AAD Notes

Magnus Goltermann (xzb187)

Januay 2024

Indhold

1 Max Flow

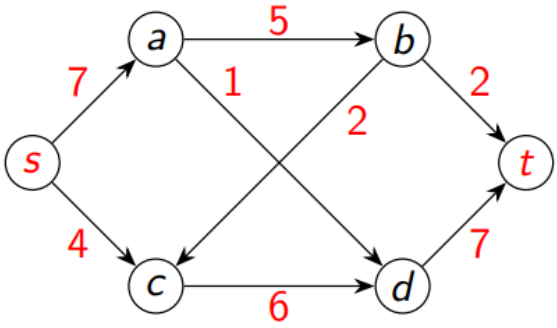
1.1 Flow network definition

A directed graph called a flow network G with vertices V and edges E, such that $G = (V, E)$. The flow network contains a sink $s \in V$ and a sink $t \in V \backslash \{s\}$, where we flow from s to t. To denote how much can be

flowed on each edge we have a capacity function $c : V \times V \to \mathbb{R}$. Thus

$$\begin{aligned} G = (V, E) \quad &: \text{Flow network with vertices } V \text{ and edges } E \\ s \in V \quad &: \text{The source} \\ t \in V \backslash \{s\} \quad &: \text{The sink} \\ c : V \times V \to \mathbb{R} \quad &: \text{The capacity function} \end{aligned}$$

A flow network has no self-loops and no antiparallel edges (can easily be allowed for by modifying the graph before doing any actual flows by adding an extra vertex in-between). An example of a flow network can be seen below in figure 1.



Figur 1: Example of a flow network

1.1.1 Capacity function constraints

The capacity function has two constraints:

- No negative capacities: $c(u, v) \geq 0$ for all $u, v \in V$
- If there is not an edge from u to v then the capacity is 0: if $(u, v) \notin E$ then $c(u, v) = 0$

1.2 Flow definition

A flow in a flow network (G, s, t, c) is a function $f : V \times V \to \mathbb{R}$ that satisfies:

1. Capacity conservation: $\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$.
2. Flow conservation: $\forall v \in V \backslash \{s, t\} : \sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$.

   Equivalently: $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w)$ (as we can just look at the edges that actually connect vertices.)

The capacity constraint ensures that the flow of any edge never exceeds its capacity. And the flow conservation makes sure that what flows into a vertex also comes out (except for the source and sink), and thus the sum of all flows into a vertex is the same as out of a vertex always.

1.2.1 Value of a flow

The value of a flow $|f|$ is given as

$$|f| := \sum_{v \in V} f(s, v) - \sum_{u \in V} f(u, s) = \sum_{v \in V} (f(s, v) - f(v, s)) \tag{1}$$

Intuitively it can be found by just counting everything that either goes out of the source or into the sink, as we have flow conservation (everything that leaves the source must enter the sink).

1.3 Max Flow definition

A max flow is the **maximum flow value** that can exist in a flow network.

1.4 Ford-Fulkerson Method

Given the min cut theorem (section 1.10), if the method terminates it will give a max flow since there are no more augmenting paths, which the theorem says is equivalent to the flow being a max flow (point 2).

1.4.1 Ford-Fulkerson Method informal

The Ford-Fulkerson method is a basic approach to finding a max flow.

```
function FORD-FULKERSON(G = (V, E), s, t, c)
    f ← 0
    while ∃ "augmenting path" p from s to t do
        Send as much flow as possible along p and "add" this to f.
    return f
```

Figur 2: The Ford-Fulkerson method psoudocode of the informal method.

The method simply takes an augmenting path p, add flow to that, and keep going until all paths are "full". Informal since it does not specify how to find the augmenting path.

1.4.2 Ford-Fulkerson Method

A more formalized method where augmenting paths are defined as paths in the residual network G_f from the source to the sink, as that are all the possible "actions"we can do.

```
function FORD-FULKERSON(G = (V, E), s, t, c)
    f ← 0
    while ∃ (augmenting) path p from s to t in G_f do
        Find a max flow f_p along p in G_f.
        f ← f ↑ f_p
    return f
```

Figur 3: Ford Fulkersons method

1.4.3 Worst case analysis

In general the method is not guaranteed to terminate, but requries some capacities to be irrational.
Assuming integer capacities, the running time is The time complexity is $\mathcal{O}(E \cdot |f^*|)$ where f^* is a max flow. It is proportional with the max flow value, since it adds at least 1 to the flow each iteration (since they are integers).

1.5 Edmonds-Karp Algorithm

The algorithm follows the idea of the Ford-Fulkersons method, but always picks the shortest augmenting path

```
1: function EDMONDS-KARP(G = (V, E), s, t, c)
2:     f ← 0
3:     while ∃ (augmenting) path from s to t in G_f do
4:         p ← shortest such path.
5:         Find a max flow f_p along p in G_f.
6:         f ← f ↑ f_p
7:     return f
```

Figur 4: Edmonds-Karp Algorithm

The number of iterations of Edmonds-Karp is $\mathcal{O}(V \cdot E)$, and thus it can be implemented to run in $\mathcal{O}(V \cdot E^2)$ (using e.g. breadth-first search to find shortest paths).
The proof for the number of iterations is based on the shortest distance from the source to the sink (of the augmenting paths) keeps increasing, and will at most be the same distance for $\mathcal{O}(E)$ iteration, and thus we have $\mathcal{O}(V)$ runs with at most $\mathcal{O}(E)$ iterations, giving the max number of iterations to be $\mathcal{O}(V \cdot E)$.

The proof consists of 2 claims (which I will not prove):
Claim 1: For $i = 0, \ldots, k$, Edmonds-Karp finds an augmenting path in G_f, consisting only of edges that are forward edges in G_{f_0}[1].
Claim 2: If there is an augmenting path in $G_{f_{k+1}}$, then

$$\delta_{f_{k+1}}(s,t) \geq \delta_{f_k}(s,t).$$

Claim 1 and Claim 2 together gives the claimed #iterations for Edmonds-Karp.

1.6 Integrality Theorem

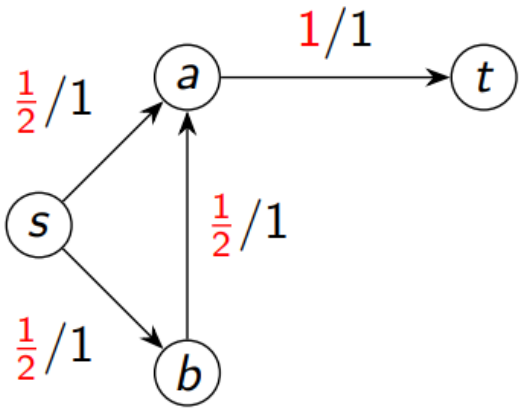
Given integer capacities, Ford-Fulkerson (and therefore Edmonds-Karp) will find an integer-valued flow

$$f : V \times V \to \mathbb{Z}_{\geq 0}$$

with

$$|f| \in \mathbb{Z}_{\geq 0}$$

being an integer.
Note: Not all max flows in a network with integer capacities have to be integer-valued. See example below in figure 5.



Figur 5: Example of non-integer value flows for integer capacities.

[1]Note that Claim 1 implies $k = \mathcal{O}(E)$, since at least one forward edge gets saturated in iteration i, which removes it from $G_{f_{i+1}}$.

1.7 Residuals in a flow

1.7.1 Residual capacity

Given a flow f in (G, s, t, c), the residual capacity is the function $c_f : V \times V \to \mathbb{R}$ defined by

$$c_f(u,v) := \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \text{ (i.e. how much more could be sent)} \\ f(v,u) & \text{if } (v,u) \in E \text{ (i.e. how much can be cancelled)} \\ 0 & \text{otherwise} \end{cases}$$

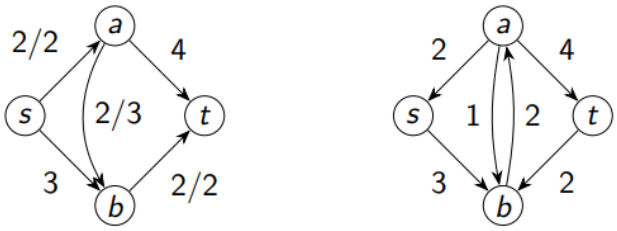
Thus it is how we can change the flow on an edge. If there is more capacity than what we are flowing, then we can flow more in the direction from u to v. If we are already flowing from u to v, then we can "send"flow backwards by canceling some flow. Note: no self-loops or anti-parallel edges for the definition to make sense.

1.7.2 Residual network

A residual network is the collection of all the residual capacities with a positive value, and thus all the "action"within the flow network we can try to edit to improve the flow value. It is defined as the graph $G_f := (V, E_f)$ where

$$E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$$

together with s, t, and the capacity function c_f. A Residual network can have parallel edges if we e.g. flow $2/3$ from a to b, we still have 1 more from a to b, and 2 from b to a.



Figur 6: Example of a flow (left) with the corresponding residual network (right)

1.8 Augmented flow $f \uparrow f'$

Given a flow f in G and a flow f' in the residual network G_f, the augmented flow is $f \uparrow f' : V \times V \to \mathbb{R}$ is

$$(f \uparrow f')(u,v) := \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Thus it is the amount of flow f already there plus the added flow from the residual network f' minus the flow from f' that we flow the other way. It's the "math"way of editing a flow, such that it conserves both the capacity constraint and the flow conservation.

1.8.1 Lemma: Augmented flow is a flow in G

The lemma:

$$f \uparrow f' \text{ is a flow in } G \text{ of value } |f \uparrow f'| = |f| + |f'| \tag{2}$$

which states that the augmented flow $f \uparrow f'$ is also a valid flow in G, with a flow value of $|f| + |f'|$

Proof

To prove the lemma, we have to prove the flow conservation and the capacity constraints are kept.

Capacity Constraint:

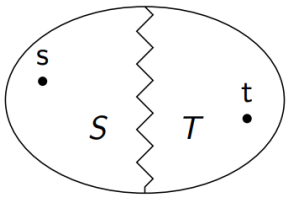
$$\begin{aligned}
&\text{Let } (u,v) \in E \text{ (otherwise it is trivial), then} \\
(f \uparrow f')(u,v) &= f(u,v) + f'(u,v) - f'(v,u) \\
&\leq f(u,v) + c_f(u,v) - 0 \\
&= f(u,v) + c(u,v) - f(u,v) = c(u,v) \\
(f \uparrow f')(u,v) &= f(u,v) + f'(u,v) - f'(v,u) \\
&\geq f(u,v) + 0 - c_f(v,u) \\
&= f(u,v) + 0 - f(u,v) = 0
\end{aligned}$$

Flow Convervation:
We want to show flow conservation in u:
Let $u \in V \backslash \{s,t\}$ Then the total amount of flow leaving u subtracted with flow entering u must be 0:

$$\begin{aligned}
&\sum_{(u,v)\in E} (f \uparrow f')(u,v) - \sum_{(v,u)\in E} (f \uparrow f')(u,v) \\
&= \sum_{(u,v)\in E} \big(f(u,v) + f'(u,v) - f'(v,u)\big) - \sum_{(v,u)\in E} \big(f(v,u) + f'(v,u) - f'(u,v)\big) \\
&= \sum_{(u,v)\in E} \big(f'(u,v) - f'(v,u)\big) - \sum_{(v,u)\in E} \big(f'(v,u) - f'(u,v)\big) \\
&= 0
\end{aligned}$$

1.9 Cut

A cut is a partition of V into subsets $S \ni s$ and $T \ni t$.



Figur 7: Illustration of a cut

Thus a cut divides the vertices into two parts, where the source and sink are each in their own subset.

1.9.1 Net flow / flow across

Given a flow f and a cut (S,T) we define the net flow across (S,T) as

$$f(S,T) := \sum_{u\in S} \sum_{v\in T} (f(u,v) - f(v,u)) \tag{3}$$

Thus it is the flow across a cut, but taking the sum of all flows in minus all the flows out of the subset.

1.9.2 Capacity of a subset

Given a cut (S,T) we define the capacity of (S,T) as

$$c(S,T) := \sum_{u\in S} \sum_{v\in T} c(u,v) \tag{4}$$

Thus the sum of all possible flows out of a subset.

1.9.3 Lemma: Flow across cut is equal to flow value

Given a flow f in G, for all cuts (S,T) we have $f(S,T) = |f|$.

The net flow across the cut is equal to the value of the flow. Thus any way of cutting a flow, the flow across that cut is equal to the flow out of the source, and thus the flow value.

Proof

$$
\begin{aligned}
\mathbf{1} \quad f(S,T) &= \sum_{u \in S} \sum_{v \in T} (f(u,v) - f(v,u)) \\
\mathbf{2} \quad &= \sum_{u \in S} \sum_{v \in S} (f(u,v) - f(v,u)) + \sum_{u \in S} \sum_{v \in T} (f(u,v) - f(v,u)) \\
\mathbf{3} \quad &= \sum_{u \in S} \sum_{v \in V} (f(u,v) - f(v,u)) \\
\mathbf{4} \quad &= \sum_{u \in \{s\}} \sum_{v \in V} (f(u,v) - f(v,u)) + \sum_{u \in S \setminus \{s\}} \sum_{v \in V} (f(u,v) - f(v,u)) \\
\mathbf{5} \quad &= \sum_{v \in V} (f(s,v) - f(v,s)) + 0 \\
\mathbf{6} \quad &=: |f|
\end{aligned}
\tag{5}
$$

Explanation

1. The definition from (3).
2. We add the double sum of taking the flows within subset S, but since that will add and subtract all the same flows, it is equivalent to adding 0.
3. Since the subsets S and T combined give all vertices V, we rewrite to just take the flow from all u in S to all v in V (thus the net flow going out of S).
4. The only u that contributes to the sum is the u from the source (as all other flows cancel out due to flow conservation).
5. All u's not in the source cancel themselves out.
6. This equals the flow value from (1).

Corollary: The flow value is at most the capacity of the cut

For any flow f and any cut $(S,T), |f| \leq c(S,T)$. Which intuitively means we can look at any cut, and the flow within G can never exceed that.

Proof

$$
\begin{aligned}
|f| &= f(S,T) && \text{(By Lemma given in 1.9.3)} \\
&= \sum_{u \in S} \sum_{v \in T} \big(f(u,v) - f(v,u)\big) && \text{((3) by definition of } f(S,T)) \\
&\leq \sum_{u \in S} \sum_{v \in T} \big(c(u,v) - 0\big) && \text{(Since } f(u,v) \leq c(u,v) \text{ and } -f(v,u) \leq 0 \text{ by the capacity constraints)} \\
&= c(S,T)
\end{aligned}
$$

1.10 Min Cut Theorem: Max flow

The minimum cut in G is equal to the max flow of G.
Let f be a flow in (G, s, t, c). Then the following 3 statements are equivalent:

1. f is a max flow.
2. There is no augmenting path (in G_f).
3. There exists a cut (S, T) such that $|f| = c(S, T)$.

Proof for (1) $\implies$ (2)
Assume for contradiction f is a max flow and there exists an augmenting path p. Then by Lemma in 1.8.1, $f \uparrow f_p$ is a flow in G of value

$$|f \uparrow f_p| = |f| + |f_p| > |f|,$$

which is a contradiction.

Proof for (2) $\implies$ (3)
Let $S = \{v \in V \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T = V \setminus S$. Then S, T partition V, $s \in S$ and $t \in T$ (no augmenting path), so (S, T) is a cut.
Now let $u \in S, v \in T$. Then $f(u, v) = c(u, v)$, otherwise $c_f(u, v) > 0$ so $(u, v) \in E_f$. Since u is reachable from s in G_f, that implies v is reachable from s in G_f, thus $v \in S$, contradicting $v \in T = V \setminus S$.
Similarly, $f(v, u) = 0$, otherwise $c_f(u, v) > 0$ and the same problem.
Thus,

$$\begin{aligned}
|f| &= f(S, T) \\
&= \sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u)) && \text{(Definition (3) of } f(S, T)) \\
&= \sum_{u \in S} \sum_{v \in T} (c(u, v) - 0) && \text{(Above argument)} \\
&= c(S, T) && \text{(Definition (4) of } c(S, T)).
\end{aligned}$$

Proof for (3) $\implies$ (1)
Let (S, T) be the cut from (3), and f' be any other flow in G. By the Corollary (from 1.9.3),

$$|f'| \leq c(S, T) = |f|,$$

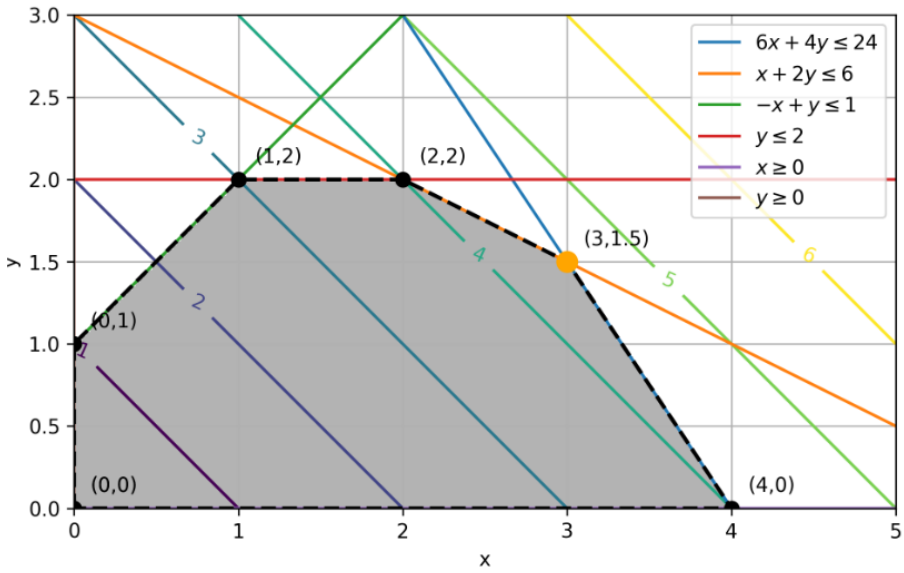
so f is a max flow.

