Algorithms & Probability - Essentials

Simon Sure
https://simonsure.com

June 25, 2023

For comments/improvement suggestions etc. feel free to contact me at any time. I give NO guarantee regarding correctness or completeness for this content. The content is strongly influenced by the course and contains e.g. images from it. However, the script is not affiliated with the course.

Contents

I Graph Theory 4

1 Fundamentals 4

2 Trees 4

3 Paths 4

4 Connectivity 5
- 4.1 cut vertex (”Artikulationsknoten”) 5
- 4.2 cut edge (”Brücke”) 7
- 4.3 block decomposition (”Block-Zerlegung”) 7

5 Cycle graphs 8
- 5.1 Eulerian trial 8
- 5.2 Hamiltonian cycle (”Hamiltonkreis”) 9
  - 5.2.1 Special Cases 10
  - 5.2.2 Exponential Algorithm 10
- 5.3 NP 12
- 5.4 Traveling Salesman Problem 12
  - 5.4.1 NP completeness 13
  - 5.4.2 metric TSP 13
  - 5.4.3 general TSP approximation 14

6 Matchings (”Matchings”) 14
- 6.1 Maximal Matchings - Greedy Algorithm 15
- 6.2 Maximum Matchings 15
  - 6.2.1 Bipartite graphs 15
  - 6.2.2 General graphs 16
  - 6.2.3 weighted graphs 16
- 6.3 (Perfect) Matchings in bipartite graphs 16

7 coloring **18**
7.1 Greedy algorithm . . . 19
7.2 Brook's Theorem . . . 20
7.3 3-colorable graphs . . . 21
7.4 additional stuff . . . 22

II Probability & Randomized Algorithms **23**

8 Fundamentals **23**

9 Conditional Probability **24**
9.1 Common Problems . . . 26

10 Independence **27**
10.1 pseudorandom number generators ("Pseudozufallsgeneratoren") . . . 29

11 Random Variables **29**
11.1 Expected Value . . . 29
11.1.1 Conditional Random Variables . . . 30
11.1.2 Linearity of the Expected Value . . . 31
11.1.3 Proof of the Inclusion/Exclusion Principle . . . 32
11.2 Variance . . . 32
11.3 Examples . . . 33

12 Important Discrete Distributions **34**
12.1 Bernoulli distribution ("Bernoulli-Verteilung") . . . 34
12.2 Binomial distribution ("Binomial-Verteilung") . . . 34
12.3 Poisson distribution ("Poisson-Verteilung") . . . 35
12.4 Geometric distribution . . . 35
12.5 Coupon Collector . . . 36
12.6 Negative binomial distribution . . . 37

13 Multiple Random Variables **37**
13.1 Independence of Random Variables . . . 38
13.2 Compound Random Variables ("zusammengesetzte Zufallsvariablen") . . . 39
13.3 Moments of Compound Random Variables . . . 39
13.4 Wald's Equations . . . 40

14 Estimating Probabilities **40**
14.1 Markov's Inequality . . . 40
14.2 Chebyshev's Inequality . . . 41
14.3 Chernoff's Inequality . . . 41
14.4 Comparison of Inequalities . . . 42

15 Randomized Algorithms **43**
15.1 Las-Vegas Algorithms . . . 43
15.2 Monte-Carlo Algorithms . . . 44
15.3 Prime Number Test . . . 45
15.3.1 Euclid Prime Test . . . 45
15.3.2 Fermat Prime Test . . . 46

15.3.3 'not prime' certificate, i.e., "algorithm efficiency summary" 46
15.3.4 Miller-Rabin certificate 47
15.3.5 Remarks 47
15.4 Target Shooting 48
15.5 QuickSort 49
15.6 QuickSelect 50
15.7 Finding Duplicates 51

III Algorithms **52**

16 Longest Paths **52**
16.1 Colorful Paths ("Bunte Pfade") 53
16.2 Long Path 54
16.3 Remarks 55

17 Network Flows **55**
17.1 Cuts 56
17.2 Augmenting Paths 58
17.3 Residual Network 58
17.4 Algorithm 59
17.5 Applications 60
17.5.1 Bipartite Matchings 60
17.5.2 Edge-disjunct Paths 61
17.5.3 Vertex-disjunct Paths 62
17.5.4 Image Segmentation 62
17.5.5 Flows and Convex Sets 63

18 Minimal Cut **63**
18.1 Flow-Based 64
18.2 Edge Contraction 64
18.3 Randomized Edge Contraction 65
18.4 Bootstrapping 67

19 Smallest Enclosing Disk **67**
19.1 Sampling Lemma 72

20 Convex Hull **73**
20.1 Algorithm: Jarvis Wrap 74
20.2 Algorithm: Local Improvements 76
20.3 Lower Bound 79

IV Optional **80**

Part I
Graph Theory

This is a general overview of common definitions:

English term	German term	meaning
walk	Weg/Kantenzug	sequence of vertices
trail		walk with unique edges
path	Pfad (?)	walk with unique vertices
closed walk	Zyklus	walk with same start and end vertex
circuit		closed walk with unique edges
cycle	Kreis	close walk with unique vertices[1]

Figure 1

- Eulerian trail, sometimes Eulerian path (”Eulerzyklus”/”Euler-Tour”)
- Hamiltonian Path/Hamiltonian Cycle (”Hamiltonkreis”)

1 Fundamentals

The fundamentals of graph theory do majorly overlap with what has already been discussed in ”Algorithms & Data Structures”. Hence, it will not be reiterated here.

2 Trees

The same applies to trees. As part of this section, Prim’s Algorithm, Kruskal’s Algorithm, and more is discussed. All relevant topics should still be remembered from the previous course ”Algorithms & Data Structures”.

3 Paths

Again, the same holds for shortest paths. In ”Algorithms & Data Structures” some various appraoches and algorithms were discussed, including:

- Dijkstra’s Algorithm
- Floyd-Warshall Algorithm
- Bellman-ford Algorithm
- Johnson’s Algorithm

[1] without start/end vertex

4 Connectivity

Here we consider connectivity. The basic concept of a connected graph has also been already discussed. But the notion of connectivity will be extended here.

Definition connectivity ("Zusammenhang"): Let $G = (V, E)$ be a graph. G is connected ("zusammenhängend") $\Leftrightarrow \forall u, v \in V, u \neq v, \exists u$-$v$-path

Definition k-connectivity ("k-Zusammenhang"): Some graph $G = (V, E)$ is said to be k-connected ("k-zusammenhängend") if $|V| \geq k + 1$ and for all subsets $X \subseteq V$ with $|X| < k$, the graph $G[V \backslash X]$ is still connected. The complete graph with k vertices is $k - 1$ connected.

This about means that at lest k vertices must be removed so that G is not connected any more. Any k connected graph also is $k - 1$-connected. If G contains a vertex with degree $< k$, then G can not be k connected.

Definition vertex separator ("(Knoten-)Separator"): A subset $X \subseteq V$ of the vertices of some $G = (V, E)$ is a (vertex) separator ("(Knoten-)Separator") if $G[V \backslash X]$ is not connected.

Definition u-v-separator ("u-v-Separator"): If $u, v \in V$ and $X \subseteq V \backslash \{u, v\}$ is a vertex (sub)set so that u and v are in separate connected components of $G[V \backslash X]$, then X is called u-v-separator ("u-v-Separator").

Definition k-edge-connectivity ("k-Kanten-Zusammenhang"): Some graph $G = (V, E)$ is called k-edge-connected if for all subsets $X \subseteq E$ with $|X| \leq k$, the graph $(V, E \backslash X)$ is also connected.

In common terms: Some graph is k-edge-connected if at least k edges must be removed so that the graph is not connected any more. Such a set $X \subseteq V$ is called u-v-edge-separator ("u-v-Kantenseparator") if u and and v are in different connected components of $(V, E \backslash X)$.

Notice that: vertex connectivity $\leq$ edge connectivity $\leq$ minimal degree of vertices.

Theorem Menger: Let $G = (V, E)$ be some graph and $u, v \in V, u \neq v$. Then:

- Each u-v-vertex-separator has size at least $k \Leftrightarrow$ at least k intern-vertex-disjunct u-v-paths exist.
- Each u-v-edge-separator has size at least $k \Leftrightarrow$ at least k intern-edge-disjunct u-v-paths exist.

One direction of this theorem is quite clear: If there are k intern-disjunct u-v-paths, then we have at least k connectivity. The other direction will be shown later.

4.1 cut vertex ("Artikulationsknoten")

Definition cut vertex: In a 1-connected but not 2-connected graph, by definition, at least one vertex v must exist so that $G[V \backslash \{v\}]$ is not connected. Such v are called cut vertex ("Artikulationsknoten").

We now consider how cut vertices may be found. For that we consider DFS with its pre-order and post-order. We assign each edge of the dfs-tree a direction according to how it has been traversed during the dfs algorithm. All remaining edges are assigned the direction according to how they are first discovered, i.e., from the higher pre-order-valued to the lower pre-order-valued vertex.

Then, we assign each vertex v the value $low[v]$, which is the smallest dfs value which may be reached with a directed path from v that uses arbitrarily many tree edges and at most oen of the remaining edges as the last edge of the past. Formally:

$$low[v] := \min\left(dfs[v], \min_{(v,w)\in E} \begin{cases} dfs[w], & (v,w) \text{ is remaining edge} \\ low[w], & (v,w) \text{ is tree edge} \end{cases} \right)$$

To compute those *low* values we proceed as follows

- compute dfs/pre-order values with dfs
- initialize low values with dfs values
- compute all low values in post-order/reverse pre-order

Then, v is a cut vertex if

- $v \neq$ root: v has a child in the dfs tree with $low[u] \geq pre[v]$
- $v =$ root: v has at least two children in the dfs tree

Proof. We consider some $v \neq$ root. Notice that the subtrees of the children of v may not be connected among each other. If they were, there would only be one child as all the vertices of that one subtree would be reachable (and would be reached during dfs) from the other subtree.

Second, the subtree of a child with a higher dfs value than v may only be connected to an area with lower dfs values with a remaining edge as otherwise that area would have lower dfs values

Then, $low[u] \geq dfs[v]$ implies that there is no remaining (back) edge from some child's subtree with higher dfs-value to a lower-dfs subtree. Hence, if v is removed, that subtree has no connection to the remaining graph and v must be a cut vertex.

The special case $v =$ root is obvious. □

Theorem: For some connected graph $G = (V, E)$ (adjacency list), all cut vertices may be computed in $\mathcal{O}(|E|)$.

This is a simple implementation

```
DFS-Visit(G,v):
  num = num + 1
  dfs[v] = num
  low[v] = dfs[v]
  isCutVertex[v] = false
  for all {v,w} in E:
    if dfs[w] = 0:
      T = T + {v,w}
      val = DFS-Visit(G,w)
      if val >= dfs[v]:
        isCutVertex[v] = true
      low[v] = min{low[v], val}
    else dfs[w] != 0 and {v,w} not in T:
      low[v] = min{low[v], dfs[w]}
  return low[v]

DFS(G,s):
  num = 0
  T = empty set
  DFS-Visit(G,s)
  if deg(s) in T >= 2:
    isCutVertex[s] = true
  else:
    isCutVertex[s] = false
```

4.2 cut edge (”Brücke”)

Definition cut vertex: In a 1-edge-connected but not 2-edge-connected graph, by definition, at least one edge e must exist so that $(V, E\backslash\{e\})$ is not connected. Such e are called cut edge (”Brücke”).

Lemma: Let $G = (V, E)$ be a connected graph. If $\{x, y\} \in E$ is a bridge/cut edge, then $\deg(x) = 1$ or x is a cut vertex.

To determine cut edges, we can use the same approach/algorithm idea as to determine cut vertices. Specifically, we can compute the *low* values identically. For some edge $e = (v, w)$ in the dfs-tree including directed remaining edges, e is a cut edge if and only if $low[w] > dfs[v]$.

This is as then there is no other connection than this edge which connects the subgraph of w to the subgraph on the other side of v. Remaining edges are obviously never

4.3 block decomposition (”Block-Zerlegung”)

Definition blocks (”Blöcke”): Let $G = (V, E)$ be a connected graph. For $e, v \in E$:

$$e \sim f :\Leftrightarrow \begin{cases} e = f \\ \exists \text{ common circle through } e \text{ and } f \end{cases}$$

This is an equivalence relation (”Äquivalenzrelation”). Its equivalence classes (”Äquivalenzklassen”) are called blocks (”Blöcke”).

Notice that this is defined on the edges, not vertices!

Proof. Reflexivity is given by definition.

Symmetry is trivially given.

Transitivity: We must only consider $e \neq f, e \neq g, f \neq g$. With $e \sim f, f \sim g$, the circles with e and f as well as f and g may have shared edges, which makes this more complicated. We reason like this:

$$\begin{gathered}\exists \text{ two edge disjunct paths between } v_e \text{ and } v_f \\ \wedge\exists \text{ two edge disjunct paths between } v_f \text{ and } v-g \\ \\ \Rightarrow \neg\exists \text{ edge separator between } v_e \text{ and } v_f \text{ of size 1 from } V\backslash\{v_e, v_f\} \\ \wedge\neg\exists \text{ edge separator between } v_f \text{ and } v_g \text{ of size 1 from } V\backslash\{v_f, v_g\} \\ \\ \Rightarrow \neg\exists \text{ edge separator between } v_e \text{ and } v_g \text{ of size 1 from } V\backslash\{v_e, v_g\} \\ \Rightarrow \exists \geq \ 2 \text{ vertex disjunct } v_e\text{-}v_g\text{-paths (MENGER)} \\ \Rightarrow \exists \text{ a circle}\end{gathered}$$

$\square$

Lemma: Two blocks can only intersect in one cut vertex (if at all). I.e., there are not edges between equivalence classes.

We now replace each block with one vertex for that block. Those block vertices are then connected only to cut vertices. Based on that we define:

Definition block decomposition ("Block-Zerlegung"): T is a bipartite graph with vertex set $V = A \uplus B$ with A being the set of cut vertices and B corresponding to the set of blocks. We define

$$\{a \in A, b \in B\} \in E :\Leftrightarrow a \text{ incident to } b$$

5 Cycle graphs

5.1 Eulerian trial

Definition Eulerian trial ("Eulertour"/"Eulerzyklus"): A closed walk in $G = (V, E)$ ("Zyklus") which contains every edge exactly once.

The algorithm to find Eulerian trials has been extensively discussed in a previous course.

Theorem:

- A connected graph $G = (V, E)$ is Eulerian if and only if all degrees are even.
- In a connected Eulerian graph, an Eulerian cycle may be found in $\mathcal{O}(|E|)$.

The proof is also omitted as extensively discussed in a previous course.

Theorem Ore's Theorem: Let $G = (V, E)$, $|V| \geq 3$. If $\forall v, w \in V$, v not adjacent to w we have $deg(v) + deg(w) \geq |V|$, then G is Hamiltonian.

This was not mentioned in the lecture but is quite useful. Accordingly, this also has not been proven.

5.2 Hamiltonian cycle (”Hamiltonkreis”)

Definition Hamiltonian cycle (”Hamiltonkreis”): A cycle in $G = (V, E)$, which contains every vertex exactly once.

Finding Hamiltonian cycles is significantly harder than finding Eulerian trials. The problem ahs been proven to be NP complete.

Theorem Inclusion-exclusion principle (”Siebformel”/”Prinzip der Inklusion/Exklusin”): For finite sets $A_1, ..., A_n$ with $n \geq 2$:

$$|\cup_{i=1}^{n} A_i| = \sum_{l=1}^{n} (-1)^{l+1} \sum_{1 \leq i_1 \leq ... \leq i_l \leq n} |A_{i_1} \cap ... \cap A_{i_l}|$$

$$= \sum_{i=1}^{n} |A_i| - \sum_{1 \leq i_1 < i_2 \leq n} |A_{i_1} \cap A_{i_2}|$$

$$+ \sum_{1 \leq i_1 < i_2 < i_3 \leq n} |A_{i_1} \cap A_{i_2} \cap A_{i_3}| - ... + (-1)^{n+1} |A_1 \cap ... \cap A_n|$$

This will be formally proven later.

Theorem Direc’s Theorem (”Satz von Dirac”): Each graph $G = (V, E)$ with $|V| \geq 3$ and minimal degree $\delta(G) \geq \frac{|V|}{2}$

Proof. (correctness)

When we discuss some k-(cycle-)path here, k denotes the number of vertices, not of the number of edges!

First, we show that G is connected. Consider arbitrary $u, v \in V$:

- $\{u, v\} \in E$: trivial
- $\{u, v\} \notin E$ and $u = v$: trivial
- $\{u, v\} \notin E$ and $u \neq v$

  If $|N(u) \cap N(v)|$ is not empty, then there is a path of length two between u and v.

  Using the inclusion-exclusion principle: $|N(u) \cap N(v)| = |N(u)| + |N(v)| - |N(u) \cup N(v)|$. By assumption the first two summands are $\geq \frac{|V|}{2}$. The latter can be at most $|V| - 2$. Hence, the sum is ≥ 2 and u and v are connected via path of length 2.

Second: k-cycle-path in $G \Rightarrow k + 1$-path in G (if $k < N$). This follows directly from connectivity.

Third: k-path in $G \Rightarrow k + 1$-path in G or k-cycle-path in G.

- $N(v_1) \not\subseteq \{v_2, ..., v_k\}$: $k + 1$-path exists
- $N(v_k) \not\subseteq \{v_1, ..., v_{k-1}\}$: $k + 1$-path exists
- $N(v_1) \subseteq \{v_2, ..., v_k\}$ and $N(v_k) \subseteq \{v_1, ..., v_{k-1}\}$

  We will show that $N(v_k)$ always has an element which is left from an element of $N(v_1)$ on the path. We see trivially that can then construct a k-cycle.

With $L(v_1)$ we denote all elements left from elements of $N(v_1)$. From $|L(v_1) \cap N(v_k)| = |L(v_1)| + |N(v_k)| - |L(v_1) \cup N(v_k)|$ we learn that such a combination must exist. That is as $|L(v_1)|$ and $|N(v_k)|$ each must be $\geq \frac{|V|}{2}$, and the union must be $\leq |V| - 1$. Hence, the cardinality of the intersection is ≥ 1.

□

The runtime of this algorithm is $\mathcal{O}(|V|^2)$ as finding a longer cycle takes $\mathcal{O}(|V|)$ and we have to do that $|V|$ times.

5.2.1 Special Cases

Now we consider some special cases in which finding a Hamiltonian cycle is easier.

Grid

Theorem: A $n \times m$ grid has a Hamiltonian cycle if and only if $n \cdot m$ is even.

Proof.

- $\Rightarrow$: This is easily understood when recognizing that the grid is a chess board and the graph is bipartite with an uneven number of vertices.
- $\Leftarrow$: We can provide a constructive proof, which is very obvious.

□

We can generalize:

Lemma: If $G = (A \uplus B, E)$ is a bipartite graph with $|A| \neq |B|$, then G does not contain a Hamiltonian cycle.

Hyper Cube As motivation for this consideration one might think of rotary encoders. Those encode angles by having a circular sheet, which has holes along the radius. Each angle range is associated with a specific hole pattern. The question is whether we can arrange those holes in such a way that only one hole/bit changes in adjacent angle ranges to enable easy error detection.

We now consider a d-dimensional hyper cube H_d. It has the vertex set $\{0,1\}^d$ and an edge exists if both vertices differ in exactly one position. We ask whether H_d has a Hamiltonian cycle. The answer for $d \geq 2$ is yes.

Do provide a constructive proof. For $d = 2$ we have $00 - 10 - 11 - 01$. To extend one dimension, we simply add the inverse of this and append 0 to the first half/1 to the second half.

5.2.2 Exponential Algorithm

To compute Hamiltonian cycles in general one could simple try all combinations of which there are $\frac{1}{2}(n-1)! = \Theta(n!)$. We now improve that runtime to $\mathcal{O}(2^n)$, which is still bad but compared to $\Theta(n!)$ not so bad after all.

Version 1 Let $G = (V, E)$ with $V = [n]$ wlog. A Hamiltonian cycle exists if an $x \in N(1)$ exists so that a 1-x-path exists which contains all vertices.

We now use dynamic programming and define this dp table:

$$P_{S,x} := \begin{cases} 1, & \text{a path from } x \text{ to } 1 \text{ exists, which contains all vertices} \\ 0, & \text{otherwise} \end{cases},$$

where S is the vertex set

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

The dp table can be filled with the following recursion. The base case is $P_{\{1,x\},x} = 1 \Leftrightarrow \{1, x\} \in E$ for $1 \neq x \in V$.

$$P_{S,x} = \max\{P_{S\setminus\{x\},x'} | x' \in S \cap N(x), x' \neq 1\}$$

```
Hamiltonkreis (G=([n],E)):
  //Initialisierung
  for all x in [n], x != 1:
    P[1,x][x] = (1,x) in E ? 1 : 0

  // Rekursion
  for all s=3 to n:
    for all S subset [n] with 1 in S and |S|=s:
      for all x in S, x != 1:
        P(S,x) = max{ P[S\{x}][x'] | x' in (S union N(x)),x' != 1 }

  // Ausgabe
  if exists x in N(1) with P[[n]][x]=1:
    return G has Hamiltonian cycle
  else:
    return G does not have Hamiltonian cycle
```

Theorem: This algorithm is correct and requires storage $\mathcal{O}(n \cdot 2^n)$ and has runtime $\mathcal{O}(n^2 \cdot 2^n)$ with $n = |V|$.

Proof. Correctness shown above.

Storage comes from 2^{n-1} sets S and $n - 1$ elements at most for x.

Runtime: $\sum_{s=3}^{n} \sum_{S\subseteq[n],1\in S,|S|=s} \sum_{x\in S,x\neq 1} \mathcal{O}(n) = \sum_{s=3}^{n} \binom{n-1}{s-1}(s-1)\mathcal{O}(n) = \mathcal{O}(n^2 2^n)$ □

Version 2 We consider the same scenario but reduce the storage requirements further.

We choose $s \in V$ arbitrarily and define for all $S \subseteq [n], v \notin S$

$W_S :=$ {path of length n in G with start/end vertex s, which does not visit any vertex from S}

W_S can be efficiently computed. If A_S is the adjacency matrix of $G[V \setminus S]$, then $|W_S|$ is at position (s, s) in $(A_S)^n$. Computing this matrix power requires $\mathcal{O}(n^{2.81} \log n)$ with Strassen's algorithm to multiply matrixes. Because those matrix multiplications can be done in-place, we only require $\mathcal{O}(n^2)$ storage. Here we neglect that we can, in practice, not use floating point storage of elements and thus have higher storage requirements. But those requirements are still polynomial, so the advantage of this algorithm remains.

$W_\varnothing$ contains all walks of length n with start and end vertex s. Each Hamiltonian cycle is contained twice in that set (once for each direction). All unwanted walks are $\cup_{i=1}^n W_{\{i\}}$, being walks which do not contain all vertices. We get:

$$\begin{aligned}
&\frac{1}{2}(|W_\varnothing| - |\cup_{i=1}^n W_i|) \\
&= \frac{1}{2}(|W_\varnothing| + \sum_{l=1}^{n}(-1)^l \sum_{1\leq i_1<\ldots<i_l\leq n} |W_{i_1} \cap \ldots \cap W_{i_l}|) \\
&= \frac{1}{2}(|W_\varnothing| + \sum_{l=1}^{n}(-1)^l \sum_{1\leq i_1<\ldots<i_l\leq n} |W_{i_1,\ldots,i_l}|)
\end{aligned}$$

```
CountHamiltonianCylces(G=([n],E)):
  s := 1 //random in V
  Z := |W[empty]|
  for all S subset [n] with s not in S and S not empty:
    compute |W[S]]| //with adjacency matrix of G[V\S]
    Z := Z+(-1)^{|S|}*|W[S]|
  Z := Z/2
  return Z
```

Theorem: The above algorithm computes the number of Hamiltonian cycles in $G = (V, E)$ correctly, requires $\mathcal{O}(n^2)$ storage, and has runtime $\mathcal{O}(n^{2.81} \log n \cdot 2^n)$ with $n = |V|$.

