Algorithms and Probability

Janis Hutz
https://janishutz.com

August 19, 2025

$$\Pr[X|Y] \geq \Pr[X|\overline{Y}]$$
$$X := \text{I pass the exam}$$
$$Y := \text{I read this summary}$$

"*Dieses Algorithmus mapped E nach E Strich*"
- Rasmus Kyng, 2025

FS2025, ETHZ
Summary of the Script and Lectures

Contents

1 Graphs **4**
1.1 Repetition . . . 4
1.4 Connectivity . . . 4
1.4.1 Articulation points . . . 5
1.4.2 Bridges . . . 6
1.4.3 Block-Decomposition . . . 6
1.5 Cycles . . . 7
1.5.1 Eulerian cycles / circuits . . . 7
1.5.2 Hamiltonian Cycles . . . 8
1.5.3 Special cases . . . 10
1.5.4 Travelling Salesman Problem . . . 10
1.6 Matchings . . . 11
1.6.1 Algorithms . . . 11
1.6.2 Hall's Theorem . . . 13
1.7 Colourings . . . 14

2 Combinatorics **16**
2.1 Introduction . . . 16
2.2 Simple counting operations . . . 16
2.3 Basic rules of counting . . . 16
2.3.1 Multiplication rule . . . 16
2.3.2 Addition rule . . . 16
2.4 Factorial . . . 17
2.4.1 Operations . . . 17
2.5 Permutations . . . 17
2.5.1 Permutation with repetition . . . 17
2.6 Variations . . . 18
2.6.1 Variations with repetition . . . 18
2.7 Combinations . . . 18
2.7.1 Combination with repetition . . . 18
2.8 Binomial Expansion . . . 18
2.9 Overview . . . 19

3 Probability **20**
3.1 Basics . . . 20
3.2 Conditional Probability . . . 22
3.3 Independence . . . 23
3.4 Random Variables . . . 24
3.4.1 Expected value . . . 24
3.4.2 Variance . . . 26
3.4.3 Intuition . . . 26
3.5 Discrete distribution . . . 27
3.5.1 Bernoulli-Distribution . . . 27
3.5.2 Binomial Distribution . . . 27
3.5.3 Geometric Distribution . . . 27
3.5.4 Poisson distribution . . . 28
3.6 Multiple random variables . . . 29
3.6.1 Independence of random variables . . . 29
3.6.2 Composite random variables . . . 30
3.6.3 Moments of composite random variables . . . 30
3.6.4 Wald's Identity . . . 30
3.7 Approximating probabilities . . . 31
3.7.1 Markov's & Chebyshev's inequalities . . . 31
3.7.2 Chernoff bounds . . . 31
3.8 Randomized Algorithms . . . 32
3.8.1 Reduction of error . . . 32

3.8.2 Sorting and selecting . . . 33
3.8.3 Primality test . . . 34
3.8.4 Target-Shooting . . . 35
3.8.5 Finding dulicates . . . 35

4 Algorithms **36**
4.1 Graph algorithms . . . 36
4.1.1 Long path problem . . . 36
4.1.2 Flows . . . 38
4.1.3 Min-Cuts in graphs . . . 43
4.2 Geometric Algorithms . . . 45
4.2.1 Smallest enclosing circle . . . 45
4.2.2 Convex hull . . . 47

5 Coding **49**
5.1 Tricks . . . 49

1 Graphs

1.1 Repetition

The graph algorithms of Kruskal, Prim, Boruvka, Dijkstra, Bellman-Ford, Floyd-Warshall and Johnson are still of importance. See A & D summary for them

α-approximation algorithm **Definition 1.1**

This kind of algorithm approximates the result to level α, where α is the factor by which the result is off. Ideally, $\alpha = 1$, for a perfect approximation, which in reality is hardly ever, if ever possible.

1.4 Connectivity

k-Connected graph **Definition 1.23**

A graph $G = (V, E)$ is called *k-connected* if $|V| \geq k + 1$ and for all subsets $X \subseteq V$ with $|X| < k$ we have that the graph $G[V \backslash X]$ is connected

If $G[V \backslash X]$ is ***not*** connected, we call it a *(vertex)-separator*.

A k-connected graph is also $k - 1$ connected, etc

k-edge-connected **Definition 1.24**

A graph $G = (V, E)$ is called *k-edge-connected* if for all subsets $X \subseteq E$ with $|X| < k$ we have: The graph $(V, E \backslash X)$ is connected

We also have *edge-separators*, defined analogously to above, i.e. the set X above is called a u-v-edge-separator if by removing it from the graph, u and v no longer lay in the same connected component.

If a graph is k-edge-connected, it is also $k - 1$-edge-connected, etc

Menger's theorem **Theorem 1.25**

Let $G = (V, E)$ be a graph and $u \neq v \in V$. Then we have

1. Every u-v-vertex-separator has size at least $k \Leftrightarrow$ Exist at least k internally vertex-disjoint u-v-paths
2. Every u-v-edge-separator has size at least $k \Leftrightarrow$ Exist at least k edge-disjoint u-v-paths

1.4.1 Articulation points

If a graph is connected, but not 2-connected, there exists at least one vertex v for which, if removed, the graph is not connected, called *articulation points*. Using a modified DFS, we can find these vertices. Instead of just setting a flag for each vertex we visit, we set a number indicating the order in which the vertices were visited. The first vertex we visit, for instance, gets number 1.

We also add another value, which we call `low[v]` := the smallest DFS-number that can be reached from v using a path of arbitrarily many edges of the DFS-tree and at most one residual edge.

We also have (where s is the start / source vertex and $E(T)$ is the edge set of the DFS-tree):

v is an articulation point $\Leftrightarrow$ ($v = s$ and s has degree at least 2 in T) or ($v \neq s$ and there exists $w \in V$ with $\{(v, w)\} \in E(T)$ and `low[w]` $\geq$ `dfs[v]`)

Algorithm 1 FindArticulationPoints(G, s)

```
1  ∀v ∈ V: dfs[v] ← 0                          ▷ Stores dfs number for vertex v and also flag for visit
2  ∀v ∈ V: low[v] ← 0                          ▷ Stores low point for vertex v
3  ∀v ∈ V: isArticulationPoint[v] ← false      ▷ Indicates if vertex v is articulation point
4  num ← 0
5  T ← ∅                                       ▷ Depth-First-Search Tree
6  DFS-Visit(G, s)
7  if s has degree at least two in T then      ▷ Start vertex classification could be incorrect from DFS-Visit
8      isArticulationPoint[s] ← true
9  else
10     isArticulationPoint[s] ← false
11 procedure DFS-Visit(G, v)
12     num ← num +1
13     dfs[v] ← num
14     low[v] ← dfs[v]
15     for all {v, w} ∈ E do
16         if dfs[w] = 0 then
17             T ← T ∪ {{v, w}}
18             val ← DFS-Visit(G, w)
19             if val ≥ dfs[v] then            ▷ Check articulation point condition
20                 isArticulationPoint[v] ← true
21             low[v] ← min{low[v], val}
22         else if dfs[w] ≠ 0 and {v, w} ∉ T then   ▷ Update low if already visited
23             low[v] ← min{low[v], dfs[w]}
24     return low[v]
```

Articulation points Computation — **Theorem 1.27**

For a connected graph $G = (V, E)$ that is stored using an adjacency list, we can compute all articulation points in $\mathcal{O}(|E|)$

1.4.2 Bridges

While articulation points show that a graph is not 2-connected, bridges shows that a graph isn't 2-edge-connected. In other words, they are certificates for the graph *not* being 2-edge-connected or more formally:

An edge $e \in E$ in a connected graph $G = (V, E)$ is called a bridge if the graph $(V, E \backslash \{e\})$ is not connected.

From the definition of bridges we immediately have that a spanning tree has to contain all bridges of a graph, and we can also state that only edges of a Depth-First-Search-Tree are possible candidates for a bridge.

The idea now is that every vertex contained in a bridge is either an articulation point or has degree 1 in G. Or more formally:

A directed edge (v, w) of Depth-First-Search-Tree T is a bridge if and only if `low[w] > dfs[v]`

Bridges Computation — **Theorem 1.28**

For a connected graph $G = (V, E)$ that is stored using an adjacency list, we can compute all bridges and articulation points in $\mathcal{O}(|E|)$

1.4.3 Block-Decomposition

Block-Decomposition — **Definition 1.29**

Let $G = (V, E)$ be a connected graph. For $e, f \in E$ we define a relation by

$$e \sim f \iff e = f \text{ or exists a common cycle through } e \text{ and } f$$

Then this relation is an equivalence relation and we call the equivalence classes *blocks*, sometimes also known as 2-connectivity-components

It is now evident that two blocks, if even, can only intersect in one articulation point. The Block-Decomposition is given by:

Let T be a bipartite graph (in this case a tree), with $V = A \uplus B$ where A is the set of articulation points of G and B the set of blocks of G (This means that every block in G is a vertex in V).
We connect a vertex $a \in A$ with a block $b \in B$ if and only if a is incident to an edge in b.
T is connected if G is and it is free of cycles if G is free of cycles, since every cycle is translatable to a cycle in G

The algorithm to determine bridges and articulation points can again be reused and allows us to determine a Block-Decomposition in linear time.

1.5 Cycles

1.5.1 Eulerian cycles / circuits

Eulerian cycle — **Definition 1.30**

A *eulerian cycle* in a graph $G = (V, E)$ is a circuit (closed cycle) that contains each edge exactly once. If a graph contains a eulerian cycle, we call it *eulerian*

If G contains a eulerian cycle, $\deg(v)$ of all vertices $v \in V$ is even. For connected graph, we even have a double-sided implication.

If we combine the entirety of the explanations of pages 43-45 in the script, we reach the following algorithm, where $N_G(v)$ is the function returning the neighbours of vertex v in graph G:

Algorithm 2 EulerianCycle(G, v_{start})

```
1  procedure RandomCycle(G, v_start)
2      v ← v_start
3      W ← ⟨v⟩                                  ▷ Prepare the cycle (add the start vertex to it)
4      while N_G(v) ≠ ∅ do
5          Choose v_next arbitrarily from N_G(v)   ▷ Choose arbitrary neighbour
6          Attach v_next to the cycle W
7          e ← {v, v_next}
8          Delete e from G
9          v ← v_next
10     return W
11 W ← RandomCycle(v_start)                        ▷ Fast runner
12 v_slow ← start vertex of W
13 while v_slow is not the last vertex in W do
14     v ← successor of v_slow in W
15     if N_G(v) ≠ ∅ then
16         W' ← RandomCycle(v)
17         W ← W_1 + W' + W_2                      ▷ We union the different branches of the Euler cycle
18     v_slow ← successor of v_slow in W
19 return W
```

Eulerian Graph — **Theorem 1.31**

a) A connected graph G is eulerian if and only if the degree of all vertices is even
b) In a connected eulerian graph, we can find a eulerian cycle in $\mathcal{O}(|E|)$

1.5.2 Hamiltonian Cycles

Hamiltonian Cycle — **Definition 1.32**

A *Hamiltonian Cycle* in a graph $G = (V, E)$ is a cycle passing through each vertex *exactly once*. If a graph contains a Hamiltonian cycle, we call it *Hamiltonian*

A classic example here is the Travelling Salesman Problem (TSP), covered later on.

The issue with Hamiltonian cycles is that the problem is $\mathcal{NP}$-complete, thus it is assumed that there does not exist an algorithm that can determine if a graph is Hamiltonian in polynomial time.

Hamiltonian Cycle Algorithm — **Theorem 1.34**

The algorithm HamiltonianCycle is correct and has space complexity $\mathcal{O}(n \cdot 2^n)$ and time complexity $\mathcal{O}(n^2 \cdot 2^n)$, where $n = |V|$

In the below algorithm, $G = (V, E)$ is a graph for which $V = [n]$ and $N(v)$ as usual the neighbours of v and we define S as a subset of the vertices of G with $1 \in S$. We define

$$P_{S,x} := \begin{cases} 1 & \text{exists a 1-}x\text{-path in } G \text{ that contains exactly the vertices of } S \\ 0 & \text{else} \end{cases}$$

We then have:

$$G \text{ contains a Hamiltonian cycle } \iff \exists x \in N(1) \text{ with } P_{[n],x} = 1$$

Or in words, a graph contains a Hamiltonian Cycle if and only if for any of the neighbours of vertex 1, our predicate $P_{S,x} = 1$ for $S = V = [n]$ and x being that vertex in the neighbours set $N(1)$.

This means, we have found a recurrence relation, albeit an exponential one.

