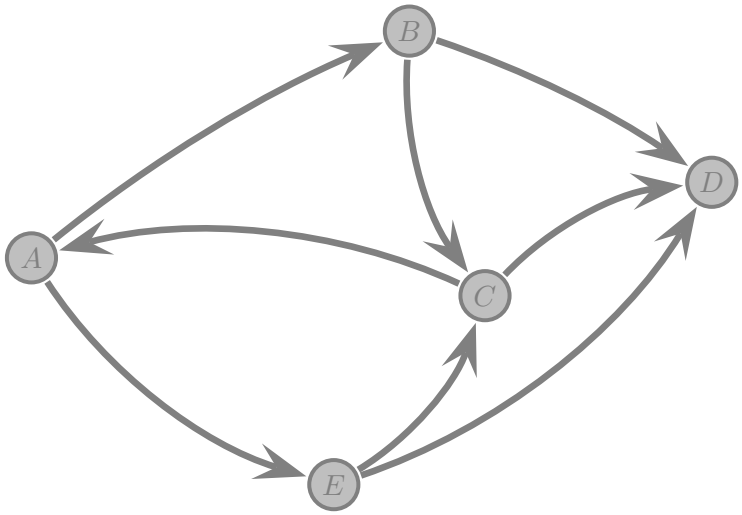
Discrete Optimization

Spring 2015

Thomas Rothvoss



UNIVERSITY of
WASHINGTON

Last changes: May 27, 2015

Contents

Chapter 1

Introduction to Discrete Optimization

Roughly speaking, **discrete optimization** deals with finding the best solution out of finite number of possibilities in a computationally efficient way. Typically the number of possible solutions is larger than the number of atoms in the universe, hence instead of mindlessly trying out all of them, we have to come up with insights into the **problem structure** in order to succeed. In this class, we plan to study the following classical and basic problems:

- Minimum spanning trees
- The Shortest Path problem
- Maximum flows
- Matchings
- The Knapsack problem
- Min Cost flows
- Integer Programming

The purpose of this class is to give a proof-based, formal introduction into the theory of discrete optimization.

1.1 Algorithms and Complexity

In this section, we want to discuss, what we formally mean with **problems**, **algorithms** and **running time**. This is made best with a simple example. Consider the following problem:

FIND DUPLICATE
Input: A list of numbers $a_1, \ldots, a_n \in \mathbb{Z}$
Goal: Decide whether some number appears at least twice in the list.

Obviously this is not a very interesting problem, but it will serve us well as introductory example to bring us all one the same page. A straightforward algorithm to solve the problem is as follows:

(1) FOR $i = 1$ TO n DO
    (2) FOR $j = i + 1$ TO n DO
        (3) If $a_i = a_j$ then return "yes"
(4) Return "no"

The algorithm is stated in what is called **pseudo code**, that means it is not actually in one of the common programming languages like Java, C, C++, Pascal or BASIC. On the other hand, it takes only small modifications to translate the algorithm in one of those languages. There are no consistent rules what is allowed in pseudo code and what not; the point of pseudo code is that it does not need to be machine-readable, but it should be human-readable. A good rule of thumb is that everything is allowed that also one of the mentioned programming languages can do.

In particular, we allow any of the following operations: addition, subtraction, multiplication, division, comparisons, etc. Moreover, we allow the algorithm an infinite amount of memory (though our little algorithm above only needed the two variables i and j).

The next question that we should discuss in the analysis of the algorithm is its **running time**, which we define as the **number of elementary operations** (such as adding, subtracting, comparing, etc) that the algorithm makes. Since the variable i runs from 1 to n and j runs from $j = i + 1$ to n, step (3) is executed $\binom{n}{2} = \frac{n(n-1)}{2}$ many times. On the other hand, should we count only step (3) or shall we also count the FOR loops? And in the 2nd FOR loop, shall we only count one operation for the comparison or shall we also count the addition in $i + 1$? We see that it might be very tedious to determine the exact number of operations. On the other hand we probably agree that the running time is of the form Cn^2 where C is some constant that might be, say 3 or 4 or 8 depending on what exactly we count as an elementary operation. Let us agree from now on, that we only want to determine the running time up to constant factors.

As a side remark, there is a precisely defined formal computational model, which is called a **Turing machine** (interestingly, it was defined by Alan M. Turing in 1936 before the first computer was actually build). In particular, for any algorithm in the Turing machine model one can reduce the running time by a constant factor while increasing the state space. An implication of this fact is that running times in the Turing machine model are actually only well defined up to constant factors. We take this as one more reason to be content with our decision of only determining running times up to constant factors.

So, the outcome of our runtime analysis for the FIND DUPLICATE algorithm is the following:

> There is some constant $C > 0$ so that the FIND DUPLICATE algorithm finishes after at most Cn^2 many operations. (1.1)

Observe that it was actually possible that the algorithm finishes much faster, namely if it finds a match in step (3), so we are only interested in an upper bound. Note that the simple algorithm that we found is not the most efficient one for deciding whether n numbers contain a duplicate. It is actually possible to answer that question in time $C'n\log(n)$ using a sorting algorithm. If we want to compare the running times Cn^2 and $C'n\log(n)$, then we do not know which of the constants C and C' is larger. So for small values of n, we don't know which algorithm would be faster. But $\lim_{n\to\infty} \frac{Cn^2}{C'n\log(n)} = \infty$, hence if n is large enough the $C'n\log(n)$ algorithm would outperform the Cn^2 algorithm. Thus, we do consider the $C'n\log(n)$ algorithm as the more efficient one[1].

It is standard in computer science and operations research to abbreviate the claim from (1.1) using the ***O*-notation** and replace it by the equivalent statement:

> The FIND DUPLICATE algorithm takes time $O(n^2)$. (1.2)

[1]Most constants that appear in algorithms are reasonable small anyway. However, there are fairly complicated algorithm for example for matrix multiplication which theoretically are faster than the naive algorithms, but only if n exceeds the number of atoms in the universe.

The formal definition of the O-notation is a little bit technical:

Definition 1. If $f(s)$ and $g(s)$ are two positive real valued functions on $\mathbb{N}$, the set of non-negative integers, we say that $f(n) = O(g(n))$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all n greater than some finite n_0.

It might suffice to just note that the statements (1.1) and (1.2) are equivalent and we would use the latter for the sake of convenience.

Going once more back to the FIND DUPLICATE algorithm, recall that the **input** are the numbers $a_1, \ldots, a_n$. We say that the **input length** is n, which is the number of numbers in the input. For the performance of an algorithm, we always compare the running time with respect to the input length. In particular, the running time of $O(n^2)$ is bounded by a polynomial in the input length n, so we say that our algorithm has **polynomial running time**. Such polynomial time algorithms are considered **efficient** from the theoretical perspective[2]. Formally, we say that any running time of the form n^C is polynomial, where $C > 0$ is a constant and n is the length of the input. For example, below we list a couple of possible running times and sort them according to their asymptotic behavior:

$$\underbrace{100n \ll n\ln(n) \ll n^2 \ll n^{10}}_{\text{efficient}} \ll \underbrace{2^{\sqrt{n}} \ll 2^n \ll n!}_{\text{inefficient}}$$

1.1.1 Complexity theory and NP-hardness

We want to conclude with a brief and informal discussion on **complexity theory**. The **complexity class P** is the class of problems that admit a **p**olynomial time algorithm. For example, our problem of deciding whether a list of numbers has duplicates is in **P**. However, not all problems seem to admit polynomial time algorithms. For example, there is no polynomial time algorithm known for the following classical problem:

PARTITION
Input: A list of numbers $a_1, \ldots, a_n \in \mathbb{N}$
Goal: Decide whether one can partition $\{1, \ldots, n\}$ into $I_1 \dot{\cup} I_2 = \{1, \ldots, n\}$ so that

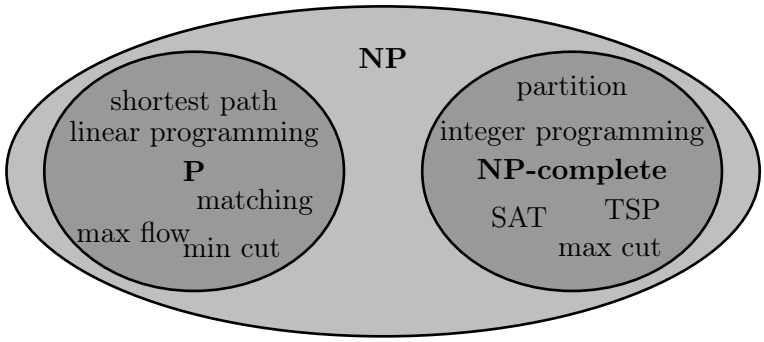
$$\sum_{i \in I_1} a_i = \sum_{i \in I_2} a_i.$$

To capture problems of this type, one defines a more general class: **NP** is the class of problems that admit a **n**on-deterministic **p**olynomial time algorithm. Intuitively, it means that a problem lies in **NP** if given a solution one is able to verify in polynomial time that this is indeed a solution. For example for PARTITION, if somebody claims to us that for a given input $a_1, \ldots, a_n$ the answer is "yes", then (s)he could simply give us the sets I_1, I_2. We could then check that they are indeed a partition of $\{1, \ldots, n\}$ and compute the sums $\sum_{i \in I_1} a_i$ and $\sum_{i \in I_2} a_i$ and compare them. In other words, the partition I_1, I_2 is a **computational proof** that the answer is "yes" and the proof can be verified in polynomial time. That is exactly what problems in **NP** defines. Note that trivially, $\mathbf{P} \subseteq \mathbf{NP}$.

We say that a problem $P \in \mathbf{NP}$ is **NP-complete** if with a polynomial time algorithm for P, one could solve any other problem in **NP** in polynomial time. Intuitively, the **NP**-complete problems

[2]This is true for theoretical considerations. For many practical applications, researchers actually try to come up with near-linear time algorithms and consider anything of the order n^2 as highly impractical.

are the hardest problems in **NP**. One of the 7 Millenium problems (with a $1,000,000 award) is to prove the conjecture that **NP**-complete problems do not have polynomial time algorithms (i.e. **NP** $\neq$ **P**). An incomplete overview over the complexity landscape (assuming that indeed **NP** $\neq$ **P**) is as follows:



From time to time we want to make some advanced remarks that actually exceed the scope of this lecture. Those kind of remarks will be in gray box labeled **advanced remark**. Those comments are not relevant for the exam, but they give some background information for the interested student.

Advanced remark:

Now with the notation of **P** and **NP**, we want to go back to how we determine the running time. We used what is called the **arithmetic model / RAM model** where any arithmetic operation like addition, multiplication etc, counts only one unit. On the other hand, if we implement an algorithm using a Turing machine, then we need to encode all numbers using bits (or with a constant number of symbols, which has the same effect up to constant factors). If the numbers are large, we might need a lot of bits per number and we might dramatically underestimate the running time on a Turing machine if we only count the number of arithmetic operations. To be more concrete, consider the following (useless) algorithm

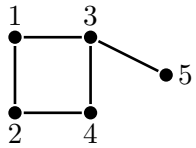
(1) Set $a := 2$
(2) FOR $i = 1$ TO n DO
    (3) Update $a := a^2$.

The algorithm only performs $O(n)$ arithmetic operations. On the other hand, the variable at the end is $a = 2^{2^n}$. In other words, we need 2^n bits to represent the result, which leaves an exponential gap between the number of operations in the arithmetic model and the **bit model** where we count each bit operation. It is even worse: One can solve **NP**-complete problems using a polynomial number of arithmetic operations by creating numbers with exponentially many bits and using them to do exponential work.
For a more formal accounting of running time, it is hence necessary to make sure that the gap between the arithmetic model and the bit model is bounded by a polynomial in the input length. For example it suffices to argue that all numbers are not more than single-exponentially large in the the input. All algorithms that we consider in this lecture notes will have that property, so we will not insist on doing that (sometimes tedious) part of the analysis.

1.2 Basic Graph Theory

Since most of the problems that we will study in this course are related to **graphs**, we want to review the basics of graph theory first. Recall the following graph theory definitions:

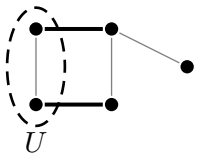
- An **undirected graph** $G = (V, E)$ is the pair of sets V and E where V is the set of **vertices** of G and E the set of (undirected) **edges** of G. An edge $e \in E$ is a set $\{i, j\}$ where $i, j \in V$. We write $G = (V(G), E(G))$ if there are multiple graphs being considered.



Graph $G = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$
and $E = \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{3, 4\}, \{3, 5\}\}$

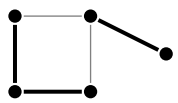
  While it is popular to draw the nodes of graphs as dots and the edges as lines or curves between those dots, one should keep in mind that the position of the nodes has actually no meaning.

- A subset $U \subseteq V$ induces a **cut** $\delta(U) = \{\{u, v\} \in E \mid u \in U \text{ and } v \notin U\}$



bold edges form the cut $\delta(U)$

- The **degree** of a node is the number of incident edges. We write $\deg(v) = |\delta(v)|$.
- If the edges of G are precisely the $\binom{|V|}{2}$ pairs $\{i, j\}$ for every pair $i, j \in V$ then G is the **complete graph** on V. The complete graph on $n = |V|$ vertices is denoted as K_n.
- A **subgraph** of $G = (V(G), E(G))$ is a graph $H = (V(H), E(H))$ where $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ with the restriction that if $\{i, j\} \in E(H)$ then $i, j \in V(H)$.
- If $V' \subseteq V(G)$, then the subgraph **induced** by V' is the graph $(V', E(V'))$ where $E(V')$ is the set of all edges in G for which both vertices are in V'.
- A subgraph H of G is a **spanning subgraph** of G if $V(H) = V(G)$.



spanning subgraph

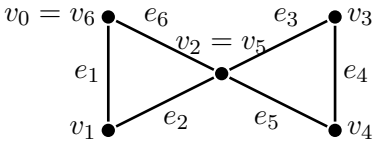
- Given $G = (V, E)$ we can define subgraphs obtained by **deletions** of vertices or edges.
  - If $E' \subseteq E$ then $G \backslash E' := (V, E \backslash E')$.

– If $V' \subseteq V$ then $G\backslash V' = (V\backslash V', E\backslash\{\{i,j\} \in E : i \in V' \text{ or } j \in V'\})$.

- A **path** in $G = (V, E)$ in a sequence of vertices and edges $v_0, e_1, v_1, e_2, v_2, e_3, \ldots, e_k, v_k$, such that for $i = 0, \ldots, k$, $v_i \in V$, $e_i \in E$ where $e_i = (v_{i-1}, v_i)$. This is called a (v_0, v_k)-**path**. The **length** of the path is the number of edges in the path which equals k. This path is **closed** if $v_0 = v_k$. We call a path **simple** if all the used edges are distinct (but vertices may be revisited). A simple path that is also closed is called a **cycle**.

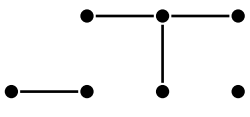
> **Advanced remark:**
>
> Looking at the literature one sees that the definition of a "path" is not standardized. For example the book "Graph Theory" of Diestel would call a path as we have it a **walk** and a path itself would be required to have distinct nodes and edges.

A closed path is a **circuit** if the vertices are distinct (except the first and the last).

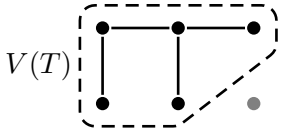


cycle of length 6 (not a circuit)

- A graph G is **connected** if there is a path in the graph between any two vertices of the graph.
- A graph G is **acyclic** if it contains no circuits as subgraphs.
- An acyclic graph is called a **forest**. A connected forest $T = (V(T), E(T))$ is a **tree**.

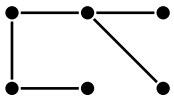


forest



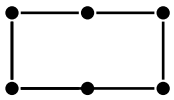
tree T

- $T \subseteq E$ is a **spanning tree** if T is spanning, connected and acyclic.



spanning tree

- A **Hamiltonian circuit/tour** is a circuit visiting every node exactly once



Hamiltonian circuit/tour

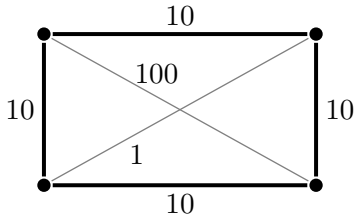
- A set $M \subseteq E$ of edges with degree ≤ 1 for each vertex is called **matching**.
- A set $M \subseteq E$ of edges with degree exactly 1 for each vertex is called **perfect matching**.

matching perfect matching

1.3 The traveling salesperson problem

We want to come to a more complex example of an algorithm; this time for one of the most prominent problems in combinatorial optimization:

TRAVELING SALESPERSON PROBLEM (TSP)
Input: We are given a complete graph $K_n = (V, E)$ with edge cost $c_{ij} \in \mathbb{R}_{\geq 0}$ for each edge $\{i, j\} \in E$.
Goal: Find the Hamiltonian circuit $C \subseteq E$ that minimizes the total cost $\sum_{e \in C} c_e$.



TSP instance with opt. tour in bold

Note that this problem again has an **input**, which defines the concrete **instance** (here, the concrete set of numbers c_{ij}). Then the problem has an implicitly defined set of solutions (here, all tours in an n-node graph). Usually, the problem also has an **objective function** that should be optimized (here, minimize the sum of the edge cost in the tour).

The first trivial idea would be to simply enumerate all possible tours in the complete graph; then for each tour we compute the cost and always remember the cheapest one that we found so far, which we output at the end. If we ignore the overhead needed to list all tours, certainly the time is dominated by the **number of tours**. Unfortunately, the number of tours in an n-node graph is

$$(n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1 = (n-1)!$$

If we remember Stirlings formula $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$, then $n! \approx 2^{n \ln(n) - n}$. For example if $n = 50$, then $(n-1)! \approx 10^{62}$ and if our computer can enumerate 10^9 tours per second, it would take $\approx 10^{43}$ centuries to enumerate all of them. Obviously, this approach is not very practicable. But in fact, we can solve the problem in a slightly smarter way (though not that much smarter).

The trick in the **Held-Karp algorithm** is to first compute the best **subtours** using **dynamic programming**. For each subset $S \subseteq V$ and points $i, j \in S$ we want to compute **table entries**

$$C(S, i, j) \quad := \quad \text{length of the shortest path from } i \text{ to } j \text{ visiting each node in } S \text{ exactly once (and no node outside of } S)$$

Now we simply compute all entries $C(S, i, j)$ with increasing size of S:

Held-Karp algorithm

Input: Distances $c_{ij} \geq 0$ in a complete n-node graph

Output: Cost of an optimum TSP tour

(1) $C(\{i, j\}, i, j) := c_{ij}$ for all $i \neq j$

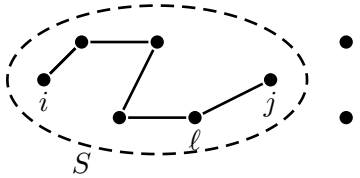
(2) FOR $k = 3$ TO n DO

(3) FOR ALL SETS $S \subseteq V : |S| = k$ DO

$$C(S, i, j) := \min_{\ell \in S \setminus \{i,j\}} \{C(S \setminus \{j\}, i, \ell) + c_{\ell,j}\}$$

(4) output the optimum cost $\min_{j \neq 1}\{C(V, 1, j) + c_{j1}\}$

We can visualize the computation in step (3) as



Let us make the following observation:

Lemma 1. *The algorithm computes indeed the cost of the optimum tour.*

Proof. Prove by induction on $|S|$ that the quantity $C(S, i, j)$ computed in the algorithm really satisfies the definition. □

Now we should discuss how we analyse the **running time** for the algorithm. For example for the considered TSP algorithm, we have at most $n^2 2^n$ many entries of the form $C(S, \{i, j\})$; for each entry, the $\min_\ell\{\ldots\}$ expression needs to be evaluate for at most n many choices of ℓ and evaluating each term $C(S\setminus\{j\}, i, \ell) + c_{\ell,j}$ takes $O(1)$ many elementary operations. The overall running time of the algorithm is hence $O(n^3 2^n)$. We might wonder, whether one could come up with a much more efficient algorithm for TSP than the one that we have discussed. But in fact, it is unknown whether there is any algorithm that solves TSP in time $O(1.9999^n)$. For the input length we count the number of numbers/objects, i.e. in this case $\binom{n}{2} \sim n^2$.

Apart from the obvious application to schedule a traveling salesperson/postman tour, the vertices $v_1, \ldots, v_n$ could be holes that need to be drilled on a circuit board and c_{ij} could be the time taken by the drill to travel from v_i to v_j. In this case, we are looking for the fastest way to drill all holes on the circuit board if the drill traces a tour through the locations that need to be drilled. Yet another application might be that $v_1, \ldots, v_n$ are n celestial objects that need to be imaged by a satellite and c_{ij} is the amount of fuel needed to change the position of the satellite from v_i to v_j. In this case, the optimal tour allows the satellite to complete its job with the least amount of fuel. These and many other applications of the TSP as well as the state-of-the-art on this problem can be found on the web page `http://www.tsp.gatech.edu/`. In many cases, the distances c_{ij} have special properties like, they define a metric or are Euclidean distance. In those cases one can design more efficient algorithms.

Chapter 2

Minimum spanning trees

In this chapter we study the problem of finding a minimum cost spanning tree in a graph. In the formal definition, we considered a tree T as being a subgraph, but it is usually easier just to think of T as being the set of edges that form the subgraph.

> MINIMUM SPANNING TREE
> **Input:** Undirected graph $G = (V, E)$, cost $c_{ij} \geq 0$ $\forall \{i, j\} \in E$
> **Goal:** A spanning tree $T \subseteq E$ minimizing $c(T) := \sum_{e \in T} c_e$.

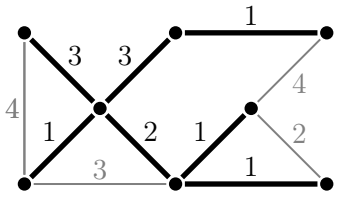
Then recall that in a graph $G = (V, E)$ we call the edge set $T \subseteq E$ a **spanning tree** if:

(i) T is acyclic, (ii) T is connected, (iii) T is spanning (= incident to each node)

A very useful equivalent definition is:

An edge set T is a spanning tree if and only if for each pair $u, v \in V$ of nodes there is exactly one simple path from u to v.

This is clear because if you have more than 2 simple paths between two nodes u, v, then this implies a cycle; on the other hand being connected and spanning means you have at least one path between each pair of vertices. For example, in the graph below, we marked the minimum spanning tree with bold edges (edges $\{i, j\}$ are labelled with their cost c_{ij}):



We start by establishing some basic properties of a spanning tree. Some of those might appear obvious, but let us be formal here.

Lemma 2. *For $G = (V, E)$ connected and spanning with $|V| = n$ nodes. Then one has:*

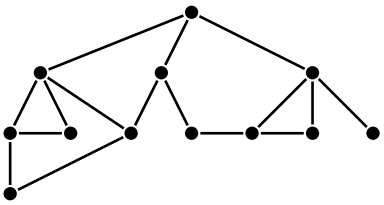
a) $|E| \geq n - 1$

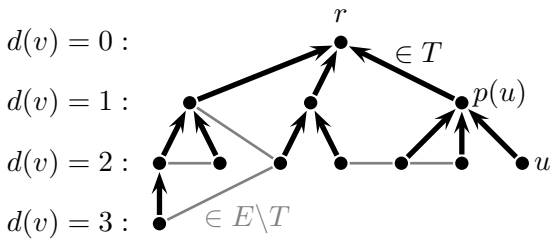
b) $\exists$ *spanning tree* $T \subseteq E$

c) G acyclic $\Leftrightarrow |E| = n - 1$

d) G is spanning tree $\Leftrightarrow |E| = n - 1$

Proof. Luckily, in all 4 cases we can use almost the same proof. We begin with finding some spanning tree in G. To do that, we fix an arbitrary node $r \in V$ and call it the **root**. We give each node v a label $d(v)$ which tells the distance (in terms of number of edges) that this node has from the root. Since G is connected, there is a path from v to r and $d(v)$ is well-defined. For node $v \in V \backslash r$, we select an arbitrary neighbor node $p(v)$ that is closer to r, that means $\{u, p(u)\} \in E$ and $d(p(u)) = d(v) - 1$ (here the p stands for **p**arent node). We now create a set $T := \{\{u, p(u)\} \mid u \in V \backslash \{r\}\}$ of all such edges. In other words, each node u has exactly one edge $\{u, p(u)\} \in T$ that goes to a node with a lower d-label.





