist es, ein paar Hinweise zu geben wie
 - Mathematiker*innen denken
 - Wie man mathematische Texte liest
 - Wie man Mathematik lernt
 - Wie man Mathematik aufschreibt
- Hierzu ein paar erste Ratschläge
 - *Der Erfolg des Studiums hängt von Ihnen ab* - Ihr Handeln macht aus wie viel Sie von der Mathematik verstehen. Machen Sie die Übungen selbstständig? Wiederholen Sie den Vorlesungsstoff? Entwickeln Sie eigene Beispiele? Wenn ja, sollten Sie erfolgreich in der Mathematik sein. Ist es leicht? Nein!!! Aber es ist machbar.
 - *Seien Sie aktiv* - Durch “Lesen” und “Abschreiben” erreichen Sie nichts. Sie müssen aktiv Mathematik betreiben, um sie zu verstehen. Lassen Sie sich auch nicht alles vorkauen. Überlegen Sie selber, wo Sie noch Fragen oder Lücken haben und suchen Sie sich die entsprechenden Informationen.
 - *Lernen Sie nichts auswendig, sondern versuchen Sie zu verstehen* - Ein paar Begriffe müssen präzise gelernt werden. Viel wichtiger und viel schwerer ist es die Aussagen und Beweisstrategien zu verstehen. Bemühen sie sich. Geben sie nicht auf. Reden Sie mit anderen. Versuchen Sie das Gelernte in ihre eigenen Worte zu fassen und es anderen zu erklären. Wenn das gut gelingt, haben sie den Stoff verstanden.
 - *Arbeiten Sie mit anderen zusammen* - Tauschen Sie sich mit anderen aus. Das Mathematikstudium ist kein Wettbewerb. Forschung geschieht auch hauptsächlich im Team. Üben Sie das so früh wie möglich. Nur: einfach abschreiben bringt nichts. Arbeiten Sie gemeinsam.
 - *Seien Sie Neugierig und stellen Sie Fragen* - Gute Fragen zu stellen ist eine hohe Kunst und erfordert Mitdenken und Verstehen des Stoffes. Versuchen Sie möglichst früh alles zu hinterfragen und weiterführende Fragen zu stellen. Stellen Sie diese ihren KomilitonInnen oder Dozierenden. Suchen Sie nach Antworten. Eine der wichtigsten Kompetenzen in der Forschung ist es gut Fragen zu stellen.

¹Der gesamte Text basiert auf dem Buch “Wie man mathematisch denkt” von Kevin Houston [?] und ist ergänzt um meine Erfahrungen

A.1. Struktur mathematischer Texte und sie verstehen

- Mathematische Texte
 - bestehen aus einer Mischung aus “gewöhnlicher” Sprache und Formeln
 - zeichnet sich durch eine hohe Informationsdichte aus (wenige Füllworte oder Wiederholungen, Symbole mit Informationsgehalt, ...)
- Es gibt sehr viele Begriffe, die in der Umgangssprache nicht vorkommen (abstrakte Begriffe)
- Oft greift sie auf symbolische Darstellungen von Sachverhalten zurück
- Um präzise zu formulieren nutzt die Mathematik
 1. Symbole und
 2. Klare Struktur mathematischer Texte in: Definition/Theorem/Beweis; Zwischenexte werden selten verwendet
- Ein mathematischer Text besteht entsprechend aus vielen kleinen Informationsbrocken
- Grob gesprochen haben wir

Definitionen	eine Erklärung der mathematischen Bedeutung eines Begriffs
Satz/Theorem	eine sehr wichtige wahre und bewiesene Aussage
Proposition	eine weniger wichtige, aber immer noch interessante wahre und bewiesene Aussage
Lemma	eine wahre und bewiesene Aussage, die zum Beweis anderer wahrer Aussagen benötigt werden
Korollar	eine wahre Aussage, die auf einfache Weise aus einem Satz oder einer Proposition folgt
Beweis	die Begründung, warum eine Aussage wahr ist
Vermutung	eine Aussage, die für wahr gehalten wird, für die wir jedoch keinen Beweis haben
Axiom	eine grundlegende Voraussetzung über mathematische Gegebenheiten

Symbole

- Symbole helfen Text kürzer zu halten
- Oft ist es hilfreiche, Aussagen sowohl in Symbolen als auch in Worte zu fassen. Oft versteht man dabei die Aussagen/Definitionen/Texte besser