2 Linear Programming

In linear programming (LP) is a mathod to optimize a function $f(x)$ given some constraints, where in LP f and the constraints are all linear. The LP problem defined by $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{k \times n}$, and $b \in \mathbb{R}^k$ is

$$\max \langle \mathbf{c}, \mathbf{x} \rangle \quad \text{subject to} \quad \mathbf{A}\mathbf{x} \leq \mathbf{b}.$$

Where $\langle \mathbf{c}, \mathbf{x} \rangle$ is the dot product of the constants c (which denotes how each element in x affect the function f) and the values we optimize over x. All while the constraint of $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ are kept satisfied.
Geometrically the optimization space will look like a polytope (an object with flat sides), see example below in figure 8.



Figur 8: Example of an LP problem that tries to find $\max x + y$. With max in the orange vertex $(3, 1.5)$

We optimize a linear function over a convex polytope. There are three options:

1. The set $\{x : Ax \leq b\}$ is empty.
2. The set $\{x : Ax \leq b\}$ is non-empty and the maximum is infinite.
3. The set $\{x : Ax < b\}$ is non-empty and the maximum is finite.

If the LP is feasible and bounded, then the **maximum is attained at a vertex of the polytope**, as we see in figure 8.

2.1 Reductions

Many problems can be reduced to LP and thus solved optimally. Max flow and shortest path are examples of problems that can be reduced to LP

2.1.1 Example of shortest path as a LP problem

We have a directed graph $G = (V, E)$ with edge weights given by $W : E \to \mathbb{R}$. We are also given two fixed vertices $s, t \in V$. Our goal is to compute the minimum weight of a path from s to t in G. This turns into an LP over $\mathbb{R}^V$ as follows:

$$\max x_t : \quad x_s = 0$$

$$\forall e = (u, v) \in E \quad x_v \leq x_u + W(e)$$

x_s is 0, since it is the weight from the source to itself. The constraint says, that the weight on the path from s to v is smaller or equal to the weight on the path to u plus the weight from u to v.

- If we set $x_v = \text{dist}(s, v)$ then all constraints are satisfied, which implies that the maximum is at least $\text{dist}(s, t)$.
- In the other direction, choose a shortest path $s = v_0 \to v_1 \to \cdots \to v_n = t$. Then, for all i,

  $$x_{v_{i+1}} \leq x_{v_i} + W(v_i, v_{i+1})$$

  and get that the maximum is at most the shortest path.

2.2 Duality

Every LP maximization problems comes with a dual minimization problem, and vice versa (like we saw in 2.1.1).
The standard form of an LP is

$$\max\langle c, x\rangle : Ax \leq b, \quad x \geq 0$$

The dual program is

$$\min\langle b, y\rangle : A^T y \geq c, \quad y \geq 0$$

The transformation can be done as

$$(A, b, c) \mapsto (-A^T, -c, -b)$$

And the dual of the dual is the primal.

Primal

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{29.16}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \tag{29.17}$$

$$x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n\,. \tag{29.18}$$

Dual

$$\text{minimize} \quad \sum_{i=1}^{m} b_i y_i \tag{29.83}$$

subject to

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \quad \text{for } j = 1, 2, \ldots, n\,, \tag{29.84}$$

$$y_i \geq 0 \quad \text{for } i = 1, 2, \ldots, m\,. \tag{29.85}$$

Figur 9: Using the notation from CLRS.

2.2.1 Weak Duality

Lemma 8 (weak duality). If x is feasible in the primal and y is feasible in the dual then

$$\langle c, x\rangle \leq \langle b, y\rangle.$$

The values of the dual, which is a minimization problem, provide upper bounds on the values of the primal, which is a maximization problem (and vice versa). And thus the primal can never exceed the minimum from the dual, but it is not necessarily a tight bound.

Strong Duality gives a thight bound, and thus the minimum from the dual is equal to the max of the primal[2].

2.3 Simplex Algorithm

The algorithm walks on the vertices of the polytope until it finds an optimum, as we know an optimal solution must be on a vertex.

[2]A special case of strong duality is the max-flow min-cut theorem. The dual of the "flow LP"is the cut LP".

2.4 Ellipsoid Algorithm

2.5 Interior Point Methods

These algorithms iteratively move a point inside the feasible region following various rules that are based on gradients, etc."until an optimal solution is found.

3 Randomized Algorithms

3.1 Randomized Quick Sort (RandQS)

Randomized quick sort is a Las Vegas randomized algorithm, as it always returns the correct answer. The pseudocode can be seen below in figure 10.

```
1: function RandQS(S = {s_1, ..., s_n})
   ▷ Assumes all elements in S are distinct.
2:     if |S| ≤ 1 then
3:         return S
4:     else
5:         Pick pivot x ∈ S, uniformly at random
6:         L ← {y ∈ S | y < x}        For each y ∈ S \ {x},
7:         R ← {y ∈ S | y > x}        compare to y to x once
8:         return RandQS(L)+[x]+RandQS(R)
```

Figur 10: Pseudocode of the RandQS

The algorithm return the number if its the only one in the input. Else it picks a random "pivot"number x in the list, and then concatenates that number between two recursive calls where the left calls input are all the numbers smaller than x and the right contains all the values larger than x.

3.1.1 Running time analysis

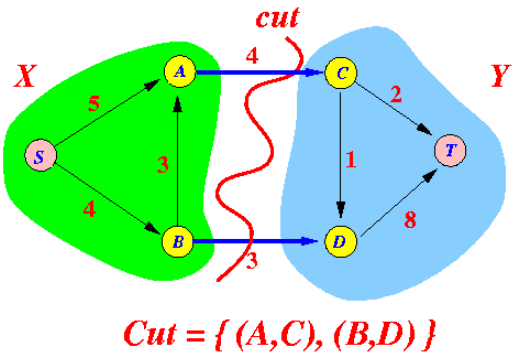
Lucky case:
We always choose the median, then $T(n) = \mathcal{O}(n) + 2T(n/2) = \mathcal{O}(nlog(n))$.
Unlucky case:
x is always the minimum, then $T(n) = \Omega(n) + T(n-1) = \Omega(n^2)$.
Average case:
The theorem is that

$$\mathbb{E}[X] = \mathcal{O}(n \log n) \tag{6}$$

Where X is number of comparisons. The proof is given on the slides.

3.2 Randomized min-cut

Given a graph $G = (V, E)$, $|V| \geq 2$. We want to find the minimum set of edges $C \subseteq E$ such that $G \setminus C$ is disconnected. We define $\lambda(G) = |C|$ to be the size of such cut (edge connectivity in G).

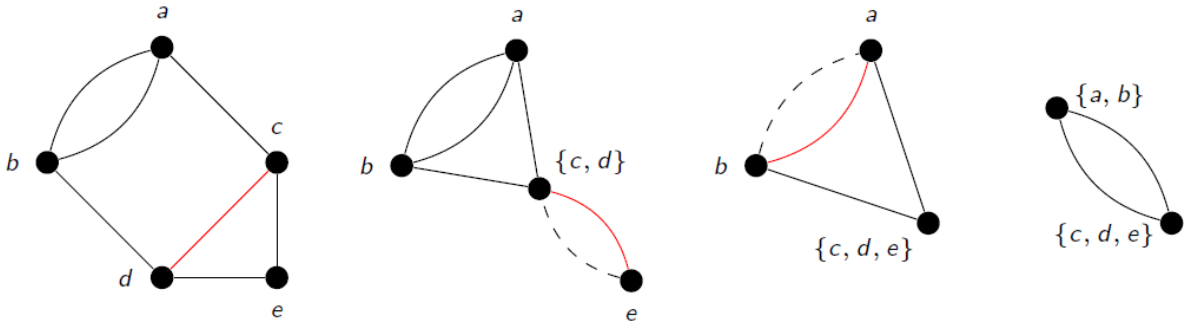


Figur 11: Example of a min cut with $\lambda(G) = 2$

```
1: function RandMinCut(V, E)
2:     while |V| > 2 and E ≠ ∅ do
3:         Pick e ∈ E uniformly at random.
4:         Contract e and remove self-loops.
5:     return E
```

Figur 12: Pseudocode of RandMinCut

The algorithm iterates as long as there are more than two vertices and the edges in G in non-empty. For each iteration we pick a random edge, and contract the two vertices that edge connects, and remove self-loops. When we only have 2 vertices left, we return the number of edges in the remaining graph. The algorithm is a monte carlo algorithm, since the number of iterations are deterministic but it does not always return the correct answer.



Figur 13: Example of running the RandMinCut.

The algorithm always returns a cut, but it is not certain that it is a min-cut.
Theorem:
For any min-cut C, the probability that RandMinCut(G) returns C is

$$Pr\left[\text{Returns } C\right] \geq \frac{2}{n(n-1)}$$

Since it is not guaranteed to return the min cut, we could run the algorithm multiple times, which will have the probability of not returning the min cut to be

$$\Pr[\text{not finding min-cut}] \leq (1 - 1/x)^x \leq (e^{-1/x})^x = 1/e$$

Where x is the number of times we run the algorithm and e is just a constant.

3.3 Las Vegas & Monte Carlo

Las Vegas algorithms always return the right answer, but the number of iterations is random.
Monte Carlo algorithms does not always return the right answer but does return in a deterministic amount of iterations.

4 Hashing

Given a universe U of keys and a positive integer m, then a random hash function is a randomly chosen function[3] that maps U to $[m]$[4]. It can also be seen as a function h where $x \in U$, then $h(x) \in [m]$ is a random variable (since the mapping of $h(x)$ to $[m]$ is random).

A hash function is **truly random** if $h(x)$ for $x \in U$ are independent and uniform

4.1 Universal hash function

A random hash function is universal if for all

$$\forall x \neq y \in U : Pr_h(h(x) = h(y)) \leq \frac{1}{m}$$

Thus a random hash function is universal if the probability of a collision is less than $\frac{1}{m}$.

4.1.1 c-approximate universal

Instead of being bounded by $\frac{1}{m}$ it is bounded by $\frac{c}{m}$ with c being some constant

$$\forall x \neq y \in U : Pr_h(h(x) = h(y)) \leq \frac{c}{m}$$

4.1.2 strongly universal

Aka "2-independent". For any two distinct values x, y in the universe U, then the probability that x, y respectively maps to q, r is exactly $\frac{1}{m^2}$. Formally given as for all $x \neq y \in U$, and $q, r \in [m]$

$$\Pr_h[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}$$

It is equivalent to:

- Each key is hashed uniformely into $[m]$:

$$\forall x \in U, q \in [m] : \Pr_h[h(x) = q] = \frac{1}{m}$$

- Any two distinct keys hash independently

Thus knowing what one key hashes to tells you nothing about what the other key hashes to.

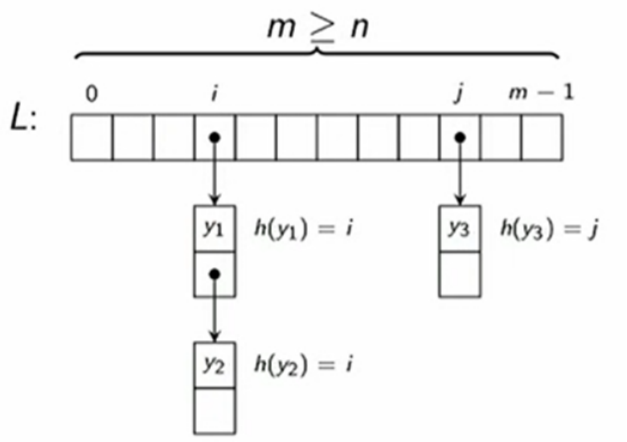
[3] Where each function it chooses from all does a mapping of U to $[m]$
[4] $[m] = \{0, \ldots, m-1\}$

4.2 Hashing with chaining

Given $m \geq n$, a random hash function $h : U \to [m]$, and the head of a linked list at index i such that

$$L[i] = \text{linked list over } \{y \in S | h(y) = i\}$$

Thus we have a possible array of $n - 1$ linked lists, each at their own index i, illustrated below in figure 14.



Figur 14: Illustration of the linked lists in hashing with chaining.

The linked list at $L[i]$ will then contain all the elements that hashed to i.
This structure will enable insert, delete, and member to be performed in $\mathcal{O}(|L[h(x)]| + 1)$.

Theorem
The expected length of the list is at must 1 if you search for an element not in the set ($x \notin S$). The set S refers to elements that we actually have hashed, and is contained in a list, for $x \notin S$

$$\mathbb{E}\left[L\left(h(x)\right)\right] \leq \frac{n}{m} \leq 1 \tag{7}$$

Theorem
Similarly if we search for an element that is in the set S ($x \in S$), and thus has actually been hashed, we get the

$$\mathbb{E}_h\left[|L\left(h(x)\right)|\right] \leq 1 + \frac{n-1}{m} \quad (\leq 2) \tag{8}$$

4.3 Signatures

If we want to assign a unique signature to each $x \in S \subseteq U$ with $|S| = n$, we can use a universal hash function with $s : U \to [n^3]$. Thus we need n unique signatures. The probability of getting that without collision is

$$\begin{aligned}
&\Pr_s[\exists\{x, y\} \subseteq S \text{ with } s(x) = s(y)] \\
&= \Pr_s\left[\bigcup_{\{x,y\}\subseteq S} \text{event } s(x) = s(y)\right] \\
&\leq \sum_{\{x,y\}\subseteq S} \Pr_s[s(x) = s(y)] \quad \text{(Union bound)} \\
&\leq \frac{\binom{n}{2}}{n^3} \quad \text{(s universal)} \\
&< \frac{1}{2n}
\end{aligned}$$

4.4 Multiple-mod-prime

Let $U = [u]$ and pick prime $p \geq u$. For any $a, b \in [p]$, and $m < u$, let $h^m_{a,b} : U \to [m]$ be

$$h^m_{a,b}(x) = (((ax + b) \mod p) \mod m) \tag{9}$$

In words, we first pick a prime number p larger or equal to u. We then pick two numbers a and b from $1 \dots p$, and a number m smaller than u. We then simply use (9) as the hash function.
It is however **not random**, since we don't pick the values at random.

If we however choose a and b independently uniformly at random, and we defined $h(x) = h^m_{a,b}(x)$. Then $h : U \to [m]$ is a 2-approximately strongly universal hash function. a and b have to be chosen at random every time we run the program.

4.5 Multiply-shift

Let $U = [2^w]$ and $m = 2^\ell$. For any odd $a \in [2^w]$, define

$$h_a(x) := \left\lfloor \frac{(ax) \mod 2^w}{2^{w-\ell}} \right\rfloor \tag{10}$$

The universe is defined as a range with the max being a power of 2. We then pick an odd a within the range of $1 \dots 2^w$ and then use the equation in (10).
This is efficiently implemented, as we work in bit-sizes arrays and with bit-shifts.
With $h(x) := h_a(x)$, then $h : U \to [m]$ is a 2-approximately universal hash function.

4.5.1 Strong Multiply-shift

It is the same concept as the normal multiply-shift hash function, but we have to pick a sufficiently large $\bar{w}$ (and thus restricting the size of the universe in comparison), and now also a and b (where a no longer needs to be odd).
Let $U = [2^w]$ and $m = 2^\ell$, and pick $\bar{w} \geq w + \ell - 1$. For any pair $(a, b) \in [2^{\bar{w}}]^2$, define

$$h_{a,b}(x) := \left\lfloor \frac{(ax + b) \mod 2^{\bar{w}}}{2^{\bar{w}-\ell}} \right\rfloor .$$

Choose $a, b \in [2^{\bar{w}}]$ independently and uniformly at random, and let $h(x) := h_{a,b}(x)$.
Then $h : U \to [m]$ is a strongly universal hash function.

4.6 Coordinated sampling

All agents that see an event make the same decision about whether to store it or not. Thus if we a set S_i and S_j, it must follow that

$$S_i \cup S_j \text{ is a sample of } A_i \cup A_j$$

$$S_i \cap S_j \text{ is a sample of } A_i \cap A_j$$

To do this, we pick a strongly universal hash function $h : U \to [m]$ and some value $t \in \{0, \dots, m\}$, which all the agents have. Then we simply keep the value if it hashes to something below t.
Thus if an agent sees the set $A_i \subseteq U$, the set

$$S_{h,t}(A_i) := \{x \in A_i \mid h(x) < t\}$$

is sampled. And thus we get the property as before:

$$S_{h,t}(A_i) \cup S_{h,t}(A_j) = S_{h,t}(A_i \cup A_j)$$

$$S_{h,t}(A_i) \cap S_{h,t}(A_j) = S_{h,t}(A_i \cap A_j)$$

We see that each $x \in A$ is sampled with probability $Pr[h(x) < t] = \frac{t}{m}$ (each x as $\frac{1}{m}$ probability of being any number from $1 \dots m$, and t of those are under t).
The expected size of the sample set is

$$\begin{aligned}
\mathbb{E}_h[|S_{h,t}(A)|] &= \mathbb{E}_h\left[\sum_{x \in A}[h(x) < t]\right] \\
&= \sum_{x \in A} \mathbb{E}_h[[h(x) < t]] \\
&= \sum_{x \in A} \Pr_h[h(x) < t] \\
&= \sum_{x \in A} \frac{t}{m} \\
&= |A| \cdot \frac{t}{m}
\end{aligned}$$

We can further use Chebyshev's inequality to get a bound on the size of the coordinated sample.
Let $X = |S_{h,t}(A^\star)|$[5] and for $a \in A^\star$, let $X_a = [h(a) < t]$. Then $X = \sum_{a \in A^\star} X_a$, and for any $a, b \in A^\star$, X_a and X_b are independent. Also, let $\mu = \mathbb{E}_h[X] = \frac{t}{m}|A^\star|$.