We make a couple of observations:

i) *T is connected and spanning*: This is because from each node u you can find a path to the root by repeatedly taking the edge to its parent.

ii) $|E| \geq |T| = n - 1$: That is because we have a bijective pairing between nodes $u \in V \backslash \{r\}$ and edges $\{u, p(u)\}$, thus $|T| = |V \backslash \{r\}| = n - 1$.

iii) *T is acyclic*: If not, we can delete an edge in the cycle and obtain a still connected spanning subgraph $T' \subseteq T$ with only $|T'| = n-2$ edges; but if we apply ii) we obtain that each connected spanning graph on n nodes must have $\geq n - 1$ edges; that gives a contradiction.

iv) *T is a spanning tree in G*. Follows from observations i)+iii).

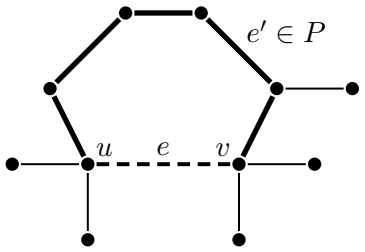
v) *For any edge* $\{u, v\} \in E \backslash T$, $T \cup \{u, v\}$ *contains a cycle*: This is because there was already a path from u to v in T. That means if $|E| > n - 1$, then G is not acyclic, if $|E| = n - 1$ then $E = T$ and G is acyclic by iii).

vi) *d)* follows from *c)*: That is since G is connected and spanning by assumption.

□

A very useful operation to modify a spanning trees is the following:

Lemma 3. *Let $T \subseteq E$ be a spanning tree in $G = (V, E)$. Take an edge $e = \{u, v\} \in E \backslash T$ and let $P \subseteq T$ be the edges on the simple path from u to v. Then for any edge $e' \in P$, $T' = (T \backslash \{e'\}) \cup \{e\}$ is a spanning tree.*

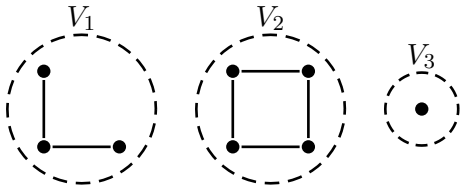


Proof. Swapping edges in a cycle $P \cup \{e\}$ leaves T connected and the number of edges is still $n-1$, hence T' is still a spanning tree. □

The operation to replace e' by e is called an **edge swap**. It leads to a spanning tree of cost $c(T') = c(T) - c(e') + c(e)$. If T was already the minimum spanning tree, then we know that $c(e) \geq c(e')$ or in other words: if T is optimal, then any edge $\{u, v\} \in E \backslash T$ is at least as expensive as all the edges on the path $P \subseteq T$ that connects u and v (because otherwise we could swap that edge with $\{u, v\}$ and we would obtain a cheaper spanning tree).

Surprisingly it turns out that the other direction is true as well: if no edge swap would decrease the cost, then T is optimal.

Before we show that, we need one more definition that is very useful in graph theory. For any graph $G = (V, E)$ we can partition the nodes into $V = V_1 \dot\cup \ldots \dot\cup V_k$ so that a pair u, v is in the same set V_i if and only they are connected by a path. Those sets $V_1, \ldots, V_k$ are called **connected components**. Formally, one could define relation $\sim$ with $u \sim v$ if and only if there is a path from u to v in G. Then one can easily check that $\sim$ is an equivalence relation and the equivalence classes define the connected components, see the following example:



connected components

Also recall that for a node set $U \subseteq V$ and $\delta(U) = \{\{u, v\} \in E \mid |U \cap \{u, v\}| = 1\}$ denotes the edges that are cut by U.

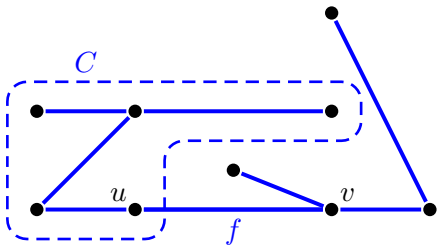
Theorem 4. *Let T be a spanning tree in graph G with edge costs c_e for all $e \in E$. Then the following statements are equivalent:*

1. *T is a minimum spanning tree (MST).*

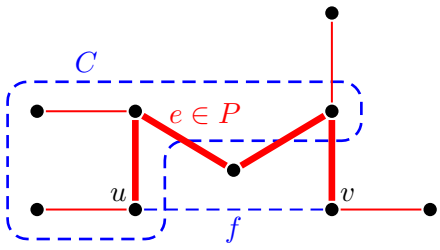
2. *No edge swap for T decreases the cost.*

Proof. It only remains to show (2) ⇒ (1). In other words, we fix a non-optimal spanning tree T; then we will find an improving edge swap. Let T^* be an optimum minimum spanning tree. If this choice is not unique, then we choose a T^* that maximizes $|T \cap T^*|$; in other words, we pick on optimum solution T^* that has as many edges as possible in common with T.

Take any edge $f = \{u, v\} \in T \backslash T^*$. The subgraph $T \backslash \{f\}$ consists of two connected components, let us call them $C \subseteq V$ and $V \backslash C$, say with $u \in C$ and $v \in V \backslash C$.



tree T



tree T^*

On the other hand, there is a path $P \subseteq T^*$ that connects u and v. As this path "starts" in C and "ends" in $V \backslash C$, there must be be an edge in the path that lies in the cut $\delta(C) \cap T^*$. We call this edge $e \in \delta(C) \cap T^* \cap P$. Let us make 2 observations:

- $T' = (T \backslash \{f\}) \cup \{e\}$ is a spanning tree! This is because e connects the two connected components of $T \backslash \{f\}$, hence it is connected and T' also has $n-1$ edges, thus T' is a spanning tree.
- $T^{**} = (T^* \backslash \{e\}) \cup \{f\}$ is a spanning tree as well.

Now it depends which of the edges e and f was the cheaper one:

- Case $c(f) > c(e)$: Then $c(T') < c(T)$ and we found an edge swap that improved T.
- Case $c(f) < c(e)$: Then $c(T^{**}) < c(T^*)$ which contradicts the optimality of T^*.
- Case $c(f) = c(e)$: Then T^{**} is also optimal. But T^{**} has one edge more in common with T than T^* has. This contradicts the choice of T^* and the claim follows.

□

We want to emphasize at this point that spanning trees have a very special structure which implies that a "local optimal solution" (that we cannot improve by edge swaps) is also a "global optimal solution". Most other problems like TSP do not share this property.

2.1 Kruskal's algorithm

In this section, we want to discuss an algorithm for finding the Minimum Spanning Tree, which is called **Kruskal's Algorithm**. Interestingly, it suffices to consider each edge only once and one can make an immediate decision whether or not to include that edge.

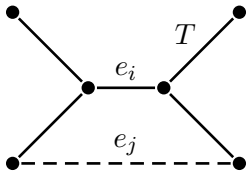
Kruskal's Algorithm

Input: A connected graph G with edge costs $c_e \in \mathbb{R}$, $\forall\ e \in E(G)$.
Output: A MST T of G.
(1) Sort the edges such that $c_{e_1} \leq c_{e_2} \leq \ldots \leq c_{e_m}$.
(2) Set $T = \emptyset$
(3) For i from 1 to m do
If $T \cup \{e_i\}$ is acyclic then update $T := T \cup \{e_i\}$.

Theorem 5. *The graph T found by Kruskal's algorithm is an MST of G.*

Proof. Let T be the output of Kruskal's algorithm. We need to first show that T is a spanning tree of G. Obviously, T will be acyclic at the end. Now suppose for the sake of contradiction that T was not connected and that $C \subseteq V$ is one connected component with $1 \leq |C| < n$. We assumed that the graph G itself was spanning and connected hence G did contain some edge $e = \{u, v\} \in E$ that was running between C and $V \backslash C$. But the algorithm considered that edge and some point and decided not to take it. That means there was already a path in T between u and v, which gives a contradiction.

We now argue that T cannot be improved by edge swaps, which by Theorem 4 proves that T is the cheapest spanning tree. Consider any a pair of edges $e_i \in T$ and $e_j \in E \backslash T$ so that $(T \backslash \{e_i\}) \cup \{e_j\}$ is again a spanning tree.



In particular, $(T \backslash \{e_i\}) \cup \{e_j\}$ is acyclic, but then at the time that e_i was added, it would have been possible to add instead e_j without creating a cycle. That did not happen, hence e_j was considered later. This means $i < j$ and $c(e_i) \leq c(e_j)$, which implies that an edge swap would not have paid off. □

It should be clear that the algorithm has polynomial running time (in the arithmetic and in the bit model). In fact, it is very efficient:

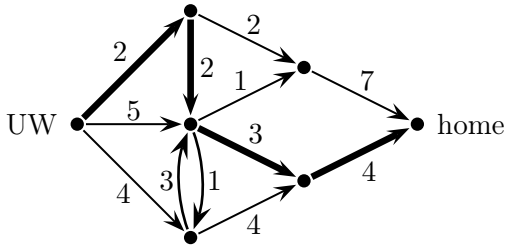
Theorem 6. *Kruskal's algorithm has a running time of $O(m \log(n))$ (in the arithmetic model) where $n = |V|$ and $m = |E|$.*

Here we need a running time of $O(m \log m)$ to sort the edges according to their size. Note that since $\log m \leq \log \binom{n}{2} \leq 2 \log n$, this quantity is of the form $O(m \log n)$ (just with a higher constant hidden in the O-notation). Moreover, the algorithm needs to repeatedly check whether two nodes $u, v \in V$ are already in the same connected component. Using a **union-find data structure**, this can be done in time $O(\log n)$ per test. We refer to the corresponding computer science lecture for more details.

Chapter 3

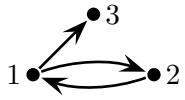
Shortest paths

To motivate the next problem that we want to study in detail, imagine that you would like to drive home from UW and suppose you want to select a route so that you spend as little as possible on gas. Suppose you already isolated a couple of road segments that you consider of using on your way home and from your experience, you know how much gas your car needs on each segment. We draw a graph with a vertex for each location/crossing and an edge for each segment that we label with the amount of gas that is needed for it.



Now we are interested in a path in this graph that goes from UW to your home and that minimizes the amount of gas. Observe that some roads are one way streets and also the amount of gas could depend on the direction that you drive, e.g. because you need more gas to drive uphill than downhill and because the traffic in one direction might be heavier than in the other one (at a given time of the day). In other words, we allow the edges have **directions**, but so far we can assume that $c_e \geq 0$.

Definition 2. A **directed graph (digraph)** $G = (V, E)$ consists of a vertex set V and directed edges $e = (i, j) \in E$ (also called **arcs**) such that i is the **tail** of e and j is the **head** of e. A **directed path** is of the path $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ with $(v_i, v_{i+1}) \in E$ for all i. A path is called an s-t **path** if it starts in s and ends in t ($s, t \in V$).

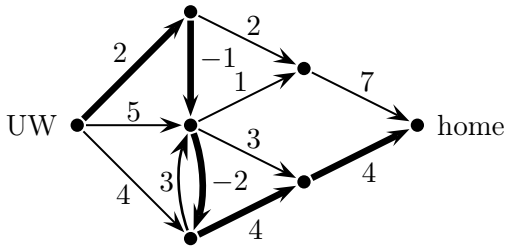


Example: directed graph $G = (V, E)$ with
$V = \{1, 2, 3\}$ and $E = \{(1, 2), (2, 1), (1, 3)\}$

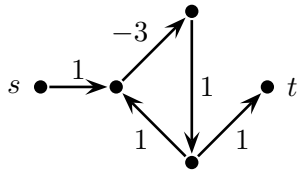
SHORTEST PATH PROBLEM
Input: Directed graph $G = (V, E)$, cost c_{ij} $\forall (i, j) \in E$, nodes $s, t \in V$
Goal: s-t path $P \subseteq E$ minimizing $c(T) := \sum_{e \in P} c_e$.

In the following, whenever we have a cost function $c : E \to \mathbb{R}$, then "shortest path" means the one minimizing the cost (not necessarily the one minimizing the number of edges). In the example above, we depicted the shortest "UW-home path".

Now let us go back to our introductory example where we want to find the most gas-efficient way from UW to your home. Now imagine you just bought a hybrid car, which can charge the battery when breaking. In other words, when you drive downhill, the gas-equivalent that you spend maybe **negative** (while before we assumed $c(e) \geq 0$). Maybe the road network with gas consumption now looks as follows (again with the shortest path in bold):



So, we might need to drop the assumption that edge cost are non-negative (i.e. $c(e) \geq 0$). But could we allow arbitrary (possibly negative) edge cost? Well, consider the following example instance:



What is the length of the shortest s-t path? Observe that the length of the cycle is -1. Hence if we allow to revisit nodes/edges, then the length of the shortest path is $-\infty$ and hence not well defined. If we restrict the problem to simple paths (= paths that do not revisit nodes), then one can prove that the problem becomes **NP**-complete[1], which we want to avoid. It seems the following two assumptions are meaningful, where the 1st one implies the 2nd one.

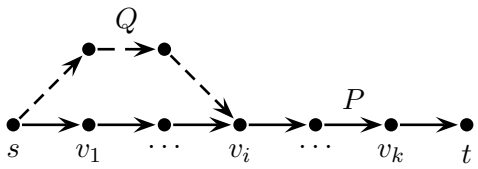
- **Strong assumption:** $c_e \geq 0 \; \forall e \in E$
- **Weak assumption:** All cycles have cost ≥ 0.

In this chapter, we will see several algorithms; some need the stronger assumption (but are more efficient) and others have higher running time, but work with the weaker assumption.

Lemma 7 (Bellman's principle)**.** *Let $G = (V, E)$ be a directed, weigthed graph with no negative cost cycle. Let $P = \{(s, v_1), (v_1, v_2), \ldots, (v_k, t)\}$ be the shortest s-t path. Then $P_{s,i} = \{(s, v_1), (v_1, v_2), \ldots, (v_{i-1}, v_i)\}$ is a shortest s-v_i path.*

Proof. If we have a s-v_i path Q that is shorter, then replacing the s-v_i part in P with Q would give a shorter s-t path.

[1]Given an undirected graph (without costs) and nodes s,t, it is known to be **NP**-hard to test, whether there is a s-t path visiting all nodes exactly once. Now put edge cost -1 on all edges. If you can find a simple s-t path of cost $n-1$, then this must be a path that visits all nodes once.



□

3.1 Dijkstra's algorithm

In this lecture we see the most efficient algorithm to compute shortest paths from a given vertex s to all vertices v in a digraph G without negative cost cycles.

Dijkstra's algorithm (Dijkstra 1959)

Input: A directed graph $G = (V, E)$ with edge costs c_e. A source vertex s.
Assumption: $c_e \geq 0 \; \forall e \in E$
Output: Length $\ell(v)$ of shortest s-v-paths for all $v \in V$.

(1) Set $\ell(s) = 0$. $R = \{s\}$ and for all $v \in V \backslash \{s\}$: $\ell(v) = \begin{cases} c(s,v) & (s,v) \in E \\ \infty & \text{otherwise} \end{cases}$

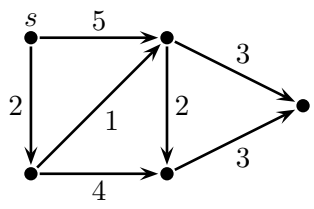
(2) WHILE $R \neq V$ DO

(3) Select $v \in V \backslash R$ attaining $\min\{\ell(v) \mid v \in V \backslash R\}$

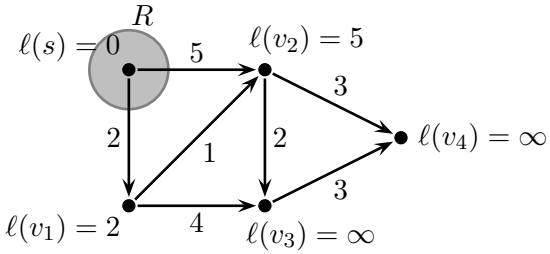
(4) FOR all $w \in V \backslash R$ with $(v, w) \in E$ DO $\ell(w) := \min\{\ell(w), \ell(v) + c(u, w)\}$

(5) Set $R = R \cup \{v\}$.

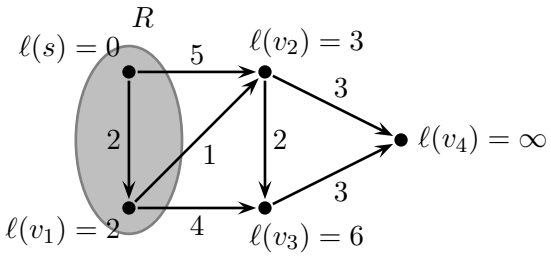
An example run for Disjkstra's algorithm is as follows:



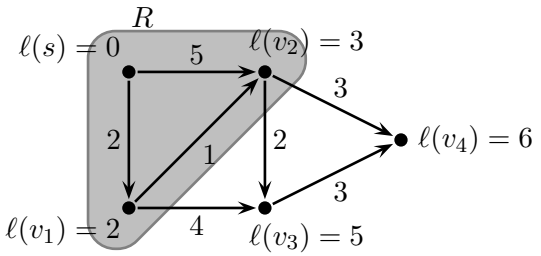
instance



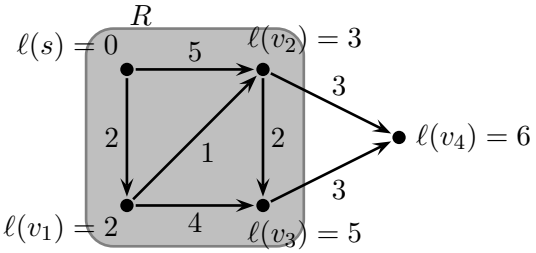
before it 1



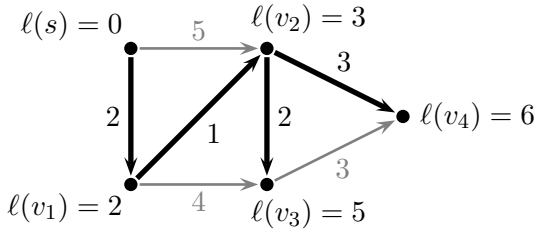
after it. 1



after it. 2



after it. 3



solution

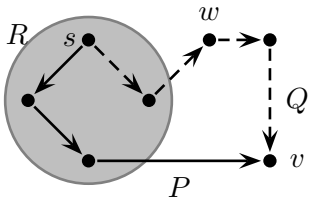
We can easily modify the algorithm so that it also computes the shortest paths itself. For each node $v \in V \backslash \{s\}$, we pick one node $p(v)$ so that $d(v) = \ell(p(u)) + c(u, v)$. Then from each node v we can find the shortest path from the root, by repeatedly going back to $p(v)$. Those edges $(p(v), v)$ are depicted in bold in the last figure. If there is no s-v path, then we will have $\ell(v) = \infty$.

Lemma 8. *Let $d(u, v)$ be the length of the shortest u-v path in graph $G = (V, E)$. After the termination of Dijkstra's algorithm we have $\ell(v) = d(s, v)$ for all $v \in V$.*

Proof. We prove that the following two invariants are satisfied after each iteration of the algorithm:

- **Invariant I.** For all $v \in R$ one has: $\ell(v) = d(s, v)$.
- **Invariant II.** For all $v \in V \backslash R$ one has: $\ell(v) =$ length of the shortest path from s to v that has all nodes in R, except of v itself.

At the end of the algorithm, we have $R = V$ and the claim follows from invariant I. Both invariants are true before the first iteration when $R = \{s\}$. Now consider any iteration of the algorithm; assume that both invariants were satisfied at the beginning of the iteration and that v is the node that was selected. Let P be the shortest s-v path that has all nodes except of v itself in R; the cost is $c(P) = \ell(v)$ by invariant II. Now suppose that this is not the shortest s-v path and that there is a cheaper s-v path Q with $c(Q) < c(P) = \ell(v)$. Let w be the first node on the s-v path Q that lies outside of R (this point must exist since Q starts in R and ends outside; also that point cannot be v since otherwise by invariant II, we would have $c(P) = c(Q)$). Since we have only non-negative edge costs, $\ell(w) \leq c(Q) < \ell(v)$. But that means the algorithm would have processed w instead of u. This is a contradiction. It follows that invariant I is maintained.



For invariant II, consider a node $w \in V \backslash R$. There are two possibilities for the shortest s-w path in Q: either it does not use the newly added node v, then $\ell(w)$ remains valid. Or the shortest s-w path does use v, then $\ell(w) = \ell(u) + c(u, w)$ is the correct length. Hence also invariant II remains valid. □

Lemma 9. *Dijkstra's algorithm runs in $O(n^2)$ time in an n-node graph.*

Proof. The algorithm has n iterations and the main work is done for updated the $\ell(w)$-values. Note that one update step $\ell(w) := \min\{\ell(w), \ell(v) + c(u,w)\}$ takes time $O(1)$ and we perform n such updates per iteration. In total the running time is $O(n^2)$. □

Again with more sophisticated data structures (so called Fibonacci heaps), the running time can be reduced to $O(|E| + |V| \log |V|)$. Recall that Dijkstra's algorithm computes the lengths of n shortest paths $d(s,v)$. Interestingly, there is no known algorithm that is faster, even if we are only interested in a *single* distance $d(s,t)$.

3.2 The Moore-Bellman-Ford Algorithm

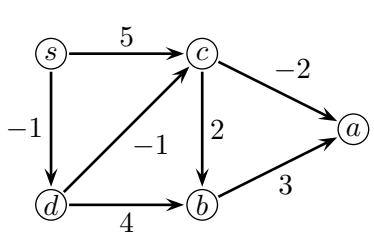
We now study a second algorithm for finding shortest paths in a directed graph. Like Dijkstra's algorithm it computes shortest paths from a source node to *all* other nodes. But in contrast to Dijkstra's algorithm, it can deal with negative edge cost — at the expense of a higher running time.

Moore-Bellman-Ford Algorithm

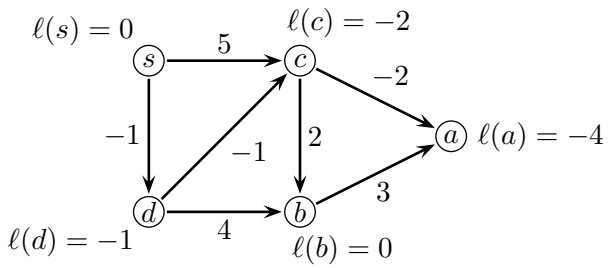
Input: a directed graph $G = (V, E)$; edge cost $c(e) \in \mathbb{R}$ $\forall e \in E$; a source node $s \in V$.
Assumption: There are no negative cost cycles in G
Output: The length $\ell(v)$ of the shortest s-v path, for all $v \in V$.

