AAD noter

Thomas Busk

21. januar 2025

Maximum flow

Source Is the starting vertex often denoted with s
Sink Is the ending vertex often denoted with
Capacity Is the amount of units that can be send through each edge per unit of time.

Flow network A flow network consist of a directed graph $G = (V, E)$ a sorce $s \in V$ a sink $t \in V \backslash \{s\}$, and a capacity function $c : V \times V \to \mathcal{R}$(maps vertex pairs to real numbers) such that $c(u, v) \geq 0$ for all edges in V. This means that if there is not an edge then the capacity is 0.
We require that G has no self lops and no anti parallel edges
Flow Is a function $V \times V \to \mathcal{R}$ such that the it withholds the capacity constraint.

Capacity constraint The flow is not negative and the flow can not be larger than capacity.
Flow conservation constraints the flow going out of a vertex must equal the flow going into that same vertex.

Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: For all $u \in V - \{s, t\}$, we require

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

Figur 1: Formular for capacity and flow conservation

The value of the flow Is defined as the netflow from the source. This means the flow from the source subtracted to the flow going to the source
Max flow Is a flow of maximum value
Ford-Fulkerson Is a method to find a maximum flow in a flow network. Runningtime is O(|E|·|f^*|), where f^* is a max flow.

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

Figur 2: Idea behind Ford-Fulkerson

Residual capacity Is how much capacity is left on an edge, formally defined as:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\ , \\ f(v,u) & \text{if } (v,u) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

Figur 3: Residual capacity

Residual network Is the same graph as G but where the edges are only the ones that have positive residual capacity. Meaning $G_f = (V, E_f)$ where $E_f = \{(u,v) \in V \times V | c_f(u,v) > 0\}$. G_f is a flow network with capacity function C_f. This flow network can have parallel edges.
Augmenting path is defined as: Given a flow network $G = (V, E)$ and a flow f, an *augmenting path* p is a simple path from s to t in the residual network G_f.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min{c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

Figur 4: Full Ford-Fulkerson method

A cut is a partition of the graph into subset S and T where the source is in the S side and the sink is in the T side
Flow accross cut The flow accross the cut is the flow going from S to T meaning if there are edges between the subsets then the sum of the flow between these edges.
Capaity of the cut Is the same as the flow accros the cut but with capacity.
Lemma 2 if you have any flow and any cut, then the flow is equal to the flow going across this cut.
Max flow-min cut theorem

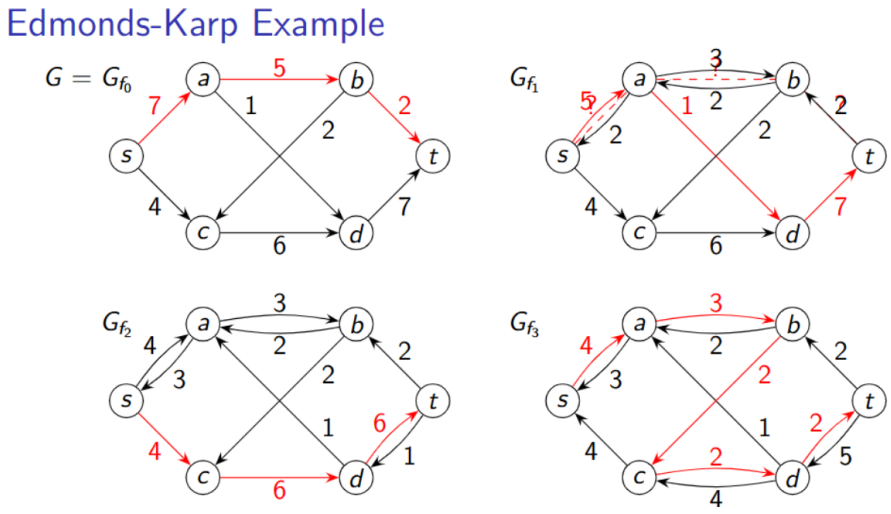
Theorem 26.6 (Max-flow min-cut theorem)
If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

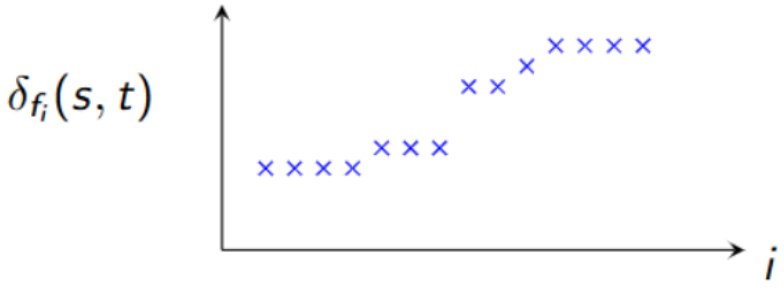
Figur 5: Max flow-min cut theorem

Edmonds-Karp is a instance of the Ford-Fulkerson method but where we have specified

how to pick the augmented path as the path with the least edges. We do a breath first search from source to sink and pick the one that reaches the sink first. This runs in $O(VE^2)$. This is because the number of augmentation that EK will make is $E \cdot V$