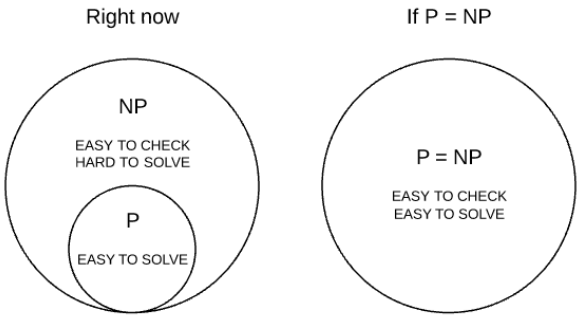
Then for any $q > 0$,

$$\begin{aligned}
\Pr_h\left[\left|\frac{m}{t}\frac{|S_{h,t}(A^\star)|}{|A^\star|} - 1\right| \geq q \cdot \frac{1}{\sqrt{\frac{t}{m}|A^\star|}}\right] &= \Pr_h\left[\left||S_{h,t}(A^\star)| - \frac{t}{m}|A^\star|\right| \geq q \cdot \sqrt{\frac{t}{m}|A^\star|}\right] \\
&= \Pr_h\left[|X - \mu| \geq q \cdot \sqrt{\mu}\right] \\
&\leq \frac{1}{q^2}.
\end{aligned}$$

It requires h to be uniform and pairwise independent to use Chebyshev's inequality.

5 Computational complexity, P, NP, and NP-completeness

In general, we have that P are the problems we can solve in polynomial time, and NP are the problems we can verify in polynomial time.



Figur 15: Diagram of P vs NP

[5] $|A^\star| \approx \frac{m}{t}|S_{h,t}(A^\star)|$

Property	P	NP	co-NP	NP-Complete	NP-Hard
Definition	Problems solvable in polynomial time	Problems verifiable in polynomial time	Complement of problems in NP	Hardest problems in NP	At least as hard as hardest problems in NP
Example Problem	Sorting, Matrix Multiplication	SAT (decision), Hamiltonian Path	Validity of Logical Formulas	SAT, 3-SAT, Knapsack (decision version)	TSP (optimization), Halting Problem
Solvable in Polynomial Time	Yes	Unknown (depends on P vs NP)	Unknown (depends on P vs NP)	Unknown (depends on P vs NP)	Not necessarily
Verifiable in Polynomial Time	Yes	Yes	No (unless co-NP = NP)	Yes	Not required
Belongs to NP	Yes	Yes	No	Yes	Not necessarily
Belongs to co-NP	Yes	No	Yes	Unknown	Not necessarily
Reduction	Not applicable	Not applicable	Not applicable	All NP problems can reduce to it	All NP problems can reduce to it
Complement Exists in Class	Yes	No	Yes	Unknown	Not necessarily

Figur 16: Table of the different classes, and how they relate.

5.1 Decision problem

Given I as the instances/problems, then the solutions S are all binary thus "yes/"no": $S = \{0, 1\}$. E.g. the PATH(G, u, v, k) problem that is 1 if there's a path from u to v in G with at most k edges (else 0).
Usually, optimization problems can be turned into decision problems (like SHORTEST-PATH to PATH)

5.2 Polynomial-time solvable problems

Assuming the problems are encoded as binary strings, and an algorithm solves a problem in time $\mathcal{O}(T(n))$ for an instance with length n if it returns 0 or 1 in $\mathcal{O}(T(n))$. It is further solved in polynomial time if $T(n) = \mathcal{O}(n^k)$

5.3 Encoding the problem

When defining a problem, you also have to define how you encode/represent an instance of the problem (and thus in what terms the problem scales). In general, we choose a "smart"encoding, e.g. binary for numbers. $\langle x \rangle$ denotes that the instance x has been encoded. And we always choose encodings that does not affect the running time of our problem.

5.4 Languages

An alphabet Σ is defined as a finite set of symbols (often as 0's and 1's).
A language L over Σ is defined as a set of strings from symbols of Σ. And thus strings can be made from whatever symbols/characters there exists in the alphabet.

Example of a Σ and a L
If $\Sigma = \{a, b, c\}$, then we can make $L = \{a, ba, cab, bbac, \dots\}$.

The empty string is given as ϵ and the empty language is denoted as $\emptyset$ (also without the empty string). Σ^* denotes the language of all possible strings (also ϵ).

5.5 Decision problems as a language

Given $Q(x)$ that maps instances x to $\Sigma = \{0, 1\}$, and thus provides the answer to the decision problem. And thus

$$L = \{x \in \Sigma^* \mid Q(x) = 1\}$$

Which denotes the language as all the instances x where $Q(x) = 1$. These are all the languages/combinations of binary strings that cause the decision problem to be a "yes"instance.

5.6 Language accepted/decided by an Algorithm

The algorithm A with the outputs $A(x) \in \{0, 1\}$ for a decision problem x. A accepts a string x if $A(x) = 1$ and rejects if $A(x) = 0$. It is possible for A to never return if the algorithm never terminates. A language accepted by A is

$$L = \{x \in \{0, 1\}^* \mid A(x) = 1\}$$

Then we say L is decided by A, which is stronger than just accepting it.

5.6.1 Accepting in polynomial time

Language L is accepted by A in polynomial time if A accepts L and runs in polynomial time on strings from L. Likewise is L decided by A in polynomial time if A decides L and runs in polynomial time on all strings (and not just those strings from the language L).

5.7 Complexity class P

The complexity class P has the definition

$$\begin{aligned} P &= \{L \subseteq \{0, 1\}^* \mid \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\} \\ &= \{L \subseteq \{0, 1\}^* \mid \text{there exists an algorithm } A \text{ that accepts } L \text{ in polynomial time}\}. \end{aligned}$$

In other words, the class P are all the problems that can be solved in polynomial time.

5.8 Verification

Again given a language L and an algorithm A which takes $x, c \in \Sigma^*$ as input. Instead of finding a solution, the algorithm just verifies that c is a solution to x.

5.8.1 Example of HAM-CYCLE problem

An undirected graph G is Hamiltonian if it contains a simple cycle containing every vertex of G

$$\text{HAM-CYCLE} = \{\langle G\rangle | \text{G is Hamiltonian}$$

Unknown if it can be solved in polynomial time, time can be verified in polynomial time.

Verifying HAM-CYCLE
Given algorithm A_{ham} taking $\langle G\rangle$ $\langle C\rangle$ as input. Then A_{ham} checks that $\langle G\rangle$ defines an undirected graph and that $\langle C\rangle$ encodes a cycle containing every vertex exactly once.

5.9 Veryfying a language

A verification algorithm is an algorithm A taking a problem x and a certificate y where $x, y \in \{0,1\}^*$. And thus A verifies the string x if there is a certificate y such that $A(x, y) = 1$. Defined as

$$L = \{x \in \{0,1\}^* \mid \text{there is a } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\}.$$

Which in the examples of the HAM-CYCLE would look like

$$\text{HAM-CYCLE} = \{x \in \{0,1\}^* \mid \text{there is a } y \in \{0,1\}^* \text{ such that } A_{\text{ham}}(x, y) = 1\}.$$

5.10 Complexity class NP

The class NP[6] is the class of languages that can be verified in polynomial time. $L \in NP$ if there is an A that runs in polynomial time and a constant c such that

$$L = \{x \in \{0,1\}^* \mid \text{there is a } y \in \{0,1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}$$

And thus we see that the length of the certificate y is bounded by $\mathcal{O}(|\S|^{\lrcorner})$.
We thus have whenever a problem/language is $L \in P$ then $L \in NP$, as if we can solve it in polynomial time, we can at least verify it in polynomial time.

5.11 $co - NP$

$co - NP$ is the class of languages L such that $\bar{L} \in NP$. Thus if the complement of L is in NP, then $L \in co - NP$. We have that $P \subseteq NP \cap co - NP$

5.12 NP complete problems

NP complete problems are the "hardest"to solve in the NP class, and thus if we can solve them in polynomial time, then we should be able to solve all problems in NP in polynomial time. E.g. HAM-CYCLE is NP-complete, and thus if we could show HAM-CYCLE $\in P$ then $P = NP$.

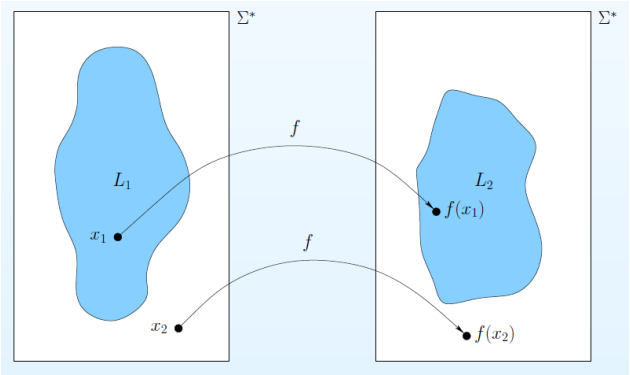
5.12.1 Polynomial time reducibility ($L_1 \leq_P L_2$)

Language L_1 is polynomial-time reducible to language L_2 if there is a polynomial time computable function f such that

$$f : \{0,1\}^* \to \{0,1\}^* \text{ such that for all } x \in \{0,1\}^*, x \in L_1 \iff f(x) \in L_2.$$

If that is the case, then we write $L_1 \leq_P L_2$, which in words means that L_1 is no harder than L_2 (or L_2 is at least as difficult as L_1).

[6] Non-deterministic polynomial time



Figur 17: Example of the mapping of the function that reduces L_1 to L_2

This gives the implication that if L_2 is in P and L_1 can be reduced to L_2, then must L_1 also be in P:

$$L_1 \leq_P L_2 \wedge L_2 \in P \implies L_1 \in P$$

5.12.2 NP complete languages (NPC)

A language L is NP complete if

1. $L \in NP$
2. $L' \leq_P L$ for every $L' \in NP$

Thus it must be in NP, and every problem in NP can be reduced to it, meaning it is at least as hard as any of the problems in NP (or that all problems are at most as difficult as that).
If some language of NPC belongs to be, we have that $P = NP$.

L is further NP hard if property 2 holds, but 1 doesn't. It means it is harder than all problems, and thus we may not solve it in polynomial time.

5.13 Examples of NP completeness

We have one language that we now is NPC (e.g. CIRCUIT-SAT), then we just have to reduce other languages to that, then they must also be NPC.

5.13.1 SAT

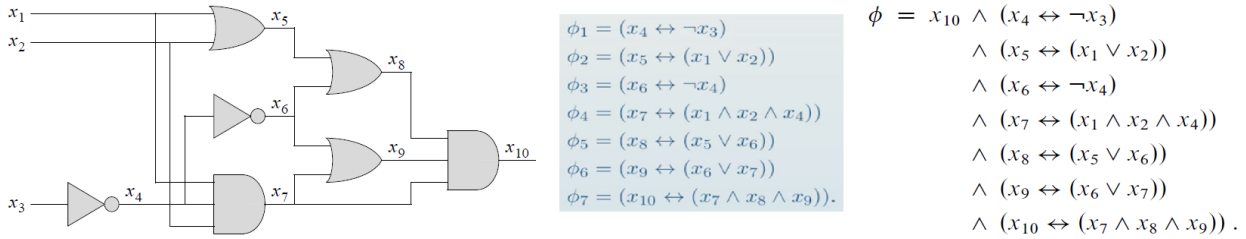
A **boolean formula** ϕ consists of boolean variables $x_1, \ldots, x_n$, boolean connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$, and parentheses (and).
Example: $\phi = (x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$.
A **satisfying assignment** for a boolean formula ϕ is an assignment of 0/1-values to variables that makes ϕ evaluate to 1. ϕ is **satisfiable** if there exists a satisfying assignment for ϕ.
We can now define the problem **SAT**:

$$\text{SAT} = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable boolean formula}\}.$$

Showing that SAT is NPC we first need to show that its NP, and that CIRCUIT-SAT $\leq_P$ SAT

Showing NPC:
We create an algorithm A that taking inputs x and y. x is the boolean formula ϕ (the instance) and y the assignment of values in ϕ (the certificate). Then A returns 1 if y satisfies ϕ, and 0 otherwise. This can easily be done in polynomial time, and thus it is in NPC.

Showing CIRCUIT-SAT $\leq_P$ SAT: Given circuit C, we transform it into a boolean function ϕ. To do this, we give each wire of C an associate variable x_i and lets x_m be the output wire. We then construct sub-formulas for each gate in C, such that the output wire variables are a function of the input wire variables. When we have these sub-formulas $\phi_1, \ldots, \phi_k$, we can define the full ϕ to be $x_m \wedge \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_k$. See example below in figure 18.



Figur 18: Example of C turned into ϕ. Taken from CLRS fig 34.10

This reduction can be constructed in polynomial time. And we see that C is satisfiable if and only if ϕ also is:

$$\langle C \rangle \in \text{CIRCUIT-SAT} \iff \langle \phi \rangle \in \text{SAT}.$$

5.13.2 3-CNF-SAT

Let ϕ be a boolean formula.
A *literal* in ϕ is an occurrence of a variable or its negation.
Suppose ϕ is the AND of sub-formulas, called *clauses*.
Furthermore, suppose that each clause is the OR of exactly 3 distinct literals.
Then we say that ϕ is in *3-conjunctive normal form*, or **3-CNF**.
Example:

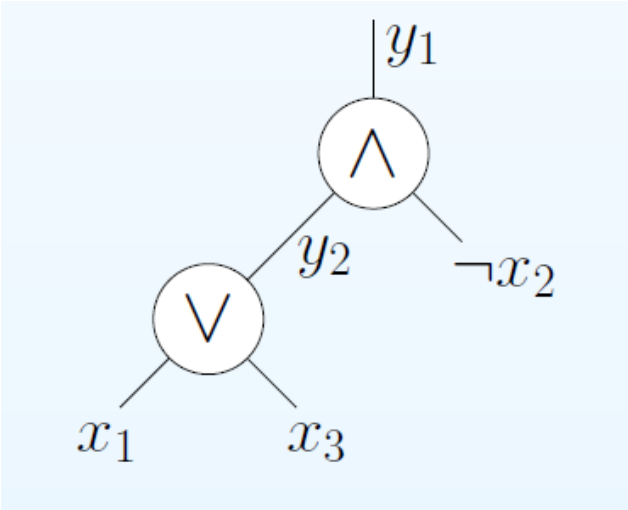
$$\phi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4).$$

And thus can be defined as

$$\text{3-CNF-SAT} = \{\langle \phi \rangle \mid \phi \text{ is in 3-CNF and satisfiable}\}.$$

To show that 3-CNF-SAT is NPC, we will show that it is in NP and SAT $\leq_P$ 3-CNF-SAT.
Showing NP:
Same argument as above in section 5.13.1 for SAT, we plug the values in for the variables and check if it satisfies.

Showing SAT $\leq_P$ 3-CNF-SAT:
Let $\langle \phi \rangle$ be an instance of SAT.
We construct a *parse tree* for ϕ where leaves are literals and internal nodes are connectives.



Figur 19: Example of parse tree for $\phi = (x_1 \vee x_3) \wedge \neg x_2$

Regard the tree as a circuit with internal nodes as gates. We can construct formulas $\phi'_1, \ldots, \phi'_k$ for each of these gates, as before. Let $\phi' = y_1 \wedge \phi'_1 \wedge \phi'_2 \wedge \cdots \wedge \phi'_k$, where y_1 is the output wire.
ϕ' is satisfiable iff ϕ is satisfiable. Each clause of ϕ' has at most 3 literals.

As an example if we consider the clause ϕ'_i, where $\phi'_i = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$. This will give the truth table as can be seen below in figure 20

y_1	y_2	x_2	ϕ'_i	$\phi''_i \equiv \neg\phi'_i$
0	0	0	1	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

Figur 20: Truth table for the $\phi'_i = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

The formula for ϕ''_i can then be found by making a sub-formula for each satisfiable combination (most right column in the example) to give

$$\neg\phi'_i \equiv \phi''_i = (\neg y_1 \wedge y_2 \wedge \neg x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge y_2 \wedge x_2).$$

If the variables were set to 1 we negate it. This is however in disjunctive normal form, to get it into conjunctive normal form, we apply De Morgan's law[7]:

$$\neg\phi''_i \equiv (y_1 \vee \neg y_2 \vee x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee \neg y_2 \vee \neg x_2).$$

[7] $\neg(P_1 \wedge P_2 \wedge \cdots \wedge P_n) \iff \neg P_1 \vee \neg P_2 \vee \cdots \vee \neg P_n$ and $\neg(P_1 \vee P_2 \vee \cdots \vee P_n) \iff \neg P_1 \wedge \neg P_2 \wedge \cdots \wedge \neg P_n$

And thus we have $\phi_i'' \equiv \neg\phi_i'$.
It follows that $\neg\phi_i'' \equiv \neg(\neg\phi_i') \equiv \phi_i'$.
In words, $\neg\phi_i''$ is equivalent to the original ϕ_i'.
This gives that it has at most 3 literals, but not exactly. But this can easility be fixed by introducing dummy variables p and q:

$$\ell_1 \vee \ell_2 \equiv (\ell_1 \vee \ell_2 \vee p) \wedge (\ell_1 \vee \ell_2 \vee \neg p)\ell \equiv (\ell \vee p \vee q) \wedge (\ell \vee p \vee \neg q) \wedge (\ell \vee \neg p \vee q) \wedge (\ell \vee \neg p \vee \neg q)$$

And thus we can convert ϕ' into ϕ''' in 3-CNF in polynomial time, and thus we have shown that

$$\langle\phi\rangle \in \text{SAT} \iff \langle\phi'''\rangle \in \text{3-CNF-SAT}$$

Thus SAT $\leq_P$ 3-CNF-SAT, and 3-CNF-SAT is therefore NP complete.

