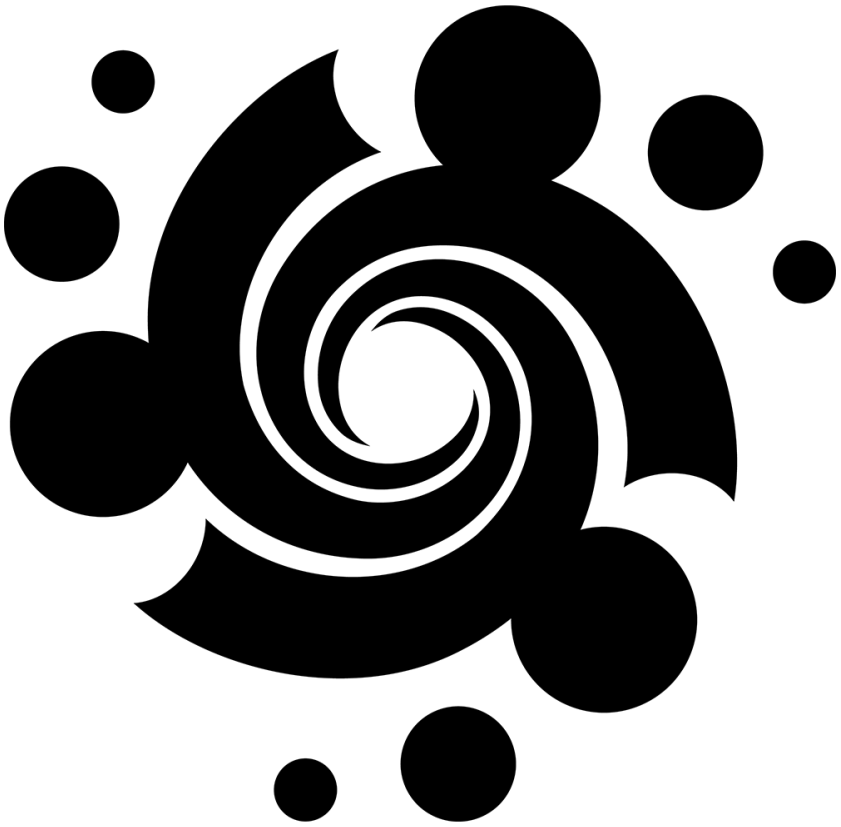
ETH Zurich

Department of Computer Science

Algorithms and Probability PVK Skript

Author:
Marc Himmelberger, Soel Micheletti, Simone Guggiari



Preface

This script is a summary of the ETH Course *Algorithmen und Wahrscheinlichkeit*. Some topics are more presented more in depth than others, but we hope you'll get a good overview about all the important concepts taught in the course.

This script only serves as additional material for practice purposes and should not serve as a substitute for the lecture material. We neither guarantee that this script covers all relevant topics for the exam, nor that it is always correct. If an attentive reader finds any mistakes or has any suggestions on how to improve the scrips, they are encouraged to contact the authors under anw-pvw-skript@vis.ethz.ch or, preferably, through a gitlab issue on https://gitlab.ethz.ch/vis/luk/pvw_script_anw.

Contents

Graph Theory 1

In the course *Algorithms and Data Structures* last semester, you had a first encounter with graphs: you have seen the definition, some properties and algorithms such as BFS, DFS, Topological Sorting, algorithms for shortest paths... In this course we go deeper in the topic, and we introduce other exciting concepts. Some of them are exposed in this chapter, others that exploit randomization are presented in Chapter 3.

1.1 Recap from Algorithms and Data Structures

Concepts and Notation

Graphs are a powerful mathematical tool that allows us to model many different problems. The main idea is to define a collection of elements called *nodes* or *vertices* (Knoten), and a relation between them that we call the *edges* (Kanten). Edges can be directed or undirected, yielding directed or undirected graphs respectively. An example could be to model locations as nodes and the streets between them as edges, or people and the relationship between them (*e.g.* "is the father of" or "are friends"). Both nodes and edges can also have additional information associated with them such as color, location, length, capacity, flow and more.

If the underlying relationship used for the edges is symmetric (meaning that v is connected to u if and only if also u is connected to v) then the graph is said to be *undirected* (ungerichtet).

Definition 1.1.1 (Undirected Graph) *An undirected graph is a tuple* $G = (V, E)$ *where*
$V = \{v_1, ..., v_n\}, |V| = n$ *is the set of nodes, and*
$E = \{e_1, ..., e_m\} \subseteq \{\{u, v\} | u, v \in V\}, |E| = m$ *is the set of edges between them.*

We call vertices u, v *adjacent* (benachbart) if and only if $u, v \in E$ and we call a vertex v and an edge e *incident* (inzident) if and only if $v \in e$.

If however there is the possiblity of having v in relation to u without u being in relation to v (*e.g.* father-of relationship) then the graph is called *directed* (gerichtet). The direction is represented by an arrow. We use some different letters to make the distinction easier.

Definition 1.1.2 (Directed Graph) *A directed graph is a tuple* $D = (V, A)$ *where*
$V = \{v_1, ..., v_n\}, |V| = n$ *is the set of nodes, and*
$A = \{e_1, ..., e_m\} \subseteq V \times V, |E| = m$ *is the set of edges between them.*

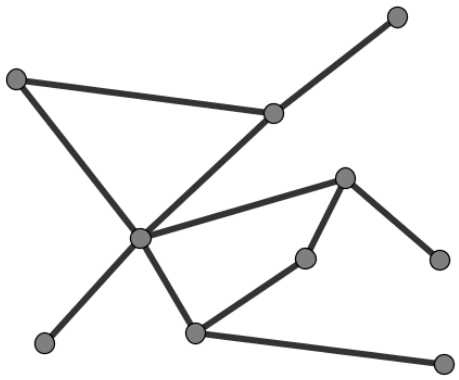


Figure 1.1: Simple undirected graph

Important Graphs

Some graphs are used a lot and deserve their own name and symbol:

- A *Complete Graph* (vollständiger Graph) K_n has n vertices and every pair of different vertices is connected by an edge.
- A *Circle* (Kreis) C_n is a graph that has n vertices that are connected in one big circle. (A connected graph that has exactly once cycle through any pair of different edges)
- A *Path* (Pfad) P_n is C_n, but with any one edge removed.
- A *Hypercube* (Hyperwürfel) of dimension d, called Q_d, is a graph with $V = \{0,1\}^d$ (bitstrings of length d) and edges between exactly those vertices that differ at exactly one position.

It's important to remember these, as they can often serve as counterexamples, even in small sizes.

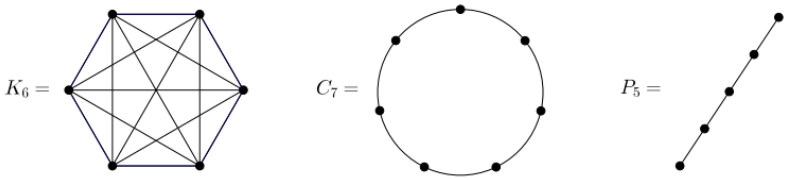


Figure 1.2: The graphs K_6, C_7 and P_5

Bipartite Graphs

If our graph can be split into two sets of vertices such that there are no edges within those sets then the graph is said to be *bipartite* (bipartit). (This definition is easily extended to k *independent sets* (stabile Mengen) to yield *k-partite* Graphs). Formally:

$V = A \cup B$, two disjoint sets of nodes ($A \cap B = \varnothing$)
$E \subseteq \{\{u, w\} | u \in U, w \in W\}$ (undirected bipartite graph)
$E \subseteq (U \times W) \cup (W \times U)$ (directed bipartite graph)

Sequences

We can select an ordered sequence of vertices, and based on edges between them and repetition of edges/vertices in our sequence, we

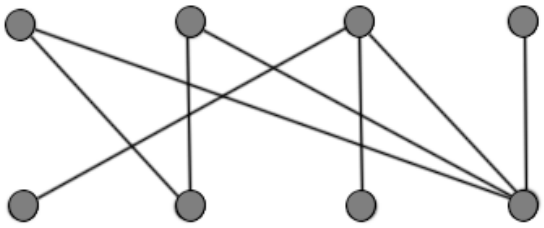


Figure 1.3: Simple bipartite graph

can distinguish them as follows (note that this is not consistent in the literature, we simply refer to the definitions in the lecture):

Let's consider the sequence $\langle v_1, v_2, \ldots, v_k \rangle$

- *Walk* (Weg): a sequence with edges between v_i and v_{i+1} for all $i \in \{1, \ldots, k-1\}$. The length is $k-1$.
- *Path* (Pfad): a walk without repeated vertices.
- *Closed Walk* (Zyklus): a walk where $v_1 = v_k$
- *Cycle* (Kreis): a closed walk where all vertices except start and end are different
- *Loop* (Schleife): a sequence $\langle v_i, v_i \rangle$.
  If nothing else is mentioned, we only consider graphs without loops and without multiple edges between the same vertices (note that strictly speaking, our definitions already enforce that).

Here it's a handy table that summarize these concepts:

Restrictions	Name	Name when closed
any vertices	Walk (Weg)	Closed Walk (Zyklus)
distinct vertices	Path (Pfad)	Cycle (Kreis)

A path starting at a given vertex s and ending at another given vertex t is called an *s-t-path* (s-t-Pfad).

Degree

The *degree* (Grad) of a vertex v of a graph is the number of edges incident to the vertex (with loops counted twice). It is denoted $deg(v) := |N_G(v)|$, where $N_G(v)$ is the neighborhood of v (all vertices adjacent to v).
For directed graphs, we have the *in-degree* (Eingangsgrad) $\deg_G^-(v)$ and the *out-degree* (Ausgangsgrad) $\deg_G^+(v)$ which are defined as the number of incoming and outgoing edges respectively.

Theorem 1.1.1

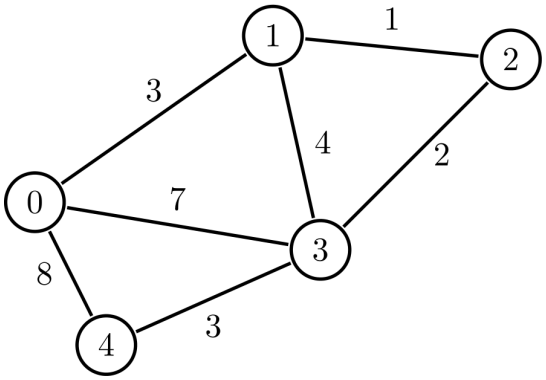
$$\sum_{v \in V} \deg(v) = 2|E|$$

This formula implies the following:

- the amount of vertices with odd degree is even
- the average degree in a graph is $\frac{2m}{n}$

Graph Data Structures

Until now, a graph $G = (V, E)$ is an abstract data type that can be represented as follows (ignore the weights on the edges, but we'll use the numbers on each node to refer to that node):



But how can we represent a graph in a computer? How can you code algorithms that works with graphs? In order to do this, you need a *data structure* to represent the graph. Here there are the three most popular solutions.

Adjacency matrix We simply create a matrix with $|V|$ rows and $|V|$ columns. The entry at position (i, j) (counting from zero here) has value 0 if there is no edge between nodes i and j, and it has value 1 if there is an edge. In the case of the previous image we would have the following adjacency matrix (again, ignoring weights):

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Note that if the graph is undirected, the matrix is symmetric, *i.e.* $A^T = A$. You don't need to know a lot of runtimes for this course, and most depend on how exactly the data structures are implemented. Important to know are:

- **Memory space / time to create:** $\mathcal{O}(|V|^2)$
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(1)$
- **Are u and v adjacent?** $\mathcal{O}(1)$

Adjacency list We create a list of lists. The big list has an entry list for every node (again, a list). The list corresponding to the node v contains all edges that are incident to v.
For example, in the case of the previous image, we would have the following adjacency list:

$$\{\{\{0,1\},\{0,3\},\{0,4\}\},\{\{1,0\},\{1,3\},\{1,2\}\},\{\{2,1\},$$
$$\{2,2\}\},\{\{3,2\},\{3,1\},\{3,0\},\{3,4\}\},\{\{4,0\},\{4,3\}\}\}$$

Here, the important runtimes look as follows (assuming edges don't have references to their copies):

- **Memory space / time to create:** $\mathcal{O}(|V| + |E|)$
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge** $\{u, v\}$**:** $\mathcal{O}(deg(u) + deg(v))$
- **Are** u **and** v **adjacent?** $\mathcal{O}(\min(deg(u), deg(v)))$

Obviously, if you have a complete graph (*i.e.* every node is connected to all other nodes), the runtimes of many operations become the same as the ones for adjacency matrices. The intuition behind using adjacency matrices is that the efficiency of many operations is inversely proportional to the number of edges (which, you should remember, is $\leq |V|^2$ in the graphs you consider in this course): if the graph has no edges, the operations are very efficient, but if the graph has many edges, the operations have the same efficiency they would have in adjacency matrices.

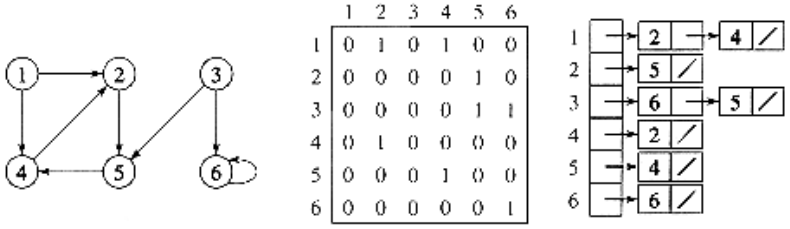


Figure 1.4: different ways of storing a graph. Adj. Matrix and Adj. List

Trees

A *tree* (Baum) T is a graph on n vertices that satisfies the following properties:

- it is connected
- it has no cycles
- it has $n - 1$ edges

Note that if T satisfies two of these properties, it automatically also satisfies the third.

In this context, a *leaf* (Blatt) is a vertex with degree 1.

Trees also have a few simple properties. (Let T be a tree with $n \geq 2$)

- T contains at least two leaves.
- by removing a leaf from T, we obtain a graph G' which is also a tree.
- between any two nodes u, v, there is exactly one u-v-path in T. (any graph that satisfies this is also a tree!)

1.2 Minimum Spanning Tree

Given a graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}$ that assigns a cost $c(e)$ to each edge in G, a minimum spanning tree (MST) T over G is a tree nodes V and edges $E(T)$, such that the sum

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

is minimized. This means that among all possible complete trees over G, T minimizes the total cost. Spanning Trees have many applications, for example in networking, where they are applied to prevent forwarding network packets in cycles while maintaining connectivity.

1.3 Advanced Graph Concepts

Definition 1.3.1 *A graph is* k-connected *(k-zusammenhängend) if and only if*

- $|V| \geq k + 1$, *and*
- $\forall X \subseteq V, |X| < k$: $G[V \setminus X]$ *is connected*

Definition 1.3.2 *A graph* k-edges-connected *(k-kanten-zusammenhängend) if and only if*

- $|E| \geq k + 1$
- $\forall X \subseteq E, |X| < k$: $(V, E \setminus X)$ *is connected*

Informally: The graph has at least $k + 1$ vertices/edges, and we need to remove at least k vertices/edges to make the graph disconnected.

Theorem 1.3.1 (Menger) *A graph is* k-connected *if and only if for all $u, v \in V, u \neq v$ there exist at least k internally-vertex-disjoint u-v-paths. (only u, v may appear on multiple paths)*

A graph is k-edge-connected *if and only if for all $u, v \in V, u \neq v$ there exist at least k edge-disjoint u-v-paths. (no edges may appear on multiple paths)*

Since there are k paths that do not share vertices/edges, we cannot disconnect the graph by removing any $k - 1$ vertices/edges, which implies that the graph is k-(edge)-connected.

Single nodes and edges that disconnect the graph (therefore with $k = 1$) are called *Articulation points* (Artikulationsknoten) and *Bridges* (Brücken) respectively.