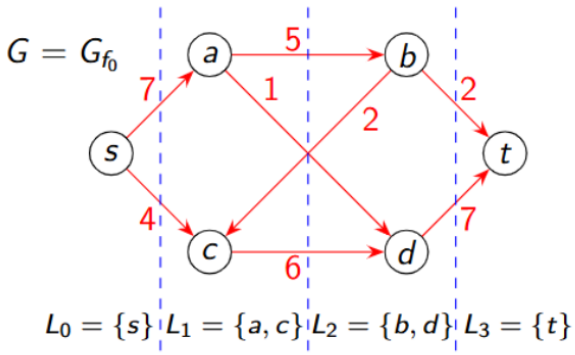


Figur 6: EK example



Figur 7: EK number of iterations

Layers in EK is defined as how far they are from the sink, in terms of when they were discovered in the BFS



Figur 8: Layers in EK

Forward edge Is an edge where the layers increase
Backwards edge Is the reverse of a forward edge.

1 Linear programming

Her mangler virkeligt meget skal læses meget op på og hvordan man løser det.

Standard form for LP can be used to minimize/maximise linear objective function subject to linear inequalities or equations
The variables are $\hat{X}$ (vector) and the objective function is denoted $\hat{c} \cdot \hat{x}$. The inequality $A \cdot \leq b$.
Formally this is:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \\ & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \\ & x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n \, . \end{aligned}$$

Figur 9: Standard form

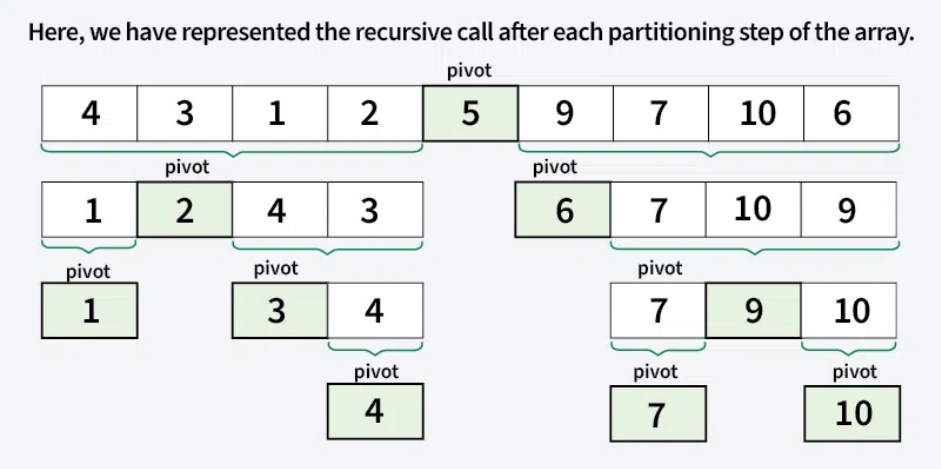
LP Duality
Weak duality
Strong duality
Simplex method
Ellipsoid method

2 Randomized Algorithms

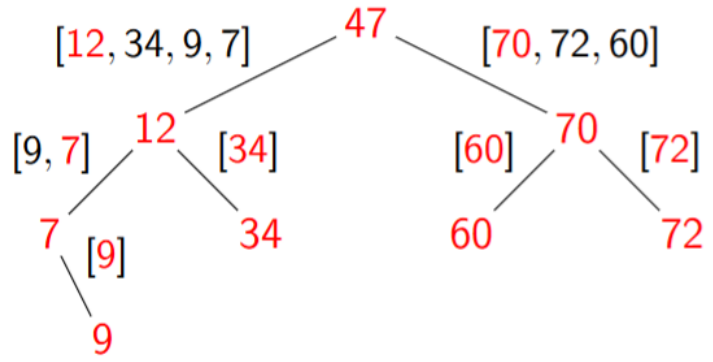
Randomized Quick sort is like quick sort but the pivot is chosen randomly.

```
1: function QS(S = {s_1, ..., s_n})
   ▷ Assumes all elements in S are distinct.
2:     if |S| ≤ 1 then
3:         return list(S)
4:     else
5:         Pick pivot x ∈ S, (How?)
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 QS(L)+[x]+QS(R)
```

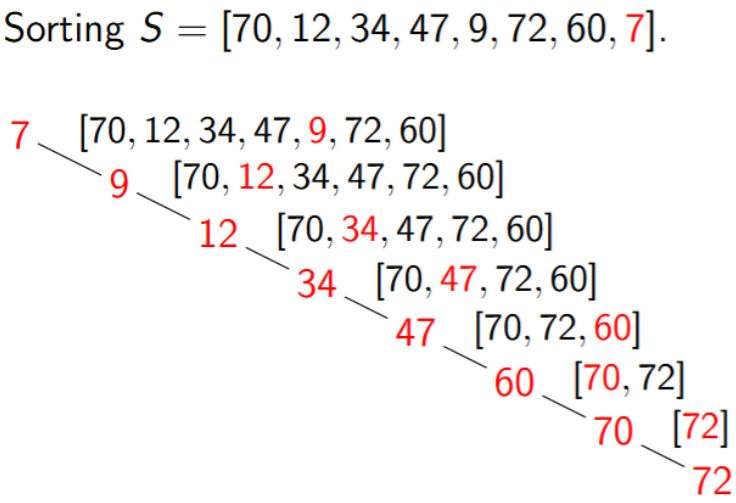
Figur 10: Pseudo code forrandomized quick sort