Symbol	Meaning	Examples
\forall	for all	$\forall x : x - x = 0$
\exists	there exists	$\exists k \in \mathbb{N} \text{ s.t. } k \cdot k = 4$
\in	is an element of	$1 \in \mathbb{N}$
\notin	is no element of	$\pi \notin \mathbb{N}$
\nexists	there exists no	$\nexists n \in \mathbb{N} \text{ such that } n + 1 = 0$
\Rightarrow	it follows	$n \text{ even} \Rightarrow \exists k \text{ with } n = 2 \cdot k$
\Leftrightarrow	equivalent	$n \text{ even} \Leftrightarrow n + 1 \text{ odd}$
$x \subseteq y$	x is a subset of or equal to y	$\{1, 2\} \subseteq \{2, 3, 1\}$
$x \subset y$	x is a proper subset of y	$\{2, 3\} \subset \{1, 3, 2\}$
$x \cup y$	union of x, y	$\{1\} \cup \{2\} = \{1, 2\}$
$x \cap y$	intersection of x, y	$\{1, 2\} \cap \{2, 3\} = \{2\}$
$x \Delta y$	symmetric difference	$x \cup y - x \cap y = x \Delta y$
\mathbb{N}	natural numbers	$42 \in \mathbb{N}; 0 \notin \mathbb{N}$
\mathbb{N}^0	natural numbers including zero	$0 \in \mathbb{N}^0, -1 \notin \mathbb{N}^0$
\mathbb{Z}	whole numbers	$-2 \in \mathbb{Z}, 0.2 \notin \mathbb{Z}$
\mathbb{Q}	fractional numbers	$0.5 \in \mathbb{Q}, \pi \notin \mathbb{Q}$
\mathbb{R}	real numbers	$e \in \mathbb{R}, -1 + i \notin \mathbb{R}$
\mathbb{C}	complex numbers	$3 + 2i \in \mathbb{C}$
\sum	sum	$\sum_{i=1}^n i = n(n-1)/2$
\prod	product	$\prod_{i=1}^3 i = 1 \cdot 2 \cdot 3$
$\{a, b, \dots, z\}$	set	$\{1, 2\} = \{2, 1\}$
$\{\{a, b\}, \{c, d\}, \{a, c\}\}$	multiset	
(a, b, \dots, z)	tuple, i.e., ordered set	$(1, 2) \neq (2, 1)$
$ M $	number of the elements in M	$ \{1, 2, 3\} = 3$

Tabelle A.1.: Common formalism in graph theory and combinatorics.

Definitionen

- Definition: Bestimmung eines Begriffs aus Grundbegriffen oder bereits definierten Begriffen
- Zweck von mathematischen Definitionen:
 - Präzise und unmissverständliche Kommunikation
 - wichtige und interessante mathematische Objekte mit einem eigenen Namen versehen
- Definitionen sind die Vokabeln der mathematischen Sprache
- Verstehen von Definitionen:
 1. Kenne ich alle in der Definition verwendeten mathematischen Begriffe genau?
 2. Ist der definierte Begriff oder ein dazu ähnlicher schon an anderer Stelle vorgekommen? Wo? Welche Rolle spielte er dort?
 3. Unterscheidet sich der Begriff von einem bereits bekannten Begriff? Worin? Wo kam der bereits bekannte Begriff vor? Welche Rolle spielt er dort?
- Lernen von Definitionen
 1. Lernen Sie die Definition präzise auswendig
 2. Überlegen Sie sich Standardbeispiele, an denen man alle in der Definition vorkommenden Eigenschaften klar erkennt; diese Standardbeispiele können Sie auch verwenden, um später Aussagen von Sätzen zu überprüfen

3. Machen Sie sich eine Vorstellung von der Allgemeinheit/Breite des definierten Begriffs durch: triviale Beispiele, einfache Beispiele, extreme Beispiele
4. Versuchen Sie neue Beispiele aus gegebenen zu konstruieren
5. Überlegen Sie sich “Nichtbeispiele”, d.h. ein einfaches Beispiel, für das die Definition nicht gilt

Theorem/Satz/Lemma/Proposition/Korollar

- Theorem/Satz/...: Wahre und bewiesene Aussage
- Grob gilt: Theorem wichtiger Aussage als Satz, Satz wichtigere Aussage als Lemma als Proposition als Korollar
- Weiter: Lemma, Theoreme, Sätze mit Namen sind wichtiger als ohne (Satz von Euler, Lemma von Zorn, Hauptsatz der Differenzial- und Integralrechnung)
- Wir reden im Folgenden von Theoremen, es gilt für alle anderen Arten auch
- Theoreme beinhalten mathematischen Sachverhalt und Aussagen über Eigenschaften von mathematischen Objekten oder Algorithmen
- Theoreme (und ihre Beweise) sind die wichtigsten Element der Mathematik
- Grundstruktur der Sätze ist meist “wenn Voraussetzungsteil dann Schlussfolgerungsteil”
 - Voraussetzungsteil: Angabe der Voraussetzungen unter denen der Satz gilt. Achtung: alles ist hier wichtig
 - Schlussfolgerungsteil: Folgerung aus den Voraussetzungen
- Verstehen/Lernen von Theoremen
 1. Trennen Sie Voraussetzungen und Schlussfolgerungen klar
 2. Lesen Sie jeden Ausdruck genau und denken Sie über dessen Bedeutung nach
 3. Schreiben Sie das Theorem in Textform und in symbolischer Form
 4. Zeichnen Sie ein Bild, um sich den Sachverhalt klarer zu machen. Gerade in der kombinatorischen Optimierung und Graphentheorie ist dies oft gut möglich
 5. Bewerten Sie die “Stärke” des Theorems: Schwache Voraussetzung, starke konkrete Schlussfolgerung; Starke komplexe Voraussetzung, schwache Schlussfolgerungen
 6. In welcher Beziehung steht das betrachtete Theorem zu bereits bekannten? Wie unterscheiden sich die Voraussetzungen und Schlussfolgerungen von anderen Theoremen? Sind sie mehr oder weniger einschränkend? Sind sie so stärker oder schwächer?
 7. Was bedeutet das Theorem “wirklich”? Liefert es einen Algorithmus (oder ggf. der Beweis)? Klassifiziert es mathematische Objekte nach ihren Eigenschaften? Stellt es neue Zusammenhänge her?
 8. Warum gilt das Theorem **nicht**, wenn eine der Voraussetzungen wegfällt? An welchen Stellen des Beweises wird welche Voraussetzung verwendet?
 9. Wenden Sie das Theorem auf verschiedene Beispiele und Nichtbeispiele an
 10. Falls die Aussage eine Implikation ist: Gilt ihre Umkehrung?
 11. Versuchen Sie Verallgemeinerungen des Satzes zu formulieren