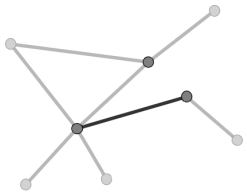


Figure 1.5: One Bridge and three articulation points marked darker in the graph (there are more bridges)

Finding articulation points and bridges efficiently We can use the following to find articulation points and bridges for each connected component of the graph. Note that the search tree we will use now is a directed graph.

- perform DFS on G starting at an arbitrary vertex s and assign a number $dfs[v]$ to each vertex v according to the order they were visited (the first vertex gets 0, the next 1, etc.)
- compute for each vertex v $low[v]$ = minimum dfs number of all vertices reachable from v via arbitrarily many edges in the search tree and at most one other edge that is in the graph (use DP)
- Profit:
  - v is an articulation point if and only if
    - $v = s$ and s has degree ≥ 2 in T, or
    - $v \neq s$ and $\exists w \in V$ with $\{v, w\} \in E(T)$: low$[w] \geq$ dfs$[v]$.

    (So the root is an articulation point if DFS was only able to explore part of the graph before having to return, and the other vertices are articulation points if one of their children in the search tree has a low number at least as big as their dfs number)
  - $\{u, v\}$ is a bridge if and only if $(u, v) \in E(T) \wedge$ low$[v] >$ dfs$[u]$ or $(v, u) \in E(T) \wedge$ low$[u] >$ dfs$[v]$ (note the different orientations of the edge in the search tree).
    (So an edge can only be a bridge if it's part of the search tree and the parent node has a strictly lower dfs number than the low number of the child)

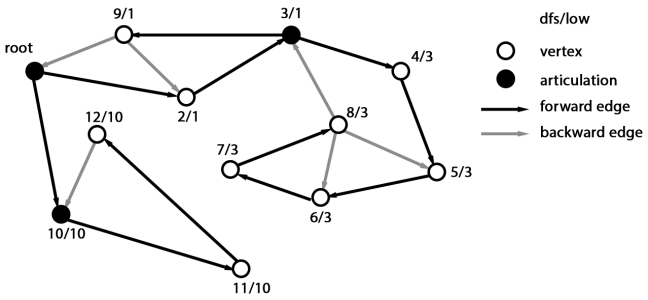


Figure 1.6: A possible execution of this algorithm. Dark edges are in the search tree, light edges are in the original graph but were not used by DFS, and dark vertices are articulations. The notation x/y indicates the *dfs* and *low* numbers, respectively.

Both articulation points and bridges can be found in $\mathcal{O}(|E|)$ after DFS (which takes a further $\mathcal{O}(|V| + |E|)$) with this algorithm.
The full algorithm can be found in the lecture script on page 35.

1.4 Blocks and the Block-Cut Tree

This section offers a possibly less confusing approach to the problem of finding bridges and articulation points than above.

Given two edges e, f, we say that they are in the same *block* if and only if there exists a cycle (Kreis) that contains both of them. This gives us the set of all blocks B.

There might be some vertices that are incident to edges in different blocks (you could say the blocks "overlap"), these are exactly the articulation

points. This way, we can easily find all the set of articulation points P given the blocks B.

Theorem 1.4.1 *Let $G = (V, E)$ be a graph.*

A vertex is an articulation point in G if and only if it is incident to at least two edges that are in different blocks of G.

An edge is a bridge if and only if it is the only edge in its block.

Finally, given a graph $G = (V, E)$, we can create its *block-cut tree* (Block-Zerlegung) as a graph with vertices $B \cup P$ where a block is connected to an articulation point if and only if that point is incident to an edge in that block (and there are no edges between blocks or between articulation points).
Though this does not help us with finding bridges or articulation points, it can be useful for various divide-and-conquer methods.

All of this is possible in linear time, *i.e.* $\mathcal{O}(|V| + |E|)$.

1.5 Matchings

A set of edges $M \subseteq E$ is called *matching* in a graph $G = (V, E)$ if no vertex in the graph is incident to more than an edge in M. Formally:

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

We call all the vertices that are incident to an edge from M *covered* (überdeckt) by the matching.

"as big as possible" If we are interested in computing a matching M that is "as big as possible", we can mean two things:

- A matching M is *maximal* (inklusions-maximal) if and only if no edge in $E \setminus M$ can be added to the matching without violating the definition of a matching.
  It's easy to find, *e.g.* with *greedy* strategy by iterating over all the edges (and picking only those who have both vertices still free) in time $\mathcal{O}(|E|)$.
- A matching M is *maximum* (kardinalitäts-maximal) if and only if there is no matching in the graph that contains more edges.
  The algorithm by Hopcroft & Karp computes a maximum matching in $\mathcal{O}(\sqrt{|V|}(|V| + |E|))$.

For any maximal matching M and any maximum matching M^*, it holds that

$$|M| \leq |M^*| \wedge |M| \geq 1/2|M^*|$$

Perfect Matching

A matching M_p is called *perfect* (perfekt) if and only if $|M_p| = \frac{|V|}{2}$, *i.e.* every vertex of the graph is incident to exactly one edge of the matching.

Not every graph possesses a perfect matching (*e.g.* no perfect matching can exist if there is an odd number of vertices).

Matchings in Regular Graphs If $G = (V, E)$ is a 2^k-regular bipartite graph, then we can find a perfect matching in $\mathcal{O}(|E|)$.

If $G = (A \uplus B, E)$ is a k-regular bipartite graph, then there exist $M_1, \ldots, M_k$ such that $E = M_1 \uplus \ldots \uplus M_k$ and all M_i with $1 \leq i \leq k$ are perfect matchings.

Augmenting Paths

A path in $G = (V, E)$ is called an *augmenting path* (augmentierender Pfad) for a given matching M if and only if exactly every second edge in the path is in M, starting and ending with vertices not covered by M. This means that an augmenting path has odd length.

We can use an augmenting path to increase the size of our matching by 1 if we remove all the edges along the path from M and add the ones previously not contained. (in a way "swapping" them)

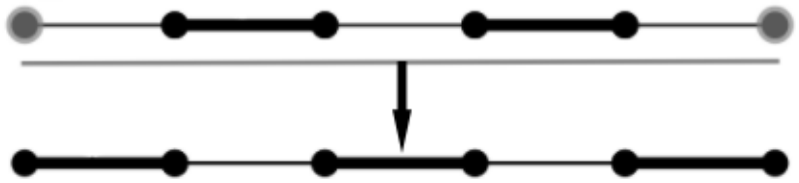


Figure 1.7: Augmenting path once the edges are swapped

Hopcroft-Karp Algorithm

This algorithm works on a bipartite graph $G = (A \uplus B, E)$. This has the advantage that an augmenting path can be found in $\mathcal{O}(n + m)$ simply using a slightly modified BFS.
The algorithm works by finding all the augmenting paths that have minimal length, selecting an inclusion-maximal set of vertex-disjoint paths and augmenting along all of those at once. This repeats until there are no more augmenting paths, implying that the matching is maximum.

The runtime is $\mathcal{O}(\sqrt{|V|}(|V| + |E|))$.

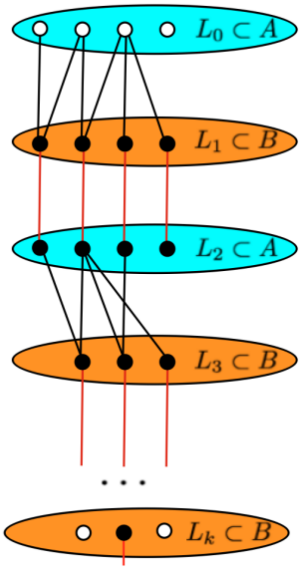


Figure 1.8: The layer structure produced by BFS. Black edges are not in the matching, red ones are. White vertices are not covered by the matching, black ones are.

Theorem 1.5.1 (Berge) *Let M be a matching in the graph $G = (V, E)$.*

M is a maximum matching in G if and only if there exists no M-augmenting path in G.

Blossom's Algorithm

This one works for any graph. It repeatedly finds augmenting paths and uses those to increase the size of the matching until no more augmenting path exists (meaning the matching is maximum).
You don't need to know how it finds augmenting paths, just know that it has runtime $\mathcal{O}(|E| \cdot |V|^2)$.

Hall's Marriage Theorem

Necessary and sufficient condition to have a matching that covers at least one side of the bipartite graph.

Theorem 1.5.2 (Hall) *Let $G = (A \cup B, E)$ be a bipartite graph and let $N(X)$ denote the neighbourhood of $X \subseteq A \cup B$ in G (all vertices in $A \cup B$ adjacent to at least one vertex in X).*

There exists a matching that covers all vertices in A if and only if

$$\forall W \subseteq A : |W| \leq |N(W)|$$

Informally Every subset W has enough adjacent vertices in B. Otherwise, we could not pair up the vertices in W with different vertices in B.

1.6 Eulerian Tour

An *Eulerian Tour* (Eulertour) is a closed walk that visits every edge in the graph exactly once.
Graphs that contain an Eulerian tour are called *Eulerian* (eulersch).

Theorem 1.6.1 *A connected graph $G = (V, E)$ is Eulerian if and only if all vertices in G have even degree.*

One algorithm to find an Eulerian tour is the following:
Imagine two runners, a fast and a slow one. They start at an arbitrary vertex. First, the fast runner starts and moves from vertex to vertex, randomly choosing which neighbor to visit next (but not using the same edge twice) until the runner arrives at the start.
The fast runner has now traversed a closed walk. Only now, the slow runner starts and moves one step along the closed walk we just found. At the new vertex, we check if there are any edges that the fast runner has not yet traversed. If so, we send the fast runner off again along one of these unvisited edges. When the fast runner returns, we will have a new closed walk that contains none of the edges we already found. We merge this new walk into the first one by inserting it at the current vertex (think of it as a detour).
Now, the slow runner makes another step along the newly merged path, checks for unvisited edges and sends off the fast runner again if it finds any and so on. The algorithm terminates once we arrive back at the vertex we started at in the very beginning.

This algorithm finds an Eulerian tour in $\mathcal{O}(|E|)$. An implementation is given below (the fast runner works like RandomTour and the slow runner like EulerTour)

```
1 W ← RandomTour(G, v_start)
2 v_slow ← successor of v_start in W
3 while v_slow is not equal to v_start do
4     if v_slow has unvisited incident edges
5         W' ← RandomTour(G, v_slow)
6         Merge W' into W
7         // We have W = W_1 + ⟨v⟩ + W_2 and create W = W_1 + W' + W_2
8     v_slow ← successor of v_slow in W
```

Algorithm 1.1: EulerTour(G,v)

```
1 W ← ⟨v⟩
2 v_init ← v
3 while v is not v_init do
4     Choose an arbitrary unvisited edge {v, u} incident to v
5     Add u to W
6     Mark {v, u} as visited
7     v ← u
8 return W
```

Algorithm 1.2: RandomTour(G,v)

1.7 Hamiltonian Cycles

In the previous chapter we studied Eulerian Tours, and we have seen a linear time algorithm to find an Eulerian Tour. In this chapter, we study an apparently similar problem: finding a cycle in a graph that uses each vertex exactly once. Although this might seem just a variant of the previous problem, this is very different and more difficult. In facts, determining whether a graph contains a Hamiltonian cycle is an NP-complete problem. This means that, given a possible solution, we can check in polynomial time whether it is correct or not. However, it is not known whether a polynomial time algorithm for finding such a solution exists (and it's not likely).
In the script, there is a dynamic programming approach that solves the problem in exponential time, but we don't present it here. However, we state some important results about special cases of this problem:

- A grid graph with m rows and n columns contains a Hamiltonian cycle if and only if $m \cdot n$ is even.
- A bipartite graph with partitions A and B can only contain a Hamiltonian cycle if $|A| = |B|$.
- Hypercubes of any dimension $d \geq 1$ contain a Hamiltonian cycle.
- Dirac's theorem: every graph $G = (V, E)$ with $|V| \geq 3$ and minimal degree $\geq |V|/2$ contains a Hamiltonian cycle.

1.8 Travelling Salesman Problem

The *travelling salesman problem* (TSP) is a famous generalization of the Hamiltonian cycle problem. A businessman wants to visit n cities, each exactly once, by starting from their own city and returning in the end. He knows the time it takes to travel between every pair of cities, and he wants to pick the shortest cycle possible. Formally: we are given a complete graph K_n and a function $\ell : \binom{[n]}{2} \to \mathbb{N}_0$ that assigns a cost to each edge. We want to find a Hamiltonian cycle C in K_n with $\sum_{e \in C} \ell(e)$ minimal.

The problem is of course at least as hard as determining whether a graph contains a Hamiltonian cycle. In fact, if we had an algorithm to find the optimal tour, we could decide whether a graph G contains a Hamiltonian cycle in the following way: we build a complete graph with the same number of vertices as G and we give cost zero to the edges in G and cost one to the others. If and only if the optimal tour has weight zero, then G contains a Hamiltonian cycle.

In general, in the context of optimization problems such as TSP, one could be happy about finding a solution that is not worse than α times the value of the optimal solution.
But this is still not feasible: If we had such an algorithm, we could determine whether a graph contains a Hamiltonian cycle just as before (because α time zero, *i.e.* the value of the optimal solution if the graph contains a Hamiltonian cycle, is still zero). Hence, finding an α-approximation algorithm for TSP has to be at least as hard as finding a Hamiltonian cycle.

For this reason, we introduce a more restricted version of the TSP that can still be useful: the *metric TSP* (metrisches TSP). The setting is the same as before, but now we also require that

$$\ell(\{x,z\}) \leq \ell(\{x,y\}) + \ell(\{y,z\}) \quad \text{for all } x,y,z$$

This property is called the triangle inequality, and it expresses that going from x to z directly is never longer than taking a detour over y.

2-Approximation Algorithm for the Metric TSP

In this section, we explain an algorithm that finds an answer to the metric TSP that not more than twice as costly as the optimal solution. The runtime of the algorithm is $\mathcal{O}(n^2)$. The algorithms work as follows:

1. We compute an MST in the graph in $\mathcal{O}(n^2)$.
2. We create a multi-graph by doubling every edge in the MST.
3. We find an Eulerian Tour in that multi-graph (which exists because all degrees are even).
4. We now translate this tour in the multi-graph to a cycle in the original graph: We try to use the same sequence of vertices, but whenever we would visit a vertex for the second time, we instead go directly to the next unvisited vertex in the tour.

We can show that the MST has at most the same cost as an optimal solution. Thus, the tour has at most twice that cost, and that's already enough to conclude that this gives us a 2-approximation for the metric TSP in $\mathcal{O}(n^2)$.

1.5-Approximation Algorithm for the Metric TSP

We can do this in exactly the same way, but we improve step 2. The goal there is to end up in a graph where we can find an Eulerian Tour (which is easy) that we can then correct to an answer to the TSP.

We notice that the problem is that we might have some vertices with odd degrees in the MST. But, we also know that there is always an even number of vertices with odd degree in any graph. We can now look at a new complete graph with only these vertices, and there we can find a perfect matching with minimal cost in $\mathcal{O}(n^3)$ (you don't need to know how, this is Satz 1.50 in the lecture script). The new algorithm is:

1. We compute an MST in the graph in $\mathcal{O}(n^2)$.
2. Find a perfect matching with minimal cost in the subgraph containing only the vertices with odd degree. Then add these edges to the MST to obtain a multi-graph where all vertices have even degrees.
3. We find an Eulerian Tour in that multi-graph (which exists because all degrees are even).
4. We now translate this tour in the multi-graph to a cycle in the original graph: We try to use the same sequence of vertices, but whenever we would visit a vertex for the second time, we instead go directly to the next unvisited vertex in the tour.

This is a 1.5-approximation, because the perfect matching we find can at most have half the cost of an optimal solution to the TSP.