Algorithm 3 HamiltonianCycle$(G = ([n], E))$

```
1  for all x ∈ [n], x ≠ 1 do                                   ▷ Initialization
2      P_{{1,x},x} := { 1  if {1,x} ∈ E
                      { 0  else
3  for s = 3, ..., n do                                        ▷ Recursion
4      for all S ⊆ [n] with 1 ∈ S and |S| = s do               ▷ See implementation notes in Section 5
5          for all x ∈ S, x ≠ 1 do                             ▷ Fill table for all x in the subset
6              P_{S,x} = max{P_{S\{x},x'} | x' ∈ S ∩ N(x), x' ≠ 1}
7  if ∃x ∈ N(1) with P_{[n],x} = 1 then                        ▷ Check condition
8      return true
9  else
10     return false
```

Improved algorithm

There are algorithms that can find Hamiltonian cycles without using exponential memory usage. The concept for that is the inclusion-exclusion principle (more on that in Section 3.1)

Inclusion-Exclusion-Principle — **Theorem 1.35**

For finite sets $A_1, \dots, A_n$ $(n \geq 2)$ we have

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{l=1}^{n}\left((-1)^{l+1} \sum_{1\leq i_1<\dots<i_l\leq n} |A_{i_1} \cap \dots \cap A_{i_l}|\right)$$
$$= \sum_{i=1}^{n} |A_i| - \sum_{1\leq i_1<i_2\leq n} |A_{i_1} \cap A_{i_2}| + \sum_{1\leq i_1<i_2<i_3\leq n} |A_{i_1} \cap A_{i_2} \cap A_{i_3}| - \dots + (-1)^{n+1} \cdot |A_1 \cap \dots \cap A_n|$$

Since it is easier to find walks compared to paths, we define for all subsets $S \subseteq [n]$ with $v \notin S$ for a start vertex $s \in V$

$$W_S := \{\textit{walks} \text{ of length } n \text{ in } G \text{ with start and end vertex } s \text{ that doesn't visit any vertices of } S\}$$

We thus reach the following algorithm:

Algorithm 4 CountHamiltonianCycles($G = ([n], E)$)

```
s ← 1                                   ▷ Start vertex, can be chosen arbitrarily
Z ← |W_∅|                               ▷ All possible paths with length n in G
for all S ⊆ [n] with s ∉ S and S ≠ ∅ do
    Compute |W_S|                       ▷ With adjacency matrix of G[V\S]
    Z ← Z + (−1)^|S| |W_S|              ▷ Inclusion-Exclusion
Z ← Z/2                                 ▷ There are two cycles for each true cycle (in both directions, we only care about one)
return Z                                ▷ The number of Hamiltonian cycles in G
```

Count Hamiltonian Cycles Algorithm — **Theorem 1.36**

The algorithm computes the number of Hamiltonian cycles in G with space complexity $\mathcal{O}\left(n^2\right)$ and time complexity $\mathcal{O}\left(n^{2.81}\log(n)\cdot 2^n\right)$, where $n = |V|$

The time complexity bound comes from the fact that we need $\mathcal{O}(\log(n))$ matrix multiplications to compute $|W_S|$, which can be found in entry (s, s) in $(A_S)^n$, where A_S is the adjacency matrix of the induced subgraph $G[V\backslash S]$. Each matrix multiplication can be done in $\mathcal{O}\left(n^{2.81}\right)$ using Strassen's Algorithm. The 2^n is given by the fact that we have that many subsets to consider.

1.5.3 Special cases

Bipartite graph — **Lemma 1.38**

If $G = (A \uplus B, E)$ is a bipartite graph with $|A| \neq |B|$, G cannot contain a Hamiltonian cycle

A hypercube H_d with dimension d has the vertex set $\{0,1\}^d$. Two vertices are connected if and only if their 0-1-sequences differ in exactly one bit.

Every hypercube of dimension $d \geq 2$ has a Hamiltonian cycle

Grid graphs (also known as mesh graphs) are graphs laid out in a (typically) square grid of size $m \times n$

A grid graph contains a Hamiltonian cycle if and only if n or m (or both) are even. If both are odd, there is no Hamiltonian cycle

Dirac — **Theorem 1.40**

If G is a graph with $|V| \geq 3$ vertices, for which every vertex has at least $\frac{|V|}{2}$ neighbours, G is Hamiltonian.

In other words, every graph with minimum degree $\frac{|V|}{2}$ is Hamiltonian.

1.5.4 Travelling Salesman Problem

Given a graph K_n and a function $l : \binom{[n]}{2} \to \mathbb{N}_0$ that assigns a length to each edge of the graph, we are looking for a Hamiltonian cycle C in K_n with

$$\sum_{e \in C} l(e) = \min \left\{ \sum_{e \in C'} l(e) \mid C' \text{ is a Hamiltonian cycle in } K_n \right\}$$

In words, we are looking for the hamiltonian cycle with the shortest length among all hamiltonian cycles.

Travelling Salesman Problem — **Theorem 1.42**

If there exists for $\alpha > 1$ a α-approximation algorithm for the travelling salesman problem with time complexity $\mathcal{O}(f(n))$, there also exists an algorithm that can decide if a graph with n vertices is Hamiltonian in $\mathcal{O}(f(n))$.

This obviously means that this problem is also $\mathcal{NP}$-complete. If we however use the triangle-inequality $l(\{x,z\}) \leq l(\{x,y\}) + l(\{y,z\}))$, which in essence says that a direct connection between two vertices has to always be shorter or equally long compared to a direct connection (which intuitively makes sense), we reach the metric travelling salesman problem, where, given a graph K_n and a function l (as above, but this time respecting the triangle-inequality), we are again looking for the same answer as for the non-metric problem.

Metric Travelling Salesman Problem — **Theorem 1.43**

There exists a 2-approximation algorithm with time complexity $\mathcal{O}\left(n^2\right)$ for the metric travelling salesman problem.

Proof: This algorithm works as follows: Assume we have an MST and we walk around the outside of it. Thus, the length of our path is 2 $\texttt{mst}(K_n, l)$. If we now use the triangle inequality, we can skip a few already visited vertices and at least not lengthen our journey around the outside of the MST. Any Hamiltonian cycle can be transformed into an MST by removing an arbitrary edge from it. Thus, for the optimal length (minimal length) of a Hamiltonian cycle, we have $\text{opt}(K_n, l) \geq \texttt{mst}(K_n, l)$. If we now double the edge set (by duplicating each edge), then, since for $l(C) = \sum_{e \in C} l(e)$ for our Hamiltonian cycle C, we have $l(C) \leq 2\text{opt}(K_n, l)$, we can simply find a eulerian cycle in the graph in $\mathcal{O}(n)$, and since it takes $\mathcal{O}\left(n^2\right)$ to compute an MST, our time complexity is $\mathcal{O}\left(n^2\right)$

1.6 Matchings

Matchings are assignment problems, which could take the form of assigning a job to a specific CPU core or system. That system will have to fulfill the performance requirements of the task, but performance should not be wasted, as there could be a different job with higher performance requirements that would not be processable simultaneously otherwise.

Matching **Definition 1.44**

An edge set $M \subseteq E$ of a graph G is called a *matching* if no vertex of the graph is assigned to more than one vertex, or more formally:

$$e \cap f = \emptyset \text{ for all } e, f \in M \text{ with } e \neq f$$

We call a vertex v *covered* by M if there exists an edge $e \in M$ that contains v.
A matching is called a *perfect matching* if every vertex is covered by an edge of M, or equivalently $|M| = \frac{|V|}{2}$

Maxima **Definition 1.46**

Given a graph G and matching M in G

- M is called a *maximal matching* (or in German "inklusionsmaximal") if we have $M \cup \{e\}$ is no matching for all $e \in E \backslash M$
- M is called a *maximum matching* (or in German "kardinalitätsmaximal") if we have $|M| \geq |M'|$ for all matchings M' in G

1.6.1 Algorithms

Algorithm 5 GREEDY-MATCHING(G)

```
1 M ← ∅
2 while E ≠ ∅ do
3     choose an arbitrary edge e ∈ E        ▷ Randomly choose from E
4     M ← M ∪ {e}
5     delete e and all incident edges (to both vertices) in G
```

The above algorithm doesn't return the maximum matching, just a matching

Greedy-Matching **Theorem 1.47**

The GREEDY-MATCHING determines a maximal matching M_{Greedy} in $\mathcal{O}(|E|)$ for which we have

$$|M_{Greedy}| \geq \frac{1}{2}|M_{\max}|$$

where $M_{\max}$ is a maximum matching

Berge's Theorem **Theorem 1.48**

If M is a not a maximum matching in G, there exists an augmenting path to M

Proof: If M is not a maximum matching, there exists a matching M' with higher cardinality, where $M \oplus M'$ (M xor M') has a connected component that contains more edges of M' than M. Said connected component is the augmenting path for M

This idea leads to an algorithm to determine a maximum matching: As long as a matching isn't a maximum matching, there exists an augmenting path that allows us to expand the matching. After *at most* $\frac{|V|}{2} - 1$ steps, we have a maximum matching. For bipartite graphs, we can use modified BFS with time complexity $\mathcal{O}((|V| + |E|) \cdot |E|)$ to determine the augmenting paths.

Algorithm 6 AugmentingPath($G = (A \uplus B, E), M$)

```
L_0 := {set of all non-covered vertices in A}
if L_0 = ∅ then
    return M is a maximum matching
Mark all vertices in L_0 as visited
for i = 1, ..., n do
    if i is odd then                          ▷ We start with an unmatched vertex (by definition)
        L_i := {all unvisited neighbours of L_{i-1} using edges in E\M}
    else
        L_i := {all unvisited neighbours of L_{i-1} using edges in M}
    Mark all vertices in L_i as visited       ▷ We used them in our augmenting path, note that
    if L_i contains non-covered vertex v then
        Find path P starting at L_0 and ending at v using backtracking
        return P
return M is already a maximum matching
```

Augmenting Path: An *alternating path* that (here) starts from unmatched vertices, where an alternating path is a path that starts with an unmatched vertex and whose edges alternately belong to the matching and not.

The algorithm discussed above uses layers L_i to find the augmenting paths. Each of the layers is alternatingly part of the matching and not part of it, where the first one is not part of the matching. Augmenting paths also always have length m odd, so the last layer is *not* part of the matching.

Hopcroft and Karp Algorithm

Algorithm 7 MaximumMatching($G = (A \oplus B, E)$)

```
M ← {e} for any edge e ∈ E      ▷ Initialize Matching with simple one of just one edge (trivially a matching)
while there are still augmenting paths in G do
    k ← length of the shortest augmenting path
    find a maximal set S of pairwise disjunct augmenting paths of length k
    for all P of S do
        M ← M ⊕ P                           ▷ Augmenting along all paths of S
return M
```

To find the shortest augmenting path, we observe that if the last layer has more than one non-covered vertex, we can potentially (actually, likely) find more than one augmenting path. We find one first, remove it from the data structure and find more augmenting paths by inverting the tree structure (i.e. cast *flippendo* on the edges) and using DFS to find the all augmenting paths. We always delete each visited vertex and we thus have time complexity $\mathcal{O}(|V| + |E|)$, since we only visit each vertex and edge once.

Hopcroft and Karp Algorithm — **Theorem 1.49**

The algorithm of Hopcroft and Karp's while loop is only executed $\mathcal{O}\left(\sqrt{|V|}\right)$ times. Hence, the maximum matching is computed in $\mathcal{O}\left(\sqrt{|V|} \cdot (|V| + |E|)\right)$

Other matching algorithms

In Section 4, using flows to compute matchings is discussed.

Weighted Matching problem — **Theorem 1.50**

Let n be even and $l : \binom{[n]}{2} \to \mathbb{N}_0$ be a weight function of a complete graph K_n. Then, we can compute, in time $\mathcal{O}\left(n^3\right)$, a minimum perfect matching with

$$\sum_{e \in M} l(e) = \min \left\{ \sum_{e \in M'} l(e) \;\middle|\; M' \text{ is a perfect matching in } K_n \right\}$$

MTSP with approximation — **Theorem 1.51**

There is a $\frac{3}{2}$-approximation algorithm with time complexity $\mathcal{O}\left(n^3\right)$ for the metric travelling salesman problem

1.6.2 Hall's Theorem

Hall's Theorem — **Theorem 1.52**

For a bipartite graph $G = (A \uplus B, E)$ there exists a matching M with cardinality $|M| = |A|$ if and only if

$$|N(X)| \geq |X| \quad \forall X \subseteq A$$

The following theorem follows from Hall's Theorem immediately. We remember that a graph is k-regular if and only if every vertex of the graph has degree exactly k

Matching in k-regular bipartite graphs — **Theorem 1.53**

Let G be a k-regular bipartite graph. Then there exists $M_1, \ldots, M_k$ such that $E = M_1 \uplus \ldots \uplus M_k$ where each M_i is a perfect matching

Algorithm for the problem — **Theorem 1.54**

If G is a 2^k-regular bipartite graph, we can find a perfect matching in time $\mathcal{O}\left(|E|\right)$

It is important to note that the algorithms to determine a perfect matching in bipartite graph do not work for non-bipartite graphs, due to the fact that when we remove every other edge from the eulerian cycle, it is conceivable that the graph becomes disconnected. While this is no issue for bipartite graphs (as we can simply execute the graph on all connected components), for $k = 1$, such a connected component can could contain an odd number of vertices, thus there would be a eulerian cycle of odd length from which not every other edge can be deleted. Since that component has odd length, no perfect matching can exist.

1.7 Colourings

Good examples for problems that can be solved using colourings are the channel picking of wireless devices, for compilers to pick registers and for creating timetables and exam schedules.

Colouring — **Definition 1.56**

A *(vertex-)colouring* of a graph G with k colours is an image $c : V \to [k]$ such that

$$c(u) \neq c(v) \quad \forall \{u, v\} \in E$$

The chromatic number $\mathscr{X}(G)$ is the minimum number of colours that are required to colour the graph G

Graphs with chromatic number k are also called *k-partite*, where from the naming of a *bipartite* graph comes, since we can *partition* the graph into k separate sets

Bipartite graph — **Theorem 1.58**

A graph $G = (V, E)$ is bipartite if and only if it does not contain a cycle of uneven length as sub-graph

On political maps, two neighbouring countries are coloured in different colours. We assume that the territories of every country is connected and that all countries that only touch at one point can be coloured with the same colour.

Four colour theorem — **Theorem 1.59**

Every land map can be coloured in four colours

Again, the problem of determining if the chromatic number of a graph is smaller than a value t is another $\mathcal{NP}$-complete problem. We thus have to again proceed with an approximation.

The following algorithm correctly computes ***a*** valid colouring.