5.3 NP

Theorem Karp: The problem whether $G = (V, E)$ contains a Hamiltonian cycle is NP complete.

Generally, we divide problems into two categories:

- P (polynomial)

  Those are efficiently decidable problems.

- NP (non-deterministic polynomial)

  Those are (one-sided) efficiently verifiable problems.

We do not know whether $P = NP$ or $P \neq NP$. But most scientists believe the latter is the case.

We say that some problem is NP complete if we could conclude that $P = NP$ when that problem is in P. NP-completeness is shown with implications: If said problem is P, then the solution can be mapped to other NP problems, which then also would have to be in P.

5.4 Traveling Salesman Problem

The problem is, given some complete graph on n vertices with weights/distances between any two vertices, to find the shortest Hamiltonian cycle.

$$H = \min_{\text{Hamiltonian cycle}} \sum_{e\in E(H)} l(e)$$

5.4.1 NP completeness

That this problem is NP-complete follows from that we know that finding some Hamiltonian cycle is NP complete and we can map this problem to the TSP problem. Specifically, we consider $l(e) \in \{1, 2\}$ for all $e \in \binom{[n]}{2}$. The TSP problem then indirectly answers the question whether there is a Hamiltonian cycle in the subgraph which has all the edges with weight 1. This solves the Hamiltonian cycle problem if we assign edge weight 1 if the edge exists is the graph we want to find a Hamiltonian cycle in. All other edges then have weight 2.

Theorem: If for some $\alpha > 1$, there exists an α-approximation algorithm of the TSP with runtime $\mathcal{O}(f(n))$, then also an algorithm exists, which decides in $\mathcal{O}(f(n))$ whether a graph on n vertices is Hamiltonian.

Proof. The proof for this follows from the above elaborated process. However we chose edge weights from $\{0, 1\}$ instead of form $\{1, 2\}$ as $\alpha \cdot 0 = 0$ independent from α. □

5.4.2 metric TSP

We now consider the metric TSP problem, which introduces additional conditions for allowed weights:

- $l(e) \geq 0, \forall e \in E$
- $l(x, z) \leq l(x, y) + l(y, z), \forall x, y, z \in [n]$.

We design an algorithm which finds a Hamiltonian cycle H such that $\cos(H) \leq 2 \cdot cost(OPT)$, where OPT is the cheapest Hamiltonian cycle. This is called a 2-approximation.

The algorithm works like this:

1. compute a MST T in G
2. create T' by doubling all edges in T
3. compute an Eulerian cycle E in T'
4. reduce E to a Hamiltonian cycle by skipping vertices, which have already been visited.

Step 2 creates a multigraph. Step 3 can be done by simply walking 'around' the MST as each edge may be visited twice. Formal reasoning for existence from even degrees. Step 4 can be done, because the edge for skipping exists in G (is complete). This alternative path can only be shorter or have the same length (metric condition).

We get this analysis with $cost(T) \leq cost(OPT - e) \leq cost(OPT)$:

$$cost(T') = 2 \cdot cost(T) \leq 2 \cdot cost(OPT)$$
$$cost(E) = cost(T') \leq 2 \cdot cost(OPT)$$
$$cost(H) \leq cost(E) \leq 2 \cdot cost(OPT)$$

Theorem: For the metric TSP problem, a 2-approximation algorithm with runtime $\mathcal{O}(|V|^2)$ exists.

5.4.3 general TSP approximation

One believes that a 2-approximation is also possible. However, so far only a 4-approximation has been achieved.

6 Matchings ("Matchings")

Definition matching ("Matching"): A set of edges $M \subseteq E$ is called matching ("Matching") in $G = (V, E)$ if not vertex is incident to more than one edge. Formally: $e \cap f = \varnothing, \forall e, f \in M, e \neq f$.

Definition maximal/maximum matchings ("inklusionsmaximale/kardinalitätsmaximale Matchings"):

- A matching M is maximal ("inklusionsmaximal"/"nicht erweiterbar") if no matching M' with $M \subseteq M'$ and $|M'| > |M|$ exists. I.e., if $M \cup \{e\}$ is not a matching for all $e \in E \backslash M$.
- A matching M is maximum ("(kardinalitäts-)maximal"/"größtmöglich") if no matching M' with $|M'| > |M|$ exists.

Theorem: If M_{inc} is maximal and M_{max} is maximum, then $|M_{inc}| \geq \frac{|M_{max}|}{2}$

Proof. Every edges from M_{max} must have at least on vertex in M_{inc} as the edge could otherwise be added to M_{inc}, which would then not be maximal.

$$|M_{max}| \leq |\text{endpoints in } M_{inc}| = 2 \cdot |M_{inc}|$$

□

Definition M-augmenting path ("M-augmentierender Pfad"): An M-augmenting path is a path which edges alternate between being in M and not being in M and which starts and ends in vertices not matched in M.

Assuming we know some M-augmenting path in M, we can increase the matching by switching edge membership in the matching along the augmenting path.

Theorem Berge: If M is a matching in $G = (V, E)$, which is not maximum, then an M-augmented path exists.

Proof. Let M_1 and M_2 be two arbitrary matchings in some graph G. We know that every vertex in $G_M = (V, M_1 \oplus M_2)$ ($\oplus$ being XOR) has degree at most 2. The connected components of G are hence paths and cycle paths (the latter must have an even length, as edges from M_1 and M_2 are alternating).

We now assume $|M_1| < |M_2|$ and consider the connected components of $M_1 \cup M_2$. The connected components are paths or cycle paths. This also considers edges from $M_1 \cap M_2$. Those are paths of length 1, which correspond to one element in M_1 and one in M_2. Each such path or path from $M_1 \oplus M_2$ with even length or cycle path comprises as many edges from M_1 as from M_2. Because of $|M_1| < |M_2|$, a path must exist in $M_1 \cup M_2$, which contains more edges from M_2 than from M_1. If $|M_2| = |M_1| + k$, even at least k

such paths must exist. With such an M_1-augmented path, we can get a larger matching $M_1' := M_1 \oplus P$ (P to be understood as a set of edges).

This shows that as long as some matching M is not maximum, i.e. a larger matching M_2 exists, an M-augmented path exists and can be used to get a larger matching. □

6.1 Maximal Matchings - Greedy Algorithm

```
GreedyMatching(G):
  M is empty
  WHILE E not empty:
    chose an arbitrary edge e in E
    M = M union {e}
    remove e and all incident edges from G
```

This provides a maximal matching by definition and runs in $\mathcal{O}(|E|)$.

6.2 Maximum Matchings

We find maximum matchings by finding augmenting paths. When we start with a matching comprising one arbitrary edge, we have to find at most $\frac{|V|}{2} - 1$ matchings consecutively to get a maximum matching. We will now deal with finding those augmenting paths in different kind of graphs.

6.2.1 Bipartite graphs

This algorithm finds one augmenting path in a bipartite graph using modified BFS.

```
Augmenting_Path(G=(A⊎B,E),M):
  L_0 := { vertices in $A$ not covered }
  mark all vertices from $L_0$ as visited
  IF L_0 = ∅:
    RETURN M is maximal
  FOR i = 1..n:
    IF i is odd:
      L_i := { neighbors of L_{i-1} via edges in E\M, which are not
          covered }
    ELSE:
      L_i := { neighbors of L_{i-1} via edges in M, which are not covered
          }
    mark all vertices from L_i as visited
    IF L_i contains vertices $v$ which are not covered:
      find path P from L_0 to $v$ by backtracking
      return P
  return M is already maximal
```

This modified BFS runs in $\mathcal{O}(|V|+|E|)$. Often we only consider $\mathcal{O}(|E|)$ as the problem can be reduced to only connected graphs. And we we must do this $\mathcal{O}(|V|)$ times as states before, we get $\mathcal{O}(|E| \cdot |V|)$.

An algorithm called Hopcroft-Karp algorithm exists, which reduces the runtime even further to $\mathcal{O}(\sqrt{|V|}|E|)$. This was not discussed besides mentioning of the runtime.

Furthermore, state-of-the-art algorithms reach runtimes of $\tilde{\mathcal{O}}(|E|)$, where the $\sim$ indicates almost-linear runtime.

6.2.2 General graphs

This case is very complex and we do not deal with it in detail. The complexity comes from the possibility of circle sof uneven length in the graph. The algorithm which extends the above idea to work with this condition is called Blossom algorithm and can find augmented paths in $\mathcal{O}(|V| \cdot |E|)$, leading to a total runtime of $\mathcal{O}(|V|^2 \cdot |E|)$.

State-of-the-art algorithms reach:

- $\mathcal{O}(|V|^{\frac{1}{2}} \cdot |E|)$ - Micali, Vazirani
- $\mathcal{O}(|V|^{2.373})$ - with amtrix multiplication, Mucha, Snakowski

6.2.3 weighted graphs

Theorem Gabow: If n is even and $l : \binom{[n]}{2} \to \mathbb{N}_0$ a weight function of the complete graph K_n, a perfect matching can be found in $\mathcal{O}(n^3)$.

Proof. The proof of this statement is beyond the scope of this course. □

Theorem Christofiedes: A $\frac{3}{2}$-approximation for the metric TSP problem exists with runtime $\mathcal{O}(|V|^3)$.

Proof. The approach is similar to the 2-approximation algorithm. The noticable difference is how the MST is modified so that an Eulerian cycle can be found.

Now, let S be the set of edges with an uneven degree in the MST T. Because $|S|$ is even, there is a perfect matching in $K_n[S]$. Such a minimal perfect matching can be found in $\mathcal{O}(n^3)$ according to Theorem 1.50. In $T \cup M$ we can then find an Eulerian cycle with $l(C) \leq l(M) + l(T)$.

We already know that $l(T) = mst(K_n, l) \leq opt(K_n, l)$. We show that $l(M) \leq \frac{1}{2} opt(K_n, l)$. Consider a Hamiltonian cycle C_{opt}. The vertices of S separate C_{opt} into $|S|$ paths. Those paths can be reduced to vertices without making the cycle path longer as the triangle inequality holds. We now have a cycle path with its vertices being S and $l(C_S) \leq l(C_{opt}) = opt(K_n, l)$. This cycle path can be written as the union of two matchings of which at least one must have length $\leq \frac{1}{2} l(C_s)$. □

6.3 (Perfect) Matchings in bipartite graphs

Finding matchings in bipartite graphs is of high interest in many applications. Notice that we could add a super source S and super end vertex T to some bipartite graph and then consider Menger's theorem. The size of the matching would correspond to the connectivity between S and T, i.e., the number of intern-vertex-disjunct paths from S to T.

Theorem Hall's Marriage Theorem ("Heiratssatz"/"Hall"): A bipartite graph $G = (A \uplus B, E)$ contains a matching M of cardinality $|M| = |A| \Leftrightarrow \forall X \subseteq A : |X| \leq |N(X)|$.

Proof.

First, we consider $\Rightarrow$: Let M be the matching with $|M| = |A|$. In the subgraph $H = (A \uplus B, M)$, every subset $X \subseteq A$ has exactly $|X|$ neighbors by definition (of the matching). Because $M \subseteq E$, we have $|N(X)| \geq |X|$ for all $X \subseteq A$.

Second, we consider $\Leftarrow$: We use induction over $a = |A|$.

- Base Case ($a = 1$): As we only have one vertex with at least one neighbor, this follows easily.
- Hypothesis: The statement holds for all $|A| < a$.
- Step:
  - Case 1: $\forall X \subsetneq A, X \neq \varnothing, |X| < |N(X)|$
    We chose an arbitrary edge $e = \{x, y\}$, add e to the matching, and consider $G' = G[V \backslash \{x, y\}]$ (and all incident edges to x, y). Because we had $<$ and only one vertex is removed from B, we can use the hypothesis to show that a matching exists for the remaining vertices from A, being $A \backslash \{x\}$.
  - Case 2: $\exists X_0 \subsetneq A, X_0 \neq \varnothing, |X_0| = |N(X_0)|$
    We consider the vertex-disjunct graphs $G' = G[X_0 \uplus N(X_0)]$ and $G'' = G[A \backslash X_0 \uplus B \backslash N(X_0)]$. For G' the hypothesis shows our statement. It also holds for G'' as we will show now: We have some arbitrary $X \subseteq A \backslash X_0$ and know

$$\begin{aligned} |X| + |X_0| = |X \cup X_0| &\overset{\text{from assumption}}{\leq} |N(X \cup X_0)| \\ = |N(X_0)| + |N(X) \backslash N(X_0)| &= |N(X_0)| + |N(X) \cap (B \backslash N(X_0))| \\ \Rightarrow |X_0| \leq |N(X) \cap (B \backslash N(X_0))|&, \text{ as } |X| = |N(X)| \end{aligned}$$

□

We use the term k-regular. It means that all degrees in the given graph are k.

Theorem: Let $G = (A \uplus B, E)$ be a k-regular bipartite graph. Then, there are $M_1, ..., M_k$ so that $E = M_1 \uplus ... \uplus M_k$ and all $M_i, 1 \leq i \leq k$, are perfect matchings in G.

Proof. It follows directly from 1.52 that k-regular bipartite graphs (all degrees are k) do always contain a perfect matching. Some vertices from A must connect to at least the same amount of vertices in B, because of their degrees. And the matching is perfect, because $|A| = |B|$ as the matching is perfect.

If we remove that matching from the graph, we get a $k - 1$-regular graph. Again, this must have a perfect matching my the above reasoning. Hence, we can find k perfect matchings. □

Now, we consider how to fund such perfect matchings in the case that k is a power of 2.

Theorem Gabow: If $G = (V, E)$ is 2^k-regular, a perfect matching can be found in $\mathcal{O}(|E|)$.

Proof. As G is 2^k-regular, each connected component satisfies the Euler condition and we can find an Eulerian Cycle in each component in $\mathcal{O}(|E|)$. Then, we remove every second edge from those Eulerian Cycle, which results in a 2^{k-1}-regular graph. This procedure can be repeated until we have some 2^0-regular graph, which is a perfect matching then.

Runtime: We can find in $\mathcal{O}(|E|)$ an Eulerian cycle in each connected component. We have k iterations:

$$\sum_{i=1}^{k} \mathcal{O}(|E|)/2^{i-1} \leq \sum_{i \geq 0} \mathcal{O}(|E|)/2^i = 2\mathcal{O}(|E|) \leq \mathcal{O}(|E|)$$

□

This is the generalized form, which we do not prove or elaborate on.

Theorem: In k-regular bipartite graphs, a perfect matching can be found in $\mathcal{O}(|V|)$.

7 coloring

This tries to solve problems in which we have to find sets of vertices, which have no internal edges.

Definition vertex coloring ("(Knoten-)Färbung"):
A (vertex) coloring of some graph $G = (V, E)$ with k colors is a map $c : V \to [k]$ so that $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$.

Definition chromatic number ("chromatische Zahl"):
The chromatic number ("chromatische Zahl") $X(G)$ is the minimal number of colors required for a vertex coloring of G.

Graphs with the chromatic number g are called k-partite with $k \geq g$. So an equivalent phrasing of the definitions is $X(G) \leq k \Leftrightarrow G$ is k-partite, i.e., there are k edges within which there are no edges. Such sets are called independent sets ("stabile Mengen").

Theorem: Some graph $G = (V, E)$ is bipartite if and only if it contains no cycle with odd length as a subgraph.

Proof.
First, $\Rightarrow$: Cycle paths of uneven length have the chromatic number three. Hence, they can not be contained in a bipartite graph.
Second, $\Leftarrow$: Without loss of generality we can assume that G is connected. We then start BFS at some arbitrary vertex s and color it with 1 if its distance $d[v]$ to s is even or with 2 if it is odd. As there is no cycle path of uneven length, no vertex can be assigned two different colors. □

Theorem Four Color Theorem ("Vier-Farbe Theorem"): Every land map can be colored with four colors.

Proof. This is very complex. It consists of a theoretical part, which reduces the problem to a finite number of finite cases. In a second part, those cases are then checked my a computer program.
Mathematicians sometimes mock this to not be a 'real' proof as it depends on program correctness etc. □

The question rises, one one can show whether G is k-partite. We can use DFS/BFS to easily show whether G is bipartite in $\mathcal{O}(|E|)$ as already stated in an above theorem. However:

Theorem: For all $k \geq 3$, the problem whether for some graph $G = (V, E)$ $X(G) \leq k$ is NP complete.

Proof. Given without proof. □

Alternatively, there are some general ideas on how to approach such a problem:

- exponential algorithm

  Yes, such an algorithm using the inclusion/exclusion principle exists and has runtime $\mathcal{O}(2.2^n)$.

- approximation

  No, for each $\epsilon > 0$ finding an $n^{1-\epsilon}$ approximation has NP difficulty.

- special cases

  Yes, special cases exist. We will consider some of them.

Theorem: $\forall k \in \mathbb{N}, \forall r \in \mathbb{N}$: Graphs without cycles of length $\leq k$ exist that do have chromatic number $\geq r$.

This is to give some intuition why finding 'global' colorings is difficult.

Proof. ... □

7.1 Greedy algorithm

```
Greedy-Coloring(G):
  choose an arbitrary order of the vertices: V={$v_1$,...,$v_n$}
  c[$v_1$] = 1
  FOR i=2..n:
    c[$v_i$] = min{ $k \in \mathbb{N} | k \neq c(u)$ for all $u \in N(v_i) \cap${$v_1$,...,$v_{i-1}$} }
```

Theorem: Let G be some connected graph. For the number of colors $C(G)$ required by the greedy algorithm the following holds: $X(G) \leq C(G) \leq \Delta(G) + 1$ ($\Delta(G)$ being the maximum degree in G).
If the graph is stored as an adjacency list, the coloring is found in $\mathcal{O}(|E|)$

Proof. Quite trivial. □

It is interesting to observe that some order V for the greedy algorithm always exists so that it only requires $X(G)$ colors. Specifically, coloring one partition at a time. But that such a best order is not always chosen can be seen when considering that bipartite graphs exist so that the greedy algorithm requires $\frac{|V|}{2}$ colors (despite being 2-colorable).

If the order has the property $|N(v_i) \cap \{v_1, ..., v_{i-1}\}| \leq k, \forall 2 \leq i \leq n$, then the greedy algorithm requires at most $k + 1$ colors. From that follows the heuristics for a good order that v_n is the vertex with the smallest degree, v_{n-1} with the 2nd smallest degree, ...

This is a formal treatment of that heuristic:

Theorem: Let $G = (V, E)$ be some graph and $k \in \mathbb{N}$ so that each induced subgraph of G contains a degree of at most k. Then, $X(G) \leq k + 1$ and a $k + 1$-coloring can be found in $\mathcal{O}(|E|)$.

Proof. We create an order of the vertices of G $v_1, ..., v_n$ like this:

- We choose some vertex in G with degree at most k, which we name v_n, and remove it from the graph.

- Again, we choose some vertex in G' with degree at most k, which we name v_{n-1}, and remove it from the graph.

- Repeat...

So, for all $2 \leq i \leq n$: v_i has the maximum degree k in the induced subgraph $\{v_1, ..., v_i\}$. If we color G in that order, we hence know that v_i has at most k already colored neighbors. Hence, we can color it with $k+1$ colors. This holds for all vertices and we hence can color G with $k+1$ colors.

Actually coloring the graph once a proper order is known can be clearly done in $\mathcal{O}(|E|)$ (or rather $\mathcal{O}(|E|+|V|)$). Hence, we consider the runtime of finding the order.

- Consider the array $d[]$, which stores for each vertex (and time $1 \leq i \leq n$) the degree of vertex v in $G[V \backslash \{v_{i+1,...,v_n}\}]$. We consider n, $n-1$, ... In the beginning, $d[v] := deg(v)$. For each newly found vertex, we can update the array in $\mathcal{O}(deg(v_i))$. For all vertices, this is $\mathcal{O}(|E|)$. Now, we still have to find some vertex with degree at most k in the subgraph $G[V \backslash \{v_{i+1,...,v_n}\}]$.

- We maintain the boolean array *removed*[] to member, which vertices have already been removed, being initialized with false. Still, finding some vertex with $d[v] \leq k$ and $removed[v] = false$ would still take $\mathcal{O}(|V|)$, leading to a total runtime of $\mathcal{O}(|V|^2)$.

- To optimize, we use the set Q which contains all not removed vertices with degree at most k. We add element to Q after initialization and whenever vertices are removed/$d[]$ is updated. We also set *removed* to true when adding some vertex to Q to not add it a 2nd time later. Hence, one step can then be done in $\mathcal{O}(1)$.

The total runtime then is $\mathcal{O}(|V| \cdot 1 + (|V| + |E|))$, which can be approximated with $\mathcal{O}(|E|)$. □

Corollary: If $G = (V, E)$ is connected and $\exists v \in V$ with $deg(v) < \Delta(G)$, then we can efficiently find an order so that the greedy algorithm requires at most $\Delta(G)$ colors.

A special case, in which coloring becomes easier is if we can create some block graph. If G is a graph and we can color each block with k colors respective, also the entire graph can be colored with k colors. The approach for one cut vertex is to color both blocks (each including the cut vertex) respectively. Then there are two cases:

- The cut vertex has the same color.

- The cut vertex has different colors: switch the naming of the colors in one block so that the color matches.

Notice that in trees each edge is a block by itself by definition. Hence, it follows that trees can always be colored with two colors/are bipartit.

Theorem: In a planar graph, a vertex with degree ≤ 5 always exists.

This was given without proof. It follows directly that we can efficiently find a coloring with ≤ 6 colors for planar graphs.

7.2 Brook's Theorem

Above we learned how to find colorings with at most $\Delta(G)+1$ colors. Here, we will show how to find colorings with at most $\Delta(G)$ colors.

Theorem Brook: Let $G = (V, E)$ be a connected graph, which is neither complete ($G = K_n$) nor a cycle of uneven length ($G \neq C_{2n+1}$). Then $X(G) \leq \Delta(G)$ and there exists an algorithm, which colors the vertices of G with $\Delta(G)$ in $\mathcal{O}(|E|)$.

Proof. If G has a vertex with degree less than $\Delta(G)$ we can use the already known Greedy algorithm to achieve this. Similarly, we can consider the special case of blocks when there is a cut vertex. Hence, we assume that all vertices have degree $\Delta(G)$ and that there is no cut vertex.

Because G is connected but not complete, $\exists v$ with $v_1, v_2 \in N(v)$ and $v_1 \neq v_2$ and $\{v_1, v_2\} \notin E$.

- Case 1: $G \backslash \{v_1, v_2\}$ is connected.

  Then we do dfs/bfs in $G[V \backslash \{v_1, v_2\}]$ from v. We order the vertices like this: First v_1 and v_2, then the elements from dfs / bfs in reverse order so that v is the last vertex. The greedy algorithm then clearly works with $\Delta(G)$ colors at most.

- Case 2: $G \backslash \{v_1, v_2\}$ is not connected.

  Let $V_1, ..., V_s$ be the connected components of $G[V \backslash \{v_1, v_2\}]$ with $s \geq 2$. $G_i := G[V_i \cup \{v_1, v_2\}], 1 \leq i \leq s$.

  - Case 2.1: If in all G_i, v_1 or v_2 has at most degree $\Delta(G) - 2$
    Then, we can add the edge $\{v_1, v_2\}$ and color the graph with $\Delta(G)$ colors with v_1 and v_2 having different colors. The remaining part is analogous to the block graph approach.
  - Case 2.2
    The degrees of v_1 and v_2 across all G_i can sum up to at most $\Delta(G)$ respectively. Hence, in all but one G_i, the degree can be at most 1. If the degree would be 0 in some G_i, then the other vertex would be a cut vertex, which has been already excluded as a special case above. Hence, we must have $s = 2$ and v_1 and v_2 have exactly one respective neighbor u_1 and u_2 in G_2. We then color G_1 and G_2 with $\Delta(G)$ colors respectively.
    * If $\Delta(G) > 2$:
      It might require some color changing so that u_1 and u_2 do have different colors than v_1 and v_2.
      Together, then the colorings of G_1 and G_2 are a $\Delta(G)$ coloring of G.
    * If $\Delta(G) = 2$:
      Then, G must be a circle. And because we excluded odd circle length, G must have even length and can be colored with two colors.