1.9 Graph coloring

One can solve a lot of problems by finding a partition of vertices such that edges connect only vertices in different partitions. As an example, consider exam scheduling: We have a graph $G = (V, E)$ where V is the set of exams, and we have an edge $\{u, v\}$ if and only if a student takes both exam u and exam v; hence we can say that edges represent "conflicts" (we call these kinds of graphs "Interference Graphs" or "Interferenz-Graphen"; they are often useful in exams too).
Our goal is to find the minimal number of partitions (here used as time slots) of this graph such that there are no edges within the same partition (an edge within the same partition would mean that two exams with at least one student in common would take place at the same time). Such a partition can then give us an easy way to schedule exams without conflicts.

In general, we define a (vertex) *coloring* (Färbung) of a graph $G = (V, E)$ with k colors as a mapping

$$c : V \to [k]$$

such that $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$. Moreover, we define the *chromatic number* (chromatische Zahl) $\chi(G)$ as the minimum number of colors needed to color G.

An important example is the case where $\chi(G) = 2$, such graphs are **bipartite**. In general, we have:

Theorem 1.9.1 *A graph $G = (V, E)$ is k-partite $\iff$ it can be colored with no more than k colors (i.e. $\chi(G) \leq k$).*

A classical graph coloring problem is the coloring of maps, where neighbouring lands should be assigned to different colors. This problem comes with the assumption that the territory of each land is connected and that lands that touch in a single point can be colored with the same color. An important theoretical result in graph coloring is that every such map can be colored with (at most) four colors.

How can we determine the coloring of a graph with the least number of colors? In order to decide, whether a graph is bipartite or not, a BFS is sufficient, but how can we extend this to larger chromatic numbers? In general, graph coloring is a difficult problem. Already the question *does a given graph $G = (V, E)$ have $\chi(G) \leq 3$?* is NP complete. This means that there is (probably) no polynomial time algorithm that computes the chromatic number of a graph. In practice, this means that we have to find approximations of the optimal solution.

The following algorithm computes a coloring of the vertices of the graph by visiting the vertices in an arbitrary order $v_1, \ldots, v_n$ and assigning to each vertex the lowest color that is not used among its neighbours.

Algorithm 1.3: GreedyColoring(G)

```
1 Choose an arbitrary order of the nodes v_1, ..., v_n
2 for i = 1, ..., n do
3     c(v_i) ← min(ℕ \ {c(u) | u ∈ N(v_i) ∩ {v_1, ..., v_{i-1}})
```

This can be implemented in $\mathcal{O}(|V| + |E|)$.

It is clear that the algorithm returns a valid coloring, because given this construction, the color of a vertex is always different from the one of its neighbours. Now the question is, how many colors does the algorithm use in the worst case? Since the algorithm chooses the smallest color that is not yet used in one of the neighbours, we have the worst case scenario when the neighbours of a vertex v_i are already colored with colors $1, \ldots, deg(v_i)$. In this case, v_i gets the color $deg(v_i) + 1$. This means that the algorithm uses at most $\Delta(G) + 1$ colors, where $\Delta(G)$ is the maximal degree of a node in G. The number of colors used by the algorithm depends on which sequence of vertices we consider. There is always a sequence that yields to the best solution, but since we don't know this sequence we can get a worse result.

Exercise 1. Given a graph $G = (V, E)$ and a coloring c that uses k colors, show how we could construct a sequence of vertices where the greedy algorithm uses $\leq k$ colors.

Then show why this implies that there is a vertex sequence where the greedy algorithm uses exactly $\chi(G)$ colors for any graph.

In general, choosing the correct sequence is not an easy task, and there are various heuristics that could be used to improve our chances. Some additional results are given below:

Theorem 1.9.2 (Brooks) *Given a connected graph that is neither complete nor a circle of odd length, there is an algorithm that computes a sequence with which we will use at most $\Delta(G)$ colors in time $\mathcal{O}(|E|)$.*

Complete graphs and circles of odd length are the only graphs where $\chi(G) = \Delta(G) + 1$ colors are needed (the coloring is trivial). In all other graphs $\chi(G) \leq \Delta(G)$.

By extension:

Theorem 1.9.3 *Let $G = (V, E)$ be a graph and k a natural number such that any induced sub-graph of G has at least one vertex with degree $\leq k$. It holds, $\chi(G) \leq k + 1$ and there is an algorithm that computes a coloring with $k + 1$ colors in $\mathcal{O}(|E|)$*

Proof. Repeatedly remove one of the vertices with maximum degree and use those to fill in the vertex sequence starting at the end. Since the maximum degree is $\leq k$ for all the intermediate graphs, we can always color them with $k + 1$ colors (Brooks) and the next vertex in the sequence may never have more than k neighbors, so it cannot be assigned a color greater than $\Delta(G) + 1$. □

The following theorem shows us that there are also non-complete graphs that need many colors.

Theorem 1.9.4 (Mycielski Construction) *For all $k \geq 2$ there is a triangle-free graph G_k with $\chi(G_k) \geq k$.*

Theorem 1.9.5 *A graph $G = (V, E)$ with $\chi(G) \leq 3$ can be colored in $\mathcal{O}(|E|)$ with $\mathcal{O}(\sqrt{|V|})$ colors.*

The algorithm can be found on page 84 of the lecture script.

Applications We already discussed an example where we were able to use colorings in a conflict graph to find conflict-free schedules of events. Colorings can also be applied to the following scenarios:

- Map coloring (duh)
- Frequency/channel assignment
- Register allocation
- Sudoku (try this on your own or check out this blog post *)

It always comes down to some notion of "conflicts" between entities (expressed as edges) and "groupings" of entities (the color assignments). If this makes sense to you, you should be able to apply this easily to many similar problems.

1.10 Network Flow

Network Definition A *network* (Netzwerk) is a tuple (V, A, c, s, t) with

- $G = (V, A)$ a directed graph
- $c : A \to \mathbb{R}_0^+$ the capacity function
- $s, t \in V, s \neq t$ two special nodes, the *source* (Quelle) and the *sink* (Senke)

This is simply a directed graph with two nodes marked as source and target, and a non-negative value assigned to each edge stating the capacity for transport of that edge.

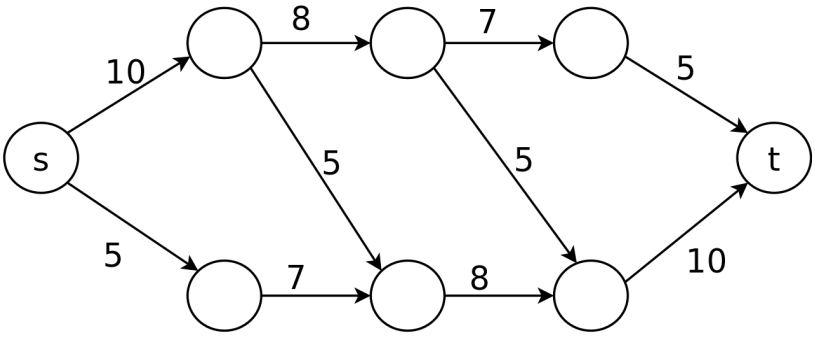


Figure 1.9: Simple network with capacities marked above the edges

* https://medium.com/code-science/sudoku-solver-graph-coloring-8f1b4df47072

Flow Definition A *flow* is a function $f : A \to \mathbb{R}_0^+$ with the following conditions:

- Capacity constraint (Zulässigkeit):

$$\forall e \in A : 0 \leq f(e) \leq c(e)$$

- Flow Conservation (Flusserhaltung):

$$\forall v \in V \setminus \{s, t\} : \sum_{\substack{u \in V \\ (u,v) \in A}} f(u, v) = \sum_{\substack{u \in V \\ (v,u) \in A}} f(v, u)$$

The first condition means that on every edge a non-negative amount of flow not exceeding the capacity of the edge.
The second condition states that no flow disappears or appears at any given node other than the source and sink, *i.e.* flow is conserved in the network.

Inflow and Outflow We can define the following quantities (they already appear above):

- $\text{inflow}(v) := \sum_{\substack{u \in V \\ (u,v) \in A}} f(u, v)$
- $\text{outflow}(v) := \sum_{\substack{u \in V \\ (v,u) \in A}} f(v, u)$

With these quantities we can simplify the second condition to

$$\forall v \in V \setminus \{s, t\} : \text{inflow}(v) = \text{outflow}(v)$$

Flow Value Let's first define two corresponding quantities for s and t:

$$\begin{aligned} \text{netoutflow}(s) &:= \quad \text{outflow}(s) - \text{inflow}(s) \\ \text{netinflow}(t) &:= \quad \text{inflow}(t) - \text{outflow}(t) \end{aligned}$$

It follows from flow conservation that $\text{netoutflow}(s) = \text{netinflow}(t)$.

The *value* (Wert) of a flow is defined as

$$\text{val}(f) := \text{netoutflow}(s)$$

Max-Flow Min-Cut The algorithmic problem we study in this section, is to efficiently compute a maximum flow in the graph, *i.e.* a flow with maximum value. In order to gain some insights about this, we introduce a dual problem.

A $s - t$ cut $C = (S, T)$ in a network is a partition of V into $S, T \subset V$ such that $S \cap T = \varnothing, S \cup T = V$ with $s \in S, t \in T$.
The capacity of this cut is informally how much flow can at most pass from S to T and is defined as

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

Since a $s-t$ cut limits how much flow can move through the network, we know for every $s-t$ cut $C=(S,T)$ and every flow f:

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

Theorem 1.10.1 (Max-Flow Min-Cut) *In any network $N=(V,A,c,s,t)$, there exists a flow f_{max} and a $s-t$ cut $C=(S,T)$ such that*

$$val(f_{max}) = cap(S,T)$$

We can now be sure that, yes, there is a maximum flow in any network and its value is equal to the minimum $S-T$ cut in the graph. We will look to efficient algorithms to compute minimal cuts in Chapter 3, but since there are finitely many cuts in a graph, we already have a naive algorithm to determine a minimal cut (*i.e.* enumerating them and keeping the minimum).

Further (important!): If we find a flow f and a $S-T$ cut with $\text{cap}(S,T) = \text{val}(f)$, this is a proof that f is a maximum flow.

Augmenting Paths The idea of the algorithm that we present here is to improve a given flow. In order to do that, we'll first try to look for a path from s to t in the graph, such that the flow on each edge of the path is strictly less than the capacity of the edge. Then, we can increase the flow on each edge on the path by the quantity $\varepsilon := \min_{e\in \text{ path}} c(e)-f(e)$ without violating capacity constraints or flow conservation.

Unfortunately, there are sub-optimal flows that can not be improved in this way[†]. A key observation here is that increasing the flow on an incoming edge can not only be compensated with an increase of the flow on an outgoing edge, but also with the decrease of the flow of another incoming edge. (instead of sending out more flow, we can accept less and still satisfy flow conservation)

Residual Network We want to fix the problem above by introducing two new concepts: residual networks and augmenting paths (mind you that these have *nothing* to do with augmenting paths for matchings!).

Given a network $N=(V,A,c,s,t)$ with a flow f, we define the *residual network* (Restnetzwerk) $N_f=(V,A',c',s,t)$ as follows:

- For each edge $e\in A$ with $f(e)<c(e)$ in N (not yet at capacity), we add an edge in the same direction with capacity $c'(e)=c(e)-f(e)$ (remaining capacity) to N_f.
- For each edge $e\in A$ with $f(e)>0$ in N (there is flow we could potentially decrease), we add an edge in the opposite direction with capacity $c'(e)=f(e)$ to N_f .

This way, for every edge in N, there are edges in the residual network with their capacities representing the "margins" of the flow over this edge in N.

[†] for an example, you can take a look at https://de.wikipedia.org/wiki/Algorithmus_von_Ford_und_Fulkerson#Beispiel

We can now apply the same idea as before and try to find $s-t$ paths in N_f, and indeed: given such an *augmenting path* we can increase the value of f as follows:
First, calculate $\varepsilon := \min_{e \in \text{ path in } N_f} c'(e)$.
If the augmenting path traverses an edge in N_f in the same direction as it exists in N, we increase the flow over that edge by ε, and
if the augmenting path traverses an edge in N_f in the opposite direction as it exists in N, we decrease the flow over that edge by ε.

Exercise 2. Prove that a flow f' constructed in this way from a flow f given an augmenting path in N_f is again a valid flow (in particular, prove that it satisfies the capacity constraints and flow conservation) and also show that it has a strictly bigger value.

Residual networks are very useful because we can now prove the following:

Theorem 1.10.2 *Let f be a flow in the network N.*

f is a maximum flow in N if and only if there is no directed $s-t$ path in N_f

Ford-Fulkerson Algorithm This algorithm finds a maximal flow in a network by repeatedly finding an augmenting flow in the residual network and using that to increase the flow value. The algorithm is guaranteed to terminate only if the capacities are integers or rational (yielding an integer flow or a rational flow respectively), but not if we allow irrational capacities! ‡

An *integer flow* (ganzzahliger Fluss) is a flow where $f(e) \in \mathbb{N}_0$ for every edge e (and consequently the value of such a flow is also an integer). Similarly, for rational flows.

It works as follows:

```
1 f(e) ← 0 for all e ∈ A
2 while there is an s–t path P in the residual network do
3     increase flow f along P
4 return f
```

Algorithm 1.4: Ford-Fulkerson

In networks with integer capacities and without opposite edges:
Since in every iteration the flow is augmented by at least 1 unit, and the algorithm stops when we have a flow with maximum value (as only then, no augmenting path exists), we know the algorithm will run through a maximum of $\text{val}(f_{max})$ iterations. Each time we must construct the residual network, and this takes $\mathcal{O}(|E|)$. Therefore, the algorithm has a worst case runtime of $\mathcal{O}(\text{val}(f_{max}) \cdot |E|)$. We can give an upper bound for $\text{val}(f_{max})$ as $|V| \cdot U$, where $U = \max_{e \in E} \text{cap}(e)$. This corresponds to the fact that no flow can be higher than the maximum capacity in the network times the amount of vertices (you could use a $s-t$ cut with

‡ A counterexample is given here: https://faculty.math.illinois.edu/~mlavrov/docs/412-spring-2018/infinite-loop.pdf

$S = \{s\}$).
This gives a total runtime of: $\mathcal{O}(U \cdot |V| \cdot |E|)$

For networks with rational capacities, we can simply multiply them all by the largest denominator to get a network with integer capacities.

Useful Tricks Flow problems have many application in the real world, most of which have to do with scheduling the use of a finite amount of resources or find some optimal utilization and transport of quantity in a network. The following tricks allow adapting problems that appear to not fit into this model:

- We can model Multiple sources/sinks by adding a "master" source/sink and connect it to each source/sink that we are given with a big enough capacity (you could again use $|V| \cdot U$)
- We can model undirected graphs by doubling all the edges and making them directed using the original capacity
- We can model capacities at vertices by splitting the vertex into two vertices with only incoming/outgoing edges respectively, and limit the capacity from "incoming" to "outgoing" using a new edge between them.

The following figure should help remember these tricks.

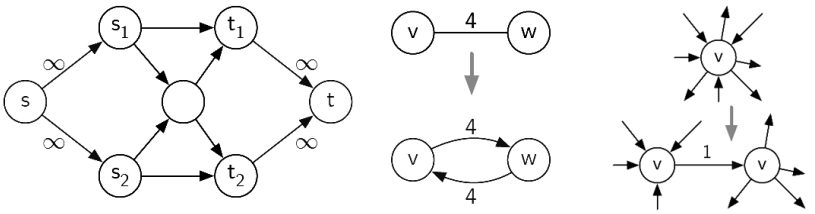


Figure 1.10: The tricks shown graphically

Another interesting observation is that we can compute maximum matchings in bipartite graphs with a network flow approach: we add a source s with edges of capacity 1 to each node of the first partition, and from each node of the other partition we add edges with capacity 1 to the target. Edges of the bipartite graph have capacity one. By following a maximum *integer* flow in this graph, we find a maximum matching in the underlying bipartite graph.

Exercise 3.
1. Sheryl tries to use this algorithm in graphs that are not bipartite as follows: She connects s with capacity 1 to all vertices in the graph, she connects those according to the edges in the graph to a second copy of V and then connects all those vertices in the second copy to t with capacity 1 (so instead of using A and B as described above, she uses V twice).
   Show why this approach does not work (Give an example of a graph where this fails).
2. Show why it's essential to use an integer flow to find the matching. (Give an example of a maximum non-integer flow where it's not clear how we could construct a maximum matching)
   This is one of these things where you can easily lose points on an exam even though you might have a perfectly correct idea!