(1) Set $\ell(s) := 0$ and $\ell(s) = \infty$ for all $v \in V \backslash \{s\}$
(2) For $i = 1, \ldots, n-1$ do
  (3) For each $(v,w) \in E$ do:
    If $\ell(w) > \ell(v) + c_{vw}$ then set $\ell(w) = \ell(v) + c_{vw}$ ("update edge (u,w)")

An example run of the algorithm can be found below (after iteration 3 we already have all the correct labels in this case):



instance



labels at end

		$\ell(s)$	$\ell(a)$	$\ell(b)$	$\ell(c)$	$\ell(d)$
start		0	∞	∞	∞	∞
Iteration 1	(b,a)					
	(c,a)					
	(c,b)					
	(d,b)					
	(d,c)					
	(s,c)				5	
	(s,d)					-1
Iteration 2	(b,a)					
	(c,a)		3			
	(c,b)			7		
	(d,b)			3		
	(d,c)				-2	
	(s,c)					
	(s,d)					
Iteration 3	(b,a)					
	(c,a)		-4			
	(c,b)			0		
	(d,b)					
	(d,c)					
	(s,c)					
	(s,d)					
end		0	-4	0	-2	-1

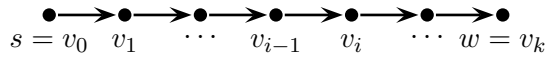
Theorem 10. *The Moore-Bellman-Ford Algorithm runs in $O(nm)$-time.*

Proof. The outer loop takes $O(n)$ time since there are $n-1$ iterations. The inner loop takes $O(m)$ time as it checks the l-value at the head vertex of every edge in the graph. This gives an $O(nm)$-algorithm. □

Theorem 11. *The Moore-Bellman-Ford algorithm works correctly.*

Proof. Let $d(u,v)$ be the actual length of the shortest u-v path. We want to show that indeed after the termination of the algorithm one has $\ell(u) = d(s,u)$ for all $u \in V$. First, by induction one can easily prove that in any iteration one has $\ell(u) \geq d(s,u)$. To see this, suppose in some iteration we update edge (v,w), then $\ell(w) = \ell(v) + c_{vw} \geq d(s,v) + c(v,w) \geq d(s,w)$ by the triangle inequality.

In the following, we will show that after $n-1$ iterations of the outer loop of the algorithm, we have $\ell(w) = d(s,w)$ for each node. Hence, consider a shortest path $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k = w$ from s to some node w.



Suppose after $i-1$ iterations, we have $\ell(v_{i-1}) = d(s,v_{i-1})$. Then in the ith iteration we will consider edge (v_{i-1},v_i) for an update and set $\ell(v_i) := \ell(v_{i-1}) + c(v_{i-1},v_i) = d(s,v_{i-1}) + c(v_{i-1},v_i) = d(s,v_i)$ (if that was not already the case before). Note that here we have implicitly used that $s \to v_1 \to$

$\ldots \to v_{i-1}$ is also the shortest path from s to v_{i-1}. Since all shortest paths include at most $n-1$ edges, at the end we have computed all distances from s correctly. □

Again, one can also extract the shortest paths itself, once we know the labels $\ell(v)$. Finally, we should remark that the speed at which the values $\ell(u)$ propagate does depend on the order of the edges. Suppose we had sorted the edges as $e_1, \ldots, e_m$ so that $d(s, \text{tail}(e_1)) \leq \ldots \leq d(s, \text{tail}(e_m))$, then already after *one* iteration, all the values $\ell(v)$ would coincide with the actual distance $d(s, v)$.

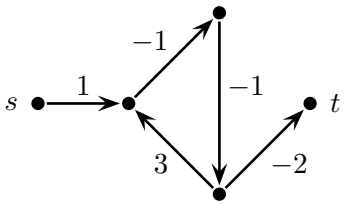
3.3 Detecting negative cycles

The algorithms we saw so far compute the shortest (s, v)-paths in a digraph G from a start vertex s and any vertex v in G. These algorithms require that G has no negative cost cycles, in which case we say that the vector of costs, c, is **conservative**. In this lecture we see how one can detect whether G has a negative cost cycle.

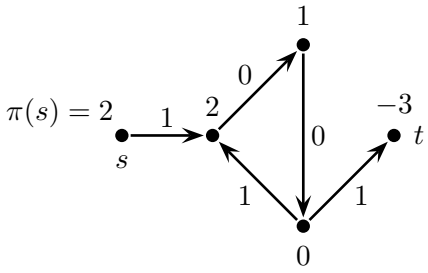
Definition 3. Let G be a digraph with costs $c_e \in \mathbb{R}$ for all $e \in E$. Let $\pi : V \to \mathbb{R}$ be a function that assigns a real number $\pi(v)$ to every vertex $v \in V$.

1. For an edge $(i, j) \in E$, define the **reduced cost** with respect to π to be $c_{ij} + \pi(i) - \pi(j)$.
2. If the reduced costs of all edges of G are nonnegative, we say that π is a **feasible potential** on G.

For example, below we have an example with negative edge cost (but no negative cycle) and a feasible potential on the right:



graph with cost c_{ij}



edges labelled with reduced cost $c_{ij} + \pi(i) - \pi(j)$

nodes labelled with feasible potentials π

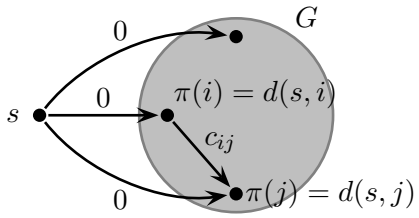
Theorem 12. *The directed graph G with vector of costs c has a feasible potential if and only if c is conservative.*

Proof. ($\Rightarrow$): Suppose π is a feasible potential on (G, c). Consider a directed cycle with edges C. Then adding up the cost of the edges gives

$$\sum_{e \in C} c(e) = \sum_{e \in C} \underbrace{\big(c(e) + \pi(\text{tail}(e)) - \pi(\text{head}(e))\big)}_{\geq 0} \geq 0$$

using that each node on the cycle appears once as $tail(e)$ and once as $head(e)$.

($\Leftarrow$): Suppose c is conservative. Augment G to a graph $\tilde{G}$ by adding a new vertex s and adding edges (s, v) with cost $c(s, v) = 0$. This does not create negative cost cycles because s has only outgoing edges.



Now let $d(s,i)$ be the length of the shortest s-i path in $\tilde{G}$. Note that these values might be negative, but they are well defined since there are no negative cost cycles and from s we can reach every node. We claim that $\pi(i) := d(s,i)$ is a feasible potential. Consider any edge $(i,j) \in E$, then its reduced cost is

$$c(i,j) + d(s,i) - d(s,j) \geq 0 \Leftrightarrow d(s,j) \leq d(s,i) + c(i,j)$$

which is just the triangle inequality. □

Lemma 13. *Given (G,c), in $O(nm)$ time we can find either a feasible potential on G or a negative cost cycle in G.*

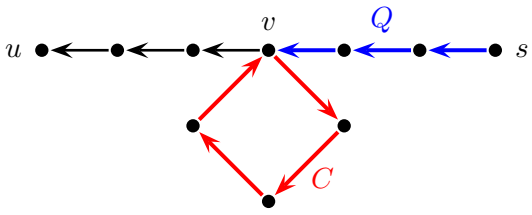
Proof. Let G be the given graph and add again a source node s that is connected to each node at cost 0. Now run the Moore-Bellman-Ford algorithm to compute labels $\ell(v)$. If no edge update is made in the last iteration, then we know that the labels satisfy $\ell(v) \leq \ell(u) + c_{uv}$ for all edges $(u,v) \in E$. In fact, in the last proof we implicitly argued that such labels define a feasible potential.

Now, let us make a couple of observations concerning the behaviour of the algorithm. Where do the labels $\ell(v)$ actually come from? Well, an easy inductive argument shows that at any time in the algorithm and for any v, there is an s-v path P so that $\ell(v) = c(P)$.

Claim: Suppose a label $\ell(v)$ has been updated in iteration k. Then there is a s-v path with *at least* k edges so that $\ell(u) = c(P)$.

Proof: If we updated $\ell(v)$, then we had $\ell(v) = \ell(u) + c(u,v)$ at that time with $(u,v) \in E$. The only reason why $\ell(v)$ was not updated earlier is that u itself was updated either in iteration k or in iteration $k-1$. The argument follows by induction. □

Now suppose that a node u was updated in the nth iteration. Then we can find a path P with at least n edges so that $\ell(u) = c(P)$. Since the path has n edges, there must be a node, say v that appears twice on this path. Say $\ell'(v)$ was the label at the earlier update and $\ell''(v) < \ell'(v)$ was the label at the later update. Let Q be the part of P that goes from s to v and C be the cycle from v to v. Then $\ell'(v) = c(Q)$ white $\ell''(v) = c(Q) + c(C) < \ell'(v) = c(Q)$. Hence $c(C) < 0$.



□

Intuitively, the concept of potentials to "correct" the edge cost so that they become non-negative while not changing the shortest paths itself. This provides the following insight:

Corollary 14. *Given a directed weighted graph with no negative cost cycles, $|V| = n$ and $|E| = m$. Then one can compute the lengths $(d(u,v))_{u,v\in V}$ of all shortest paths simultaneously in time $O(nm + n^2 \log n)$.*

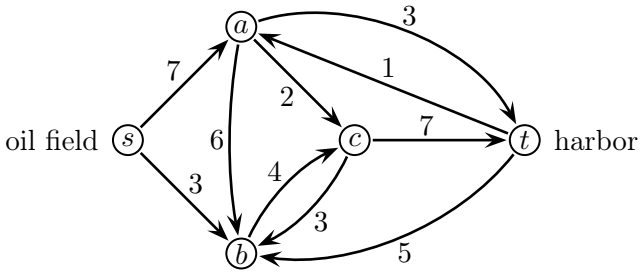
Proof. Since there are no negative cost cycles, we can compute a feasible potential π in time $O(nm)$. Suppose that $d_\pi(u,v)$ gives the length of the shortest path w.r.t. the reduced costs. Observe that $d_\pi(u,v) = d(u,v) + \pi(u) - \pi(v)$. Hence a path is a shortest path w.r.t. the original costs if and only if it is a shortest path for the reduced costs. The advantage is that the reduced costs are non-negative, hence we can apply Dijkstra's algorithm n times (once with each node as source node). Each application takes time $O(m + n \log n)$. Hence we obtain a total running time of $O(nm) + n \cdot O(n \log(n))$ which is of the claimed form. □

Chapter 4

Network flows

In this chapter, we want to discuss network flows and in particular how to compute maximum flows in directed graphs.

Imagine you work for an oil company that owns an oil field (at node s), a couple of pipelines and a harbor (at node t). The pipeline network is depicted below. Each pipeline segment e has a maximum capacity u_e (say tons of oil per minute that we write on the edges). For technical reasons, each pipeline segment has a direction and the oil can only be pumped into that direction. What is the maximum amount of oil that you can transport from the oil field to the harbor?



Let us make a couple of definitions that will help us to formally state the underlying mathematical problem.

Definition 4. A **network** (G, u, s, t) consists of the following data:

- A directed graph $G = (V, E)$,
- edge capacities $u_e \in \mathbb{Z}_{\geq 0}$ for all $e \in E$, and
- two specified nodes s (source) and t (sink) in V.

Definition 5. A ***s*-*t* flow** is a function $f : E \to \mathbb{R}_{\geq 0}$ with the following properties:

- The flow respects the capacities, i.e. $0 \leq f(e) \leq u_e \; \forall e \in E$.
- It satisfies **flow conservation** at each vertex $v \in V \backslash \{s, t\}$:

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

  where $\delta^-(v) = \{(w, v) \in E\} = \{$ edges entering $v \}$ and $\delta^+(v) = \{(v, w) \in E\} = \{$ edges leaving $v \}$.

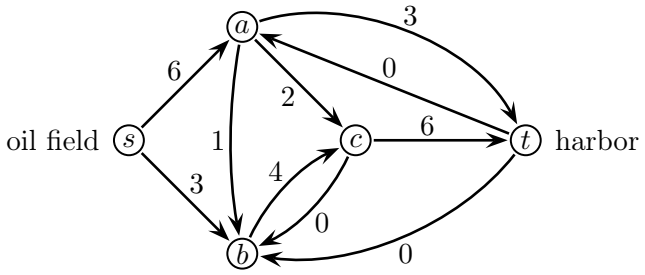
The **value** of an s-t flow f is

$$\text{value}(f) := \sum_{\delta^+(s)} f(e) - \sum_{\delta^-(s)} f(e)$$

which is the total flow leaving the source node s minus the total flow entering s.

MAXIMUM FLOW PROBLEM
Input: A network (G, u, s, t).
Find: An s-t flow in the network of maximum value.

In our example application, it turns out that the maximum flow f has a value of $\text{value}(f) = 9$ and the flow f itself is depicted below label edges e with $f(e)$; observe that indeed we respect the capacities.



Note that we do allow flows to be non-integral.

4.1 The Ford-Fulkerson algorithm

We want to develope some intuition on what a maximum flow algorithm should do. The first, naive approach might be as follows:

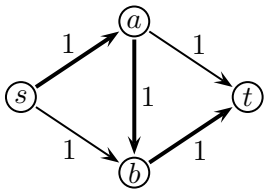
(1) Set $f(e) := 0 \; \forall e \in E$

(2) REPEAT

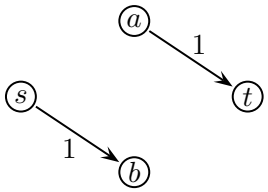
  (3) Find any s-t path $P \subseteq E$

  (4) FOR $e \in E$ set $f(e) := f(e) + 1$ and $u(e) := u(e) - 1$ (remove e if $u(e) = 0$)

Consider the following example instance (again, edges are labelled with capacities $u(e)$):



Obviously the maximum flow value is 2. Now let us see how our algorithm would perform on the example. We could decide to select the path $P = s \to a \to b \to t$ and send a flow of 1 along it. But after removing a capacity of 1 along P, we end up with the following remaining capacities:



That means we cannot improve the flow of value 1 anymore. The problem is that we made the mistake of sending flow along edge (a, b) and we need a way of "correcting" such mistakes later in the algorithm. The key idea to do that is using **residual graphs**.

Definition 6. Given a network $(G = (V, E), u, s, t)$ with capacities u and an s-t flow f. The **residual graph** $G_f = (V, E_f)$ has edges

$$E_f = \underbrace{\{(i,j) \in E : u(i,j) - f(i,j) > 0\}}_{\text{forward edges}} \cup \underbrace{\{(j,i) \mid (i,j) \in E \text{ and } f(i,j) > 0\}}_{\text{backward edges}}$$

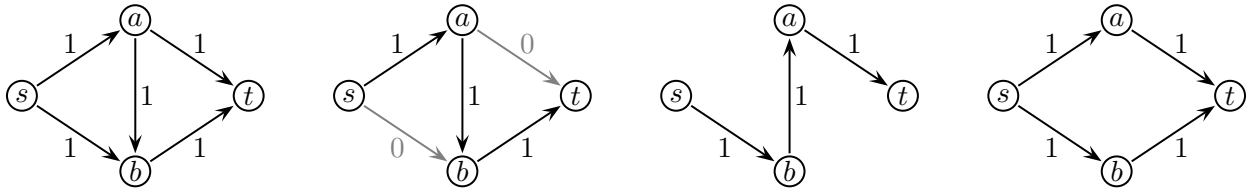
An edge e has **residual capacity**

$$u_f(i,j) = \underbrace{(u(i,j) - f(i,j))}_{\text{forward capacity}} + \underbrace{f(j,i)}_{\text{backward capacity}}$$

An s-t path in G_f is called an f **augmenting path**.

Given a flow f and an s-t path P in G_f, to **augment** f along P by γ means to do the following for each $e \in P$: Increase $f(e)$ by γ on all the forward edges of the path and decrease $f(e)$ by γ on all the backward edges of P.

We can solve our small example using the concept of residual graphs as follows:



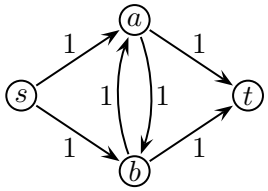
capacities u_e | flow f | residual graph G_f & augmenting path $P \subseteq E_f$ | f augmented by P

For a flow f, we define the **excess** of a node as $\text{ex}_f(v) = \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e)$. In particular, for an s-t flow, we have $\text{ex}_f(v) = 0$ for all $v \in V \backslash \{s, t\}$ and $\text{ex}_f(s) = \text{value}(f)$ and $\text{ex}_f(t) = -\text{value}(f)$.

Lemma 15. *Let (G, u, s, t) be a network with an s-t flow f. Let P be a f-augmenting path so that $u_f(e) \geq \gamma \; \forall e \in P$. Let us augment f along P by γ and call the result f'. Then f' is a feasible s-t flow of value value$(f') =$ value$(f) + \gamma$.*

Proof. By definition, we never violate any edge capacity (and also $f'(e) \geq 0$). Now we need to argue that $\text{ex}_{f'}(v) = 0$ for all $v \in V \backslash \{s, t\}$. Let us define a flow g that we get by naively adding value to edges in P

$$g(e) = \begin{cases} f(e) + \gamma & e \in P \\ f(e) \end{cases}$$



flow g in our example above

For any node $v \in V \backslash \{s,t\}$ on the path P has now γ more flow incoming and γ more flow outgoing. Hence $ex_g(v) = 0$. Moreover, t has flow γ more ingoing than before, hence $\mathrm{ex}_g(t) = \mathrm{ex}_f(t) + \gamma$. But g is not yet f' because P might have contained a backward edge (i,j). That means $g(i,j) \geq \gamma$ and $g(j,i) \geq \gamma$. Now we reduce the flow value on both (i,j) and (j,i) by γ and we obtain f'. Note that this "flow cancelling" does not change the excess. Hence also $\mathrm{ex}_{f'}(v) = 0$ for all $v \in V \backslash \{s,t\}$. □

Ford and Fulkerson's algorithm for Max Flows

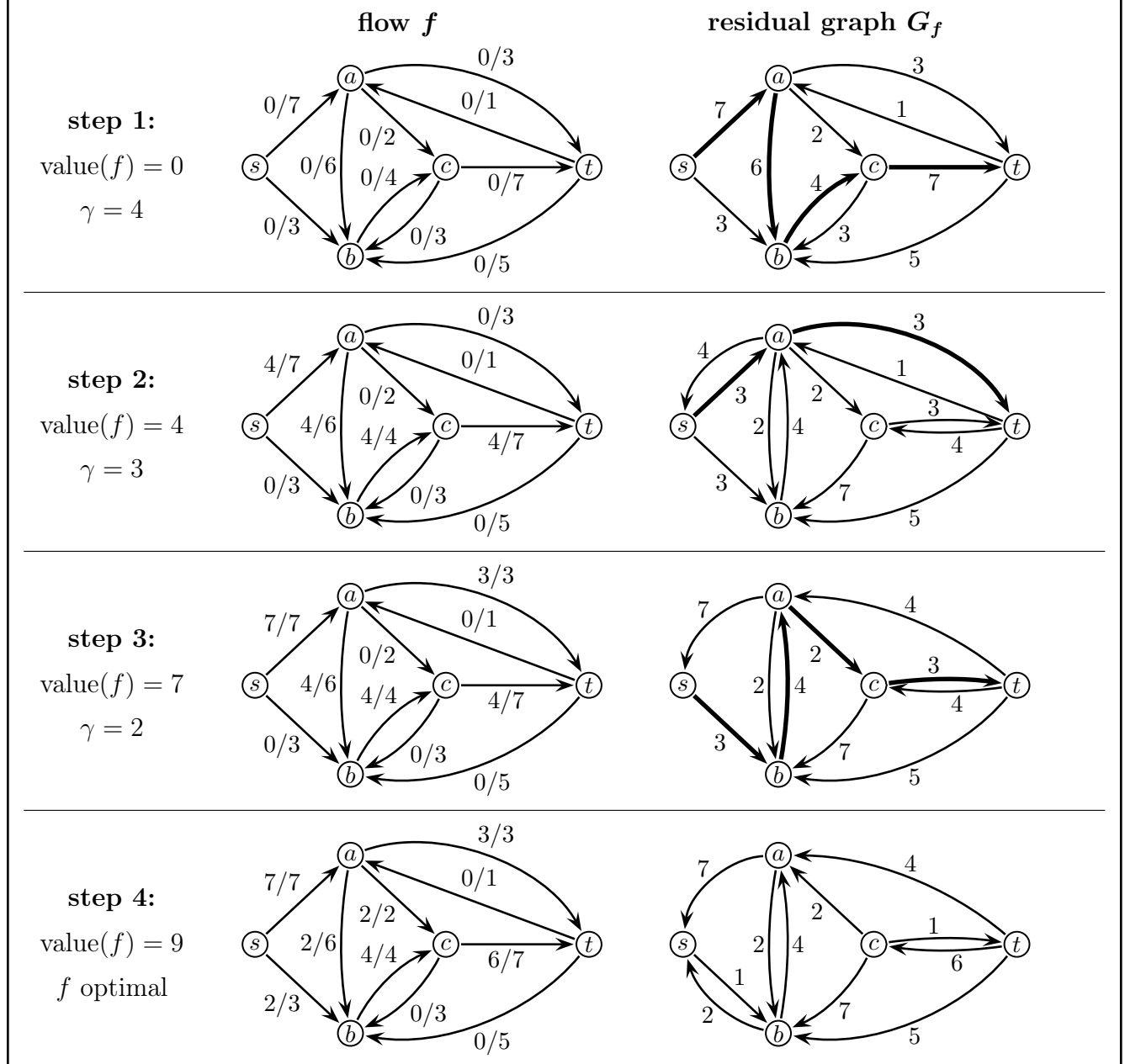
Input: A network (G, u, s, t).
Output: A max (s,t)-flow in the network.

(1) Set $f(e) = 0$ for all $e \in E$.

(2) Find an f-augmenting path P. If none exists then stop.

(3) Compute $\gamma := \min\{u_f(e) \mid e \in P\}$. Augment f along P by γ and go to (2).

Example: Ford-Fulkerson Algorithm



Edges on left hand side are labelled with $f(e)/u(e)$. Edges on right hand side are labelled with residual capacities $u_f(e)$. The augmenting path $P \subseteq E_f$ that is selected is drawn in G_f in bold. The value $\gamma = \min\{u_f(e) \mid e \in P\}$ gives the minimum capacity on any edge on P.

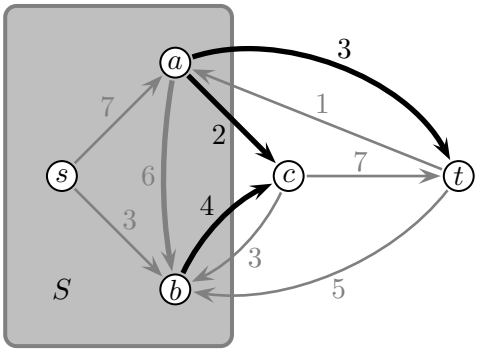
The natural question is: does this algorithm actually find the optimum solution? We will find out that the answer is yes. Before we proceed with proving that, we want to introduce some more concepts.

4.2 Min Cut problem

Next, we want to introduce a problem that will turn out to be the **dual** to the Max Flow problem.

> s-t Min Cut Problem
> **Input:** A network (G, u, s, t).
> **Find:** A set $S \subseteq V$ with $s \in S, t \notin S$ (i.e. an s-t **cut**) minimizing $u(\delta^+(S)) = \sum_{(i,j)\in E: i\in S, j\notin S} u(i,j)$.

In our example application, the minimum s-t cut would look as follows (the bold edges are in the cut $\delta^+(S)$).



An s-t cut of value $2 + 4 + 3 = 9$.