Figur 11: Normal quick sort example



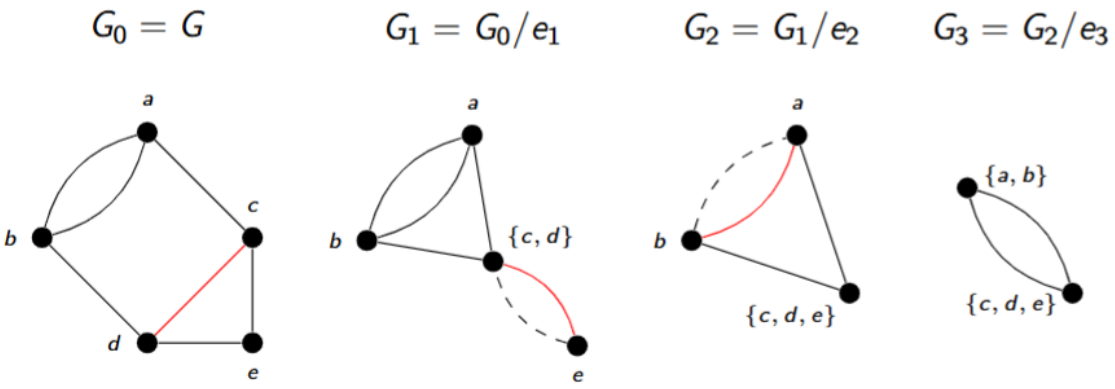
Figur 12: Lucky random QS example



Figur 13: Unlucky random QS example

n'th harmonic number is a number defined as $H_n = \sum_{k=1}^{n} \frac{1}{k}$
Randomized min-cut For a graph, contract an edge, remove self loops and then do it again.

```
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 14: Randomized min-cut code and example

Monte carlo algorithm Is a random algorithm where we know the runtime but we dont always get the right answer (run it many times and we probably get the correct answer)
Las vegas algorithm Is a random algorithm where we always get the right answer but we dont know the runtime only an expectation

3 Hashing

[H]

Universe is denoted U consists of keys that we can map to.
Random hash function is a chosen function that randomly maps a key to a value.

Notation:
For $n \in \mathbb{N}$:

$$[n] = \{0, \ldots, n-1\}$$
$$[n]_+ = \{1, \ldots, n-1\}$$

Iverson bracket:

$$[\text{condition}] = \begin{cases} 1 & \text{if condition is true} \\ 0 & \text{if condition is false} \end{cases}$$

For a random variable X:

$$\mu_X = \mathbb{E}[X] \quad \text{(expectation)}$$
$$\text{Var}[X] = \mathbb{E}[(X - \mu_X)^2] \quad \text{(variance)}$$
$$\sigma_X = \sqrt{\text{Var}[X]} \quad \text{(std. deviation)}$$

Inequalities:
Expectation of indicator variable X:

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

Linearity of expectation:

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

Sum of pairwise indep. variances:

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

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 $t > 0$

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

Figur 15: Notation used for hashing

Definition

A *random hash function* $h : U \to [m]$ is a randomly chosen function from some family of functions mapping U to $[m]$. Equivalently, it is a function h such that for each $x \in U$, $h(x) \in [m]$ is a random variable.

Figur 16: Random hash function definition

Truly random hash functio is defined if the variables h(x) for $x \in U$ are independent and uniform. This is however impractiacal as this requires to much space.
Universal hash function is when the probability of hashing two values into the same key is less than $\frac{1}{m}$
C-aprocimately hash function same as universal but with probability $\frac{c}{m}$