3. Argue why it's always possible to find a maximum integer flow in such a network (the correct one, not the one from 1.).

Exercise 4. Design an algorithm that, given a graph $G = (V, E)$ and two vertices u, v, finds the number of edge-disjoint $u - v$ paths using network flow. (Describe a suitable network, describe how maximum flow values are related to the amount of disjoint paths and prove that your claim is correct)

Probability Theory 2

In this course, you had your first contact with discrete probability theory, at least with respect to your ETH studies in Computer Science. Probability theory is a wonderful theory that has both mathematical foundations (which you will study in more detail in the course *Wahrscheinlichkeit and Statistik* in the fourth semester) and practical applications (which you have seen in this course, and you will see in other courses at CADMO or in the area of Machine Learning). In the context of this course, the chapter about probability theory can be divided into two fundamental parts: a mathematical approach to the essential concepts of (discrete) probability and an algorithmic approach to some problems that can be solved by exploiting the idea of randomization.

In this chapter, we first introduce notions of probability theory which will be useful in the next chapter, where we will exploit them in the context of randomized algorithms.

2.1 Basic Concepts

When doing a random experiment (flipping a coin, picking a group of 10 people by chance, trying not to lose all of your money in a casino), there is a set of possible outcomes.

Definition 2.1.1 *We describe such an experiment as a* probability space *(Wahrscheinlichkeitsraum). For us, a probability space consists of two parts:*

A discrete sample space *(Ereignisraum)* $\Omega = \{\omega_1, \ldots, \omega_n\}$ *(that is in turn composed of* elementary events *(Elementarereignisse)* $\omega_i \in \Omega$*)*
and a probability function *(Wahrscheinlichkeitsfunktion)* $\Pr : \Omega \to [0, 1]$.

The probability function is a function that measures how "likely" an event is, and it has two fundamental properties:

- $\forall \omega_i \in \Omega : 0 \leq \Pr[\omega_i] \leq 1$
- $\sum_{i=1}^{n} \Pr[\omega_i] = 1$

A set $E \subseteq \Omega$ *is called* event *(Ereignis). The probability* $\Pr[E]$ *is defined as the sum of the elementary events included in* E*. We also define the* complementary event *(Komplementärereignis)* $\bar{E} := \Omega \setminus E$.

It's noteworthy that we usually ignore elementary events with probability 0 in this course.

When trying to figure out what the probability space is, it can be useful to notice that every time you run the experiment, *exactly one* of the elementary events takes place. So if your experiment has multiple "results" (rolling two dice at once, selecting 10 people from 100, shuffling cards) you need to make sure that one experiment results in exactly one element of the sample space being randomly selected. You can sometimes make a

choice about how to best model your experiment and there is not only one correct way, it's just important to then be consistent.

For example, you could model two dice rolled at the same time using 36 different ordered 2-tuples (in a way assuming you can distinguish one die from the other, usually the best approach) or using 21 different unordered sets with up to two elements (sets with one element could be used if you roll the same number twice) or if you're only interested in the sum you rolled, you could use the 12 different possible sums.

Roll	1st option	2nd option	3rd option
3 and 5	$(3,5)$	$\{3,5\}$	8
5 and 3	$(5,3)$	$\{3,5\}$	8
4 and 4	$(4,4)$	$\{4,4\} = \{4\}$	8

These probability spaces are entirely different, but they will not disagree on questions like "how likely is it to roll a sum of 8" if you're consistent when calculating.

Exercise 5. Explicitly (and formally) write down these three ways of modelling the experiment and calculate the probability of getting a sum equal to 8 for each one. Which one is most intuitive for you?

In general, the probability is a function from the set of events (often denoted 2^Ω or $\mathcal{P}(\Omega)$) to the interval $[0,1]$. The probability function satisfies the following properties (some of those are axioms, others can be easily derived):

$$\Pr[\Omega] = 1$$
$$\Pr[\varnothing] = 0$$
$$\Pr\left[\bar{A}\right] = 1 - \Pr[A]$$
$$A \subseteq B \Rightarrow \Pr[A] \leq \Pr[B]$$
$$\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$$

Inclusion/ Exclusion Principle

If we are interested in the probability of the union of some events, we could use the following:

Theorem 2.1.1 *If the events $A_1, \ldots, A_n$ are pairwise disjoint (i.e. if for all pairs $i \neq j$ it holds $A_i \cap A_j = \varnothing$), then*

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

But what happens if the events are not disjoint? The general case is covered by the *inclusion/exclusion principle*, stated in the Theorem 2.1.2.

Theorem 2.1.2 (Inclusion/ exclusion principle) *For any events $A_1, \ldots, A_n$ ($n \geq 2$), we have*

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

This result is very important and gives the exact result to the problem. However, for a large value of n, it gets tedious to compute. The *Union Bound* comes to rescue by giving a upper bound to the value of the same probability.

Theorem 2.1.3 *For any events $A_1, \ldots, A_n$ it holds*

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

For the sake of simplicity, we avoid giving a formal proof of Theorems 2.1.2 and 2.1.3. The intuition for Theorem 2.1.2 can be given with Venn diagrams, for Theorem 2.1.3 just consider that we don't subtract any overlap and that the intersection of more events becomes strictly less likely.

Laplace Spaces

We call a probability space with finitely many elementary events that all have the same probability (namely $1/|\Omega|$) a *Laplace space* (Laplace-Raum). This simplifies all the calculations a lot since we only need to worry about how big the events we're interested in are and not exactly which elementary events comprise them:

If we are considering a discrete sample space Ω of cardinality n, then all elementary events have probability $\frac{1}{n}$. And we get

$$\Pr[A] = \frac{|A|}{n}$$

In other words, we describe the probability of an event as the number of "favorable" events divided by the total number of "possible" events. This reduces complex probability calculations to counting how many favorable and possible outcomes there are.

Combinatorics A quick refresher on things you should already know: If we want to count how many ways of picking k out of n elements there are, it matters whether we are allowed to pick the same element twice (repetitions) and whether we consider the order important.

The number of different ways of picking those elements is best summarized in a small table:

	order matters	order does not matter
repetition allowed	n^k	$\frac{(n+k-1)!}{k!(n-1)!}$
no repetition allowed	$\frac{n!}{(n-k)!}$	$\frac{n!}{k!(n-k)!}$

Exercise 6. Find a real-world example for each of the situations in the table above.

Conditional Probability

Until now, we considered the case where we don't have any prior information regarding the probability space. But what happens if we already have some information about the outcome of an experiment?
e.g. our friend could tell us they rolled two fair dice and did not get a sum of 7; it makes sense to say that the event corresponding to a sum of 7 should now have probability 0

Example 2.1.1 Throw a fair dice. You know that the result is an odd number. What is the probability that you got a prime number?

The prior knowledge that the result is odd reduces the probability space from $\{1, 2, \ldots, 6\}$ to $\{1, 3, 5\}$. Since both 3 and 5 are prime, we get a probability of $\frac{2}{3}$.

We can formalize the argument in the following way. Let $A := \{1, 3, 5\}$ be the event that the result is odd and $B := \{2, 3, 5\}$ the event that the result is prime. We can read the probability of observing a result from B when we already know that we observed a result from A as follows: we know that the "good events" are the one contained in the set $A \cap B$ and the "possible events" are the one contained in A. Hence, we get

$$\Pr\left[B \text{ given } A\right] = \frac{\Pr\left[A \cap B\right]}{\Pr\left[A\right]}$$

In general, given two events A and B in a sample space Ω, we want to determine the probability of event B knowing already that some elementary event in A happened. We write $\Pr[B \mid A]$ and we speak of the *probability of B given A* (B gegeben A). Just as above, we argue that the fact that A happened leads to a new probability space. This explains the following theorem.

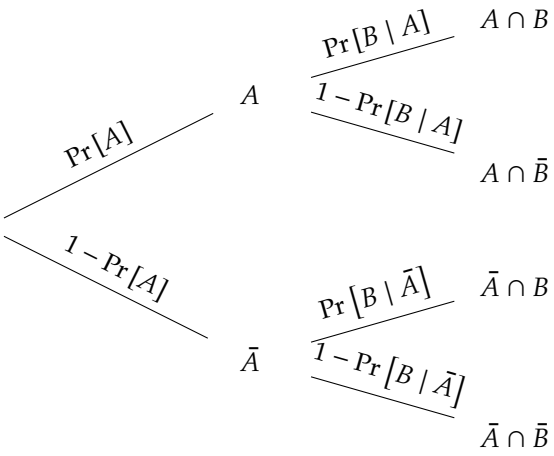
Definition 2.1.2 (Conditional Probability) *Let Ω be a sample space and A, B two events over Ω. We define*

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

We can rearrange the formula for condition probability in the following way

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

This situation (first knowing about A and then finding out about B) can be graphically represented as a tree:



Exercise 7. Prove or disprove: For any events A, B with $0 < \Pr[B] < 1$, it holds that $\Pr[A \mid \overline{B}] = 1 - \Pr[A \mid B]$

The above tree can be useful to compute other probabilities: $\Pr[B]$ and $\Pr[A \mid B]$. We first compute them in the example above, and then we present two theorems than generalize the concept.

- $\Pr[B] = \Pr[A] \cdot \Pr[B \mid A] + \Pr\left[\bar{A}\right] \cdot \Pr\left[B \mid \bar{A}\right]$

  Theorem 2.1.4 (Total Probability) *If we divide Ω into a disjoint partition of A_i for $i \in [m]$, we have*

  $$\Pr[B] = \sum_{i=1}^{m} \Pr\left[A_j\right] \cdot \Pr\left[B \mid A_j\right]$$

- $\Pr[A \mid B] = (\Pr[B \mid A] \cdot \Pr[A]) / \Pr[B]$

  Theorem 2.1.5 (Bayes) *If we divide Ω into a disjoint partition of A_i*

for $i \in [m]$ and we consider an event B with $\Pr[B] > 0$, we get

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

Most often, we simply pick $A_1 = A$ and $A_2 = \overline{A}$ given some event A.

Exercise 8. You are on a TV show and the questions are chosen u.a.r (uniformly at random) from a pool of questions. With probability p you know the answer, and you answer correctly. Otherwise, you guess, and you give the right answer with probability $1/4$. What is the probability that you answer correctly to a randomly chosen question?

Exercise 9. You are on another TV show and you have to choose one out of three closed doors. Behind one door (chosen uniformly at random in advance) there is a wonderful race car, behind the other two doors some friendly goats. You choose a door uniformly at random. The TV host opens a door that you have not chosen (he will never open your chosen door) and behind it is a goat (he will never open the door with the car). Then he makes you an offer: You can change your choice to the other closed. Do you have a probabilistic advantage in accepting the offer?
First describe a suitable probability space (there are three random choices happening in each experiment), then define suitable events and then calculate your chance of winning with and without switching.

Independence of Events

In the previous subsection, we studied the formula for conditional probability. When we consider $\Pr[B \mid A]$, the prior knowledge that A happened can increase, decrease or not influence the probability of A. If the event A does not influence the event B, we have $\Pr[B \mid A] = \Pr[B]$. This fact also implies that B does not influence A:

$$\begin{aligned} &\Pr[B] = \Pr[B \mid A] = \frac{\Pr[A \cap B]}{\Pr[A]} \\ \iff &\Pr[A] \cdot \Pr[B] = \Pr[A \cap B] \\ \iff &\Pr[A] = \frac{\Pr[A \cap B]}{\Pr[B]} = \Pr[A \mid B] \end{aligned}$$

The intuitive concept of independence is the following: two events are independent if they don't influence each other. In other words, knowing that an event happened does not give us any additional information about the probability of another event. This last observation motivates the following definition.

Definition 2.1.3 (Independence of two Events) *Two events A and B are (stochastically)* independent *(unabhängig) if and only if one of the following three (equivalent) proposition hold.*

- $\Pr[B \mid A] = \Pr[B]$

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

How can we generalize the concept of independence of more than two events?

Definition 2.1.4 (Independence of n Events) *The events $A_1, \ldots, A_n$ are (stochastically) independent if and only if for all subsets $I \subseteq \{1, \ldots, n\}$ with $I = \{i_1, \ldots, i_k\}$ we have*

$$\Pr\left[A_{i_1} \cap \cdots \cap A_{i_k}\right] = \prod_{j=1}^{k} \Pr\left[A_{i_j}\right]$$

We point out that checking independence only for all pair of events does not work, as shown in the following example.

Example 2.1.2 Consider the Laplace space $\Omega = \{1, 2, 3, 4\}$ and the following events:

- $A := \{1, 2\}$
- $B := \{1, 3\}$
- $C := \{1, 4\}$

We have

$$\Pr[A] = \Pr[B] = \Pr[C] = \frac{1}{2}$$

We also have that

$$\Pr[A \cap B] = \Pr[A \cap C] = \Pr[B \cap C] = \Pr[\{1\}] = \frac{1}{4}$$

Hence

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

Thus, the events A, B, C are *pairwise* independent. Now consider

$$\Pr[A \cap B \cap C] = \Pr[\{1\}] = \frac{1}{4}$$

But $\Pr[A] \cdot \Pr[B] \cdot \Pr[C] = \frac{1}{8} \neq \frac{1}{4} = \Pr[A \cap B \cap C]$

Summarizing: Independence implies pairwise independence, but the opposite direction does not hold.

Independence is useful because it allows us to compute the probability of the intersection of events simply by multiplying the individual events. But what can we do to compute the intersection of non-independent events? The next theorem helps us to solve the problem in general.

Theorem 2.1.6 (Full Multiplication Rule) *For any events $A_1, \ldots, A_n$ with $\Pr[A_1 \cap \cdots \cap A_n] > 0$ we have*

$$\Pr[A_1 \cap \cdots \cap A_n] = \Pr[A_1] \cdot \Pr[A_2|A_1] \cdot \Pr[A_3|A_1 \cap A_2] \ldots \Pr[A_n|A_1 \cap \cdots \cap A_{n-1}]$$

For independent events, this theorem reduces nicely to the definition of independence.

Exercise 10. Show that if $\Pr[A \mid B] = \Pr\left[A \mid \bar{B}\right]$ then A and B are independent.

2.2 Discrete Random Variables

Often we are not really interested in the result of a random experiment. What we are really interested in are the consequences of such a result. For example, when we play roulette, we are not really interested in the number that comes out, but in the possible increase or decrease of our money.

Example 2.2.1 Consider an unfair coin with a probability of head (H) of 0.4 and a probability of tail (T) of 0.6. We make the following bet: if the coin shows head, we lose \$100, otherwise we win \$80. We model this idea using a function X that maps every possible result of the random experiment (H or T) to the corresponding gain or loss. We have:

$$\begin{aligned} X(H) &= -100 && \text{with probability } 0.4 \\ X(T) &= 80 && \text{with probability } 0.6 \end{aligned}$$

This example motivates the following definition.

Definition 2.2.1 (Random Variable) *A function*

$$X : \Omega \to W_X \subseteq \mathbb{R}$$

is called random variable *(Zufallsvariable) - W_X or $X(\Omega) = \{X(\omega) \mid \omega \in \Omega\}$ is the set of all possible values X can take.*

The function

$$\begin{aligned} f_X : W_X &\to [0,1] \\ x &\mapsto f_X(x) := \Pr[X = x] \end{aligned}$$

is called probability mass function *(Dichtefunktion) of the random variable X.*

The function

$$\begin{aligned} F_X : W_X &\to [0,1] \\ x &\mapsto F_X(x) := \Pr[X \leq x] = \sum_{\substack{k \in \mathbb{R} \\ k \leq x}} f_X(k) \end{aligned}$$

is called cumulative distribution function *(Verteilungsfunktion) of the random variable X.*

The PMF tells us how likely each value of X is while the CDS tells us how likely it is to see a value of X that is at most some value.