□

7.3 3-colorable graphs

Theorem: A 3-colorable graph can be colored in $\mathcal{O}(|V| + |E|)$ with $\mathcal{O}(\sqrt{|V|})$ colors.
Important: The knowledge of the graph being 3-colorable is necessary in-advance!

Proof. We then chose some vertex v and can color $N(v)$ with two colors (none can have color of v and three colors at most). If we choose multiple v, we can color their respective neighborhood with three new colors every time.

The algorithm now consists of two partsts. First:

```
while exists uncolored v with 'large' degree (>= x): // x specified
    later
  color v and uncolored elements of N(v) with three new colors
```

The while loop can execute $\frac{|V|}{x+1}$ times at most, because each iteration colors colors $x+1$ vertices.

After all while iterations are complete, we have a graph with maximum degree $\leq x-1$. The greedy algorithm can color that graph with x colors efficiently.

For an upper bound of the number of colors we get: $3\frac{|V|}{x+1}+x \leq \mathcal{O}(4\sqrt{|V|}) \leq \mathcal{O}(\sqrt{|V|})$ (we choose $x = \sqrt{|V|}$ here).

The runtime is quite obvious. □

The state-of-the-art algorithm requires $\mathcal{O}(|V|^{0.211})$, which is only about two roote better. We do not cover this algorithm due to its high complexity.

7.4 additional stuff

Theorem: For all $k \geq 2$, some graph G_k without triangles with $X(G_k) \geq k$ exists.

Proof. We prove this by induction over k.

- Base Case ($k = 2$)

  trivial, such as cycle with odd length ≥ 5.

- Hypothesis

  Let $k \geq 2$ and G_k be some graph without triangles with $x(G_k) \geq k$.

- Step

  We add some vertices and edges to G_k. Let $v_1, v_2, ..., v_n$ be the vertices of G_k. For each v_i we add w_i to G and connect it to all neighbors of v_i. Finally, we add vertex z and connect it to all $w_1, ..., w_n$. Notice that the new graph does not contain triangles.

  - Because w_i are not connected among themselves, z is not in any triangle.
  - Because w_i is incident to the neighbors of v_i, it can not be in any triangle as G_k has not triangles by assumption.

  Hence, G_{k+1} has no triangles.

  Assume G_{k+1} would be colorable with k colors. At least one of the k colors could not be amont $w_1, ..., w_n$, because of z. As all pairs v_i and w_i have the same neighbors, they can share the same color. Hence, G_k would have to be colorable with $k-1$ colors, which contradicts our assumption.

□

Part II

Probability & Randomized Algorithms

8 Fundamentals

Definition probability space: A discrete probability space ("Wahrscheinlichkeitsraum") is defined by its sample space ("Ereignismenge") $\Omega = \{\omega_1, \omega_2, ...\}$ of elements ("Elementarereignisse"). Each element is assigned some elementary probability ("(Elementar-)Wahrscheinlichkeit") $\Pr[\omega_i]$ with $0 \leq \Pr[\omega_i] \leq 1$ and

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

Some set $E \subseteq \Omega$ is called outcome ("Ereignis"). The probability of an outcome is defined by:

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

If E is an outcome, $\overline{E} := \Omega \backslash E$ is called the complementary outcome of E ("Gegenereignis").

We limit ourselves to finite/countably infinite probability spaces, whose study is often denoted as (elementary) stochastic ("(elementare) Stochastik").

Lemma: For outcomes A and B we have:

- $\Pr[\varnothing] = 0$ and $\Pr[\Omega] = 1$
- $0 \leq \Pr[A] \leq 1$
- $\Pr[\overline{A}] = 1 - \Pr[A]$
- If $A \subseteq B$, then $\Pr[A] \leq \Pr[B]$

Theorem ("Additionssatz"): If outcomes $A_1, ..., A_n$ are pairwise disjunct:

$$\Pr[\cup_{i=1}^{n} A_i] = \sum_{i=1}^{n} \Pr[A_i]$$

For an infinite set of pairwise disjunct outcomes: $\Pr[\cup_{i \geq 1} A_i] = \sum_{i \geq 1} \Pr[A_i]$

Theorem principle of inclusion/exclusion ("Inklusions-/Exklusionsprinzip"): For outcomes $A_1, ..., A_n$ $(n \geq 2)$:

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{l=1}^{n} (-1)^{l+1} \sum_{1 \leq i_1 \leq ... \leq i_l \leq n} \Pr\left[A_{i_1} \cap ... \cap A_{i_l}\right]$$

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

This theorem has already been stated before in regard to cardinality instead of probability. However that does not make any difference. The proof will be considered later.

Computing this for large n is very tedious. Hence, this approximation:

Theorem Boole's Inequality: for outcomes $A_1, ..., A_n$, we have

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

Analogously for countably infinite A_i.

Proof. We consider the finite case from which the infinite case can be derived.

For $i \geq 1$: $B_i := A_i \backslash (A_1 \cup ... \cup A_{i-1})$. Obviously $\Pr[B_i] \leq \Pr[A_i]$. And if $i \neq j$, B_i and B_j are disjunct. Still, $\cup_{i=1}^{n} A_i = \cup_{i=1}^{n} B_i$.

We use the "Additionssatz":

$$\Pr[\cup_{i=1}^{n} A_i] = \Pr[\cup_{i=1}^{n} B_i] = \sum_{i=1}^{n} \Pr[B_i] \leq \sum_{i=1}^{n} \Pr[A_i]$$

□

Definition La-Place Space: A La-Place space is a finite probability space in which all (elementary) events have the same probability $\frac{1}{|\Omega|}$. From that follows that $\Pr[E] = \frac{|E|}{|Q|}$.

For La-Place spaces we say that the result of the (probability) experiment ("Zufallsexperiment") is uniformly distributed ("uniform verteilt"/"gleichverteilt"). This probability space corresponds to the space with maximum entropy and, hence, is a good starting point when having no information on a system one tries to model.

9 Conditional Probability

We provide a generally helpful table here, which provides formulas to compute the number of variations in which one can choose k elements from some set with n elements.

$\underline{k}$ corresponds to the continuously decreasing sequence starting at k. By definition $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. The two formulas for the ordered cases are quite obvious. For the unordered cases consider this:

- without putting back

  There are two intuitive understandings for this. First, see that $\frac{n^{\underline{k}}}{k!}$ divides the number of ordered choices by the number of orders, i.e., how often there is a duplicate of each case when not considering the order. This corresponds to $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

  The second option is to explain the meaning of the factors of $\frac{n!}{k!(n-k)!}$ directly. $n!$ considers all orders of our n elements. We say, from each order we choose the first k elements. Then we divide by $k!$ because we do not care about the order of the elements we choose. And we divide by $(n-k)!$ because we do not care about the order of the elements we do not choose.

- putting back

  There is some intuition for this too. We construct this scenario: Write n+k-1 fields next to each other. Then, write $n-1$ strokes in arbitrary of those fields. Now give the n numbers an order and fill empty fields between the i-th and $i+1$-th stroke with the $i+1$-th number. This results in all possible combinations. Instead of putting strokes, we can also choose the empty k fields, which corresponds to computing $\binom{n+k-1}{k}$ for the number of combinations.

	ordered	unordered
putting back	n^k	$\binom{n+k-1}{k}$
without putting back	$n^{\underline{k}}$	$\binom{n}{k}$

Figure 2

We now consider conditional probability. $\Pr[A|B]$ denotes the probability that A holds under the assumption that B holds.

Definition conditional probability: A, B are outcomes with $\Pr[B] > 0$. The conditional probability $\Pr[A|B]$ (the probability of A given B) is defined as:

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

The conditional probabilities $\Pr[\cdot|B]$ for some $0 < B \subseteq \Omega$ is a probability space itself.

Theorem Law of total probability: $A_1, ..., A_n$ are pairwise disjunct outcomes and $B \subseteq A_1 \cup ... \cup A_n$.

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

Analogously for the countably finite case.

Proof. Notice: $B = (B \cap A_1) \cup ... \cup (B \cap A_n)$. For $i \neq j$: As $A_i \cap A_j \neq \varnothing$, also $B \cap A_i$ and $B \cap A_j$ are disjunct. Hence, we can use the addition theorem to write:

$$Pr[B] = Pr[B \cap A_1] + ... + Pr[B \cap A_n]$$

Then, we use the definition of conditional probability ($Pr[B \cap A_i] = Pr[B|A_i] \cdot Pr[A_i]$)

$$= Pr[B|A_1] \cdot Pr[A_1] + ... + Pr[B|A_n] \cdot Pr[A_n]$$

The countably infinite case holds, because the addition theorem and conditional probability also hold for the countably infinite case. □

Theorem multiplication rule ("Multiplikationssatz"): $A_1, ..., A_n$ are arbitrary outcomes with $\Pr[A_1 \cap ... \cap A_n] > 0$.

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

Proof. First, all probabilities are well defined: $r[A_1] \geq Pr[A_1 \cap A_2] \geq ... \geq Pr[A_1 \cap ... \cap A_n] > 0$.

Second, the following (just having applied the formula to the above claim) easily simplifies to the desired form:

$$\frac{Pr[A_1]}{1} \cdot \frac{Pr[A_1 \cap A_2]}{Pr[A_1]} \cdot ... \cdot \frac{Pr[A_1 \cap ... \cap A_n]}{Pr[A_1 \cap ... \cap A_{n-1}]}$$

□

Theorem Bayes' theorem ("Satz von Bayes"): $A_1, ..., A_n$ are pairwise disjunct outcomes. $B \subseteq A_1 \cup ... \cup A_n$ is an outcome with $\Pr[B] > 0$. For any $i = 1, ..., n$:

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

The case for the countably finite case is analogously.

Proof. This follows directly from the definition of conditional probability and the law of total probability. □

9.1 Common Problems

Monty-Hall-Problem The Monty-Hall-Problem is a logic/stochastic problem, which demonstrates counterintuition towards those basic laws of probability.

Medicine testing Statistical analysis can result in

- Pr["test positive"|"patient is sick"]
- Pr["test positive"|"patient is not sick"]

But wer are interested in Pr["patient is sick"|"test is positive"]. This can be computed with Bayes' Theorem.

Birthday problem The birthday problem is a classic probability puzzle that asks, "What is the probability that, in a group of n randomly chosen people, at least two of them share the same birthday?" The answer may be surprising to many people: even in a group as small as 23 individuals, the probability of a shared birthday is greater than 50 %. This is because there are 365 possible birthdays, and as the number of people in the group increases, the number of possible pairings that could result in a shared birthday increases exponentially.

We can rephrase the problem to throwing m balls into n bins ($n = 365$ when considering birthdays and m the number of people). To simplify the computation, we consider the complementary event. We consider: $A_i :=$ "the i-th ball lands in a so far unoccupied bin".

For the conditional probability assuming that all so far landed in their own bin: $Pr[A_i|A_1 \cap ... \cap A_{i-1}] = \frac{n-(i-1)}{n}$.

So that all m balls/people land in/on different bins/dates, we have $Pr[A_1 \cap ... \cap A_m] = 1 \cdot \frac{n-1}{n} \cdot ... \cdot \frac{n-(m-1)}{n}$ by using the multiplication rule. For the initially interesting probability, we then have $1 - Pr[A_1 \cap ... \cap A_m]$.

Hashing The Birthday Problem is a toy problem. However, its concept of distributing m data sets among n storage locations is more common.

This introduces hashing as real-world application: Hashing. We have a universe $\mathcal{K} \subseteq \mathbb{N}_0$ and $\mathcal{K} = [p]$, because we model the numbers as bits on computers without loss of generality. p must be a prime here and we then define

$$h_{a,b} := \mathcal{K} \to [n]$$

with

$$k \mapsto ((ak + b) \mod p) \mod n$$

From the fact that p is prime, it follows that for each $k' \in \mathcal{K}$, there is a unique $k \in \mathcal{K}$ so that $k' = (ak + b) \mod p$.

For all pairs $a, b \in \mathcal{K}$ with $a \neq 0$, the function has the characteristic that $|h_{a,b}^{-1}| \leq \lceil p/n \rceil$. In other words, the function $h_{a,b}$ distributes the elements of $\mathcal{K}$ uniformly to the memory $[n]$.

10 Independence

Colloquially, independence means that knowledge about one outcome does not imply knowledge about some other outcome. We define independence, but intuition comes from $\Pr[A|B] = \Pr[A] \Leftrightarrow \frac{\Pr[A \cap B]}{\Pr[B]} = \Pr[A]$.

Definition independence: Outcomes A, B are independent if $\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$.

It is important to realize that physical and stochastic independence are not the same. Two things can be physically linked but we still consider them independent in regards to their probability. For instance, we may throw a coin and consider whether the result is even/odd and $\leq 3 / \geq 4$.

Definition independence: $A_1, ..., A_n$ are independent if $\forall I \subseteq \{1, ..., n\}$:

$$\Pr[A_{i_1} \cap ... \cap A_{i_k}] = \Pr[A_{i_1}] \cdots \Pr[A_{i_k}]$$

A countably infinite set is said to be independent if the above holds for all $I \subseteq \mathbb{N}$.

Lemma: The outcomes $A_1, ..., A_n$ are independent if and only if for all $(s_1, ..., s_n) \in \{0, 1\}^n$:

$$\Pr[A_1^{s_1} \cap ... \cap A_n^{s_n}] = \Pr[A_1^{s-1}] \cdots \Pr[A_n^{s_n}]$$

we have $A_i^0 = \overline{A}_i$ and $A_i^1 = A_i$.

Proof.

First, we show that the definition implies this lemma.

We do induction over the count of zeros in $s_1, ..., s_n$. Base Case: If $s_1 = ... = s_n = 1$, then we must show nothing.

Step: From Theorem 2.3 (addition theorem), we get

$$\Pr[\overline{A}_1 \cap A_2^{s_2} \cap ... \cap A_n^{s_n}] = \Pr[A_2^{s_2} \cap ... \cap A_n^{s_n}] - \Pr[A_1 \cap A_2^{s_2} \cap ... \cap A_n^{s_n}]$$

When using the induction hypothesis, we get

$$\Pr[\overline{A}_1 \cap A_2^{s_2} \cap ... \cap A_n^{s_n}] = \Pr[A_2^{s_2}] ... \Pr[A_n^{s_n}] - \Pr[A_1] \Pr[A_2^{s_2}] ... \Pr[A_n^{s_n}] = (1 - \Pr[A_1]) \Pr[A_2^{s_2}] ... \Pr[A_n^{s_n}]$$

From this, the claim follows.

Second, we give the idea of how this lemma implies the definition in one special case. The general case follows but is more complicated to write down.

$$\begin{aligned} \Pr[A_1 \cap A_2] &= \sum_{s_3, ..., s_n \in \{0,1\}} \Pr[A_1 \cap A_2 \cap A_3^{s_3} \cap ... \cap A_n^{s_n}] \\ &= \sum_{s_3, ..., s_n \in \{0,1\}} \Pr[A_1] \Pr[A_2] \Pr[A_3^{s_3}] ... \Pr[A_n^{s_n}] \\ &= \Pr[A_1] \Pr[A_2] \sum_{s_3 \in \{0,1\}} \Pr[A_3]^{s_3} ... \sum_{s_n \in \{0,1\}} \Pr[A_n^{s_n}] \\ &= \Pr[A_1] \Pr[A_2] \end{aligned}$$

□

Lemma: A, B, C are independent outcomes. Then, $A \cap B$ and C as well as $A \cup B$ and C are independent.

Proof. For $\cap$:

$$\Pr[(A \cap B) \cap C] = \Pr[A] \Pr[B] \Pr[C] = \Pr[A \cap B] \Pr[C]$$

For $\cup$, we use the inclusion/exclusion principle:

$$\begin{gathered} \Pr[(A \cup B) \cap C] \\ = \Pr[(A \cap C) \cup (B \cap C)] \\ = \Pr[A \cap C] + \Pr[B \cap C] - \Pr[A \cap B \cap C] \\ = \Pr[C](\Pr[A] + \Pr[B] - \Pr[A \cap B]) \\ = \Pr[C] \Pr[A \cup B] \end{gathered}$$

□

10.1 pseudorandom number generators (”Pseudozufallsgeneratoren”)

When working with deterministic computers, we can not get true randomness. However, we require it for randomized algorithms to work. So we use PRNGs:

$$f : \{0,1\}^m \to \{0,1\}^m \times \{0,1\}^n$$
$$s_{i-1} \to (s_i, a_i)$$

s_i are internal states and a_i are ’random’ numbers. m is the state length. Starting with the initial ’seed’ $s_0 \in \{0,1\}^m$, we continuously can generate numbers.

This becomes useful, if a_i are independent (and uniformly distributed) according to our above definition of independent. Then, the generated numbers appear to be random to the user/program.

To this date it is unknown whether a function such that different a_i are actually independent exists. But we have sufficient functions for current applications.

11 Random Variables

Definition random variable (”Zufallsvariable”): A random variable (”Zufallsvariable”) is a function $X : \Omega \to \mathbb{R}$.

For some discrete probability space, the image of a random variable W_X (with random variable X) is always finite or countable.

A convention for writing outcomes is to write $X \leq 5$ instead of $\{\omega \in \Omega | X(\omega) \leq 5\}$. This kind of notation is generally accepted/correct here, but not in the sense of ’pure’ mathematics.

One useful application is to define an indicator variable for some event.

Definition: $A \subseteq \Omega$ is some outcome. The indicator variable X_A is defined with $X_A(\omega) := 1 \Leftrightarrow \omega \in A$ and $X_A(\omega) := 0 \Leftrightarrow \omega \notin A$
For the expected value of X_A: $\mathbb{E}[X_A] = \Pr[A]$.

Definition density function (”Dichtefunktion”): For some random variable X:

$$f_X : \mathbb{R} \to [0,1]$$
$$x \mapsto \Pr[X = x]$$

Definition distribution function (”Verteilungsfunktion”): random variable X:

$$F_X : \mathbb{R} \to [0,1]$$
$$x \mapsto \Pr[X \leq x]$$

The density/distribution function uniquely describe a random variable.

11.1 Expected Value

Definition expected value (”Erwartungswert”): For a random variable we define the expected value $\mathbb{E}[X]$ with

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

With infinite X_x this is an infinite sum, which might converge. Then, the expected value is not defined. We only consider probability distributions for which the expected value is properly defined.

Lemma: X is a random variable.

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

Proof. This follows by using the definition of $\Pr[X = x]$. Consider:

$$\mathbb{E}\left[(\right] X) = \sum_{i \in W_X} i \cdot \Pr[X = i]$$

$$= \sum_{\omega \in \Omega, i \in W_X} i \cdot \Pr[X = i \cap \omega] \text{ (law of total probability)}$$

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

□

Theorem: X is a random variable with $W_X \subseteq \mathbb{N}_0$.

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

Proof.

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

$$= \sum_{j=1}^{\infty} j \cdot \Pr[X = j] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] = \mathbb{E}[X].$$

□

11.1.1 Conditional Random Variables

This is how we treat conditional probabiliy/conditional random variables:

$$\Pr[(X|A) \leq x] := \Pr[X \leq x|A] = \frac{\Pr[\{\omega \in A | X(\omega) \leq x\}]}{\Pr[A]}, \Pr[A] > 0$$

The conditional random variable $X|A$ is just as X but with the domain limited to A. We write $X|A : A \to \mathbb{R}$. From that we derive this from the above definitions:

- $f_{X|A} : \mathbb{R} \to [0, 1], x \mapsto \Pr[X = x|A]$
- $F_{X|A} : \mathbb{R} \to [0, 1], x \mapsto \Pr[X \leq x|A]$
- $\mathbb{E}[X|A] := \sum_{x \in W_X} x \Pr[X = x|A] = \frac{1}{\Pr[A]} \sum_{\omega \in A} X(\omega) \Pr[\omega]$

Theorem: X some random variable. $A_1, ..., A_n$ are pairwise disjunct with $A_1 \cup ... \cup A_n = \Omega$ and $\Pr[A_1], ..., \Pr[A_n] > 0$.

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

The infinite case is analogously.

Proof.

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

□

X is independent of A if $f_{X|A} = f_X$.

11.1.2 Linearity of the Expected Value

From random variables $X_1, ..., X_n : \Omega \to \mathbb{R}$ we get random variable

$$f : \mathbb{R}^n \to \mathbb{R}$$
$$f(X_1, ..., X_n) : \Omega \to \mathbb{R}$$

If f is an affine linear function $(x_1, ..., x_n) \mapsto a_1x_1 + .. + a_nx_n + b$ $(a_1, ..., a_n, b \in \mathbb{R})$, we also write $X := a_1X_1 + ... + a_nX_n + b$.

Theorem: $X_1, ..., X_n$ are random variables and $X := a_1X_1 + ... + a_nX_n + b$ with $a_1, ..., a_n \in \mathbb{R}$.

$$\mathbb{E}[X] = a_1\mathbb{E}[X_1] + ... + a_n\mathbb{E}[X_n] + b$$

Proof.

$$\mathbb{E}[X] = \sum_{\omega \in \Omega} (a_1X_1(\omega) + ... + a_nX_n(\omega) + b) \Pr[\omega]$$
$$= a_1(\sum_{\omega \in \Omega} X_1(\omega) \Pr[\omega]) + ... + a_n(\sum_{\omega \in \Omega} X_n(\omega) \Pr[\omega]) + b$$
$$= a_1\mathbb{E}[X_1] + ... + a_n\mathbb{E}[X_n] + b$$

□

11.1.3 Proof of the Inclusion/Exclusion Principle

Let $A_1, ..., A_n$ be outcomes and $B := A_1 \cup ... \cup A_n$. We want to compute $\Pr[B]$.

We define the indicator variable $I_i := I_{A_i}$ and $I_{\overline{B}}$ for $\overline{B} := \Omega \backslash B$. We know $\mathbb{E}\left[I_{\overline{B}}\right] = \Pr[\overline{B}] = 1 - \Pr[B]$. Also, $I_{\overline{B}} = \prod_{i=1}^{n}(1 - I_i)$. From the latter we get

$$I_{\overline{B}} = 1 - \sum_{1 \leq i \leq n} I_i + \sum_{1 \leq i_1 < i_2 \leq n} I_{i_1} I_{i_2} - +... + (-1)^n I_1 \cdot ... \cdot I_n$$

With linearity of the expected value follows

$$\mathbb{E}\left[I_{\overline{B}}\right] = 1 - \sum_{1 \leq i \leq n} \mathbb{E}\left[I_i\right] + \sum_{1 \leq i_1 < i_2 \leq n} \mathbb{E}\left[I_{i_1} I_{i_2}\right] - +... + (-1)^n \mathbb{E}\left[I_1 \cdot ... \cdot I_n\right]$$

When recognizing that the product of indicator variables is an indicator variable itself, which stands for the cut of corresponding events we see $\mathbb{E}\left[I_{i_1} \cdot ... \cdot I_{i_k}\right] = \Pr[A_{i_1} \cap ... \cap A_{i_k}]$ for all $1 \leq i_1 < ... < i_k \leq n$.

11.2 Variance

The expected value alone does not always sufficiently describes some scenario. We generally want $\Pr[|X - \mathbb{E}[X]| = \text{"big"}] = \text{"small"}$. But often that is not given. Instead of using the absolute difference to introduce a measure for this property we use the difference square. That has two benefits: It is differentiable and larger deviations have a higher weight, which conforms to our intuition.