Greedy-Colouring Algorithm — **Theorem 1.60**

For the number of colours $C(G)$ the Greedy-Colouring needs to colour the connected graph G we have

$$\mathscr{X}(G) \leq C(G) \leq \Delta(G) + 1$$

where $\Delta(G) := \max_{v \in V} \deg(v)$ is the maximum degree of a vertex in G. If the graph is stored as an adjacency list, the algorithm finds a colouring $\mathcal{O}(|E|)$

Algorithm 8 Greedy-Colouring(G)

```
1 Choose an arbitrary order of vertices V = {v_1, ..., v_n}
2 c[v_1] ← 1
3 for i = 2, ..., n do
4     c[v_i] ← min{k ∈ ℕ | k ≠ c(u)  ∀u ∈ N(v_i) ∩ {v_1, ..., v_{i-1}}}   ▷ Find minimum available colour
```

Brook's theorem — **Theorem 1.61**

Let G be a connected graph that is neither complete nor a cycle of uneven length (uneven cycle), we have

$$\mathscr{X}(G) \leq \Delta(G)$$

and there is an algorithm that colours the graph using $\Delta(G)$ colours in $\mathcal{O}(|E|)$. Otherwise $\mathscr{X}(G) \leq \Delta(G)+1$

Of note is that a graph with an even number of vertices and edges does not contain an uneven cycle, so for an incomplete graph with an even number of edges and vertices, we always have that $\mathscr{X}(G) \leq \Delta(G)$

Maximum degree **Theorem 1.62**

Let G be a graph and $k \in \mathbb{N}$ the number representing the maximum degree of any vertex of any induced subgraph of G. Then we have $\mathscr{X}(G) \leq k+1$ and a $(k+1)$-coloring can be found in $\mathcal{O}(|E|)$

Mycielski-Construction **Theorem 1.63**

For all $k \geq 2$ there exists a triangle-free graph G_k with $\mathscr{X}(G_k) \geq k$

It is even possible to show that for all $k, l \geq 2$ there exists a graph $G_{k,l}$ such that said graph doesn't contain a cycle of length *at most* l, but we still have $\mathscr{X}(G_{k,l}) \geq k$.

To conclude this section, one last problem:

We are given a graph G which we are told has $\mathscr{X}(G) = 3$. This means, we know that there is exists an order of processing for the Greedy-Colouring algorithm that only uses three colours. We don't know the colours, but we can find an upper bound for the number of colours needed

3-colourable graphs **Theorem 1.64**

Every 3-colourable graph G can be coloured in time $\mathcal{O}(|E|)$ using at most $\mathcal{O}\left(\sqrt{|V|}\right)$ colours

Since the graph has to be bipartite (because for each vertex v, its neighbours can only be coloured in 2 other colours, because the graph can be 3-coloured), we can use BFS and thus have linear time. The algorithm works as follows: We choose the vertices with the largest degree and apply three colours to them. For the ones of smaller degree, we apply Brook's theorem.

2 Combinatorics

2.1 Introduction

Combinatorics was developed from the willingness of humans to gamble and the fact that everybody wanted to win as much money as possible.

2.2 Simple counting operations

The easiest way to find the best chance of winning is to write down all possible outcomes. This can be very tedious though when the list gets longer.

We can note this all down as a list or as a tree diagram. So-called Venn Diagrams might also help represent the relationship between two sets or events. Essentially a Venn Diagram is a graphical representation of set operations such as $A \cup B$.

2.3 Basic rules of counting

2.3.1 Multiplication rule

If one has n possibilities for a first choice and m possibilities for a second choice, then there are a total of $n \cdot m$ possible combinations.

When we think about a task, and we have an **and** in between e.g. properties, we need to multiply all the options.

2.3.2 Addition rule

If two events are mutually exclusive, the first has n possibilities and the second one has m possibilities, then both events together have $n + m$ possibilities.

When we think about a task, and we have an **or** in between e.g. properties, then we need to add all the options.

2.4 Factorial

Factorial **Definition 2.1**

The factorial stands for the product of the first n natural numbers where $n \geq 1$. Notation: !

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Additionally, $0! = 1$. We read $n!$ as "*n factorial*"

2.4.1 Operations

We can rewrite $n!$ as $n \cdot (n-1)!$ or $n \cdot (n-1) \cdot (n-2)!$ and so on.

It is also possible to write $7 \cdot 6 \cdot 5$ with factorial notation: $\frac{7!}{4!}$, or in other words, for any excerpt of a factorial sequence:

$$n \cdot (n-1) \cdot \ldots \cdot m = \frac{n!}{(m-1)!}$$

2.5 Permutations

Permutations **Definition 2.2**

A permutation of a group is any possible arrangement of the group's elements in a particular order

Permutation rule without repetition: The number of n ***distinguishable*** elements is defined as: $n!$

2.5.1 Permutation with repetition

For n elements $n_1, n_2, \ldots, n_k$ of which some are identical, the number of permutations can be calculated as follows:

$$p = \frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}$$

where n_k is the number of times a certain element occurs. As a matter of fact, this rule also applies to permutations without repetition, as each element occurs only once, which means the denominator is 1, hence $\frac{n!}{(1!)^n} = n!$

Example 2.3: CANADA has 6 letters, of which 3 letters are the same. So the word consists of 3 A's, which can be arranged in 3! different ways, a C, N and D, which can be arranged in 1! ways each. Therefore, we have:

$$\frac{6!}{3! \cdot 1! \cdot 1! \cdot 1!} = \frac{6!}{3!} = 6 \cdot 5 \cdot 4 = 120$$

Since 1! equals 1, we can always ignore all elements that occur only once, as they won't influence the final result.

2.6 Variations

Variations **Definition 2.4**

A ***variation*** is a selection of k elements from a universal set that consists of n *distinguishable* elements.

Variation rule without repetition: The ${}_n\mathrm{P}_k$ function is used to ***place*** n elements on k places. In a more mathematical definition: The number of different variations consisting of k different elements selected from n distinguishable elements can be calculated as follows:

$$\frac{n!}{(n-k)!} = {}_n\,\mathrm{P}_k$$

2.6.1 Variations with repetition

If an element can be selected more than once and the order matters, the number of different variations consisting of k elements selected from n distinguishable elements can be calculated using n^k

2.7 Combinations

Combination **Definition 2.5**

A combination is a selection of k elements from n elements in total without any regard to order or arrangement.
Combination rule without repetition:

$${}_n\mathrm{C}_k = \binom{n}{k} = \frac{{}_n\mathrm{P}_k}{k!} = \frac{n!}{(n-k)! \cdot k!}$$

2.7.1 Combination with repetition

In general the question to ask for combinations is, in how many ways can I distribute k objects among n elements?

$${}_{n+k-1}\mathrm{C}_k = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$$

2.8 Binomial Expansion

Binomial expansion is usually quite hard, but it can be much easier than it first seems. The first term of the expression of $(a+b)^n$ is always $1a^nb^0$. Using the formula for combination without repetition, we can find the coefficients of each element:

$$\text{6th row} \begin{cases} {}_6C_0 & {}_6C_1 & {}_6C_2 & {}_6C_3 & {}_6C_4 & {}_6C_5 & {}_6C_6 \\ 1 & \frac{6}{1} & \frac{6\cdot5}{1\cdot2} & \frac{6\cdot5\cdot4}{1\cdot2\cdot3} & \frac{6\cdot5\cdot4\cdot3}{1\cdot2\cdot3\cdot4} & \frac{6\cdot5\cdot4\cdot3\cdot2}{1\cdot2\cdot3\cdot4\cdot5} & \frac{6!}{6!} \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{cases}$$

This theory is based on the Pascal's Triangle and the numbers of row n correspond to the coefficients of each element of the expanded term.

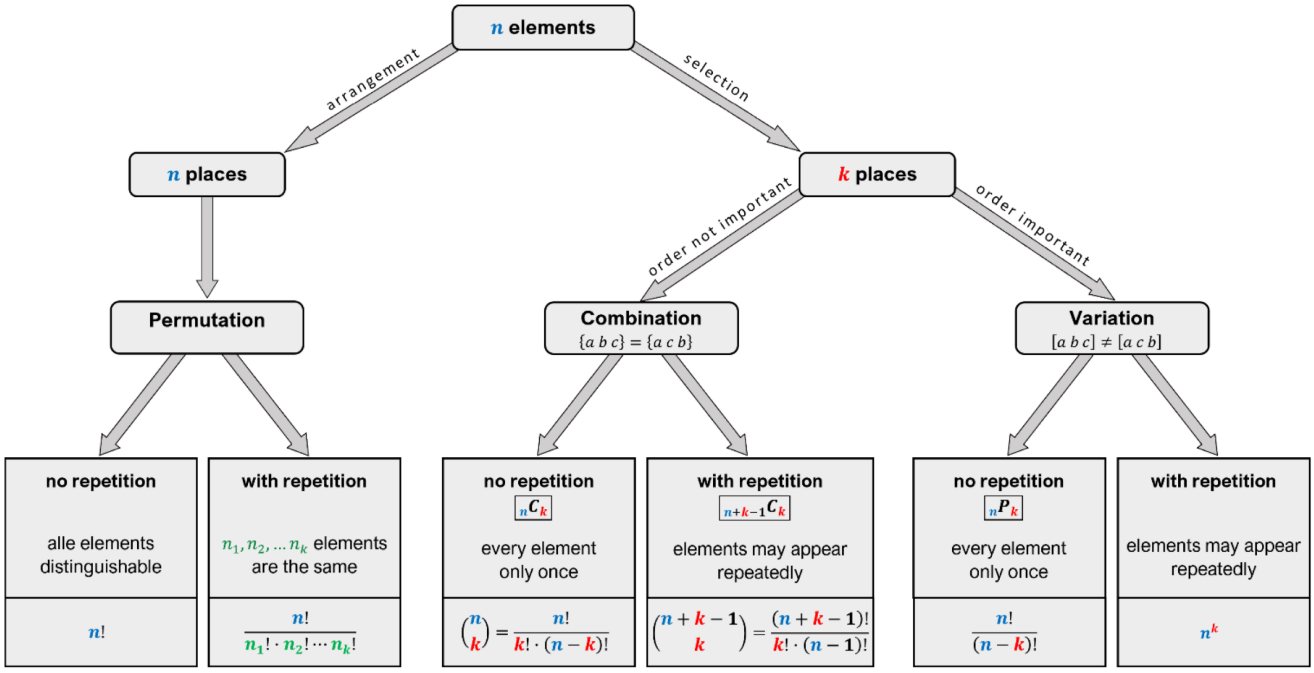
We can calculate the coefficient of each part of the expanded term k with combinatorics as follows: $\binom{n}{k}$

Binomial Expansion **Formula 2.6**

In general:

$$(a+b)^n = 1a^nb^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \ldots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n$$

2.9 Overview


n elements
arrangement
selection
n places
k places
order not important
order important
Permutation
Combination
$\{a\ b\ c\} = \{a\ c\ b\}$
Variation
$[a\ b\ c] \neq [a\ c\ b]$
no repetition
alle elements distinguishable
$n!$
with repetition
$n_1, n_2, \ldots n_k$ elements are the same
$\frac{n!}{n_1! \cdot n_2! \cdots n_k!}$
no repetition
${}_nC_k$
every element only once
$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$
with repetition
${}_{n+k-1}C_k$
elements may appear repeatedly
$\binom{n+k-1}{k} = \frac{(n+k-1)!}{k! \cdot (n-1)!}$
no repetition
${}_nP_k$
every element only once
$\frac{n!}{(n-k)!}$
with repetition
elements may appear repeatedly
n^k

3 Probability

3.1 Basics

Discrete Sample Space — **Definition 3.1**

A sample space S consists of a set Ω consisting of *elementary events* ω_i. Each of these elementary events has a probability assigned to it, such that $0 \leq \Pr[\omega_i] \leq 1$ and

$$\sum_{\omega \in \Omega} \Pr[\omega] = 1$$

We call $E \subseteq \Omega$ an *event*. The probability $\Pr[E]$ of said event is given by

$$\Pr[E] := \sum_{\omega \in E} \Pr[\omega]$$

If E is an event, we call $\overline{E} := \Omega \backslash E$ the *complementary event*

Events — **Lemma 3.2**

For two events A, B, we have:

1. $\Pr[\emptyset] = 0, \Pr[\Omega] = 1$
2. $0 \leq \Pr[A] \leq 1$
3. $\Pr[\overline{A}] = 1 - \Pr[A]$
4. If $A \subseteq B$, we have $\Pr[A] \leq \Pr[B]$

Addition law — **Theorem 3.3**

If events $A_1, \ldots, A_n$ are relatively disjoint (i.e. $\forall (i \neq j) : A_i \cap A_j = \emptyset$), we have (for infinite sets, $n = \infty$)

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i]$$

The below theorem is known as the Inclusion-Exclusion-Principle, or in German the "Siebformel" and is the general case of the addition law, where the events don't have to be disjoint.

Inclusion/Exclusion — Theorem 3.5

Let $A_1, \dots, A_n$ be events, for $n \geq 2$. Then we we have

$$\begin{aligned}\Pr\left[\bigcup_{i=1}^{n} A_i\right] &= \sum_{l=1}^{n}(-1)^{l+1} \sum_{1\leq i_1<\dots<i_l\leq n} \Pr[A_{i_1} \cap \dots \cap A_{i_l}] \\ &= \sum_{i=1}^{n} \Pr[A_i] - \sum_{1\leq i_1<i_2\leq n} \Pr[A_{i_1} \cap A_{i_2}] + \sum_{1\leq i_1<i_2<i_3\leq n} \Pr[A_{i_1} \cap A_{i_2} \cap A_{i_3}] - \dots \\ &\quad + (-1)^{n+1} \cdot \Pr[A_1 \cap \dots \cap A_n]\end{aligned}$$

What is going on here? We add all intersections where an even number of $\cap$-symbols are used and subtract all those who have and odd number of intersections.

Use: This is useful for all kinds of counting operations where some elements occur repeatedly, like counting the number of integers divisible by a list of integers (see Code-Expert Task 04)

Of note here is that we sum up with e.g. $\sum_{1\leq i_1<j_1\leq n} \Pr[A_{i_1} \cap A_{i_2}]$ is all subsets of the whole set Ω, where two events are intersected / added.