Strongly universal hash function For any two distinct elements in the universe the probability that x hashes tp q and y hashes to r is equal to $\frac{1}{m^2}$. This we can also describe as each key is hashed independently into [m] and two distinct keys hash independently

Definition
A hash function $h : U \to [m]$ is *truly random* if the variables $h(x)$ for $x \in U$ are independent and uniform in $[m]$.

Impractical, why? Space! Require $|U| \log_2 m$ bits to represent.

Definition
A random hash function $h : U \to [m]$ is *universal* if, for all $x \neq y \in U$: $\Pr_h[h(x) = h(y)] \leq \frac{1}{m}$.

Definition
A random hash function $h : U \to [m]$ is *strongly universal* (a.k.a. 2-independent) if,

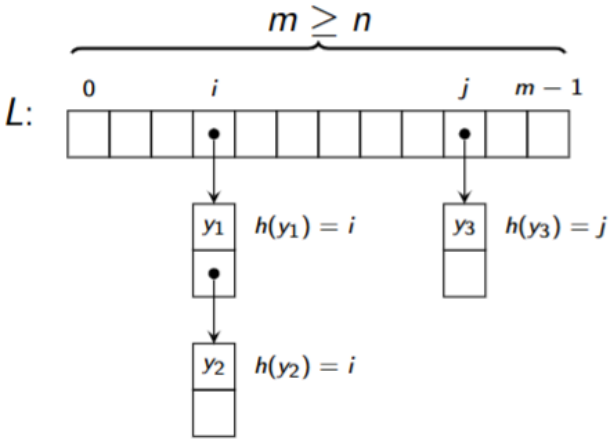
- Each key is hashed uniformly into $[m]$.
  (i.e. $\forall x \in U, q \in [m] : \Pr_h[h(x) = q] = \frac{1}{m}$)
- Any two distinct keys hash independently.

Or equivalently, if for all $x \neq y \in U$, and $q, r \in [m]$:
$\Pr_h[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}$.

Figur 17: Definitions for hashing

Hashing with chaining Is when we store our keys in a linked list, so if we want to see if x is in our datastucture, we simply hash x and look at the coresponding key, then we can only look there. Therefore insert, delete and lookup has a constant runtime.

Idea: Pick $m \geq n$ and a *universal* $h : U \to [m]$.
Store array L, where
$L[i] =$ linked list over $\{y \in S \mid h(y) = i\}$.



Figur 18: Hashing with chaining

Multiply-mod-prime hash function

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

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

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

Figur 19: Multiply-mod-prime hash function

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

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

Choose odd $a \in [2^w]$ uniformly at random, and let $h(x) := h_a(x)$.
Then $h : U \to [m]$ is a 2-approximately universal hash function.

(Assignment 3 exercise 3.4 asks you to show that it is not c-approximately strongly universal for any constant c).

Figur 20: Multiply-shifft hash function

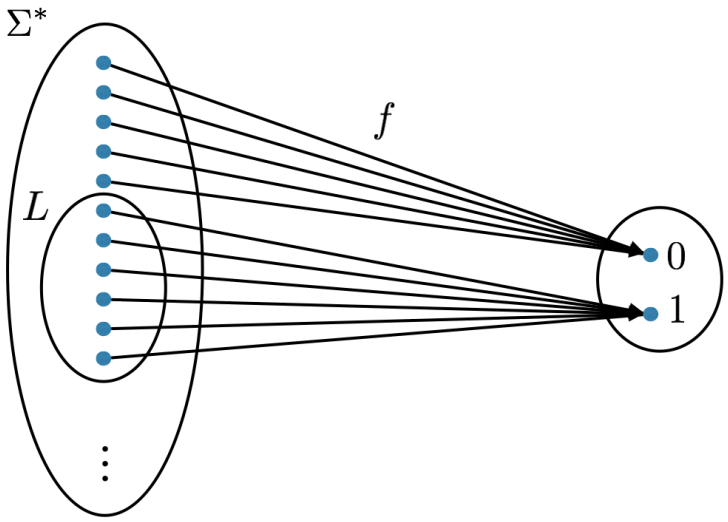
Coordinated sampling If two agent see the same occurence then they both either sample it or not sample it. This can be implemented with a hash function where all the agents has the same hash function and they chose a value t and if it hashes to a value lower than t then they sample it otherwise they throw it away.