Definition variance ("Varianz"): X random variable with $\mu = \mathbb{E}[X]$. The variance ("Varianz") $\mathrm{Var}[X]$ is defined as

$$\mathrm{Var}[X] := \mathbb{E}\left[(X - \mu)^2\right] = \sum_{x \in W_X} (x - \mu)^2 \cdot \Pr[X = x]$$

The measure $\sigma := \sqrt{\mathrm{Var}[X]}$ is called standard deviation ("Standardabweichung") of X.

Just as the expected value, the variance may not be defined. We don't consider such cases.

Theorem: Let X be some arbitrary random variable.

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

Proof.

$$\mathbb{E}\left[(X - \mu)^2\right] = \mathbb{E}\left[X^2 - 2\mu X + \mu^2\right]$$
$$= \mathbb{E}\left[X^2\right] - 2\mu\mathbb{E}[X] + \mu^2 = \mathbb{E}\left[X^2\right] - 2\mu\mu + \mu^2 = \mathbb{E}\left[X^2\right] - \mu^2$$

□

Theorem: Arbitrary random variable X and $a, b \in \mathbb{R}$.

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

Proof. With the definition of variance and linearity of the expected value:

$$Var[X+b] = \mathbb{E}[(X+b-\mathbb{E}[X+b])^2] = \mathbb{E}[(X-\mathbb{E}[X])^2] = Var[X]$$

For thsi we use the previous theorem and linearity of the expected value:

$$Var[a \cdot X] = \mathbb{E}[(a \cdot X)^2] - \mathbb{E}[a \cdot X]^2 = a^2\mathbb{E}[X^2] - (a\mathbb{E}[X])^2 = a^2 \cdot Var[X]$$

□

Definition moment (”Moment”): For some arbitrary random variable X, $\mathbb{E}\left[X^k\right]$ is called the k-th moment (”k-te Moment”) and $\mathbb{E}\left[(X-\mathbb{E}[X])^2\right]$ is called the k-th central moment (”k-te zentrale Moment”).

11.3 Examples

Coin Throw This example showcases that an expected value is not always defined according to our definition.

We throw a coin until it shows head. Let k denote the number of throws. For the profit of the game provider, we define

$$G := \begin{cases} 2^k & k \text{ odd} \\ -2^k & k \text{ even} \end{cases}$$

As the individual throws have independent probability $\frac{1}{2}$, we get

$$\Pr[\text{”number of throws} = k\text{”}] = \frac{1}{2}^k$$

For the expected value we then get

$$\sum_{k=1}^{\infty} (-1)^{k-1} \cdot 2^k \frac{1}{2}^k = +1-1+1-1+1-1+1-\ldots$$

Grid Randomization Algorithm We consider some network and intend to identify a largest possible independent set (”stable set”). A stable setis a set of vertices, which are not connected with edges. The randomized algorithm has two basic steps:

1. Each vertex deletes itself with probability $1-p$.
2. All remaining edges randomly delete one vertex.

S :=”number of vertices in the stable set computed by the algorithm with $n = |V|$ and $m = |E|$”. We use the random variables X :=”number of vertices, which remain” and Y :=”number of edges, which remain”, each after the first step. So, after the first algorithm step we have X vertices. In the second step, we remove vertices. For each edge, one vertex may be removed at most. Hence, we have $S \geq X - Y$. From linearity of tthe expected value: $\mathbb{E}[S] \geq \mathbb{E}[X] - \mathbb{E}[Y]$. We can compute those separately.

For $\mathbb{E}[X]$ we consider X_i :=”indicator variable that the i-th vertex remains”. Then $X = \sum_{i=1}^n X_i$. From $\mathbb{E}[X_i] = p$ we get $\mathbb{E}[X] = n \cdot p$.

For $\mathbb{E}[Y]$, Y_i :=”indicator variable that the i-th edge remains.” Then $Y = \sum_{i=1}^n Y_i$. From $\mathbb{E}[Y_i] = p^2$ (both vertices remain), we get $\mathbb{E}[Y] = m \cdot p^2$.

Then clearly $\mathbb{E}[S] \geq np - mp^2$

Theorem: For each $G = (V, E)$, the above given algorithm compute san independent set S with $\mathbb{E}[S] \geq np - mp^2$. $n = |V|, m = |E|$.

Proof. This is implied by the above algorithm analysis. □

Corollary: For each $G = (V, E)$, the above given algorithm computes an independent set S with $\mathbb{E}[S] \geq \frac{n^2}{2m}$ for $p = \frac{n}{2m}$. This S is the one with the highest expected cardinality.

Proof. This follows directly by finding the maximum of $np - mp^2$, which we can get from the derivative being 0: $0 = n - 2mp \Rightarrow p = \frac{n}{2m}$. □

12 Important Discrete Distributions

Here, we consider some common classes of random variables/distributions (remember that a random variable is fully defined by its density/distribution function).

12.1 Bernoulli distribution ("Bernoulli-Verteilung")

Random variable X with $W_X = \{0, 1\}$ and $f_X(x)$:

- $f_X(x) = p \Leftrightarrow x = 1$
- $f_X(x) = 1 - p \Leftrightarrow x = 0$
- $f_X(x) = 0$, otherwise

p is called success probability ("Erfolgswahrscheinlichkeit"). We write $X \sim \text{Bernoulli}(p)$ is X is Bernoulli distributed with success probability p.

- $\mathbb{E}[X] = p$
- $\text{Var}[X] = \mathbb{E}\left[(X-p)^2\right] = p(1-p)^2 + (1-p)(0-p)^2 = p(1-)$

12.2 Binomial distribution ("Binomial-Verteilung")

This can be understood as running an Bernoulli experiment n times and counting success.

Random variable X with $X_X = \{0, 1\}$ and $f_X(x)$:

- $f_X(x) = \binom{n}{x} p^x (1-p)^{n-x}$ if $x \in \{0, 1, ..., n\}$
- $f_X(x) = 0$, otherwise

We write $X \sim \text{Bin}(n, p)$.

- $\mathbb{E}[X] = np$
- $\text{Var}[X] = np(1-p)$, comes from $\text{Var}[X] = \text{Var}[X_1] + ... + \text{Var}[X_n]$ (independent Bernoulli experiments)

12.3 Poisson distribution (”Poisson-Verteilung”)

Random variable X with $W_X = \mathbb{N}_0$ and $f_X(x)$:

- $f_X(i) = \frac{e^{-\lambda}\lambda^i}{i!}$, $i \in \mathbb{N}_0$
- $f_X(i) = 0$, otherwise

We write $X \sim \text{Po}(\lambda)$.

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

Intuition for this is harder. The Poisson distribution is a probability distribution that is commonly used to model the number of events that occur within a fixed interval (of time/space/...), given that these events occur with a known average rate and independently of each other.

- λ is the average rate of value for some interval
- $\Pr[X = k]$ is the probability of k events in the interval

This comes as the limit of the Binomial distribution with $\text{Bin}(n, \frac{\lambda}{n}), n \to \infty$.

If for a Binomial distribution n is very large and np (expected value) is a small constant: The Binomial distribution can be approximated with the Poisson distribution. The parameter for the poisson distribution is then given by the expected value of the Binomial distribution.

12.4 Geometric distribution

Random variable X with $W_X = \mathbb{N}_0$ and $f_X(x)$:

- $f_X(i) = p(1-p)^{i-1}$, $i \in \mathbb{N}_0$
- $f_X(i) = 0$, otherwise

We write $X \sim \text{Geo}(p)$.

- $\mathbb{E}[X] = \frac{1}{p}$, $\mathbb{E}[X] = \sum_{k \geq 1} \Pr[X \geq k] = \sum_{k \geq 0} (1-p)^k = \frac{1}{p}$
- $\text{Var}[X] = \frac{1-p}{p^2}$

Notice that we also have $F_X(n) = \Pr[X \leq n] = \sum_{i=1}^{n} \Pr[X = i] = \sum_{i=1}^{n} p(1-p)^{i-1} = 1 - (1-p)^n$. This can also be reasoned with $\Pr[X \geq k] = \sum_{i=k}^{\infty} \Pr[X = i] = \sum_{i=k}^{\infty} p(1-p)^{i-1} = p(1-p)^{k-1} \sum_{i \geq 0} (1-p)^i = p(1-p)^{k-1} \frac{1}{p} = (1-p)^{k-1}$, which is also the foundation for the expected value.

For intuition: This can be understood as repeating some Bernoulli experiment until we see failure. Then the number of throws is x (the number of successes $x - 1$). With this we can compute the probability for some number of throws.

Oen important characteristic of this experiment is ”Gedächtnislosigkeit”.

Theorem: $X \sim \text{Geo}(p)$. For all $s, t \in \mathbb{N}$:

$$\Pr[X = s + t | X > s] = \Pr[X = t]$$

Proof. We know $\Pr[X \geq n] = (1-p)^{n-1}$. With that:

$$\Pr[X = s+t | X > s] = \frac{\Pr[(X = s+t) \cap (X > s)]}{\Pr[X > s]} = \frac{\Pr[X = s+t]}{\Pr[X > s]} = \frac{(1-p)^{s+t-1}p}{(1-p)^s} = (1-p)^{t-1}p = \Pr[X$$

$\square$

12.5 Coupon Collector

There are n items and in each 'round', we get draw one of those items. In each round the probability foreach item is the same (LaPlace). Hence, the probabily for any item in any round is $\frac{1}{n}$. $X :=$"number of rounds until we have all n items".

Considering individual rounds is difficult, because the probability of getting a new card during each round depends on earlier rounds. The solving idea is to consider phases instead. Phases are an important general concept used in many instances in probability theory. We define phases in such a way that they are independent and then utilize the independence of the expected value/variance.

We define phases i to be all rounds in which we have exactly $i-1$ distinct items. For phase i, we then know the probability of success is $\frac{n-(i-1)}{n}$. For each phase, we also define $X_i :=$"length of phase i". We then clearly get X as the sum of the lengths of all those phases: $X = \sum_{i=1}^{n} X_i$.

We compute $\mathbb{E}[X]$ by considering all $\mathbb{E}[X_i]$. Notice that each phase is geometrically distributed (probability constant and wait for success): $X_i \sim \text{Geo}(\frac{n-(i-1)}{n})$ with $\mathbb{E}[X_i] = \frac{n}{n-i+1}$.

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

H_n is the n-th harmonic number ("harmonische Zahl"). Analysis tells us that $H_n = \ln n + \mathcal{O}(1)$.

For the variance we use $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$.

$$\begin{aligned}
Var[X] &= \sum_{i=1}^{n} Var[X_i] = \sum_{i=1}^{n} \frac{1-p_i}{p_i^2} \leq \sum_{i=1}^{n} \frac{1}{p_i^2} \\
&= \sum_{i=1}^{n} (\frac{n}{n-i+1})^2 = n^2 \sum_{i=1}^{n} \frac{1}{i^2} \leq n^2 \frac{\pi^2}{6}
\end{aligned}$$

Chebyshev, we then get

$$\Pr[|X - \mathbb{E}[X]| \geq f(n)] \leq \frac{Var[X]}{(f(n))^2} \leq \frac{\pi^2 n^2}{6(f(n))^2}$$

To better understand this, consider whether getting the first or last k cards would better. It is always better to get the last card(s), even if they are noticeably less.

- Getting the first cards

  If we get the first k cards, we can disregard the first k phases, as we have basically skipped those. For $\mathbb{E}[X]$, we hence must only consider this sum:

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

  With $k = \frac{n}{2}$, then $= n \ln(\frac{n}{2}) + \mathcal{O}(n) = n \ln n + \mathcal{O}(n)$.

- Getting the last cards

  If we get the last k cards, we can disregard the last k phases, as we skip those. For $\mathbb{E}[X]$, we hence must only consider this sum:

  $$\mathbb{E}[X] = \sum_{i=1}^{n-k} \mathbb{E}[X_i] = \sum_{i=k+1}^{n} \frac{1}{i} = n(H_n - H_k)$$

  With $k = 1$, then $= n(H_n - H_k) + \mathcal{O}(n) = n \ln 2 + \mathcal{O}(n) = \mathcal{O}(n)$.

12.6 Negative binomial distribution

Random variable X with $W_X = \mathbb{N}$ and $f_X(x)$:

- $f_X(i) = \binom{i-1}{n-1}(1-p)^i p^n$, $i \in \mathbb{N}$
- $f_X(i) = 0$, otherwise

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

- $\mathbb{E}[X] = \frac{n}{p}$
- $\text{Var}[X]$ also comes with linearity from geometric distribution

For intuition: This considers doing an Bernoulli experiment with probability p until we have seen success n times. $\Pr[X = k]$ computes the probability that we manage to do so in exactly k tries.

13 Multiple Random Variables

To work with multiple random variables X, Y in the same probability space, we consider $\Pr[X = x, Y = y] = \Pr[\{\omega \in \Omega | X(\omega) = x, Y(\omega) = y\}]$.

Also, we have a shared density function $f_{X,Y} := \Pr[X = x, Y = y]$.

Definition marginal density ("Randdichte"): Consider random variables X, Y

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

This holds because $Y = y$ is a disjunct partitioning ("Zerlegung") of the probability space, and we can use the addition rule.

Analogously for the distribution function and marginal distribution:

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

13.1 Independence of Random Variables

Definition: Random variables $X_1, ..., X_n$ are independent if and only if for all $(x_1, ..., x_n) \in W_{X_1} \times ... \times W_{X_n}$

$$\Pr[X_1 = x_1, ..., X_n = x_n] = \Pr[X_1 = x_1] \cdot ... \cdot \Pr[X_n = x_n]$$
$$\text{equivalently: } f_{X_1,...,X_n}(x_1, ..., x_n) = f_{X_1}(x_1) \cdot ... \cdot f_{X_n}(x_n)$$

We do not require this to hold for subsets as those equations follows from this.

Lemma: $X_1, ..., X_n$ independent random variables. $S_1, ..., S_n \subseteq \mathbb{R}$ arbitrary sets.

$$\Pr[X_1 \in S_1, ..., X_n \in S_n] = \Pr[X_1 \in S_1] \cdot ... \cdot \Pr[X_n \in S_n]$$

Proof. We can assume that S_i are subsets of the codomain wlog.

$$\begin{aligned} &\Pr[X_1 \in S, ..., X_n \in S_n] \\ &= \sum_{x_1 \in S_1} ... \sum_{x_n \in S_n} \Pr[X_1 = x_1, ..., X_n = x_n] \\ &= \sum_{x_1 \in S_1} ... \sum_{x_n \in S_n} \Pr[X_1 = x_1] \cdot ... \cdot \Pr[X_n = x_n] \\ &= (\sum_{x_1 \in S_1} \Pr[X_1 = x_1]) \cdot ... \cdot \left(\sum_{x_n \in S_n} \Pr[X_n = x_n]\right) \\ &= \Pr[X_1 \in S_1] \cdot ... \cdot \Pr[X_n \in S_n] \end{aligned}$$

□

Corollary: $X_1, ..., X_n$ independent random variables. $I = \{i_1, ..., i_k\} \subseteq [n]$. Then, $X_{i_1}, ..., X_{i_k}$ are also independent.

Proof. This is the proof from the script. However, this follows from the intuition why we do not require to consider subsets in the definition.

For $x_{i_j} \in W_{X_{i_j}}, 1 \leq j \leq k$, we define $S_i = W_{X_i}$ if $i \notin I$ and $S_i = \{x_i\}$ if $i \in I$. For $i \notin I$, $X_i \in S_i$ is then trivially valid and with Lemma 2.53 we get

$$\begin{aligned} \Pr[X_{i_1} = x_1, ..., X_{i_k} = x_k] &= \Pr[X_1 \in S_1, ..., X_n \in S_n] \\ = \Pr[X_1 \in S_1] ... \Pr[X_n \in S_n] &= \Pr[X_{i_1} = x_{i_1}] ... \Pr[X_{i_k} = x_{i_k}] \end{aligned}$$

□

Theorem: $f_1, ..., f_n$ are real functions. If $X_1, ..., X_n$ are independent, then also $f(X_1), ..., f(X_n)$ are independent.

Proof. We consider arbitrary $z_1, ..., z_n$ with $z_i \in W_{f(X_i)}$ for $i = 1, ..., n$. For z_i we define the set $S_i = \{x | f(x) = z_i\}$. With the Lemma above:

$$\begin{aligned} \Pr[f_1(X_1) = z_1, ..., f_n(X_n) = z_n] &= \Pr[X_1 \in S_1, ..., X_n \in S_n] \\ = \Pr[X_1 \in S_1] ... \Pr[X_n \in S_n] &= \Pr[f_1(X_1) = z_1] ... \Pr[f_n(X_n) = z_n] \end{aligned}$$

□

Theorem: X, Y are two indicator variables. Then X and Y are independent $\Leftrightarrow$ $f_{X,Y}(1,1) = f_X(1) \cdot f_Y(1)$.

Proof. This follows when considering the complementary elements. $\square$

13.2 Compound Random Variables ("zusammengesetzte Zufallsvariablen")

g is some function and $X_1, ..., X_n$ are random variables. We construct random variable $Y := g(X_1, ..., X_n)$.

Theorem: X, Y independent random variables. $Z := X + Y$. Then:

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

Proof. This follows from the law of total probability.

$$f_Z(z) = \Pr[Z = z] = \sum_{x \in W_X} \Pr[X + Y = z | X = x] \cdot \Pr[X = x]$$
$$= \sum_{x \in W_X} \Pr[Y = z - x] \cdot \Pr[X = x] = \sum_{x \in W_X} f_X(x) \cdot f_Y(z - x)$$

$\square$

With this, one can show for instance

- $\text{Bin}(n, p) + \text{Bin}(m, p) = \text{Bin}(n + m, p)$
- $\text{Poisson}(\lambda_1) + \text{Poisson}(\lambda_2) = \text{Poisson}(\lambda_1 + \lambda_2)$

13.3 Moments of Compound Random Variables

Theorem: $X_1, ..., X_n$ independent random variables.

$$\mathbb{E}[X_1 \cdot ... \cdot X_n] = \mathbb{E}[X_1] \cdot ... \cdot \mathbb{E}[X_n]$$

Proof. The general case follows by induction from this:

$$\begin{aligned}\mathbb{E}[X \cdot Y] &= \sum_{x \in W_X} \sum_{y \in W_Y} xy \Pr[X = x, Y = y] \\ &= \sum_{x \in W_X} \sum_{y \in W_Y} xy \Pr[X = x] \Pr[Y = y] \\ &= \sum_{x \in W_X} x \Pr[X = x] \sum_{y \in W_Y} y \Pr[Y = y] \\ &= \mathbb{E}[X] \cdot \mathbb{E}[Y]\end{aligned}$$

$\square$

Theorem: $X_1, ..., X_n$ independent random variables. $X = X_1 + ... + X_n$.

$$\text{Var}[X] = \text{Var}[X_1] + ... + \text{Var}[X_n]$$

Proof. The general case follows by induction from this. We intend to show:

$$\mathbb{E}[(X+Y)^2] - \mathbb{E}[X+Y]^2 = \mathbb{E}[X^2] - \mathbb{E}[X]^2 + \mathbb{E}[Y^2] - \mathbb{E}[Y]^2$$

This holds as

$$\mathbb{E}[(X+Y)^2] = \mathbb{E}[X^2 + 2XY + Y^2] = \mathbb{E}[X^2] + 2\mathbb{E}[X]\mathbb{E}[Y] + \mathbb{E}[Y^2]$$

and

$$\mathbb{E}[X+Y]^2 = (\mathbb{E}[X] + \mathbb{E}[Y])^2 = \mathbb{E}[X]^2 + 2\mathbb{E}[X]\mathbb{E}[Y] + \mathbb{E}[Y]^2$$

□

Multiplicity of the variance does generally NOT hold.

13.4 Wald's Equations

Theorem Wald's equation: N, X independent random variables. $W_N \subseteq \mathbb{N}$. $Z := \sum_{i=1}^{N} X_i$ where X_i are independent copies of X. Then:

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

Proof.

$$\mathbb{E}[Z] = \sum_{n \in W_\mathbb{N}} \mathbb{E}[Z|N=n] \cdot \Pr[N=n]$$

$$= \sum_{n \in W_\mathbb{N}} \mathbb{E}[X_1 + ... + X_n] \cdot \Pr[N=n]$$

$$= \sum_{n \in W_\mathbb{N}} n \cdot \mathbb{E}[X] \cdot \Pr[N=n] = \mathbb{E}[X] \sum_{n \in W_\mathbb{N}} n \cdot \Pr[N=n] = \mathbb{E}[X] \cdot \mathbb{E}[N]$$

□

14 Estimating Probabilities

The intention here is similar as with the variance. The variance is a general metric to measure the deviation from the expected value. Here, we compute specific bounds for for deviations of the expected value for random variables.

14.1 Markov's Inequality

Theorem Markov's Inequality ("Ungleichung von Markov"): X is some random variable, which is never negative. $\forall t \in \mathbb{R}, t > 0$:

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

Equivalently: $\Pr[X \geq t \cdot \mathbb{E}[X]] \leq \frac{1}{t}$.

Proof.

$$\begin{aligned}\mathbb{E}[X] &= \sum_{x \in W_X} x \cdot \Pr[X = x] \\ &= \sum_{x \in W_X, x<t} x \cdot \Pr[X = x] + \sum_{x \in W_X, x \geq t} x \cdot \Pr[X = x] \\ &\geq \sum_{x \in W_X, x \geq t} x \cdot \Pr[X = x] \\ &\geq \sum_{x \in W_X, x \geq t} t \cdot \Pr[X = x] \\ &= t \cdot \sum_{x \in W_X, x \geq t} \Pr[X = x] \\ &= t \cdot \Pr[X \geq t]\end{aligned}$$

□

14.2 Chebyshev's Inequality

Theorem Chebyshev's Inequality ("Ungleichung von chebyshev"): X random variable. $t \in \mathbb{R}, t > 0$. Then:

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

Equivalently: Pr[].

Proof. We have $\Pr[|X - \mathbb{E}[X]| \geq t] = \Pr[(X - \mathbb{E}[X])^2 \geq t^2]$.
We define $Y := (X - \mathbb{E}[X])^2$. Notice that $\mathbb{E}[Y] = \text{Var}[X]$.
Because $Y \geq 0$, we can apply Markov's Inequality and get:

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

□

With the standard deviation σ we get

$$\Pr[|X - \mathbb{E}[X]| \geq C\sigma] \leq \frac{Var[X]}{(C\sigma)^2} = \frac{1}{C^2}$$

14.3 Chernoff's Inequality

Theorem Chernoff's Inequality ("Ungleichung von Chernoff"): $X_1, ..., X_n$ independent Bernoulli distributed random variables. $\Pr[X_i = 1] = p_i$ and $\Pr[X_i = 0] = 1 - p_i$. We have $X := \sum_{i=1}^{n} X_i$.

1. $\Pr[X \geq (1+\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{3}\delta^2 \mathbb{E}[X]}$, $\forall 0 < \delta \leq 1$
2. $\Pr[X \geq (1-\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{2}\delta^2 \mathbb{E}[X]}$, $\forall 0 < \delta \leq 1$
3. $\Pr[X \geq t] \leq 2^{-t}$, $t \geq 2e\mathbb{E}[E]$

Proof. The basic idea is to as follows:

$$\Pr[X \geq (1+\delta)\mathbb{E}[X]] \Leftrightarrow \Pr[e^{tX} \geq e^{t(1+\delta)\mathbb{E}[X]}] \leq \frac{\mathbb{E}[e^{tX}]}{e^{t(1+\delta)\mathbb{E}[X]}} \leq ... \leq e^{-\frac{1}{3}\delta^2\mathbb{E}[X]}$$

The ... corresponds to some clever transformation steps and a smart choice of t.