If $\Omega = A_1 \cup \dots \cup A_n$ and $\Pr[\omega] = \frac{1}{|\Omega|}$, we get

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{l=1}^{n}(-1)^{l+1} \sum_{1\leq i_1<\dots<i_l\leq n} |A_{i_1} \cap \dots \cap A_{i_l}$$

Since for $n \geq 4$ the Inclusion-Exclusion-Principle formulas become increasingly long and complex, we can use a simple approximation, called the **Union Bound**, also known as the *Boolean inequality*

Union Bound — Corollary 3.6

For events $A_1, \dots, A_n$ we have (for infinite sequences of events, $n = \infty$)

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} \Pr[A_i]$$

Laplace principle: We can assume that all outcomes are equally likely if nothing speaks against it

Therefore, we have $\Pr[\omega] = \dfrac{1}{|\Omega|}$ and for any event E, we get $\Pr[E] = \dfrac{|E|}{|\Omega|}$

3.2 Conditional Probability

Conditional Probability — **Definition 3.8**

Let A, B be events, with $\Pr[B] > 0$. The *conditional probability* $\Pr[A|B]$ of A given B is defined as

$$\Pr[A|B] := \frac{\Pr[A \cap B]}{\Pr[B]}$$

We may also rewrite the above as

$$\Pr[A \cap B] = \Pr[B|A] \cdot \Pr[A] = \Pr[A|B] \cdot \Pr[B]$$

Multiplication law — **Theorem 3.10**

Let $A_1, \ldots, A_n$ be events. If $\Pr[A_1 \cap \ldots \cap A_n] > 0$, we have

$$\Pr[A_1 \cap \ldots \cap A_n] = \Pr[A_1] \cdot \Pr[A_2|A_1] \cdot \Pr[A_3|A_1 \cap A_2] \cdot \ldots \cdot \Pr[A_n|A_1 \cap \ldots \cap A_n]$$

The proof of the above theorem is based on the definition of conditional probability. If we rewrite $\Pr[A_1] = \frac{\Pr[A_1]}{1}$, apply the definition of $\Pr[A_2|A_1]$, and do the same to all subsequent terms, the equation simplifies to $\Pr[A_1 \cap \ldots \cap A_n]$

Use: The law of total probability is used, as the name implies, to calculate the total probability of all possible ways in which an even B can occur.

Law of total probability — **Theorem 3.13**

Let $A_1, \ldots, A_n$ be relatively disjoint events and let $B \subseteq A_1 \cup \ldots \cup A_n$. We then have

$$\Pr[B] = \sum_{i=1}^{n} \Pr[B|A_i] \cdot \Pr[A_i]$$

The same applies for $n = \infty$. Then, $B = \bigcup_{i=1}^{\infty} A_i$

Using the previously defined theorem, we get Bayes' Theorem

Bayes' Theorem — **Theorem 3.15**

Let $A_1, \ldots, A_n$ be relatively disjoint events and let $B \subseteq A_1 \cup \ldots \cup A_n$ be an event with $\Pr[B] > 0$. Then for each $i = 1, \ldots, n$, we have

$$\Pr[A_i|B] = \frac{\Pr[A_i \cap B]}{\Pr[B]} = \frac{\Pr[B|A_i] \cdot \Pr[A_i]}{\sum_{j=1}^{n} \Pr[B|A_j] \cdot \Pr[A_j]}$$

The same applies for $n = \infty$. Then $B = \bigcup_{i=1}^{\infty} A_i$

Use: Bayes' Theorem is commonly used to calculate probabilities on different branches or in other words, to rearrange conditional probabilities. The sum in the denominator represents all posible paths to the event summed up

Example 3.16: Assume we want to find the probability that event X happened given that event Y happened. **Important:** Event X happened *before* event Y happened and we do *not* know the probability of X. Therefore we have $\Pr[X|Y]$ as the probability. But we don't actually know that probability, so we can use Bayes' Theorem to restate the problem in probabilities we can (more) easily determine.

3.3 Independence

Definition 3.18: *(Independence of two events)* Two events A and B are called **independent** if

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$

Independence **Definition 3.22**

Events $A_1, \ldots, A_n$ are called *independent*, if for all subsets $I \subseteq \{1, \ldots, n\}$ with $I = \{i_1, \ldots, i_k\}$ and $|I| = k$, we have that

$$\Pr[A_{i_1} \cap \ldots \cap A_{i_k}] = \Pr[A_{i_1}] \cdot \ldots \cdot \Pr[A_{i_k}]$$

The same in simpler terms: If all events $A_1, \ldots, A_n$ are relatively disjoint, they are independent. We can determine if they are, if the probability of the intersection of all events is simply their individual probabilities multiplied with each other.

Independence **Lemma 3.23**

Events $A_1, \ldots, A_n$ are independent if and only if for all $(s_1, \ldots, s_n) \in \{0,1\}^n$ we have

$$\Pr[A_1^{s_1} \cap \ldots \cap A_n^{s_n}] = \Pr[A_1^{s_1}] \cdot \ldots \cdot \Pr[A_n^{s_n}]$$

where $A_i^0 = \overline{A_i}$ (i.e. $s_i = 0$) and $A_i^1 = A_i$ (i.e. $s_i = 1$)

$\{0,1\}^n$ is the space of n-bit binary numbers, representing subsets of the sample space, each of them being any of the subsets intersected with up to n other subsets

The s_i in this expression are very straight forward to understand as simply indicating if we consider the event or its complement.

Lemma 3.24: *(Let A, B and C be independent events. Then, $A \cap B$ and C as well $A \cup B$ and C are independent)*

In this lecture, we are always going to assume that we can use actual random numbers, not just pseudo random numbers that are generated by PRNGs (Pseudo Random Number Generators).

3.4 Random Variables

Random Variable **Definition 3.25**

A *random variable* is an image $\mathcal{X} : \Omega \to \mathbb{R}$ that maps the sample space to a real number.
The range $W_\mathcal{X} := \mathcal{X}(\Omega) = \{x \in \mathbb{R} : \forall \omega \in \Omega \text{ with } \mathcal{X}(\omega) = x\}$'s countability depends on the countability of Ω, and is either *countable* or *countably infinite*

For those who don't have an intuition for what a random variable actually is: See Section 3.4.3.

Often times when looking at random variables, we are interested in the probabilities at which $\mathcal{X}$ takes certain values. We write either $\mathcal{X}^{-1}(x_i)$ or more intuitively $\mathcal{X} = x_i$. Analogously, we have (short: $\Pr[\text{“}\mathcal{X} \leq x_i''] \text{ as } \Pr[\mathcal{X} \leq x_i]$)

$$\Pr[\text{“}\mathcal{X} \leq x_i''] = \sum_{x \in W_\mathcal{X} : x \leq x_i} \Pr[\text{“}\mathcal{X} = x''] = \Pr[\{\omega \in \Omega : \mathcal{X}(\omega) \leq x_i\}]$$

From this notation, we easily get two real functions. We call $f_\mathcal{X} : \mathbb{R} \to [0,1]$ for which $x \mapsto \Pr[\mathcal{X} = x]$ the ***probability mass function*** (PMF, Dichtefunktion) of $\mathcal{X}$, which maps a real number to the probability that the random variable takes this value.

The ***cumulative distribution function*** (CDF, Verteilungsfunktion) of $\mathcal{X}$ is a function, which maps a real number to the probability that the value taken by the random variable is lower than, or equal to, the real number. Often times it suffices to state the PMF of the random variable (since we can easily derive the CDF from it)

$$F_\mathcal{X} : \mathbb{R} \to [0,1], \quad x \to \Pr[\mathcal{X} \leq x] = \sum_{x' \in W_\mathcal{X} : x' \leq x} \Pr[\mathcal{X} = x'] = \sum_{x' \in W_\mathcal{X} : x' \leq x} f_\mathcal{X}(x')$$

3.4.1 Expected value

Expected Value **Definition 3.27**

The *expected value* $\mathbb{E}[\mathcal{X}]$ describes the average value the random variable $\mathcal{X}$ takes.
We define the *expected value* $\mathbb{E}[\mathcal{X}]$ as

$$\mathbb{E}[\mathcal{X}] := \sum_{x \in W_\mathcal{X}} x \cdot \Pr[\mathcal{X} = x]$$

only if the sum converges absolutely. Otherwise, the *expected value* is undefined. This is trivially true for finite sample spaces.

In this lecture, only random variables with an expected value are covered, so that condition does not need to be checked here

Alternative to the above definition over the elements of the range of the random variable, we can also define it as

Expected Value **Lemma 3.29**

If $\mathcal{X}$ is a random variable, we have

$$\mathbb{E}[\mathcal{X}] = \sum_{\omega \in \Omega} \mathcal{X}(\omega) \cdot \Pr[\omega]$$

If the range of the random variable consists only of non-zero integers, we can calculate the expected value with the following formula

Expected Value **Theorem 3.30**

Let $\mathcal{X}$ be a random variable with $W_\mathcal{X} \subseteq \mathbb{N}_0$. We then have

$$\mathbb{E}[\mathcal{X}] = \sum_{i=1}^{\infty} \Pr[\mathcal{X} \geq i]$$

Conditional Random Variables

Conditional Random Variable — Definition 3.31

Let $\mathcal{X}$ be a random variable and let A be an event with $\Pr[A] > 0$

$$\Pr[(\mathcal{X}|A) \leq x] = \Pr[X \leq x|A] = \frac{\Pr[\{\omega \in A : \mathcal{X}(\omega) \leq x\}]}{\Pr[A]}$$

Expected Value (Conditional) — Theorem 3.32

Let $\mathcal{X}$ be a random variable. For relatively disjoint events $A_1, \ldots, A_n$ with $A_1 \cup \ldots \cup A_n = \Omega$ and $\Pr[A_1], \ldots, \Pr[A_n] > 0$, we have (analogously for $n = \infty$)

$$\mathbb{E}[\mathcal{X}] = \sum_{i=1}^{n} \mathbb{E}[\mathcal{X}|A_i] \cdot \Pr[A_i]$$

Linearity of the expected value

We can calculate the expected value of a sum of any number of random variables $\mathcal{X}_1, \ldots, \mathcal{X}_n : \Omega \to \mathbb{R}$ simply by summing the expected values of each of the random variables $\mathcal{X}_i$

Linearity of expected value — Theorem 3.33

Given random variables $\mathcal{X}_1, \ldots, \mathcal{X}_n$ and let $\mathcal{X} := a_1\mathcal{X}_1 + \ldots + a_n\mathcal{X}_n + b$ for any $a_1, \ldots, a_n, b \in \mathbb{R}$, we have

$$\mathbb{E}[\mathcal{X}] = a_1 \cdot \mathbb{E}[\mathcal{X}_1] + \ldots + a_n \cdot \mathbb{E}[\mathcal{X}_n] + b$$

Very simply with two random variables X and Y, we have $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$

Indicator Variable — Definition 3.35

We use *indicator variables* to formalize the probability that an event A occurs using the expected value
For an event $A \subseteq \Omega$ the accompanying indicator variable $\mathcal{X}_A$ is given by

$$\mathcal{X}_A(\omega) := \begin{cases} 1 & \text{if } \omega \in A \\ 0 & \text{else} \end{cases}$$

For the expected value of $\mathcal{X}_A$ we have: $\mathbb{E}[\mathcal{X}_A] = \Pr[A]$

We can now prove the Inclusion-Exclusion-Principle using a fairly simple proof. See Example 2.36 in the script for it.

Use: We use the indicator variable for experiments where we perform a certain action numerous times where each iteration does not (or does for that matter) depend on the previous outcome.

3.4.2 Variance

Even though two random variables may have the same expected value, they can still be significantly different. The Variance describes the dispersion of the results, or how far off the expected value the different values are maximally (up to a certain limit, that is)

Variance — **Definition 3.39**

For a random variable $\mathcal{X}$ with $\mu = \mathbb{E}[\mathcal{X}]$, the *variance* $\text{Var}[\mathcal{X}]$ is given by

$$\text{Var}[\mathcal{X}] := \mathbb{E}[(\mathcal{X} - \mu)^2] = \sum_{x \in W_\mathcal{X}} (x - \mu)^2 \cdot \Pr[\mathcal{X} = x]$$

$\sigma := \sqrt{\text{Var}[\mathcal{X}]}$ is called the *standard deviation* of $\mathcal{X}$

Variance (easier) — **Theorem 3.40**

For any random variable $\mathcal{X}$ we have

$$\text{Var}[\mathcal{X}] = \mathbb{E}[\mathcal{X}^2] - \mathbb{E}[\mathcal{X}]^2$$

We also have

Variance — **Theorem 3.41**

For any random variable $\mathcal{X}$ and $a, b \in \mathbb{R}$ we have

$$\text{Var}[a \cdot \mathcal{X} + b] = a^2 \cdot \text{Var}[\mathcal{X}]$$

The moments of a random variable are given by the expected value and the variance.

Moment — **Definition 3.42**

The ***k*th moment** of a random variable $\mathcal{X}$ is $\mathbb{E}[\mathcal{X}^k]$ whereas $\mathbb{E}[(\mathcal{X} - \mathbb{E}[\mathcal{X}])^k]$ is called the ***k*th central moment**.

Note The expected value is thus the first moment and the variance the second central moment.

3.4.3 Intuition

If you struggle to imagine what a random variable $\mathcal{X}$ is, or what for example $\mathcal{X}^2$ is, read on. As definition 3.25 states, a random variable is a function, which is why people tend to get confused. It is not a variable in the normal way of understanding.

With that in mind, things like $\mathcal{X}^2$ makes much more sense, as it's simply the result of the function squared, which then makes theorem 3.40 make much more sense, given the definition of the expected value.

Of note is that remembering the summation formulas for the variance (or knowing how to get to it) is handy for the exam, as that formula is not listed on the cheat-sheet provided by the teaching team as of FS25. Deriving it is very easy though, as it's simply applying the expected value definition to the initial definition, which is listed on the cheat-sheet.