4 NP-Completness

P Are problems we can solve in polynomial time
NP Are problems we can verify in polynomial time
NP-complete are the hardest problems in NP and all other NP problems can be reduced to that problem so if a soloution to a NP-complete problem can be found in polinomial time then we have solved the P=NP. **Decicion problem** is a problem where the answer is a yes and no. For example a decicion tree where we say we can come from u to v in k steps.
Optimization problem is where the output is not a yes no but a optimal soloution for example a shortest path.
Alphabet Finite set Σ of symbolx
Language L over Σ is a set of strings of symbols from Σ

- *Alphabet*: finite set Σ of symbols.
- *Language* L over Σ: a set of strings of symbols from Σ.
- Example: $\Sigma = \{a, b, c\}$ and $L = \{a, ba, cab, bbac, \ldots\}$.
- We also allow an empty string and denote it by ϵ.
- The empty language is denoted $\emptyset$ (it does not contain ϵ).
- Σ^* denotes the language of all strings (including ϵ).
- Any language L over Σ is a subset of Σ^*.

Figur 21: Definitions of language



Figur 22: If a string is in the alphabet then it is a yes instance of the decicion problem

This can also be written as:

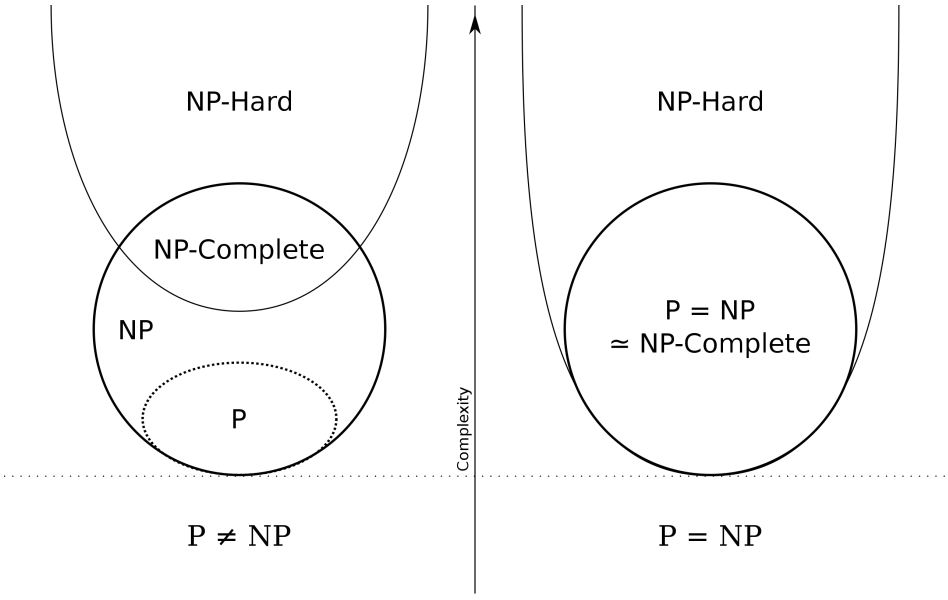
Language accepted/decided by an algorithm

- Let A be an algorithm for a decision problem and denote by $A(x) \in \{0, 1\}$ its output (if any) on input x.
- A *accepts* a string x if $A(x) = 1$.
- A *rejects* a string x if $A(x) = 0$.
- There may be strings that A neither accepts nor rejects.
- The language *accepted* by A is:

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

- Suppose in addition that all strings not in L are rejected by A, i.e., $A(x) = 0$ for all $x \in \{0, 1\}^* \setminus L$.
- Then we say that L is *decided* by A.

Figur 23: Language decided by algorithm



Figur 24: Enter Caption

Polinomial time reductible is if a language is reducible to another language with a polynomial time function

NP-complete languages

- Language L is *NP-complete* if
  1. $L \in$ NP and
  2. $L' \leq_P L$ for every $L' \in$ NP.
- L is *NP-hard* if property 2 holds (and possibly not property 1).
- The class of NP-complete languages is denoted NPC.
- If some language of NPC belongs to P then P = NP. Why?
- It is not immediately clear from the definition that NP-complete languages even exist.

Figur 26: Definitions of NP-complete languages

Polynomial-time reducibility