Case 3 is a bit different (in choosing a different base) but follows the same concept. We consider it as an example.

$$\Pr[X \geq t] \leq 2^{-t}$$
$$\Leftrightarrow \Pr[4^X \geq 4^t] \leq \frac{\mathbb{E}\left[4^X\right]}{4^t}$$

using Markov & e increasing

Notice that X_i are independent. We can, hence, use multiplicity of the expected value to compute $\mathbb{E}\left[4^X\right]$:

$$\mathbb{E}\left[4^X\right] = \mathbb{E}\left[4^{\sum_{i=1}^n X_i}\right] = \mathbb{E}\left[\prod_{i=1}^n 4^{X_i}\right] = \prod_{i=1}^n \mathbb{E}\left[4^{X_i}\right]$$

$\mathbb{E}\left[4^{X_i}\right]$ is either 1 or 4 as X_i is an indicator variable. We can, hence, directly use the definition and some analysis to compute a bound for the expected value:

$$\mathbb{E}\left[4^{X_i}\right] = 4p_i + (1-p_i) = 1 + 3p_i \leq e^{3p_i}$$

From that follows the solution

$$\mathbb{E}[4^X] \leq \prod_{i=1}^n e^{3p_i} = e^{3\sum_{i=1}^n p_i} = e^{3\mathbb{E}[X]} \overset{t \geq 2e\mathbb{E}[X]}{\leq} e^{\frac{3t}{2e}} \leq 2^t$$

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[4^X]}{4^t} \leq \frac{2^t}{4^t} = 2^{-t}$$

□

14.4 Comparison of Inequalities

Depending on δ and n, the different Inequalities provide different bounds. It can not be generally said, which inequality is better as that depends on the parameters.

n	Chebyshev	Chernoff
1000	0.1	0.270961
2000	0.05	0.0424119
5000	0.02	0.000244096
10000	0.01	$5.77914 \cdot 10^{-8}$
100000	0.001	$4.14559 \cdot 10^{-73}$

Figure 3: Comparison of Chebyshev und Chernoff

15 Randomized Algorithms

Randomized algorithms are an application of probability theory in computer science. Remember that a non-randomized/standard algorithm works by taking an input I, running an algorithm $\mathcal{A}$, and by doing so computing an output $\mathcal{A}(I)$. We must always prove correctness, i.e., that $\mathcal{A}(I)$, for some I, is the intended output. Similarly, we must show that for some I with $|I| = n$ that some bound $\mathcal{O}(f(n))$ holds.

Randomized algorithm work differently. They have some input I but the algorithm also has access to random bits/numbers $\mathcal{R}$ from which the output $\mathcal{A}(I, \mathcal{R})$ is computed. So, the output can generally not be reproduced (not considering PRNGs with the same seed).

Here, we must also prove correctness: Either as before or just giving a correctness probability $\Pr[\mathcal{A}(I, \mathcal{R}) \text{ is correct}] \geq ....$ The runtime for some $|I| = n$ may similarly be as before or $\Pr[\text{runtime} \leq \mathcal{O}(f(n))]$.

The goal is to design algorithms, where the bounds leave no practically relevant error, i.e., correctness is effectively 1.

Considering the random numbers: We assume that the algorithm may specify some random variable X and then receive numbers, which could not be reasoned to not be random (but may be pseudo-generated). The algorithm must be able to assume independence.

15.1 Las-Vegas Algorithms

Las-Vegas algorithms are characterized by never returning a false answer. There are two options for this: Either the runtime is not bound and only has an expected value etc., or the algorithm terminates when reaching some runtime and returns unknown. This makes only sense if the expected runtime is efficient, i.e., polynomial. Otherwise, why should we even worry about introducing a new kind of algorithm?

We can easily convert both kinds of algorithms into each other. If the algorithm only has an expected runtime, we simply run it for twice the expected runtime. Markov then tells us that the probability that we do not get an answer is $\leq \frac{1}{2}$. We may now repeat this algorithm some times to increase the probability for a useful result. Thus, we now have a constant runtime and only an estimated probability to get a result.

If we have an algorithm which would have a finite runtime and only gives an expected value to deliver a useful result, specifically it returns unknown with probability δ, we can do the inverse. We repeatedly run this algorithm until we get a result. This is a geometric distribution. Hence, we have success probability $1 - \delta$ and the expected runtime $\frac{1}{1-\delta}$.

We consider 'improving' the probability of the version which has a finite runtime now.

Theorem Las-Vegas Algorithm: Let $\mathcal{A}$ be some randomized algorithm, which never returns a wrong answer but unknown at times. We know

$$\Pr[\mathcal{A}(I) \text{ correct}] \geq \epsilon$$

For all $\delta > 0$, $\mathcal{A}_\delta$ is the algorithm which calls $\mathcal{A}$ until it gets a useful return value or has gotten unknown for $N = \epsilon^{-1} \ln \delta^{-1}$ times. Then, it returns unknown. $\mathcal{A}_\delta$

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

Proof. The probability that $\mathcal{A}$ outputs only unknown for N times is $(1 - \epsilon)^N$. For $\mathcal{A}_\delta$'s correctness we then get we get:

$$(1-\epsilon)^N \leq e^{-\epsilon N} = e^{\ln \delta} = \delta \qquad\qquad 1-x \leq e^{-x}$$

□

15.2 Monte-Carlo Algorithms

Monte-Carlo Algorithm are a different kind of randomized algorithm. Those always have a polynomial runtime and at times they produce a wrong output. The goal is that Pr[wrong answer] =”minimal”.

The error of such a Monte-Carlo algorithm can either be one- or two-sided. We will consider the one-sided case first as it is easier to understand. Here, only inputs which should get one of two responses may have a wrong return value. The other type always produces a correct output.

Theorem Monte-Carlo Algorithm, one-sided: Let $\mathcal{A}$ be a randomized algorithm, which always outputs YES or NO.

- $\Pr[\mathcal{A}(I) = \text{YES}] = 1$, if I is a YES instance
- $\Pr[\mathcal{A}(I) = \text{NO}] \geq \epsilon$, if I is a NO instance

For $\delta > 0$, $\mathcal{A}_\delta$ then is the algorithm which calls $\mathcal{A}$ until it gets NO (which then is the answer) or it gets YES $N = \epsilon^{-1} \ln \delta^{-1}$ times (which then is the answer).

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

Proof. The case that I is a YES instance is trivially correct.

If I is a NO-instance, each instance returns NO with probability at least ϵ. The probability that among $N = \epsilon^{-1} \ln \delta^{-1}$ independent calls there is no NO is at most

$$(1-\epsilon)^N \leq e^{-\epsilon N} = e^{-\ln \delta^{-1}} = \delta$$

Accordingly, $\Pr[A_\delta(I) \text{ correct}] \geq 1-\delta$. □

Now, we consider the two-sided case.

Theorem:
Let $\epsilon > 0$ and $\mathcal{A}$ be a randomized algorithm, which always outputs YES or NO.

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

For $\delta > 0$, $\mathcal{A}_\delta$ is the algorithm, which makes $N = 4\epsilon^{-2} \ln \delta^{-1}$ independent calls to $\mathcal{A}$ and returns the majority result. Then:

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

Proof. Trivially, $\epsilon \leq \frac{1}{2}$. We set $p := \Pr[\mathcal{A}(I) \text{ correct}]$. Let X be the number of correct responses from $\mathcal{A}$ among the N calls to $\mathcal{A}(I)$. We then have $X \sim \text{Bin}(N, p)$.

$$\Pr[\mathcal{A}_\delta(I) \text{ correct}] \geq \Pr[X \geq \frac{N}{2}] = 1 - \Pr[X \leq \frac{N}{2}]$$

$\frac{N}{2} \leq \mathbb{E}[X]$ as $\mathbb{E}[X] = pN \geq \frac{N}{2} + \epsilon N$. From that follows:

$$\frac{N}{2} \leq (1-\epsilon)(\frac{N}{2} + \epsilon N) \leq (1-\epsilon)\mathbb{E}[X]$$

So we can use $\mathbb{E}[X] \geq \frac{N}{2} = 2\epsilon^{-2} \ln \delta^{-1}$ and Chernoff's Inequality:

$$\Pr[X \leq \frac{N}{2}] \leq \Pr[X \leq (1-\epsilon)\mathbb{E}[X]] \leq e^{-\frac{1}{2}\epsilon^2 \mathbb{E}[X]} \leq \delta$$

From that then follows $\Pr[\mathcal{A}_\delta(I) \text{ correct}] \geq 1 - \delta$. □

Theorem: $\epsilon > 0$ and $\mathcal{A}$ some maximization algorithm with $\Pr[\mathcal{A}(I) \geq f(I)] \geq \epsilon$. Then, for all $\delta > 0$: $\mathcal{A}_\delta$ is the algorithm which makes $N = \epsilon^{-1} \ln \delta^{-1}$ independent calls to $\mathcal{A}$ and returns the best/largest result. Then:

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

Proof.

$$(1-\epsilon)^{\mathbb{N}} \leq \exp(-\epsilon N) = \exp(-\ln \delta^{-1}) = \delta$$

□

15.3 Prime Number Test

For various applications we would like to know whether some number is prime. First, we know that $\pi(x) := |\{n \in \mathbb{N} | n \leq x, n \text{ prime}\}| \sim \frac{x}{\ln x}$ (true asymptotic bound) as can be shown with Target Shooting (see concept below). But that does not give information about individual numbers. But it shows that if we had some way to test numbers for primality this would be very helpful as the prime function just given implies a relatively high density of primes, even for large numbers. Considering 200 bits, about $\frac{1}{139}$.

We exclude brute force immediately because it's effort is way to high for 200 bits and more, what is required for modern applications.

15.3.1 Euclid Prime Test

We design a Monte-Caro algorithm based on those facts. Notice that the largest common divisor (ggT) can be computed in $\mathcal{O}((\log nm)^3)$.

$$ggT(a,n) > 1, a \in [n-1] \Rightarrow n \text{ not primt}$$
$$ggT(a,n) = 1 \not\Rightarrow n \text{ prime}$$

```
EuclidPrimeTest(n):
  choose a ∈ [n-1], randomly, uniformly
  if $ggT(a,n) > 1$: return 'not prime'
  else: return 'prime'
```

If n is not prime this may wrongly output 'prime' with probability $\frac{|\mathbb{Z}_n^*|}{n-1}$, $\mathbb{Z}_n^* = \{a \in [n-1] | ggT(a,n) = 1\}$. From discrete mathematics we know that $\mathbb{Z}_n^*$ is a multiplicative group and that some function $\varphi(n) := |\mathbb{Z}_n^*|$ exists.

If $n = p^2$ (p being prime), we get $\varphi(n) = p(p-1) = n - \sqrt{n}$. Then, $\frac{|\mathbb{Z}_n^*|}{n-1} \approx 1 - \frac{1}{\sqrt{n}}$.

15.3.2 Fermat Prime Test

Remember this from group theory (Lagrange's theorem): If H is a subgroup of G ($H \leq G$), then $|H|$ is a divider of $|G|$.

Also, $ord(a) := \min\{i \in \mathbb{N} | a^i = 1\}$ is the order of the subgroup generated by a. And as $ord(a) || G|$: $a^{|G|} = a^{ord(a)\frac{|G|}{ord(a)}} = 1^{\frac{|G|}{ord(a)}} = 1$.

So, in $\mathbb{Z}_n^*$: $a^{\phi(n)} = 1$ for all $a \in \mathbb{Z}_n^*$. If n prime, even $a^{n-1} = 1$.

The latter is Fermat's little theorem: If $n \in \mathbb{N}$ is prime, for all $a \in [n-1]$, we have $a^{n-1} \equiv 1(\mod n)$ or rather $a^{n-1} = 1$ in $\mathbb{Z}_n^*$.

From that, we get the Fermat prime test.

```
FermatPrimeTest(n):
  choose a from [n-1] randomly, uniformly
  IF $ggT(a,n) > 1$ OR $a^{n-1} \not\equiv 1(\mod n)$ THEN
    RETURN 'not prime'
  ELSE
    RETURN 'prime'
```

$a^{n-1} \not\equiv 1 \mod n$ can be effectively computed with binary exponentiation. And $ggT(a,n) > 1$ is redundant, because $ggT(a,n) > 1 \Rightarrow a^{n-1} \not\equiv 1 \mod n$.

This is a Monte-Carlo algorithm too as 'prime' may be also outputted for non-prime numbers. The error probability is $< \frac{1}{2}$ except for Carmichael numbers. The error occurs if n is not prime but for some a: $a^{n-1} \equiv 1 \mod n$. Such a are called pseudoprime bases of the pseudoprime n. We define $PB_n := \{a \in \mathbb{Z}_n^* | a^{n-1} \equiv 1 \mod n\}$ to be all pseudoprime bases for n. PB_n is a subgroup of $\mathbb{Z}_n^*$.

If $PB_n \neq \mathbb{Z}_n^*$ (which holds for all non-primes n except Carmichael numbers, see below), then $|PB_n|$ must be a proper divisor of $|\mathbb{Z}_n^*|$. So $|PB_n| \leq \frac{\varphi(n)}{2} \leq \frac{n-1}{2}$. If n is not prime, which is the case for which we intend to compute the error probability, we can even state $< \frac{n-1}{2}$.

So, for non-Carmichael numbers we can justify that the one-sided error is $\frac{|PB_n|}{n-1} < \frac{1}{2}$.

As already mentioned before, this does not work for Carmichael numbers. Some n is a Carmichael number if n is not prime but $PB_n = \mathbb{Z}_n^*$.

Another interesting remark is that $f(n) := 2^{n-1} \mod n$ for $n \geq 2$ is a pretty good prime indicator. It first fails at 341, ...

15.3.3 'not prime' certificate, i.e., "algorithm efficiency summary"

To prove that some $n \in \mathbb{N}$ is not prime, we can provide a certificate/proof. For n not prime, $a \in [n-1]$ is ...

- trivial certificate

  If $a \geq 2$ and $a|n$. There are $> \frac{1}{n}$ such certificates. Considering $a \in [\lfloor\sqrt{n}\rfloor]$ is sufficient.

- Euclid certificate

  If $ggT(a,n) > 1$. There are $> \frac{1}{\sqrt{n}}$ such certificates.

- Fermat certificate

  If $a^{n-1} \not\equiv 1 \mod n$. Neglecting Carmichael numbers there are $> \frac{1}{2}$ such certificates.

What's special about the Fermat certificate is that it shows that n is not prime without providing a divisor.

15.3.4 Miller-Rabin certificate

Before, we considered $\mathbb{Z}_n^*$. Here we consider $\mathbb{Z}_n$, which is a field if n is prime. In a field, $x^2 = 1$ has two solutions: $x = 1$ and $x = -1 = n - 1$. (We always consider $\mod n$ here.)

We now iteratively do a process, starting with $k = 1$:

- Compute $c = a^{\frac{n-1}{k}}$. There are only two options for the result by what has just been said.
- If $c = 1$ and $\frac{n-1}{2}$ is even, we can take the root by setting $k = k * 2$ and restart our iteration.
- If $c = -1$ or $\frac{n-1}{k}$ is odd we must stop.
- If $c \notin \{1, n-1\}$, then we have found a certificate for n not being prime.

Initially, $a^{n-1} = 1$ if $n \geq 2$ is prime. So, we always have a place to start.

Formally: Consider $2 < n \in \mathbb{N}$, $n - 1 = 2^k d$ with $d \in \mathbb{N}$ uneven and $k \in \mathbb{N}_0$. $a \in [n-1]$ is a Miller-Rabin certificate for 'n not prime' if

$$(a^d, a^{2d}, ..., a^{2^k d}) \neq \begin{cases} (1, 1, 1, ..., 1) \\ (*, ..., *, n-1, 1, ..., 1) \end{cases}$$

```
MillerRabinPrimeTest(n):
  d,k from N with $n-1 = 2^k d$, d uneven
  choose a in [n-1] randomly, uniformly
  IF (($a^d \mod n! = 1$) AND (not exists $i < k : a^{2^i d} \mod n = n-1$))
    RETURN 'not prime'
  ELSE
    RETURN 'prime'
```

This is a simpler to understand version.

```
MillerRabinPrimeTest(n):
  IF n==2: RETURN 'prime'
  ELSE IF n even OR n==1: RETURN 'not prime'
  chosse a from 2,...,n-1 randomly, uniformly
  k,d from Z with $n-1 = d2^k$ and d uneven
  x = $a^d$
  IF x==1 OR x==n-1: RETURN 'prime'
  REPEAT k-1 times:
    x = $x^2$ (mod n)
    IF x==1: RETURN 'not prime'
    IF x==n-1: RETURN 'prime'
  RETURN 'not prime'
```

The output 'prime' is wrong with probability $\leq \frac{1}{4}$. There was no justification given for this.

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

15.3.5 Remarks

The above protocols show that one can efficiently determine whether a number is prime probabilistically. However, finding divisors is considered way harder, even probabilistically. That finding the divisor is difficult is the foundation for many cryptography protocols including RSA. The Miller-Rabin test is used in practice. Additionally, the Solovay-Strassen test is also used in practice. Both are classical randomized algorithms from the 70s.

Note that a deterministic polynomial prime test exists: the AKS prime number test (Agrawal-Kayal-Saxena, 2002).

15.4 Target Shooting

Target shooting algorithms try to approximate some value, specifically the relative size of two sets.

We have sets $S \subseteq U$ of unknown size. We aim to identify $\frac{|S|}{|U|}$. We have $I_s : U \to \{0,1\}$, which indicates $u \in S$.

```
TARGET-SHOOTING:
  choose $u_1, ..., u_N$ from U randomly, independent, uniformly
  return $(\frac{1}{N}) * \sum_{i=1}^{N} I_S(u_i)$
```

There are two assumptions to this: I_S must be efficiently computable and finding $u \in U$ randomly, independently, and uniformly must also be efficiently possible.

We define the random variable $Y_i := I_S(u_i)$ for $i \in [N]$. Notice that $\Pr[Y_i = 1] = \frac{|S|}{|U|}$ for $i \in [N]$. Y_i are independent Bernoulli variables.

$Y := \frac{1}{N}\sum_{i=1}^{N} Y_i = \frac{1}{N}\sum_{i=1}^{N} I_S(u_i)$. We get $\mathbb{E}[Y] = \frac{|S|}{|U|}$, independent of N. However, the variance depends on N and shows that higher N increase proximity to the actual value.

$$\mathrm{Var}[Y] = \frac{1}{N}\left(\frac{|S|}{|U|} - \left(\frac{|S|}{|U|}\right)^2\right)$$

That comes from $\mathrm{Var}[Y] = \mathbb{E}\left[Y^2\right] - \mathbb{E}\left[Y\right]^2$:

$$\begin{aligned}
\mathrm{Var}[Y] &= \mathbb{E}\left[Y^2\right] - \mathbb{E}\left[Y\right]^2 \\
&= \mathbb{E}\left[\frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} Y_i Y_j\right] - \frac{|S|}{|U|}^2 \\
&= \frac{1}{N^2}\left((N^2 - N)\frac{|S|}{|U|}^2 + N\frac{|S|}{|U|}\right) - \frac{|S|}{|U|}^2 \\
&= \frac{N^2 - N}{N^2}\frac{|S|}{|U|}^2 + \frac{N}{N^2}\frac{|S|}{|U|} - \frac{|S|}{|U|}^2 \\
&= \frac{1}{N}\left(\frac{|S|}{|U|} - \frac{|S|}{|U|}^2\right)
\end{aligned}$$

Theorem: $\delta, \epsilon > 0$. If $N \geq 3\frac{|U|}{|S|}\epsilon^{-2}\ln(\frac{2}{\delta})$. Then, the output of the target shooting algorithm is in the interval $[(1-\epsilon)\frac{|S|}{|U|}, (1+\epsilon)\frac{|S|}{|U|}]$ with probability at least $1 - \delta$.

Proof. We must show

$$\Pr[|Y - \mathbb{E}[Y]| \geq \epsilon \cdot \mathbb{E}[Y]] \leq \delta$$

With $Z := \sum_{i=1}^{N} Y_i = NY$ this is equivalent to

$$\Pr[|Z - \mathbb{E}[Z]| \geq \epsilon \cdot \mathbb{E}[Z]] \leq \delta$$

Because Y_i are independent Bernoulli variables, we can use Chernoff.

$$\Pr[|Z - \mathbb{E}[Z]| \geq \epsilon \cdot \mathbb{E}[Z]] \leq 2e^{-\epsilon^2 \mathbb{E}[Z]/3} = 2e^{-\epsilon^2 \frac{N|S|}{3|U|}}$$

Because of $n = 3\frac{|U|}{|S|}\epsilon^{-2}\ln(\frac{2}{\delta})$ this is at most δ. □

15.5 QuickSort

We have discussed QuickSort in an earlier course and shown worst-case runtime $\mathcal{O}(n^2)$. It was mentioned that the algorithm is still relevant due to its expected runtime of $\mathcal{O}(n \log n)$, which we will show now.

```
QuickSort(A,l,r):
  if $l \geq r$: return
  p = Uniform({l,l+1,...,r-1,r}) # chosing a random pivot element
  t = Partition(A,l,r,p) # returns the index of the pivot after
      partitioning
  QuickSort(A,l,t-1)
  QuickSort(A,t+1,r)
```

$T_{l,r}$:=”(random) number of comparisons is ‘QuickSort(A,l,r)‘”. We want to show that $\mathbb{E}[T_{1,n}] \leq 2(n-1)\ln n + \mathcal{O}(n)$. For $l < r$ we get

$$\begin{aligned}\mathbb{E}[T_{l,r}] &= \sum_{i=l}^{r} \Pr[t = i](r - l + \mathbb{E}[T_{l,i-1}] + \mathbb{E}[T_{i+1,r}]) \\ &= \frac{1}{r-l+1}\sum_{i=0}^{r-l}(r - l + \mathbb{E}[T_{l,l+i-1}] + \mathbb{E}[T_{l+i+1,r}])\end{aligned}$$

$\mathbb{E}[T_{l,r}]$ does not depend on the specific values l and r but only on their difference/the number of elements which we consider.

- $t_n = 0$, $n \leq 1$
- $t_n = \frac{1}{n}\sum_{i=0}^{n-1}(n - 1 + t_i + t_{n-i-1})$, $n \geq 2$

See that $\mathbb{E}[T_{l,r}] = t_{r-l+1}$ for arbitrary l and r (can be formally shown with induction).

We now do some computations to find an upper bound for t_n:

Some pre-calculations, which follows directly from above.

$$n \cdot t_n = \sum_{i=0}^{n-1}(n-1+t_i+t_{n-i-1})$$
$$(n-1)t_{n-1} = \sum_{i=0}^{n-2}(n-2+t_i+t_{n-i-2})$$

Now we subtract the second from the first equation and get:

$$n \cdot t_n - (n-1)t_{n-1} = 2(n-1) + 2t_{n-1} - 1$$
$$\Rightarrow t_n = \frac{n+1}{n}t_{n-1} + \frac{2(n-1)}{n} - \frac{1}{n} \leq \frac{n+1}{n}t_{n-1} + 2$$

With induction over $n \geq 2$ one can show that this hodls:

$$t_n \leq 2\sum_{i=3}^{n+1}\frac{n+1}{i}$$

Remember that $\sum_{i=1}^{n}\frac{1}{i} = H_n = \ln n + \mathcal{O}(1)$. Using that, we can show that $\mathbb{E}\left[T_{1,n}\right] = t_n \leq 2(n+1)\ln n + \mathcal{O}(n)$.

This corresponds directly to our definition of a Las-Vegas algorithm: Never a false answer and the runtime being a random variable. However, we can rewrite the algorithm to confirm to our other definition. Specifically, we define 'SuperQuickSelect'. This algorithm runs standard 'QuickSelect'. But as soon as the runtime is double the expected runtime, 'QuickSelect' is terminated and the algorithm returns unknown. From Markov's inequality we get that the probability for ??? is $\leq \frac{1}{2}$ as $\Pr[X \geq 2\mathbb{E}[X]] \leq \frac{\mathbb{E}[X]}{2\mathbb{E}[X]}$.

Then, we define 'SuperSuperQuickSelect'. This algorithm runs 'SuperQuickSelect' n times. If only one 'SuperQuickSelect' returns a valid result, then also this algorithm does. So, this algorithm only returns unknown if 'SuperQuickSelect' returns unknown n times, which as probability 2^{-n}. The runtime, of course, increases with some constant factor n from 'SuperQuickSelect', but that is irrelevant for $\mathcal{O}$ notation. And when choosing n sufficient, such as $n = 100$, the probability that this algorithm returns unknown is negligible.