5.13.3 TSP

5.13.4 SUBSET-SUM

Given a set S of positive integers and given integer target $t > 0$.
Is there a subset S' of S summing to t?
As a language:

$$\text{SUBSET-SUM} = \{\langle S, t\rangle \mid \exists S' \subseteq S \text{ so that } t = \sum_{s \in S'} s\}.$$

Showing NPC:
Clearly in NP, as we just sum the numbers and check if the sum is the what we expect.

Showing 3-CNF-SAT $\leq_P$ SUBSET-SUM:
Consider a 3-CNF-formula ϕ with n variables $x_1, \ldots, x_n$ and k clauses $C_1, \ldots, C_k$.
We will construct an instance $\langle S, t\rangle$ such that:

$$\langle\phi\rangle \in \text{3-CNF-SAT} \iff \langle S, t\rangle \in \text{SUBSET-SUM}.$$

In other words, we want that ϕ is satisfiable if and only if S has a subset summing to t.

To construct S and t we do

- For each variable x_i, create two decimal numbers v_i and v_i'.
- For each clause C_j, create two decimal numbers s_j and s_j'.
- Finally, create a decimal number t.
- Each number has $n + k$ digits.
- The n most significant digits are associated with $x_1, \ldots, x_n$.
- The k least significant digits are associated with $C_1, \ldots, C_k$.
- S consists of numbers $v_1, v_1', v_2, v_2', \ldots, v_n, v_n'$ and $s_1, s_1', s_2, s_2', \ldots, s_k, s_k'$.

And we further specify

- Digit x_i of v_i and digit x_i of v_i' is 1.
- Digit C_j of v_i is 1 if $x_i \in C_j$.
- Digit C_j of v_i' is 1 if $\neg x_i \in C_j$.
- Digit C_i of s_i is 1 and digit C_i of s_i' is 2.

- Target t is defined to be:

$$t = \underbrace{11\ldots1}_{n}\underbrace{44\ldots4}_{k}.$$

- All digits not specified above are 0.

We further assume that no clause contains both a variable and its negation, and each variable appears somewhere.

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

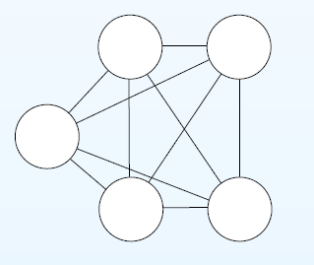
Figur 21: The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$, and $C_4 = (x_1 \vee x_2 \vee x_3)$. A satisfying assignment of ϕ is $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. The set S produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. The target t is 1114444. The subset $S' \subseteq S$ is lightly shaded, and it contains v'_1, v'_2, and v_3, corresponding to the satisfying assignment. It also contains slack variables $s_1, s'_1, s_2, s'_2, s_3, s_4$, and s'_4 to achieve the target value of 4 in the digits labeled by C_1 through C_4. CLRS fig 34.19.

Assume $\langle\phi\rangle \in$ 3-CNF-SAT.
In other words, assume that ϕ is in 3-CNF and satisfiable.
We will find a subset S' of S with $\sum_{s\in S'} s = t$.
Assign values to $x_1, \ldots, x_n$ that satisfy ϕ.
For $i = 1, \ldots, n$, if $x_i = 1$ then include v_i in S'; otherwise include v'_i.
Include additional numbers from $\{s_1, s'_1, s_2, s'_2, \ldots, s_k, s'_k\}$ to reach target t. Let S' be a subset of S summing to t.
We need to find a satisfying assignment for ϕ.
We set x_i to 1 if and only if $v_i \in S'$.

$$\langle S, t\rangle \in \text{SUBSET-SUM} \implies \langle\phi\rangle \in \text{3-CNF-SAT}$$

5.13.5 CLIQUE

Let $G = (V, E)$ be an undirected graph. A *clique* in G is a subset $V' \subseteq V$ such that $(u, v) \in E$ for all distinct $u, v \in V'$. The size of the clique is $|V'|$.



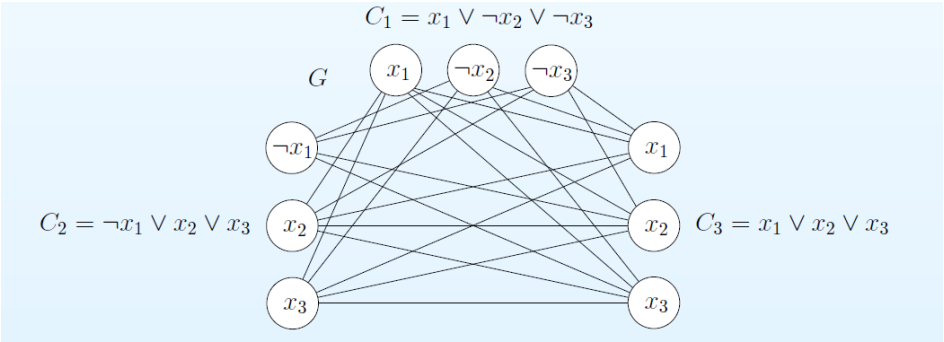
Figur 22: Example of a clique with size 5.

And thus the problem is defined as

$$\text{CLIQUE} = \{\langle G, k\rangle \mid G \text{ is a graph containing a clique of size } k\}.$$

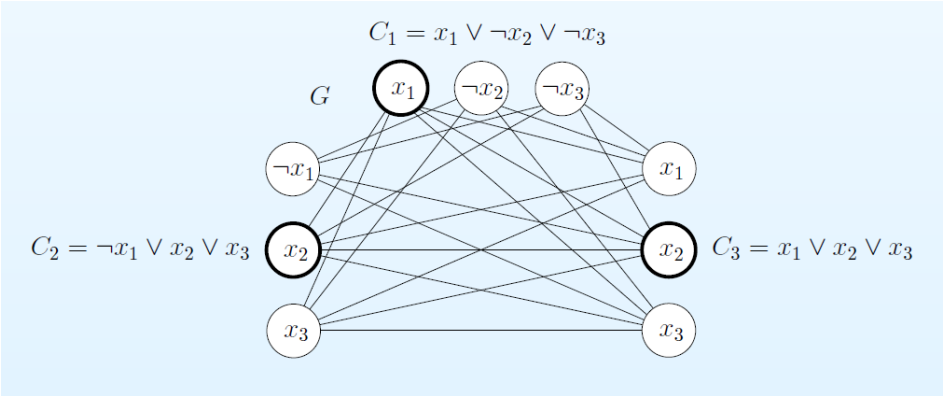
Showing NP:
To show CLIQUE $\in$ NP, consider an algorithm A taking two inputs, $\langle G, k\rangle$ and a certificate y. y specifies a subset V' of vertices of G. A checks that $|V'| = k$ and that V' is a clique in G. This can easily be done in polynomial time. Thus, CLIQUE $\in$ NP.

Showing 3-CNF-SAT $\leq_P$ CLIQUE:
Given a formula $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ in 3-CNF.
We will construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k. For each $C_r = \ell_1^r \vee \ell_2^r \vee \ell_3^r$, we include three vertices v_1^r, v_2^r, v_3^r to G.
There is an edge (v_i^r, v_j^s) in G if and only if $r \neq s$ and ℓ_i^r is not the negation of ℓ_j^s.



Figur 23: Example with $\phi = C_1 \wedge C_2 \wedge C_3$

We then need to show that ϕ is satisfiable if and only if G has a clique of size k. We first assume that $\langle\phi\rangle \in$ 3-CNFSAT, where in a satisfying assignment for ϕ, each clause C_i of ϕ has at least one true literal. We pick the corresponding vertex in G, which gives the total of k vertices. Continuing the example from figure 23, with a satisfying example below in figure 24.

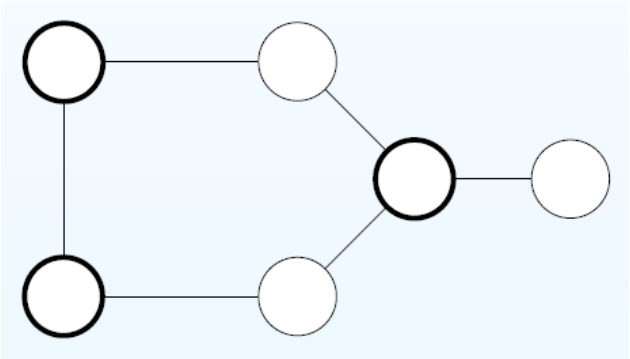


Figur 24: Satisfying example with $x_1 = 1, x_2 = 1, x_3 = 0$.

This gives a clique of size K. Since there must be an edge between each pair of these vertices since no picked literal can be the negation of another picked literal (we only picked true literals).
We however still remain to show that if G has a clique of size k then ϕ is satisfiable:
Let V' be a clique G of size k, then each vertex triple of G has exactly one vertex in V'. We assign 1 to the literal of ϕ corresponding to that vertex (in figure 24, it corresponds to $x_1 = 1, x_2 = 1$. x_3 can set to an arbitrary value). No variable of ϕ is assigned to both 0 and 1, since there is no edge between a variable and its negation. This assignment satisfies ϕ, and thus makes each clause true.

5.13.6 VERTEX-COVER

A vertex cover of $G = (V, E)$ is a subset $V' \subseteq V$ such that every edge of E has at least one endpoint in V'. In the vertex cover problem, we try and find a minimum-size vertex cover of G.



Figur 25: Example of a vertex cover with size 3.

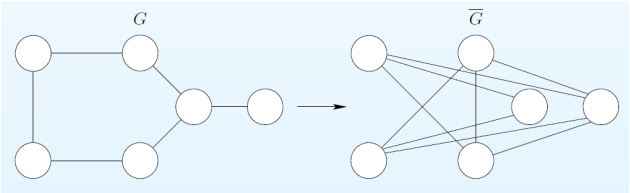
Formally the problem is

$$\text{VERTEX-COVER} = \{\langle G, k\rangle \mid G \text{ has a vertex cover of size } k\}$$

In other words, the decision problem is if G has a vertex cover of size k.
Showing NP:
The verification algorithm takes an instance $\langle G, k\rangle$ and a certificate denoting a subset V' of vertices of G. It checks that V' has size k and that every edge of G is incident to at least one vertex of V'. This can of course be done in polynomial time.

Showing NP **hard**:
We show CLIQUE $\leq_P$ VERTEX-COVER. Given an instance $\langle G, k\rangle$ of the clique problem. Let n denote the number of vertices of G. We transform $\langle G, k\rangle$ in polynomial time to the instance $\langle \overline{G}, n - k\rangle$ of the vertex

cover problem. Here, $\overline{G}$ is the *complement* of G which has the same vertex set as G and has an edge between two vertices u and v if and only if there is no edge between u and v in G. This can be illustrated with the example below in figure 26.

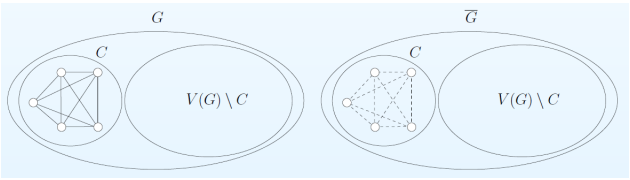


Figur 26: Example of converting G to $\bar{G}$.

Thus we just take a vertex, remove its current edges, and then make edges to all other vertices other than the ones it had before.
We then need to show that

$$\langle G, k\rangle \in \text{CLIQUE} \iff \langle \overline{G}, n-k\rangle \in \text{VERTEX-COVER}.$$

We see that any vertex set C is a clique in G if and only if C contains no edges in $\overline{G}$.



Figur 27: Enter Caption

Hence, C is a clique of size k in G if and only if $V(G) \setminus C$ is a vertex cover of size $n-k$ in $\overline{G}$.
To convert between them, we take the compliment, then solve the vertex cover, then the max clique will be the size of the vertex cover of the new one to be k, and then the max clique will be $|V| - k$.

6 Exact exponential algorithms and parameterized complexity

6.1 Exact exponential algorithms

We usually want algorithms in polynomial time, but sometimes we need to relax that to allow for exponential algorithms.

6.1.1 $\mathcal{O}^*$ notation

When dealing with exponential time algorithms, we don't care about any polynomial time part of the time complexity, as the dominant part is the exponential anyways. We define it as.

$$f(n) \in \mathcal{O}^*(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words, it is the same as $\mathcal{O}$ but ignoring any polynomial factors. We then have for all $a > 1$ and $\epsilon > 0 : \mathcal{O}(a^n) \subset \mathcal{O}^*(a^n) \subset \mathcal{O}((a+\epsilon)^n)$.
Using the brute-force algorithms of just trying all certificates the language can produce then has a time complexity of $\mathcal{O}^*(2^{m(x)})$.

6.1.2 Size of a problem

For the problems we usually encounter, the size of the input is often as follows in table 1.

Expression	Description
n, or $m+n$	for graphs with n vertices and m edges.
$\|S\|$	for problems involving some set S.
#variables	for SAT-type problems.

Tabel 1: Usual size definitions.

We further specified the certificate size, brute-force time complexity, and the best exact exponential time algorithm discussed in this course. This can be seen below in table 2.

Problem	Certificate size	Brute-force time	This course
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^*(2^n)$	$\mathcal{O}^*(1.45^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^*(n!)$	$\mathcal{O}^*(2^n)$
k-Vertex Cover	$m(x) = k\log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \text{poly}\|x\|)$	$\mathcal{O}_k(m+n)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^*(k^n)$	?

Tabel 2: Summary of problems, certificate sizes, and complexities discussed in this class.

6.1.3 TSP (Bellman-Held-Karp / Fomin-Kratsch)

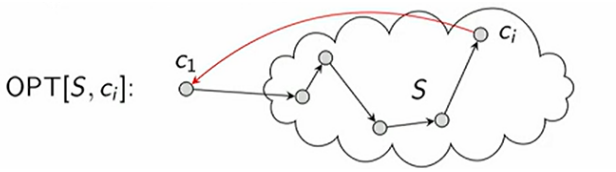
The traveling salesman problem (TSP) is stated as: Given cities $c_1, \ldots, c_n$ and distances $d_{ij} = d(c_i, c_j)$, find a tour of minimal length, visiting all cities exactly once. Equivalent it can be defined as finding the permutation π that minimizes

$$d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$$

The idea of the Bellman-Held-Karp algorithm is to defined a subset $S \subseteq \{c_2, \ldots, c_n\}$ $c_i \in S$ defines

$\text{OPT}[S, c_i] :=$ length of any shortest path in $S \cup \{c_1\}$ that starts in c_1 and visits all of S once and ends in c_i

Thus opt is just the minimum length of any path that starts in c_1, then goes into S and goes through all S, and then ends the path in c_i. This is illustrated below in figure 28.



Figur 28: Definition of OPT visualized, where the red arrow corresponds to $d(c_i, c_1)$ since it completes the tour.

Thus we have the length of the minimal tour is

$$\min\{\text{OPT}[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$$

We thus get the following **lemma**:

$$\text{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\}, \\ \min\{\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\} & \text{if } \{c_i\} \subset S. \end{cases} \tag{11}$$

The algorithm based on this lemma can be seen below

Algorithm 1 TSP Algorithm

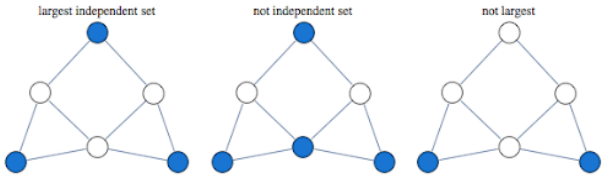
Input: $\{c_1, \ldots, c_n\}$, distance function d
Output: Optimal TSP cost

for $i \leftarrow 2$ **to** n **do**
  $\text{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
end
for $j \leftarrow 2$ **to** $n-1$ **do**
  for $S \subseteq \{c_2, \ldots, c_n\}$ ***with*** $|S| = j$ **do**
    for $c_i \in S$ **do**
      $\text{OPT}[S, c_i] \leftarrow \min\{\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
    end
  end
end
return $\min\{OPT[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$

The algorithm first calculates the distance from c_1 to all c_i. It then keeps increasing the size of the subset S from 2 to $n-1$, and for each c_k in S, we try and set that to c_i and calculate (11). The algorithm finally returns the minimum length of the paths containing all c's in S. This solves the TSP in $=(n^2 \cdot 2^n) = \mathcal{O}^*(2^n)$. This uses dynamic programming and is also called Fomin-Kratsch.

6.1.4 MIS via Branching

The problem maximum independent set (MIS) is defined as: Given an undirected graph (V, E), find the maximum cardinality of $I \subseteq V$ such that each edge has at most one endpoint in I.



Figur 29: Example of what is a MIS and what is not.

The naive method to finding this is to just try all 2^n subsets ($m = |V|$), which takes $\mathcal{O}^*(2^n)$.