- Language L_1 is polynomial-time *reducible* to language L_2 if there is a polynomial-time computible function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

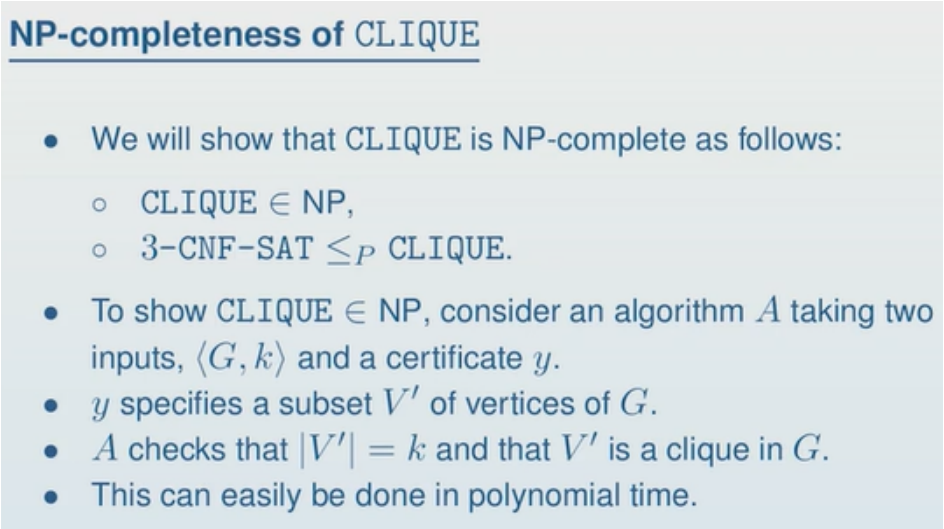
- In this case, we write $L_1 \leq_P L_2$.
- If $L_1 \leq_P L_2$ then L_1 is in a sense no harder to solve than L_2.
- More precisely,

$$L_1 \leq_P L_2 \wedge L_2 \in P \Rightarrow L_1 \in P.$$

- This follows since any instance I_1 of L_1 can be solved by transforming it in polynomial time to an instance I_2 of L_2 and then solving I_2 with a polynomial-time algorithm for L_2.

Figur 25: Polynomial-time reducibility

Prove something is NP-complete we have to prove first that it is NP and then that it can be reduced to from an already NP problem

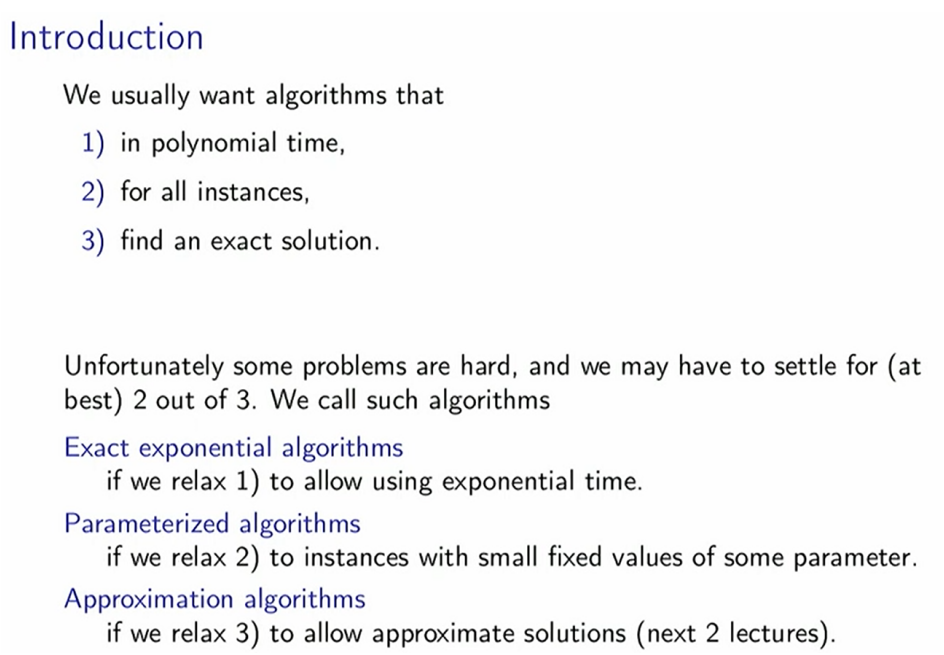


Figur 27: NP-Completeness of Clique

SÆT IND DE GENNEMGÅEDE NP PROBLEMS

4.1 Exact exponential algorithms and parameterized compexity

Motivation for lecture



Figur 28: Motivation for exponential algo

Traveling salesman via dynamic programming Is a way to solve the TSM in expo-

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \ldots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.
Idea: For all $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$ define

$$\boxed{\text{OPT}[S, c_i] := \begin{array}{l}\text{minimum length of any path that starts in } c_1, \text{ visits} \\ \text{all of } S \text{ once without leaving } S, \text{ and ends in } c_i.\end{array}}$$

Then $\min\{\text{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$ is the length of the minimal tour.

Lemma

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

Proof.

Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. □

Figur 29: TSM problem with dynamic programing

nential time and therefore not in n!.

Maximum independent set via Branching

Exact MIS via Branching

Problem: Given undirected graph (V, E), find the maximum cardinality of $I \subseteq V$ so each edge has at most one endpoint in I.

Such a set I is called a *Maximum Independent Set (MIS)* for the graph.

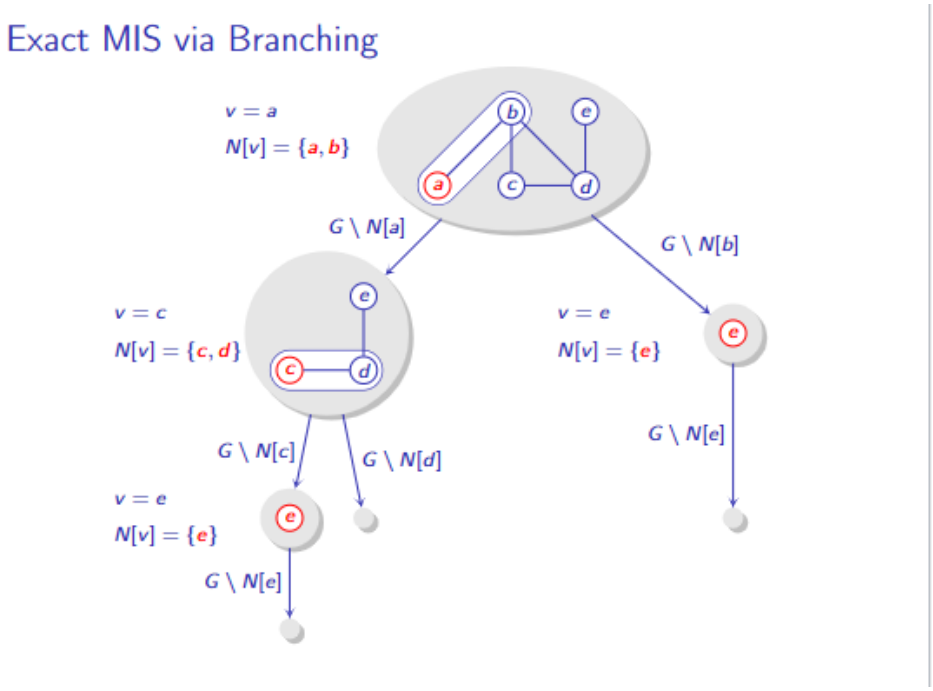
Naive: Try all 2^n subsets (where $n = |V|$). This takes $\mathcal{O}^*(2^n)$ time.

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.

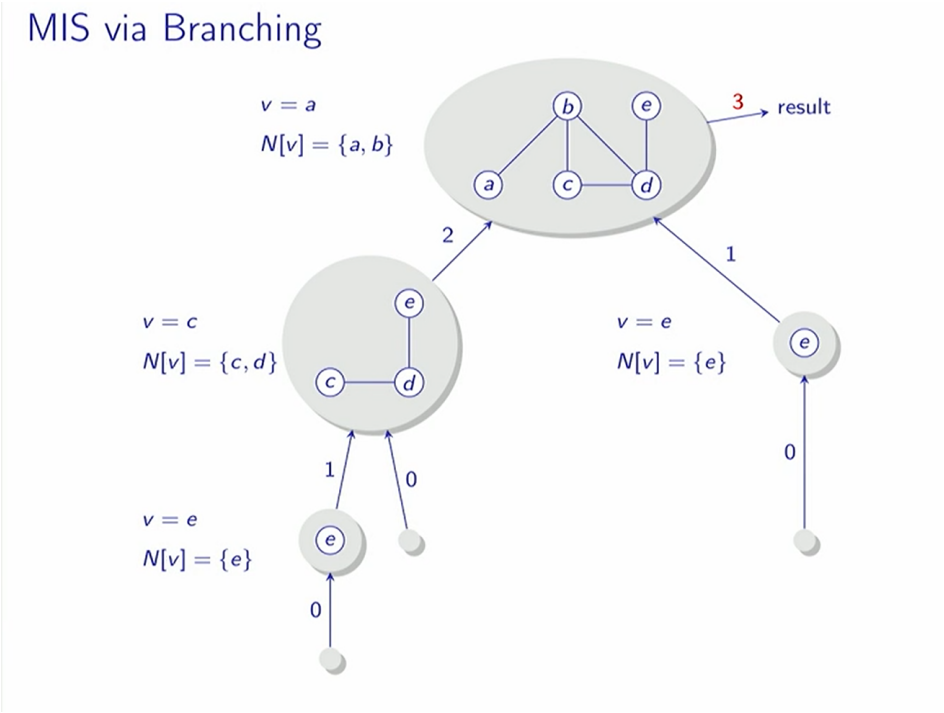
Observation: $N[v] \cap I \neq \emptyset$ for all $v \in V$ and all MIS I.
Why? If $N[v] \cap I = \emptyset$ for some $v \in V$, $I \cup \{v\}$ would be a larger solution.