Recall that $u(\delta^+(S)) = \sum_{(i,j)\in E: i\in S, j\notin S} u(i,j)$ denotes the capacity of the edges leaving S. Let us now show a kind of "weak duality theorem" for MinCut/MaxFlow:

Lemma 16. *Let (G, u, s, t) be a network with an s-t flow f and an s-t cut $S \subseteq V$. Then value$(f) \leq u(\delta^+(S))$.*

Proof. The claim itself is not surprising: if the flow goes from s to t, then at some point it has to "leave" S. Mathematically, we argue as follows:

$$\text{value}(f) \overset{\text{def. of flow}}{=} \sum_{e\in\delta^+(s)} f(e) - \sum_{e\in\delta^-(s)} f(e) = \sum_{v\in S} \Big(\underbrace{\sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e)}_{=0 \text{ if } v\neq s} \Big)$$

$$\overset{(*)}{=} \sum_{e\in\delta^+(S)} \underbrace{f(e)}_{\leq u(e)} - \sum_{e\in\delta^-(S)} \underbrace{f(e)}_{\geq 0} \leq u(\delta^+(S))$$

In $(*)$, we drop edges e that run inside S from the sums. The reason is that they appear twice in the summation: once with a "+" and once with a "−", so their contribution is 0 anyway. □

Actually, the last proof gives us an insight for free by simply inspecting when the inequality would be an equality:

Corollary 17. *In the last proof we have value$(f) = u(\delta^+(S))$ if and only if both conditions are satisfied:*

- *All outgoing edges from S are saturated:* $f(e) = u(e)\ \forall e \in \delta^+(S)$
- *All incoming edges into S have 0 flow:* $f(e) = 0\ \forall e \in \delta^-(S)$.

Now we can show that if the Ford-Fulkerson stops, then it has found an optimum flow.

Theorem 18. *An (s,t)-flow f is optimal if and only if there does not exist an f-augmenting path in G_f.*

Proof. ($\Rightarrow$): This is clear, because we already discussed that if there exists an f-augmenting path in G_f, then f is not maximal (simply because the augmented flow will have a higher value).
($\Leftarrow$): Now suppose that we have an s-t flow f and there is no augmenting path in G_f anymore. We need to argue that this flow is optimal. The proof is surprisingly simple: Choose

$$S := \{v \in V \mid \exists s\text{-}v \text{ path in } G_f\}$$

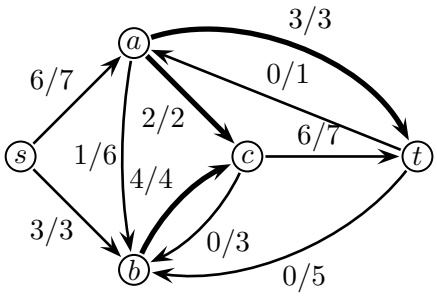
Note that $s \in S$ and $t \notin S$, hence S is an s-t cut. We claim that value$(f) = u^+(\delta(S))$, which by Lemma 16 shows that the flow is optimal. We verify the conditions from Cor. 17:

- *An outgoing edge $e = (i,j) \in \delta^+(S)$ has $f(e) = u(e)$.* That is true because if $f(i,j) < u(i,j)$, then (i,j) is an edge in the residual network G_f and j would be reachable from s via i. That's a contradiction.
- *An incoming edge $e = (j,i) \in \delta^-(S)$ has $f(e) = 0$.* Similarly, if $f(j,i) > 0$, then the edge (i,j) exists in the residual network G_f and again j would be reachable via i.

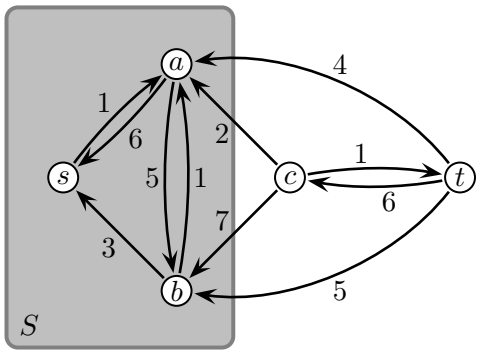
□

Combining Lemma 16, Cor. 17 and Theorem 18 we conclude the max flow min cut theorem:

Theorem 19 (MaxFlow = MinCut (Ford-Fulkerson 1956))**.** *The max value of an $s-t$ flow in the network (G,u,s,t) equals the min capacity of a $s-t$ cut in the network.*



maximum flow; edges labelled with $f(e)/u(e)$



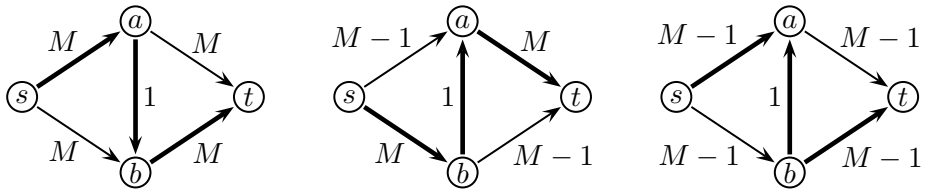
residual graph G_f with cap u_f

Remark 1. Note that the Ford-Fulkerson algorithm not only finds a max flow but also the min cut in the network.

Lemma 20. *For the network (G,u,s,t), let $U := \max\{u(e) \mid e \in E\}$ be the maximum capacity. Then the Ford-Fulkerson algorithm runs in time $O(n \cdot m \cdot U)$.*

Proof. Since s has at most n edges of capacity at most U outgoing, any flow has value$(f) \leq n \cdot U$. In each iteration of the algorithm, the value of the flow increases by at least 1. Hence, the algorithm needs at most $n \cdot U$ iterations. Each iteration (i.e. constructing E_f and finding an augmenting path) can be done in time $O(m)$. □

Note that the quantity U is exponential in the bit encoding length of the input, hence $O(n \cdot m \cdot U)$ is not a polynomial running time. Moreover, we cannot improve the analysis by much. Consider the following example where we always augment along the bold paths $P = s \to a \to b \to t$ or $P = s \to b \to a \to t$.



G with cap u | G_{f_1} with cap u_{f_1}, value$(f_1) = 1$ | G_{f_2} with cap u_{f_2}, value$(f_2) = 2$

We see that in each iteration, we indeed gain only one flow unit. Thus it takes $2M$ iterations to find the optimum. It seems that the problem was that Ford Fulkerson chooses an **arbitrary** augmenting path instead of making a smarter choice. This is what we want to do next.

4.3 The Edmonds-Karp algorithm

So, we want to modify the Ford-Fulkerson algorithm by selecting an augmenting path in a more clever way — it turns out that it suffices to always pick the shortest path in the residual graph G_f!

Edmonds-Karp algorithm for Max Flows

Input: A network (G, u, s, t).
Output: A max s-t flow in the network.

(1) Set $f(e) = 0$ for all $e \in E$.

(2) WHILE $\exists$ s-t path in G_f DO

  (3) Find a **shortest** f-augmenting path P.

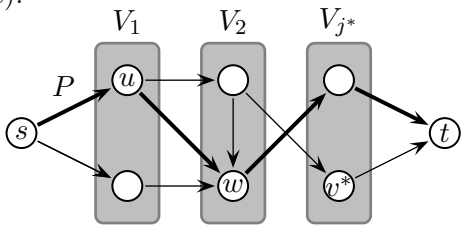
  (4) $\gamma := \min\{u_f(e) \mid e \in P\}$ (e attaining minimum is "bottleneck edge")

  (5) Augment f along P by γ

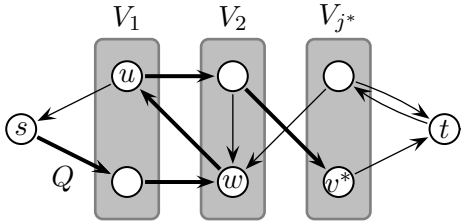
Theorem 21. *The Edmonds-Karp algorithm requires at most $m \cdot n$ iterations and has running time $O(m^2 n)$.*

Proof. Each iteration can be done in time $O(m)$ using **breadth first search**. For bounding the number of iterations it suffices to show that each edge e appears at most n times as bottleneck edge. Let $f_0, f_1, \ldots, f_T$ be the sequence of flows with $0 = \text{value}(f_0) < \text{value}(f_1) < \ldots < \text{value}(f_T)$ that the algorithm maintains. Let $d_i(s, v)$ be the length of a shortest s-v path in the residual graph G_{f_i} w.r.t. the number of edges. Note that the flow f changes from iteration to iteration and with it

possibly $d_i(s,v)$. We make the important observation that the distances from s never decrease:
Claim: For all $v \in V$, $d_0(s,v) \leq d_1(s,v) \leq \ldots \leq d_T(s,v)$.
Proof. Consider a fixed node v^* and some iteration i. We want to argue that $d_i(s,v^*) \leq d_{i+1}(s,v^*)$. We partition the nodes into blocks $V_j := \{w \mid d_i(s,w) = j\}$ of nodes that have distance j to the source. Note that the augmenting path P that is used in iteration i will have exactly one node in each block V_j and it will only use edges that go from V_j to V_{j+1} (since P is the shortest path possible).



graph G_{f_i} with augm. path P



graph $G_{f_{i+1}}$ with s-v path Q

Suppose that $v^* \in V_{j^*}$. The only edges that are in $E_{f_{i+1}} \backslash E_{f_i}$ are edges on P reversed, hence the only new edges in $E_{f_{i+1}} \backslash E_{f_i}$ run from V_{j+1} to V_j for some j. Hence an s-v^* path Q in $G_{f_{i+1}}$ can never "skip" one of the blocks V_j, hence $d_{i+1}(s,v^*) \geq j^*$. That shows the claim. □

Claim: Suppose that (u,w) was the bottleneck in two iterations i and $i' > i$. Then $d_{i'}(s,u) > d_i(s,u)$.
Proof: Consider the situation after iteration i: the edge (u,w) has been the bottleneck and now it exists only as (w,u) edge in the residual graph. The edge (u,w) can only reappear in the residual graph if at some point (w,u) lies on the shortest path. In particular, it must be the case that $d_{i''}(s,w) = d_{i''}(s,u) - 1$ for some iteration $i' < i'' < i'$. But since $d_i(s,w) = d_i(s,u) + 1$ and distances never decrease, we must have $d_{i'}(s,u) \geq d_{i''}(s,u) > d_i(s,u)$. □

Since $d_i(s,u) \in \{1,\ldots,n-1,\infty\}$ the last claim implies that each edge can be bottleneck edge at most n times. □

There are many more Max Flow algorithms. The so-called **Push-Relabel algorithm** needs running time $O(n^3)$. Orlin's algorithm from 2012 solves MaxFlow even in time $O(n \cdot m)$ (where again $n = |V|$ and $m = |E|$).

4.3.1 Some remarks

We want to conclude this algorithmic section with two facts that we already proved implicitly:

Corollary 22. *In a network (G,u,s,t) one can find an optimum s-t MinCut in time $O(m^2 n)$.*

Proof. Run the Edmonds-Karp algorithm to compute a maximum flow f. As in the MaxCut=MinCut Theorem, we know that the set S of nodes that are reachable from s in the residual graph G_f will be a MinCut! □

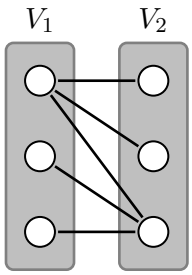
Corollary 23. *In a network (G,u,s,t) with integral capacities (i.e. $u(e) \in \mathbb{Z}_{\geq 0}$ $\forall e \in E$) there is always a maximum flow f that is integral.*

Proof. If the capacities $u(e)$ are integral, then the increment γ that is used in either Ford-Fulkerson or in Edmonds-Karp is integral as well. Then also the resulting flow is integral. □

4.4 Application to bipartite matching

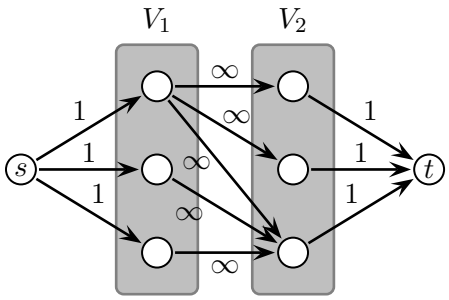
Recall that **matching** $M \subseteq E$ in an undirected graph $G = (V, E)$ is a set of edges so that each node is incident to at most one those edges. A **maximum matching** is a matching that maximizes the number $|M|$ of edges. Here we want to show that Max Flow algorithms can also be used to compute maximum matchings.

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph, like the one below.



Now, make all the edges directed from V_1 to V_2 and add new nodes s, t. We set capacities u as

$$u(i,j) = \begin{cases} 1 & i = s \text{ or } j = t \\ \infty & (i,j) \in V_1 \times V_2 \end{cases}$$

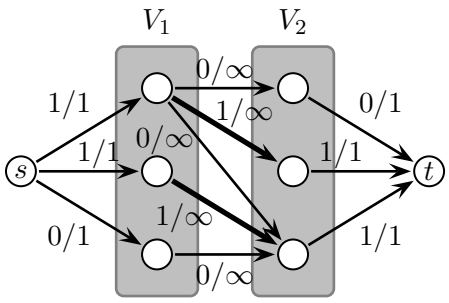


We claim the following

Lemma 24. *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph and (G', u, s, t) be the network constructed as above. Then*

$$\max\{|M| : M \subseteq E \text{ matching } \} = \max\{\text{value}(f) \mid f \text{ is } s\text{-}t \text{ flow in } G'\}$$

Proof. Suppose that M is a matching. Then for any edge $(u, v) \in M$ we can put 1 unit of flow on the path $s \to u \to v \to t$. Hence there is also a flow of value $|M|$. Now suppose that the maximum flow value is k. The Ford-Fulkerson algorithm shows that there is always an **integral** flow f achieving this value, i.e. $f(e) \in \mathbb{Z}$. In our example, we would have a bijection between the bold matching and the depicted flow (edges labelled with $f(e)/u(e)$):



Then

$$M = \{(u, v) \in E \mid f(u, v) = 1\}$$

is a matching of value k as well. □

Actually we can argue the following:

Lemma 25. *Maximum matching in bipartite graphs can be solved in time $O(n \cdot m)$ ($|V| = n$, $|E| = m$).*

Proof. Remember that in a network with integral capacities and maximum flow value of k, the Ford-Fulkerson algorithm has a running time of at most $O(m \cdot k)$. In this case, we have $k \leq n$ and the claim follows. □

4.4.1 Kőnig's Theorem

But the technique of Maximum flows does not only provide us with a more efficient algorithm for maximum matching in bipartite graphs — it also gives us a proof of a very nice structural result. Recall that a **vertex cover** in an undirected graph, is a set of nodes $U \subseteq V$ so that $|\{u, v\} \cap U| \geq 1$ for each edge $\{u, v\} \in E$.

Theorem 26 (Kőnig 1931)**.** *For a bipartite graph $G = (V = V_1 \cup V_2, E)$ one has*

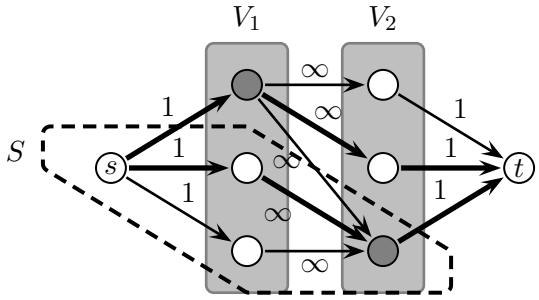
$$\max\{|M| : M \subseteq E \text{ is matching}\} = \min\{|U| : U \subseteq V_1 \cup V_2 \text{ is vertex cover}\}$$

Proof. We will see in the homework, that in every graph (not just in bipartite ones), one has $|M| \leq |U|$ for each matching M and each vertex cover U. Now suppose that the size of the maximum matching is k. We need to argue that there is a vertex cover U with $|U| = k$ nodes. Since the maximum matching in G has size k, also the maximum flow in the network $G' = (V \cup \{s, t\}, E')$ as constructed above has value k. By the **MaxFlow=MinCut Theorem**, we know that this implies that there is an s-t cut $S \subseteq V$ (i.e. $s \in S$, $t \notin S$) of cost k. It remains to show:
Claim: $U := (V_1 \backslash S) \cup (V_2 \cap S)$ is a vertex cover for G of size k.
Proof. Observe that the nodes in the vertex cover can be written as

$$V_1 \backslash S = \{u \in V_1 : (s, u) \in \delta^+(S)\} \quad \text{and} \quad V_2 \cap S = \{v \in V_2 \mid (v, t) \in \delta^+(S)\}$$

Since exactly k edges of the form (s, u) or (v, t) are cut, we have $|U| = k$. It remains to show that U is a vertex cover. Consider any edge $(u, v) \in E$. If (u, v) is not covered by U, then his means that $u \in S$ and $v \notin S$. But then $(u, v) \in \delta^+(S)$ which is impossible because it means that the cut cuts an edge of infinite capacity. □

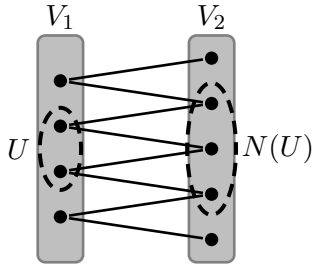
In our little example, we draw a maximum flow (of value 2) together with the mincut S (also of value 2); the two filled nodes in the vertex cover.



For a general graph G, let M be the largest matching and U be the smallest vertex cover. Then it is possible that $|M| < |U|$. On the other hand, one can show that $|M| \leq |U| \leq 2|M|$, that means both quantities are within a factor of 2 of each other. Interestingly, finding the maximum matching in a general graph is solvable in polynomial time (with a non-trivial algorithm), while vertex cover in general graphs is **NP**-hard and hence there won't be a polynomial time algorithm.

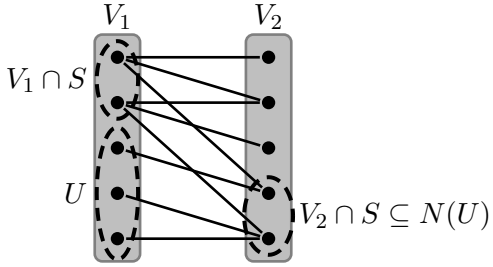
4.4.2 Hall's Theorem

Using our theory of MaxFlows and the developed Kőnigs Theorem we get anothher beautiful theorem in graph theory almost for free. For a bipartite graph $G = (V_1 \cup V_2, E)$ and nodes $U \subseteq V_1$, let $N(U) = \{v \in V_2 \mid \exists (u, v) \in E\}$ be the **neighbors** of U.



Theorem 27 (Hall's Theorem)**.** *A bipartite graph $G = (V_1 \cup V_2, E)$ with $|V_1| = |V_2| = n$ has a perfect matching if and only if $|N(U)| \geq |U|$ for all $U \subseteq V_1$.*

Proof. If there is a set $U \subseteq V_1$ with $|N(U)| < |U|$, then obviously there cannot be a perfect matching, simply because the nodes in U cannot all be matched. For the other direction, assume that the maximum size matching has size at most $n - 1$. Then by Kőnig's Theorem, there is a vertex cover $S \subseteq V_1 \cup V_2$ of size $|S| \leq n - 1$. We make the choice of $U := V_1 \backslash S$.



In particular, S being a vertex cover means that U is only incident to nodes in $V_2 \cap S$, thus $N(U) \subseteq V_2 \cap S$. But since $|V_1 \cap S| + |U| = n$ and $|V_1 \cap S| + |V_2 \cap S| \leq n - 1$, we must have $|N(U)| \leq |V_2 \cap S| < |U|$. $\square$

Chapter 5

Linear programming

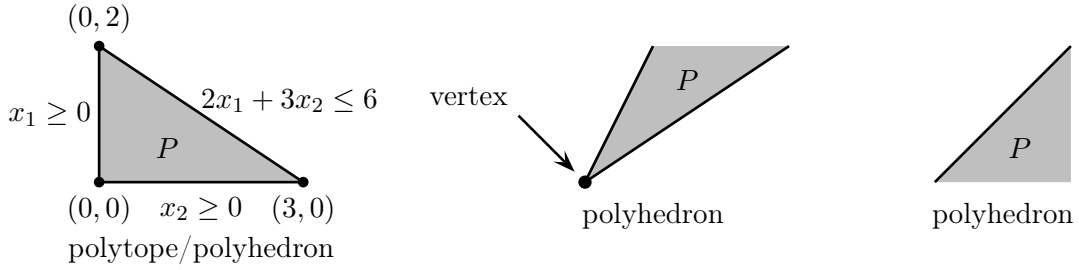
One of the main tools in combinatorial optimization is **linear programming**. We want to quickly review the key concepts and results. Since most statements and proofs are known from course 407, from time to time we will be satisfied with informal proof sketches.

To clearify notation, the set of all real numbers will be denoted as $\mathbb{R}$ and the set of all integers by $\mathbb{Z}$. A function f is said to be real valued if its values are real numbers. For instance a vector $c = (c_1, \ldots, c_n) \in \mathbb{R}^n$ defines the linear function $f : \mathbb{R}^n \to \mathbb{R}$ such that $x \mapsto c \cdot x := c_1x_1 + c_2x_2 + \cdots + c_nx_n$. Linear programs always have linear objective functions $f(x) = c \cdot x$ as above. Note that this is a real valued function since $c \cdot x \in \mathbb{R}$.

A **polyhedron** $P \subseteq \mathbb{R}^n$ is the set of all points $x \in \mathbb{R}^n$ that satisfy a finite set of linear inequalities. Mathematically,

$$P = \{x \in \mathbb{R}^n \, : \, Ax \leq b\}$$

for some matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$. A polyhedron can be presented in many different ways such as $P = \{x \in \mathbb{R}^n \, : \, Ax = b, x \geq \mathbf{0}\}$ or $P = \{x \in \mathbb{R}^n \, : \, Ax \geq b\}$. All these formulations are equivalent. A polyhedron is called a **polytope** if it is bounded, i.e., can be enclosed in a ball of finite radius.

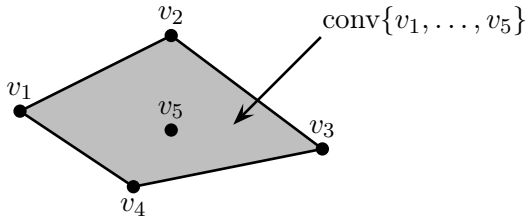


Definition 7. A set $Q \subseteq \mathbb{R}^n$ is **convex** if for any two points x and y in Q, the line segment joining them is also in Q. Mathematically, for every pair of points $x, y \in Q$, the **convex combination** $\lambda x + (1 - \lambda)y \in Q$ for every λ such that $0 \leq \lambda \leq 1$.

Obviously, polyhedra are convex.

Definition 8. A **convex combination** of a finite set of points $v_1, \ldots, v_t$ in $\mathbb{R}^n$, is any vector of the form $\sum_{i=1}^t \lambda_i v_i$ such that $0 \leq \lambda_i \leq 1$ for all $i = 1, \ldots, t$ and $\sum_{i=1}^t \lambda_i = 1$. The set of all convex combinations of $v_1, \ldots, v_n$ is called the **convex hull** of $v_1, \ldots, v_n$. We denote it by

$$\text{conv}\{v_1, \ldots, v_n\} = \Big\{ \sum_{i=1}^n \lambda_i v_i \mid \lambda_1 + \ldots + \lambda_n = 1; \lambda_i \geq 0 \; \forall i = 1, \ldots, n \Big\}$$



Theorem 28. *Every polytope P is the convex hull of a finite number of points (and vice versa).*

For a convex set $P \subseteq \mathbb{R}^n$ (such as polytopes or polyhedra) we call a point $x \in P$ an **extreme point** / **vertex** of P if there is no vector $y \in \mathbb{R}^n \backslash \{\mathbf{0}\}$ with both $x + y \in P$ and $x - y \in P$.