Example 2.2.2 (Indicator Random Variable) A very important example of random variable is the *indicator variable* (Indikatorvariable). For an event $A \subseteq \Omega$, we define the corresponding indicator variable as

$$I_A(\omega) = \begin{cases} 1 & \text{if } \omega \in A \\ 0 & \text{otherwise} \end{cases}$$

Note that we can use $F_X(x)$ also to express $\Pr[X > x]$

$$\Pr[X > x] = 1 - \Pr[X \leq x] = 1 - F(x)$$

There are also *conditional random variables* (bedingte Zufallsvariablen) $X \mid A$ that are only defined on the elementary events of some event A instead of all of Ω. While this does not change the values X takes given an outcome of the experiment, the distribution of X might change drastically because certain outputs might now be more or less likely.
Many concepts from conditional probability can be applied to conditional random variables as well, but if in doubt, you can always refer to

$$\begin{aligned} \mathbb{E}[X \mid A] &= \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\{\omega\} \mid A] \\ &= \sum_{\omega \in A} X(\omega) \cdot \Pr[\{\omega\} \mid A] \end{aligned}$$

Independence of Random Variables

Definition 2.2.2 *The random variables $X_1, \ldots, X_n$ are independent if and only if for all $\{x_1, \ldots, x_n\}$ in the codomain of $X_1, \ldots, X_n$ it holds*

$$\Pr[X_1 = x_1, \ldots, X_n = x_n] = \prod_{i=1}^{n} \Pr[X_i = x_i]$$

Note that, interestingly enough, here we don't have to look at subsets of the variables being independent.

An important theorem about independent random variables is the following.

Theorem 2.2.1 *Let $f_1, \ldots, f_n$ be real functions. If the random variables $X_1, \ldots, X_n$ are independent, then so are the transformed random variables $f_1(X_1), \ldots, f_n(X_n)$.*

(Given a real function $g : \mathbb{R} \to \mathbb{R}$ and a random variable X, the transformed random variable $g(X)$ is simply a function that maps $\omega \mapsto g(X(\omega))$)

This means that we cannot make independent random variables dependent by only applying functions to each variable separately.
For example: Given any independent random variables X, Y, the variables X^X and $\big(\log(Y) + \sin(Y)\big)\frac{1}{Y^2+1}$ are also independent.

Expected Value

The expected value is useful to determine the "average" outcome of a random experiment (though most of the time that's not an outcome that can appear in any one experiment).

Definition 2.2.3 (Expected Value) *The* expected value *(Erwartungswert) of a random variable X is*

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

or sometimes more useful if we don't know the probability space but only the distribution of X:

$$\mathbb{E}[X] = \sum_{x \in W_X} x \cdot f_X(x)$$

It's possible to show that, if the random variable maps results of a random experiment to natural numbers, an equivalent definition of expected value is given by

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

A crucial property of the expected value is *linearity of expectation* (Linearität des Erwartungswertes). This property is very useful to solve *a lot* of exercises in this course.

Theorem 2.2.2 (Linearity of Expectation) *For any two random variables X, Y and $a \in \mathbb{R}$ it holds that:*

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

$$\mathbb{E}[a + X] = a + \mathbb{E}[X]$$

$$\mathbb{E}[a \cdot X] = a \cdot \mathbb{E}[X]$$

We can generalize this to finitely many random variables:
For any random variables $X_1, \ldots, X_n$ and $a_1, \ldots, a_n, b \in \mathbb{R}$, we have

$$\mathbb{E}\left[b + \sum_{i=1}^{n} a_i X_i\right] = b + \sum_{i=1}^{n} a_i \mathbb{E}[X_i]$$

This property is particularly useful if a random variable can be expressed as a sum of simpler random variables with known expected value (*e.g.* indicator variables).

Example 2.2.3 We toss a fair coin 100 times. How many heads do we expect? We denote with X the number of heads and for $i = 1, \ldots, 100$ we define X_i as the indicator random variable for head. We get

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{100} X_i\right] = \sum_{i=1} 100\mathbb{E}[X_i] = \sum_{i=1}^{100} \frac{1}{2} = 50$$

Theorem 2.2.3 *For independent random variables $X_1, \ldots, X_n$, we have*

$$\mathbb{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n} \mathbb{E}[X_i]$$

Exercise 11. Given a graph G with $2n$ vertices, with n vertices blue and n vertices red. The probability that there is an edge between any two nodes is $\frac{1}{2}$ for all pair of nodes. What is the expected number of edges between vertices of the same color? Do you need the existence of those edges to be independent?

Exercise 12. In the same situation as above, let X be the random variable defined as the number of edges that stay within a color. Assume now that the existence of the edges are independent events. Calculate $\mathbb{E}\left[X^2\right]$. (Write X as a sum of indicator variables $I_1, \ldots, I_m$ and use the fact that $(\sum_{i=1}^{m} I_i)^2 = \sum_{i=1}^{m} \sum_{j=1}^{m} I_i \cdot I_j$)

Variance

The expected value $\mathbb{E}[X]$ of a random variable X gives some useful information about it, but it does not say how X is "spread out". We would "like" to be able to distinguish random variables whose values all lie very close to the expected value and ones where the spread is very big. To that end, we introduce another quantity to measure the spread of the distribution around its mean.

Definition 2.2.4 (Variance) *For a random variable X with $\mu := \mathbb{E}[X]$, we define its* variance *(Varianz) (sometimes written as σ^2) as*

$$Var[X] := \mathbb{E}\left[(X-\mu)^2\right] = \sum_{\omega \in \Omega} (X(\omega) - \mu)^2 \Pr[\omega]$$

We also define the standard deviation $\sigma := \sqrt{Var}$

You can think of the variance as "average distance from the mean squared".

Computing the variance with the definition can be quite tedious. The following formula is often easier to use:

$$\mathrm{Var}[X] = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2$$

Two other important results about variance are given in the following theorems.

Theorem 2.2.4 *For any random variable X and $a, b \in \mathbb{R}$ we have*

$$Var[a \cdot X + b] = a^2 \cdot Var[X]$$

(shifting the values has no effect on the spread, but scaling them is even more important)

Theorem 2.2.5 *For independent random variables $X_1, \ldots, X_n$, we have*

$$Var\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} Var\left[X_i\right]$$

Exercise 13. Given is the following distribution of a random variable X

x	-4	-2	5	8	10
$f_X(x)$	0.25	0.10	0.20	0.15	0.30

Compute $\mathbb{E}[2X + 8]$ and $\mathrm{Var}[2X + 8]$.

Multiple Random Variables

We might find experiments in which we always observe multiple random variables at the same time, such as throwing a die with both number and color on each face. We then obtain a joint probability density function

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

which expresses the probability of observing values x and y for the two random variables X and Y respectively.

It is possible to extract the individual probability distributions as *marginal distributions* (Randdichten)

$$f_X(x) = \sum_{y \in W_Y} f_{X,Y}(x, y)$$

And similarly for $\Pr[Y = y]$.

Theorem 2.2.6 *Let X and Y be two independent random variables, and $Z = X + Y$. We then have*

$$f_Z(z) = \sum_{x \in W_X} f_X(x) \cdot f_Y(z - x)$$

This allows us to compute the density function of a random variable using those of others, which might be helpful in some cases.

A special case that can come in very handy in an exam is the following.

Definition 2.2.5 *We call the random variables $X_1, \ldots, X_n$* independent and identically distributed *(unabhängig und identisch verteilt) - or just i.i.d. - if and only if*

- *they all have the same distribution ($f_{X_1} = f_{X_2} = \ldots$ or equivalently $F_{X_1} = F_{X_2} = \ldots$)*
- *and they are all independent*

(We know you didn't cover this in the course, but you should be able to grasp this concept fairly well if you understand the density function, and it makes it easier to talk about certain things here)

Exercise 14. Find an example of a set of random variables that are i.i.d.

Find an example of a set of random variables that are independent but not i.i.d.

Find an example of a set of random variables that are identically distributed but not i.i.d.

Theorem 2.2.7 (Wald's Identity) *Let X and N be two independent random variables, with $W_N \subseteq \mathbb{N}$.*
If we now define $Z(\omega) = \sum_{i=1}^{N(\omega)} X_i(\omega)$ where the variables $X, X_1, X_2, \ldots, X_n$ are i.i.d.
Then it holds that

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

This can be used when we have many independent outcomes, we randomly select a number N and then sum up the first N of the independent outcomes.

2.3 Important Discrete Distributions

Following, some of the most common and useful distributions are given, with density, expected value and variance.

Bernoulli Distribution

A Bernoulli random variable can take two values, 1 and 0 with probability p and $1-p$ respectively. This is useful to describe events that either happen or don't (indicator variables also follow a Bernoulli Distribution).

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

$$f_X(x) = \begin{cases} p & \text{for } x = 1 \\ 1-p & \text{for } x = 0 \\ 0 & \text{else} \end{cases}$$

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

Exercise 15. Compute the variance of $X \sim \text{Bernoulli}(p)$ by hand.

Binomial Distribution

A Binomial random variable is useful when we sum up multiple independent(!) Bernoulli random variables that all have the same success probability. For example, we can describe the probability of scoring a certain number of points in a game or look at how many light bulbs are expected to break during delivery.

$$X \sim \text{Bin}(n, p) = \sum_{i=1}^{n} \text{Bernoulli}(p)$$

$$f_X(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & \text{for } x \in \mathbb{N}_0 \\ 0 & \text{else} \end{cases}$$

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

Exercise 16. Consider a football league. A victory gives 3 points and a loss 1 points. There is no draw possible. Each team plays 25 games. Consider a team that wins each game independently with probability $p = 0.6$. Let X be the random variable for the number of points obtained by the team. Compute $\mathbb{E}[X]$ and $\text{Var}[X]$.

Geometric Distribution

A geometric random variable is used to describe the amount of time or number of events we have to wait for a Bernoulli event to happen. For example, it can be used to estimate the MTBF (mean time between failures), or the expected life of a hard disk before it dies.

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

$$f_X(x) = \begin{cases} p(1-p)^{x-1} & \text{for } x \in \mathbb{N} \\ 0 & \text{else} \end{cases}$$

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

An important property of the geometric distribution is the fact that it is *memoryless* (gedächtnislos). The fact that an event happened in the past doesn't change the probability of it happening in the future (for independent variables).

$$\Pr[X \geq s + t \mid X > s] = \Pr[X \geq t]$$

Exercise 17. Compute the expected value of $X \sim \text{Geo}(p)$ by hand using $\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}$.

Negative Binomial Distribution

The negative binomial distribution is a generalization of the geometric distribution: instead of waiting for the first success, we repeat the experiment until we have n successes. Of course, if $n = 1$, we are back to the geometric distribution. But what if n is larger? The random variable X describes the number of repetitions until we see n successes of an independently repeated event with probability p.

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

$$f_X(x) = \begin{cases} \binom{x-1}{n-1} p^n (1-p)^{x-n} & \text{for } x \in \mathbb{N}_0 \\ 0 & \text{else} \end{cases}$$

$$\mathbb{E}[X] = \frac{n}{p} \qquad \text{Var}[X] = \frac{n(1-p)}{p^2}$$

Poisson Distribution

Let's say you wanted to model births per week in some city (or any random event in time) as a random variable, knowing that there are on average 6 births per week. Two problems arise if you try to use a Binomial distribution: Firstly, 6 cannot be the p parameter of the Binomial distribution and secondly, if you do try to do Bin(7, 6/7) (modelling births per day for each of the seven days), it's impossible to have 8 births per week.
Maybe Bin($7 \cdot 24$, $\frac{6}{7 \cdot 24}$) (modelling births per hour for each hour) is better. If you continue to do this, you will end up (in the limit) with the Poisson distribution.

The Poisson distribution is used to model events happening randomly in time when you know the rate of those events (in our case, 6 births per week) and when you assume that a birth in any equally sized time frame is equally likely. The parameter λ now does not take a probability, but rather a rate over time for the interval we're interested in!

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

$$f_X(x) = \begin{cases} \frac{e^{-\lambda}\lambda^x}{x!} & \text{for } x \in \mathbb{N}_0 \\ 0 & \text{else} \end{cases}$$

$$\mathbb{E}[X] = \lambda \qquad \text{Var}[X] = \lambda$$

When we consider the limit $\lim_{n\to\infty} \text{Bin}(n, \frac{\lambda}{n})$ we could see that it equals $\text{Po}(\lambda)$.

2.4 Coupon Collector Problem

Imagine that you have to complete a collection of n items $a_0, \ldots, a_{n-1}$. At each iteration, you receive one object independently and uniformly at random from the set of all objects. The Coupon Collector Problem

studies the random variable X that describes the number of iterations until the collection is complete.

The key idea to approach the situation is to break it down into phases: phase i describes the time between getting our $(i-1)$-st object until we get the i-th object. Let X_i be the number of iterations in phase i. We have $X = \sum_{i=1}^{n} X_i$. We observe that phase i ends, when we get one of the $n-i+1$ items that we don't have yet. Hence, X_i has a geometric distribution with parameter $\frac{n-i+1}{n}$ and $\mathbb{E}[X_i] = \frac{n}{n-i+1}$. We can now compute the expected value of the Coupon Collector Problem.

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{n-i+1}$$

We can now observe that this sum is simply $\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}$ and we can rearrange those terms to give us a sum that's easier to work with:

$$= n \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n$$

Where H_n is the n-th harmonic number defined simply as $H_n = \sum_{i=1}^{n} \frac{1}{i}$. It's a very useful fact now that we can write H_n also as $\ln n + \mathcal{O}(1)$.

$$= n \ln n + \mathcal{O}(n)$$

And finally we have the expected number of iterations (asymptotically anyway, but this is fine).

Exercise 18. Consider the coupon collector of n elements, starting with 0 elements and ending after collecting $n/2$ elements. Compute the expected number of rounds until completion. Watch carefully for where you need to do something different from above.

2.5 Important Inequalities

In the script, you have seen an important example that shows that the expected value describes the mean of the results of several repetitions of a random experiment. However, if we consider a single realization of the random experiment, this value could be quite far from the expected value in some cases. This fact motivates us to introduce three useful inequalities.

Theorem 2.5.1 (Markov's Inequality) *Let X be a random variable that takes only non-negative values. Then, for all $t \in \mathbb{R}^+$, we have*

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$$

Proof. We have

$$\begin{aligned}\mathbb{E}[X] &= \sum_{\omega\in\Omega} X(\omega)\Pr[\omega] \\ &\geq \sum_{\omega\in\Omega, X(\omega)\geq t} X(\omega)\Pr[\omega] \\ &\geq t \sum_{\omega\in\Omega, X(\omega)\geq t} \Pr[\omega] = t\cdot\Pr[X\geq t]\end{aligned}$$

□

Theorem 2.5.2 (Chebyshev's Inequality) *Let X be any random variable with non-zero variance and $t \in \mathbb{R}^+$. We have*

$$\Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{Var[X]}{t^2}$$

Proof. We have $\Pr[|X - \mathbb{E}[X]| \geq t] = \Pr\left[(X - \mathbb{E}[X])^2 \geq t^2\right]$. Since the latter is a non-negative random variable, we can apply Markov's inequality and get

$$\Pr[|X - \mathbb{E}[X]| \geq t] = \Pr\left[(X - \mathbb{E}[X])^2 \geq t^2\right] \leq \frac{\mathbb{E}\left[X - \mathbb{E}[X])^2\right]}{t^2} = \frac{\mathrm{Var}[X]}{t^2}$$

□

Informally, this estimates the probability that the realization of a random variable will be far from the expected value. This is the probability that $X \notin [\mathbb{E}[X] - t, \mathbb{E}[X] + t]$. An upper bound is given, that depends on the variance (the larger the variance, the more likely that X will lie far from the mean) and t (the bigger, the less likely it is to find a value that far out).

We conclude this section by stating, without proof, three other important inequalities known as Chernoff's bounds. The inequality of Chernoff gives a usually much more precise result than Markov and Chebyshev but is not applicable to all random variables. This is because Chernoff works only for the sum of independent Bernoulli variables with the same parameter p.