Beweise

- Ein Beweis ist eine fehlerfreie Herleitung einer mathematischen Aussage basierend auf Axiomen, die als wahr vorausgesetzt werden, und Aussagen, die bereits bewiesen sind
- Beweise sind unsere Garantie dafür, dass unsere Lösungen korrekt sind
- Beweise sind meist in kleine Schritte gegliedert:
 - Ausgangspunkt der Beweisschritte sind ist der Voraussetzungsteil der Aussage
 - Dann gehen wir in kleinen logischen Schritten voran
 - Jeder Schritt folgt auf den vorherigen
 - Am Ende steht die Aussage
- Der Maßstab für die Richtigkeit des Beweises ist dabei die mathematische Logik
- Beweise sind schwer zu verstehen:
 - Beim Aufschreiben von Beweisen wird ein Großteil der ursprünglichen Arbeit weggelassen
 - Meist sind das die Überlegungen, die dem*der Beweisführenden halfen, den Beweis zu finden
 - Da Mathematiker*innen sich gerne kurz und präzise ausdrücken, finden diese extra Touren keinen Platz in den Beweisen
- Warum brauchen wir Beweise?
 - So können wir uns 100%ig auf die Aussagen verlassen
 - Ohne Beweise verliert die Mathematik ihre Schlagkraft und wird beliebig
 - Wir brauchen nicht darauf vertrauen, dass andere für uns denken
 - Beweise zeigen, wie wir ebenfalls denken können
- Beweise zu finden ist schwer
 - Genauso wie das Lösen von Problemen Übung erfordert, benötigt das Beweisen von Aussagen viel Übung
 - Je mehr Beweise Sie lesen und analysieren und je mehr Sie selber aufschreiben, desto besser werden sie
- Es gibt verschiedene Beweistechniken (direkter Beweis, Widerspruchsbeweis, vollständige Fallunterscheidung, vollständige Induktion, ...)
- Achtung: Beweis per Beispiel ist keine Beweistechnik. Es kann helfen sich einen Beweis an einem Beispiel zu verdeutlichen. Für die Wahrheit einer Aussage reicht es nicht, dass die Aussage für ein Beispiel stimmt.
- Verstehen/Lernen von Beweisen
 1. Überfliegen Sie zuerst den Text, um einen Überblick erhalten. Was ist wichtig? Was ist die Grundidee?
 2. Zerlegen Sie den Beweis in logische unabhängige Abschnitte. Können Sie diese Zwischenaussagen auch anders beweisen?
 3. Gehen Sie jeden Schritt durch: Warum gilt der nächste Schritt? Auf welchen Voraussetzungen/ bekannten Aussagen basiert er? Falls andere Aussagen verwendet werden, überprüfen Sie, ob alle Voraussetzungen dafür vorhanden sind. Das Ende eines Beweises wird meist mit \square oder q.e.d. (quod erat demonstrandum, was zu beweisen wäre) gekennzeichnet.
 4. Visualisieren Sie sich die einzelnen Schritte und wenden Sie diesen auf ein einfaches Beispiel an; wenden Sie den Beweis auf ein Nichtbeispiel an.

5. Überprüfen Sie: sind alle Fälle abgedeckt? Wo genau liegt der Widerspruch? Suchen Sie nach Fehlern: testen Sie alle Extremfälle. Gehen Sie erst mal davon aus, dass alles falsch ist.
6. Ordnen Sie den Beweis in die unterschiedlichen Beweistechniken ein: ist es eine Berechnung, ein direkter Beweis, eine Induktion, eine Fallunterscheidung, ...
7. Ist der Beweis konstruktiv? Kann ich aus dem Beweis Algorithmen/Strategien ableiten?
8. Gehen Sie noch mal zurück: Ähnelt der Beweis anderen?