3.5 Discrete distribution

3.5.1 Bernoulli-Distribution

A random variable $\mathcal{X}$ with $W_\mathcal{X}$ is called ***Bernoulli distributed*** if and only if its probability mass function is of form

$$f_\mathcal{X}(x) = \begin{cases} p & x = 1 \\ 1-p & x = 0 \\ 0 & \text{else} \end{cases}$$

The parameter p is called the probability of success (Erfolgswahrscheinlichkeit). Bernoulli distribution is used to describe boolean events (that can either occur or not). It is the trivial case of binomial distribution with $n = 1$. If a random variable $\mathcal{X}$ is Bernoulli distributed, we write

$$\mathcal{X} \sim \text{Bernoulli}(p)$$

and we have

$$\mathbb{E}[\mathcal{X}] = p \qquad \text{and} \qquad \text{Var}[\mathcal{X}] = p(1-p)$$

3.5.2 Binomial Distribution

If we perform a Bernoulli trial repeatedly (e.g. we flip a coin n times), the number of times we get one of the outcomes is our random variable $\mathcal{X}$ and it is called ***binomially distributed*** and we write

$$\mathcal{X} \sim \text{Bin}(n, p)$$

and we have

$$\mathbb{E}[\mathcal{X}] = np \qquad \text{and} \qquad \text{Var}[\mathcal{X}] = np(1-p)$$

3.5.3 Geometric Distribution

If we have an experiment that is repeated until we have achieved success, where the probability of success is p, the number of trials (which is described by the random variable $\mathcal{X}$) is ***geometrically distributed***. We write

$$\mathcal{X} \sim \text{Geo}(p)$$

The density function is given by

$$f_\mathcal{X}(i) = \begin{cases} p(1-p)^{i-1} & \text{for } i \in \mathbb{N} \\ 0 & \text{else} \end{cases}$$

whilst the expected value and variance are defined as

$$\mathbb{E}[\mathcal{X}] = \frac{1}{p} \qquad \text{and} \qquad \text{Var}[\mathcal{X}] = \frac{1-p}{p^2}$$

The cumulative distribution function is given by

$$F_\mathcal{X}(n) = \Pr[\mathcal{X} \leq n] = \sum_{i=1}^{n} \Pr[\mathcal{X} = i] = \sum_{i=1}^{n} p(1-p)^{i-1} = 1 - (1-p)^n$$

Note Every trial in the geometric distribution is unaffected by the previous trials

Geometric Distribution **Theorem 3.45**

If $\mathcal{X} \sim \text{Geo}(p)$, for all $s, t \in \mathbb{N}$ we have

$$\Pr[\mathcal{X} \geq s + t | X > s] = \Pr[X \geq t]$$

Coupon Collector problem

First some theory regarding waiting for the nth success. The probability mass function is given by $f_{\mathcal{X}}(x) = \binom{z-1}{n-1} \cdot p^n \cdot (1-p)^{z-n}$ whereas the expected value is given by $\mathbb{E}[\mathcal{X}] = \sum_{i=1}^{n} \mathbb{E}[\mathcal{X}_i] = \frac{n}{p}$

The coupon collector problem is a well known problem where we want to collect all coupons on offer. How many coupons do we need to obtain on average to get one of each? We will assume that the probability of getting coupon i is equal to all other coupons and getting a coupon doesn't depend on what coupons we already have (independence)

Let $\mathcal{X}$ be a random variable representing the number of purchases to the completion of the collection. We split up the time into separate phases, where $\mathcal{X}$ is the number of coupons needed to end phase i, which ends when we have found one of the $n-i+1$ coupons not previously collected (i.e. we got a coupon we haven't gotten yet)

Logically, $\mathcal{X} = \sum_{i=1}^{n} \mathcal{X}_i$. We can already tell from the experiment we are conducting that it is going to be geometrically distributed and thus the probability of success is going to be $p = \frac{n-i+1}{n}$ and we have $\mathbb{E}[\mathcal{X}_i] = \frac{n}{n-i+1}$

With that, let's determine

$$\mathbb{E}[\mathcal{X}] = \sum_{i=1}^{n} \mathbb{E}[\mathcal{X}_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n$$

where $H_n := \sum_{i=1}^{n} \frac{1}{i}$ is the nth harmonic number, which we know (from Analysis) is $H_n = \ln(n) + \mathcal{O}(1)$, thus we have $\mathbb{E}[\mathcal{X}] = n \cdot \ln(n) + \mathcal{O}(n)$.

The idea of the transformation is to reverse the $(n-i+1)$, so counting up instead of down, massively simplifying the sum and then extracting the n and using the result of H_n to fully simplify

3.5.4 Poisson distribution

The ***Poisson distribution*** is applied when there is only a small likelihood that an event occurs, but since the cardinality of the sample space in question is large, we can expect at least a few events to occur. We write

$$\mathcal{X} \sim \text{Po}(\lambda)$$

An example for this would be for a person to be involved in an accident over the next hour. The probability mass function is given by

$$f_{\mathcal{X}}(i) = \begin{cases} \frac{e^{-\lambda}\lambda^i}{i!} & \text{for } i \in \mathbb{N}_o \\ 0 & \text{else} \end{cases} \qquad \text{and} \qquad \mathbb{E}[\mathcal{X}] = \text{Var}[\mathcal{X}] = \lambda$$

Using the Poisson distribution as limit for the binomial distribution

We can approximate the binomial distribution using the Poisson distribution if we have large n and small constant np. $\lambda = \mathbb{E}[\mathcal{X}] = np$ in that case.

3.6 Multiple random variables

There are times when we are interested in the outcomes of multiple random variables simultaneously. For two random variables $\mathcal{X}$ and $\mathcal{Y}$, we evaluate probabilities of type

$$\Pr[\mathcal{X} = x, \mathcal{Y} = y] = \Pr[\{\omega \in \Omega : \mathcal{X}(\omega) = x, \mathcal{Y}(\omega) = y\}]$$

Here $\Pr[\mathcal{X} = x, \mathcal{Y} = y]$ is a shorthand notation for $\Pr[\text{“}\mathcal{X} = x'' \cap \text{“}\mathcal{Y} = y'']$

We define the *common probability mass function* $f_{\mathcal{X},\mathcal{Y}}$ by

$$f_{\mathcal{X},\mathcal{Y}}(x, y) := \Pr[\mathcal{X} = x, \mathcal{Y} = y]$$

We can also get back to the individual probability mass of each random variable

$$f_{\mathcal{X}} = \sum_{y \in W_{\mathcal{Y}}} f_{\mathcal{X},\mathcal{Y}}(x, y) \qquad \text{or} \qquad f_{\mathcal{Y}}(y) = \sum_{x \in W_{\mathcal{X}}} f_{\mathcal{X},\mathcal{Y}}(x, y)$$

We hereby call $f_{\mathcal{X}}$ and $f_{\mathcal{Y}}$ *marginal density* (Randdichte)

We define the ***common cumulative distribution function*** by

$$F_{\mathcal{X},\mathcal{Y}}(x, y) := \Pr[\mathcal{X} \leq x, \mathcal{Y} \leq y] = \Pr[\{\omega \in \Omega : \mathcal{X}(\omega) \leq x, \mathcal{Y} \leq y\}] = \sum_{x' \leq x} \sum_{y' \leq y} f_{\mathcal{X},\mathcal{Y}}(x', y')$$

Again, we can use marginal density

$$F_{\mathcal{X}}(x) = \sum_{x' \leq x} f_{\mathcal{X}}(x') = \sum_{x' \leq x} \sum_{y \in W_{\mathcal{Y}}} f_{\mathcal{X},\mathcal{Y}}(x', y) \quad \text{and} \quad F_{\mathcal{Y}}(y) = \sum_{y' \leq y} f_{\mathcal{Y}}(y') = \sum_{y' \leq y} \sum_{x \in W_{\mathcal{X}}} f_{\mathcal{X},\mathcal{Y}}(x, y')$$

3.6.1 Independence of random variables

Independence — **Definition 3.52**

Random variables $\mathcal{X}_1, \ldots, \mathcal{X}_n$ are called ***independent*** if and only if for all $(x_1, \ldots, x_n) \in W_{\mathcal{X}_1} \times \ldots \times W_{\mathcal{X}_n}$ we have

$$\Pr[\mathcal{X}_1 = x_1, \ldots, \mathcal{X}_n = x_n] = \Pr[\mathcal{X}_1 = x_1] \cdot \ldots \cdot \Pr[\mathcal{X}_n = x_n]$$

Or alternatively, using probability mass functions

$$f_{\mathcal{X}_1,\ldots\mathcal{X}_n}(x_1, \ldots, x_n) = f_{\mathcal{X}_1}(x_1) \cdot \ldots \cdot f_{\mathcal{X}_n}(x_n)$$

In words, this means that for independent random variables, their common density is equal to the product of the individual marginal densities

The following lemma shows that the above doesn't only hold for specific values, but also for sets

Independence — **Lemma 3.53**

Let $\mathcal{X}_1, \ldots, \mathcal{X}_n$ be independent random variables and let $S_1, \ldots, S_n \subseteq \mathbb{R}$ be any set, then we have

$$\Pr[\mathcal{X}_1 \in S_1, \ldots, \mathcal{X}_n \in S_n] = \Pr[\mathcal{X}_1 \in S_1] \cdot \ldots \cdot \Pr[\mathcal{X}_n \in S_n]$$

Independence — **Corollary 3.54**

Let $\mathcal{X}_1, \ldots, \mathcal{X}_n$ be independent random variables and let $I = \{i_1, \ldots, i_k\} \subseteq [n]$, then $\mathcal{X}_{i_1}, \ldots, \mathcal{X}_{i_k}$ are also independent

Independence — **Theorem 3.55**

Let $f_1, \dots, f_n$ be real-valued functions ($f_i : \mathbb{R} \to \mathbb{R}$ for $i = 1, \dots, n$). If the random variables $X_1, \dots, X_n$ are independent, then this also applies to $f_1(\mathcal{X}_1), \dots, f_n(\mathcal{X}_n)$

3.6.2 Composite random variables

Using functions we can combine multiple random variables in a sample space.

Two random variables — **Theorem 3.58**

For two independent random variables $\mathcal{X}$ and $\mathcal{Y}$, let $\mathcal{Z} := \mathcal{X} + \mathcal{Y}$. Then we have

$$f_{\mathcal{Z}}(z) = \sum_{x \in W_{\mathcal{X}}} f_{\mathcal{X}} \cdot f_{\mathcal{Y}}(z - x)$$

We call, analogously to the terms used for power series, $f_{\mathcal{Z}}(z)$ "convolution"

3.6.3 Moments of composite random variables

Linearity of the expected value — **Theorem 3.60**

For random variables $\mathcal{X}_1, \dots, \mathcal{X}_n$ and $\mathcal{X} := a_1\mathcal{X}_1 + \dots + a_n\mathcal{X}_n$ with $a_1, \dots, a_n \in \mathbb{R}$ we have

$$\mathbb{E}[\mathcal{X}] = a_1\mathbb{E}[\mathcal{X}_1] + \dots + a_n\mathbb{E}[\mathcal{X}_n]$$

There are no requirements in terms of independence of the random variables, unlike for the multiplicativity

Multiplicativity of the expected value — **Theorem 3.61**

For independent random variables $\mathcal{X}_1, \dots, \mathcal{X}_n$ we have

$$\mathbb{E}[\mathcal{X}_1 \cdot \dots \cdot \mathcal{X}_n] = \mathbb{E}[\mathcal{X}_1] \cdot \dots \cdot \mathbb{E}[\mathcal{X}_n]$$

Variance of multiple random variables — **Theorem 3.62**

For independent random variables $\mathcal{X}_1, \dots, \mathcal{X}_n$ and $\mathcal{X} = \mathcal{X}_1 + \dots + \mathcal{X}_n$ we have

$$\mathrm{Var}[\mathcal{X}] = \mathrm{Var}[\mathcal{X}_1] + \dots + \mathrm{Var}[\mathcal{X}_n]$$

3.6.4 Wald's Identity

Wald's identity is used for cases where the number of summands is not a constant, commonly for algorithms that repeatedly call subroutines until a certain result is attained. The time complexity of such an algorithm can be approximated by splitting up the algorithm into phases, where each phase is a call of the subroutine. The number of calls to the subroutine, thus the number of phases, is usually not deterministic in that case but rather bound to a random variable.

Wald's Identity — **Theorem 3.65**

Let $\mathcal{N}$ and $\mathcal{X}$ be two independent random variables with $W_{\mathcal{N}} \subseteq \mathbb{N}$. Let

$$\mathcal{Z} := \sum_{i=1}^{\mathcal{N}} \mathcal{X}_i$$

where $\mathcal{X}_1, \mathcal{X}_2, \dots$ are independent copies of $\mathcal{X}$. Then we have

$$\mathbb{E}[\mathcal{Z}] = \mathbb{E}[\mathcal{N}] \cdot \mathbb{E}[\mathcal{X}]$$

3.7 Approximating probabilities

Since it can be very expensive to calculate the true probabilities in some cases, we will now cover some tools that allow us to approximate the probabilities using upper or lower bounds.

3.7.1 Markov's & Chebyshev's inequalities

Markov's inequality — **Theorem 3.67**

Let $\mathcal{X}$ be a random variable that may only take non-negative values. Then for all $t > 0 \in \mathbb{R}$, we have

$$\Pr[\mathcal{X} \geq t] \leq \frac{\mathbb{E}[\mathcal{X}]}{t} \iff \Pr[\mathcal{X} \geq t \cdot \mathbb{E}[\mathcal{X}]] \leq \frac{1}{t}$$

Markov's inequality is fairly straight forward to prove, and it already allows us to make some useful statements, like that for the coupon collector problem, we only need to make more than $100n\log(n)$ purchases with probability $\frac{1}{100}$. The following inequality usually gives a much more precise bound than Markov's inequality

Chebyshev's inequality — **Theorem 3.68**

Let $\mathcal{X}$ be a random variable and $t > 0 \in \mathbb{R}$. Then we have

$$\Pr[|\mathcal{X} - \mathbb{E}[\mathcal{X}| \geq t]] \leq \frac{\mathrm{Var}[\mathcal{X}]}{t^2} \iff \Pr[|\mathcal{X} - \mathbb{E}[\mathcal{X}]| \geq t \cdot \sqrt{\mathrm{Var}[\mathcal{X}]}] \leq \frac{1}{t^2}$$

A common tactic when using these is to restate the original probability $\Pr[X \geq t]$ as $\Pr[|X - \mathbb{E}[X]| \geq t - \mathbb{E}[X]]$ and then set $t = t'$ for $t' = t - \mathbb{E}[X]$

3.7.2 Chernoff bounds

The Chernoff bounds are specifically designed for Bernoulli-variables

Chernoff bounds — **Theorem 3.70**

Let $\mathcal{X}_1, \dots, \mathcal{X}_n$ be independent Bernoulli-distributed random variables with $\Pr[\mathcal{X}_i = 1] = p_i$ and $\Pr[\mathcal{X}_i = 0] = 1 - p_i$. Then we have for $\mathcal{X} := \sum_{i=1}^{n} \mathcal{X}_i$

(i) $\Pr[\mathcal{X} \geq (1+\delta)\mathbb{E}[\mathcal{X}]] \leq e^{-\frac{1}{3}\delta^2\mathbb{E}[\mathcal{X}]}$ for all $0 < \delta \leq 1$

(ii) $\Pr[\mathcal{X} \leq (1-\delta)\mathbb{E}[\mathcal{X}]] \leq e^{-\frac{1}{2}\delta^2\mathbb{E}[\mathcal{X}]}$ for all $0 < \delta \leq 1$

(iii) $\Pr[\mathcal{X} \geq t] \leq 2^{-t}$ for $t \geq 2e\mathbb{E}[\mathcal{X}]$

We determine the δ in the inequality by finding it such that $t = (1+\delta)\mathbb{E}[X]$ or, for the second one, $t = (1-\delta)\mathbb{E}[X]$. For the third one, no δ is required

3.8 Randomized Algorithms

In comparison to *deterministic* algorithms, here, the output is ***not*** guaranteed to be equal for the same input data after reruns. While this can be an issue in some cases, it allows us to usually *significantly* reduce time complexity and thus allows us to solve $\mathcal{NP}$-complete problems in some cases in polynomial time even.