15.6 QuickSelect

QuickSelect builds on the same idea as QuickSort. However, it only tries to find some element in an ordered array instead of sorting the entire array. Hence, we can save some work. Specifically, we must only call the recursion once.

```
QuickSelect(A,l,r,k):
  p = Uniform({l,l+1,...,r-1,r})
  t = Partition(A,l,r,p)
  if t=l+k-1 then
    return A[t]
  else if t > l+k-1 then
    return QuickSelect(A,l,t-1,k)
  else
    return QuickSelect(A,t+1,r,k-t)
```

When running ‘QuickSelect(A,1,n,k)‘ we get N calls to ‘Partition‘ and a sequence of further calls to ‘QuickSelect‘ for subarrays: $(l_0, r_0, k_0), (l_2, r_2, k_2), ..., (l_N, r_N, k_N)$ with $(l_0, r_0, k_0) = (1, n, k)$.

During iteration i, t is chosen uniformly as t_i from $\{l_i, ..., r_i\}$. So either $(l_{i+1}, r_{i+1}) = (l_i, t-1)$ or $(l_{i+1}, r_{i+1}) = (t+1, r_i)$. The number of comparisons for ‘QuickSelect(A,1,n,k)‘ then is $T = \sum_{i=1}^{N}(r_i - l_i)$.

We define N_j as the number of calls to ‘QuickSelect‘ with $\frac{3}{4}^j n < r_i - l_i + 1 \leq \frac{3}{4}^{j-1} n$.

$$T \leq \sum_{j=1}^{\infty} N_j \frac{3}{4}^{j-1} n$$

With linearity of the expected value follows:

$$\mathbb{E}[T] \leq n \sum_{j=1}^{\infty} \mathbb{E}[N_j](\frac{3}{4})^{j-1}$$

Now we want to find an upper bound for $\mathbb{E}[N_j]$ to solve this equation. Being at step j, the probability that $r_{i+1} - l_{i+1} + 1$ is $\leq \frac{3}{4}(r_i - l_i + 1)$ is $\geq \frac{1}{2}$. This is apparent when considering that with probability $\frac{1}{2}$ we choose the new element from the middle $\frac{1}{2}(r_i - l_i + 1)$ elements. Following the laws of geometric distributions, we get $\mathbb{E}[N_j] \leq 2$.

$$\mathbb{E}[T] \leq 2n \sum_{j=1}^{\infty} \frac{3}{4}^{j-1} = 8n$$

So for QuickSelect we have $\mathcal{O}(n)$, i.e., linear runtime.

15.7 Finding Duplicates

... Not done.

Part III
Algorithms

16 Longest Paths

As many graph problems, finding longest paths has many applications such as network analysis, scheduling, and genetics.

Definition long-path problem: Given (G, B), G being some graph and $B \in \mathbb{N}_0$. Identify whether a path of length B exists in G.

Remember that a path is a sequence of unique, adjacent vertices. A path of length l has $l + 1$ vertices.

The long-path problem is NP-complete and we can find Hamiltonian cycles using it. Specifically, we can find the longest path/all longest paths. If length is $n-1$ and endpoints connect, we have found a Hamiltonian cycle. If endpoints do not connect or the maximum length is not $n - 1$, a Hamiltonian cycle does not exist.

Formally, for some graph G with n vertices we can efficiently construct G' with $n' \leq 2n - 2$ vertices such that G has a Hamiltonian cycle if and only if G' has a path of length n.

- In G, choose arbitrary vertex v, remove v, and replace all incident edges $\{v, w_1\}, ..., \{v, w_{deg(v)}\}$ with $\{\hat{w}_1, w_1\}, ..., \{\hat{w}_{deg(v)}, w_{deg(v)}\}$ where $\hat{w}_i$ are new vertices. This is graph G' with $(n - 1) + \deg(v) \leq 2n - 2$ vertices.

- We show: Hamiltonian cycle in $G \Rightarrow$ path of length n in G'.

  $\langle v_1, ..., v_n, v_1\rangle$ Hamiltonian cycle in G. Without loss of generality $v = v_1$. Then, $\langle \hat{v}_2, v_2, ..., v_n, \hat{v}_n\rangle$ is a path of length n in G'.

- We show: Path of length n in $G' \Rightarrow$ Hamiltonian cycle in G.

  Let $\langle u_0, u_1, ..., u_n\rangle$ be a path of length n in G'. Notice that $u_1, u_2, ..., u_{n-1}$ must have degree at least 2. Hence, those must be the $n - 1$ vertices, which survived the construction from G to G'. Also, $u_0 = \hat{w}_i$ and $u_n = \hat{w}_j$ must be two different of the new vertices with degree 1 in G'. Hence, $u_1 = w_i$ and $u_{n-1} = w_j$ and $\langle v, u_1, ..., u_{n-1}, v\rangle$ is a Hamiltonian cycle in G.

The construction from G to G' can be clearly done in $\mathcal{O}(n^2)$ steps.

Theorem: If we could decide the LONG-PATH problem for graphs with n vertices in $t(n)$, then we could determine whether a graph with n vertices has a Hamiltonian cycle in $t(2n - 2) + \mathcal{O}(n^2)$.

Nevertheless, in many applications the shortest paths are relatively short compared to the graph size (genetics for instance). For the length being in $\mathcal{O}(\log n)$ we now consider an efficient probabilistic algorithm. Instead of addressing the problem directly, we will first solve a related problem, from which we will derive the solution. We need some prerequisites to understand the following:

- $[n] := \{1, 2, ..., n\}$

  $[n]^k$ is the set of the sequences over $[n]$ of length k, from which follows that $|[n]^k| = n^k$

  $\binom{[n]}{k}$ is the set of k-element subsets of $[n]$, from which follows that $|\binom{[n]}{k}| = \binom{n}{k}$

- For some undirected $G = (V, E)$, we have $\sum_{v \in V} deg(v) = 2|E|$.
- k vertices can be colored with $[k]$ in k^k ways (also considering 'invalid' colorings). $k!$ of those colorings use each color only once.
- For $c, n \in \mathbb{R}^+$: $c^{\log n} = n^{\log c}$.
  $2^{\log n} = n^{\log 2} \Rightarrow 2^{\mathcal{O}(\log n)} = n^{\mathcal{O}(1)}$ is polynomial in n.
- For $n \in \mathbb{N}_0$: $\sum_{i=0}^{n} \binom{n}{i} = 2^n$.
  - Reason with binomial theorem: $\sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i} = (x + y)^n$
  - Reason by considering $\sum_{i=0}^{n} \binom{n}{i}$ as the sum of the count of all subsets of certain sizes, i.e., the number of all subsets (2^n)
- For $n \in \mathbb{N}_0$: $\frac{n!}{n^n} \geq e^{-n}$.
  Reason with $e^n = \sum_{i=0}^{\infty} \frac{n^i}{i!} \geq \frac{n^n}{n!}$
- Repeating some experiment with success probability p yields the expected value $\frac{1}{p}$ until success. (geometric distribution)
- dynamic programming

16.1 Colorful Paths ("Bunte Pfade")

For some $k \in \mathbb{N}$ and $G = (V, E)$ we consider some (not necessarily 'valid') coloring $\gamma : V \to [k]$. We say that a path is colorful if all its vertices have different colors.

Definition colorful-path problem: Given (G, γ), $G = (V, E)$ and $\gamma : V \to [k]$, decide whether a colorful path of length $k - 1$ (i.e., with k vertices) exists in G.

We fix some vertex v and $i \in \mathbb{N}_0$ and consider colorful paths of length i ($i + 1$ vertices) which end in v and consider all color sets, which we might encounter:

$$P_i(v) := \{S \in \binom{[k]}{i+1} | \exists \text{ colorful path colored with } S \text{ which ends in } v\ \}$$

Notice:

- $\forall S \in P_i(v) : \gamma(v) \in S$
- $P_0(v) = \{\{\gamma(v)\}\}$
- $P_1(v) = \{\{\gamma(x), \gamma(v)\} | x \in N(v), \gamma(x) \neq \gamma(v)\}$
- $\exists$ colorful path with k vertices $\Leftrightarrow \bigcup_{v \in V} P_{k-1}(v) \neq \varnothing$

We can then dynamically compute $P_i(u)$ from P_{i-1}:

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

This translates the computation of $P_i(v)$ from $P_{i-1}(v)$ to code:

```
Bunt(G,i): // G being $\gamma$-colored
  FOR ALL v in V:
    $P_i(v) = \varnothing$
    FOR ALL x in N(v):  // runtime: deg(v)
      FOR ALL R in $P_{i-1}(x)$ with $\gamma(v) \notin$R: // runtime: $|P_{i-1}(x)| \cdot i$
        $P_i(v) = P_i(v) \cup \{R \cup \{\gamma(v)\}\}$
```

This function needs to be executed multiple times to fill the dp table and solve the problem:

```
Rainbow(G,$\gamma$): // G being $\gamma$-colored
  FOR ALL v in V:
    $P_0(v) = \{\{\gamma(v)\}\}$
  FOR ALL i=1..k-1:
    Bunt(G,i)
  return $\bigcup_{v \in V} P_{k-1}(v) \neq \varnothing$
```

For the runtime of ‘Bunt‘, we get

$$\begin{aligned}\mathcal{O}(\sum_{v\in V} deg(v) \cdot |P_{i-1}(x)| \cdot i) &\leq \mathcal{O}(\sum_{v\in V} deg(v)\binom{k}{i} i)\\ &= \mathcal{O}(\binom{k}{i} i \sum_{v\in V} deg(v)) = \mathcal{O}(\binom{k}{i} \cdot i \cdot m)\end{aligned}$$

For the entire algorithm:

$$\begin{aligned}\mathcal{O}(|V| + \sum_{i=1}^{k-1}(\binom{k}{i} \cdot i \cdot m) + |V|) &\leq \mathcal{O}(|V| + \sum_{i=0}^{k}\binom{k}{i} \cdot k \cdot m)\\ &= \mathcal{O}(|V| + 2^k km) = \mathcal{O}(2^k km)\end{aligned}$$

So, if $k \leq \log n$, then we have runtime $\mathcal{O}(mn\log n)/\mathcal{O}(\text{poly}(n))$.

16.2 Long Path

We return to the long-path problem and want to solve it using our colorful paths solution with a randomized algorithm. Remember: We want to determine whether some graph G has a path of length B.

We set $k := B + 1$ and color G randomly with k colors and look for a (colorful) path with k vertices using the above constructed polynomial (for $\log n$ path length) algorithm. Considering that our coloring was random:

- If there is no path with k vertices in G, the algorithm will not find such a path, i.e., is always correct.
- If there is such a path with k vertices, the probability that in (G, γ) a colorful path exists is

$$\begin{aligned}p_{\text{success}} &:= \Pr[\exists \text{ colorful path with } k \text{ vertices}]\\ &\geq \Pr[P \text{ is colorful}] = \frac{k!}{k^k} \geq e^{-k}\end{aligned}$$

This is the basis for a Monte-Carlo algorithm. The expected number of tried until one finds a colorful path (and thus a path with k vertices) is $\leq e^k$ according to the laws of geometric distributions.

Theorem: Let G be some graph with a path of length $k-1$.

1. A random coloring with k colors creates a colorful graph of length $k-1$ with probability $p_{\text{success}} \geq e^{-k}$.
2. With repeated random colorings, the expected value until one finds some colorful path of length $k-1$ is $\frac{1}{p_{success}} \leq e^k$.

Assume that such a path k does exist, if we do $\lceil \lambda e^k \rceil$ repetitions, we have the runtime $\mathcal{O}(\lambda e^k 2^k km) = \mathcal{O}(\lambda (2e)^k km)$. And the probability that the algorithm does not find said path is $\leq (1-e^{-k})^{\lceil \lambda e^k \rceil} \leq (e^{-e^{-k}})^{\lceil \lambda e^k \rceil} \leq e^{-\lambda}$.

Theorem:

1. The algorithm has runtime $\mathcal{O}(\lambda (2e)^k km)$.
2. If the algorithm answers with 'Yes', a path of length $k-1$ exists.
3. If the graph has a path of length $k-1$, the probability that the graph answers with 'No' is at most $e^{-\lambda}$.

16.3 Remarks

$\exists$ a randomized improvement, which runs in $\mathcal{O}(2^k \text{poly}(n))$ instead of our $\mathcal{O}((2e)^k \text{poly}(n))$. Reducing the exponential base by e is a major improvement. But we do not discuss that algorithm here.

If one intends to find the path of length B and not only know that such a path exists, one needs to slightly adapt the algorithm. Basically, one does backtracking: For each $S \in P_i(v)$ one stores **one** path colored with S (i.e., the preceding element of $P_{i-1}(x)$ used to construct this S).

It is eqsy to see that this problem becomes easy in directed acyclic graphs. One can simply assign weight -1 to all edges and compute shortest paths.

17 Network Flows

This problem has a lot of practical applications and, luckily, is just on the easier side of the edge between P and NP.

Definition network ("Netzwerk"): A network is a tuple $N = (V, A, c, s, t)$.

- (V, A) being a directed graph (without loops)
- $s \in V$ being the source ("Quelle")
- $t \in V \backslash \{s\}$ being the target ("Senke")
- $c : A \to \mathbb{R}_0^+$ being the capacity function ("Kapazitätsfunktion")

Definition flow (”Fluss”): Given a network $N = (V, A, c, s, t)$, a flow N is a function $f : A \to \mathbb{R}$ which satisfies some conditions.

- feasability (”Zulässigkeit”): $0 \leq f(e) \leq c(e)$, $\forall e \in A$
- flow conservation (”Flusserhaltung”): $\forall v \in V \backslash \{s, t\}$:

$$\sum_{u\in V,(u,v)\in A} f(u, v) = \sum_{u\in V,(v,u)\in A} f(v, u)$$

The value (”Wert”) of some flow f is defined as

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

Lemma: The netinflow of the target t is equal to the value of the flow:

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

Proof.

$$0 = \sum_{(v,u)\in A} f(v, u) - \sum_{(u,v)\in A} f(u, v)$$

$$= \sum_{v\in V} \left(\sum_{u\in V,(v,u)\in A} f(v, u) - \sum_{u\in V,(u,v)\in A} f(u, v) \right)$$

Whats in the brackets is 0 for $v \notin \{s, t\}$.

$$= \left(\sum_{u\in V,(s,u)\in A} f(s, u) - \sum_{u\in V,(u,s)\in A} f(u, s) \right) + \left(\sum_{u\in V,(t,u)\in A} f(t, u) - \sum_{u\in V,(u,t)\in A} f(u, t) \right)$$

$$= val(f) - netinflow(f)$$

$$\Rightarrow val(f) = netinflow(f)$$

$\square$

The problem we are trying to solve is this.

Definition MaxFlow Problem: Given a network, find the flow with the largest value, i.e., the maximum flow.

Such a maximum flow always exists as will be seen.

17.1 Cuts

Definition: An s-t-cut for some network (V, A, c, s, t) is a partition (S, T) of V (i.e., $S \cup T = V$ and $S \cap T = \varnothing$) with $s \in S$ and $t \in T$. The capacity of some s-t-cut (S, T) is defined as

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

It is important to realize that we do not consider edges from T to S.

Lemma: Let f be a flow and (S, T) a s-t-cut in some network. Then:

$$val(f) \leq cap(S, T)$$

Proof. We define $f(S, T) := \sum_{(s,t) \in (S \times T) \cap A} f(s, t)$ for some partition (S, T) of V.

We claim

$$val(f) = f(S, T) - f(T, S) \leq f(S, T) \leq cap(S, T)$$

Prove the three relations from right to left:

- $f(S, T) = \sum_{(s,t) \in (S,T) \cap A} f(s, t) \leq \sum_{(s,t) \in (S,T) \cap A} c(s, t) = cap(S, T)$
- holds as $f(T, S) \geq 0$ because all flow values are non-negative by definition
-

$$\begin{aligned} val(f) &= \sum_{u \in V, (s,u) \in A} f(s, u) - \sum_{u \in V, (u,s) \in A} f(u, s) \\ &= \sum_{v \in S} \left(\sum_{u \in V, (v,u) \in A} f(v, u) - \sum_{u \in V, (u,v) \in A} f(u, v) \right) \end{aligned}$$

The content of the brackets is 0 for $v \neq s$ (mustn't consider t here)

$$\begin{aligned} &= \sum_{(u,w) \in (S,T) \cap A} f(u, w) - \sum_{(u,w) \in (T \times S) \cap A} f(u, w) \\ &= f(S, T) - f(T, S) \end{aligned}$$

□

So if we find f and (S, T) with $cap(S, T) = val(f)$, then f is a maximum flow. Hence, (S, T) is a simple proof/certificate for f being maximum. We will see now that such a flow must always exist.

Theorem: For each network (V, A, c, s, t):

$$\max_{f \text{ flow}} val(f) = \min_{(S,T) \text{ s-t-cut}} cap(S, T)$$

We prove the Maxflow-Mincut Theorem by considering an algorithm to compute the maximum flow below, which then also provides us with a way to get a s-t-cut with the capacity of the value of that flow. And from 3.8, we can then conclude the Maxflow-Mincut Theorem. A direct proof is not provided.

We will only consider a special case in this proof, however the theorem also holds generally. Specifically, we only consider integer flow values and do not allow edges in both directions between two vertices in the network. It is important to understand that we can not simply combine opposing edges into one edge. However, we can divide an edge by adding a vertex. But that increases the number of vertices up to the count of the number of directed possible edges. Our runtime is increased to $\mathcal{O}(m^2 U)$.

Notice that there are only finitely many s-t-cuts. Hence, the s-t-cut problem is finite (other than the MaxFlow problem). This implies that a minimal cut always exists and hence also a maximal flow.

17.2 Augmenting Paths

We will use the concept of augmenting paths in our proof. The notion is to consider some existing flow and find augmenting paths, along which we can improve the flow. This is how a single node can be augmented to still validate flow conservation. (Feasability also needs to hold afterwards of course.)

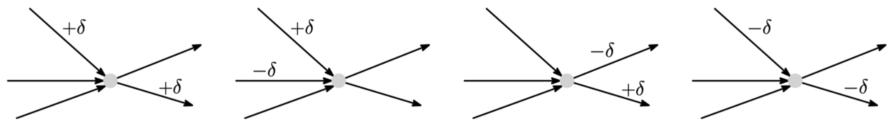

Figure 4: Node Augmentation Options

A flow augmentation ("Flussaugmentierung") then is an undirected path from the source to the target, which maintains flow conservation in each node (as described above) and feasability for each edge.

Notice that one is only guaranteed to end up/terminate at some point and has a maximum flow if the capacities are rational. Otherwise, one might infinitely long find and adapt augmenting paths.

17.3 Residual Network

The residual network is to help find flow augmentations.

Definition residual network: Let $N = (V, A, c, s, t)$ be a network without directed edges. f is a flow in N. We define the residual network $N_f := (V, A_f, r_f, s, t)$:

- $e \in A, f(e) < c(e) \Rightarrow e$ is in A_f with $r_f(e) := c(e) - f(e)$
- $e \in A, f(e) > 0 \Rightarrow e^{\text{opp}}$ is in A_f with $r_f(e^{\text{opp}}) := f(e)$
- A_f only contains edges such as in (1) or (2).

$r_f(e), e \in A_f$, is called residual capacity ("Restkapazität") of e.

All residual capacities are strictly positive.

Theorem: Let N be some network without opposing edges.

- some flow f is maximum
$$\Leftrightarrow$$
no directed s-t-path exists in the residual network N_f
- For each maximum flow f, some s-t-cut (S, T) with $val(f) = cap(S, T)$ exists.

Proof. We only proof the first item as the second item is implied by $\Leftarrow$ of the first item.

- We consider $\Rightarrow$ now.

  Indirect proof of implication: If in N_f some directed s-t-path exists $\Rightarrow$ f can be augmented $\Rightarrow$ f is no maximum!

Consider some directed s-t-path in N_f. Then, consider the smallest residual capacity $\epsilon := \min_i \epsilon_i$ This ϵ must be > 0 because. Then, augment along f by ϵ.

For the algorithm to follow, one then needs to update the network N and the residual network N_f. We then get a new flow f' with $val(f') = val(f) + \epsilon$. Hence, f could not have been a maximum flow.

- We consider $\Leftarrow$ now.

  We show: In N_f no directed s-t-path exists $\Rightarrow \exists$ s-t-cut (S, T) with $cap(S, T) = val(f) \Rightarrow f$ is maximum. We already know the latter implication from Lemma 3.8. Hence, we only consider the first one. We define:

  - $S :=$ vertices reachable in N_f from s
  - $T := V \backslash S$

  As s is reachable from s: $s \in S$. And as t not reachable from s (no s-t-path exists), $t \notin S$, i.e., $t \in T$. Hence, (S, T) is an s-t-cut. Then, we want to show $val(f) = f(S, T) - f(T, S) = cap(S, T)$. We have already shown the left equality before. For the right equality, consider:

  - edge $e \in A$ from S to T.

    We must have that $f(e) = c(e)$ as otherwise the residual network would contain some edge in the direction of e between its two vertices. Then, both vertices of e would be in S and it would not be an edge between S and T. We get this contradiction, which confirms $f(e) = c(e)$.
  - edge $e' \in A$ from T to S.

    We must have $f(e') = 0$ for the same reason as above. If it would not hold, then the flow could be reduced and the residual network would contain and edge directed from S to T with that value of $f(e')$. Then, both vertices of e' would be in S, a contradiction to our assumption that e' is from T to S.

  Using thsoe two considerations, we get $f(S, T)$ as the sum of capacities, i.e., $f(S, T) = cap(S, T)$. And we get $f(T, S) = 0$ as all flow values must be 0. Hence, $f(S, T) - f(T, S) = cap(S, T)$ as required.

□

17.4 Algorithm

We now combine the tools and insights from above.

```
Ford-Fulkerson(V,A,c,s,t):
  f = 0 // constant 0 flow
  WHILE exists s-t-path P in $N_f$:
    augment along path P
  return f
```

We notice that if this algorithm terminates, we get a maximum flow. But we can not guarantee that this algorithm terminates. And with capacities from $\mathbb{R}$ it is indeed possible that this algorithm never terminates. (But some maximum flow still exists!) However, with capacities from $\mathbb{N}_0$, flows and residual capacities also remain integers. Hence, during each augmentation step, the flow value is improved by ≥ 1. Accordingly, also the results are integers.

Also, using the approach of this algorithm, we can also easily find the minimum s-t-cut. Specifically by putting all reachable vertices in the final N_f in S and all others in T. But notice that we can not easily do the conversion in the other direction, i.e., converting a minimum s-t-cut to a specific maximum flow.

For $N = (V, A, c, s, t)$ consider $n := |V|$ and $m := |A|$. We now do the runtime analysis of this algorithm.

- Consider $c : A \to \mathbb{N}_0$ and $U := \max_{e \in A} c(e)$.

$$val(f) \leq cap(\{s\}, V \backslash \{s\}) \leq (n-1)U$$

  Hence, there are at most $(n-1)U$ augmentation steps.

- One augmentation step (finding an s-t-path in N_f, augmenting, updating N_f) can be done in $\mathcal{O}(m)$. Finding the path is trivial with only positive edge values. Augmenting requires updating some number of edges. (The same for N_f with at most twice the amount of edges).

Theorem Ford-Fulkerson algorithm: Let $N = (V, A, s, c, t)$ some network with $c : A \to \mathbb{N}_0^{\leq U}, U \in \mathbb{N}$ without opposing edges (not necessary but simplifying assumption). Then, an integer maximum flow exists. It can be computed in $\mathcal{O}(mnU)$.

This then also proves the Maxflow-Mincut Theorem for the case of networks with integer-valued capacities and without opposing edges.