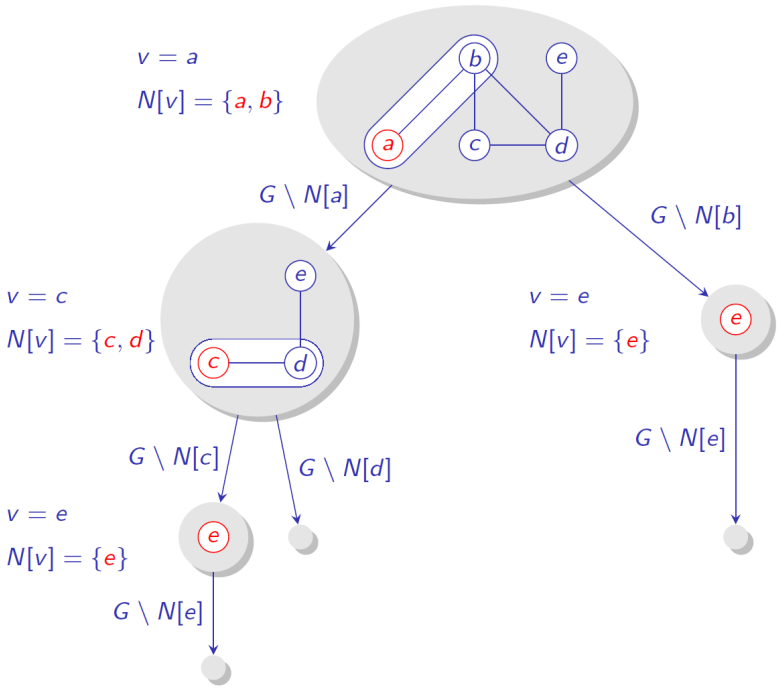
Instead, we see that if I is a solution, then for all $v \in V \setminus I$, there exists $w \in I$ such that $(v, w) \in E$, otherwise $I \cup \{v\}$ would be a larger solution.
For $v \in V$, define $N[v] := \{v\} \cup \{w \in V \mid (v, w) \in E\}$. This is called the *closed neighborhood* of v. Then this means $N[v] \cap I \neq \emptyset$ for all $v \in V$.
We use this to create the algorithm below

Algorithm 2 MISsize

Input: $G = (V, E)$, an undirected graph
Output: Size of the maximum independent set

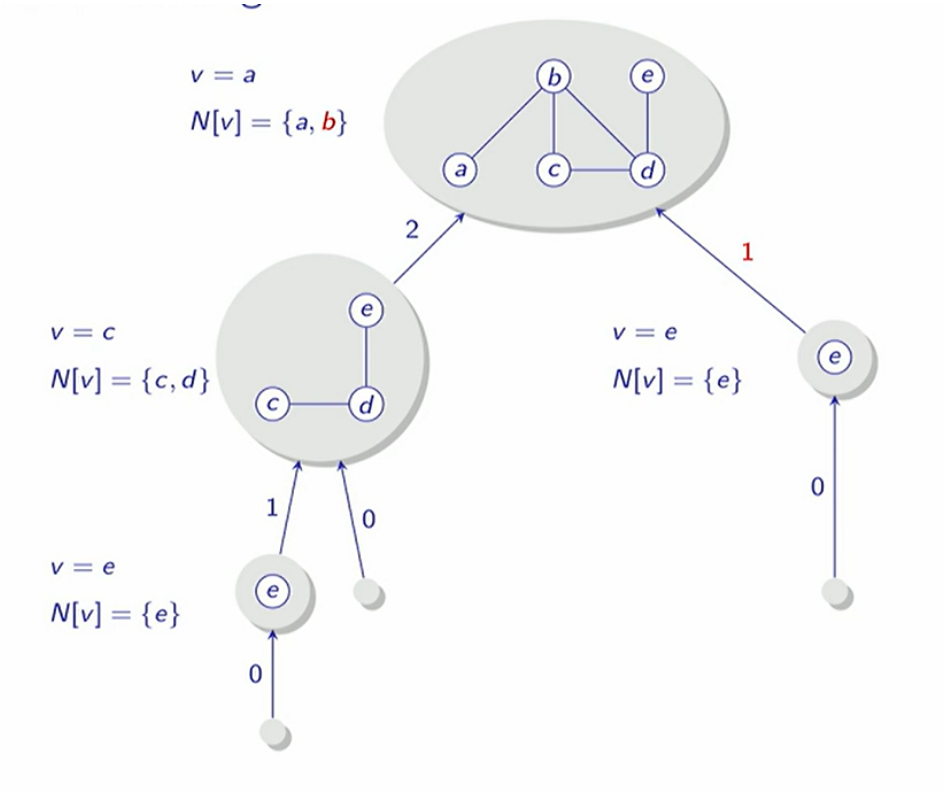
if $V = \emptyset$ **then**
  return 0
$v \leftarrow$ vertex in V of minimum degree;
return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

The algorithm returns 0 if there are no vertices, else it first sets v to be the vertex with the minimum degree. It then adds 1 to the recursive call of removing the neighborhood of one of the neighbors of v in G. An example of the algorithm can be seen below in figure 30.



Figur 30: Example of running the MIS algorithm.

The algorithm first picks an arbitrary vertex v of minimum degree (between a and e) and sets $v = a$. The neighborhood of $N[v] = N[a] = \{a, b\}$, and thus calls itself recursively with the subset of G by removing $N[a]$. It then continues to do the same, picks c, removes the neighborhood and calls recursively until the set is empty, and we then go up the recursion tree, calls with $N[d]$ removed, and then for each recursive call the subsets makes, the return value of the maximum size of the subsets plus 1. The final values can further be seen in the figure 31 below with return values added.



Figur 31: Example of the MIS algorithm with return values of the subproblems.

This algorithm runs in $\mathcal{O}(1, 44225^n)$

6.2 Parameterized algorithms

6.2.1 Bar fight prevention / k-vertex cover

The two problems bar fight prevention and k-vetex cover are equivalent:
Bar fight problem:
A bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq$ k of the n people who want in. Is that enough to prevent fights, and if so, who should be blocked?

k-vertex cover problem:
Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a k-Vertex Cover in the graph, and its complement $V \setminus C$ is an Independent Set of size $n - k$.

There are multiple ways of solving this:
For concreteness in the following, suppose $n = 1000$ and $k = 10$.

- **Naive 1:** Try all 2^n subsets of people.

$$2^{1000} \approx 1.07 \cdot 10^{301} \text{ cases.}$$

- **Naive 2:** Use Exact MIS algorithm.

$$2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159} \text{ cases.}$$

- **Better 1:** Try all $\binom{n}{k}$ subsets of k people.

$$\binom{1000}{10} \approx 2.63 \cdot 10^{23} \text{ cases.}$$

6.2.2 Using kernelization

To improve the algorithm we can use kernelization, where we utilize specific properties to decrease the size of the problem set, or make rules to capture different subsets of the problem with particular properties:

1. **If $d(v) = 0$:**
   - **Action:** Let v in and remove v from G.
   - **Reasoning:** This is safe because there are no conflicts involving v.
2. **If $d(v) > k$:**
   - **Action:** Reject v, remove v from G, and decrease k.
   - **Reasoning:** Allowing v in would mean rejecting $d(v) > k$ other vertices, which is not feasible.
3. **If $d(v) \leq k$ for all v but $|E| > k$:**
   - **Observation:** No solution is possible.
   - **Reasoning:** Each rejection resolves at most k conflicts, and with $|E| > k$, not enough conflicts can be resolved.

Problem Reduction

The above ideas reduce the problem to a graph H with at most $|V| \leq 2k^2$ vertices.
Why?

$$|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$$

At this point, you can try all subsets of k vertices from H:

$$\binom{2k^2}{k} \approx \left(\frac{2 \cdot 10^2}{10}\right)^{10} \approx 2.24 \cdot 10^{16}.$$

Further Ideas

1. **If $N[v] = \{v, w\}$:**
   - **Action:** Let v in, reject w, remove $N[v]$ from G, and decrease k.
   - **Reasoning:** In any valid solution where w is included, v can be included instead without making things worse.

Further Reduction

The additional ideas reduce the problem to a graph H with at most $|V| \leq k^2$ vertices.
Why?

$$|V| = \sum_{v \in V} 1 = \frac{1}{2} \sum_{v \in V} 2 = \frac{1}{2} \sum_{v \in V} d(v) = |E| \leq k^2$$

At this stage, you can try all subsets of k vertices from H:

$$\binom{k^2}{k} \approx \left(\frac{10^2}{10}\right)^{10} \approx 1.73 \cdot 10^{13}.$$

Using these ideas and combining them into the algorithm.

Algorithm 3 BarFightPrevention

Input: k, G
Output: A solution C or "No solution"
$k', H, C \leftarrow$ **BFP-Kernel**(k, G) **if** *H has $\leq (k')^2$ edges* ***and BFP-Brute-Force***(k', H) *returns a solution C'* **then**
  return $C \cup C'$
return *"No solution"*

Algorithm 4 BFP-Kernel

Input: k, G
Output: k', H, C
$k' \leftarrow k$, $H \leftarrow G$, $C \leftarrow \emptyset$ **while** ***true*** **do**
  if *Some v has $d(v) = 0$* **then**
    $H \leftarrow H \setminus \{v\}$
  else if $k' > 0$ ***and*** *some v has $d(v) > k'$* **then**
    $H \leftarrow H \setminus \{v\}$, $C \leftarrow C \cup \{v\}$, $k' \leftarrow k' - 1$
  else if $k' > 0$ ***and*** *some v has $N[v] = \{v, w\}$ for some w* **then**
    $H \leftarrow H \setminus N[v]$, $C \leftarrow C \cup \{w\}$, $k' \leftarrow k' - 1$
  else
    return k', H, C

Algorithm 5 BFP-Brute-Force

Input: $k, G = (V, E)$
Output: A solution C or "No solution"
for ***every subset*** $C \subseteq V$ ***of size*** k **do**
  if *C is a vertex cover of G* **then**
    return C
return *"No solution"*

The total running time of this algorithm is

$$\mathcal{O}(m + n + \binom{k}{k} k^2) \subseteq \mathcal{O}(m + n + k^{2k+2}) \subseteq \mathcal{O}_k(m + n)$$

6.2.3 Using bounded search tree

We see that for each edge $(u, v) \in E$, at least one of u, v must be rejected.
Thus we can pick an arbitrary edge (u, v), and recursively try with u rejected and with v rejected.
Using this to make the algorithm below.

Algorithm 6 BFP-Bounded-Search

Input: k, G
Output: A solution C or "No solution"

```
if G has no edges then
    return ∅
if k > 0 then
    Let (u, v) be an arbitrary edge of G  for w ∈ {u, v} do
        if BFP-Bounded-Search(k − 1, G \ {w}) returns a solution C then
            return C ∪ {w}
return "No solution"
```

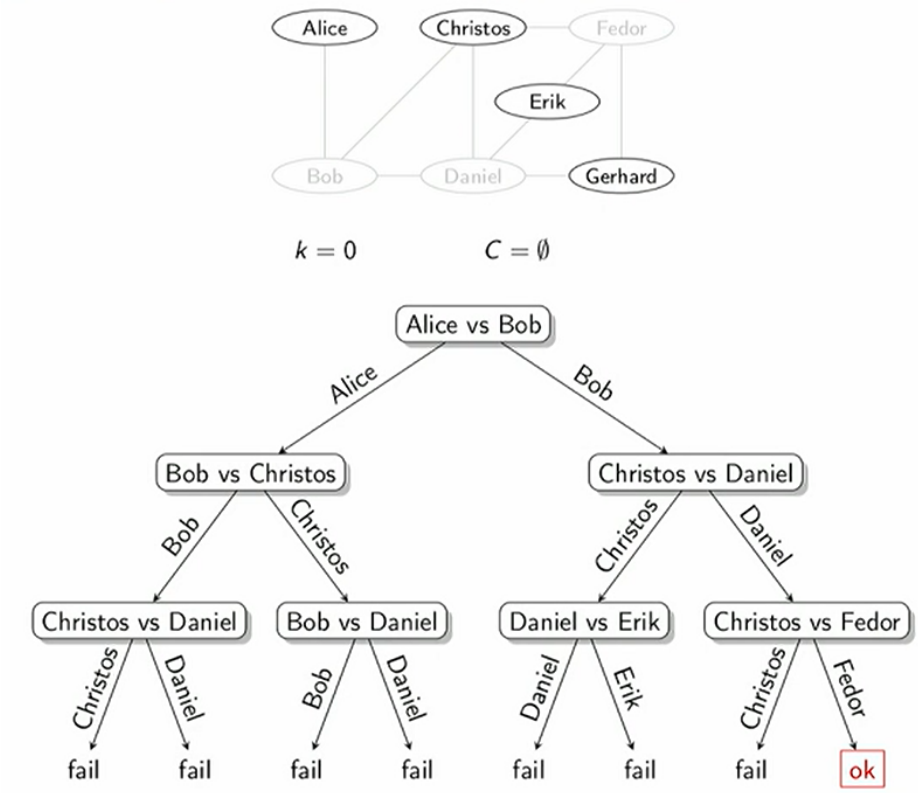
This recursive procedure has depth at most k. Thus the total number of subproblems considered at most 2^k.
If we start by rejecting all vertices of degree $d(v) > k$ (like in the kernelization approach), the resulting graph has at most

$$|E| = \frac{1}{2} \sum_{v \in V} d(v) \leq \frac{1}{2} nk$$

edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.
The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. Following the example from earlier, we get down to

$$1000 \cdot 10 \cdot 2^{10} \approx 10^7$$

We can even improve this by doing more/better kernelization.
An example of running the algorithm can be seen below in figure 32.



Figur 32: example of running the algorithm

To get a better idea of the algorithm, the full animations are in the slides.

6.3 Kernelization

Kernelization is a preprocessing technique in parameterized algorithms in which an input problem instance is transformed into an equivalent instance, called a kernel, whose size depends only on a parameter k, not the overall input size. The kernelization process ensures that the reduced instance can be solved efficiently, and the solution to the kernel maps back to the original problem.

6.4 Fixed Parameter Tractable (FPT)

A parameterized problem is *Fixed Parameter Tractable (FPT)* if it has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$. And that the problem is parameterized by the constant k

6.5 Slice-wise Polynomial (XP)

A parameterized problem is *Slice-wise Polynomial (XP)* if it has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.
We have that FPT $\subset$XP, since we can just set $g(k) = c$

6.6 Example: Vertex coloring

For an integer k, given a graph G does G have a proper vertex coloring with k colors?
Lemma
Unless $P = NP$, this problem is not XP and therefore not FPT.

Proof sketch.
The problem is NP-hard even for $k = 5$, so unless $P = NP$ there can be no algorithm for general k with running time $f(k) \cdot n^{g(k)}$.

6.7 Example: k-clique

For an integer k, given a graph G does G have a clique of size k?
Lemma
k-clique is XP.

Proof.
A simple brute-force algorithm is to check every k-subset of the vertices. There are $\binom{n}{k} \leq n^k$ such subsets, and we can check in $\mathcal{O}(k^2)$ time whether a given subset forms a clique. Thus the running time of this algorithm is $\mathcal{O}(k^2 \cdot n^k)$ which proves the problem is in XP.

It is unknown whether k-clique is FPT, but it is widely believed that $\mathcal{O}(n^k)$ is optimal which would prove it is not.

6.8 Example: Clique parameterized by Δ

For integer Δ, given a graph G with maximum degree Δ and an integer k, does G have a clique of size k?

Lemma
Clique is FPT when parameterized by the maximum degree Δ.

Proof.
If $k > \Delta + 1$, the answer is just no. Otherwise, a naive algorithm is for each vertex to try all subsets of its neighbors. There are at most $n \cdot 2^\Delta$ such subsets, and each can be checked in $\mathcal{O}(\Delta^2)$ time. The total time is thus $\mathcal{O}((2^\Delta \cdot \Delta^2) \cdot n)$, which proves the problem is FPT.

In fact, we can easily improve this algorithm to run in $\mathcal{O}((\binom{\Delta}{k-1}) \cdot k^2 \cdot n)$ time when $k \leq \Delta + 1$.
There are often many possible choices of parameter. Choosing the right one for a specific problem is an art.

7 van Emde Boas Trees

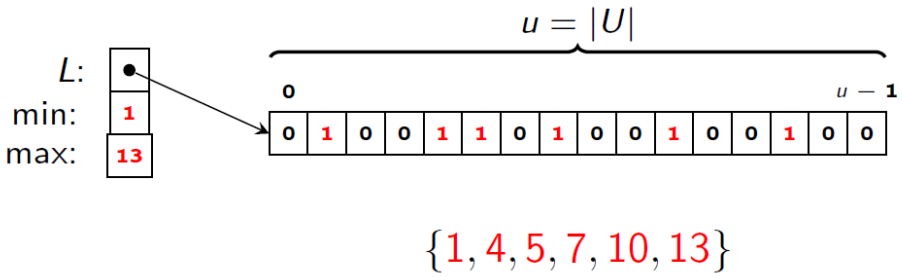
The problem is given as a universe $U = [u]$ where $u = 2^w$, maintain subset $S \subseteq U$, $|S| = n$ under:

- **member**(x, S): Return $[x \in S]$.
- **insert**(x, S): Add x to S (assumes $x \notin S$).
- **delete**(x, S): Remove x from S (assumes $x \in S$).
- **empty**(S): Return $[S = \emptyset]$.
- **min**(S): Return $\min S$ (assumes $S \neq \emptyset$).
- **max**(S): Return $\max S$ (assumes $S \neq \emptyset$).
- **predecessor**(x, S): Return $\max\{y \in S \mid y < x\}$
  (assumes $\{y \in S \mid y < x\}$ is nonempty,
  i.e. $S \neq \emptyset$ and $x > \min(S)$).
- **successor**(x, S): Return $\min\{y \in S \mid y > x\}$
  (assumes $\{y \in S \mid y > x\}$ is nonempty,
  i.e. $S \neq \emptyset$ and $x < \max(S)$).

Thus those are all the operations the data structure should be able to perform, and in a fast way as well.

7.1 Naive approach

If we are willing to spend $\mathcal{O}(|U|)$ space, then we can store S as a bit-array L of length $|U|$ such that $L[x] = [x \in S]$, and keep track of the min and max values explicitly.