The problem with *true* randomness is that it is hardly attainable inside computers, some kind of predictability will always be there in some form or another, especially if the random number generator is algorithm-based, not on random events from the outside. In this course, we will though assume that random numbers generated by random number generators provided by programming languages actually provide independent random numbers

In the realm of randomized algorithms, one differentiates between two approaches

Types of randomized algorithms		
	Monte-Carlo-Algorithm	**Las-Vegas-Algorithm**
Always correct	✗	✓
Constant runtime	✓	✗

While this is the normal case for Las-Vegas Algorithms, we can also consider the following:

Let the algorithm terminate if a certain runtime bound has been exceeded and return something like "???" if it has not found a correct answer just yet. We will most commonly use this definition of a Las-Vegas-Algorithm in this course

3.8.1 Reduction of error

Error reduction Las-Vegas-Algorithm — **Theorem 3.72**

Let $\mathcal{A}$ be a Las-Vegas-Algorithm, where $\Pr[\mathcal{A}(I)\text{correct}] \geq \varepsilon$
Then, for all $\delta > 0$ we call $\mathcal{A}_\delta$ an algorithm that calls $\mathcal{A}$ until we either get a result that is not "???" or we have executed $N = \varepsilon^{-1} \ln(\delta^{-1})$ times. For $\mathcal{A}_\delta$ we then have

$$\Pr[\mathcal{A}_\delta(I)\text{correct}] \geq 1 - \delta$$

On the other hand, for Monte-Carlo-Algorithms, the probability of error decreases rapidly. It is not easy though to determine whether or not an answer is correct, unless the algorithm only outputs two different values *and* we know that one of these values is *always* correct

Error reduction — **Theorem 3.74**

Let $\mathcal{A}$ be a randomized algorithm, outputting either YES or NO, whereas

$$\begin{aligned} \Pr[\mathcal{A}(I) = \text{YES}] &= 1 \quad \text{if } I \text{ is an instance of YES} \\ \Pr[\mathcal{A}(I) = \text{NO}] &\geq \varepsilon \quad \text{if } I \text{ is an instance of NO} \end{aligned}$$

Then, for all $\delta > 0$ we call $\mathcal{A}_\delta$ the algorithm that calls $\mathcal{A}$ until either NO is returned or until we get YES $N = \varepsilon^{-1} \ln(\delta^{-1})$ times. Then for all instances I we have

$$\Pr[\mathcal{A}_\delta(I)\text{correct}] \geq 1 - \delta$$

This can also be inverted and its usage is very straight forward.

If we however have Monte-Carlo-Algorithms that have two-sided errors, i.e. there is error in both directions, we have

Two-Sided Error reduction — **Theorem 3.75**

Let $\varepsilon > 0$ and $\mathcal{A}$ be a randomized algorithm, that always outputs either Yes or No whereas

$$\Pr[\mathcal{A}(I)\text{correct}] \geq \frac{1}{2} + \varepsilon$$

Then, for all $\delta > 0$ we call $\mathcal{A}_\delta$ the algorithm that calls $\mathcal{A}$ $N = 4\varepsilon^{-2}\ln(\delta^{-1})$ independent times and then returns the largest number of equal responses. Then we have

$$\Pr[\mathcal{A}_\delta(I)\text{correct}] \geq 1 - \delta$$

For randomized algorithms for optimization problems like the calculation of a largest possible stable set, as seen in Example 2.37 in the script, we usually only consider if they achieve the desired outcome.

Optimization problem algorithms — **Theorem 3.76**

Let $\varepsilon > 0$ and $\mathcal{A}$ be a randomized algorithm for a maximization problem, for which

$$\Pr[\mathcal{A}(I) \geq f(I)] \geq \varepsilon$$

Then, for all $\delta > 0$ we call $\mathcal{A}_\delta$ the algorithm that calls $\mathcal{A}$ $N = 4\varepsilon^{-2}\ln(\delta^{-1})$ independent times and then returns the *best* result. Then we have

$$\Pr[\mathcal{A}_\delta(I) \geq f(I)] \geq 1 - \delta$$

For minimization problems, analogously, we can replace $\geq f(I)$ with $\leq f(I)$

3.8.2 Sorting and selecting

The QuickSort algorithm is a well-known example of a Las-Vegas algorithm. It is one of the algorithms that *always* sorts correctly, but its runtime depends on the selection of the pivot elements, which happens randomly.

QuickSort — **Recall**

As covered in the Algorithms & Data Structures lecture, here are some important facts

- Time complexity: $\Omega(n\log(n))$, $\Theta(n\log(n))$, $\mathcal{O}(n^2)$
- Performance is dependent on the selection of the pivot (the closer to the middle the better, but not in relation to its current location, but rather to its value)
- In the algorithm below, *ordering* refers to the operation where all elements lower than the pivot element are moved to the left and all larger than it to the right of it.

Algorithm 9 QuickSort

```
1 procedure QuickSort(A, l, r)
2     if l < r then
3         p ← Uniform({l, l+1, ..., r})      ▷ Choose pivot element randomly
4         t ← Partition(A, l, r, p)          ▷ Return index of pivot element (after ordering)
5         QuickSort(A, l, t−1)               ▷ Sort to the left of pivot
6         QuickSort(A, t, r)                 ▷ Sort to the right of pivot
```

We call $\mathcal{T}_{i,j}$ the random variable describing the number of comparisons executed during the execution of QuickSort(A, l, r). To prove that the average case of time complexity in fact is $\Theta(n\log(n))$, we need to show that

$$\mathbb{E}[\mathcal{T}_{i,j}] \leq 2(n+1)\ln(n) + \mathcal{O}(n)$$

which can be achieved using a the linearity of the expected value and an induction proof. (Script: p. 154)

Selection problem

For this problem, we want to find the k-th smallest value in a sequence $A[1], \dots, A[n]$. An easy option would be to simply sort the sequence and then return the k-th element of the sorted array. The only problem: $\mathcal{O}(n \log(n))$ is the time complexity of sorting.

Now, the QuickSelect algorithm can solve that problem in $\mathcal{O}(n)$

Algorithm 10 QuickSelect

```
1 procedure QuickSelect(A, l, r, k)
2     p ← Uniform({l, l+1, ..., r})          ▷ Choose pivot element randomly
3     t ← Partition(A, l, r, p)
4     if t = l + k − 1 then
5         return A[t]                          ▷ Found element searched for
6     else if t > l + k − 1 then
7         return QuickSelect(A, l, t − 1, k)   ▷ Searched element is to the left
8     else
9         return QuickSelect(A, t + 1, r, k − t)   ▷ Searched element is to the right
```

3.8.3 Primality test

Deterministically testing for primality is very expensive if we use a simple algorithm, namely $\mathcal{O}(\sqrt{n})$. There are nowadays deterministic algorithms that can achieve this in polynomial time, but they are very complex.

Thus, randomized algorithms to the rescue, as they are much easier to implement and also much faster. With the right precautions, they can also be very accurate, see theorem 2.74 for example.

A simple randomized algorithm would be to randomly pick a number on the interval $[2, \sqrt{n}]$ and checking if that number is a divisor of n. The problem: The probability that we find a *certificate* for the composition of n is very low ($\mathcal{O}\left(\frac{1}{n}\right)$). Looking back at modular arithmetic in Discrete Maths, we find a solution to the problem:

Fermat's little theorem **Theorem 3.77**

If $n \in \mathbb{N}$ is prime, for all numbers $0 < a < n$ we have

$$a^{n-1} \equiv 1 \mod n$$

Using exponentiation by squaring, we can calculate $a^{n-1} \mod n$ in $\mathcal{O}\left(k^3\right)$.

Algorithm 11 Miller-Rabin-Primality-Test

```
1  procedure Miller-Rabin-Primality-Test(n)
2      if n = 2 then
3          return true
4      else if n even or n = 1 then
5          return false
6      Choose a ∈ {2, 3, ..., n − 1} randomly
7      Calculate k, d ∈ ℤ with n − 1 = d2^k and d odd   ▷ See below for how to do that
8      x ← a^d mod n
9      if x = 1 or x = n − 1 then
10         return true
11     while not repeated more than k − 1 times do   ▷ Repeat k − 1 times
12         x ← x^2 mod n
13         if x = 1 then
14             return false
15         if x = n − 1 then
16             return true
17     return false
```

This algorithm has time complexity $\mathcal{O}(\ln(n))$. If n is prime, the algorithm always returns `true`. If n is composed, the algorithm returns `false` with probability at least $\frac{3}{4}$.

Notes We can determine $k, d \in \mathbb{Z}$ with $n - 1 = d2^k$ and d odd easily using the following algorithm

Algorithm 12 Get d and k easily

```
1 k ← 1
2 d ← n − 1
3 while d is even do
4     d ← d/2
5     k ← k + 1
```

What we are doing here is removing the all even divisors from d, to make it odd.

3.8.4 Target-Shooting

The Target-Shooting problem is the following: Given a set U and a subset thereof $S \subseteq U$, whose cardinality is unknown, how large is the quotient $\frac{|S|}{|U|}$. We define an indicator variable for S by $I_S : U \to \{0, 1\}$, where $I_S(u) = 1$ if and only if $u \in S$

The Target-Shooting algorithm approximates the above quotient:

Algorithm 13 Target-Shooting

```
1 procedure TargetShooting(N, U)
2     Choose u_1, ..., u_N ∈ U randomly, uniformly and independently
3     return N^{-1} · Σ_{i=1}^{N} I_S(u_i)
```

For this algorithm, two assumptions have to be made:

- I_S has to be efficiently computable
- We need an efficient procedure to choose a uniformly random element from the set U.

Target-Shooting **Theorem 3.78**

Let $\delta, \varepsilon > 0$. If $N \geq 3\frac{|U|}{|S|} \cdot \varepsilon^{-2} \cdot \ln\left(\frac{2}{\delta}\right)$, the output of Target-Shooting is, with probability *at least* $1 - \delta$, on the Interval $\left[(1-\varepsilon)\frac{|S|}{|U|}, (1+\varepsilon)\frac{|S|}{|U|}\right]$

3.8.5 Finding dulicates

Deterministically, this could be achieved using a HashMap or the like, iterating over all items, hashing them and checking if the hash is available in the map. This though uses significant amounts of extra memory and is also computationally expensive. A cheaper option (in terms of memory and time complexity) is to use a *Bloomfilter*.

The randomized algorithm also use Hash-Functions, but are not relevant for the exam.

Hash Function **Definition 3.79**

If we have $\mathcal{S} \subseteq U$, i.e. our dataset $\mathcal{S}$ is subset of a *universe* U, a *hash function* is an image $h : U \to [m]$, whereas $[m] = \{1, \ldots, m\}$ and m is the number of available memory cells. It is assumed that we can *efficiently* calculate a hash for an element and that all elements are randomly distributed, i.e. we have for $u \in U$ $\Pr[h(u) = i] = \frac{1}{m}$ for all $i \in [m]$

4 Algorithms

4.1 Graph algorithms

4.1.1 Long path problem

Given a tuple (G, B) where G is a graph and $B \in \mathbb{N}_0$, we need to determine whether there exists a path of length B in G.

This problem is another one of the infamous $\mathcal{NP}$-Complete problems, for which there *supposedly* doesn't exist an algorithm that can solve the problem in polynomial time. We can show that this problem belongs to said group if we can show that we can *efficiently* construct a graph G' with $n' \leq 2n-2$ vertices such that G has a Hamiltonian Cycle if and only if G' has a path of length n.

We construct a graph G' from graph G by selecting a vertex v and replacing each edge incident to that vertex with edges that lead to newly added vertices $\widehat{w_1}, \widehat{w_2}, \ldots$. G' has $(n-1) + \deg(v) \leq 2n-2$ vertices. All the vertices $\widehat{w_1}, \ldots, \widehat{w_{\deg(v)}}$ all have degree 1.

The graph G' fulfills the above implication because

(i) Let $\langle v_1, v_2, \ldots, v_n, v_1 \rangle$ be a Hamiltonian cycle. Assume for contradiction that the resulting graph, constructed according to the description above does not contain a path of length n. Let's assume $v_1 = v$ (the vertex removed during construction). However, $\langle \widehat{v_2}, v_2, \ldots, \widehat{v_n}, v_n \rangle$ is a path of length n

(ii) Let $\langle u_0, u_1, \ldots, u_n \rangle$ be a path of length n in G' and let $\deg(u_i) \geq 2 \ \ \forall i \in \{1, \ldots, n-1\}$ These vertices hence have to be the $n-1$ remaining vertices of G, thus we have $u_0 = \widehat{w_i}$ and $u_n = \widehat{w_j}$ two different ones of new vertices of degree 1 in G'. Thus, we have $u_1 = w_i$ and $u_{n-1} = w_j$ and we have $\langle v, u_1, \ldots, u_{n-1}, v \rangle$, which is a Hamiltonian cycle in G

Due to the construction of the graph G' we can generate it from G in $\mathcal{O}\left(n^2\right)$ steps. We thus have:

Long Path Problem **Theorem 4.1**

If we can find a *long-path* in a graph with n vertices in time $t(n)$, we can decide if a graph with n vertices has a Hamiltonian cycle in $t(2n-2) + \mathcal{O}\left(n^2\right)$

Short long paths

In biological applications, the long paths searched are usually small compared to n. It is possible to solve this problem in polynomial time if for the tuple (G, B) $B = \log(n)$.