Algorithmen

- Ein Algorithmus ist eine eindeutige Handlungsvorschrift, die aus einer Eingabe (Input) eine Ausgabe (Output) generiert und von oben nach unten, von links nach rechts durchgeführt wird.
- Verstehen/Lernen von Algorithmen
 1. Wählen Sie sich ein kleines Beispiel und wenden Sie den Algorithmus darauf an
 2. Sind alle Schritte klar und wohl definiert? Versuchen Sie den Algorithmus in “Fallen” zu locken
 3. Können Sie Teile des Algorithmus auch anders erhalten? Wie genau? Wie aufwendig sind die anderen Ansätze?
 4. Auf welchen theoretischen Eigenschaften beruht die Korrektheit des Algorithmus?

A.2. Überprüfung Ihres Verständnisses

- Woran erkennen Sie, dass Sie etwas wirklich verstanden haben?

Sie verstehen Definitionen, wenn Sie

- sie präzise wiedergeben können (Vokabeln!)
- sie in ihren eigenen Worten wiedergeben können
- konkrete Beispiele für sie angeben können, und zwar sowohl triviale als auch nicht-triviale Beispiele
- Nichtbeispiele für die Definition angeben können
- sie auch in anderen und ungewohnten Situationen erkennen können
- Sätze kennen, in denen sie angewandt werden können
- wissen, warum sie in diesen Sätzen angewandt werden können
- wissen, warum genau diese spezielle Definition aufgestellt wurde
- andere, ähnliche Definitionen für denselben Begriff und die Unterschiede zwischen ihnen kennen

Sie verstehen Theoreme, wenn Sie

- sie präzise und in ihren eigenen Worten wiedergeben können
- konkrete Beispiele für ihre Anwendung angeben können
- ihren Beweis verstehen
- sie in neuen und ungewohnten Situationen anwenden können
- Gegenbeispiele zu Aussagen angeben können, die durch Abschwächung der Voraussetzungen entstehen
- ihre Inversion und seine Umkehrung formulieren können
- einige seiner Folgerungen (z. B. Korollare) kennen
- sie mit wenigen Worten zusammenfassen können
- wissen, wie sie in eine Theorie passen
- wissen, ob sie sich auf eine kleinere oder eine große Klasse von Objekten beziehen

Sie verstehen Beweise, wenn Sie

- sie präzise und in Ihren eigenen Worten wiedergeben können
- wissen, an welchen Stellen die Voraussetzungen eingehen
- ihre Struktur kennen, also wissen, aus welchen Teilen sie bestehen und welche Techniken (z.B. direkter oder Widerspruchsbeweis) darin verwendet werden
- jeden Schritt als einfach und nicht als ein Wunder betrachten,
- ihre Ideen in Ihren eigenen Beweisen anderer Aussagen verwenden können
- jede Lücke füllen können
- wissen, wie streng die Beweise sind
- sie zusammenfassen (also die wesentlichen Eckpunkte ohne die Details wiedergeben) können
- wissen, wo Problem entstehen, wenn Voraussetzungen weggelassen werden

Sie verstehen ein größeres Teilgebiet, wenn Sie

- sehen können, wie alles zusammenpasst
- eine durch leichte Änderungen der Definitionen entstandene Theorie diskutieren können
- das Teilgebiet in einem Satz zusammenfassen können
- ein konkretes, für das Teilgebiet charakteristisches Beispiel angeben können
- Verbindungen, Ähnlichkeiten und Unterschiede zwischen diesem und anderen Teilgebieten sehen können
- problemlos zwischen einem intuitiven Verständnis und technischen Details einer Beweisführung hin- und herwechseln können
- die Schlüsseldefinitionen oder -sätze kennen,
- wissen, warum es interessant und nützlich ist
- wissen, welche Ideen darin immer wieder benutzt werden
- es ohne Verwendung schriftlicher Aufzeichnungen erklären können
- es einem anderen erklären können

A.3. Aufbau eines Papers/einer mathematischen Publikation

- Um ein mathematische Paper zu verstehen, hilft die es den allgemeinen Aufbau zu kennen und zu wissen wofür die einzelnen Teile stehen
- Dabei beantworten die einzelnen Teile wichtige Fragen der Leser
- **Abstract** - Teaser für Paper
 - Was ist die Motivations für das Paper?
 - Warum sollte ich das Paper lesen?
 - Was sind die wichtigsten Ergebnisse?
- **Introduction** - Motivation und Ergebnisse
 - Warum ist das Thema wichtig? Motivation (ausführlicher) mit Quellenangaben
 - Welches Problem wird genau betrachtet ? Manchmal schon mit der formalen Definition
 - Welche offenen Fragen haben gibt es? Welche Fragen sind schon gelöst?
 - Was sind die wichtigsten Resultate unsere Papers und warum sind sie wichtig?
 - Wie ist das Paper aufgebaut?
- **Literature Review** - Einbettung des behandelten Problems in die Literatur:
 - Welche ähnlichen Probleme gibt es?
 - Welche Ergebnisse/Ansätze gibt es zu den Problemen?
 - Wie unterscheidet sich das Problem von dem
- **Problem Description/Notation** - Formale Beschreibung des Problems
 - Welche mathematische Notation verwende ich?
 - Wie genau ist mein Problem formal definiert? (Wenn das nicht schon in der Einleitung passiert sind)
 - Was sind erste kleine Erkenntnisse? Ggf. erste kleine Beispiele oder Gegenbeispiele
- **Theroetical Results/Algorithms** - Hauptteil des Papers, oft in kleinere Unter- teile oder Sections strukturiert
 - Was sind die Ergebnisse?
 - Warum sind die Ergebnisse wichtig?
 - Sind die Ergebnisse korrekt? (Beweise?) Benötige ich gute Hilfslemma oder kleine weiteren Definitionen?
 - Was können wir von den Ergebnissen ableiten?
 - Wie können wir sie algorithmische/für weitere strukturelle Analysen nutzen?
- **Rechenstudie** - Funktioniert die Theorie auch in der Praxis
 - Wie haben wir die Algorithmen implementiert? Java, C++
 - Auf welcher Rechnerarchitektur/mit welchen Hilfsprogrammen haben wir unser Algorithmen rechnen lassen?
 - Wie sind die Testinstanzen zu Stande gekommen? Wie viele sind es?
 - Was sind die Ergebnisse? Welche Erklärungen gibt es für die Ergebnisse?
- **Conclusion** - Zusammenfassung und wie es weiter gehen kann