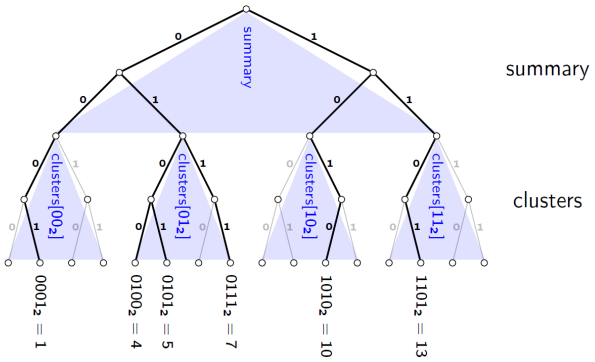


Figur 33: Example of the naive data structure

- `empty(S), min(S), and max(S):` worst case $\mathcal{O}(1)$ — `empty(S)` is tricky.
- `member(x, S):` worst case $\mathcal{O}(1)$.
- `predecessor(x, S) and successor(x, S):` worst case $\Theta(|U|)$.
- `delete(x, S):` worst case $\mathcal{O}(|U|)$ — need to update `min` / `max`.
- `insert(x, S):` worst case $\mathcal{O}(1)$.

7.2 Bit-Trie

The naive structure can be though of as just storing which leaves exist. An alternative naive choice would be to store the actual trie, which we could maintain in $\mathcal{O}(w) = \mathcal{O}(log|U|)$ time.



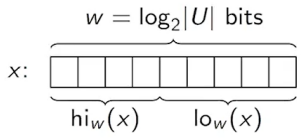
Figur 34: Example of using the trie.

7.3 Two-level

The idea is to split each key into high and low parts, and then use the naive approach for each. Each of those integers in the key can be seen as a bit string with length w, and thus we get

$$\begin{aligned} \text{hi}_w(x) &:= \left\lfloor \frac{x}{2^{\lceil w/2 \rceil}} \right\rfloor \\ \text{lo}_w(x) &:= x \mod 2^{\lceil w/2 \rceil} \\ \text{index}_w(h, \ell) &:= h \cdot 2^{\lceil w/2 \rceil} + \ell \end{aligned}$$

And that we have $x = \text{index}_w(\text{hi}_w(x), \text{lo}_w(x))$, where x is further illustrated below in figure 35.



Figur 35: x illustrated given a 9-bit universe.

Now let the structure directly store the values:

$$\text{min} := \begin{cases} \min(S) & \text{if } S \neq \emptyset, \\ 1 & \text{else.} \end{cases}$$

$$\text{max} := \begin{cases} \max(S) & \text{if } S \neq \emptyset, \\ 0 & \text{else.} \end{cases}$$

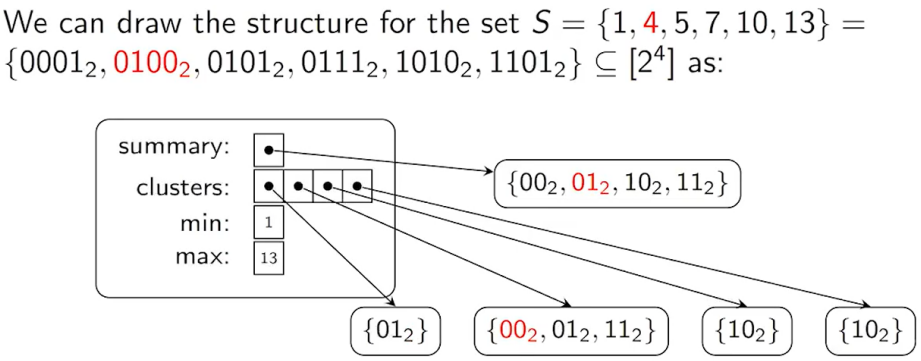
and use the naive structure to store

$$\text{summary} := \{\text{hi}_w(x) \mid w \in S\},$$

$$\text{clusters}[h] := \{\ell \in [2^{\lceil w/2 \rceil}] \mid \text{index}_w(h, \ell) \in S\}, \quad \forall h \in [2^{\lfloor w/2 \rfloor}],$$

$$\text{note that } S = \bigcup_{h \in [2^{\lfloor w/2 \rfloor}]} \{\text{index}_w(h, \ell) \mid \ell \in \text{clusters}[h]\}.$$

And thus we see that the high parts of the bit is stored in the summary and the low bits are stored in the cluster. This is much clearer in the illustration below in figure 36.



Figur 36: Illustration of the two-level data structure. The red numbers are highlighted as an example of storing 4 with the high bits 01 which are thus stored in the summary, and the low bits 00 stored in the cluster. (mistake in illustration, last cluster should have been 01).

- `empty(S), min(S), max(S):` $\mathcal{O}(1)$
- `member(x, S):` $\mathcal{O}(1) + 1 \times \text{naive} = \mathcal{O}(1)$
- `predecessor(x, S), successor(x, S):`

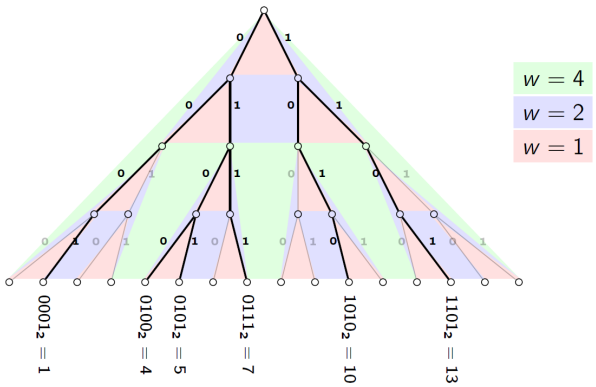
$$\mathcal{O}(1) + 1 \times \text{naive} = \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$$

- `delete(x, S):` $\mathcal{O}(1) + 2 \times \text{naive} = \Theta(\sqrt{|U|})$
- `insert(x, S):` $\mathcal{O}(1) + 2 \times \text{naive} = \mathcal{O}(1)$

This uses the naive data structure for the summary and cluster, and thus is limited by that.

7.4 Recursive two-level

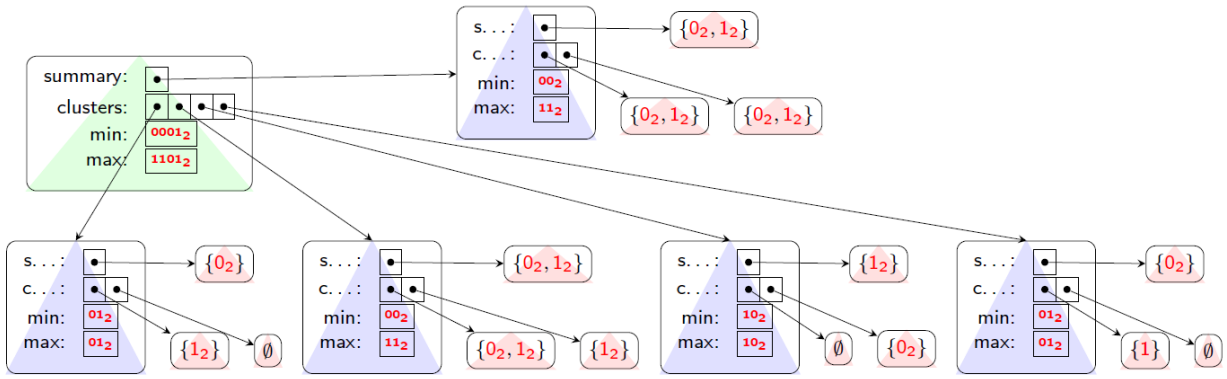
Instead of using the naive data structure for the summary and cluster part, we just use the two-level for those as well. This would give something like



Figur 37: Using recursion as the idea.

Where we stop the recursion when $w = 1$. Using the example from earlier below in figure 38.

We can draw the recursive structure for the set $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$ as:



Figur 38: Example of using the recursive two-level data structure.

This structure is also called "proto-vEB".
We further have the **theorem**:
The recursion depth of this structure, when used on the universe $U = [2^w]$, is $\lceil \log_2 w \rceil = \mathcal{O}(\log \log |U|)$. Proof in slides.

- `empty(S), min(S), and max(S):` worst case $\mathcal{O}(1)$.
- `member(x, S):` $\mathcal{O}(1) + 1 \times \text{recursion} = \mathcal{O}(d(w)) = \mathcal{O}(\log \log |U|)$.
- `predecessor(x, S) and successor(x, S):`

$$\mathcal{O}(1) + 1 \times \text{recursion} = \mathcal{O}(d(w)) = \mathcal{O}(\log \log |U|).$$

- `insert(x, S) and delete(x, S):`

$$\mathcal{O}(1) + 2 \times \text{recursion} = \Theta(2^{d(w)}) = \Theta(w) = \Theta(\log |U|).$$

7.5 vEB

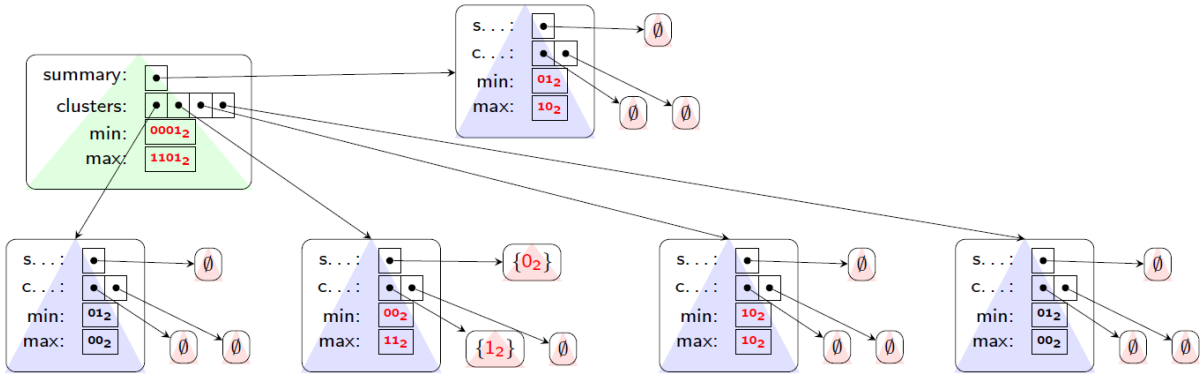
Now we use the same structure as the proto-vEB, but exclude $min(S)$ and/or $max(S)$ from the set of keys stored in summary and clusters. Such that the summary and cluster are now defined as

$$\text{summary} := \{\text{hi}_w(x) \mid x \in S \setminus \{\min, \max\}\}$$

$$\text{clusters}[h] := \{\ell \in [2^{\lceil w/2 \rceil}] \mid \text{index}_w(h, \ell) \in S \setminus \{\min, \max\}\}, \quad \forall h \in [2^{\lfloor w/2 \rfloor}]$$

Using the same example, the full van Embde Boas Tree would be

We can draw the van Emde Boas tree for the set $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$ as:



Figur 39: Example of the vEB tree

This makes sense, since we already have to keep something in the min/max, then it does not make sense to also store it in the summary/cluster. The first cluster is empty, and is thus denoted by having a min larger than the max.

7.5.1 Predecessor

$\texttt{PREDECESSOR}_w(x, S)$ takes worst case $\mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$ time. Since it makes at most one recursive call.

Algorithm 7 $\texttt{PREDECESSOR}_w(x, S)$

Input: x, S (Assumes $S \neq \emptyset$ and $x > S.\texttt{min}$)
Output: The predecessor of x in S

```
1 if x > S.max then
2     return S.max
3 if w = 1 then
4     return S.min
5 p ← hi_w(x)  s ← lo_w(x)  C ← S.clusters[p]  if not empty(C) and C.min < s then
6     return index_w(p, PREDECESSOR_⌈w/2⌉(s, C))
7 if empty(S.summary) or p ≤ S.summary.min then
8     return S.min
9 p ← PREDECESSOR_⌊w/2⌋(p, S.summary)  return index_w(p, S.clusters[p].max)
```

7.5.2 Insert

$\texttt{INSERT}_w(x, S)$ takes worst case $\mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$ time. Since it makes at most one recursive call on a non-empty substructure, and inserting in an empty substructure takes constant time.

Algorithm 8 $\texttt{INSERT}_w(x, S)$

Input: x, S (Assumes $x \notin S$)
Output: S updated with x inserted

```
1  if empty(S) then
2      S.min ← x, S.max ← x  return
3  if S.min = S.max then
4      if x < S.min then
5          S.min ← x
6      if x > S.max then
7          S.max ← x
8      return
9  if x < S.min then
10     S.min ↔ x
11 if x > S.max then
12     S.max ↔ x
13 p ← hi_w(x), s ← lo_w(x)  if empty(S.clusters[p]) then
14     INSERT_⌊w/2⌋(p, S.summary)
15     INSERT_⌈w/2⌉(s, S.clusters[p])
```

7.5.3 RS-vEB (reduced space)

The normal vEB structure uses $O(|U|)$ space. But if we use a hash table instead of an array to store the clusters, and just don't store the empty substructures we can get that down to $O(d(w)) = O(n \log\log(|U|))$. Since the empty structure uses $\mathcal{O}(1)$ space and INSERT_w only creates or updates $\mathcal{O}(d(w))$ substructures in the worst case. Each of these costs at most an additional $\mathcal{O}(1)$ space.

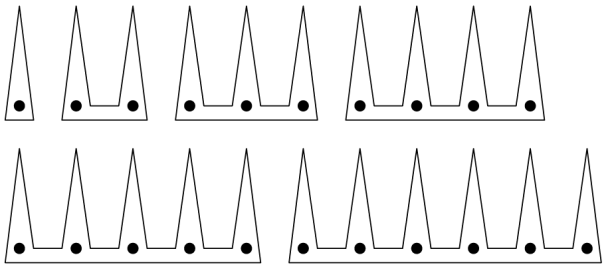
7.5.4 R^2S-vEB

This takes it down to $O(n)$ space.
Partition S into "chunks" of size $\Theta(\min\{n, \log\log|U|\})$, and store only one element representing each chunk in the RS-vEB structure. I.e., RS-vEB stores only $\mathcal{O}(\max\{1, n/\log\log|U|\})$ elements, using $\mathcal{O}(n)$ space.
We can store each chunk as a sorted linked list, and keep a hash table mapping each representative to its chunk. This also takes $\mathcal{O}(n)$ space.
Predecessor and successor use the RS-vEB structure to find the nearest two representatives in $\mathcal{O}(\log\log|U|)$ time, and can then spend linear time in the size of the two chunks to find the result.
Insert may have to split a chunk that becomes too large and insert a new representative in the RS-vEB structure. Splitting the chunk can take linear time in the size of the chunk, and together with inserting the new representative into the RS-vEB structure, this still only takes expected $\mathcal{O}(\log\log|U|)$ time.
Similarly, delete may have to join two chunks and delete a representative, but again this only takes expected $\mathcal{O}(\log\log|U|)$ time.

8 Polygon Triangulation

8.1 Art gallery problem

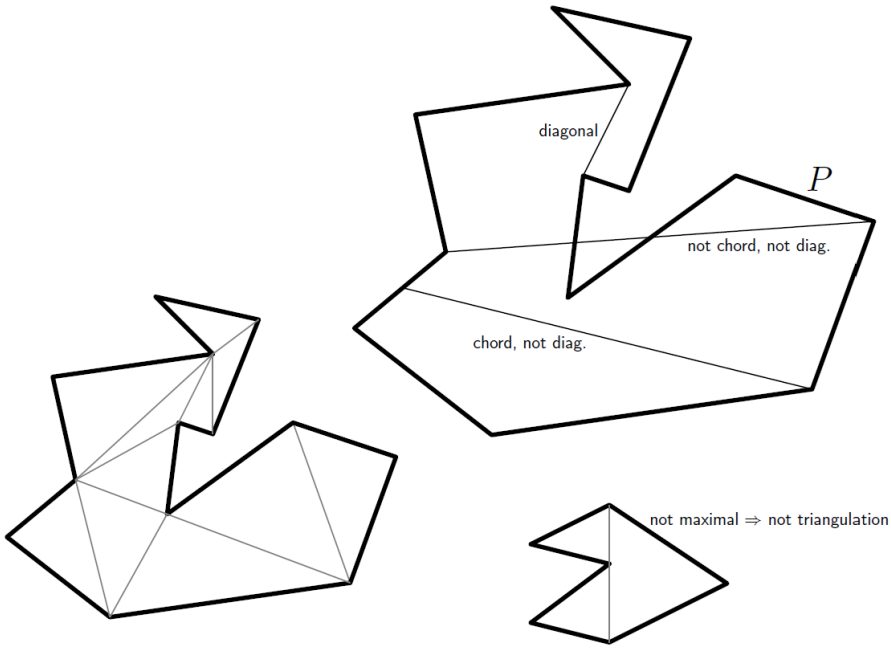
The polygon triangulation is often referred to as the art gallery problem, where we are trying to put as few guards in the art gallery as possible, while still being able to see the whole gallery.
Theorem:
$\lfloor n/3 \rfloor$ guards are occasionally necessary but always sufficient. Where n refers to the number of corners.



Figur 40: Example of needing $n/3$ guards.

8.2 Triangulation

A triangulation is a partition P into triangles by a maximum set of non-intersecting diagonals. Where a **diagonal** is defined as a segment contained in P between to vertices. See example below in figure 41.

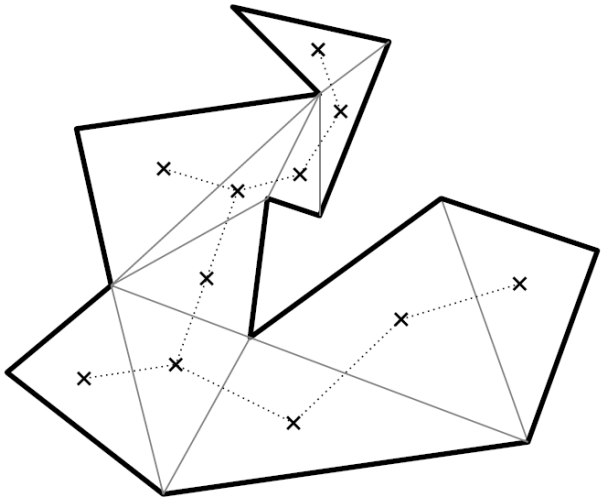


Figur 41: Example of triangulation

Lemma: A polygon P with n vertices can be triangulated, and any triangulation has $n-2$ triangles using $n-3$ diagonals.
Proof is in slides by induction, and by casing on if the line between two vertices is a diagonal or not.