Notation and useful properties:

- $[n] := \{1, 2, \ldots, n\}$. $[n]^k$ is the set of sequences over $[n]$ of length k and we have $\left|[n]^k\right| = n^k$. $\binom{[n]}{k}$ is the set of subsets of $[n]$ of cardinality k and we have $\left|\binom{[n]}{k}\right| = \binom{n}{k}$
- For every graph $G = (V, E)$ we have $\sum_{v \in V} \deg(v) = 2|E|$
- k vertices (no matter if part of a path or not) can be coloured using $[k]$ in exactly k^k ways whereas $k!$ of said colourings use each colour exactly once.
- For $c, n \in \mathbb{R}^+$ we have $c^{\log(n)} = n^{\log(c)}$
- For $n \in \mathbb{N}_0$ we have $\sum_{i=0}^{n} \binom{n}{i} = 2^n$. (Application of binomial expansion, see 2.8)
- For $n \in \mathbb{N}_0$ we have $\frac{n!}{n^n} \geq e^{-n}$
- If we repeat an experiment with probability of success p until success, $\mathbb{E}[\mathcal{X}] = \frac{1}{p}$ where $\mathcal{X} :=$ number of trials

Colourful paths

A path is called *colourful* if all vertices on it are coloured differently. For $v \in V$ and $i \in \mathbb{N}_0$ let's define

$$P_i(v) := \left\{ S \in \binom{[k]}{i+1} \;\middle|\; \exists \text{ path ending in } v \text{ coloured with } S \text{ colours} \right\}$$

Thus, $P_i(v)$ contains a set S of $i+1$ colours if and only if there exists a path with vertex v whose colours are the colours in S. It is important to note that such a path has to be always *exactly* of length i.

If we solve this problem for every $v \in V$, we solved our problem and we have

$$\exists \text{ colourful path of length } k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$$

For the algorithm, we need to also define $N(v)$ which returns the neighbours of v and $\gamma : V \to [k]$ which assigns a colour to each vertex.

Algorithm 14 Colourful path algorithm

```
1 procedure Colourful(G, i)
2     for all v ∈ V do
3         P_i(v) ← ∅
4         for all x ∈ N(v) do
5             for all R ∈ P_{i−1}(x) with γ(v) ∉ R do
6                 P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}}
```

The time complexity of this algorithm is $\mathcal{O}\left(2^k km\right)$. If we now have $k = \mathcal{O}\left(\log(n)\right)$, the algorithm is polynomial.

Random colouring

The idea to solve the short long path problem in polynomial time is to randomly colour the vertices using colours $[k]$, whereby $k := B + 1$ and check if there is a colourful path of length $k-1$. Since we are guaranteed to have a colourful path if we find one, we can develop a Las Vegas Algorithm that solves this problem. But first

Random Colouring — **Theorem 4.2**

Let G be a graph with a path of length $k-1$

(1) We have $p_{\text{success}} = \Pr[\exists \text{ colourful path of length } k-1] \geq \Pr[P \text{ is colourful}] = \frac{k!}{k^k} \geq e^{-k}$

(2) The expected number of trials required to get a colourful path of length $k-1$ is $\frac{1}{p_{\text{success}}} \leq e^k$

For our algorithm we choose a $\lambda > 1 \in \mathbb{R}$ and we repeat the test at most $\lceil \lambda e^k \rceil$. If we succeed once, we abort and output "Yes". If we haven't succeeded in any of the trials, we output "No"

Random Colouring Algorithm — **Theorem 4.3**

- Time complexity: $\mathcal{O}\left(\lambda(2e)^k km\right)$
- If we return "Yes", the graph is *guaranteed* to contain a path of length $k-1$
- If we return "No", the probability of false negative is $e^{-\lambda}$

4.1.2 Flows

Network — **Definition 4.4**

A *Network* is a tuple $N = (\mathcal{V}, \mathcal{A}, c, s, t)$ whereby
- $(\mathcal{V}, \mathcal{A})$ is a directed graph
- $c : \mathcal{A} \to \mathbb{R}_0^+$ the *capacity function*
- $s \in \mathcal{V}$ is the *source*
- $t \in \mathcal{V} \backslash \{s\}$ is the *target*

The capacity function hereby describes the maximum flow through each edge. For each vertex that is not the source or target, the flow is constant, i.e. the total amount entering vertex v has to be equal to the amount exiting it again.

Flow — **Definition 4.5**

Given a network $N = (\mathcal{V}, \mathcal{A}, c, s, t)$, a flow in said network is a function $f : \mathcal{A} \to \mathbb{R}$ where

$$0 \leq f(e) \leq c(e) \quad \forall e \in \mathcal{A} \text{ The acceptability}$$

$$\forall v \in \mathcal{V} \backslash \{s, t\} \sum_{u \in \mathcal{V} : (u,v) \in \mathcal{A}} f(u, v) = \sum_{u \in \mathcal{V} : (v,u) \in \mathcal{A}} f(v, u) \text{ the conservation of flow}$$

The value of a flow f is given by

$$\text{val}(f) := \sum_{u \in \mathcal{V} : (s,u) \in \mathcal{A}} f(s, u) - \sum_{u \in \mathcal{V} : (u,s) \in \mathcal{A}} f(u, s)$$

We call a flow *integeral* if $f(e) \in \mathbb{Z} \quad \forall e \in \mathcal{A}$

Flow — **Lemma 4.6**

The total flow to the target equals the value of the flow, i.e.

$$\text{netinflow}(f) = \text{val}(f) = \sum_{u \in \mathcal{V} : (u,t) \in \mathcal{A}} f(u, t) - \sum_{u \in \mathcal{V} : (t,u) \in \mathcal{A}} f(t, u)$$

s-t-cut — **Definition 4.7**

An s-t-cut of a network $N = (\mathcal{V}, \mathcal{A}, c, s, t)$ is a partition $P = (\mathcal{S}, \mathcal{T})$ of $\mathcal{V}$ (i.e. $\mathcal{S} \cup \mathcal{T} = \mathcal{V}$ and $\mathcal{S} \cap \mathcal{T} = \emptyset$) where $s \in \mathcal{S}$ and $t \in \mathcal{T}$. The capacity of a s-t-cut is then given by

$$\text{cap}(\mathcal{S}, \mathcal{T}) = \sum_{(u,w) \in (\mathcal{S} \times \mathcal{T}) \cap \mathcal{A}} c(u, w)$$

s-t-cut — **Lemma 4.8**

Given f is a flow and $(\mathcal{S}, \mathcal{T})$ a s-t-cut in $N = (\mathcal{V}, \mathcal{A}, c, s, t)$, we have

$$\text{val}(f) \leq \text{cap}(\mathcal{S}, \mathcal{T})$$

Max-flow - min-cut — **Theorem 4.9**

Every network $N = (\mathcal{V}, \mathcal{A}, c, s, t)$ fulfills (f a flow, $(\mathcal{S}, \mathcal{T})$ an s-t-cut)

$$\max_{f \in N} \text{val}(f) = \min_{(\mathcal{S}, \mathcal{T}) \in N} \text{cap}(S, T)$$

It is easier to calculate the min-cut, since there are only a finite number of s-t-cuts (albeit exponentially many, i.e. $2^{|\mathcal{V}|-2}$), thus the importance of the above theorem. It forms the basis of the algorithms that calculate a solution to the max-flow problem.

An approach to solve the max-flow problem is to use augmenting paths, but the issue with that is that it isn't guaranteed that we will find a max flow in a finite number of steps.

Residual capacity network

Using a residual capacity graph also known as a residual network, we can solve the max-flow problem in polynomial time. The concept is simple, yet ingenious: It is simply a network of the remaining capacity (or the exceeded capacity) of an edge $e \in \mathcal{A}$ for a flow f

Residual Network — **Definition 4.10**

Let $N = (\mathcal{V}, \mathcal{A}, c, s, t)$ be a network without bidirectional edges and let f be a flow in said network N. The residual network $N_f = (\mathcal{V}, \mathcal{A}_f, r_f, s, t)$ is given by
(1) If $e \in \mathcal{A}$ with $f(e) < c(e)$, then edge e is also $\in \mathcal{A}_f$ whereas $r_f(e) := c(e) - f(e)$
(2) If $e \in \mathcal{A}$ with $f(e) > 0$ then edge e^{opp} in $\mathcal{A}_f$ whereas $r_f(e^{\text{opp}}) = f(e)$
(3) Only edges as described in (1) and (2) can be found in $\mathcal{A}_f$
We call $r_f(e), e \in \mathcal{A}$ the *residual capacity* of edge e

When reconstructing a network from the residual network, the original network is given by:

- The capacity of an edge (u, v) in the original network is the value of (u, v) and (v, u) in the residual network added (if applicable), where (u, v) is directed *towards* the target.
- The flow of an edge is a bit more complicated: An edge that might appear to intuitively not be part of the flow may be and vice-versa. Flow preservation is the key: The same amount of fluid has to enter each vertex (that is not s or t) has to also exit it again. To check if an edge is part of the flow, simply see if the start vertex of the edge has an excess of fluid.
- If just one edge is present, the edge is flipped, if two are present, figure out which edge was part of the flow and which one is the remaining capacity.

Note: A vertex in the residual network directed towards the *target* is usually the remaining capacity!

Maximum Flow — **Theorem 4.11**

A flow f in a network $N = (\mathcal{V}, \mathcal{A}, c, s, t)$ is a maximum flow if and only if there does not exist a directed path between the source and target of the residual network.
For every such maximum flow there exists a s-t-cut with $\text{val}(f) = \text{cap}(S, T)$

Algorithms

Most algorithms for the max-flow problem use a residual network, where $\mathcal{A}_f$ is the edge-set in the residual network.

Algorithm 15 Ford-Fulkerson

```
1 procedure Ford-Fulkerson(V, A, c, s, t)
2     f ← 0                                   ▷ Flow is constantly 0
3     while ∃s-t-path P in (V, A_f) do        ▷ Augmenting path
4         Increase flow along P by minimum residual capacity in P
5     return f                                ▷ Maximum flow
```

The problem with this algorithm is that it may not terminate for irrational capacities. If we however only consider integral networks without bidirectional edges, it can be easily seen that if we denote $U \in \mathbb{N}$ the upper bound for capacities, the time complexity of this algorithm is $\mathcal{O}(nUm)$ where $\mathcal{O}(m)$ is the time complexity for constructing residual network.

Max-Flow Algorithm — **Theorem 4.12**

If in a network without bidirectional edges and all capacities integral and no larger than U, there is an integral max-flow we can compute in $\mathcal{O}(mnU)$, whereas m is the number of edges and n the number of vertices in the network.

There are more advanced algorithms than this one that can calculate solutions to this problem faster or also for irrational numbers. For the following two proposition, $m = |E|$ and $n = |V|$, i.e. m is the number of edges and n the number of vertices

Capacity-Scaling — **Proposition 4.13**

If in a network all capacities are integral and at most U, there exists an integral max-flow that can be computed in $\mathcal{O}(mn(1 + \log(U)))$

Dynamic-Trees — **Proposition 4.14**

The max-flow of a flow in a network can be calculated in $\mathcal{O}(mn \log(n))$

Bipartite Matching as Flow-Problem

We can use the concepts of flows to determine matchings in bipartite graphs.

Let $G = (V, E)$ be a bipartite graph, i.e. $\exists$ Partition $(\mathcal{U}, \mathcal{W})$ of V such that $E = \{\{u, w\} \mid u \in \mathcal{U}, w \in \mathcal{W}\}$. We construct a network $N = (V \dot\cup \{s, t\}, \mathcal{A}, c, s, t)$, i.e. to the vertices of G, we added a source and a target. The capacity function is $c(e) = 1$. We copy the edges from G, having all edges be directed ones from vertices in $\mathcal{U}$ to ones in $\mathcal{W}$. We add edges from the source s to every vertex in $\mathcal{U}$ and from every vertex in $\mathcal{W}$ to the target t.

Bipartite Matching - Max-Flow — **Lemma 4.15**

The maximum matching in a bipartite graph G is equal to the maximum flow in the network N as described above

Edge- and Vertex-disjoint paths

We can determine the degree of the connectivity of a graph (i.e. the number of vertices / edges that have to be removed that the graph becomes disconnected) by determining how many edge- or vertex-disjoint paths exist between two vertices. Again, using max-flow, we can solve this problem as follows:

Given an undirected graph $G = (V, E)$ and two vertices $u, v \in V : u \neq v$, we need to define our network $N = (\mathcal{V}, \mathcal{A}, c, s, t)$:

- Copy the vertex set V and union it with two new vertices s and t and we thus have $\mathcal{V} = V \cup \{s, t\}$
- Add two edges for each undirected edge in G, i.e. $\mathcal{A} = E \cup E'$ where E' is has all edge directions reversed
- We define the capacity function as $c(e) = 1$
- We add two edges (s, u) and (v, t) and set the capacity of these edges to $|V|$. These are the two vertices between which we evaluate the edge-disjoint paths

If instead of edge-disjoint paths, we want to find *vertex*-disjoint paths, we simply replace each vertex $x \in V \backslash \{u, v\}$ by x_{in} and x_{out} and connect all input-edges to x_{in} and all output-edges of x to x_{out}

Image segmentation

We can also use cuts to solve image segmentation, i.e. to split background from foreground. We can translate an image to an undirected graph, since every pixel has four neighbours. Whatever the pixel values mean in the end, we assume we can deduce two non-negative numbers α_p and β_p denoting the probability that p is in the foreground or background respectively.

Since this topic looks to not be too relevant for the exam, a full explanation of this topic can be found in the script on page 186-189

Flows and convex sets

From the definition of flows we have seen, there is always *at least* one flow, the flow $\mathbf{0}$.