Now we will state some advances algorithms. We only provides runtimes and characteristics.

Theorem capacity scaling, Dinitz-Gabow: If in some network all capacities are integers and at most U, some maximum flow exists and can be computed in $\mathcal{O}(mn(1 + \log U))$.

Theorem dynamic trees, Sleator-Tarjan: The maximum flow of some network can be computed in $\mathcal{O}(mn \log n)$.

17.5 Applications

It was already mentioned in the introduction that network flows have countless applications. We only provide some examples here.

17.5.1 Bipartite Matchings

We map some bipartite graph $G = (U \uplus W, E)$ to a network:

$$G = (U \uplus W, E) \mapsto N_G = (U \uplus W \uplus \{s, t\}, A, c, s, t)$$

- $s \neq t$ are new vertices
- $A := \{s\} \times U \cup \{(u, w) \in U \times W | \{u, w\} \in E\} \cup W \times \{t\}$
- $c \equiv 1$

Remember that we have shown the Ford-Fulkerson Theorem for integers. As N_G is a network with integer capacities $\leq U$, an integer-valued maximum flow exists and can be found in $\mathcal{O}(mnU) = \mathcal{O}(mn)$ as $U = 1$.

We must elaborate two implications:

- If a matching M exists in $G \Rightarrow$ a flow f_M in N_G with $val(f_M) = |M|$ exists.

  M being a maximum flow means that we can not get more connections from U to W. Now set the flow all corresponding edges in N_G to 1. And the vertices in U/W incident to some edge with flow 1 should then be connected to s/t. We would only get a flow with a higher flow value, if we can find some edge from U to W without using some already used vertex. Reusing a vertex would not improve the flow the input/output to that vertex is limited to 1 by the connection to s/t. But such another edge can not exist by the definition of a maximum matching.

- If an integer flow f in N_G exists $\Rightarrow$ a matching M in G with $|M| = val(f)$ exists.

  Notice that each path from s to t, considering a path if all edges have flow 1, may not branch. This is because 1 can not be divided as we only have integer values. And as the in-/out-flow to each veretx in U/W is limited to 1 by the one connection to s/t, also paths may not share vertices. Hence: the paths are disjunct and the U-W connections can be used as part of a matching.

When remembering that for any flow with some integer value, an integer flow with the same value exists, we get: Maximum matching in $G \simeq$ maximum integer flow in N_G.

Lemma: The cardinality of a maximum matching in the bipartite graph G has the same value as the maximum flow in the network N_G.

The construction of the matching then is quite simple given the maximum flow. All directed edges in the flow which have value 1 and correspond to some edge in G are part of the matching.

17.5.2 Edge-disjunct Paths

The problem: Given some graph G with two vertices u and v, $u \neq v$, find a maximum set of edge disjunct u-v-paths. We model the problem as a network:

$$G = (V, E), u, v \in V \mapsto N_G^* = (V, A, c, u, v)$$

- $A := \{(x, y), (y, x) | \{x, y\} \in E\}$
- $c \equiv 1$

We have opposing edges here. Despite not being proven, earlier statements on network flows also hold if we have opposing edges.

In N_G^* we then compute an/the maximum integer flow f with flow values $\in \{0, 1\}$.

We consider then the graph which only has those edges with flow value 1. First, we remove opposing edges (cycles of length 2) in that graph. This is called canceling. Second, we find the actual paths by starting at u and walkign along unused edges until we reach v. v is always reached as in-degree = out-degree. We mark al visited edges as used. We can repeat this $val(f)$ times to find $val(f)$ edge-disjunct walks. We get paths by removing cycles from those walks.

At this point we can also prove Menger's Theorem. Specifically, it follows directly from the Maxflow-Mincut theorem

$$\max_{f \text{ flow}} val(f) = \min_{s\text{-}t\text{-cut}} cap(S,T)$$

$$\max \#\text{edge disjunct } u\text{-}v\text{-paths in } G = \min \# \text{ edges whcih separate } u \text{ and } v$$

17.5.3 Vertex-disjunct Paths

Vertex disjunct paths follow from edge-disjunct paths. We only need to make a slight modification to the constructed network N_G^*. Specifically, we replace each $x \in V\backslash\{u,v\}$ with x_{in} and x_{out} where all input edges to x are redirected to x_{in} and all output edges are redirected to x_{out}. Then, we also add the edge $(x_{in}, x_{out}) = e$ with $c(e) = 1$.

17.5.4 Image Segmentation

We consider the problem: Given an image (pixels with color values), separate foreground and background.

We consider an image as a set P of pixels (mostly arranged in a grid) with color values (RGB, grayscale, b/w, ...) and neighboring relation, which states if two pixels are neighbors. From that, we model the problem as a network:

- An image is a graph (P,E) with the color information $x : P \to$ colors.
- Each pixel is individually assessed and a likelihood of being in the foreground: $\alpha : P \to \mathbb{R}_0^+$ - large $\alpha_p \Rightarrow$ likely in foreground.
- Each pixel is individually assessed and a likelihood of being in the background: $\beta : P \to \mathbb{R}_0^+$ - large $\beta_p \Rightarrow$ likely in background.
- Likelihood of two pixels being both in the fore/background: $\gamma : E \to \mathbb{R}_0^+$ - large $\gamma_e \Rightarrow$ likely in the same part (i.e. both foreground/background)

Those functions can be automatically generated, say using the brightness of a pixel for foreground (and the 'darkness' for background).

We define a quality function for some foreground/background separation (A,B) of P:

$$q(A,B) := \sum_{p\in A} \alpha_p + \sum_{p\in B} \beta_p - \sum_{e\in E, |e\cap A|=1} \gamma_e$$

With $Q := \sum_{p\in P}(\alpha_p + \beta_p)$ we can rewrite $q(A,B)$ to simplify the maximization for our network flow.

$$q(A,B) = Q - \sum_{p\in A} \beta_p - \sum_{p\in b} \alpha_p - \sum_{e\in E, |e\cap A|=1} \gamma_e$$

And as Q can be treated as a constant, we now want to minimize $q'(A,B)$:

$$q'(A,B) := \sum_{p\in A} \beta_p + \sum_{p\in b} \alpha_p + \sum_{e\in E, |e\cap A|=1} \gamma_e$$

Realize that (A,B) is a cut. So we want to use the Maxflow-Mincut Theorem to find the minimum cut. First we model the network. E $(\neq \vec{E})$ is the neighboring relation of the image.

$$N := (P \cup \{s,t\}, \vec{E}, c, s, t)$$

- $s \neq t$ are new vertices, source and target in N
- $\forall p \in P, e = (s, p) \in \vec{E}$ with $c(e) = \alpha_p$
- $\forall p \in P, e = (p, t) \in \vec{E}$ with $c(e) = \beta_p$
- $\forall e = (p, p') \in E$: $e' = (p, p') \in \vec{E}, e'' = (p', p) \in \vec{E}$ with $c(e') = c(e'') = \gamma_e$

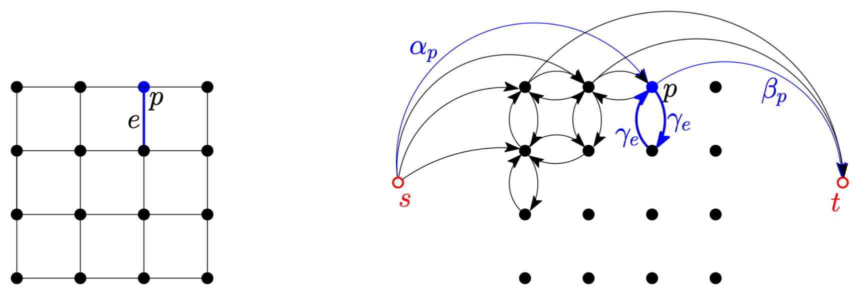

Figure 5: Image Segmentation Network

In that network N, now consider some s-t-cut (S, T) and $A := S\backslash\{s\}$ and $B := T\backslash\{t\}$. Those edges contribute to $cap(S, T)$:

- (s, p) with $p \in B$ - $\sum_{p\in B} \alpha_p$
- (p, t) with $p \in A$ - $\sum_{p\in A} \beta_p$
- $(p, p') \in A \times B$ - $\sum_{(p,p')\in A\times B, \{p,p'\}\in E} \gamma_{(p,p')}$

We notice: $cap(S, T) = \sum_{p\in A} \beta_p + \sum_{p\in B}^{\alpha_p} + \sum_{e\in E, |e\cap A|=1} \gamma_e = \cap(S, T)$. I.e., $q'(A, B) = cap(S, T)$.

We can find the (S, T) to minimize $cap(S, T)$ (and hence also q') by using Maxflow-Mincut and then get (A, B) from that.

Finally, notice that this method not only works for 2D-images but also multidimensional structures, i.e., multidimensional images, which are comprised of voxels instead of pixels.

17.5.5 Flows and Convex Sets

Refer to the optional section.

18 Minimal Cut

Definition multigraph: Undirected, unweighted graph $G = (V, E)$ without loops. However, G might have multiple edges between two vertices.
Notice that we can represent this also be considered a 'normal' weighted graph, where the weight at each edge represents the number of edges.

As degree we count the number of incident edges, not the number of neighbors as one could also do in a 'regular' graph. The handshaking lemma still clearly holds.

Definition edge cut ("Kantengraph"): Given some multigraph $G = (V, E)$, an edge cut is a set of edges C so that $(V, E\backslash C)$ is not connected.

The concept of edge cuts if linked to the concept of s-t-cuts. The latter consider the two separate sets while edge cuts consider how to get those two separate cuts.

Definition $\mu(G)$: With $\mu(G)$ we denote the cardinality of the smallest edge gut in G.

$$\mu(G) := \min_{C\subseteq E,(V,E\backslash C)\text{ not connected}} |C|$$

It is clear that $\mu(G) \leq \min_{v\in V} deg(v)$, as we can always remove all incident edges of some graph to make it unconnected.

We consider the MinCut problem: Given some multigraph G, identify the cardinality of the minimum edge cut, i.e., $\mu(G)$.

18.1 Flow-Based

Just by the term MinCut we notice that we can apply the Maxflow-Mincut theorem. Given some s and t we can already compute the smallest s-t-cut in $\mathcal{O}(mnU)/\mathcal{O}(mn(1 + \log U))$ with Hopcroft-Karp. The latter becomes $\mathcal{O}(mn \log n)$ as $U \leq n$ here.

We apply to known algorithm by fixing s and considering all $t \in V\backslash\{s\}$. This works as each cut is a s-t-cut for some $t \in V\backslash\{s\}$. So, we must run our above $\mathcal{O}(mn \log n)$ algorithm $n - 1$ times and get return the smallest cut. This has runtime $\mathcal{O}((n - 1)nm \log n) = \mathcal{O}(mn^2 \log n) = \mathcal{O}(n^4 \log n)$.

18.2 Edge Contraction

Contracting an edge $e = \{u, v\} \in E$ in some multigraph $G = (V, E)$ means unioning the two vertices u, v to a new vertex $x_{u,v}$. All edges incident to u or v are now incident to $x_{u,v}$, i.e., the respective endpoint is replaced to be $x_{u,v}$. All edges between u, v are removed. The resulting graph is denoted as G/e.

With $k :=$ #edges between u, v: $deg_{G/e}x_{u,v} = deg_G(u) + deg_G(v) - 2k$ and $|E(G/e)| = |E(G)| - k$.

$\exists$ a natural bijection

$$E(G) \text{ without vertices between } u, v \rightarrow E(G/E)$$

Note that we actually can not store edges as sets of both endpoints because we can not distinguish between multiple edges. Hence, we must actuall add an indicator, which part of $x_{u,v}$, i.e., u or v, that edge orginated from. We can add a bit to indicate that, for instance.

As cuts have been defined as a set of edges, this induces a bijection:

$$\text{cuts in } G \text{ without } e \rightarrow \text{all cuts in } G/e$$

Lemma: $G = (V, E)$ is a multigraph and $e \in E$.

$$\mu(G/e) \geq \mu(G)$$

If G has a minimal cut C with $e \notin C$: $\mu(G/e) = \mu(G)$.

18.3 Randomized Edge Contraction

Now we construct an algorithm based on what we have learned about the problem. Ideally, we want to find some e so that μ remains unchanged. We do so by guessing.

```
Cut(G):
  G' = G
  WHILE |V(G')|>2:
    e = random edge from G', uniformly chosen
    G' = G/e
  RETURN size of the unique cut in G'
```

Note that edge contractions and randomly uniformly choosing an edge both work in $\mathcal{O}(n)$. So 'Cut(G)' has runtime $\mathcal{O}(n^2)$. But the result is not guaranteed to be correct.

Lemma: Let $G = (V, E)$ be a multigraph and $n = |V|$. If e is randomly uniformly chosen in E:

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

Proof. C :=minimal cut in G. $k := |C| = \mu(G)$.
We know $\forall v \in V, deg_G(v) \geq k$, so:

$$|E| = \frac{1}{2}\sum_{v\in V} deg_G(v) \geq \frac{kn}{2}$$

As $e \notin C \Rightarrow \mu(G/e) = \mu(G)$

$$\Pr[\mu(G) = \mu(G/e)] \geq \Pr[e \notin C] = 1 - \frac{|C|}{|E|} \geq 1 - \frac{k}{\frac{kn}{2}} = 1 - \frac{2}{n}$$

□

$$\hat{p}(G) := \text{probability that } Cut(G) \text{ returns the value } \mu(G)$$
$$\hat{p}(n) := \inf_{G=(V,E),|V|=n} \hat{p}(G)$$

Lemma: For all $n \geq 3$:

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

Proof. Let $G = (V, E), n := |V|$. So that 'Cut(G)' actually returns $\mu(G)$ two (random) events must take place:

- E_1 := event that $\mu(G) = \mu(G/e)$
- E_2 := event that 'Cut(G/e)' returns the value of $\mu(G/e)$

$$\hat{p}(G) = \Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2 | E_1] \geq (1 - \frac{2}{n}) \cdot \hat{p}(n-1)$$

The last inequality holds as E_2 is independent from E_1. And tha lower bound for $\Pr[E_1]$ follows from an above lemma. As this holds for any multigraph G, we get

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

$\square$

Lemma:

$$\hat{p}(n) \geq \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}$$

$\Rightarrow \mathbb{E}\left[\#\text{iterations till correct result}\right] \leq \binom{n}{2}$

Proof. With $\hat{p}(2) = 1$:

$$\hat{p}(n) \geq \frac{n-2}{n} \frac{n-3}{n-1} \frac{n-4}{n-2} \cdots \frac{3}{5} \frac{2}{4} \frac{1}{3} \cdot \hat{p}(2) = \frac{2}{n(n-1)}$$

$\square$

Our algorithm nur runs ‘Cut(G)‘ for $\lambda\binom{n}{2}$ times ($\lambda > 0$) and returns the smallest value it ever got.

Theorem:

- The algorithm has runtime $\mathcal{O}(\lambda n^4)$.
- The smallest return value of some instance of ‘Cut(G)‘ equals $\mu(G)$ with probability at least $1 - e^{-\lambda}$.

Proof.

- $\mathcal{O}(\lambda\binom{n}{2}n^2) = \mathcal{O}(\lambda n^4)$
- With $1 + x \leq e^x$:

$$(1 - \hat{p}(n))^{\lambda\binom{n}{1}} \leq (1 - \frac{1}{\binom{n}{2}})^{\lambda\binom{n}{2}} \leq e^{-\lambda}$$

$\square$

With $\lambda = \ln n$ we have runtime $\mathcal{O}(n^4 \log n)$ with error probability $\leq \frac{1}{n}$. We will make this approach worth it, i.e., reduce the runtime with bootstrapping.

18.4 Bootstrapping

Notice that the probability $\hat{p}(n)$ of getting a correct result is reliant on late steps. We had $\hat{p}(n) \geq \frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{3}{5}\frac{2}{4}\frac{1}{3}\cdot\hat{p}(2) = \frac{2}{n(n-1)}$, where the factors come from the individual iterations of the algorithm from left to right. When we do the last step better, we will get a 3x higher success probability. When we do the second to last step better, we will get a 2x increase. And so on.

One idea is to use some deterministic, well known algorithm to replace some number of last iterations with an $\mathcal{O}(z(t))$ computation to increase the probability significantly.

```
Cut1(G):
  WHILE $|V(G) > t|$:
    e = random edge from G, uniformly chosen
    G = G/e
  return size of the unique smallest cut in G
```

This changes the runtime to $\mathcal{O}(n(n-t)+z(t))$, where $z(t)$ is used some algorithm we use for the last steps. We define $p^*(t)$ to be some lower bound for the success probability.

Instead of choosing some expensive deterministic algorithm, we may also 'reuse' the probabilistic algorithm from the earlier section we are just now improving. That algorithm (with $\lambda = 1$) has runtime $z(t) = \mathcal{O}(t^4)$ and success probability $\geq 1 - e^{-1}$. We can an updated formula for $\hat{p}_t(n)$:

$$\hat{p}_t(n) \geq \frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{t+1}{t+3}\frac{t}{t+2}\frac{t-1}{t+1}\hat{p}_t(t) \geq \frac{t(t-1)}{n(n-1)}\frac{e-1}{e}$$

The inverse of that is the expected value for the number of iterations until we get the correct result. The runtime for one iteration is $\mathcal{O}(n(n-t)+z(t))$. We can multiply that with $\lambda\cdot$#expected number of iterations to get a success probability of $\geq 1-e^{-\lambda}$ as before. The runtime becomes:

$$\lambda\frac{n(n-1)}{t(t-1)}\mathcal{O}(n(n-t)+t^4) = \mathcal{O}(\lambda(\frac{n^4}{t^2}+n^2t^2))$$

We can now choose t so that this terms becomes minimal. We choose $t = \sqrt{n}$ and get $\mathcal{O}(\lambda n^3)$ as the new runtime.

One mustn't stop here but can do the same thing again, i.e., do bootstrapping again to further improve the runtime. When considering the limit of those improvements, one gets $\mathcal{O}(n^2 \text{ polylog}(n))$ (Karger & Stein).

19 Smallest Enclosing Disk

The smallest enclosing disk problem is part of a set of problems, which try to fit some set to a given form. This has applications in object/collision simulation, precision metrics in manufacturing from data/measure points, etc.

The enclosed disk problem ias associated with the enclosed ring, enclosed ellipse, enclosing lines, ... But the problems of enclosing triangle, enclosing rectangle, ... are quite different. The difference boils down to that the below algorithm is based on the fact that a circle is defined by at most 3 points. Meanwhile, a triangle may be defined an arbitrarily large amount of points.

Definition Smallest Enclosing Disk Problem: Given a finite set of points $P \subseteq \mathbb{R}^2$, determine the circle with the smallest radius, which encloses P.

Definition circle/disk: We have radius r and center z. The circle is $C = \{p \in \mathbb{R}^2 || zp| = r\}$, $|zp|$ being the distance of z and p.
$C^\bullet$ is the closed disk bordered by C: $C^\bullet = \{p \in \mathbb{R}^2 || zp| \leq r\}$.
C encloses P, because $P \subset C^\bullet$.

Lemma: For each finite set $P \subseteq \mathbb{R}^2$, a unique smallest enclosing circle $C(P)$ exists.

Proof. Existence was not covered. Uniqueness can be proven by contradiction. Assume there are two different smallest circles: C_1 and C_2, both with radius r and different centers $z_1 \neq z_2$. It must hold that $P \subseteq C_1^\bullet \cap C_2^\bullet$.

We now construct a new circle. Let it have the center $z = \frac{1}{2}(z_1 + z_2)$ (the middle between z_1 and z_2). Its radius $\hat{r}$ is the distance from z to the intersection point of C_1 and C_2. Basic trigonometry tells us that $\hat{r} = \sqrt{r^2 - \left(\frac{|z_1 z_2|}{2}\right)^2}$.

Notice that $P \subseteq C_1^\bullet \cap C_2^\bullet \subseteq C^\bullet$. So as $\hat{r} < r$, C_1 and C_2 can not be smallest enclosing circles, which is a contradiction. □

Lemma: For each set of points $P \subseteq \mathbb{R}^2, |P| \geq 3$, a subset $Q \subseteq P$ exists so that $|Q| = 3$ and $C(Q) = C(P)$.

Proof. We define that the set B contains all points which are on the border of $C^\bullet$, i.e., in C.

If g is an arbitrary line through the center of C, it can not be that ll points of P are strictly on one side of g. This holds as we could move the center of C orthogonally to g away from the side without points, and then slightly decrease the radius. We can do so in a way that still all points are contained, which contradicts the opposite statement.

From this follows that $|B| \geq 2$ and that if $|B| \leq 3$, C is given by $C(B)$. If $|B| \geq 4$, we choose two arbitrary points p, q from B which do not follow directly after each other when walking along C.

- If they are on one line, C is defined by p and q, which proves the lemma.
- If they are not, we can remove an arbitrary $r \in B$, which is on the shorter line segment defined by p and q. $B \backslash \{r\}$ then still valides our observation regarding some line g through the center and the lemma follows by induction.

□

We define our first native algorithm from those information.

```
CompleteEnumeration(P)
  FOR ALL $Q \in \binom{P}{3}$:
    determine C(Q)
    IF $P \subseteq C^\bullet(Q)$:
      RETURN $C(Q)$
```

For correctness: At least one value is returned as three points exist so that they form an enclosing circle. If we return a circle, we can not find another circle with smaller radius later. That is because if we would find a smaller circle, that smaller circle would have to contain the three points of the previous circle, which could only be covered with that larger circle. Hence, this smaller radius circle would have to contain that larger circle - being a contradiction.

However, we can easily improve this. Notice that all circles $C(Q)$ we consider have some radius. And only the circle with the largest radius can be the enclosing circle. All other cricles could not contain the three points that construct the larger circle. Hence, we actually only need to remember the radius. Thsi improves the runtime from $\mathcal{O}(n^4)$ to $\mathcal{O}(n^3)$.

```
CompleteEnumerationSmart(P):
  r = 0
  FOR ALL $Q \in \binom{P}{3}$:
    determine $C(Q)$
    IF $radius(C(Q)) > r$:
      $C^* = C(Q)$
      r = $radius(C(Q))$
  RETURN $C^*$
```

Now we can turn to randomize this algorithm by randomly picking Q. This initial algorithm 'only' provides $\mathcal{O}(n^4)$ (which is a step back so far) as the success probability for one Q is $\geq \frac{1}{\binom{n}{3}}$/the failure probability is $\leq \binom{n}{3}$.

```
RandomizedPrimitiveVersion(P):
  REPEAT FOREVER:
    choose $Q \subseteq P$ with $|Q| = 3$, uniformly randomly
    determine $C(Q)$
    IF $P \subseteq C^\bullet(Q)$:
      RETURN $C(Q)$
```

While running this algorithm we learn about points which lie outside of $C(Q)$. Our intuition tells us that those points are important while we can neglect points inside $C(Q)$. That intuition is only right in the sense that points outside are 'more important' but the ones on the insight can not be forgotten.

This algorithm makes use of this notion by increasing the weight of points outside of $C(Q)$. The remaining parts of the algorithm will get clear later.

```
RandomizedCleverVersion(P):
  $P' = P$ // P' being a multiset, i.e., P' can contain elements
      multiple times
  REPEAT FOREVER:
    choose $Q \subseteq P'$ with $|Q| = 11$, uniformly randomly
    determine $C(Q)$
    IF $P \subseteq C(Q)$:
      RETURN $C(Q)$
    ELSE:
      double all points in $P'$ outside of $C(Q)$
```

We must be able to choose Q in $\mathcal{O}(n)$ with $n := |P|$. Specifically, we must choose from a set of weighted options.