- Was sind die wichtigsten Ergebnisse?
- Was sind weiter offene Fragen?
- **References** - Quellenangaben
 - Zitiert alles, was ich im Text nutze
- **Appendix** - für die Experten
 - Welche Beweise sind zwar wichtig, aber für das Paper zu lang?
 - Welche Rechenergebnisse (Tabellen) würden zu viel Platz einnehmen/den Lesefluss mindern?

A.4. Schritte zum Erarbeiten eines mathematischen Textes

- Mathematische Texte liest man nicht wie Harry Potter
- Zum Lesen einer Seite mathematischen Textes kann man durchaus eine Stunde benötigen
- Wichtig: mathematische Texte kann man nicht “querlesen”
 - Symbole enthalten wichtige Informationen
 - Das Weglassen einzelner Worte / Voraussetzungen kann zu vollkommen falschen Aussagen führen
- Als Leser sollten Sie aktiv bleiben
 - haben Sie immer Papier und Stift bereit, um eigene Beispiele zu entwerfen, das geschriebene zu hinterfragen, sich Skizzen zu machen
 - Prüfen Sie die Aussagen genau (es gibt immer wieder Fehler in Lehrbüchern, Skripten und wissenschaftlichen Artikeln)
- Die folgenden Schritte können zur Erarbeitung des Inhalts eines kürzeren Textes oder eines Kapitels in einem Buch verwendet werden
 1. Auf Überblick lesen
 - Was wird in welcher Reihenfolge behandelt?
 - Welche Definition, Aussagen etc. sind wesentlich, wie hängen sie zusammen?? Zählen sie wie oft eine Aussage/eine Definition im Text vorkommt
 - Gibt es Beispiele?
 - Suchen Sie nach Sätzen/Formeln/ oder Algorithmen, die es Ihnen erlauben die Aussage an einem Beispiel zu überprüfen und darauf anzuwenden. Anwenden ist eine sehr effektive Methode, um ein Gebiet kennenzulernen
 - Lassen Sie die Beweise erst mal aus
 2. Fragen stellen
 - Zu welchen wesentlichen Ergebnissen führt der Text?
 - Auf welchem Weg erhält man die Ergebnisse?
 - Weshalb sind Definitionen so und nicht anders formuliert?
 3. Detailliertes sorgfältiges Lesen
 - Stellen Sie sicher, dass Sie alle Fachbegriffe und Symbole verstehen (jedes einzelne!!)
 - Denken Sie mit und folgen Sie dem Argumentationspfad

- Prüfen Sie die Aussagen und Behauptungen: füllen Sie ggf. Lücken, die im Text gelassen wurden
- Rechnen Sie Beispiele, die entweder vorhanden sind oder entwickeln Sie eigene
- Lösen Sie Übungsaufgaben, falls diese vorhanden sind
- Lesen Sie die Beweise
- Wenn Sie einen einzelnen Punkt nicht verstehen: Lesen Sie weiter und gehen Sie später zurück. Lassen Sie sich davon nicht frustrieren. An vielen Stellen brauchen wir etwas länger, um einen Sachverhalt zu verstehen oder ein Konzept komplett zu durchdringen.
- Strukturieren Sie sich den Text: markieren Sie alle Definitionen und verwendete Symbole/Mengen, Theoreme/Sätze/Lemmata, Beweise und Algorithmen (am besten in unterschiedlichen Farben). So sehen Sie die Struktur leichter

4. Zusammenfassen

- Schreiben Sie mit eigenen Worten eine knappe Zusammenfassung
- Zeichnen Sie sich eine Mind-Map: welche Begriffe werden in welchem Theorem verwendet? Auf welchem Theorem basieren die Algorithmen? Wie hängen die einzelnen Begriffe zusammen?