Flows — **Lemma 4.16**

Let f_0 and f_1 be flows in a network N and let $\lambda \in \mathbb{R} : 0 < \lambda < 1$, then the flow f_λ given by

$$\forall e \in \mathcal{A} : f_\lambda(e) := (1 - \lambda) f_0(e) + \lambda f_1(e)$$

is also a flow in N. We have

$$\text{val}(f_\lambda) = (1 - \lambda) \cdot \text{val}(f_0) + \lambda \cdot \text{val}(f_1)$$

Number of flows in networks — **Corollary 4.17**

(i) A network N has either exactly *one* flow (the flow $\mathbf{0}$) or infinitely many flows
(ii) A network N has either exactly *one* maximum flow or infinitely many maximum flows

Convex sets We define a function $f : \mathcal{A} \to \mathbb{R}$ that induces a vector $v_f := (f(e_1), f(e_2), \dots, f(e_m)) \in \mathbb{R}^m$ whereas $e_1, \dots, e_m$ is an ordering of the vertices of $\mathcal{A}$ where $m = |\mathcal{A}|$. We can interpret the set of (maximum) flows as a subset of $\mathbb{R}^m$

Convex set **Definition 4.18**

Let $m \in \mathbb{N}$

(i) For $v_0, v_1 \in \mathbb{R}^m$ let $\overline{v_0 v_1} := \{(1 - \lambda v_0) + \lambda v_1 \mid \lambda \in \mathbb{R}, 0 \leq \lambda \leq 1\}$ be the *line segment* connecting v_0 and v_1

(ii) A set $\mathcal{C} \subseteq \mathbb{R}^m$ is called *convex* if for all $v_0, v_1 \in \mathcal{C}$ the whole line segment $\overline{v_0 v_1}$ is in $\mathcal{C}$

Examples: Spheres or convex Polytopes (e.g. dice or tetrahedra in $\mathbb{R}^3$)

Convex sets **Theorem 4.19**

The set of flows of a network with m edges, interpreted as vectors is a convex subset of $\mathbb{R}^m$. The set of all maximum flows equally forms a convex subset of $\mathbb{R}^m$

4.1.3 Min-Cuts in graphs

In the following section we use *multigraphs*.

Multigraph **Recall**

A multigraph is an undirected, unweighted and acyclic graph $G = (V, E)$, where multiple edges are allowed to exist between the same pair of vertices.
(Instead of multiple edges, we could also allow weighted edges, but the algorithms and concepts presented here are more easily understandable using multiple edges)

Min-Cut Problem

We define $\mu(G)$ to be the cardinality of the *min-cut* (this is the problem). This problem is similar to the min-cut problem for flows, only that we have a multigraph now. We can however replace multiple edges with a single, weighted edge, allowing us to use the algorithms discussed above. Since we need to compute $(n-1)$ s-t-cuts, our total time complexity is $\mathcal{O}\left(n^4 \log(n)\right)$, since we can compute s-t-cuts in $\mathcal{O}\left(n^3 \log(n)\right) = \mathcal{O}\left(n \cdot m \log(n)\right)$

Edge contraction

Let $e = \{u, v\}$ be an edge of our usual multigraph G. The *contraction of* e replaces the two vertices u and v with a single vertex denoted $x_{u,v}$, which is incident to all edges any of the two vertices it replaced were incident to, apart from the ones between the two vertices u and v. We call the new graph G/e and $\deg_{G/e}(x_{u,v}) = \deg_G(u) + \deg_G(v) - 2k$ where k denotes the number of edges between u and v.

Of note is that there is a bijection: Edges in G without the ones between u and $v \leftrightarrow$ Edges in G/e

Edge contraction **Lemma 4.20**

Let G be a graph and e be an edge of G. Then we have that $\mu(G/e) \geq \mu(G)$ and we have equality if G contains a min-cut $\mathcal{C}$ with $e \notin \mathcal{C}$.

Random edge contraction

Algorithm 16 Random Cut where G is a connected Multigraph

```
1 procedure Cut(G)
2     while |V(G)| > 2 do                ▷ Vertices of G
3         e ← uniformly random edge in G
4         G ← G/e
5     return Size of a unique cut of G
```

If we assume that we can perform edge contraction in $\mathcal{O}(n)$ and we can choose a uniformly random edge in G in $\mathcal{O}(n)$ as well, it is evident that we can compute $\textsc{Cut}(G)$ in $\mathcal{O}\left(n^2\right)$

Random edge contraction **Lemma 4.21**

If e is uniformly randomly chosen from the edges of multigraph G, then we have

$$\Pr[\mu(G) = \mu(G/e)] \geq 1 - \frac{2}{n}$$

Correctness of Cut(G) — **Lemma 4.22**

To evaluate the correctness of Cut(G), we define

$$\widehat{p}(G) := \text{Probability that Cut}(G) \text{ returns the value } \mu(G)$$

and let

$$\widehat{p}(n) := \inf_{G=(V,E),|V|=n} \widehat{p}(G)$$

Then, for all $n \geq 3$ we have

$$\widehat{p} \geq \left(1 - \frac{2}{n}\right) \cdot \widehat{p}(n-1)$$

Probability of Correctness of Cut(G) — **Lemma 4.23**

For all $n \geq 2$ we have $\widehat{p}(n) \geq \dfrac{2}{n(n-1)} = \dfrac{1}{\binom{n}{2}}$

Thus, we repeat the algorithm Cut(G) $\lambda\binom{n}{2}$ times for $\lambda > 0$ and we return the smallest value we got.

Cut(G) — **Theorem 4.24**

For the algorithm that runs Cut(G) $\lambda\binom{n}{2}$ times we have the following properties:
(1) Time complexity: $\mathcal{O}\left(\lambda n^4\right)$
(2) The smallest found value is with probability at least $1 - e^{-\lambda}$ equal to $\mu(G)$

If we choose $\lambda = \ln(n)$, we have time complexity $\mathcal{O}\left(n^4 \ln(n)\right)$ with error probability *at most* $\frac{1}{n}$

Of note is that for low n, it will be worth it to simply deterministically determine the min-cut

4.2 Geometric Algorithms

4.2.1 Smallest enclosing circle

Given a set P of n points in a plane, we want to find a circle $C(P)$ such that $C(P)$ encloses all points of P and whose *radius* is minimal (SEC). For a circle C we use $C^\bullet$ for the circular plane enclosed by C, including C and we also allow the points in P to lay on $C(P)$.

Existence of distinct SEC — **Lemma 4.25**

For each (finite) set of points P in $\mathbb{R}^2$ there exists a distinct smallest enclosing circle $C(P)$

The below lemma forms the basis of our algorithm

Subset of set of points — **Lemma 4.26**

For each (finite) set of points P in $\mathbb{R}^2$ with $|P| \geq 3$ there exists a subset $Q \subseteq P$ such that $|Q| = 3$ and $C(Q) = C(P)$

Algorithm 17 Smallest Enclosing Circle $C(P)$

```
1 procedure CompleteEnumeration(P)
2     for all Q ⊆ P with |Q| = 3 do
3         determine C(Q)
4         if P ⊆ C•(Q) then
5             return C(Q)
```

The above algorithm has time complexity $\mathcal{O}\left(n^4\right)$, which is due to having to check at most $\binom{n}{3}$ sets Q, for which we can determine $C(Q)$ in constant time, since $|Q| = 3$. We then have to check for each point in P if it is contained in $C^\bullet(Q)$, which we can do in $\mathcal{O}(n)$, since we can check for each point in constant time.

This is quite inefficient and using a randomized algorithm, we can achieve $\mathcal{O}(n \ln(n))$

Algorithm 18 Smallest Enclosing Circle $C(P)$ randomized

```
1 procedure CompleteEnumerationRandomized(P)
2     while always do
3         select Q ⊆ P with |Q| = 11 uniformly randomly
4         determine C(Q)
5         if P ⊆ C•(Q) then
6             return C(Q)
7         double all points in P outside C(Q)
```

To implement this algorithm, we also need the doubling function: We initialize an array `num` to length n to 1. `num[i]` now contains the number of copies of the i-th point. If the point lays outside $C(Q)$, we can simply set `num[i] = 2 * num[i]`

To select 11 points uniformly randomly in $\mathcal{O}(n)$, consider

Uniformly random Q — **Lemma 4.27**

Let $n_1, \dots, n_t \in \mathbb{N}$ and $N := \sum_{i=1}^{t} n_i$. We generate $X \in \{1, \dots, t\}$ randomly as seen in Algorithm 19
Then we have $\Pr[X = i] = \frac{n_i}{N}$ for all $i = 1, \dots, t$

Algorithm 19 Choose uniformly random number

```
1 procedure ChooseUniformlyRandom(t)
2     UniformInt(1, N)
3     x ← 1
4     while ∑_{i=1}^{x} n_i < k do
5         x ← x + 1
6     return x
```

Uniformly random Q — **Lemma 4.28**

Let P be a set of n not necessarily different points and for $r \in \mathbb{N}$ let R be uniformly randomly chosen from $\binom{P}{r}$. Then, the expected number of points of P that are outside $C(R)$ is at most $3\frac{n-r}{r+1} \leq 3\frac{n}{r+1}$

Correctness of CompleteEnumerationRandomized — **Theorem 4.29**

The algorithm CompleteEnumerationRandomized computes the smallest enclosing circle of P in time $\mathcal{O}(n \log(n))$

Sampling Lemma

Sampling Lemma helpers — **Definition 4.30**

Given a finite set S, $n := |S|$ and ϕ be an arbitrary function on 2^S in an arbitrary range. We define

$$V(R) = V_\phi(R) := \{s \in S \mid \phi(R) \cup \{s\} \neq \phi(R)\}$$
$$X(R) = X_\phi(R) := \{s \in S \mid \phi(R) \backslash \{s\} \neq \phi(R)\}$$

We call elements in $V(R)$ *violators of* R and elements in $X(R)$ we *call external in* R

The sampling lemma connects the expected number of violating elements to the number of external elements.

Sampling Lemma — **Lemma 4.31**

Let $k \in \mathbb{N}, 0 \leq k \leq n$. We set $v_k := \mathbb{E}[|V(R)|]$ and $x_k := \mathbb{E}[|X(R)|]$ where R is a subset of S of cardinality k, uniformly randomly selected from $\binom{s}{k}$. Then we have for $r \in \mathbb{N}, 0 \leq r < n$

$$\frac{v_r}{n-r} = \frac{x_{r+1}}{r+1}$$

Expected numbers / ranks — **Corollary 4.32**

If we choose r elements R from a set A of n numbers randomly, the expected rank of the minimum of R in A is exactly $\frac{n-r}{r+1} + 1 = \frac{n+1}{r+1}$.
If we choose r points R of a set P of n points in a plane randomly, the expected number of points of P that are outside $C(R)$ is at most $3\frac{n-r}{r+1}$

4.2.2 Convex hull

Convex hull **Definition 4.33**

Let $S \subseteq \mathbb{R}^d, d \in \mathbb{N}$. The *convex hull*, $\text{conv}(S)$ of S is the cut of all convex sets that contains S, i.e.

$$\text{conv}(S) := \bigcap_{S \subseteq C \subseteq \mathbb{R}^d} C$$

where C is convex

A convex hull is always convex again.

JarvisWrap **Theorem 4.37**

The `JarvisWrap` algorithm can find the convex hull in $\mathcal{O}(nh)$, where h is the number of corners of P and P is a set of n points in any position in $\mathbb{R}^2$.

Algorithm 20 JarvisWrap

```
1  procedure FindNext(q)
2      Choose p_0 ∈ P\{q} arbitrarily
3      q_next ← p_0
4      for all p ∈ P\{q, p_0} do
5          if p is to right of qq_next then q_next ← p        ▷ qq_next is vector from q to q_next
6      return q_next
7  procedure JarvisWrap(P)
8      h ← 0
9      p_now ← point in P with lowest x-coordinate
10     while p_now ≠ q_0 do
11         q_h ← p_now
12         p_now ← FindNext(q_h)
13         h ← h + 1
14     return (q_0, q_1, ..., q_{h-1})
```

LocalRepair

Algorithm 21 LocalRepair

```
1  procedure LocalRepair(p_1, ..., p_n)
2      q_0 ← q_1                                   ▷ input sorted according to x-coordinate
3      h ← 0
4      for i ∈ {2, ..., n} do                      ▷ Lower convex hull, left to right
5          while h > 0 and q_h is to the left of q_{h-1}p_i do   ▷ q_{h-1}p_i is a vector
6              h ← h − 1
7          h ← h + 1
8          q_h ← p_i                               ▷ (q_0, ..., q_h) is lower convex hull of {p_1, ..., p_i}
9      h' ← h
10     for i ∈ {n − 1, ..., 1} do                  ▷ Upper convex hull, right to left
11         while h > h' and q_h is to the left of q_{h-1}p_i do   ▷ q_{h-1}p_i is a vector
12             h ← h − 1
13         h ← h + 1
14         q_h ← p_i
15     return (q_0, ..., q_{h-1})                 ▷ The corners of the convex hull, counterclockwise
```

LocalRepair **Theorem 4.42**

The `LocalRepair` algorithm can find the convex hull of a (in respect to the x-coordinate of each point) sorted set P of n points in $\mathbb{R}^2$ in $\mathcal{O}(n)$.

The idea of the `LocalRepair` algorithm is to repeatedly correct mistakes in the originally randomly chosen polygon. The improvement steps add edges to the polygon such that eventually all vertices lay withing the smallest enclosing circle of the polygon.

5 Coding

5.1 Tricks

- Encoding subsets is very easy using bitshift operators. We can use `(i & (1 << j)) != 0` in combination with a loop to go through all subsets of the set. The for-loop will have to go from $i = 1$ (if we want to exclude the empty set) to $i = 2^k - 1$
- Inclusion-Excluison is very powerful
- Always check the what the input values are and expect them to provide bad code (e.g. provided code reads ints, even though we need to read doubles (or long)). An easy way to check if everything is correct, is to print the data import's results and compare with the input files if something looks incorrect
- DP (as always) can come in very handy for solving probabilities related problems