A **linear program** is the problem of maximizing or minimizing a linear function of the form $\sum_{i=1}^n c_i x_i$ over all $x = (x_1, \ldots, x_n)$ in a polyhedron P. Mathematically, it is the problem

$$\min \Big\{ \sum_{i=1}^n c_i x_i \mid Ax \leq b \Big\}$$

for some matrix A and vector b (alternatively max instead of min).

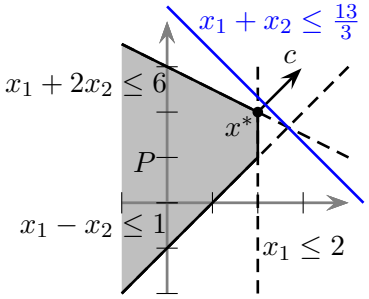
5.1 Separation, Duality and Farkas Lemma

The number 1 key concept in linear optimization is **duality**. We want to motivate this with an example. Consider the linear program

$$\max\{x_1 + x_2 \mid x_1 + 2x_2 \leq 6, \; x_1 \leq 2, \; x_1 - x_2 \leq 1\}$$

(which is of the form $\max\{cx \mid Ax \leq b\}$). First, let us make the following observation: if we add up non-negative multiples of feasible inequalities, then we obtain again an inequality that is valid for each solution x of the LP. For example we can add up the inequalities in the following way:

$$\begin{array}{rrrrl}
\frac{2}{3} \cdot (& x_1 & +2x_2 & \leq & 6) \\
0 \cdot (& x_1 & & \leq & 2) \\
\frac{1}{3} \cdot (& x_1 & -x_2 & \leq & 1) \\
\hline
 & x_1 & +x_2 & \leq & \frac{13}{3} \approx 4.33
\end{array}$$



Accidentially, the feasible inequality $x_1 + x_2 \leq \frac{13}{3}$ that we obtain has the objective function as normal vector. Hence for each $(x_1, x_2) \in P$ we must have $cx \leq \frac{13}{3}$, which provides an **upper bound** on the value of the LP. Inspecting the picture, we quickly see that optimum solution is

$x^* = (2, 2)$ with objective function $cx^* = 4$. Now, let's generalize our observations. Consider the following pair of linear programs

$$\begin{aligned} \textbf{primal} \quad (P): \quad & \max\{cx \mid Ax \leq b\} \\ \textbf{dual} \quad (D): \quad & \min\{by \mid yA = c,\ y \geq \mathbf{0}\} \end{aligned}$$

The dual LP is searching for inequalities $(yA)x \leq yb$ that are feasible for any primal solution x; moreover it is looking for a feasible inequality so that the normal vector $yA = c$ is the objective function, so that yb is an upper bound on the primal LP. In other words: the dual LP is searching for the best upper bound for the primal LP.

Theorem 29 (Weak duality Theorem)**.** *One has $(P) \leq (D)$. More precisely if we have (x, y) with $Ax \leq b$, $yA = c$ and $y \geq \mathbf{0}$, then $cx \leq by$.*

Proof. One has

$$\underbrace{c}_{=yA} x = \underbrace{y}_{\geq \mathbf{0}} \underbrace{Ax}_{\leq b} \leq yb.$$

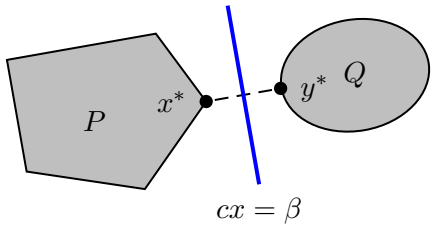
□

In the remainder of this subsection we will show that always $(P) = (D)$, that means we can always combine a feasible inequality that gives a tight upper bound. But first, some tools:

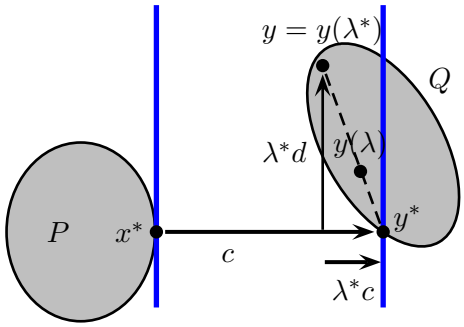
Theorem 30 (Hyperplane separation Theorem)**.** *Let $P, Q \subseteq \mathbb{R}^n$ be convex, closed and disjoint sets with at least one of them bounded. Then there exists a hyperplane $cx = \beta$ with*

$$cx < \beta < cy \quad \forall x \in P\ \forall y \in Q$$

Proof. Let $(x^*, y^*) \in P \times Q$ be the pair minimizing the distance $\|x^* - y^*\|_2$ (this must exist for the following reason: suppose that P is bounded; then P is compact. Then the distance function $d(x) := \min\{\|y - x\|_2 \mid y \in Q\}$ is well-defined and continuous, hence a minimum is attained on P). Then the hyperplane through $\frac{1}{2}(x^* + y^*)$ with normal vector $c = y^* - x^*$ separates P and Q.



To see this, suppose for the sake of contradiction that Q has a point y with $cy < cy^*$. Then we can write this point as $y = x^* + (1 - \lambda^*)c + \lambda^* d$ where d is a vector that is orthogonal to c. Since Q is convex, also the point $y(\lambda) = x^* + (1 - \lambda)c + \lambda d$ for $0 \leq \lambda \leq \lambda^*$ is in Q.



We want to argue that for $\lambda > 0$ small enough, the point $y(\lambda)$ is closer to x^* than y^*.

$$\|x^* - y(\lambda)\|_2^2 = \|(1-\lambda)c + \lambda d\|_2^2 \overset{c \perp d}{=} (1-\lambda)^2\|c\|_2^2 + \lambda^2\|d\|_2^2 = \|c\|_2^2 - 2\lambda\|c\|_2^2 + \lambda^2(\|c\|_2^2 + \|d\|_2^2) < \|c\|_2^2$$

if we choose $\lambda > 0$ small enough since the coefficient in front of the linear λ term is negative. □

This theorem is the finite dimensional version of the **Hahn-Banach separation theorem** from functional analysis.

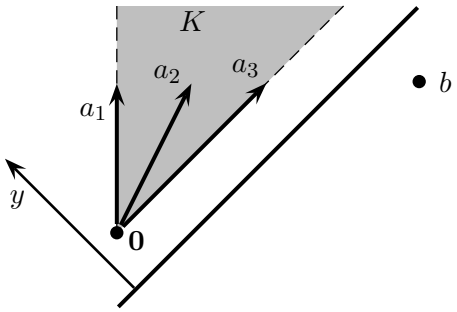
The separation theorem quickly provides us the very useful **Farkas Lemma** which is like a duality theorem without objective function. The lemma tells us that out of two particular linear systems, precisely one is going to be feasible.

Lemma 31 (Farkas Lemma)**.** *One has* $\big(\exists x \geq \mathbf{0} : Ax = b\big) \quad \dot\vee \quad \big(\exists y : y^T A \geq \mathbf{0} \text{ and } yb < 0\big)$.

Proof. First let us check that it is impossible that such x, y exist at the same time since

$$0 \leq \underbrace{yA}_{\geq \mathbf{0}} \underbrace{x}_{\geq \mathbf{0}} = y \underbrace{b}_{=Ax} < 0$$

For the other direction, assume that there is no $x \geq \mathbf{0}$ with $Ax = b$. We need to show that there is a proper y. Consider the cone $K := \{Ax \mid x \geq \mathbf{0}\} = \{\sum_{i=1}^n a_i x_i \mid x_1, \ldots, x_n \geq 0\}$ of valid right hand sides, where $a_1, \ldots, a_n$ are the columns of A. By assumption we know that $b \notin K$.



But K is a closed convex set, hence there is a hyperplane $y^T c = \gamma$ that separates K and $\{b\}$, i.e.

$$\forall a \in K : \quad y^T a > \gamma > y^T b$$

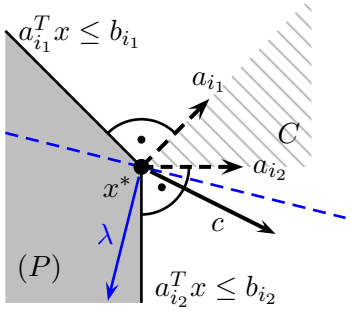
As $\mathbf{0} \in K$ we must have $\gamma < 0$. Moreover all non-negative multiples of columns are in K, that means $a_i x_i \in K$ for all $x_i \geq 0$, thus $y^T(x_i)a_i > \gamma$ for each $i \in [n]$, which implies that even $y^T a_i \geq 0$ for each i. This can be be written more compactly as $y^T A \geq \mathbf{0}$. □

Theorem 32. *One has $(P) = (D)$. More precisely, one has*

$$\max\{cx : Ax \leq b\} = \min\{by : yA = c,\ y \geq \mathbf{0}\}$$

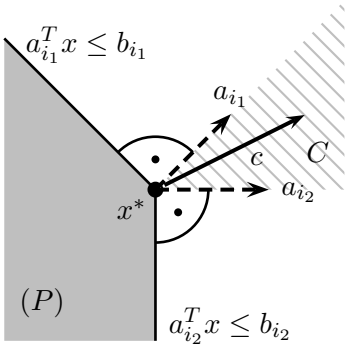
given that both systems have a solution.

Proof sketch. Suppose that (P) is feasible. Let x^* be an optimum solution[1]. Let $a_1, \ldots, a_m$ be rows of A and let $I := \{i \mid a_i^T x^* = b_i\}$ be the *tight* inequalities.



Suppose for the sake of contradiction that $c \notin \{\sum_{i \in I} a_i y_i \mid y_i \geq 0\ \forall i \in I\} =: C$. Then there is a direction $\lambda \in \mathbb{R}^n$ with $c^T\lambda > 0$, $a_i^T\lambda \leq 0\ \forall i \in I$. Walking in direction λ improves the objective function while we do not walk into the direction of constraints that are already tight. In other words, there is a small enough $\varepsilon > 0$ so that $x^* + \varepsilon\lambda$ is feasible for (P) and has a better objective function value — but x^* was optimal. That is a *contradiction!*

Hence we know that indeed $c \in C$, hence there is a $y \geq \mathbf{0}$ with $y^T A = c^T$ and $y_i = 0\ \forall i \notin I$ (that means we only use tight inequalities in y).



Now we can calculate, that the **duality gap** is 0:

$$y^T b - c^T x^* = y^T b - \underbrace{y^T A}_{=c} x^* = y^T(b - Ax^*) = \sum_{i=1}^{m} \underbrace{y_i}_{=0 \text{ if } i \notin I} \cdot \underbrace{(b_i - a_i^T x^*)}_{=0 \text{ if } i \in I} = 0$$

□

Note that moreover, if (P) is unbounded, then (D) is infeasible. If (D) is unbounded then (P) is infeasible. On the other hand, it is possible that (P) and (D) are both infeasible.

[1]That's why this is only a proof sketch. For a polytope it is easy to argue that P is compact and hence there must be an optimum solution. If P is unbounded, but the objective function value is bounded, then one needs more technical arguments that we skip here.

Theorem 33 (Complementary slackness)**.** *Let (x^*, y^*) be feasible solutions for*

$$(P):\ \max\{c^T x \mid Ax \le b\} \quad \text{and} \quad (D):\ \min\{by \mid y^T A = c; y \ge \mathbf{0}\}$$

Then (x^*, y^*) are both optimal if and only if

$$(A_i x^* = b_i \vee y_i^* = 0) \quad \forall i$$

Proof. We did already prove this in the last line of the duality theorem! □

5.2 Algorithms for linear programming

In this section, we want to briefly discuss the different methods that are known to solve linear programs.

5.2.1 The simplex method

The oldest and most popular one is the **simplex method**. Suppose the linear program is of the form

$$\max\{cx \mid Ax \le b\}.$$

We may assume that the underlying polyhedron $P = \{x \in \mathbb{R}^n \mid Ax \le b\}$ has vertices[2]. For a set of row indices $I \subseteq [m]$, let A_I be the submatrix of A that contains all the rows with index in I. A compact way of stating the simplex algorithm is as follows:

Simplex algorithm

Input: $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$ and a starting basis $I \in \binom{[m]}{n}$

Output: opt. solution x attaining $\max\{cx \mid Ax \le b\}$

(1) $x = A_I^{-1} b_I$

(2) IF $y := cA_I^{-1} \ge \mathbf{0}$ THEN RETURN x is optimal

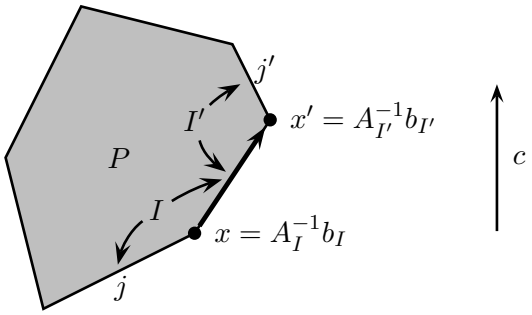
(3) select $j \in I$ and $j' \notin I$ so that for $I' := I \backslash \{j\} \cup \{j'\}$ the following 3 conditions are satisfied

  (i) $\operatorname{rank}(A_{I'}) = n$

  (ii) the point $x' = A_{I'}^{-1} b_{I'}$ lies in P

  (iii) $cx' \ge cx$

(4) UPDATE $I := I'$ and GOTO (1)



one simplex step

The maintained set I of indices is usually called a **basis** and the maintained solution x is always a **vertex** of P. In other words, the simplex method moves from one vertex to the next one so that the

[2]This is equivalent to saying that the $\ker(A) = \{\mathbf{0}\}$. A simple way to obtain this property is to substitute a variable $x_i \in \mathbb{R}$ by $x_i = x_i^+ - x_i^-$ and adding the constraints $x_i^+, x_i^- \ge 0$ to the constraint system. Now the kernel of the constraint matrix is empty.

objective function always improves (or at least does not get worse). If the condition $y = cA_I^{-1} \geq \mathbf{0}$ in step (3) is satisfied, then we have a dual solution $y \geq \mathbf{0}$ with $c = yA_I$ that uses only constraints that are tight for x; from our section on duality we know that then x must be an optimal solution.

One might object that the algorithm needs a basis I so that $x = A_I^{-1}x$ is a feasible solution. But one can also find a feasible solution using a linear program. For example, the linear program $\min\{\lambda \mid Ax \leq b + \lambda\mathbf{1}\}$ has a feasible solution $(\mathbf{0}, \lambda)$ for λ large enough and an optimum solution will be feasible for $Ax \leq b$ (given that there is any feasible solution).

Typically the simplex method is stated in a **tableau form** from which can easily determine for which choice of j one can pick a j' so that the conditions (i),(ii),(iii) are satisfied. Also, the tableau form uses that the inverted matrix $A_{I'}^{-1}$ can be computed with only $O(n^2)$ operations since A_I^{-1} is already known (in comparison, computing $A_{I'}^{-1}$ from scratch takes $O(n^3)$ if using Gauss elimination and $O(n^{2.373})$ if more advanced techniques are used).

Unfortunately the simplex algorithm is still not a polynomial time algorithm because the number of iterations may be exponentially large for some instances.

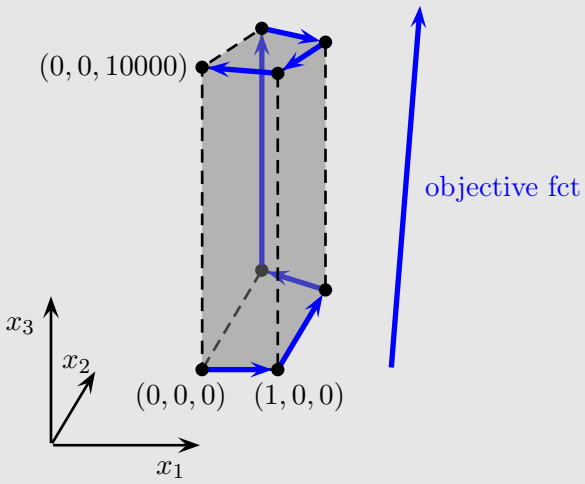
Advanced remark:

The first known pathological instance for the simplex method was found by Klee and Minty in 1973 (as a side remark, Victor Klee was on the UW faculty for many years). If one solves the linear program

$$\max\left\{\sum_{j=1}^{n} 10^{n-j}x_j \mid \left(2\sum_{j=1}^{i-1} 10^{i-j}x_j\right) + x_i \leq 100^{i-1}\ \forall i \in [n];\ \ x_j \geq 0\ \forall j \in [n]\right\}$$

with the **largest coefficient rule** starting at $x = (0, \ldots, 0)$, then the simplex will take $2^n - 1$ iterations. The underlying polytope P of this LP is a **skewed hypercube** with 2^n vertices and the simplex algorithm would go through all of them. We will not give the proof here, but want to give a visualization for $n = 3$. In that case, the LP and the polytope looks as follows:

$$\begin{aligned} \max\, & 100x_1 + 10x_2 + x_3 & & \\ & x_1 & \leq & 1 \\ & 20x_1 + x_2 & \leq & 100 \\ & 200x_1 + 20x_2 + x_3 & \leq & 10000 \\ & x_1, x_2, x_3 & \geq & 0 \end{aligned}$$



The bold arrows denote the path that is taken by the simplex algorithm if the largest coefficient pivoting rule is used. Note the the axes are skewed.

Despite the above result, the simplex method works extremely well in practice. Thus the measure

of complexity we are studying has its limitations and may not give a good feel for how an algorithm behaves on the average (on most problems). It is a **worst case** model that computes the complexity of an algorithm by considering its performance on all instances of a fixed size. A few pathological examples can skew the running time function considerably. Yet, we still get a pretty good indication of the efficiency of an algorithm using this idea.

It is a major open question in discrete optimization whether there is some pivot rule for which the simplex method runs in time polynomial in the input size.

5.2.2 Interior point methods and the Ellipsoid method

On the other hand, linear programs can be solved in polynomial time using algorithms such as *interior point methods* and the *ellipsoid method*. Both these algorithms were discovered around 1980 and use non-linear mathematics. They are quite complicated to explain compared to the simplex method. We can compare their characterisics as follows

	practical performance	worst case running time	works for
Simplex	very fast	exponential	LPs
Interiour Point	fast	$O(n^{3.5}L)$ arithmetic operations	LPs+SDPs
Ellipsoid method	slow	$O(n^6L)$ arithmetic operations	any convex set

Here L is the number of bits necessary to represent the input. For example we can write the number 5 as bit sequence 101 (since $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 6$). Note that in general it takes $1 + \lceil \log_2(|a|) + 1 \rceil \approx \log_2 |a|$ bits to encode an integer number a (we use the extra 1 to encode the sign of a and $\lceil \log_2(|a|) + 1 \rceil$ to encode $|a|$).

Both algorithms, the Interior point method and the Ellipsoid method need more time if the numbers in the input are larger, which is a contrast to the other algorithms that we encountered so far. It is a major open problem whether there is an algorithm for solving linear problems, say of the form $\max\{cx \mid Ax \leq b\}$, so that that the running time is polynomial in the number of rows and columns of A.

5.2.3 The multiplicative weights update method

We want to present at least one algorithm in more detail for which we can proof something. That algorithm is called the *multiplicative weight update (MWU) method*. The underlying principle has been used as early as the 1950's under different names such as "Fictitious Play", the Winnow algorithm, the Perceptron algorithm or the Hedge algorithm. Interestingly, the first applications were not to solve linear programs but to solve game theoretic settings.

To keep the notation simple, we want to use that algorithm to find a feasible point in a polytope of the form

$$P = \{x \in [0,1]^n \mid Ax \leq b\}.$$

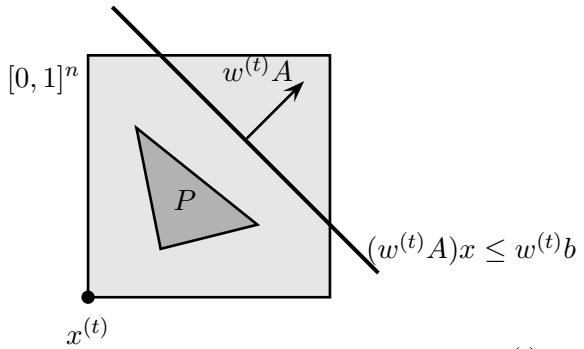
The algorithm will actually fail to do so, but it can find a point $x \in [0,1]^n$ so that $A_i x \leq b_i + \varepsilon$ for all i, where $\varepsilon > 0$ is an error parameter. We can choose ε arbitrary, but the algorithm will take longer the smaller ε is. Also, without loss of generality, we will scale the matrix so that for any row i, we have $\sum_{j=1}^n |A_{ij}| + |b_i| \leq 1$.

MULTIPLICATIVE WEIGHTS UPDATE ALGORITHM FOR LPS

Input: A system $Ax \leq b$ with $\sum_{j=1}^{n} |A_{ij}| + |b_i| \leq 1$ and a parameter $0 < \varepsilon \leq \frac{1}{3}$
Output: A point in $x \in [0,1]^n$ with $A_i x \leq b_i + \varepsilon$ for all i or a proof that $P = \emptyset$

(1) Set weights $w_i^{(0)} := 1$ for all constraints $i \in \{1, \ldots, m\}$
(2) FOR $t = 0, \ldots, T-1$ DO
  (3) Find a point $x^{(t)} \in [0,1]^n$ so that $(w^{(t)}A)x^{(t)} \leq w^{(t)}b$
  [If we can't find one, then $P = \emptyset$]
  (4) Set the **loss** $m_i^{(t)} := (A_i x^{(t)} - b_i) \in [-1, 1]$ as the amount that $x^{(t)}$ violates constraint i
  (5) Update the weight as $w_i^{(t+1)} := w_i^{(t)} \cdot (1 + \varepsilon \cdot m_i^{(t)})$
(6) Return the average $\bar{x} := \frac{1}{T} \sum_{t=0}^{T-1} x^{(t)}$

The basic idea behind the algorithm is to maintain **weights** $w_i^{(t)}$ for all constraints. The weights will always be non-negative, so at any step t, $(w^{(t)}A)x \leq w^{(t)}b$ is a valid inequality for P. In one iteration t, the situation will look as follows:



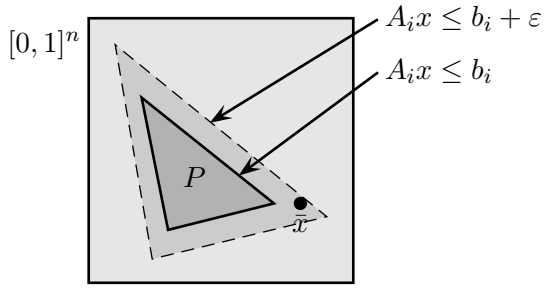
We will find a point $x^{(t)}$ that lies on the "right" side of the inequality $(w^{(t)}A)x \leq w^{(t)}b$. In fact, if we abbreviate $c := w^{(t)}A$, then it is easy to find the point $x^{(t)}$ in the cube $[0,1]^n$ that minimizes $cx^{(t)}$: just take

$$x_i^{(t)} := \begin{cases} 1 & \text{if } c_i \leq 0 \\ 0 & \text{if } c_i > 0 \end{cases}$$

If even that best point $x^{(t)}$ has $(w^{(t)}A)x^{(t)} > w^{(t)}b$, then we have an inequality separating all points in $\{x \mid Ax \leq b\}$ from $[0,1]^n$, hence $P = \emptyset$.