A.5. Beweistechniken

- Es gibt einige Beweistechniken, die immer wieder vorkommen
- Lernen Sie diese
- Ordnen Sie einen Beweis/Teile eines Beweises den unterschiedlichen Techniken zu
- Versuchen Sie eine andere Technik für einen schon bekannten Beweis zu verwenden, um das Beweisen zu üben

Direkter Beweis

- Der Beweis folgt dem Ansatz: Wenn A gilt, dann folgt daraus B , und daraus folgt C
- Jede einzelne Implikation sollte offensichtlich sein

Beweis durch Fallunterscheidungen

- Eine große Aussage wird in kleinere Möglichkeiten aufgeteilt
 - Fall 1: Sei a gerade
 - Fall 2: Sei a ungerade
- Diese Teilstücke können dann separat betrachtet werden
- Wichtig ist, dass alle möglichen Fälle abgedeckt werden
- Fallunterscheidungen können auch in Definitionen eingesetzt werden

Widerspruchsbeweis

- Wir nehmen an, dass die zu zeigende Aussage falsch ist
- Dann bauen wir eine Kette logischer Folgerungen auf, die am Ende zu einer weiteren Aussage führt, von der wir definitiv wissen, dass sie falsch ist
- Ein alternative Name ist *reductio ad absurdum*
- Beim Aufschreiben eines Widerspruchsbeweises können sich an Folgendem orientieren
 1. Erklären Sie, dass Sie annahmen, die Aussage sei falsch
 2. Erläutern Sie unter Verwendung der Negation, was die Falschheit der Aussage bedeutet
 3. Erarbeiten Sie, was daraus folgt, bis Sie zu einem Widerspruch kommen
 4. Verkünden Sie nun, dass der Widerspruch aufgetreten ist

Vollständige Induktion

- Vollständige Induktion erscheint erst Mal verwirrend, weil man das Gefühl hat man geht davon aus, dass die Aussage stimmt und beweist sie dann
- Sie hilft Aussagen zu beweisen, die durch natürliche Zahlen indiziert sind
- Das Prinzip beruht auf dem folgenden Theorem

Theorem *Es sei $A(n)$ für $n \in \mathbb{N}$ eine unendliche Folge von Aussagen. Es gelte*

1. *$A(1)$ ist wahr und*
2. *$A(k) \Rightarrow A(k+1)$ für alle $k \in \mathbb{N}$.*

Dann gilt $A(n)$ wahr für alle $n \in \mathbb{N}$

- Das Theorem lässt sich in folgende Beweistechnik überführen
 1. Stelle die Aussage $A(n)$ auf
 2. Beweise, dass $A(1)$ oder $A(0)$ wahr ist (= Induktionsanfang)
 3. Überprüfe $A(k) \Rightarrow A(k+1)$ (= Induktionsschritt/ Induktionsschluss). Hier kann insbesondere $A(k)$ ist wahr angenommen und verwendet werden
- Beim Aufschreiben eines Induktionsbeweise können Sie sich an Folgendem orientieren
 1. Kündigen Sie an, dass Sie Induktion verwenden
 2. Führen Sie den Induktionsanfang durch
 3. Erklären Sie, dass Sie annehmen, die Aussage sei für ein k wahr. Oft ist es hilfreich, diese Aussage für den späteren Gebrauch explizit hinzuschreiben.
 4. Nutzen Sie die Richtigkeit der Aussage für k im Beweis der Aussage für $k+1$ aus. Oft bedeutet dies, einen mathematischen Ausdruck in zwei Teile zu zerlegen, von denen einer den Fall für k enthält. Vergessen Sie nicht, auf die Stelle hinzuweisen, an der Sie die Induktionsannahme verwenden.
 5. Stellen Sie fest: "Wegen des Prinzips der vollständigen Induktion ist die Aussage richtig." Auf diese Weise weiß der Leser, dass der Beweis beendet ist.

Gegenbeispiel

- Um eine Aussage zu widerlegen, reicht ein Beispiel, für das die Aussage nicht gilt
- Achtung: es ist die einzige Situation, in der ein Beispiel ausreicht!
- Mehr zu Gegenbeispielen folgt

A.6. Beispiele und Gegenbeispiele

- Schulmathematik wird oft auf die folgende Art gelehrt:
 - So funktioniert die Produktregel
 - hier ist ein Beispiel
 - Bei den Übungsaufgaben können Sie dann analog vorgehen
- In der Universitätsmathematik ist da anders
- Hier muss man nachdenken und das Gelernte in Situationen anwenden, die Sie vorher noch nicht gesehen haben
- Rechenbeispiele haben ihren Sinn, doch helfen sie nicht, etwas wirklich zu verstehen
- Diese Fähigkeit erhält man, indem man Beispiele erschafft! Das ist deutlich anspruchsvoller
- Das Entwerfen eigener Beispiele ist eine hervorragende Methode, Mathematik zu lernen
- Folgenden weiteren Nutzen hat dies:
 - Aussagen testen: also zu überprüfen, ob unsere Schlussfolgerungen korrekt sind
 - Definitionen und Sätze besser zu merken
 - Sachverhalte zu klären
 - Verständnis über die Struktur des Problems
- Auch Nichtbeispiele sind nützliche: solche Beispiele verdeutlichen genau, was in der Definition oder dem Satz notwendig ist
- Ein Gegenbeispiel ist ein Beispiel, das zeigt, dass eine Aussage falsch ist. Suchen Sie diese.
 - Mein Tipp: Sammeln Sie gute Beispiele
 - Um gute Beispiele zu erschaffen hilft es, einen Vorrat von ganz einfachen Konstruktionen zu haben (einfache Graphen, Bäume, ...)
 - Üben sie Beispiele und Gegenbeispiele zu finden. Umso schneller wird es Ihnen gelingen und umso stärker wird es Ihnen helfen
 - Um eine Aussage zu beweisen, versuchen Sie Gegenbeispiele zu konstruieren, warum die Aussage falsch ist