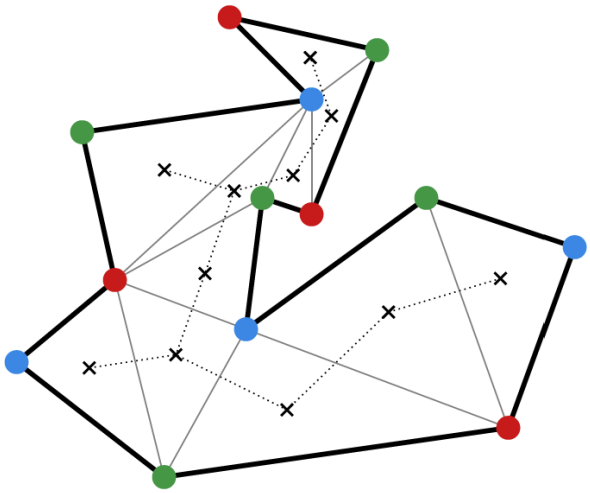
8.2.1 Proof for $\lfloor n/3 \rfloor$

The dual graph of a triangulated polygon is a tree. Each "face"(triangle within the triangulated polygon) gets a node, and two nodes are connected if the triangles are adjacent. See illustration below



Figur 42: Dual graph example.

We then see that a **3-coloring** always exists, in which each vertice of a single triangle in the triangulation gets a different color using only 3 colors. As illustrated below



Figur 43: 3-coloring way of the triangulated polygons vertices

We then see that if we pick just any of the 3 colors, then all vertices of the same color will combined cover, since each triangle is covered by at least one of that color, and since the whole polygon is triangulated it covers it all, and thus

$$n = n_r + n_g + n_b \implies$$

$$\min\{n_r, n_g, n_b\} \leq \frac{n}{3} \implies$$

$$\min\{n_r, n_g, n_b\} \leq \left\lfloor \frac{n}{3} \right\rfloor$$

The first equation comes from the pigeonhole argument, as the total number of vertices must be equal to the total number of blue + green + red vertices.

8.2.2 Simple algorithm

A simple algorithm that runs in $\mathcal{O}(n^2)$

Algorithm 9 `TRIANGULATE_POLYGON(`P`)`

Input: Polygon P
Output: Triangulation of P

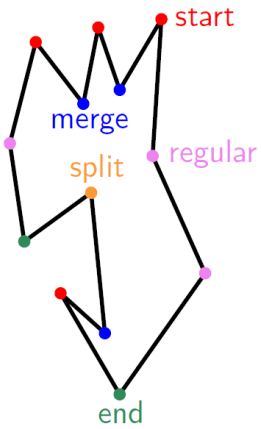
Find points u, v, w
if *uw is a diagonal* **then** `// Case 1`
    Add diagonal uw Recurse on the other side of uw
else
    `// Case 2`
Find point t Add diagonal vt Recurse on P_1 and P_2

8.3 Partion into y-monotone polygons

A polygon is y-monotone iff any horizontal line intersects it in a connected set (or not at all).
The idea is then to use a plane sweep to partition the polygon into y-monotone polygons, and then triangulate each of them.

A y-monotone polygon has a top vertex, a bottom vertex and two y-monotone chains on the left and right side. Any simple polygon with one top vertex and one bottom vertex is y-monotone.

Types of vertices for a polygon:
Start
Edges go down, and the polygon is below.
Merge
Edges go up, and the polygon is below.
Split
Edges go down, and the polygon is above.
End
Edges go up, and the polygon is above.
Regular
one edge goes up and one goes down.

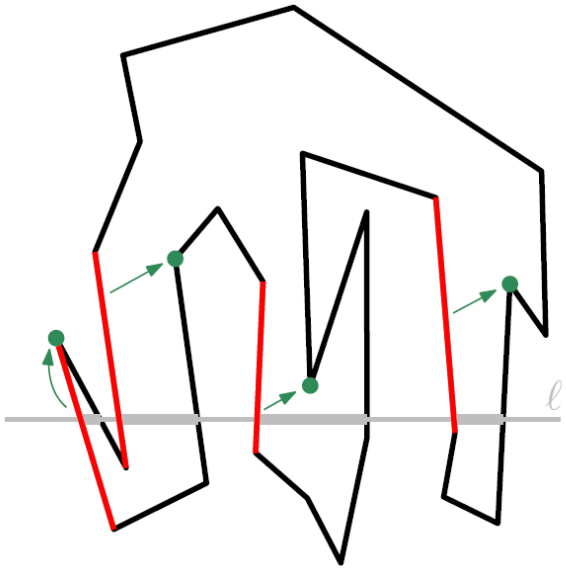


Figur 44: Types of vertices.

We then sweep down and do different things for each of the different vertex types.
The idea is to make segment to previous vertex visited by the component of $\ell \cap P$.

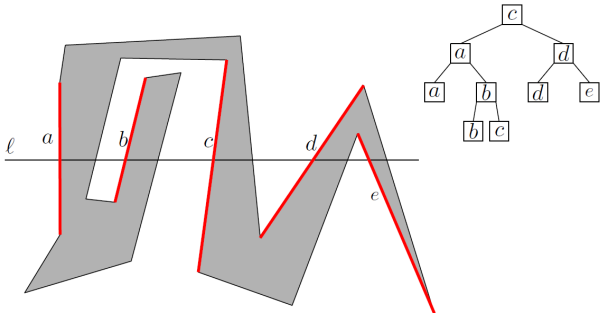
Helpers

Helper of edge e intersected by ℓ with the polygon interior of P to the right: Previous vertex visited by this connected component of $\ell \cap P$.
Equivalent: Lowest vertex above ℓ that sees e to the left.



Figur 45: Example of helpers

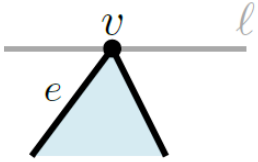
The status of the helper edges are managed by a balanced binary search tree T. Whenever we find a new vertex, we can look up in T the most immediate vertex to the left (in $\mathcal{O}(\log(n))$).



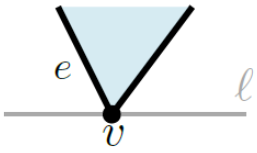
Figur 46: Example of the status.

The **events** that can happen is then:

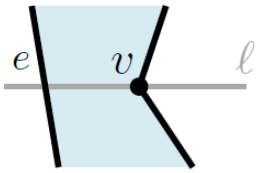
Start vertex: Insert e in T with helper v.



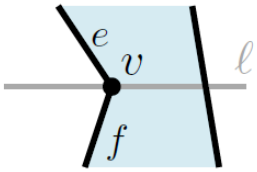
End vertex: Remove e from T.



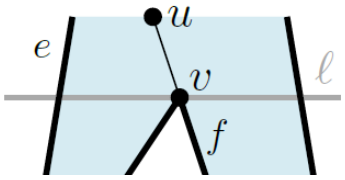
Regular vertex with P to the left: Update helper of e to v.



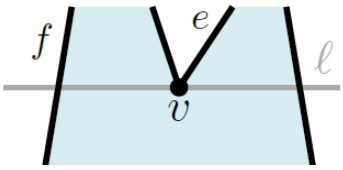
Regular vertex with P to the right: Replace e by f in T with helper v.



Split vertex: Add diagonal to helper u of e. Update helper of e to v. Add f to T with helper v.



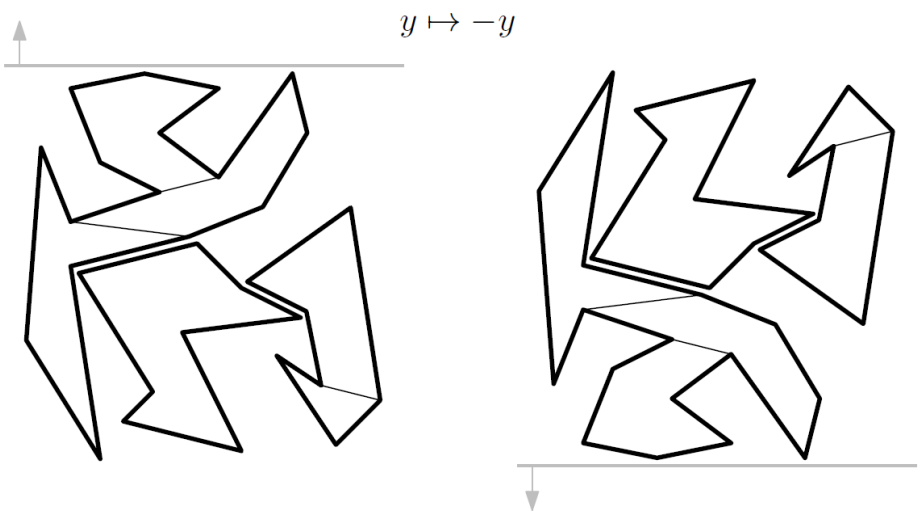
Merge vertex: Remove e from T. Update helper of f to v.



Figur 47: Events of sweeping

Sorting all events by y-coordinate takes $n\log(n)$ time and each event takes $\log(n)$ (because of inserting/-deleting in T).

After sweeping once, take the diagonals, and split the polygon on those. And then swipe up and down until all polygons are y-monotine.



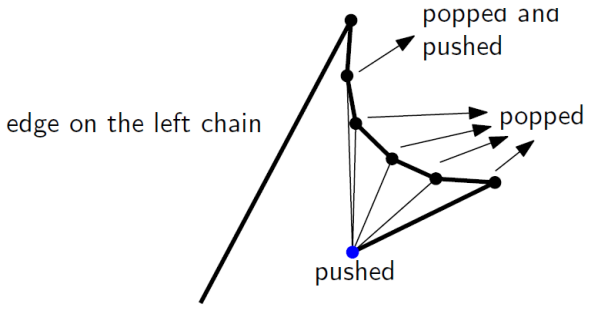
Figur 48: Splitting the polygons after sweeping.

In the end we get 2 extra vertices per diagonal and $\leq n-3$ diagonals $\implies \leq n + 2(n-3) = 3n - 6$ vertices.$A simple polygon with n vertices takes O(n \log(n))$ time to be partioned into y-monotone polygons

8.4 Triangulate y-monotone polygon

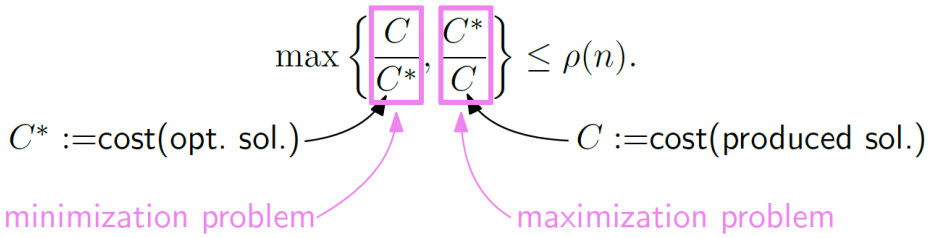
1. Merge vertices of left and right chain to get sorted order.
2. Traverse vertices top down and create diagonals to all possible vertices above.

Case 1: New vertex on the same chain (here right).

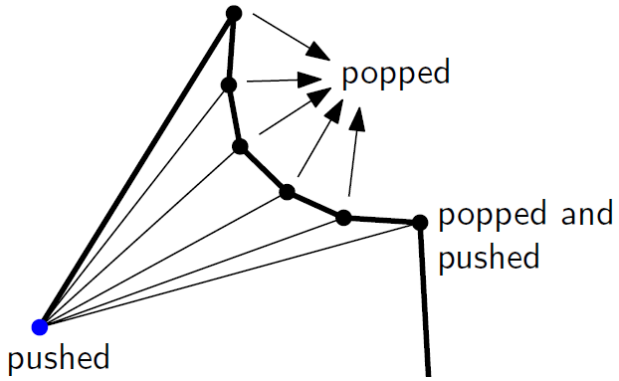


Figur 49: Case 1

Case 2: New vertex on the other chain.



Figur 51: Approximation ratio.



Figur 50: Case 2

Note: In either case, we push two vertices $\implies \leq 2n$ pushes, pops and diagonal checks.
Time complexity: $O(n)$ for merging and traversing.
n: number of vertices of this y-monotone polygon.

9 Approximation algorithms

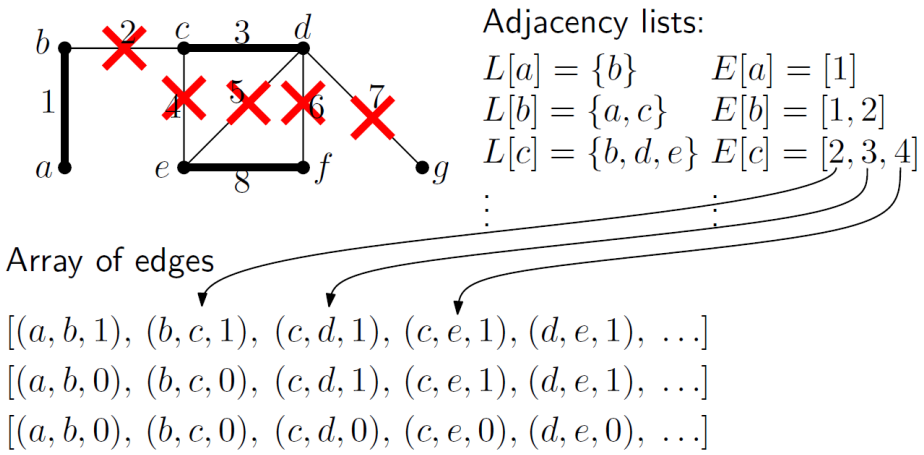
Good when a suboptimal solution is okay.
Is defined as an algorithm for an optimization problem has *approximation ratio* $\rho(n)$ if for every input of size n,

9.1 Approx vertex cover

Let $G = (V, E)$ be a graph. A set $V' \subseteq V$ of vertices is a *vertex cover* if for all $uv \in E$, we have $u \in V'$ or $v \in V'$.
NP-hard!

Algorithm 10 `APPROX-VERTEX-COVER(`G`)`

$C \leftarrow \emptyset$
while $E(G) \neq \emptyset$ **do**
  Choose $uv \in E(G)$
  $C \leftarrow C \cup \{u, v\}$
  Remove all edges incident on u or v from $E(G)$
return C



Figur 52: Example of the `APPROX-VERTEX-COVER(G)`

The time complexity is $\mathcal{O}(|V| + |E|)$.

Theorem: `APPROX-VERTEX-COVER` is a 2-approximation algorithm.
The proof uses the fact that two endpoints never share an edge, and thus size of the vertex cover is at most a ratio 2 from the optimal solution.

9.2 Method for proving approximation ratio

How can we prove $\frac{C}{C^*} \leq 2$ when we don't know C^*? **Answer:** By proving $C \leq 2|A|$ and $|A| \leq C^*$.

General technique: Find a parameter $\square$ such that $C \leq \rho \cdot \square$ and $\square \leq C^*$.

For vertex cover: $\square = |A|$ and $\rho = 2$.

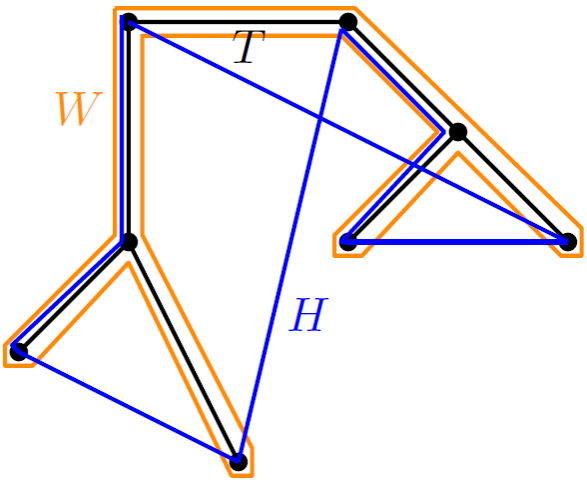
9.3 Approximate Traveling Salesperson (TSP)

Given a complete undirected graph $G = (V, E)$.
For all $u, v \in V$, we are given $c(uv) \in \{0, 1, \ldots\}$.

Goal: Find minimum weight cycle through all vertices.
Assume: Triangle inequality: $c(uw) \leq c(uv) + c(vw)$.

Algorithm 11 APPROX-TSP(G, c)

Find MST T
Make Euler tour W using each edge of T twice
Shortcut W to H by skipping duplicates
return H



Figur 53: Example of running the APPROX-TSP. The yellow line is the Euler tour and the blue is the one made by the algorithm.

Theorem: APPROX-TSP is a poly-time 2-approx. alg.
By proving $c(H) \leq 2c(T)$ and $c(T) \leq c(H^*)$.

9.4 Greedy Set cover

Input: Pair $(X, \mathcal{F})$, where X is a finite set and $\mathcal{F} \subseteq \mathcal{P}(X)$ is a family of subsets of X.
Goal: Find $\mathcal{C} \subseteq \mathcal{F}$ covering X, i.e., $\bigcup_{S \in \mathcal{C}} S = X$, with $|\mathcal{C}|$ minimum.

```
GREEDY-SET-COVER(X, F)
  i := 0
  while X \ S_{<i+1} ≠ ∅
    i := i + 1
    Pick S_i ∈ F with max |S_i \ S_{<i}|
  Return C := {S_1, ..., S_i}
```

Here, $S_{<i} := \bigcup_{j=1}^{i-1} S_j$.