It will also be helpful to make the following observation: if the update point $x^{(t)}$ violates constraint i, then $m_i = A_i x^{(t)} - b_i > 0$, hence we will **increase** the weight of that constraint i in the weight update step. Moreover, if the points $x^{(1)}, \ldots, x^{(t)}$ up to some iteration would all violated the constraint i a lot, then the weight $w_i^{(t)}$ had grown dramatically, forcing the algorithm in the next step to satisfy the constraint. One might think of the $w_i^{(t)}$'s as penalties and it is very costly for the algorithm in step $t+1$ to violate a constraint that has a high penalty. That's somewhat the intuition why the algorithm works. Here is the formal analysis to prove that the point $\bar{x}$ at the end is at least approximately feasible.

Theorem 34. *After $T = \frac{8 \ln(m)}{\varepsilon^2}$ iterations, the MWU algorithm finds a point $\bar{x} \in [0,1]^n$ with $A_i \bar{x} \leq b_i + \varepsilon$ for all $i = 1, \ldots, m$.*



Proof. We consider the **potential function** $\Phi^{(t)} := \sum_{i=1}^{m} w_i^{(t)}$ that gives the sum of all the weights in iteration t. The first observation is that the algorithm always selects update points $x^{(t)}$ so that the potential function does not increase. Namely, by plugging in the definition of the weight update, we see that

$$\Phi^{(t+1)} = \sum_{i=1}^{m} w_i^{(t+1)} = \sum_{i=1}^{m} w_i^{(t)} \cdot \left(1 + \varepsilon \cdot m_i^{(t)}\right) = \underbrace{\sum_{i=1}^{m} w_i^{(t)}}_{=\Phi^{(t)}} + \varepsilon \underbrace{\sum_{i=1}^{m} w_i^{(t)} \cdot (A_i x^{(t)} - b_i)}_{\leq 0} \leq \Phi^{(t)}$$

Here we crucially used that we can always find a point $x^{(t)}$ that satisfies the single constraint $(w^{(t)}A)x \leq w^{(t)}b$ so that the weighted sum of the losses is non-positive. Now we can apply induction and get $\Phi^{(T)} \leq \Phi^{(0)} = m$.

Next, suppose for the sake of contradiction that there is a constraint i with $A_i\bar{x} > b_i + \varepsilon$. This is equivalent to $\sum_{t=1}^{T} m_i^{(t)} > \varepsilon T$. We claim that this causes the weight of the single constraint i to shoot through the roof:

$$\Phi^{(T)} \geq w_i^{(T)} = \prod_{t=1}^{T}(1 + \varepsilon m_i^{(t)}) \overset{1+x \geq e^{x-0.75x^2} \ \forall |x| \leq \frac{1}{3}}{\geq} \exp\left(\varepsilon \underbrace{\sum_{t=1}^{T} m_i^{(t)}}_{\geq \varepsilon T} - \frac{3}{4}\varepsilon^2 \sum_{t=1}^{T} \underbrace{(m_i^{(t)})^2}_{\leq 1}\right) \geq \exp\left(\frac{\varepsilon^2}{4} T\right)$$

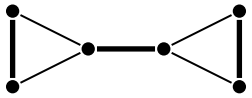
Now, if we choose $T := \frac{8}{\varepsilon^2}\ln(m)$, then $\Phi^{(T)} \geq \exp(2\ln(m)) = m^2 > m$ which is a contradiction. □

5.3 Connection to discrete optimization

Now that we have seen that there are very efficient algorithms for linear programming, we wonder whether we could use those to solve our discrete optimization problems. We want to motivate that approach using a case study with the **Perfect Matching Problem**:

> Max Weight Perfect Matching
> **Input:** Undirected graph $G = (V, E)$, weights $c_{ij} \geq 0$
> **Goal:** A perfect matching $M \subseteq E$ maximizing $\sum_{e \in M} c_e$.

Recall that a perfect matching $M \subseteq E$ is a set of edges that has degree exactly 1 for each node in the graph. For example in the instance below we mark the only perfect matching in the graph:



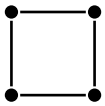
Obviously not all graphs have perfect matchings. We try a bit naively to set up a linear program that is supposed to solve our matching problem. It appears to be natural be have a variable $x_e \in [0,1]$ that tells us whether or not the edge e should be contained in our matching. A perfect matching is defined by requiring that the degree of a node is 1.

$$\begin{aligned} \max & \sum_{e \in E} c_e x_e \\ \sum_{e \in \delta(v)} x_e &= 1 \quad \forall v \in V \\ x_e &\geq 0 \quad \forall e \in E \end{aligned}$$

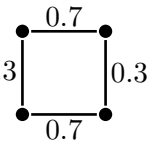
Observe that for each matching M, the vector $x \in \mathbb{R}^E$ with

$$x_e = \begin{cases} 1 & e \in M \\ 0 & e \notin M \end{cases}$$

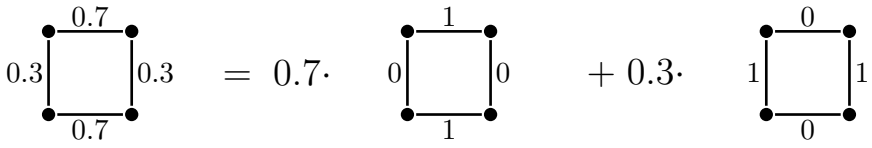
is a feasible LP solution. We want to try out our LP with a small example instance. Already finding a perfect matching itself is not a trivial problem, hence let us consider the following instance with $c_e = 0$ for each edge:



We run an LP solver and it might return a solution like this one:

$x_e =$ 

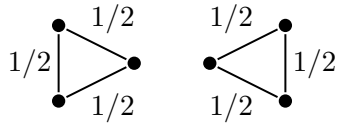
The graph indeed has perfect matchings, but the fractional numbers on the edges are not very useful. Observe that the point $x = (0.3, 0.3, 0.7, 0.7)$ is not an extreme point of the underlying polyhedron. This can be easily seen because we can write it as a convex combination of two feasible LP solutions:



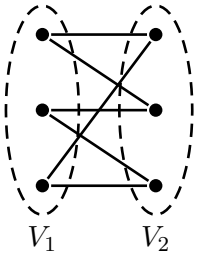
But we know that the simplex algorithm would always return us an extreme point. In fact, also the interior point method and the ellipsoid method can compute an optimum extreme point solution in polynomial time (given that the polyhedron has extreme points). So, what happens if we always compute an extreme point?

For example for the double triangle graph

there is no perfect matching, yet the linear program has the extreme point solution



So, the whole approach does not look very promising for matching. But it turns out that for graphs $G = (V, E)$ that are **bipartite**, this approach does work. Note that we call a graph $G = (V, E)$ bipartite, if the node set V can be partitioned into $V = V_1 \dot\cup V_2$ so that all edges are running between V_1 and V_2.



bipartite graph

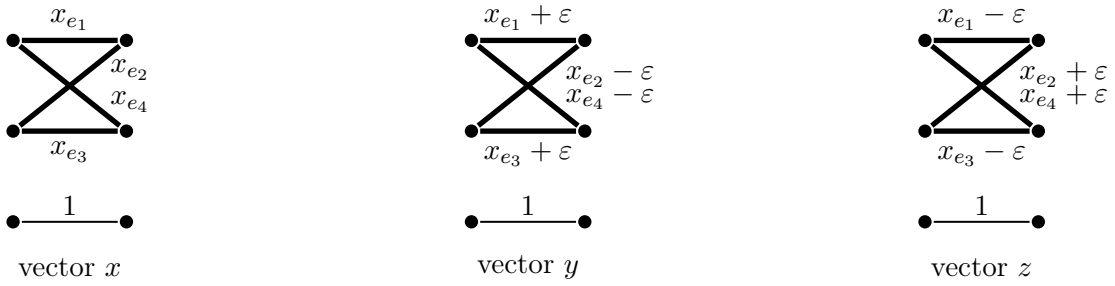
We want to remark that bipartite graphs already cover a large fraction of applications for matching.

Lemma 35. *Let $G = (V, E)$ be a bipartite graph and x be an optimum extreme point solution to the matching LP. Then x is integral and $M = \{e \in E \mid x_e = 1\}$ is an optimum perfect matching.*

Proof. Suppose for the sake of contradiction that x is an extreme point, but it is not completely integral. We will lead this to a contradiction by finding two other feasible LP solutions y and z with $x = \frac{1}{2}(x + z)$.

Let $E_f = \{e \in E \mid 0 < x_e < 1\}$ be the fractional edges and let $V_f = \{v \in V \mid |\delta(v) \cap E_f| > 0\}$ be the nodes that are incident to fractional edges. Observe that if a node v is incident to one fractional edge $\{u, v\}$, then it must be incident to at least another fractional edge (since $1 - x_{uv} \in]0, 1[$). In other words, in the subgraph (V_f, E_f), each node has degree at least 2. Each such graph must contain at least one circuit $C \subseteq E_f$ (simply start an arbitrary walk through E_f until you hit a node that you have previously visited; this will close a circuit). Since the graph is bipartite, any cycle must have even length. Denote the edges of the circuit with $e_1, \dots, e_{2k}$ and let $\varepsilon := \min\{x_e \mid e \in C\} > 0$. Now we define $y \in \mathbb{R}^E$ and $z \in \mathbb{R}^E$ by

$$z_e = \begin{cases} x_{e_i} + \varepsilon & e = e_i \text{ and } i \text{ odd} \\ x_{e_i} - \varepsilon & e = e_i \text{ and } i \text{ even} \\ x_e & \text{if } e \notin C \end{cases} \qquad y_e = \begin{cases} x_{e_i} - \varepsilon & e = e_i \text{ and } i \text{ odd} \\ x_{e_i} + \varepsilon & e = e_i \text{ and } i \text{ even} \\ x_e & \text{if } e \notin C \end{cases}$$

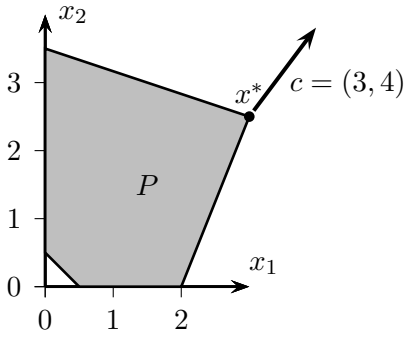


(the bold edges are in C). Then both y and z are feasible LP solutions since each node on the circuit C is in one odd edge and one even edge, the $\sum_{e\in\delta(v)} y_e = \sum_{e\in\delta(v)} x_e + \varepsilon - \varepsilon = 1$ (same for z). Also we have chosen $\varepsilon > 0$ small enough so that $y_e, z_e \geq 0$. Moreover $x = \frac{1}{2}(y + z)$, hence x is not an extreme point. The contradiction shows that x is indeed integral, that means $x_e \in \{0, 1\}$ for each edge $e \in E$. □

5.4 Integer programs and integer hull

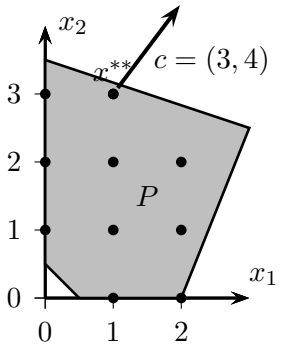
We want to better understand, what was going on with the linear program in the matching case and what it means to have integral vertices. For the sake of a better visualization, we consider a 2-dimensional linear program

$$\begin{aligned} \max\, & 3x_1 + 4x_2 & & \\ & 2x_1 + 6x_2 & \leq & \ 21 \\ & 6x_1 - x_2 & \leq & \ 12 \\ & x_1 + x_2 & \geq & \ \frac{1}{2} \\ & x_1, x_2 & \geq & \ 0 \end{aligned}$$



with optimum $x^* = (3, 2.5)$ and objective function value $cx^* = 3 \cdot 3 + 2.5 \cdot 4 = 19$. The **integer program** associated to the above linear program is

$$\begin{aligned} \max\, & 3x_1 + 4x_2 & & \\ & 2x_1 + 6x_2 & \leq & \ 21 \\ & 6x_1 - x_2 & \leq & \ 12 \\ & x_1 + x_2 & \geq & \ \frac{1}{2} \\ & x_1, x_2 & \geq & \ 0 \\ & x_1, x_2 & \in & \ \mathbb{Z} \end{aligned}$$



and the optimum solution is $x^{**} = (1, 3)$ with objective function value $cx^{**} = 15$. In general, for any polyhedron $P \subseteq \mathbb{R}^n$, the linear program $\max\{cx \mid x \in P\}$ is associated with the integer program $\max\{cx \mid x \in P \cap \mathbb{Z}^n\}$. The LP is also called an **LP relaxation** of its integer program. Obviously, the value of the linear program and the integer program can be very different. It is also not difficult to construct an example where the linear program is feasible, while the integer program is infeasible. Note that always

$$\max\{cx \mid x \in P\} \geq \max\{cx \mid x \in P \cap \mathbb{Z}^n\}$$

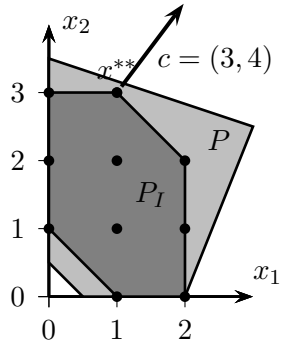
(given that both systems have a finite value), simply because the left hand maximum is over a larger set. Also, we want to remark that the same integer program could have several different LP relaxations.

Definition 9. For a polyhedron $P \subseteq \mathbb{R}^n$, we call

$$P_I = \text{conv}(P \cap \mathbb{Z}^n)$$

the **integer hull**.

Phrased differently, the integer hull P_I is the smallest polyhedron that contains all the integral points in P. That means P_I is the "perfect" relaxation for the integer program. The integer hull is particularly useful since optimizing any function over it gives the same value as the integer program.

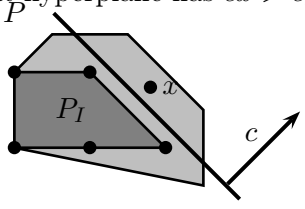


Lemma 36. *Let $P \subseteq \mathbb{R}^n$ be a polytope. Then the following conditions are equivalent*

(a) $P = P_I$

(b) For each $c \in \mathbb{R}^n$, $\max\{cx \mid x \in P\} = \max\{cx \mid x \in P \cap \mathbb{Z}^n\}$

Proof. For $(a) \Rightarrow (b)$, just observe that a linear function over $\text{conv}(P \cap \mathbb{Z}^n)$ is maximized at one of the "spanning" points, i.e. for $x \in P \cap \mathbb{Z}^n$. For the 2nd direction we show that $\neg(a) \Rightarrow \neg(b)$. In words, we assume that $P \neq P_I$. Let $x \in P \backslash P_I$. Recall that P_I is a convex set, hence there is a hyperplane separating x from P_I hand that hyperplane has $cx > cy \; \forall y \in P_I$ which gives the claim.



□

The claim is actually true for unbounded polyhedra.

What we ideally would like to solve are integer programs. The reason is that those are powerful enough to model every discrete optimization problem that we will encounter in this lecture. For example, the max weight perfect matching problem can be stated as

$$\max \Big\{ \sum_{e \in E} c_e x_e \mid \sum_{e \in \delta(v)} x_e = 1 \;\; \forall v \in V; \;\; x_e \geq 0 \; \forall e \in E \quad x_e \in \mathbb{Z} \; \forall e \in E \Big\}$$

(in arbitrary graphs). The issue is that integer programming is **NP**-hard, which means that assuming $\mathbf{NP} \neq \mathbf{P}$, there is no polynomial time algorithm for such problems in general. Instead we want to solve the underlying LP relaxation; for many problems we will be able to argue that $P = P_I$ and hence a computed extreme point is always integral.

Chapter 6

Total unimodularity

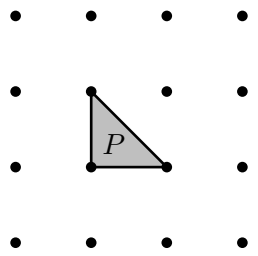
In this chapter we want to investigate, which type of integer programs

$$\max\{cx \mid Ax \leq b;\ x \in \mathbb{Z}^n\}$$

we can solve in polynomial time simply by computing an extreme point solution to the corresponding linear relaxation

$$\max\{cx \mid Ax \leq b\}.$$

In other words, we wonder when all the extreme points of $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ are integral like in this figure:



Remember the following fact from a previous Chapter:

Lemma 37. *Let $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$ be a polyhedron with $A \in \mathbb{R}^{m \times n}$. For each extreme point x of P, there are row indices $I \subseteq \{1, \ldots, m\}$ so that A_I is an $n \times n$ matrix of full rank and x is the unique solution to $A_I x = b_I$.*

Here A_I denotes the rows of A that have their index in I. We wonder: when can we guarantee that the solution to $A_I x = b_I$ is integral? Recall the following lemma from linear algebra:

Lemma 38 (Cramer's rule)**.** *Let $B = (b_{ij})_{i,j \in \{1,\ldots,n\}}$ be an $n \times n$ square matrix of full rank. Then the system $Bx = b$ has exactly one solution x^* where*

$$x_i^* = \frac{\det(B^i)}{\det(B)}$$

Here B^i is the matrix B where the ith column is replaced by b.

Cramer's rule suggests to ask for the following property:

Definition 10. A matrix A is **totally unimodular (TU)** if every square submatrix of A has determinant 0,1 or -1.

Note that A itself does not have to be square. We just need all the square submatrices of A to have $0, \pm 1$ determinant. However, if A is totally unimodular, then every entry of A has to be $0, \pm 1$ since every entry is a 1×1 submatrix of A.

Lemma 39. *Let $P = \{x \in \mathbb{R}^n \mid Ax \leq b\}$. If A is TU and $b \in \mathbb{Z}^m$, then all extreme points of P are integral.*

Proof. Let x be an extreme point of P and let A_I be the submatrix so that x is the unique solution to $A_I x = b_I$. Then by Cramers rule

$$x_i = \frac{\det(A_I^i)}{\det(A_I)}$$

Since A is TU we know that $\det(A_I) \in \{\pm 1\}$ (the case $\det(A_I) = 0$ is not possible since A_I has full rank). Since A and b are integral, A_I^i has only integral entries, and hence $\det(A_I^i) \in \mathbb{Z}$. This shows that $x_i \in \mathbb{Z}$. □

The TU property is somewhat robust under changes as we will see in the following:

Lemma 40. *Let $A \in \{-1, 0, 1\}^{m \times n}$. Then the following operations do not change whether or not A is TU:*

a) multiplying a row/column with -1

b) Permuting rows/columns

c) Adding/removing a row/column that has at most one ± 1 entry and zeros otherwise

d) Duplicating rows

e) Transposing the matrix

Proof. We can study the effect of those operations on a square submatrix $B = (b_{ij})_{i,j \in \{1,\dots,n\}}$ of A. The claims follow quickly from linear algebra facts such as (I) $\det(B) = \det(B^T)$; (II) multiplying a row/column by λ changes the determinant by λ; (III) permuting rows/columns only changes the sign of det. For (c), we can use the **Laplace formula**: Let M_{ij} be the $(n-1) \times (n-1)$ matrix that we obtain by removing the ith row and jth column from B. Then for all $i \in \{1, \dots, n\}$,

$$\det(B) = \sum_{j=1}^{n} (-1)^{i+j} \cdot b_{ij} \cdot \det(M_{ij}).$$

□

We get:

Corollary 41. *Let A be a TU matrix with $b \in \mathbb{Z}^m$ and $\ell, u \in \mathbb{Z}^{n \times n}$. Then*

a) All extreme points of $\{x \in \mathbb{R}^n \mid Ax \leq b; x \geq \mathbf{0}\}$ are integral.

b) All extreme points of $\{x \in \mathbb{R}^n \mid Ax = b; x \geq \mathbf{0}\}$ *are integral.*

b) All extreme points of $\{x \in \mathbb{R}^n \mid Ax \leq b; \ell \leq x \leq u\}$ *are integral.*

Proof. From Lemma 40 we know that also the matrices

$$\begin{pmatrix} A \\ -I \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} A \\ -A \\ -I \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} A \\ -I \\ I \end{pmatrix}$$

that define the 3 systems are TU. Then all the claims follow. □

As a technical subtelty, remember that not all polyhedra have extreme points and the statement of Lemma 39 would be useless in such a case. But remember from 407, that all polyhedra that have non-negativity constraints, must have extreme points.

Verifying that all submatrices of A have determinant in $\{0, \pm 1\}$ sounds like a very tedious and complicated approach. Luckily, there is a sufficient condition that is very easy to check and that suffices in many applications:

Lemma 42. *A matrix* $A \in \{-1, 0, 1\}^{m \times n}$ *is TU if it can be written in the form*

$$A = \left(B \;\middle|\; \begin{matrix} C_1 \\ \hline C_2 \end{matrix} \right)$$

so that the the following conditions hold:

- *B has at most one non-zero entry per column*
- *C_1 has exactly one $+1$ entry per column*
- *C_2 has exactly one -1 entry per column*

Proof. Suppose for the sake of contradiction that the claim is false and consider a minimal counterexample A. Then A itself must be the square submatrix and we need to check that $\det(A) \notin \{-1, 0, 1\}$, while we can assume that this is the case for each proper submatrix. If some column has only one non-zero entry, then we can apply Laplace Formula to that column and can write $\det(A) = (\pm 1) \cdot \det(A') \in \{\pm 1\}$ since A is already a minimal counterexample. In fact, that means that A is of the form $A = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}$. But then, adding up all rows of A gives the $\mathbf{0}$-vector, i.e. $\det(A) = 0$. □

An example matrix for which this lemma proves total unimodularity is the following:

$$A = \left(\begin{array}{ccc|cccc} & B & & & C_1 & & \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & -1 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 & -1 & 0 \\ & & & & C_2 & & \end{array} \right)$$

Before we show some applications, we want to mention one more insight. Remember that going to the **dual LP** essentially means we transpose the constraint matrix. Since transposing a matrix does not destroy the TU property we immediately get:

Lemma 43. *Suppose A is TU and both b and c are integral. Then there are optimum integral solutions x^*, y^* to*

$$\max\{cx \mid Ax \leq b; x \geq \mathbf{0}\} \quad \textit{and} \quad \min\{by \mid A^T y \geq c; y \geq \mathbf{0}\}$$

with $cx^* = by^*$.

A very surprising fact is the following:

Theorem 44 (Seymour '80, Truemper '82)**.** *For a matrix $A \in \{0, \pm 1\}^{m \times n}$ one can test in polynomial time whether A is TU.*

We will generously skip the proof of this 50+ page fact.

6.1 Application to bipartite matching

Now, let $G = (V, E)$ be an undirected graph and consider the **matching polytope**

$$P_M = \Big\{x \in \mathbb{R}^E \mid \sum_{e \in \delta(v)} x_e = 1 \; \forall v \in V; x_e \geq 0 \; \forall e \in E\Big\}.$$

We already know from Section 5.3 that for bipartite graphs all extreme points of P_M are integral, while this is not necessarily the case for non-bipartite graphs. Now we want to understand why! We can now reprove Lemma 35:

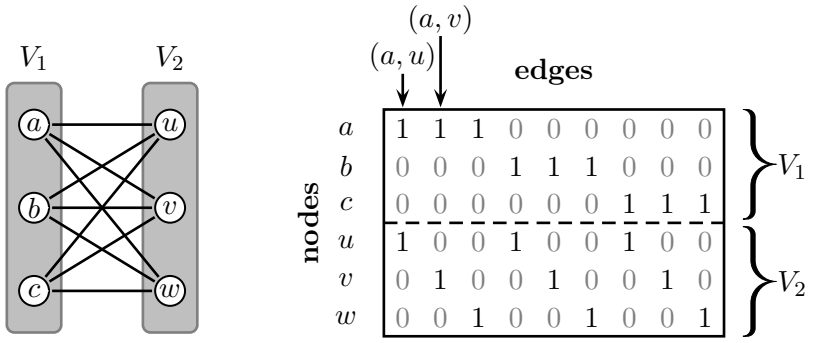
Lemma 45. *If G is bipartite, then all extreme points x of P_M satisfy $x \in \{0, 1\}^E$.*

Proof. We can write $P_M = \{x \in \mathbb{R}^E \mid Ax = \mathbf{1}; x \geq \mathbf{0}\}$ where $A \in \{0, 1\}^{V \times E}$ is the **node-edge incidence matrix** of graph G that is defined by

$$A_{v,e} = \begin{cases} 1 & v \text{ incident to } e \\ 0 & \text{otherwise} \end{cases}$$

We claim that A is TU. Since G is bipartite, the matrix has rows that belong to nodes in V_1 and rows that belong to V_2. A column belonging to an edge (u, v) has one 1 in the V_1-part and the other 1 in the V_2-part. Now flip the signs in the V_2-part, then Lemma 42 applies and shows that A is TU. □

An example for a node edge incidence matrix can be found below:



graph G node edge incidence matrix A

We already showed one direction of the following consequence — we will not show the other direction.