A.7. Wie man (mathematische) Probleme löst

- Mathematische Probleme zu lösen ist schwierig. Es gibt keine Formel, keinen Ansatz, keine Denkweise, die alle Problem gleichzeitig löst.
- Die beste Art das Lösen von Problem zu lernen ist es Probleme zu lösen und dieses immer wieder zu üben. Dafür sind die Übungsaufgaben da.
- Folgende Schritte können Sie versuchen

1. Versuchen Sie das Problem zu verstehen

- Überprüfen Sie alle Symbole und Wörter, die in der Aufgabe vorkommen: kennen Sie sie? Kennen sie die genauen Definitionen? Haben Sie die Definitionen wirklich verstanden? Haben Sie jede einzelne Voraussetzung der Definitionen verstanden und berücksichtigt?
- Wissen Sie alles über die Voraussetzungen und die Schlussfolgerung: schreiben Sie auf, was Sie über beides wissen.
- Arbeiten Sie vorwärts und rückwärts: Starten Sie mit den Voraussetzungen und gehen sie weiter. Was folgt? Welche Eigenschaften finden Sie? Nehmen Sie die Schlussfolgerung und gehen sie rückwärts. Welche Implikationen hat die Schlussfolgerung? Was muss gelten?
- Arbeiten Sie mit Anfangs- und Spezialfällen: Schauen Sie sich "einfache" Spezialfälle an (vollständiger Graph, keine Kantengewichte, ...). Versuchen Sie die einzelnen Eigenschaften zu beweisen; überlegen Sie an welcher Stelle dies nicht für allgemeine Settings funktioniert oder eben doch; bauen Sie sich ein einfaches Beispiel, an dem Sie die Aussage überprüfen
- Zeichnen Sie ein Bild: Versuchen Sie sich alles zu visualisieren. Gerade in der Graphentheorie und kombinatorischen Optimierung ist das unheimlich hilfreich. Vergessen Sie aber nicht, den Beweis allgemein unabhängig von ihrem Beispiel zu zeigen.
- Finden Sie ähnlich Probleme: Wurde eine ähnliche Aussage schon bewiesen? Vielleicht funktioniert dann auch hier ein ähnlicher Lösungswegs.
- Wechseln Sie die Schreibweise: Wie kann das Problem mit Symbolen ausgedrückt werden? Wie kann man es mit Worten Beschreiben? Oft hilft es zwischen den beiden Arten hin- und her zu wechseln

2. Entwickeln Sie einen Plan

- Zerlegen Sie das Problem in viele kleine einzelne Teile
- Geben Sie Objekten/Eigenschaften Namen: das hilft einen Überblick zu behalten und nicht in viel zu langem Text das Wesentliche zu übersehen. Achtung: Wenn Sie an Definitionen/Bezeichnungen etwas ändern überprüfen Sie die Auswirkungen auf den folgenden Text
- Nutzen Sie Standardmethoden: überlegen Sie ob es besser ist einen direkten Beweis, einen Widerspruchsbeweis, Fallunterscheidungen oder eine vollständige Induktion anzuwenden. Versuchen Sie systematisch die unterschiedlichen Ansätze

3. Führen Sie den Plan aus

- Prüfen Sie jeden Schritt. Schauen Sie, dass jeder Schritt korrekt ist.
- Überprüfen Sie, wie Sie alles aufgeschrieben haben (siehe oben): kann es verbessert werden? Kann der Text gekürzt oder mit Beispielen besser verständlich gestaltet werden?
- Überprüfen Sie noch mal: könnte es sein, dass Ihre Aussage falsch ist?

4. Reflexion des Lösungsprozesses

- Was hat zum Erfolgreichen lösen geführt? Welche Strategie sind Sie gefahren?
Was können Sie fürs nächste Mal lernen?
- In welcher Weise ähnelt diese Lösung einer anderen?
- Nutzen Sie außerdem immer die Möglichkeit mit anderen über Ihre Ideen zu diskutieren. Oft hilft es ein Problem laut zu erklären, um es selber besser zu verstehen. Gleichzeitig schult ihr Gegenüber gute Fragen zu stellen und genau zuzuhören. Nutzen Sie dieses intensiv. Das Studium ist Teamarbeit!!!