Figur 54: Algorithm for the greedy set cover.

The greedy algorithm keeps picking the set that increases the cover the most.
Theorem:
For opt. sol. C^*, we have

$$|C| \leq H_{|X|} \cdot |C^*|,$$

where

$$H_n := \sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1.$$

Thus, Greedy-Set-Cover is a $O(\log n)$-approx. alg.
For $x \in S_i \setminus S_{<i}$, define

$$c_x := \frac{1}{|S_i \setminus S_{<i}|}.$$

For $Y \subset X$, define

$$c(Y) := \sum_{x \in Y} c_x.$$

We have

$$c(X) = \sum_{i=1}^{|\mathcal{C}|} \sum_{x \in S_i \setminus S_{<i}} c_x = \sum_{i=1}^{|\mathcal{C}|} 1 = |\mathcal{C}|.$$

Lemma: For all $S \in \mathcal{F}$:

$$c(S) \leq \sum_{i=1}^{|S|} \frac{1}{i} = H_{|S|}$$

$|\mathcal{C}| = c(X) \leq \sum_{S \in \mathcal{C}^*} c(S) \leq \sum_{S \in \mathcal{C}^*} H_{|S|} \leq \sum_{S \in \mathcal{C}^*} H_{|X|} = |\mathcal{C}^*| \cdot H_{|X|}$.

9.5 Greedy vertex cover

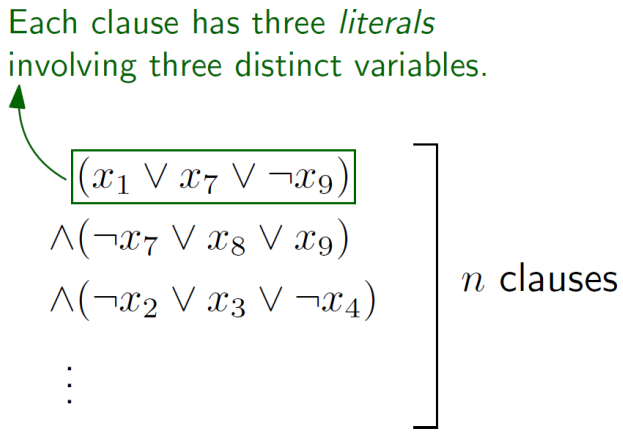
Gives $\Theta(\log |E|)$-approximation.

```
GREEDY-VERTEX-COVER(G)
  C := ∅
  while E ≠ ∅
    Choose v ∈ V of maximum degree
    C := C ∪ {u}
    Remove edges incident to u from E
  return C
```

Figur 55: Algorithm for the greedy vertex cover

9.6 3-SAT

The 3-SAT problems are given by n clauses, where each contains 3 literals combined with an "or". The decision problem is then to decide whether it evaluates to true. This is NP-complete.



Figur 56: Example of the 3-SAT problem

9.6.1 MAX-3-SAT: Random assignment

Theorem: RANDOM-ASSIGNMENT is a 8/7-approximation algorithm.

```
RANDOM-ASSIGNMENT(Φ)        → Φ is a MAX-3-SAT instance
  for each variable x_i of Φ
    choose x_i ∈ {0,1} by flipping fair coin
  return assignment
```

Figur 57: Algorithm of the MAX-3-SAT random assignment

APPROX-VERTEX-COVER(G)
  $C := \emptyset$
  while $E(G) \neq \emptyset$
    choose $uv \in E(G)$
    $C := C \cup \{u, v\}$
    remove all edges incident on u or v from $E(G)$
  return C

Figur 58: Approximate vertex cover

The proof of the theorem is essential getting the expected amount of true cluases, where each clause has a $7/8$ probability of being true, and thus taking the sum of this gives $7n/8$ and thus the ratio $8/7$.

$$\frac{\mathcal{C}^*}{\mathcal{C}} = \frac{\mathcal{C}^*}{7n/8} \leq \frac{n}{7n/8} = 8/7$$

9.7 Vertex cover

An approximation algorithm for vertex cover

9.7.1 Weighted vertex cover

Def.: Let $G = (V, E)$ be a graph. A set $V' \subseteq V$ of vertices is a vertex cover if for all $uv \in E$, we have $u \in V'$ or $v \in V'$.
We are also given weight $w(v) > 0$ for each $v \in V$.
Goal: Find vertex cover C with minimum

$$w(C) = \sum_{v \in C} w(v)$$

This can be defined as a $0-1$ integer program (IP)

$$\begin{aligned} & x_v \in \{0,1\}, \forall v \in V \quad (x_v = 1 \Leftrightarrow v \in C) \\ & x_u + x_v \geq 1, \forall uv \in E \quad (\text{edge } uv \text{ covered}) \\ & \text{minimize} \sum_{v \in V} w(v) x_v \end{aligned}$$

Thus it sets $x_v = 1$ if x_v is in the cover, and we have that at least one of x_u and x_v must be in the cover. This problem is NP-complete, but this can be reformulated to a linear programming problem: LP-relaxation: replace $x_v \in \{0, 1\}$ with $0 \leq x_v \leq 1$. Result:

$$\begin{aligned} & 0 \leq x_v \leq 1, \forall v \in V \\ & x_u + x_v \geq 1, \forall uv \in E \\ & \text{minimize} \sum_{v \in V} w(v) x_v \end{aligned}$$

The relaxed solution can be smaller, but not larger than the original problem.

APPROX-MIN-WEIGHT-VC(G, W):
  Compute opt. sol. $\bar{x}$ to LP
  return $C := \{v \in V \mid \bar{x}_v \geq 1/2\}$

Figur 59: Approximate min weight vertex cover algorithm

Theorem:
The algorithm is a polynomial-time 2-approximation algorithm for minimum-weight vertex cover.

9.8 PTAS

Polynomial-time approximation scheme (PTAS) is an approximation algorithm that takes instance I of an optimization problem P and $\varepsilon > 0$ as input. For any fixed ε works as $(1+\varepsilon)$-approximation algorithm for P.
Ex: Runtime $O\left(2^{1/\varepsilon} \cdot n^3\right)$ or $O\left(n^{1/\varepsilon}\right)$ or $O\left(n \log n/\varepsilon^2\right)$.
Smaller ϵ gives a higher runtime (but also a better approximation).

9.9 FPTAS

A fully polynomial-time approximation scheme (FPTAS) is a PTAS with runtime polynomial in $1/\varepsilon$ and the size of I.
Ex: Runtime $O\left(n \log n/\varepsilon^2\right)$.

9.10 Subset sum

Input: Set $S = \{x_1, \ldots, x_n\} \subset \mathbb{N}$, and $t \in \mathbb{N}$.
Goal: Find $U \subset S$ s.t. $\sum_{x \in U} x \leq t$ with maximum $\sum_{x \in U} x$.

In the earlier subset sum problem we wanted the sum to be exact, now we just want it as high as possible, but not exceeding the target value.
Example: $S = \{1, 4, 5\}, t = 8$.
NP-complete to decide if $\exists U \subset S : \sum_{x \in U} x = t$.

EXACT-SUBSET-SUM(S, t)
  $L_0 = [0]$
  for $k = 1, \ldots, n$
    L_k=MERGE-LISTS$(L_{k-1}, L_{k-1} + x_k)$
    remove from L_k duplicates and elements $> t$
  return last(L_n)

Figur 60: Exact subset sum algorithm

Example: $S = \{1, 4, 5\}$, $t = 8$.
$L_0 = [0]$
$L_1 = L_0 \cup (L_0 + 1) = [0] \cup [1] = [0, 1]$
$L_2 = L_1 \cup (L_1 + 4) = [0, 1] \cup [4, 5] = [0, 1, 4, 5]$
$L_3 = L_2 \cup (L_2 + 5) = [0, 1, 4, 5] \cup [5, 6, 9, 10] = [0, 1, 4, 5, 6]$

Figur 61: Example of the subset sum exact algorithm.

Running time: Computing $L_k : O\left(|L_{k-1}|\right)$.
Total: $O\left(\sum_{k=1}^{n} |L_k|\right) = O(nt) = O\left(n 2^{\log t}\right)$.
Thus it is polynomial in the input size, we can make an approximation algorithm as follows below.

```
TRIM(L = [s_1, ..., s_m], δ)
  L' = [s_1]
  for i = 2, ..., m
    if s_i > last(L') · (1 + δ)
      L' = L' ∪ [s_i]
  return L'
```

```
APPROX-SUBSET-SUM(S, t, ε)
  L'_0 = [0]
  for k = 1, ..., n
    L'_k=MERGE-LISTS(L'_{k-1}, L'_{k-1} + x_k)
    L'_k=TRIM(L'_k, ε/2n)
    remove duplicates and elm.s > t
  return last(L'_n)
```

Figur 62: Approximation algorithm for subset sum and with the trim algorithm.

For $\delta = 0.1$ we remove elements that are less than a tenth larger than the element.

Example: $L = [0, 9, 10, \cancel{11}, 12, \cancel{13}, 16]$, $\delta = 0.1$.

Figur 63: Example of trimming

Theorem:
The algorithm is an FPTAS.
Proof is in slides.

The approximation ratio is $(1+\delta)^n \leq 1+\epsilon$.
Total running time: $O\left(\sum_{k=1}^{n} |L'_k|\right) = O\left(\frac{n^2 \ln t}{\varepsilon}\right)$.

10 Extra

10.1 Harmonic sums

Definition

The **harmonic sum** H_n is defined as:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Key Properties

- **Asymptotic Growth:**
$$H_n \sim \ln(n) + \gamma$$
where γ is the Euler-Mascheroni constant ($\gamma \approx 0.577$).
- **Bounds:**
$$\ln(n) < H_n < \ln(n) + 1$$
- **Logarithmic Behavior:** Harmonic sums grow logarithmically, making them fundamental in analyzing algorithms with iterative divisions or inversions.

Applications

- **Algorithm Analysis:** Found in divide-and-conquer algorithms (e.g., quicksort) and data structure operations (e.g., heaps).
- **Amortized Analysis:** Used in evaluating the performance of operations in disjoint sets and other structures.
- **Prime Number Theory:** Related to the distribution of primes via its logarithmic growth.

10.2 Geometric sum

Definition

The **geometric sum** is defined as:

$$S_n = \sum_{k=0}^{n} ar^k = a + ar + ar^2 + \cdots + ar^n$$

where a is the initial term, r is the common ratio, and n is the number of terms.

Key Formulas

- **Finite Sum Formula:**
$$S_n = a\frac{1 - r^{n+1}}{1 - r}, \quad \text{for } r \neq 1$$
- **Infinite Sum:**
$$S_\infty = \frac{a}{1 - r}, \quad \text{for } |r| < 1$$

Properties

- **Convergence:** The infinite sum converges only if $|r| < 1$.
- **Growth Behavior:** For $r > 1$, the terms grow exponentially. For $0 < r < 1$, the terms decay geometrically.

Applications

- **Algorithm Analysis:** Commonly appears in analyzing the total work of algorithms with exponentially shrinking work (e.g., divide-and-conquer).
- **Amortized Analysis:** Used in geometric decay scenarios, such as halving-based operations in data structures.
- **Financial Models:** Widely used in calculating compound interest and annuities.

10.3 Inequalities

Inequality Name	Mathematical Formulation	Description
Triangle Inequality	$\lvert x+y\rvert \;\le\; \lvert x\rvert + \lvert y\rvert$	In any metric space or normed space, the distance (or norm) of a sum is at most the sum of the distances (norms).
Jensen's Inequality	If f is convex, then $f\big(\mathbb{E}[X]\big) \;\le\; \mathbb{E}[f(X)]$.	For a convex function f, the function value at the expectation is at most the expectation of the function.
Markov's Inequality	$\Pr\big(X \ge a\big) \;\le\; \dfrac{\mathbb{E}[X]}{a}, \quad a>0.$	Bounds the probability that a non-negative random variable exceeds a given threshold a.
Chebyshev's Inequality	$\Pr\big(\lvert X-\mu\rvert \ge k\,\sigma\big) \;\le\; \dfrac{1}{k^2}.$	Gives an upper bound on the probability that a random variable deviates from its mean by more than k standard deviations.
Hölder's Inequality	$\sum_i \lvert a_i\,b_i\rvert \;\le\; \Big(\sum_i \lvert a_i\rvert^p\Big)^{\frac{1}{p}} \Big(\sum_i \lvert b_i\rvert^q\Big)^{\frac{1}{q}}, \quad \dfrac{1}{p}+\dfrac{1}{q}=1.$	Generalizes Cauchy–Schwarz; relates sums in different L^p spaces.
Minkowski's Inequality	$\lVert x+y\rVert_p \;\le\; \lVert x\rVert_p \;+\; \lVert y\rVert_p.$	A "triangle inequality" for L^p norms; underpins many geometric and analytic results.
Cauchy–Schwarz Inequality	$\big\lvert\langle x,y\rangle\big\rvert \;\le\; \lVert x\rVert_2\,\lVert y\rVert_2.$	Fundamental inner-product inequality; special case of Hölder's with $p=q=2$.
Bernoulli's Inequality	$(1+x)^n \;\ge\; 1 \;+\; n\,x, \quad x\ge -1,\; n\ge 0.$	Gives a simple lower bound on $(1+x)^n$; useful in bounding or approximating exponentials.
AM–GM Inequality	$\dfrac{x_1+x_2+\cdots+x_n}{n} \;\ge\; \sqrt[n]{x_1x_2\cdots x_n}.$	Relates arithmetic mean and geometric mean; equality holds if and only if $x_1=x_2=\cdots=x_n$.
Young's Inequality	$a\,b \;\le\; \dfrac{a^p}{p}+\dfrac{b^q}{q}, \quad \dfrac{1}{p}+\dfrac{1}{q}=1,\; p,q>1.$	Useful for splitting products in analysis; closely related to Hölder's and convex duality.

Figur 64: Relevant inequalities

10.4 Probability rules

For a random variable X :

$$\begin{aligned} \mu_X &= \mathbb{E}[X] && \text{(expectation)} \\ \operatorname{Var}[X] &= \mathbb{E}\left[(X-\mu_X)^2\right] && \text{(variance)} \\ \sigma_X &= \sqrt{\operatorname{Var}[X]} && \text{(std. deviation)} \end{aligned}$$

Expectation of indicator variable X :

$$\mathbb{E}[X] = \Pr[X = 1]$$

Linearity of expectation:

$$\mathbb{E}\left[\sum_i X_i\right] = \sum_i \mathbb{E}\left[X_i\right]$$

Sum of pairwise indep. variances:

$$\operatorname{Var}\left[\sum_i X_i\right] = \sum_i \operatorname{Var}\left[X_i\right]$$

Union bound:

$$\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$$

Markovs Inequality: For $X \geq 0, t > 0$

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

Chebyshevs Inequality: For $t > 0$

$$\Pr\left[|X - \mu_X| \geq t\sigma_X\right] \leq \frac{1}{t^2}$$

10.5 Combinatorics

	With rep	Without rep
Ordered	n^r	$\frac{n!}{(n-r)!}$
Unordered	$\binom{n+r-1}{r}$	$\frac{n!}{r!\,(n-r)!}$

Figur 65: Formula for combinations.

10.6 Logarithmic rules

Logarithm Rules

$$\log_a xy = \log_a x + \log_a y$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y$$

$$\log_a x^n = n \log_a x$$

$$\log_a b = \frac{\log_c b}{\log_c a}$$

$$\log_a b = \frac{1}{\log_b a}$$

The following can be derived from the above rules.

$$\log_a 1 = 0$$

$$\log_a a = 1$$

$$\log_a a^r = r$$

$$\log_a \frac{1}{b} = -\log_a b$$

$$\log_{\frac{1}{a}} b = -\log_a b$$

$$\log_a b \log_b c = \log_a c$$

$$\log_{a^m} a^n = \frac{n}{m}, m \neq 0$$

Figur 66: Logarithmic rules

11 Example solutions

11.1 Hashing

Let H be a universal hash function that maps $[n]$ to $[n]$. What is the expected number of collisions when inserting k distinct elements into the hash table?

Expected number of collisions

A common way to count collisions in the context of hashing is by counting the number of *colliding pairs* (i, j) with $i < j$. If k distinct elements $x_1, x_2, \ldots, x_k$ are hashed, a collision occurs between the pair (x_i, x_j) if and only if $H(x_i) = H(x_j)$.

- **Step 1**: For each pair (i, j) with $1 \leq i < j \leq k$, let the indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{if } H(x_i) = H(x_j), \\ 0 & \text{otherwise.} \end{cases}$$

- **Step 2**: The total number of collisions X among these k items is then

$$X = \sum_{1 \leq i < j \leq k} X_{ij}.$$

- **Step 3**: We want $\mathbb{E}[X]$, the expected value of X. By the linearity of expectation,

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{1 \leq i < j \leq k} X_{ij}\right] = \sum_{1 \leq i < j \leq k} \mathbb{E}[X_{ij}].$$

- **Step 4**: Because the hash family is universal,

$$\mathbb{E}[X_{ij}] = \Pr[H(x_i) = H(x_j)] = \frac{1}{n}.$$

  This is true for *each* pair (i, j).

- **Step 5**: There are $\binom{k}{2} = \frac{k(k-1)}{2}$ such pairs. Therefore,

$$\mathbb{E}[X] = \sum_{1 \leq i < j \leq k} \frac{1}{n} = \binom{k}{2} \cdot \frac{1}{n} = \frac{k(k-1)}{2n}.$$

Hence, the expected number of collisions when inserting k distinct elements under a universal hash function is

$$\boxed{\frac{k(k-1)}{2\,n}}.$$

Figur 67: Enter Caption