Lemma 46. *The incidence matrix of an undirected graph G is TU if and only if G is bipartite.*

6.2 Application to flows

In this section, we want to apply our knowledge of totally unimodularity to **flows**. Let (G, u, s, t) be a **network** with a directed graph G, integral capacities $u : E \to \mathbb{Z}_{\geq 0}$ a source s and a sink t. For a number $k \in \mathbb{Z}_{\geq 0}$, the set of feasible s-t flows of value k can be described by the polytope

$$P_{k-\text{flow}} = \left\{ f \in \mathbb{R}^E \mid \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) = b(v); 0 \leq f(e) \leq u(e) \; \forall e \in E \right\}$$

with

$$b(v) = \begin{cases} -k & v = s \\ k & v = t \\ 0 & \text{otherwise.} \end{cases}$$

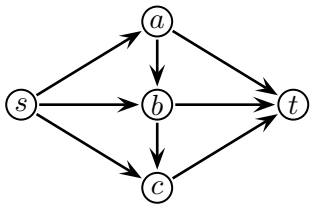
We want to have a closer look, how the constraint matrix of this polytope looks like. And in fact, if we write

$$P_{k-\text{flow}} = \{ f \in \mathbb{R}^E \mid Af = b; \; 0 \leq f \leq u \}$$

then the matrix $A \in \{-1, 0, 1\}^{V \times E}$ is defined by

$$A_{v,e} = \begin{cases} 1 & e \in \delta^-(v) \\ -1 & e \in \delta^+(v) \\ 0 & \text{otherwise.} \end{cases}$$

This matrix is also called the **node edge incidence matrix** of the directed graph G.



graph G

nodes \ edges	(s,a)	(s,b)	(s,c)	(a,b)	(b,c)	(a,t)	(b,t)	(c,t)
s	-1	-1	-1	0	0	0	0	0
a	1	0	0	-1	0	-1	0	0
b	0	1	0	1	-1	0	-1	0
c	0	0	1	0	1	0	0	-1
t	0	0	0	0	0	1	1	1

node edge incidence matrix A

Lemma 47. *The node edge incidence matrix $A \in \{-1, 0, 1\}^{V \times E}$ of a directed graph is TU.*

Proof. Again, look at the smallest square submatrix B of A that is a counterexample. Then each column must contain 2 non-zero entries, otherwise, we could obtain a smaller counterexample. But then each column must contain exactly one 1 and one -1 entry. Then summing up all rows of B gives $(0, \ldots, 0)$. That means the rows are not linearly independent and hence $\det(B) = 0$. □

Corollary 48. *For any integer k, the polytope $P_{k-\text{flow}}$ of value k s-t flows has only integral extreme points.*

Let us see how far we can push this method. The following is actually a very general problem.

Minimum Cost Integer Circulation Problem
Input: A directed graph $G = (V, E)$ with lower bounds $\ell(e) \in \mathbb{Z}_{\geq 0}$, upper bounds $u(e) \in \mathbb{Z}_{\geq 0}$, cost $c : E \to \mathbb{R}$
Goal: Solve

$$\min \Big\{ \sum_{e \in E} c(e) f(e) \mid f \text{ is circulation with } \ell(e) \leq f(e) \leq u(e) \text{ and } f(e) \in \mathbb{Z} \; \forall e \in E \Big\}$$

Note that one can easily use this problem to solve minimum cost matchings in bipartite graphs or minimum cost max flow problems.

Lemma 49. *Minimum Cost Integer Circulation Problem can be solved in polynomial time.*

Proof. Simply observe that the relaxation can be written as $\min\{c^T f \mid Af = \mathbf{0}; \ell \leq f \leq u\}$ where the constraint matrix is the node-edge incidence matrix, which is TU. Then we can solve the LP with the Interior point method to find an optimum extreme point f^*. This solution is going to be integral since A is TU. □

6.3 Application to interval scheduling

In this section we want to study the total unimodularity for a problem that is not related to graphs.

Interval scheduling
Input: A list of intervals $I_1, \ldots, I_n \subseteq \mathbb{R}$ with profits c_i for interval I_i.
Goal: Select a disjoint subset of intervals that maximize the sum of profits.

Note that there are at most $2n$ many points that are end points of intervals let us denote those points with $t_1 < t_2 < \ldots < t_m$. Then it suffices to check that we select intervals so that no point in $t_1, \ldots, t_m$ lies in more than one interval. Hence we can formulate the interval scheduling problem as the following integer program:

$$\max\Big\{cx \mid \sum_{i:j\in I_i} x_i \leq 1 \; \forall j \in \{1,\ldots,m\}; x_i \in \{0,1\} \; \forall i \in \{1,\ldots,n\}\Big\}$$

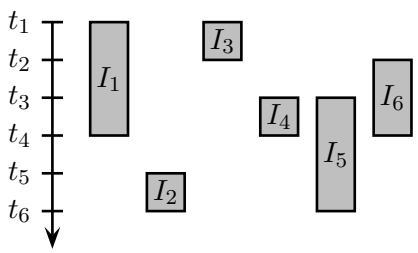
We can rewrite the relaxation of this problem in matrix form as

$$\max\{cx \mid Ax \leq \mathbf{1}; 0 \leq x \leq 1\} \quad \text{with} \quad A_{ji} = \begin{cases} 1 & t_j \in I_i \\ 0 & \text{otherwise} \end{cases}$$

Observe that the matrix $A \in \{0,1\}^{m\times n}$ has the **consecutive-ones property**, that means the ones in each column are consecutive.

Lemma 50. *A matrix $A \in \{0,1\}^{m\times n}$ with consecutive ones property is TU.*

Proof. Again, we consider a minimal counterexample. Since any submatrix of A also has the consecutive ones property, it suffices to show that $\det(A) \in \{-1,0,1\}$ assuming that A itself is a square matrix. Let $A_1, \ldots, A_n$ be the rows of A. For all i, in decreasing order, we subtract row $i-1$ from row i, which does not change the determinant. Formally, we define $A'_i := A_i - A_{i-1}$ and $A'_1 = A_1$, then $\det(A') = \det(A)$ (see the example below). Now each column in A' contains one 1 and one -1 (or only one 1 in case that the interval contains t_n). □



instance for interval scheduling

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

constraint matrix A

$$A' = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

transformed matrix A'

Theorem 51. *The interval scheduling problem can be solved in polynomial time.*

Proof. Follows, since the constraint matrix of the problem is TU. □

Chapter 7

Branch & Bound

We already saw many problems for which we can devise an efficient, polynomial time algorithm. On the other hand, we also discussed that for **NP**-complete problems like the Travelling Salesmen problem, it is widely believed that no polynomial time algorithm exists. While from a theoretical perspective we can be perfectly happy with the non-existence of an algorithm, this is not very satisfying for practitioners.

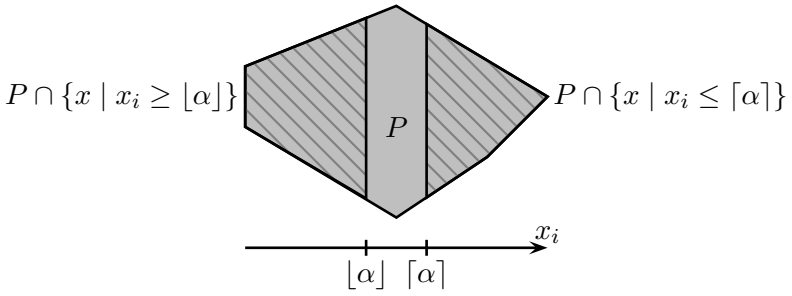
Suppose you in fact do have an instance of an **NP**-complete problem and you *really* need a solution! For all the problems that we discuss in this lecture and also the vast majority of problems that appear in practical applications, it is very straightforward to formulate it as an **integer linear program**

$$\max\{cx \mid Ax \leq b;\ x \in \mathbb{Z}^n\} \quad (*)$$

Suppose for a moment that we have the additional restriction that $x \in \{0,1\}^n$. Then the naive approach would just try out all 2^n solutions and then pick the best. A quick calculation shows that even for very moderate instance sizes of, say $n = 200$, we would obtain an astronomically high running time.

We wonder: is there an algorithm that solves the IP (*) in many cases much faster than 2^n? The answer is "yes"! The **Branch & Bound algorithm** is a much more intelligent search algorithm that usually avoids searching the whole solution space. See the figure on page 68 for the description of the algorithm and see the figure on page 69 for an example performance of the algorithm. We want to emphasize the three points that make the algorithm smarter than trivial enumeration for solving $\max\{cx \mid x \in P;\ x \in \mathbb{Z}^n\}$ with $P = \{x \mid Ax \leq b\}$. In the following $P' \subseteq P$ denotes a subproblem.

- **Insight 1:** Suppose $\alpha \notin \mathbb{Z}$ and $i \in \{1, \ldots, n\}$. Then all integer points in P are either in $P \cap \{x \mid x_i \leq \lfloor \alpha \rfloor\}$ or in $P \cap \{x \mid x_i \geq \lceil \alpha \rceil\}$.

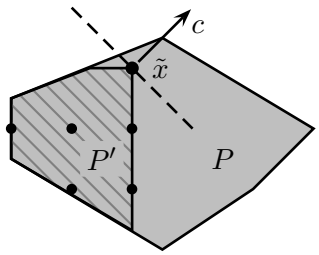


Branch & Bound algorithm

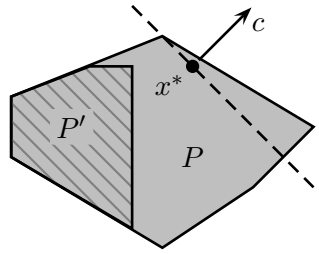
Input: $c \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$
Output: An optimum solution to $\max\{cx \mid Ax \leq b;\ x \in \mathbb{Z}^n\}$

(1) Set $x^* :=$ `UNDEFINED` *(best solution found so far)*
(2) Put problem $P := \{x \mid Ax \leq b\}$ on the stack
(3) WHILE stack is non-empty DO
  (4) Select a polytope P' from the stack
  (5) Solve $\max\{cx \mid x \in P'\}$ and denote LP solution by $\tilde{x}$
  (6) IF $P' = \emptyset$ THEN goto (3) *("prune by infeasibility")*
  (7) IF $cx^* \geq c\tilde{x}$ THEN goto (3) *("prune by bound")*
  (8) IF $cx^* < c\tilde{x}$ and $\tilde{x} \in \mathbb{Z}^n$ THEN (*let* $cx^* = -\infty$ *if* $x^* =$ `UNDEFINED`
    update $x^* := \tilde{x}$ and goto (3) *("prune by optimality")*
  (9) Otherwise (i.e. $\tilde{x} \notin \mathbb{Z}^n, cx^* < c\tilde{x}$) do *("branch")*
    (10) Select coordinate i with $\tilde{x}_i \notin \mathbb{Z}$.
    (11) Add problems $P' \cap \{x \mid x_i \leq \lfloor \tilde{x}_i \rfloor\}$ and $P' \cap \{x \mid x_i \geq \lceil \tilde{x}_i \rceil\}$ to stack

- **Insight 2:** If the optimal LP solution $\tilde{x}$ to $\max\{cx \mid x \in P'\}$ happens to be integral, then $\tilde{x}$ is the optimum integral point in P'.



- **Insight 3:** If there is a point $x^* \in P \cap \mathbb{Z}^n$ with $cx^* > \max\{cx \mid x \in P'\}$, then the optimum integral solution for the IP does not lie in P'.



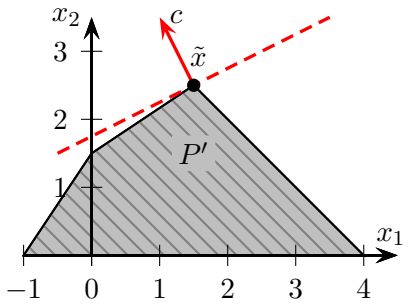
From the discussion above we obtain:

Theorem 52. *Branch and bound does find the optimum for* $\max\{cx \mid Ax \leq b;\ x \in \mathbb{Z}^n\}$.

Example: Branch & Bound. Instance: $\max\{-x_1 + 2x_2 \mid (x_1, x_2) \in P\}$
with $P := \{x \in \mathbb{R}^2 \mid -x_1 + 6x_2 \leq 9, x_1 + x_2 \leq 4, x_2 \geq 0, -3x_1 + 2x_2 \leq 3\}$

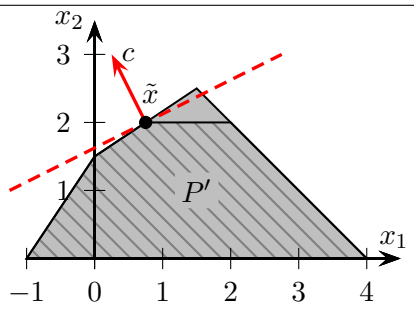
Iteration 1:

- $x^* =$ `UNDEFINED`
- Stack: $\{P\}$
- Select: $P' := P$
- LP opt: $\tilde{x} = (1.5, 2.5)$
- Case: Branch on $i = 2$.
  Add $P' \cap \{x_2 \leq 2\}, P' \cap \{x_2 \geq 3\}$ to stack



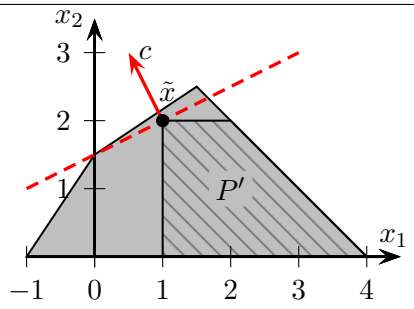
Iteration 2:

- $x^* =$ `UNDEFINED`
- Stack: $\{P \cap \{x_2 \leq 2\}, P \cap \{x_2 \geq 3\}\}$
- Select: $P' := P \cap \{x_2 \leq 2\}$
- LP opt: $\tilde{x} = (0.75, 2)$
- Case: Branch on $i = 1$. Add $P' \cap \{x_2 \leq 2, x_1 \geq 1\}, P' \cap \{x_2 \leq 2, x_1 \leq 0\}$ to stack



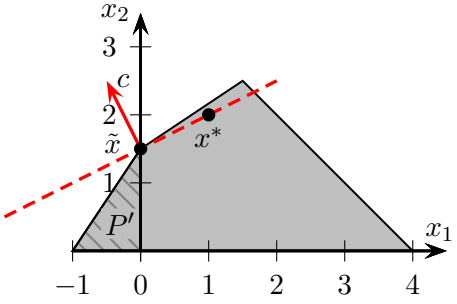
Iteration 3:

- $x^* =$ `UNDEFINED`
- Stack: $\{P \cap \{x_2 \leq 2, x_1 \geq 1\}, P \cap \{x_2 \leq 2, x_1 \leq 0\}, P \cap \{x_2 \geq 3\}\}$
- Select: $P' := P \cap \{x_2 \leq 2, x_1 \geq 1\}$
- LP opt: $\tilde{x} = (1, 2)$
- Case: Prune by optimality. Update $x^* := \tilde{x}$



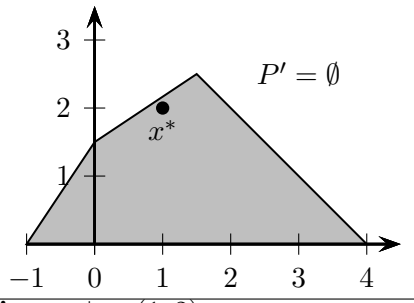
Iteration 4:

- $x^* = (1, 2)$
- Stack: $\{P \cap \{x_2 \leq 2, x_1 \leq 0\}, P \cap \{x_2 \geq 3\}\}$
- Select: $P' := P \cap \{x_2 \leq 2, x_1 \leq 0\}$
- LP opt: $\tilde{x} = (0, 1.5)$
- Case: Prune by bound ($c\tilde{x} \leq cx^*$).



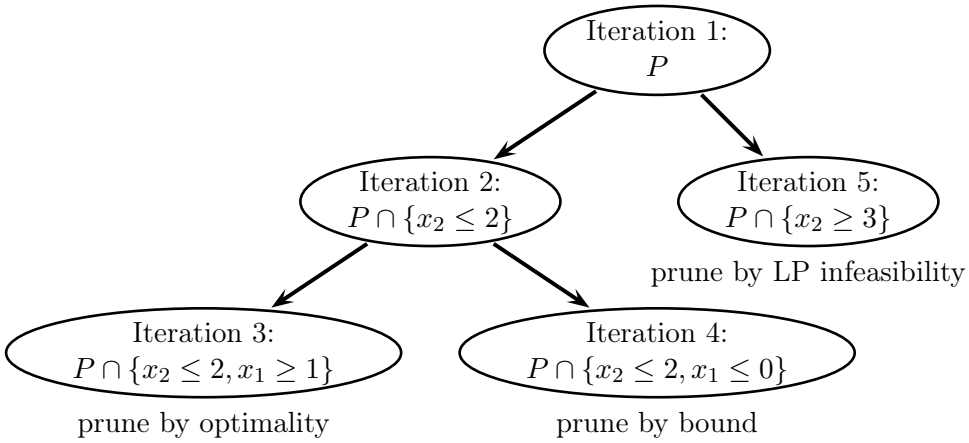
Iteration 5:

- $x^* = (1, 2)$
- Stack: $\{P \cap \{x_2 \geq 3\}\}$
- Select: $P' := P \cap \{x_2 \geq 3\}$
- LP opt: `UNDEFINED`
- Case: Prune by infeasibility ($P' = \emptyset$)



Iteration 6: Stack empty. Optimum solution: $x^* = (1, 2)$

Observe that the branch and bound process implicitly constructs a search tree where a node corresponds to a subproblem P'. For our example on page 69, the **Branch & Bound tree** looks as follows:



There are two steps in the algorithm that are "generic". More precisely, we did not specify which problem should be selected from the stack and which coordinate should be used to branch.

- *On which variable should one branch in (10)?*
  A typical strategy is to branch on the *most fractional variable*. That means that one selects the variable i that maximizes the distance $\min\{\tilde{x}_i - \lfloor \tilde{x}_i \rfloor, \lceil \tilde{x}_i \rceil - \tilde{x}_i\}$ to the nearest integer. For example in $\tilde{x} = (0.99, 0.5)$ one would rather branch on x_2 instead of x_1.

- *Which problem P' should be selected from the stack?* There are two popular strategies:

  - *Depth First Search:* Always selecting the last added problem from the stack corresponds to a depth first seach in the branch and bound tree. The hope is that this way, one quickly finds some feasible integral solution that then can be used to prune other parts of the tree.
  - *Best bound:* Select the problem that has the highest LP maximum. The hope is that there is also a good integral solution hiding.

Note that we cannot give any theoretical justification for or against one strategy. For any strategy one could cook up a pathological instance where the strategy misarably fails while another succeeds quickly.

7.1 A pathological instance

Each chapter should contain at least one proof — hence we want to show that there are nasty integer linear programs for which all Branch & Bound approaches fail.

Lemma 53. *Let $n \geq 4$ be an even integer. No matter what strategies are used by B & B to solve the following IP*

$$\max \left\{ x_0 \mid \frac{1}{2}x_0 + \sum_{i=1}^{n} x_i = \frac{n}{2};\ x \in \{0,1\}^{n+1} \right\}$$

the search tree has at least $2^{n/3}$ leafs.

Proof. Observe that the IP only has integral solutions with $x_0 = 0$. Hence the integral optimum is 0. Since $0 \leq x_i \leq 1$, a branching step consists in fixing a variable as either 0 or 1. Hence, for each node in the Branch & Bound tree, let $I_0 \subseteq \{0, \ldots, n\}$ be indices of variables that are fixed to 0 and let I_1 be the indices of variables that are fixed to 1. The point is that Branch and Bound is not intelligent enough to realize that a sum of integers will be integral. If only few variables are fixed, then there is still a feasible fractional solution that is much better than the best integral solution, hence we would not yet prune the search tree at this point. More formally:

Claim: Consider a node in the search tree with fixed variables (I_0, I_1) and $0 \notin I_0$. If $|I_0|+|I_1| \leq \frac{n}{3}$, then there is a feasible LP solution for this node with $\tilde{x}_0 = 1$ and hence the node could not be pruned.

Proof. Consider the fractional vector $\tilde{x}$ with

$$\tilde{x}_i = \begin{cases} 1 & i = 0 \\ 1 & i \in I_1 \\ 0 & i \in I_0 \\ \lambda & \text{otherwise} \end{cases}$$

where we want to choose λ uniform so that the constraint is satisfied:

$$\frac{1}{2}\tilde{x}_0 + \sum_{i=1}^{n} \tilde{x}_i = \underbrace{\frac{1}{2} + |I_1|}_{\in [0, \frac{1}{2}n] \text{ using } n \geq 3} + \underbrace{(n - |I_0| - |I_1|)}_{\geq \frac{2}{3}n} \cdot \lambda \stackrel{!}{=} \frac{n}{2}$$

We can always choose a $\lambda \in [0, 1]$ to satisfy the equation. Hence there is always a fractional solution of value $\tilde{x}_0 = 1$, hence this node could not be pruned! □

The claim shows that all the leafs of the Branch and Bound tree are in depth at least $\frac{n}{3}$, hence the tree must have at least $2^{n/3}$ many leaves as it is binary[1]. □

[1]Here we assumed that as long as $|I_0| + |I_1| \leq \frac{n}{3}$, an optimum LP solution always has $\tilde{x}_0 = 1$ and hence we would not branch on $i = 0$. One could modify Branch and Bound so that it does branch on $i = 0$, but still the number of leaves would be $2^{\Omega(n)}$.

Chapter 8

Non-bipartite matching

Apart from **NP**-complete problems, the only problem that we have mentioned before and for which have not seen a polynomial time algorithm yet is the **matching problem** on general, non-bipartite graphs. We already discussed the problem with a linear programming approach and in fact, the polynomial time algorithm is highly non-trivial and goes back to seminal work of Edmonds in 1965.

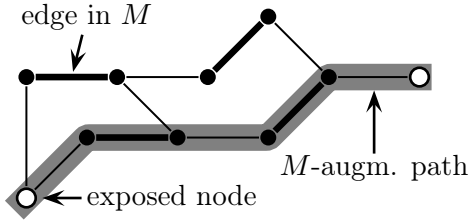
Recall that in an undirected graph $G = (V, E)$, a **matching** is a subset of disjoint edges $M \subseteq E$.

Maximum Cardinality matching
Input: An undirected graph $G = (V, E)$.
Goal: Find a matching M of maximum size $\|M\|$.