Theorem 2.5.3 (Chernoff Inequalities) *Let $X_1, \ldots, X_n$ be i.i.d. Bernoulli variables with the same parameter p. We have for $X = \sum_{i=1}^{n} X_i$ and every $0 < \delta \leq 1$*

$$\begin{aligned}\Pr[X \geq (1+\delta)\mathbb{E}[X]] &\leq e^{-\frac{1}{3}\delta^2\mathbb{E}[X]} \\ \Pr[X \leq (1-\delta)\mathbb{E}[X]] &\leq e^{-\frac{1}{2}\delta^2\mathbb{E}[X]} \\ \Pr[X \geq t] &\leq 2^{-t} \text{ for } t \geq 2e\mathbb{E}[X]\end{aligned}$$

Exercise 19. Find an example of a random variable (that can take negative values) where Markov's inequality does not hold.

Exercise 20. Consider a test with 18 questions and two possible answers to each question. If one guesses each answer, what is the probability of answering correctly to at least 5 and at most 13 answers? Give an expression for the exact probability (no need to calculate it). Then use all three inequalities if possible to give bounds on that probability and compare the results.

Exercise 21. A factory produces 45'000 screw per day. The screws are supposed to be 30 mm long, but sometimes manufacturing errors add up, and some screws have to be discarded. The probability that any given screw is too short is $\frac{1}{3}$ and the probability that a screw is too long is $\frac{1}{5}$ (the different screws' lengths are independent).

1. Let X be the number of screws discarded over a whole day. How is X distributed? Calculate the expected value μ.
2. Tom is a supervisor at the factory and if $0.9\mu < X$ and $X < 1.2\mu$, he reports to his boss that the day was uneventful. Use Chernoff to give an upper bound on the probability that a given day *is* eventful.
3. Another factory produces nails and its supervisor tells Tom that their expected number of discarded nails per day is not more than 50. Without knowing anything else about the other factory (except that their nails are also produced independently and have identical chances of being too long or too short), use Chernoff to give an upper bound on the probability that at least 300 nails have to be discarded in a given day.

3 Randomized Algorithms

In the course *Algorithms and Data Structures* and in the first chapter of this course you have encountered many algorithms: Karatsuba's algorithm, dynamic programming algorithms, merge sort, DFS, Kruskal's algorithm... All these algorithms are *deterministic*, *i.e.* given an input they always return the same output. Here we study *randomized* algorithms (QuickSort is an example you already know), *i.e.* algorithms that may return different outputs in different executions. A more formal approach to define randomized algorithms (which is not absolutely essential for the purpose of this course) could be: randomized algorithm are deterministic algorithms if we consider that their input consists not only of the data relative to the problem, but also of an (infinite long) sequence of random bits. A more intuitive definition could be that randomized algorithms have access to the (pseudo) random number generator of Java, and hence you can generate samples from a distribution of your choice. Randomized algorithms are beautiful. First, they are often very elegant, and their analysis is often very clean. Second, they are powerful: some problems that are very difficult to solve, such as the NP-complete longest path problem, can be efficiently approximated via randomized algorithms (at least for short longest paths).

We distinguish two classes of randomized algorithms: *Monte Carlo* algorithms and *Las Vegas* algorithms. Monte Carlo algorithms have a deterministic runtime, but they can return wrong answers. Las Vegas algorithms always return the correct answer, but their runtime is a random variable.

Exercise 22. Which of the following Monte Carlo algorithms are useful for a problem that requires a binary answer?

- An algorithm that returns the correct answer with probability ≥ 0.6
- An algorithm that returns the correct answer with probability ≥ 0.5
- An algorithm that returns the correct answer with probability ≥ 0.4

Exercise 23. Does the answer to the previous exercise change if we consider a maximization problem? (assuming the result is never too big)

Exercise 24. How can we transform a Las Vegas algorithm to a Monte Carlo algorithm? What about the opposite direction?

3.1 Success Probability Amplification

In this section, we explore some techniques useful to design Las Vegas and Monte Carlo algorithms with certain properties.

Monte Carlo Algorithms: One Sided Errors

We say that a Monte Carlo algorithm A for a decision problem has a one-sided error if and only if, for some $\varepsilon > 0$:

$$\Pr[A(I) = YES] = 1 \quad \text{if the correct answer for input } I \text{ is YES, and}$$
$$\Pr[A(I) = NO] \geq \varepsilon \quad \text{if the correct answer for input } I \text{ is NO}$$

for any input I.

Right now, this algorithm has a success probability of ε and whenever it returns NO, we can be sure that NO is the correct answer whereas the answer YES could be wrong. (really think about this if it doesn't make sense)
We can let this algorithm run multiple times to increase its chance of success to $1 - \delta$ for an arbitrary $\delta > 0$:

Theorem 3.1.1 *Let A be a Monte Carlo algorithm for a decision problem with one-sided error that has a success probability of ε. Let $0 < \delta \leq 1$ be a real number.*

If we repeat A at least $N = \varepsilon^{-1} \ln \delta^{-1}$ times and answer YES only if every execution of A returned YES, the resulting algorithm will have a success probability of at least $1 - \delta$.

Monte Carlo Algorithms: Two Sided Errors

We say that a Monte Carlo algorithm A for a decision problem has a two-sided error if and only if

$$\Pr[A(I) \text{ is correct}] \geq 1/2 + \varepsilon \quad \text{for some } \varepsilon > 0$$

for any input I.

Right now, this algorithm has a success probability of $1/2 + \varepsilon$ and both answers could be wrong. It should make sense that we will need to "vote" on the correct answer and that we will need more executions to have a high chance of success:

Theorem 3.1.2 *Let A be a Monte Carlo algorithm for a decision problem with two-sided error that has a success probability of $1/2 + \varepsilon$. Let $0 < \delta \leq 1$ be a real number.*

If we repeat A at least $N = 4\varepsilon^{-2} \ln \delta^{-1}$ times and answer with the majority of results from all executions, the resulting algorithm will have a success probability of at least $1 - \delta$.

Monte Carlo Algorithms: Optimisation Problems

We can also do a similar thing for maximization problems, so long as the algorithm never gives an answer that is bigger than is possible (it always returns valid answers, but not always optimal ones) and has a non-zero chance to return a "good" answer ($\geq f(I)$):

Theorem 3.1.3 *Let A be a Monte Carlo algorithm for a maximization problem as above and*

$$\Pr\left[A(I) \geq f(I)\right] \geq \varepsilon \qquad \text{for some } \varepsilon > 0$$

If we repeat A at least $N = \varepsilon^{-1} \ln \delta^{-1}$ times and answer with the maximum of the results from all executions, the resulting algorithm will output an answer that is not smaller than $f(I)$ with probability at least $1 - \delta$.

This might sound confusing, but you can simply pick $f(I)$ to be whatever suits you. *e.g.* the optimal answer or 50% of the optimal answer, etc.
We can use the exact same strategy for minimization problems if we replace "$\geq f(I)$" with "$\leq f(I)$" and return the minimum result instead.

3.2 Primality Tests

An easy example for Monte Carlo algorithms are *primality tests*. These are often used when we need to be fairly sure that a certain big number is prime (*e.g.* for applications in cryptography).
We will look at three different ways of testing whether a given number n is prime or not, using a randomly chosen number a ($1 \leq a < n$) that has some (hopefully high) chance of serving as a counterexample if n is not in fact prime.

Euclid One of the most obvious ways to check whether n is prime, given a randomly chosen a, is to see whether they have any factors in common.

$$\gcd(a, n) \neq 1 \implies n \text{ is not prime}$$

Giving us the Monte Carlo algorithm:

Algorithm 3.1: EuclidPrimalityTest(n)

```
1 Choose 1 ≤ a < n uniformly at random
2 if gcd(a, n) ≠ 1
3     return "not prime"
4 else
5     return "prime"
```

This algorithm is quite bad as we can't give a good lower bound on the success probability (it depends heavily on the amount of factors n has)

Fermat A slightly less obvious property of primes that you might remember from discrete maths is that

$$n \text{ is prime} \implies \forall a \in \{1, \ldots, n-1\} : a^{n-1} \equiv_n 1$$

Which leads us to a second algorithm:

Algorithm 3.2: FermatPrimalityTest(n)

```
1 Choose 1 < a < n uniformly at random
2 if a^(n-1) ≢_n 1
3     return "not prime"
4 else
```

```
5     return "prime"
```

This algorithm is better. It has a one-sided error and its success probability is at least 1/2.
But, sadly, there are some numbers (Carmichael numbers) that will always be categorized as prime even though they are not.

Miller-Rabin So we take the same idea one level further and also remember that

n is prime $\implies$ the equation $x^2 \equiv_n 1$ has exactly the solutions 1 and $n-1$

We first check that $a^{n-1} \equiv_n 1$. We then observe that $n-1$ is even (it has to be if n is prime), so we can apply what we just remembered and check that the square root $a^{\frac{n-1}{2}}$ is indeed either 1 or $n-1$. We can keep going like this until we either get $n-1$ as a square root or the exponent is not even anymore (in which case we don't have an equation of the form $x^2 \equiv_n 1$ anymore).

Following this train of thought, we get the Miller-Rabin primality test:

Algorithm 3.3: MillerRabinPrimalityTest(n)

```
1  Write n − 1 as 2^k · d with d odd
2  Choose 1 < a < n uniformly at random
3  if a^d ≢_n 1
4      for i = 1, \dots, k do
5          if a^{2^i · d} ≡_n n − 1
6              return "prime"
7      return "not prime"
8  else
9      return "prime"
```

This is finally good! It has one-sided error, and has a success probability of at least 3/4 for any n. This algorithm is widely used and thanks to the efficiency of modular exponentiation it is also remarkably fast.

Exercise 25. We stated multiple times that these algorithms have one-sided error. Which answer ("prime" or "not prime") is always correct?

Give an algorithm that has a success probability of at least 99.99% using the Miller-Rabin primality test.

3.3 Target Shooting

The goal of this section is to analyze an algorithm to compute $\frac{|S|}{|U|}$, where S and U are finite sets with $S \subseteq U$. Here we assume that we can generate elements $u \in U$ u.a.r. and that we can efficiently compute an indicator function $I_S(u)$ which, given an element $u \in U$, tells us whether u is in S or not. Popular examples of these algorithms include computing the area of an object in a geographic map. The algorithm that we use is simple: We

sample N elements from U and we use the samples with the indicator function to estimate the desired ratio. Formally

```
1 Choose u_1, ..., u_N u.a.r. from U
2 return 1/N Σ_{i=1}^{N} I_S(u_i)
```

Algorithm 3.4: Target-Shooting

It is intuitively clear that this value approximates $\frac{|S|}{|U|}$ and that a larger value N leads to a better result. We want to give quantitative arguments to this idea and give criteria for the choice of the parameter N.

Theorem 3.3.1 *The expected value of the returned result is equal to the true ratio* $\frac{|S|}{|U|}$.

Proof. We can define a variable Y_i for each sample u_i that indicates whether or not $u_i \in S$. Y_i is Bernoulli distributed with parameter $\frac{|S|}{|U|}$. We return $\frac{1}{N}\sum_{i=1}^{N} Y_i$, which has expected value $\frac{|S|}{|U|}$. □

The variance of the same random variable is $\frac{1}{N}(\frac{|S|}{|U|} - (\frac{|S|}{|U|})^2)$: this tells us that larger values of N lead to an approximation closer to the expected value, and hence closer to the true result. We want to find a value of N such that the relative error of our answer is less than some $\varepsilon > 0$, *i.e.*:

$$\Pr\left[\left|Y - \frac{|S|}{|U|}\right| \leq \varepsilon \frac{|S|}{|U|}\right] \geq 1 - \delta$$

for a given $\delta > 0$.

Theorem 3.3.2 *Let* $\delta, \varepsilon > 0$. *If* $N \geq 3\frac{|U|}{|S|} \cdot \varepsilon^{-2} \cdot \log(\frac{2}{\delta})$, *then the output of the Target-Shooting algorithm is in the interval* $\left[(1-\varepsilon)\frac{|S|}{|U|}, (1+\varepsilon)\frac{|S|}{|U|}\right]$ *with probability* $1 - \delta$.

3.4 Long Paths

In the course *Algorithms and Data Structures* you have seen multiple algorithms to compute shortest paths. At this point, a natural question is: Can we modify those algorithms in order to compute *longest paths*? We start by looking at a special case, directed acyclic graphs. Computing longest paths in this situation is easy: we just multiply all the weights by minus one, and we use the tools from the previous semester. For most graphs, this transformation is not useful because it creates cycles of negative length in the graph. But in the case of directed acyclic graphs, then no negative cycles can be created.

In the general case, however, this problem is NP complete and hence it is at least plausible to think that no polynomial time algorithm exists for this task. But this does not mean that we have to give up completely. For example, we might want to solve the problem of deciding whether

a graph G has a path of length (at least) B for small B, concretely for $B \in \mathcal{O}(\log n)$. In order to solve this task, we first introduce another problem, the colorful-path problem, and then we show how an algorithm for this new problem can be used as a subroutine for a randomized algorithm for the long path problem that will be relatively efficient for small B.

Colorful-Path Problem

Let $k \in \mathbb{N}$. We color a graph $G = (V, E)$ with the function $\gamma : V \to [k]$ (where γ describes an arbitrary assignment of colors, *e.g.* neighbors *can* get the same color). We say that a path is called *colorful* (bunt) if and only if all its vertices have different colors. We now define the colorful-path problem: given a graph G, an integer k and a color function γ, decide whether there is a colorful path of length $k-1$ in G colored with γ. In order to determine whether the colored graph contains a colorful path of length $k-1$, we define the following quantity

$$P_i(v) := \left\{ S \in \binom{[k]}{i+1} \,\middle|\, \begin{array}{l}\text{there exists a colorful path of length } i \text{ that} \\ \text{ends in } v \text{ with exactly the colors in } S\end{array} \right\}$$

We can easily initialize

$$P_0(v) = \{\{\gamma(v)\}\} \text{ for all } v \in V$$

This is useful because there is a colorful path of length $k-1$ in G if and only if $\bigcup_{v \in V} P_{k-1}(v) \neq \varnothing$. Hence, to solve the colorful-path problem, we compute $P_{k-1}(v)$ for all vertices v and we check, whether these sets are all empty or not.

The natural question now is how to compute $P_i(v)$ given v and all $P_{i-1}(v)$. We can do this by looking at the colors that we could use to get to a neighbor x of v and checking whether $\gamma(v)$ is already used or not:

$$P_i(v) = \bigcup_{x \in N(v)} \{R \cup \{\gamma(v)\} \mid R \in P_{i-1}(x) \text{ and } \gamma(v) \notin R\}$$

Since we have the base case for $i = 0$, we can implement this using a dynamic programming approach with increasing values of i.

Algorithm 3.5: ColoredPath(G, i)

```
1 for all v ∈ V
2     P_i(v) ← {}
3     for all x ∈ N(v)
4         for all R ∈ P_{i-1}(x) with γ(v) ∉ R
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}}
```

Algorithm 3.6: ShortLongPath(G)

```
1 for all v ∈ V
2     P_0(v) ← {{γ(v)}}
3 for i = 1, ..., k − 1
```

4 ColoredPath(G, i)
5 ***return*** whether $\bigcup_{v \in V} P_{k-1}(v) \neq \varnothing$

Since ColoredPath(G,i) has complexity

$$\mathcal{O}\left(\sum_{v \in V} deg(v) \cdot \binom{k}{i} \cdot i\right) \in \mathcal{O}\left(\binom{k}{i} \cdot i \cdot m\right)$$

our final algorithm has complexity

$$\mathcal{O}\left(|V| + \sum_{i=1}^{k-1} \left(\binom{k}{i} \cdot i \cdot m\right) + |V|\right) = \mathcal{O}(2^k km)$$

which is polynomial if $k \in \mathcal{O}(\log n)$.