Lemma: $n_1, ..., n_t$ are natural numbers and $N := \sum_{i=1}^{t} n_i$. We generate $X \in \{1, ..., t\}$ randomly.

```
k = UniformInt(1,N)
x = 1
WHILE \sum_{i=1}^{x}n_{i}<k:
  x = x+1
RETURN x
```

Then $\Pr[X = i] = \frac{n_i}{N}$ for all $i = 1, ..., t$

Proof. The probability follows from the fact that $x = i$ is outputted exactly for $k \in \{1 + \sum_{i=1}^{k-1} n_i, ..., \sum_{i=1}^{x} n_i\}$, i.e., for n_i of all possible N values of k. □

We now consider the choice of $|Q| = 11$ instead of $= 3$. We will see that any value larger than 11 works, but choosing the smallest value keeps the (quite expensive) constant work low. The reason lies in the expected number of iterations until $P \subseteq C^\bullet(Q)$.

Notice that some $B^* \subseteq P$ with $C(B^*) = C(P)$ and $|B^*| \leq 3$ must exists. Based on that definition of B^* it must hold

$$B^* \subseteq Q \subseteq P \Rightarrow C(Q) = C(P)$$

Some additional observations:

- $|P'|$ grows rapidly

  As long as $P \subseteq C^\bullet(Q)$ does not hold, some element of B^* must lie outside of $C(Q)$. This means that some point in B^* must have been doubled at least $\frac{k}{3}$ times and, hence, have multiplicity $\geq 2^{\frac{k}{3}}$.

- $|P'|$ does not grow too rapidly

  Below we will show that we expect (emphasis on probabilistic here) P' to only increases by a factor of $(1 + \frac{3}{12}) = \frac{4}{5}$ in each iteration. So, $|P'| \leq \frac{5}{4}^k n$.

- As $\frac{5}{4} \leq \sqrt[3]{2}$, those bounds are to cross at some points and we expect (again, only probabilistically) the algorithm to terminate hence. This is no formal reasoning (yet). Now follows the formal consideration.

Lemma: $r, N \in \mathbb{N}, r \leq N$, and $P' \subseteq \mathbb{R}^2$ be a multiset with $|P'| = N$. If R is uniformly randomly chosen from $\binom{P'}{r}$, we get:

$$\mathbb{E}[|P' \backslash C^\bullet(R)|] \leq 3\frac{N - r}{r + 1} \leq 3\frac{N}{r + 1}$$

Proof. $p \in P', R, Q \subseteq P'$.

$$out(p, R) := \begin{cases} 1, & p \notin C^\bullet(R) \\ 0, & \text{otherwise} \end{cases}$$

$$ess(p, Q) := \begin{cases} 1, & C(Q \backslash \{p\}) \neq C(Q) \\ 0, & \text{otherwise} \end{cases}$$

- $\sum_{p \in P' \setminus R} out(p, R) = |P' \setminus C^{\bullet}(R)|$
- $\sum_{p \in Q} ess(p, Q) \leq$
- $out(p, R) = 1 \Leftrightarrow ess(p, R \cup \{p\}) = 1$

We want to compute $\mathbb{E}\left[|P' \setminus C^{\bullet}(R)|\right]$.

$$\begin{aligned}
\mathbb{E}[|P' \setminus C^{\bullet}(R)|] &= \frac{1}{\binom{N}{r}} \sum_{R \in \binom{P'}{r}} \sum_{s \in P' \setminus R} out(s, R) \\
&= \frac{1}{\binom{N}{r}} \sum_{R \in \binom{P'}{r}} \sum_{s \in P' \setminus R} ess(s, R \cup \{s\}) \\
&= \frac{1}{\binom{N}{r}} \sum_{Q \in \binom{P'}{r+1}} \sum_{p \in Q} ess(p, Q) \\
&\leq \frac{1}{\binom{N}{r}} \sum_{Q \in \binom{P'}{r+1}} 3 \\
&= 3 \cdot \frac{\binom{N}{r+1}}{\binom{N}{r}} \\
&= 3 \frac{N - r}{r + 1}
\end{aligned}$$

□

We can now write $3\frac{N}{r+1} \leq 3\frac{N}{r+1} + N = (1 + \frac{3}{r+1})N$

Actually, we are interested not in the increase but actually in the size of $|P'|$ itself. Let $T \in \mathbb{N} \cup \{\infty\}$ be the count of iterations of the algorithm and $X_k := |P'|$ after $\min\{T, k\}$ iterations.

$$\begin{aligned}
\mathbb{E}[X_k] &= \sum_{t=0}^{\infty} \mathbb{E}[X_k | X_{k-1} = t] \cdot \Pr[X_{k-1} = t] \\
&\leq \sum_{t=0}^{\infty} (1 + \frac{3}{r+1}) t \cdot \Pr[X_{k-1} = t] && \text{(this uses the lemma result)} \\
&= (1 + \frac{3}{r+1}) \sum_{t=0}^{\infty} t \cdot \Pr[X_{k-1} = t] \\
&= \left(1 + \frac{3}{r+1}\right) \cdot \mathbb{E}[X_{k-1}]
\end{aligned}$$

With $X_0 \equiv n$ we can use induction and get $\mathbb{E}\left[X_k\right] \leq (1 + \frac{3}{r+1})^k \cdot n$.

From that follows the factor $(1 + \frac{3}{12})$ of the above bound with $r = 11$ as used in the algorithm. But still, the choice of $r = 11$ remains mysterious...

We have stated that for $T \geq k : X_k \geq \sqrt[3]{2}^k$.

$$\mathbb{E}[X_k] = \mathbb{E}[X_k | T \geq k] \cdot \Pr[T \geq k] + \mathbb{E}[X_k | T < k] \cdot \Pr[T < k] \geq 2^{\frac{k}{3}} \cdot \Pr[T \geq k]$$

This works as removing one summand just decreases the bound (and $\mathbb{E}\left[X_k | T < k\right] = 0$ by assumption after all).

When combining those two bounds, we get

$$\sqrt[3]{2}^k \cdot \Pr[T \geq k] \leq \mathbb{E}[X_k] \leq (\frac{5}{4})^k \cdot n$$
$$\Leftrightarrow \Pr[T \geq k] \leq (\frac{5}{4\sqrt[3]{2}})^k \cdot n \leq 0.993^k n$$
$$\text{for } k \geq -\log_{0.993} n \text{ we then even have } \leq 1$$

This justified the minimum of $r = 11$. If r would have been chosen larger, then the factor of the exponential term would have been ≥ 1, which means that we would have no useful upper bound here. But that bound is required to give some useful expected runtime.

Theorem: The algorithm computes the smallest enclosing disc with the expected runtime $\mathcal{O}(n \log n)$.

Proof. We choose $k_0 := \lceil -\log_{0.993} n \rceil$. For the expected number of iterations we have:

$$\mathbb{E}[T] = \sum_{k \geq 1} \Pr[T \geq k] \leq \sum_{k=1}^{k_0} 1 + \sum_{k \geq k_0} 0.993^k n$$
$$= \sum_{k=1}^{k_0} 1 + \sum_{k' \geq 0} 0.993^{k'} \cdot 0.993^{k_0+1} n = k_0 + \mathcal{O}(1) = \mathcal{O}(\log n)$$

Because there are $\mathcal{O}(n)$ iterations, we get our runtime. $\square$

The algorithm introduced above is based on an idea from Clarkson ('95) and works for the smallest enclosing sphere (or ellipse) in any dimension with respectively different constants compared to 11. Those constants usually grow exponentially with the dimension. But within some dimension we always have $\mathcal{O}(n \log n)$.

Also, there are simpler randomized and also deterministic algorithms (with linear runtime). But those are then fixed for one dimension.

19.1 Sampling Lemma

Above we discussed $\mathbb{E}\left[|P' \backslash C^{\bullet}(R)|\right] \leq 3\frac{N-r}{r+1} \leq 3\frac{N}{r+1}$. That lemma can be generalized and is then called sampling lemma.

Definition: Given some finite set S, $n := |S|$ and ϕ being an arbitrary function on 2^S to some arbitrary codomain. We define:

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

Elements in $V(R)$ are called "Verletzer" of R. Elements in $X(R)$ are called "extrem" in R.

We see that $s \in V(R) \Leftrightarrow s \in X(R \cup \{s\})$.

The sampling lemma relates the expected amount of "Verletzern" to the expected amount of "extrem" elements.

Lemma Sampling Lemma: Let $k \in \mathbb{N}, 0 \leq k \leq n$. R is a k-element subset of S, uniformly randomly from $\binom{S}{k}$. We set $v_k := \mathbb{E}[|V(R)|]$ and $x_k := \mathbb{E}[|X(R)|]$.
Then, for $r \in \mathbb{N}, 0 \leq r \leq n$:

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

Proof. $G = (A \uplus B, E)$ bipartite: average degree in $A \cdot |A| = |E|$.
First, we defien a bipartite graph.

- vertices: $\binom{S}{r} \uplus \binom{S}{r+1}$
- edges: $\{R, R \cup \{s\}\}$, $R \in \binom{S}{r}$, s "Verletzer" of R

Degree of $R \in \binom{S}{r}$ is $|V(R)|$. Number of edges is $\binom{n}{r} v_r$.

The edges can just as well be described as $\{Q \backslash \{s\}, Q\}$ with $Q \in \binom{S}{r+1}$ and s being "extrem" in Q. The degree of Q is accordingly $|X(Q)|$ and the number of edges are $\binom{S}{r+1} x_{r+1}$.

We have effectively shown that $\binom{n}{r} v_r = \binom{n}{r+1} x_r$ from which follows the claim. □

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

20 Convex Hull

The convex hull (defined below) can be computed in various dimensions. We will consider dimension $d = 2$ here but $d \in \mathbb{N}$ could be arbitrary.

Definition line segment: For $v_1, v_1 \in \mathbb{R}^d$

$$\overline{v_0 v_1} := \{(1-\lambda)v_0 + \lambda v_1 | \lambda \in \mathbb{R}, 0 \leq \lambda \leq 1\}$$

is the line segment, which connects v_0 and v_1.

Definition convexity/convex sets: Some set $C \subseteq \mathbb{R}^d$ is called convex if $\forall v_0, v_1 \in C$: $\overline{v_0 v_1} \subseteq C$.

Definition convex hull: The convex hull conv(S) of some set $S \subseteq \mathbb{R}^d$ is the cut of all convex sets, which contain S.

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

The convex hull is always a convex set itself because the cut of two convex sets is also a convex set. Intuitively, the convex hull can also be understood as the smallest convex set which contains S.

Definition Convex Hull Problem: Given some finite set of points $P \subseteq \mathbb{R}^2$, determine the convex hull of P.

Generally, some convex set can not stored/characterized with a finite amount of data, it is an infinite set after all.

However, for some finite set of points P in some plane, the convex hull is defined by some polygon, the edge of $conv(P)$, whose corners are edges from P. So, when we discuss computing $conv(S)$, we mean to determine a sequence of the corners of the polygon beginning at some q_0 and considering other edges counter-clockwise along the polygon.

$$(q_0, q_1, ..., q_{h-1}), h \leq n$$

$Q := \{q_0, q_1, ..., q_{h-1}\} \subseteq P$ then is the smallest subset of P with $\text{conv}(Q) = \text{conv}(P)$. This allows us to rephrase the problem.

Definition Convex Hull Problem, rephrased: Given some finite set of points $P \subseteq \mathbb{R}^2$, determine the corners of the enclosing polygon $\text{conv}(P)$ in counter-clockwise order.

We will now consider two different algorithms for this problem.

For those algorithms, we will at the beginning usually make the assumptions that all points in S are in 'general location' ("allgemeiner Lage"). This means that no three points are on the same line and that no two points have the same x coordinate.

20.1 Algorithm: Jarvis Wrap

We start by assuming that all points are in general location.

Definition boundary edge: A pair $qr \in P^2, q \neq r$, is called boundary edge of P if all points in $P \backslash \{q, r\}$ are left from qr/the directed edge from q to r.

Lemma:
Let $p = (p_x, p_y), q = (q_x, q_y)$, and $r = (r_x, r_y)$ be points in $\mathbb{R}^2$. We have $q \neq r$ and p being left from (the directed) qr if and only if

$$\mathsf{det}\ (p, q, r) = \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \left|\begin{matrix} q_x - p_x & q_y - p_y \\ r_x - p_x & r_y - p_y \end{matrix} \quad 1\right| > 0$$

That is equivalent to

$$(q_x - p_x)(r_y - p_y) > (q_y - p_y)(r_x - p_x)$$

Proof. No proof is given. But this follows from what has been learned about the determinant in Linear Algebra.

Intuitively, the determinant computes the area spanned by the three vectors p, q, r. The sign of the determinant/the area depens on the orientation, i.e., order of the provided vectors. Hence, for a certain order, we get a positive result if the p is to the left of qr.

Specifically, $\frac{1}{2}|\mathsf{det}\ (p, q, r)|$ is the area of the triangle pqr, where the sign determines the order of the corners in which we traverse the edges. $\square$

Lemma:
$(q_0, q_1, ..., q_{h-1})$ is the corner sequence of the $conv(P)$ enclosing polygon in counter-clockwise order if and only if all pairs $(q_{i-1}, q_i), i = 1, 2, ..., h(\mod h)$ are boundary edges of P.

From the above, we can construct the first native algorithm. We iterate over all $n(n-1)$ pairs qr and check whether it is a boundary edge by checking for all remaining $n-2$ points p in $P \backslash \{q, r\}$, whether p is to the left of qr. By doing so, we can find all boundary edges in $\mathcal{O}(n^3)$, which we then only need to order, which can be definitely done in $\mathcal{O}(n^2)$.

But that algorithm is most basic and we can do better. So, let $q_0 :=$ point with smallest x-coordinate. q_0 then certainly is a corner of the convex hull.

```
FindNext(q):
  choose $p_0 \in P\{q\}$ randomly
  $q_{next}$ = $p_0$
  FOR ALL $p \in P \backslash \{q, p_0\}$:
    IF $p$ right of $q_{next}$:
      $q_{next}$ = $p$
  return $q_{next}$
```

This works because for some $q \in P$, we can define a relation on the remaining points in P, which is a total order relation if q is a boundary edge. Given such $q \in P$ we define

$$p_1 \prec_q p_2 :\Leftrightarrow p_1 \text{ right of } qp_2$$

From q begin a corner of the convex hull we get that some line exists, which separates q and $P \backslash \{q\}$. It implies that the order is not cyclic but actually a total order so that some minimum exists.

Lemma: If q is a corner of the convex hull of P, the relation $\prec_q$ is a total order on $P \backslash \{q\}$. For the minimum p_{min} of that order, qp_{min} is a boundary edge.

So, we can use the above 'FindNext()' function to define our algorithm.

```
JarvinWrap(P):
  h = 0
  $p_{now}$ = point in P with smallest x-coordinate
  REPEAT:
    $q_h$ = $p_{now}$
    $p_{now}$ = FindNext($q_h$)
    h = h+1
  UNTIL $p_{now}$ = $q_0$
  RETURN ($q_0$,$q_1$,...,$q_{h-1}$)
```

Theorem: For a set P of n points in general location in $\mathbb{R}^2$, the JarvisWrap algorithm computes the convex hull in $\mathcal{O}(nh)$ with h being the number of corners of the convex hull of P.

With $h \leq n$ we get $\mathcal{O}(n^2)$, which is better than the naive $\mathcal{O}(n^3)$. If $h = \mathcal{O}(1)$, which is the case if we know the number of edges in advance, then we have $\mathcal{O}(n)$. This thought process is quite interesting as for different geometric forms, i.e., points being randomly inside such a form, we have a certain number of expected points.

- square: #edges $= \mathcal{O}(\log n)$

- circle: #edges $= \mathcal{O}(\sqrt[3]{n})$

Now we discuss some limitations (such as general location) we introduced earlier.

- unique x-coordinates

  The purpose of this was to be able to easily choose some initial vertex for our algorithm to start. But we can also choose q_0 as the smallest point in regard to the lexicographic order (y-coordinate as second part) to allow duplicate x-coordinates.

- no three points on one line

  The purpose of this was so that we can easily test for "p to the right of qq_{next}". But we can also replace that relation with the smallest in the lexicographic order with $|qp| > |qq_{next}|$ being the second component.

- we considered a set of unique points

  If we get the input points as an array, this may not be guaranteed. One option is to remove duplicates in the beginning in $\Theta(n \log n)$, which would increase the linear component. Instead it is sufficient to (a) in 'FindNext()' test $p_0 \neq q$ when choosing p_0 and (b) to do the comparison $p_{now} = 1_0$ in 'JarvisWrap()' based on the lexicographic order/actual points instead of indices (or just remove duplicates only of q_0 only in the beginning).

Furthermore, this algorithm has numeric difficulties. To determine whether some point if left of a line, we compute $(q_x - p_x)(r_y - p_y) > (q_y - p_y)(r_x - p_x)$. With floating pointers this may not be precise. Just imprecise results are unfortunate but not that bad. What is definitely bad if computational errors lead to wrong results or an infinite loop in the algorithm. To solve those instabilities we can use libraries which offer precise datatypes for the operations we require. While that works here, it definitely reduces efficiency.

20.2 Algorithm: Local Improvements

We now consider a second algorithm and start by considering S as a set of points in general location again.

When remembering the definition of a boundary edge, we can check local convexity of some polygon $(q_0, q_1, ..., q_{h-1})$ by checking

$$\forall i, 1 \leq i \leq h, q_{i+1} \text{ left from } q_{i-1}q_i, \text{indices } i \mod h$$

While this checks local convexity, there are some deficits in regard to global convexity:

- $\{q_0, ..., q_{h-1}\}$ must not be a subset of P.
- The polygon must not contain all points of P.
- The polygon may cross itself (violating global convexity).

Observe that if those three deficits would not hold and we would have local convexity, then we would also have global convexity. The idea for our algorithm is hence to start with some polygon which not necessarily satisfies local convexity but avoids the three mentioned deficits. Then we we do local improvements so that in the end local convexity holds without introducing any defect on the way. In the end, we then have global convexity.

Given $(q_0, ..., q_{k-1})$, a local improvement means that if q_1 is left from $q_{i-1}q_{i+1}$ (indices mod k), we remove q_i from the sequence.

We get the initial polygon by sorting P by the x-coordinate in increasing order. From that we get the point order $(p_1, ..., p_n)$ and consider the polygon $(p_1, p_2, ..., p_{n-1}, p_n, p_{n-1}, ..., p_3, p_2)$. This polygon contains only points of P and does not cross itself. It avoids all defects.

For correctness of the algorithm we now must consider some invariants, which show that we successfully avoid the defects during local improvement steps. First notice that p_1 and p_n may never be removed from the polygon as they have minimum and maximum x-coordinate.

- The partial polygon trail $(p_1, ..., p_n)$ is x-monotone (left-to-right) and has no point in P below it.

  A local improvement step may only remove elements, which can not lead to a violation of x monotony.

  A local improvement step may only push the boundary to the right, which means bottom here as we consider left-to-right monotony. If all points have been above before, then they must also be above afterwards.

- The partial polygon trail $(p_n, ..., p_1)$ is x-monotone (right-to-left) and has no point in P above it.

  A local improvement step may only remove elements, which can not lead to a violation of x monotony.

  A local improvement step may only push the boundary to the right, which means top here as we consider right-to-left monotony. If all points have been below before, then they must also be below afterwards.

- The partial polygon trail $(p_1, ..., p_n)$ is never above the partial polygon trail $(p_n, ..., p_1)$.

  As reasoned for the individual trails, $(p_1, ..., p_n)$ may only move down while $(p_n, ..., p_1)$ may only move up. So if the claim held before it will also hold afterwards

These invariants imply that the polygon can never cross itself and contains all points. We also easily see that the points of the polygon may only be some subset of P.

So doing local improvements always avoids the above mentioned defects. If we manage to do local improvements until the polygon is locally convex, we then must also have the globally convex solution polygon.

The above already fully specifies a working algorithm. So far it is non-deterministic as we did not specify an order for the local improvements. But to implement the algorithm we have to do so now.

```
LocalRepair($p_1$, $p_2$, ..., $p_n$): // input is sorted by x-coordinate
  $q_0$ = $p_1$;
  h = 0;

  FOR i=2,...,n: // this improves the lower partial polygon trail
    WHILE $h > 0$ AND $q_h$ left from $q_{h-1}p_i$:
      h = h-1
    h = h+1
    $q_h$ = $p_i$

  h' = h

  FOR i=n-1,...,1: // this improves the upper partial polygon trail
    WHILE $h > h'$ AND $q_h$ left from $q_{h-1}p_i$
```

```
        h = h-1
      h = h+1
      $q_h$ = $p_i$

  RETURN ($q_0$,$q_1$,...$q_{h-1}$)
```

We now do the runtime analysis. Notice that the fact that we introduced some specific order for the local improvement steps does not matter. Our initial polygon has $2(n-1)$ corners. Our solution polygon has h corners. Each local improvement removes one corner. Hence, we must remove $2(n-1)-h = \mathcal{O}(n)$ times.

So there are $\mathcal{O}(n)$ successful tests of "q_h left from $q_{h-1}p_i$" and for each p_i two failing tests (one on the upper partial polygon trail and one on the lower partial polygon trail). Hence, each while block has a runtime of $\mathcal{O}(n)$ accordd all for iterations. So the total algorithm with some sorted input has runtime $\mathcal{O}(n)$. And as sorting takes $\mathcal{O}(n \log n)$ in the beginning, we have a total runtime of $\mathcal{O}(n \log n)$.

Theorem: Given s sequence of points $p_1, p_2, ..., p_n$ sorted by their x-coordinate in general location in $\mathbb{R}^2$, the 'LocalRepair' algorithm compute the convex hull of $\{p_1, p_2, ..., p_n\}$ in $\mathcal{O}(n)$.

Now some final considerations.

- We again assumed in the beginning that all points of S are in general location. But that is not necessary.
  - Instead of prohibiting three points to be on the same line we simply consider the lexicographic order of points P (the y-coordinate being the second component) and remove duplicates from the order.

    That this works can be understood when considering that we can slightly rotate the whole set of points. We can do that so slightly that no order is changed and the convex hull then clearly is identical (except for being rotated). And the resulting order after this slight rotation is exactly the order of the lexicographic order.
  - Instead of prohibiting any three points to be on one line we can adapt the test "q_h left from $q_{h-1}p_i$". We would then also remove q_h if it is not left from but also on the line.
- This algorithm is numerically more robust than JarvisWrap. We might still have some slight errors (wrongly removing or taking some point which is very close to the actual boundary) caused by numerical inaccuracies. However, this algorithm will never run in an infinite loop or return a fundamentally wrong result because its runtime is bound by iterating over the points of the set P.
- The connections crated during local improvements result in a triangulation of the points in the end. With local improvements upon this triangulation one can also get a good triangulation (Delaunay Triangulation). This was only mentioned but not further discussed.
- We have said that the LocalRepair algorithm is optimal. Anyway, there are algorithms, which offer slightly better bounds in some cases but still have $\mathcal{O}(n \log n)$ for the worst case.
  - Some $\mathcal{O}(n \log h)$ algorithm exists.

– If the points for S are chosen from a square or circle, some algorithm with expected runtime $\mathcal{O}(n)$ exists.

20.3 Lower Bound

We can quite easily show a lower bound for the runtime of the convex hull problem by using a reduction.

Consider the sequence $(x_1, x_2, ..., x_n)$ of numbers in $\mathbb{R}$. We set $p_i = (x_i, x_i^2), i = 1, 2, ..., n$ (vertical projection of the x-axis on to the unit parabola in $\mathbb{R}^2$). From the sequence of corners of the convex hull of $P := \{p_1, p_2, ..., p_n\}$ we could, in linear time, also get the order of x_i in increasing order.

If we can solve convex hull in $t(n)$, we can sort in $t(n) + \mathcal{O}(n)$. And we we have the lower bound $\mathcal{O}(n \log n)$ for sorting, we can compute the convex hull in at most $\mathcal{O}(n \log n)$. This shows that the local improvements algorithm has optimal runtime.

Part IV
Optional

In the lecture, some additional topics were discussed, which are not relevant for examination and, hence, not covered here. The topics/algorithms were:

- Convex Sets
- Continuous Matching
- Expert Algorithm