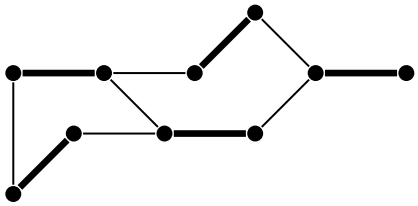
We know already a combinatorial algorithm to solve the problem in bipartite graphs: add a source and a sink and use the Ford-Fulkerson algorithm. In other words, we know that at least for bipartite graphs, an approach with iteratively augmenting a flow/matching must work. Our idea is to extend this to non-bipartite matching.

8.1 Augmenting paths

If we have a matching M, then a node that is not contained in the matching is called M**-exposed**. We call a path $P \subseteq E$ in the graph M**-alternating** if its edges are alternately in M and not in M. We call the path P M**-augmenting** if it is a simple M-alternating path and it starts and ends in different, M-exposed vertices.



matching M with augmenting path

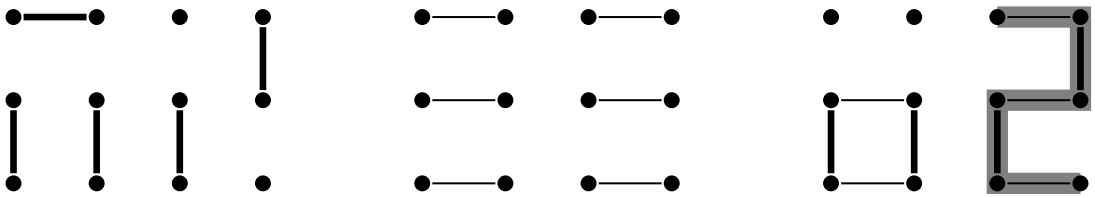


matching after augmentation

Theorem 54 (Augmenting Path Theorem for Matchings)**.** *A matching M in graph $G = (V, E)$ is maximum if and only if there is no M-augmenting path.*

Proof. For edge sets M and P, let $M\Delta P := (M\backslash P) \cup (P\backslash M)$ be their **symmetric difference**. If we have an M-augmenting path P, then $M' := M\Delta P$ is again a matching. Since P started and ended with edges that are not in M, M' contains one edge more.

Now, let us show the more difficult direction. Assume that there is a matching M' that is larger than M. We need to prove that there exists an augmenting path. Consider their symmetric difference $M\Delta M'$. The degrees of nodes in this difference are in $0, 1, 2$. Hence $M\Delta M'$ is a collection of M-alternating cycles and paths.

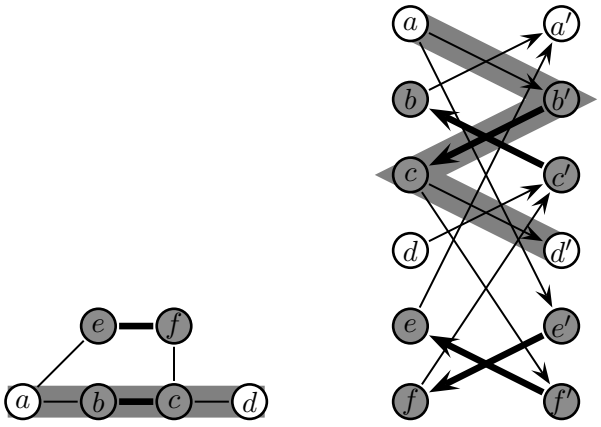


matching M matching M' difference $M\Delta M'$

Since $|M'| > |M|$ and alternating cycles contain the same number of M' and M edges, there must be one path in $M\Delta M'$ that has more edges from M' than from M. This is our M-augmenting path. □

8.2 Computing augmenting paths

Now, let us discuss how we can find these augmenting paths. Let $G = (V, E)$ be our graph, then we create a bipartite graph which for all $v \in V$ has a node v in the left part and a mirror node v' in the right part. For each non-edge $\{u, v\} \in E\backslash M$ we create two directed edges (u, v') and (v, u'). For each matching edge $\{u, v\} \in M$ we create backward edges (u', v) and (v', u).



graph G with matching and augm. path bipartite graph G' with augm. path

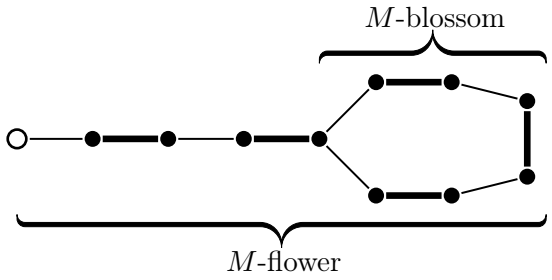
Observe that a path in this directed bipartite graph corresponds to an M-alternating path.

Lemma 55. *In time $O(mn + n^2 \log n)$ we can compute the shortest M-alternating path starting and ending in an exposed node.*

Proof. We simply run Dijkstra's algorithm for each node as source to compute shortest path distances. □

It sounds a bit like we are done. But there is a major problem: a shortest path in the directed bipartite graph G' will be simple, but that graph contains node v twice. In that case, for some $v \in V$ we found a subpath $B \subseteq P$ in the directed graph G' that goes from v to v'. In particular, that means that the length of that path is odd and it contains exactly $\frac{1}{2}(|B|-1)$ many M-edges.

Hence, the whole path P, in the original graph might look as follows



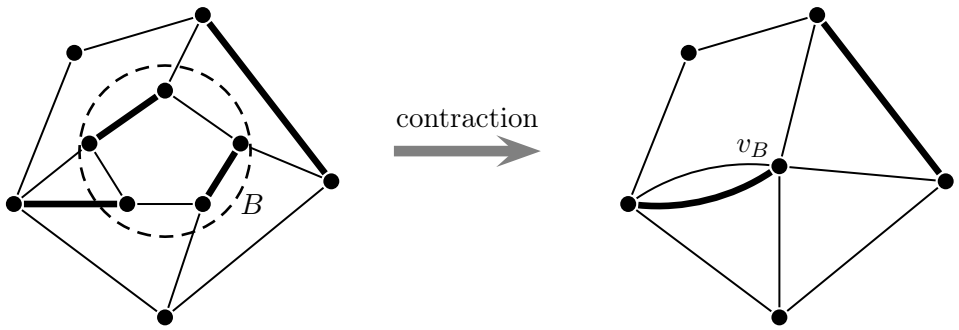
Formally we call an M-alternating path that starts and ends at an M-exposed node and contains exactly one odd cycle an M-**flower**. The odd cycle $B \subseteq E$ that contains exactly $|M \cap B| = \frac{1}{2}(|B|-1)$ edges is called an M-**blossom**. To be precise, a shortest M-alternating path could start and end at different exposed vertices, while containing an M-blossom. But there exists always a shortest M-alternating path that goes back to the **same** exposed vertex.

Lemma 56. *In time $O(nm + n^2 \log n)$ we can compute either an M-augmenting path P or we can find an M-flower B.*

It remains to discuss what to do if we find an M-flower.

8.3 Contracting odd cycles

The crucial idea of Edmonds was to **contract the blossom**. Let B be the edges of the blossom, then we write $G \times B$ as the graph that we obtain by contracting all the nodes on the odd cycle into a single super-node.



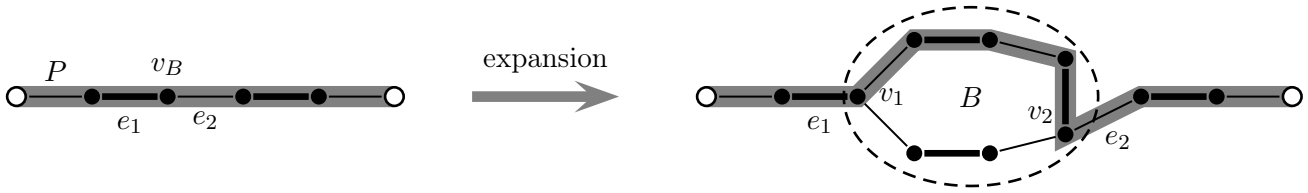
graph G with blossom B contracted graph $G \times B$

Why should we contract an odd cycle? Because the contracted graph is smaller and it suffices to find an augmenting path in the contracted graph. Hence we have made progress.

Lemma 57. *Let M be a matching and B be an odd cycle with $|M \cap B| = \frac{1}{2}(|B|-1)$ many matching edges. Then M/B is a matching in $G \times B$ and any M/B augmenting path in $G \times B$ can be extended to an M-augmenting path in G.*

Proof. First, note that v_B has degree one in M/B since there is only one edge of M in $\delta(B)$. Now, let P be an M/B augmenting path in $G \times B$ and let v_B be the supernode that was created by contraction.

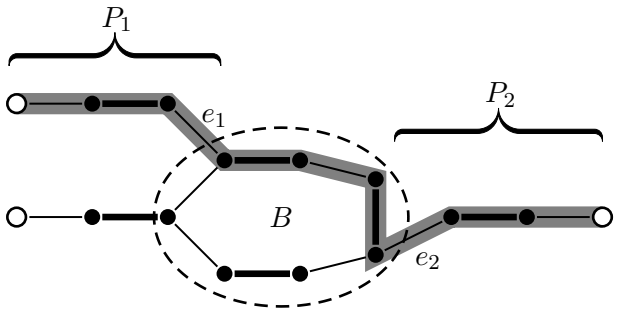
If P does not contain v_B, the claim is clear. After expansion the path P will enter B using an M-edge e_1 in a node that we call v_1 and it will leave B with a non-M edge e_2 in a node v_2. From v_2 we extend P by going either clockwise or counterclockwise so that the first edge that we take from v_2 in B is an M-edge. This way we create an M-augmenting path in G.



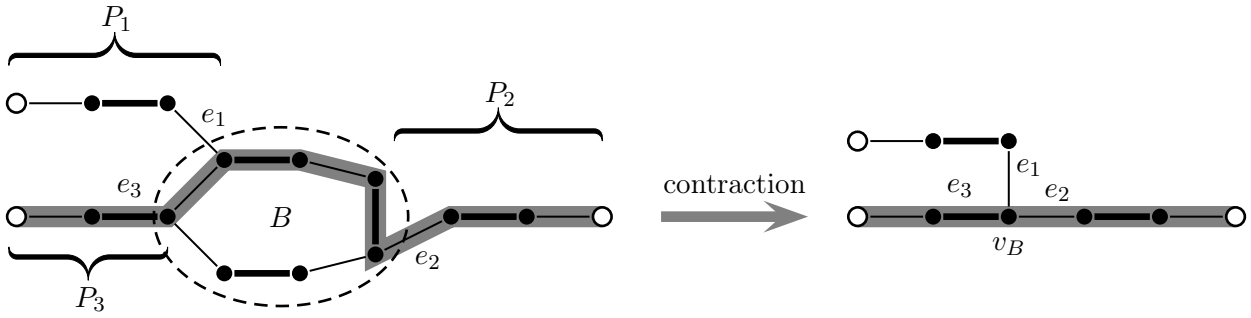
□

Lemma 58. *If there is an M-augmenting path in G, then there is an M/B-augmenting path in $G \times B$.*

Proof. Let P be an M-augmenting path in G. Suppose P uses at last one edge in B otherwise, there is nothing to show. Let P_1 and P_2 be the two pieces of the path P that go from a node in B to an exposed node. Let $e_1 \in P_1$ and $e_2 \in P_2$ be the edges that are incident to B. If one of e_1, e_2 is in M and the other one is not, then $P_1 \cup P_2$ are an M/B-augmenting path in $G \times B$ (as in the picture above). So, suppose not. Then the only possible case is that e_1, e_2 are both **not** in M (here it is important that $|M \cap B| = \frac{1}{2}(|B| - 1)$ so that only one M-edge is in $\delta(B)$).



But we still have the stem of the flower, call it P_3 which starts with an M-edge, call it e_3. At least one of the exposed end points of P_1 and P_2 does not coincide with the exposed vertex in P_3 – say this is the case for P_2. Then $P_2 \cup P_3$ plus some edges of B is an alternative M-augmenting path with the property that $P_2 \cup P_3$ is also an M/B-augmenting path in $G \times B$.



□

Now we have the description of the complete algorithm:

Blossom algorithm for maximum matching (Edmonds 1965)

Input: Undirected graph $G = (V, E)$
Output: Maximum matching $M \subseteq E$.

(1) $M := \emptyset$
(2) REPEAT
  (3) Call Subroutine to find M-augmenting path P
  (4) IF $P =$ `FAIL` THEN Return M ELSE Update $M := M \Delta P$

SUBROUTINE:
Input: Undirected graph $G = (V, E)$, matching $M \subseteq E$
Output: M-augmenting path.

(1) Construct bipartite graph G' and compute shortest path P from exposed node to exposed node
(2) IF no such path exists THEN return `FAIL`
(3) IF P is M-augmenting path THEN return P
(4) IF P corresponds to M-flower with blossom B THEN
  (5) Call subroutine on $G \times B$ and $M \backslash B$
  (6) Extend the returned path by edges in B

From the discussion above we obtain:

Theorem 59. *The Blossom algorithm finds a maximum cardinality matching in time* $O(n^3 m + n^4 \log n)$.

Proof. We need to find an M-augmenting path at most n times. While we try to find an M-augmenting path, we might contract at most n times a cycle and before contracting a cycle we need to find a shortest path in the bipartite graph, which each time takes time $O(nm + n^2 \log n)$. Hence the total running time that we obtain is $O(n^3 m + n^4 \log n)$. □

A more careful implementation (with a more careful analysis) gives a running time of $O(n^3)$.

Chapter 9

The Knapsack problem

In this chapter we want to show more techniques how one can deal with **NP**-hard optimization problems.

To give a motivating example, suppose a burglar breaks into a house and discovers a large set of n precious objects. Unfortunately, all objects together would be to heavy to carry for him, so he has to make some selection. He estimates that he can carry a weight of at most W and that the weight of the ith item is w_i and that he could receive a price of c_i for it.

In a mathematical abstract form we can formulate this as follows:

> KNAPSACK PROBLEM
> **Input:** n items with **weights** $w_1, \ldots, w_n \geq 0$, **profits** $c_1, \ldots, c_n \geq 0$ and a **budget/capacity** W
> **Goal:** Find a subset $S \subseteq \{1, \ldots, n\}$ with $\sum_{i \in S} w_i \leq W$ that maximizes the profit $\sum_{i \in S} c_i$.

An example instance would be

	$i=1$	$i=2$	$i=3$	$i=4$
weights w_i	6	7	4	1
profits c_i	2	3	2	1

budget: $W = 9$

In this case, the optimum solution is $S^* = \{2, 4\}$ and has a profit of 4.

Note that the problem can be formulated as an integer linear program in the form

$$\begin{aligned} \sum_{i=1}^{n} c_i x_i & \\ \sum_{i=1}^{n} w_i x_i & \leq W \\ x & \in \{0,1\}^n \end{aligned}$$

where x_i is a decision variable telling whether you want to select the ith item. In other words, the Knapsack Integer program is an integer program with just a single constraint (apart from the $0 \leq x_i \leq 1$ constraints). Interestingly, that single constraint makes the problem already **NP**-hard, hence there won't be a polynomial time algorithm to solve it in general. We want to design an algorithm that is smarter than just trying out all 2^n possible combinations.

9.1 A dynamic programming algorithm

We want to show that the problem is efficiently solvable if the involved numbers are nice. We will abbreviate $w(S) = \sum_{i \in S} w_i$ and $c(S) = \sum_{i \in S} c_i$.

Lemma 60. *Suppose that we have a Knapsack instance with integral profits $c_1, \ldots, c_n \in \mathbb{Z}_{\geq 0}$. Then one can solve Knapsack optimally in time $O(n \cdot C)$, where $C := \sum_{i=1}^{n} c_i$.*

The technique that we want to use to obtain this result is **dynamic programming** that we saw already at the beginning of the course. The property that makes dynamic programming work is that the problem can be broken into independent subproblems. We want to use the following **table entries** for $j \in \{0, \ldots, n\}$ and $C' \in \{0, \ldots, C\}$

$$\begin{aligned} T(j, C') &:= \text{smallest weight of a subset of items } 1, \ldots, j \text{ that gives profit exactly } C' \\ &= \min\Big\{w(S) \mid S \subseteq \{1, \ldots, j\} : c(S) = C'\Big\} \end{aligned}$$

If there is no set $S \subseteq \{1, \ldots, j\}$ with $c(S) = C'$, then we set $T(j, C') = \infty$. For $j = 0$, it is easy to see that $T(0, 0) = 0$ and $T(0, C') = \infty$ if $C' \neq 0$. Let us make an observation how the other table entries can be computed:

Lemma 61. *For $j \in \{1, \ldots, n\}$ and $C' \in \{0, \ldots, C'\}$ one has $T(j, C') = \min\{T(j-1, C' - c_j) + w_j, T(j-1, C')\}$*

Proof. Suppose that $S^* \subseteq \{1, \ldots, j\}$ is the set with $c(S^*) = C'$ and $w(S^*)$ minimal. Then there are two case:

(a) $j \notin S^*$: Then $T(j, C') = T(j-1, C')$.

(b) $j \in S^*$: Then $T(j, C') = T(j-1, C' - c_j) + w_j$

□

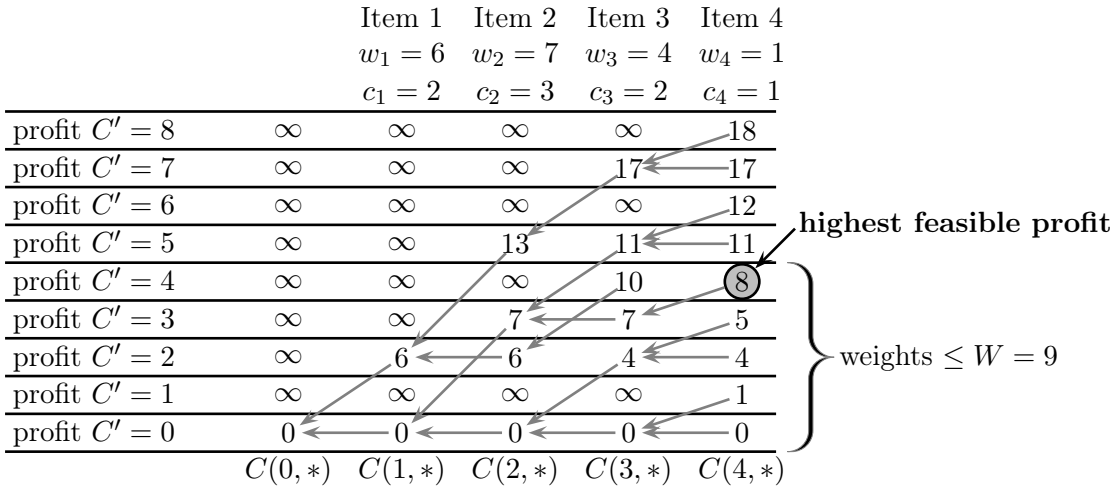
We now write down an algorithm to compute the optimal solution of the knapsack problem.

Dynamic Programming Knapsack Algorithm

Input: Profits $c_1, \ldots, c_n \in \mathbb{Z}_{\geq 0}$, weights $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$, budget $W \in \mathbb{R}_{\geq 0}$
Output: Value of optimum Knapsack solution in time $O(n \cdot C)$

(1) Set $C := \sum_{j=1}^{n} c_j$; $T(0, 0) := 0$, $T(0, C') := \infty \; \forall C' \in \{1, \ldots, C\}$
(2) FOR $i = 1$ TO n DO
$\quad T(j, C') := \min\{T(j-1, C' - c_j) + w_j, T(j-1, C')\}$
(3) Return the max C^* so that $T(n, C^*) \leq W$

We can use the dynamic program to solve the example from above; indeed we obtain that the optimum solution has a valur of 4. One can also "backtrack" that the corresponding solution will be $S^* = \{2, 4\}$ (this can be done by going backwards through the entries and checking whether the min-expression for determining the table entry was attained for the case $j \in S$ or $j \notin S$).



Note that the size of the input to the knapsack problem has a number of bits that is at most $O(n(\log C + \log W))$, hence the running time of $O(nC)$ is **not polynomial** in the size of the input. But the running time satisfies a weaker condition and it is a so-called **pseudopolynomial** running time.

In general, suppose $\mathcal{P}$ is an optimization problem such that each instance consists of a list of integers $a_1, \ldots, a_n \in \mathbb{Z}$. An algorithm for $\mathcal{P}$ is called **pseudopolynomial** if the running time is polynomial in n and $\max\{|a_i| : i \in \{1, \ldots, n\}\}$.

9.2 An approximation algorithm for Knapsack

While Knapsack in general is an **NP**-hard problem, we saw that a solution can be efficiently computed if all the profits are integer numbers in a bounded range. In fact, a similar dynamic program also works if instead the weights are integers in a bounded range.

Now suppose we have a Knapsack instance with arbitrary numbers. While we cannot hope to provably find an optimum solution, we want to find a solution that is at least close to the optimum. Intuitively, the idea is to **round the profit** values so that the dynamic program runs in polynomial time. We assume here implicitly that $w_i \leq W$ for all $i \in \{1, \ldots, n\}$.

Approximation algorithm for Knapsack

Input: Profits $c_1, \ldots, c_n \in \mathbb{R}_{\geq 0}$, weights $w_1, \ldots, w_n \in \mathbb{R}_{\geq 0}$, budget $W \in \mathbb{R}_{\geq 0}$
Output: Solution S with profit $c(S) \geq (1-\varepsilon)OPT$ where OPT denotes the value of the optimum solution.

(1) Scale all profits so that $\max_{i=1,\ldots,n}\{c_i\} = \frac{n}{\varepsilon}$
(2) Round $c'_i := \lfloor c_i \rfloor$, $C' := \sum_{i=1}^{n} c'_i$.
(3) Compute an optimum solution S for instance $((c'_i)_{i\in[n]}, (w_i)_{i\in[n]}, W)$ using the dynamic program.
(4) Return S

First, note that scaling the profits does not affect the performance of the algorithm, hence we assume that from the beginning on, we have $\max_{i=1,\ldots,n}\{c_i\} = \frac{n}{\varepsilon}$. In the following, let OPT' be the optimum value w.r.t. to the rounded weights.

Theorem 62. *The algorithm computes a feasible solution S with $c(S) \geq (1-\varepsilon)OPT$ in time $O(\frac{n^3}{\varepsilon})$.*

Proof. First of all, we gave the correct weights to the dynamic program, but the "wrong" profits. But at least the computed solution S will be feasible, i.e. $w(S) \leq W$. Next, note that the rounded instance has only integral profits as input and their sum is $C' \leq n \cdot \frac{n}{\varepsilon}$. Hence the dynamic program has a running time of $O(n \cdot C')$ which in our case is $O(\frac{n^3}{\varepsilon})$. It remains to show that S is a good solution. This will follow from two claims.

Claim 1: $c(S) \geq OPT'$.

Proof: Since the dynamic program computes the optimum S for the profits c', we know that $c'(S) = OPT'$. Since the profits c'_i are obtained by rounding down c_i we know that $c(S) \geq c'(S)$, which gives the claim. □

Claim 2: $OPT' \geq (1 - \varepsilon)OPT$.

Proof. Let S^* be the solution with $c(S^*) = OPT$. We claim that $c'(S^*) \geq (1 - \varepsilon)c(S^*)$. And in fact, the rounding can cost at most 1 per item hence

$$c'(S^*) \geq c(S^*) - n \geq c(S^*) - \varepsilon c(S^*).$$

Here we used that $c(S^*) \geq \max\{c_i \mid i \in \{1, \ldots, n\}\} = \frac{n}{\varepsilon}$. □

In fact, one can improve the running time by a factor of n. We can estimate the value of OPT up to a factor of 2 and then scale the profits so that $OPT \approx \frac{n}{\varepsilon}$. Then the analysis still goes through, but we have $C \leq O(\frac{n}{\varepsilon})$ which improves the running time to $O(\frac{n^2}{\varepsilon})$.