Short Long Path

Now that we have a deterministic algorithm for the colorful-path problem (which is polynomial for $k = \mathcal{O}(\log n)$), we go back to our original problem of determining whether G contains a path of length B. We use the following Monte Carlo algorithm:

1. Set $k = B + 1$ and color G randomly with k colors.
2. Use the algorithm of the previous section to find a colorful path with k vertices. Repeat these first two steps $\lambda \cdot e^k$ times.
3. If at least one of the repetition of the previous step found a colorful path with k vertices, we have a long path of length B. Otherwise, we return that no such path exists.

The runtime of the Monte Carlo algorithm is simply given by multiplying the complexity of the algorithm of the previous section with the number of repetitions. We get

$$\mathcal{O}\left(\lambda \cdot e^k \cdot 2^k km\right)$$

which is polynomial if $k = \mathcal{O}(\log n)$. For the success probability, we observe the following:

- If there is no path with $B + 1$ vertices, the success probability is one (we will never answer YES).
- If there is a path with $B + 1$ vertices, we fail only if in every iteration our random coloring is unlucky (we always have some color twice on this path=. The probability of coloring the path of length $B + 1$ with k different colors is $\frac{k!}{k^k} \geq e^{-k}$.

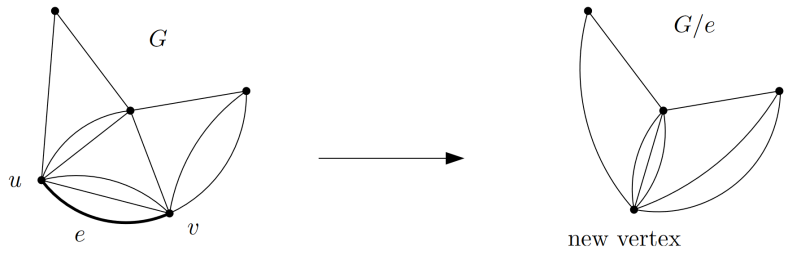
With this argument, it is easy to see that the failure probability is at most $(1 - e^{-k})^{\lambda \cdot e^k} \leq (e^{e^{-k}})^{\lambda \cdot e^k} \leq e^{-\lambda}$.

3.5 Min Cut

In this section, we consider undirected multigraphs $G = (V, E)$ without loops. We say that a set $C \subseteq E$ is a *cut* (Kantenschnitt) in G if and only if $G' = (V, E \setminus C)$ is disconnected. We are interested in finding a cut of minimum cardinality, a so-called *min cut*. With $\mu(G)$ we denote the size of such a minimum cut in G (note that the minimum cut does not have to be unique).

Some important facts

Throughout this section, we repeatedly use an operation called *edge contraction* (Kantenkontraktion). Let G be a multigraph and let $e = \{u, v\}$ be an edge of G. The contraction of e means that we "glue" u and v together into a single new vertex, removing the edges between u and v, but keeping all other edges incident to u, v, simply redirecting them to the new vertex. The resulting graph is denoted by G/e. This is illustrated in the following figure:



Every contraction reduces the number of nodes by exactly one and the number of edges by at least one.

Lemma 3.5.1 *Let G be a multigraph and e an edge of G. Then $\mu(G/e) \geq \mu(G)$. Moreover, if there exists a minimum cut C in G such that $e \notin C$, then $\mu(G/e) = \mu(G)$.*

This means that contracting edges can never decrease the size of a minimum cut.

Lemma 3.5.2 *Let $G = (V, E)$ be a multigraph with n vertices. Then the probability that we preserve the size of a min cut ($\mu(G) = \mu(G/e)$) for a uniformly randomly chosen edge $e \in E$ is at least $1 - \frac{2}{n}$.*

Basic Version

In this section, we assume that the input graph is connected. We assume that the graph is represented in a way such that we can:

- perform an edge contraction in $\mathcal{O}(n)$
- choose an edge uniformly at random among all edges of the current multigraph in $\mathcal{O}(n)$

- find the number of edges connecting two given vertices in $\mathcal{O}(1)$

The idea is very simple: we consider a graph with two vertices as base case (*i.e.* in this case we count the number of edges between the two vertices, and we return the size of the min cut). If we have more than two vertices we repeatedly choose a random edge of the current graph, and we contract it, until only two vertices are left (in which case a minimum cut has to remove every edge, giving us an easy answer).

Algorithm 3.7: Cut(G)

```
1 while G has more than two vertices do
2     e ← u.a.r. edge from G
3     G ← G/e
4 return number of edges left over
```

The runtime of this algorithm is $\mathcal{O}(n^2)$.

By using the observations of Lemma 3.5.1, we see that this algorithm always returns a number at least as large as $\mu(G)$. If C is a minimum cut in the input graph G, and if we never contract an edge of C during the whole algorithm, then the returned number is exactly $\mu(G)$.

In general, we note that the algorithm is correct if and only if:

- $\mu(G) = \mu(G/e)$ for the first contracted edge e (which has a probability of at least $1 - \frac{2}{n}$)
- Cut succeeds for G/e

Which gives us the following recurrence relation for the success probability $p(n)$ (defined as the worst-case success probability for graphs with n vertices):

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

From which we get:

$$p(n) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \cdot \underbrace{p(2)}_{=1} = \frac{2}{n(n-1)}$$

In order to get an algorithm with an arbitrary good success probability we use a Monte Carlo approach by running Cut N times and return the smallest cut size found in all these runs. If the returned size is not correct, it means that the algorithm failed N times in a row. The failures in different runs are independent, and hence the probability of N failures in a row is bounded by:

$$\left(1 - \frac{2}{n(n-1)}\right)^N \leq e^{-\frac{2N}{n(n-1)}}$$

where we used the inequality $1 + x \leq e^x$. If we set, for example, $N = 10n(n-1)$, then the failure probability is bounded above by e^{-20} (which is very small). By increasing the number of repetitions N, the failure probability can be further decreased. Altogether, by setting $N = c \cdot n^2$we have an algorithm with runtime $\mathcal{O}(n^4)$ (because we have $\mathcal{O}(n^2)$ repetitions

of the $\mathcal{O}(n^2)$ algorithm) and arbitrary small constant probability of failure.

Bootstrapping

Let's develop a better algorithm and let's call it BetterCut.

The probability of error in Cut increases with the number of contractions (*i.e.* with the first contraction we have an error probability of $\frac{2}{n}$, but the last iteration has an error probability of $\frac{2}{3}$). Why should we take the risk of contracting edges until we have only two vertices? It is better to do less contractions (let's say, until the graph has t nodes, where t has to be chosen carefully) and then moving on in a way that is perhaps slower but has a better success probability.

What algorithms can we use in order to calculate the min cut when we have reached the threshold t?
Well, once we have a constant number of vertices left (for example 6 because why not), we can use a deterministic approach that would then have constant runtime and certainly be correct. But what do we do until we get there?
We can just call BetterCut again on our smaller graph with t nodes! If we do this multiple times, we can amplify the success probability just like we did in the beginning of this chapter.

Suffice to say, we can use a recursive algorithm to achieve a runtime of $\mathcal{O}(n^2 \log^k(n))$ for a constant k, which is practically $\mathcal{O}(n^2)$.

3.6 Hashing

Hash Tables

A *hash table* is a data structure used to implement an associative array, a structure that can map keys to values. If we have many possible keys, implementing this with an actual array is extremely costly. Linked Lists come to mind, but we can do better!

A *hash function* $h(k)$ is used to map an arbitrary type of key (string, number, etc) to an index of the array, also called bucket or slot (we could for example take the hash modulo our array size to determine the bucket). The idea being to have much fewer buckets than possible keys, but enough so this approach is still better than a linked list.

Ideally, the hash function assigns to each key a unique value, but often this is not possible. This causes collisions, when two or more keys map to the same index. The better the hash function and the more buckets we use, the lower the number of collisions.

There are many ways to handle collisions. One is to save a linked list for each bucket, and if we get a collision, we simply append the new key-value pair to the list in that bucket.

Bloom Filters

Another application of hash functions are *Bloom Filters*. Those are data structures whose job it is to save a boolean value for each key. Instead of using an array and a single hash function to calculate an index into the array where we read the value, Bloom Filters use multiple hash functions at once.

When we insert an element into the Bloom Filter, we run it through all hash functions and calculate indices into our array and write a 1 into each of those positions. When we later ask ourselves whether we have already seen a certain key, we simply check whether all corresponding entries in the array are 1.
Note that Bloom Filters cannot remove elements (so they are usually cleared periodically), and can give false positives (say that we have seen an element when we have not). But they are still very useful when we want to quickly determine whether we have a certain web resource cached, for example.

3.7 Smallest Enclosing Circle

Definition 3.7.1 (Smallest Enclosing Circle) *Given a set of points P in the plane with $|P| = n$ (i.e. $P = \{p_1, ..., p_n\}, p_i = (x_i, y_i) \in \mathbb{R}^2$) the Smallest Enclosing Circle problem asks us to find a circle $C(P)$ with minimum possible radius r such that all points are contained in this circle.*

The following holds:

- points are allowed to be on the boundary
- the smallest enclosing circle is unique
- for any set P with $|P| \geq 3$, there exists a subset $Q \subseteq P$ with $|Q| = 3$ such that $C(P) = C(Q)$. The points in Q determine C uniquely.

Naive Algorithm

A trivial (and inefficient algorithm) is to simply go over all possible sets Q, compute the enclosing circle in constant time, check if it contains all points, and if so return it. In this case, we are guaranteed to have found the smallest enclosing circle. However, the algorithm has runtime $\mathcal{O}(n^4)$.

```
1 for all Q ∈ (P choose 3) do
2     compute C(Q)
3     if P ⊆ C(Q)
4         return C(Q)
```

Algorithm 3.8: Naive smallest enclosing circle - "CompleteEnumeration"

There are two more algorithms ("SmartEnumeration" and "Randomized_PrimitiveVersion") that are not particularly interesting or fast.

Randomized Algorithm

A better algorithm is to pick points randomly. Every time a point is found to be outside the circle, we know that with higher probability it will be on the border instead of those that are contained. Therefore, we increase the probability of picking it in the future by duplicating it.

```
1 while true
2     pick Q ∈ (P 11) uniformly at random
3     compute C(Q)
4     if P ⊆ C(Q)
5         return C(Q)
6     double all points outside C(Q)
```

Algorithm 3.9: Randomized smallest enclosing circle - "Randomized_CleverVersion"

The algorithm computes the smallest enclosing circle in expected time $\mathcal{O}(n \log n)$. The drastic difference comes from the fact that we make points that are unlikely to be inside $C(Q)$ exponentially more likely to be chosen. We pick 11 to strike a balance between having too many points outside $C(Q)$ (and having to deal with a lot of points) and having a bad runtime because calculating $C(Q)$ takes too long.

3.8 Convex Hull

Definition 3.8.1 *A set S is called* convex *if and only if, for every two points p_1, p_2 contained in the set S, all the points on the straight line connecting the two points are also included in the set S. Formally*

$$S \text{ convex} \iff \forall p_1, p_2 \in S, t \in [0,1] : p_1 + t(p_2 - p_1) \in S$$

Given a set of points P, the *convex hull* $conv(P)$ of P is the smallest convex set that contains P. In the plane, the convex hull is always delimited by straight lines between points in P.

We describe a convex hull by the points in P that are used in the border, starting anywhere and moving counterclockwise around the set.

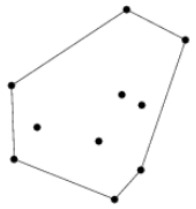


Figure 3.1: Convex hull of a set of points

Jarvis Wrap

For simplicity, we here assume that all points are in *general position* (allgemeiner Lage), meaning that no three points on the same line and no two points have the same x-coordinate.

The idea of the algorithm is to start with a point p_0 guaranteed to be on conv(P) (*e.g.* the left-most point) and iteratively looking for a next point such that all other points are left of the connecting line to the next point,

which guarantees that it will also be on conv(P). This point can be found in $\mathcal{O}(n)$ by iterating through the other points and keeping the one which is the rightmost (this is FindNext in the lecture script).

Informally, the algorithm proceeds as follows:

1. Select point p_0 that lies on convex hull (*e.g.* lowest x-coordinate)
2. Find next point p_{i+1}, such that all other points lie left of the line going through p_i, p_{i+1}.
3. Repeat until you reach p_0 again

This algorithm computes the convex hull in $\mathcal{O}(n \cdot h)$, where h is the number of points that lie on the convex hull's edge. This because the algorithm takes $\mathcal{O}(n)$ for each iteration, and it requires h loop iterations. This time isn't optimal, and there are algorithms that perform much better. Moreover, some modifications are required to address our general position assumption, but the runtime does not change.

Local Repair

LocalRepair is another algorithm for the convex hull problem. Here the idea lies in starting with a polygon without self-intersections that contains all of P and progressively fixing parts of it that are not convex. During the algorithm, we'll keep as invariants that we never have self-intersections and that no points are below the vertex sequence from p_1 to p_n and none are above the sequence from p_n to p_1. If we then manage to construct a locally convex polygon (only left turns), we have constructed the convex hull.
Note that both of the invariants are necessary for this to work, otherwise we might be locally convex but not globally convex (due to self-intersections and "loops") or we might not enclose all points.

The algorithm proceeds as follows:

1. Sort all the points in P by ascending x coordinate to give $p_1, \ldots, p_n$
2. Start with the polygon defined by the border $\langle p_1, \ldots, p_{n-1}, p_n, p_{n-1}, \ldots, p_2 \rangle$
3. Start at p_1 and move towards p_n. Every time we make a right turn (p_{i+2} is right of the line through p_i, p_{i+1}), we remove the middle vertex (p_{i+1}) and check again for possibly newly created right turns starting at the previous vertex (p_{i-1}).
4. Once we arrive at p_n, we repeat the same process to go back towards p_2.

This algorithm has runtime $\mathcal{O}(n \log n)$ including sorting and runtime $\mathcal{O}(n)$ if the vertices are already sorted.
Examples and illustrations as well as an implementation can be found in the lecture script on page 211.

Exercise Solutions 4

4.1 Graph Theory

Ex 1 We use the given coloring c as follows:
Let C_i be the set of all vertices colored with i, then the sequence we use is $C_1, C_2, C_3, \ldots, C_k$ (order of the elements in those sets doesn't matter).

This works because the greedy algorithm will assign the smallest color not in use by neighbors. Thus, we will always be able to use the color from c (no neighbors are colored the same) or a smaller one (because bigger colors have not yet been used). Which gives us a coloring with $\leq k$ colors.

Ex 2 We won't provide a solution here, as there could be many valid proofs. Just show us your solution if you're unsure.
It's probably easiest to prove flow conservation for each vertex using two nested case distinctions about whether inflow increases or decreases and whether outflow increases or decreases after augmentation. The capacity constraints should follow from the definition of ε and the capacities in N_f.
The strictly bigger value can be proved by an increase of netoutflow in s.

Ex 3

1. Consider the graph C_3. A maximum flow in the network Sheryl builds has value 3, indicating that a perfect matching should exist. However, a maximum matching in this graph has only cardinality 1.
2. Consider the graph $G = (\{a, b, c, d\}, \{a, b\} \times \{c, d\})$ and the network build (correctly) from it. A maximum matching has cardinality 2 and the maximum flow also has value 2 - so far so good.
   One way to get a flow value of 2 is to assign edges from s to a, b flow 1, each edge between a, b and c, d flow 0.5 as well as each edge from c, d to t flow 1.
   This is a maximum flow that is not an integer flow.
3. Because we have only integer capacities, a maximum integer flow exists. It could be found, for example, using Ford-Fulkerson.

Ex 4 The network is $N = (V, A, c, u, v)$ where $c(e) = 1$ for all edges e and A is built by adding two edges for each edge in E, one for each direction.
The amount of edge-disjoint $u - v$ paths is given by the value of a maximum flow.

You can prove this by *looking at an integer maximum flow* and showing how to construct the edge-disjoint paths from it. And also showing how to construct a maximum flow from all the edge-disjoint $u - v$ paths.