A.8. Mathematik schreiben

- Leider reicht das finden einer Lösung nicht aus in der Mathematik
- Genauso wichtig ist es, diese Lösung sauber und ordentlich zu formulieren
- Gewöhnen Sie sich möglichst früh an ihre Gedankengänge mathematisch korrekt aufzuschreiben
 - damit verringern Sie die Fehlerquote an falschen Resultaten
 - anderen fällt es leichter ihre Resultate zu verstehen und darauf aufzubauen/mit Ihnen zu diskutieren
- Schreiben Sie ausführlich, nehmen Sie alle mit, erklären Sie alles sorgfältig und genau
 1. Wenn ich meine Intelligenz einsetzen muss, um zu verstehen, was gemeint ist, dann bekommen Sie die Punkte für meine Intelligenz (das macht bei Übungsaufgaben wenig Sinn)
 2. Wenn ich als Reviewer*in mich durch verwirrende Symbole, oder schlecht formulierte Ideen quälen muss, dann bin ich schnell frustriert und tendiere dazu a) das Paper zu abzulehnen bzw. b) Punkte dafür abzuziehen
- Zweck des Schreibens ist die Kommunikation mit dem Ziel seine Gedanken anderen zu vermitteln. Lesende sind keine Hellseher*innen. Deshalb ist es wichtig zu erklären, was Sie als nächstes tun, was Sie gerade tun und, zusammenfassend, was sie getan haben. Zusätzlich sollte man Dinge ab und zu wiederholen.
- Die Verantwortung für eine gut formulierte Abgabe liegt bei Ihnen. Wenn Sie etwas formuliert haben, was jemand mit demselben Wissensstand nicht gut versteht, dann haben Sie das schlecht erklärt und aufgeschrieben.
- Mathematische Texte sind Texte. Sie unterliegen den normalen Rechtschreib-, Grammatik- und Satzzeichenregeln wie jeder Text. Schreiben Sie Text entsprechend in ganzen Sätzen, nutzen Sie eine Rechtschreibprüfung und setzen Sie Satzzeichen.
Nicht: $x > 0$ $y < 0$ und $x + y < 1$
Sondern: Sei $x > 0$, $y < 0$ und es gelte, dass $x + y = 1$
- Symbole, Mengen, Bezeichnungen alleine ergeben noch keinen Mathematischen Text. Auch diese müssen in Sätze richtig integriert werden.
- Mathematik wird sehr ökonomisch aufgeschrieben. Benutzen Sie deshalb auch selbst kurze Wörter und Sätze. Kurze Sätze sind leicht zu lesen. Vermeiden Sie, um Mehrdeutigkeiten auszuschließen, komplizierte Konstruktionen mit vielen Negationen.
- Manchmal ist es deutlicher, Symbole zu verwenden, manchmal nicht. Oft hilft es beides zu verwenden. Achtung: das Verwenden von vielen Symbolen macht einen Beweis noch lange nicht richtig. Seien Sie aber sicher, dass Sie alle Symbole im Text einführen und definieren, bevor Sie sie verwenden.
- Wenn Sie später im Text auf ein gewisses Ergebnis oder eine Formel verweisen, geben Sie dieser eine Nummer oder einen Namen, so dass Sie darauf verweisen können.

- Beginnen Sie einen Satz nicht mit einem Symbol.
 Nicht: f, g seien Funktionen, die differenzierbar und nicht negativ sind
 Sondern: Seien f und g zwei differenzierbare und nicht-negative Funktionen.
- Verwenden Sie übliche Symbole und Bezeichnungen. Eine andere Nutzung führt zu Verwirrungen (Achtung: „übliche Symbole“ hängen auch vom jeweiligen Teilgebiet der Mathematik ab)
 Nicht: Sei $n > 0$ und $\varepsilon \in \mathbb{N}$
 Sondern: Sei $\varepsilon > 0$ und $n \in \mathbb{N}$
- Trotz einiger Konventionen in mathematischen Texten, sollten Sie Ihre Bezeichnungen so definieren, dass sie völlig eindeutig sind.
 Nicht: Die Laufzeit des Algorithmus ist $\mathcal{O}(n \cdot m)$.
 Sondern: Wir bezeichnen mit n die Anzahl der Kanten und m die Anzahl der Kanten im Graph. Dann ist die Laufzeit des Algorithmus $\mathcal{O}(n \cdot m)$.
- Wiederholung von Worten ist schlechter Stiel und ermüdend für den Lesenden. Dies gilt allerdings nicht für Fachbegriffe. Es kann verwirren, wenn von „Rolling Stock“ und „Trains“ im Verkehrswesen gesprochen wird und nicht klar ist, ob dies jetzt dasselbe sein soll oder nicht.
- Wenn Sie einen Text geschrieben haben:
 1. Lassen Sie ihn mindestens einen Tag liegen
 2. Lesen Sie ihn noch mal kritisch durch und versuchen Sie Fehler zu finden (noch besser: fragen Sie jemand anderen)
 3. Überprüfen Sie: Macht die Lösung einen Sinn? Ist sie leicht zu lesen? Und ist sie richtig?
 4. Suchen Sie nach Tippfehlern, grammatischen Fehlern oder (noch mal) mathematischen Fehlern
 5. Überprüfen Sie: können Teile des Textes/Worte weggelassen werden? Habe ich ganze Sätze geschrieben? Wurden alle Symbole eingeführt?