4.2 Probability Theory

Ex 5

1. $\Omega = [6]^2, \forall \omega : \Pr[\omega] = 1/36$
2. $\Omega = \binom{[6]}{2} \cup [6], \Pr[\omega] = |\omega|/36$
3. $\Omega = \{2, 3, \ldots, 12\}, \Pr[\omega] = \frac{6-|7-\omega|}{36}$

Ex 6 Repetition allowed, order matters: Rolling an n-sided die k times.

Repetition allowed, order doesn't matter: Choosing which k items to buy from a store that has unlimited amounts of n different items.

No repetition allowed, order matters: The number of possible first k places on a scoreboard of n people.

No repetition allowed, order doesn't matter: Drawing k cards from a deck of n cards.

Ex 7 Let A, B be independent events with $\Pr[A] \neq 1/2$, because $\Pr[A \mid \overline{B}] = \Pr[A] = \Pr[A \mid B]$.
For example, we could use a fair six-sided die and the events $A = \{3, 6\}$ and $B = \{1, 2, 3\}$.

Ex 8 We use Total Probability:

$$\begin{aligned}
\Pr[correct] &= \Pr[correct \mid known] \cdot \Pr[known] \\
&\quad + \Pr[correct \mid \overline{known}] \cdot \Pr[\overline{known}] \\
&= 1 \cdot p + \frac{1}{4} \cdot (1 - p) \\
&= p + \frac{1}{4} - \frac{1}{4} \cdot p \\
&= \frac{3}{4}p + \frac{1}{4}
\end{aligned}$$

Ex 9 We define the probability space by looking at all possibilities for the car position (c), our guess (g) and the host's choice of door to open (h):

$$\Omega = \{(c, g, h) \in [3]^3 \mid h \neq c \wedge h \neq g\}$$

Because the host sometimes has no choice and sometimes chooses the door with probability $\frac{1}{2}$, we get:

$$\Pr[(c, g, h)] = \begin{cases} \frac{1}{9} & \text{if } c \neq g \\ \frac{1}{18} & \text{if } c = g \end{cases}$$

So for example, say we choose door 2 and the host opens door 3 (you could also do this more generally), we look at the conditional probability of finding the car behind our door:

$$\begin{aligned}\Pr[c = 2 \mid g = 2 \wedge h = 3] &= \frac{\Pr[c = 2 \wedge g = 2 \wedge h = 3]}{\Pr[g = 2 \wedge h = 3]} \\ &= \frac{\Pr[\{(2,2,3)\}]}{\Pr[\{(1,2,3),(2,2,3)\}]} \\ &= \frac{1/18}{3/18} = \frac{1}{3}\end{aligned}$$

Ex 10 We work backwards and try to stay fairly close to the terms in the question. We first notice that two events are independent if $\Pr[A] = \Pr[A \mid B]$. To get $\Pr[A]$ using $\Pr[A \mid B]$ and $\Pr[A \mid \overline{B}]$, we use Total Probability:

$$\Pr[A] = \Pr[A \mid B] \cdot \Pr[B] + \Pr[A \mid \overline{B}] \cdot \Pr[\overline{B}]$$

And using the assumption given in the question:

$$\begin{aligned}\Pr[A] &= \Pr[A \mid B] \cdot \Pr[B] + \Pr[A \mid B] \cdot \Pr[\overline{B}] \\ &= \Pr[A \mid B] \cdot \left(\Pr[B] + \cdot \Pr[\overline{B}]\right) \\ &= \Pr[A \mid B] \cdot 1\end{aligned}$$

And that concludes the proof.

Ex 11 We first define a random variable X as the number of edges within both colors. We define an indicator random variable X_i for each of the $n(n-1)$ edges that run within colors. Each indicator random variable has expected value $\frac{1}{2}$ and we can write X as their sum. Thus:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n(n-1)} X_i\right] = \sum_{i=1}^{n(n-1)} \mathbb{E}[X_i] = n(n-1)\frac{1}{2} = \frac{n(n-1)}{2}$$

We used Linearity of Expectation, but not independence here.

Ex 12 We define X just as above and use the hint:

$$\begin{aligned}\mathbb{E}[X^2] &= \mathbb{E}\left[\left(\sum_{i=1}^{n(n-1)} X_i\right)^2\right] \\ &= \mathbb{E}\left[\sum_{i=1}^{n(n-1)} \sum_{j=1}^{n(n-1)} X_i \cdot X_j\right] \\ &= \sum_{i=1}^{n(n-1)} \sum_{j=1}^{n(n-1)} \mathbb{E}\left[X_i \cdot X_j\right]\end{aligned}$$

To calculate the expected value, we have to differentiate two cases: When $i \neq j$, the two indicator random variables are independent (it says so in the question) and we can do $\mathbb{E}\left[X_i \cdot X_j\right] = \mathbb{E}\left[X_i\right] \cdot \mathbb{E}\left[X_j\right]$.
If, however $i = j$, then they are dependent, and we have to do something

else. Here, we notice that an indicator random variable squared is exactly the same indicator random variable (since it only takes values 0 and 1). There are $n(n-1)$ terms in the first case, each with expected value $\frac{1}{4}$ and n terms in the second case, each with expected value $\frac{1}{2}$. We get:

$$\begin{aligned}\mathbb{E}[X^2] &= \sum_{i=1}^{n(n-1)} \sum_{j=1}^{n(n-1)} \mathbb{E}\left[X_i \cdot X_j\right] \\ &= n(n-1)\frac{1}{4} + n\frac{1}{2} \\ &= \frac{n^2-n}{4} + \frac{n}{2} \\ &= \frac{n^2+n}{4}\end{aligned}$$

Ex 13 This exercise is mostly to show that we don't actually need to know Ω to calculate expected value and variance if we have the distribution. We can calculate the expected value as:

$$\mathbb{E}[X] = \sum_{x \in W_X} x \cdot f_X(x) = 4$$

We can get the variance as $\mathrm{Var}[X] = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2$. To calculate the first term, we can view it as a new random variable whose distribution is derived from X:

x^2	16	4	25	64	100
$f_{X^2}(x^2)$	0.25	0.10	0.20	0.15	0.30

And we can calculate this expected value in the same way as before to get:

$$\mathbb{E}[X^2] = \sum_{x \in W_{X^2}} x \cdot f_{X^2}(x) = 49$$

We now have:

$$\begin{aligned}\mathbb{E}[X] &= 4 \\ \mathrm{Var}[X] &= 49 - 4^2 = 33\end{aligned}$$

And using the Linearity of Expectation and the corresponding rule for variance:

$$\begin{aligned}\mathbb{E}[2X+8] &= 2 \cdot \mathbb{E}[X] + 8 = 16 \\ \mathrm{Var}[2X+8] &= 4 \cdot \mathrm{Var}[X] = 132\end{aligned}$$

Ex 14

1. 2 independent rolls of the same die. For example modelled as:

$$\begin{aligned}\Omega &= [6]^2 & \forall \omega : \Pr[\omega] &= \frac{1}{36} \\ X_1((a,b)) &:= a & X_2((a,b)) &:= b\end{aligned}$$

2. Rolling not two identical dice but maybe a four-sided one and a twenty-sided one.

3. Looking at the same die roll as three random variables $X_1 = X_2 = X_3$.

Ex 15 We calculate the variance again as:

$$\mathrm{Var}[X] = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2$$

And we get (Bernoulli variables are the same when squared):

$$\mathrm{Var}[X] = p - p^2 = p(1-p)$$

Ex 16 We first define a variable Y for the number of victories, and we see that it has a binomial distribution (unlike X): $Y \sim \mathrm{Bin}(25, 0.6)$.
We can write $X = 2 \cdot Y + 25$ and because

$$\begin{aligned}\mathbb{E}[Y] &= np = 15\\ \mathrm{Var}[Y] &= np(1-p) = 6\end{aligned}$$

we know that:

$$\begin{aligned}\mathbb{E}[X] &= 2 \cdot \mathbb{E}[Y] + 25 = 55\\ \mathrm{Var}[X] &= 4 \cdot \mathrm{Var}[Y] = 24\end{aligned}$$

Ex 17 Because the Geometric distribution only takes on natural numbers, we can use:

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

Giving us (with a little Analysis):

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=1}^{\infty} \Pr[X \geq i]\\ &= \sum_{i=1}^{\infty} (1-p)^{(}i-1)\\ &= \sum_{i=0}^{\infty} (1-p)^i\\ &= \frac{1}{1-(1-p)} = \frac{1}{p}\end{aligned}$$

Ex 18 This is the modified calculation. You need to watch out not to make a mistake when rewriting the sum.

$$\begin{aligned}\mathbb{E}[X] = \sum_{i=1}^{n/2} \mathbb{E}[X_i] &= \sum_{i=1}^{n/2} \frac{n}{n-i+1} = n \sum_{i=1}^{n/2} \frac{1}{n-i+1} = n \sum_{i=n/2+1}^{n} \frac{1}{i}\\ &= n \cdot \left(H_n - H_{n/2}\right)\\ &= n \cdot (\ln n + \mathcal{O}(1) - \ln(n/2) - \mathcal{O}(1))\\ &= n \cdot (\ln n + \mathcal{O}(1) - \ln n + \ln 2 - \mathcal{O}(1)\\ &= n \cdot \mathcal{O}(1) = \mathcal{O}(n)\end{aligned}$$

Ex 19 Any random variable with negative expected value will break Markov (probabilities can't be negative after all). It's also possible to construct a random variable with positive expected value that breaks Markov as follows:

1. Pick some $\mathbb{E}[X]$ that is pleasing to you
2. Pick some $t > \mathbb{E}[X]$ that you want to use later
3. Pick probability $p < 1$ you like for the event $X \geq t$
4. Now define a probability space to make this all work as follows:

$$\Omega = \{-y, t\} \qquad \Pr[-y] = 1 - p, \Pr[t] = p$$

5. And finally solve for the variable y:

$$-y \cdot (1-p) + t \cdot p = \mathbb{E}[X] \iff \frac{t \cdot p - \mathbb{E}[X]}{1-p} = y$$

Ex 20 First off, let X be the number of correct answers. It is clear that $X \sim \text{Bin}(18, 0.5)$. That means we have $\mathbb{E}[X] = 9$ and $\text{Var}[X] = \frac{9}{2}$.

The true probability is $\sum_{k=5}^{1} 3\binom{18}{k}\frac{1}{2^18}$ (no closed form expression for that sum exists) and if we were to calculate it, we'd get $\frac{7939}{8192} \approx 0.969$.

- **Markov** cannot be used because it only gives us an upper bound for the probability to have too big values but here we also need to have an upper bound for the probability of too small values, and you cannot subtract two upper bounds to get another upper bound (do it step by step if that's not clear).
- **Chebyshev** can be used to calculate an upper bound for the probability of the complementary event:

$$\begin{aligned} &\Pr\left[|X - \mathbb{E}[X]| \geq t\right] \leq \frac{\text{Var}[X]}{t^2} \\ \implies &\Pr\left[|X - 9| \geq 5\right] \leq \frac{9/2}{25} = \frac{9}{50} \\ \implies &\Pr\left[|X - 9| < 5\right] \geq 1 - \frac{9}{50} = \frac{41}{50} = 0.82 \end{aligned}$$

- **Chernoff** can be used once for the probability that X is too small (≤ 4) and once for the case where X is too big (≥ 14).

$$\begin{aligned} &\Pr\left[X \geq (1+\delta)\mathbb{E}[X]\right] \leq e^{-\frac{1}{3}\delta^2\mathbb{E}[X]} \\ \implies &\Pr\left[X \geq 14\right] \leq e^{-\frac{1}{3}\cdot\left(\frac{5}{9}\right)^2\cdot 9} = e^{-25/27} \end{aligned}$$

$$\begin{aligned} &\Pr\left[X \leq (1-\delta)\mathbb{E}[X]\right] \leq e^{-\frac{1}{2}\delta^2\mathbb{E}[X]} \\ \implies &\Pr\left[X \leq 4\right] \leq e^{-\frac{1}{2}\cdot\left(\frac{5}{9}\right)^2\cdot 9} = e^{-25/18} \end{aligned}$$

We can now add those upper bounds together to get the probability

of the complementary event again (yes, adding bounds is fine).

$$\begin{aligned} &\Pr[|X-9| \geq 5] = \Pr[X \geq 14] + \Pr[X \leq 4] \\ \implies &\Pr[|X-9| \geq 5] \leq e^{-25/27} + e^{-25/18} \\ \implies &\Pr[|X-9| < 5] \geq 1 - e^{-25/27} - e^{-25/18} \approx 0.646 \end{aligned}$$

To conclude, of course both bounds are correct. Chebyshev is better in this case, because we're very "close" to the expected value and n is very small, but there's no general rule to know which is better.

Ex 21

1. $X \sim \text{Bin}(45'000, \frac{1}{3} + \frac{1}{5} = \frac{8}{15})$. Thus the expected value is $\mathbb{E}[X] = np = 45'000 \cdot \frac{8}{15} = 24'000$.
2. We use Chernoff to get bounds for $\Pr[X \geq 1.2\mathbb{E}[X]]$ and $\Pr[X \leq 0.9\mathbb{E}[X]]$:

$$\begin{aligned} &\Pr[X \geq (1+\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{3}\delta^2\mathbb{E}[X]} \\ \implies &\Pr[X \geq 1.2\mathbb{E}[X]] \leq e^{-\frac{1}{3}\cdot 0.2^2 \cdot 24000} = e^{-320} \end{aligned}$$

$$\begin{aligned} &\Pr[X \leq (1-\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{2}\delta^2\mathbb{E}[X]} \\ \implies &\Pr[X \leq 0.9\mathbb{E}[X]] \leq e^{-\frac{1}{2}\cdot 0.1^2 \cdot 24000} = e^{-120} \end{aligned}$$

And we get:

$$\begin{aligned} &\Pr\left[\overline{0.9\mu < X < 1.2\mu}\right] = \Pr[X \geq 1.2\mathbb{E}[X]] + \Pr[X \leq 0.9\mathbb{E}[X]] \\ \implies &\Pr\left[\overline{0.9\mu < X < 1.2\mu}\right] \leq e^{-320} + e^{-120} \approx 7.67 \cdot 10^{-53} \end{aligned}$$

3. Using the normal inequality is not possible because we'd need to choose a $\delta > 1$.
   Using the third inequality instead (which we're allowed to do because $300 = 6 \cdot 50 \geq 6 \cdot \mathbb{E}[X] \geq 2e \cdot \mathbb{E}[X]$) gives:

$$\Pr[X \geq 300] \leq 2^{-300} \approx 4.9 \cdot 10^{-91}$$

4.3 Randomized Algorithms

Ex 22 Only the first algorithm is useful. As soon as we get 50% success probability, we might as well flip a coin.

Ex 23 Now all of these algorithms are suddenly useful!

Ex 24 We can stop the execution of the Las Vegas algorithm after a predetermined time and return any (possibly wrong) result we like.
The other direction is not possible (we cannot produce results that are always correct from results that are only sometimes correct).

Ex 25 The answer "not prime" is always correct. Thus, we design the algorithm as follows:

```
1 for i = 1, ..., N do
2     if MillerRabin(n) = "not prime"
3         return "not prime"
4 return "prime"
```

Correctness proof:
As we know, the Miller Rabin test has one-sided error with a success probability of $\varepsilon = 3/4$. We can use the Theorem we saw earlier to conclude that our algorithm has a 99.99% success probability by choosing $\delta = 0.0001$ and calculating N as:

$$N = \varepsilon^{-1} \ln \delta^{-1} = \frac{4}{3} \ln(10^4) = \frac{16}{3} \ln 10$$

And because $\ln 10 \approx 2.303 \leq 3$ it suffices to use:

$$N = \frac{16}{3} \cdot 3 = 16$$

(because this implies $N \geq \varepsilon^{-1} \ln \delta^{-1}$)

Runtime analysis:
Because we do a constant amount of iterations, the runtime of this algorithm is $\mathcal{O}(T_{MR})$ where T_{MR} is the runtime of the Miller Rabin test.