```
1: function MISsize(G = (V, E))
2:     if V = ∅ then return 0
3:     v ← vertex in V of minimum degree.
4:     return 1 + max{MISsize(G \ N[w]) | w ∈ N[v]}
```

Figur 30: MIS via branching

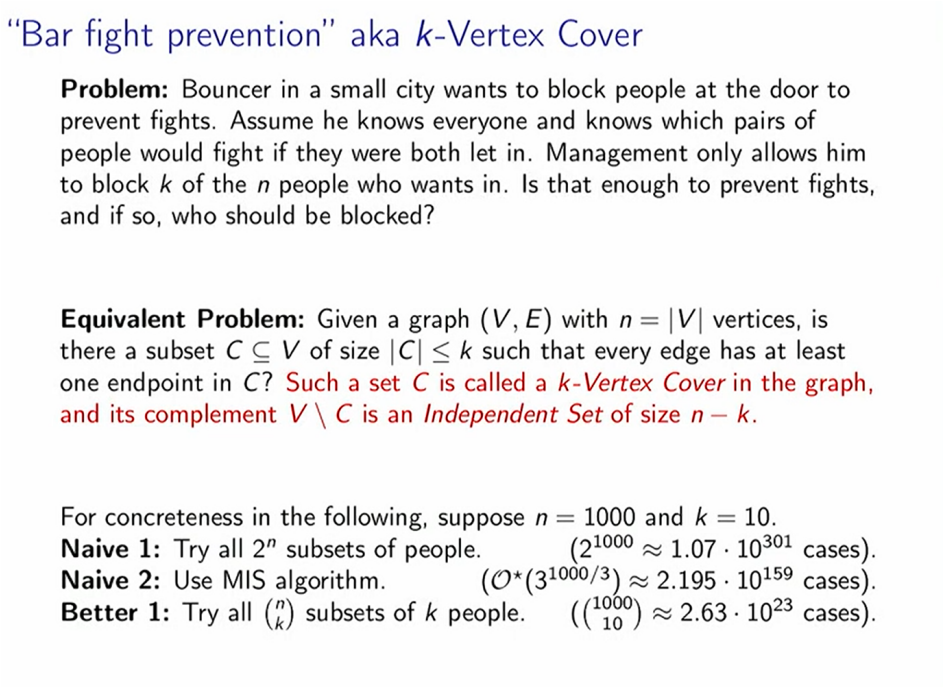


Figur 31: Mis example 1

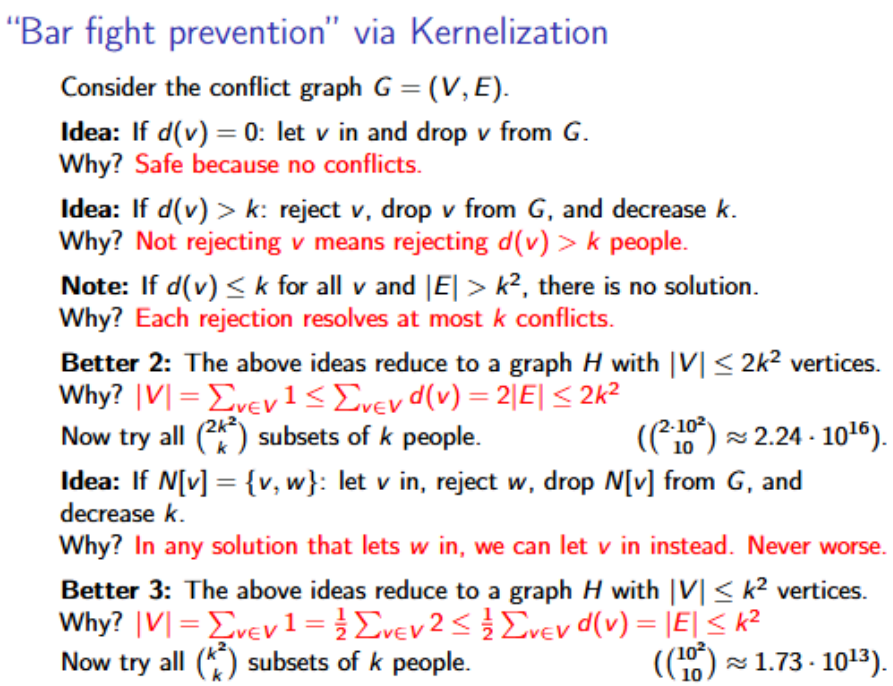


Figur 32: Mis example 2

Bar fight prevention (k-vertex cover)



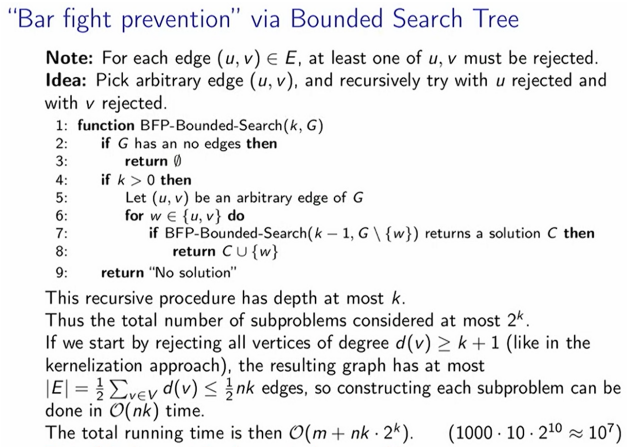
Figur 33: Bar fight prevention



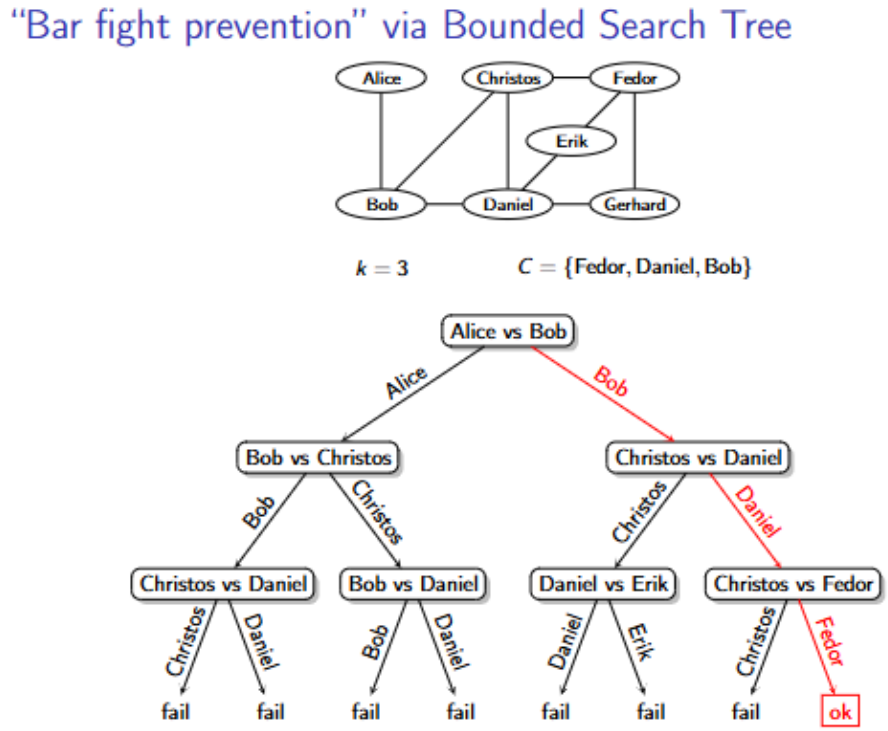
Figur 34: Bar fight prevention via kernalisation

Kernelization is removing the easy parts of a problem to reduce the size of the problem. Fore example in the bar fight then if a person has more than k conflicts he isnt a part of the

set and if a person has 0 conflicts then he is a part of the set.



Figur 35: Bar fight prevention via bounded search tree (is bounded because we reduce by k each time)



Figur 36: Bar fight prevention example bounded search tree

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

Definition: A parameterized problem is *Slice-wise Polynomial (XP)* if it is has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.

Note: FPT $\implies$ XP, why? Simply set $g(k) = c$.

Figur 37: Paremeterized problems definitions

5 van Emde Boas Trees

Reason for the tree

Problem:
Given a universe $U = [u]$ where $u = 2^w$,
maintain subset $S \subseteq U$, $|S| = n$ under:

insert(x, S): Add x to S (assumes $x \notin S$).

delete(x, S): Add x to S (assumes $x \in S$).

member(x, S): Return $[x \in S]$.

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).

successor(x, S): Return $\min\{y \in S \mid y > x\}$
(assumes $\{y \in S \mid y > x\}$ is nonempty).

Figur 38: Reason for van Emde Boas Tree

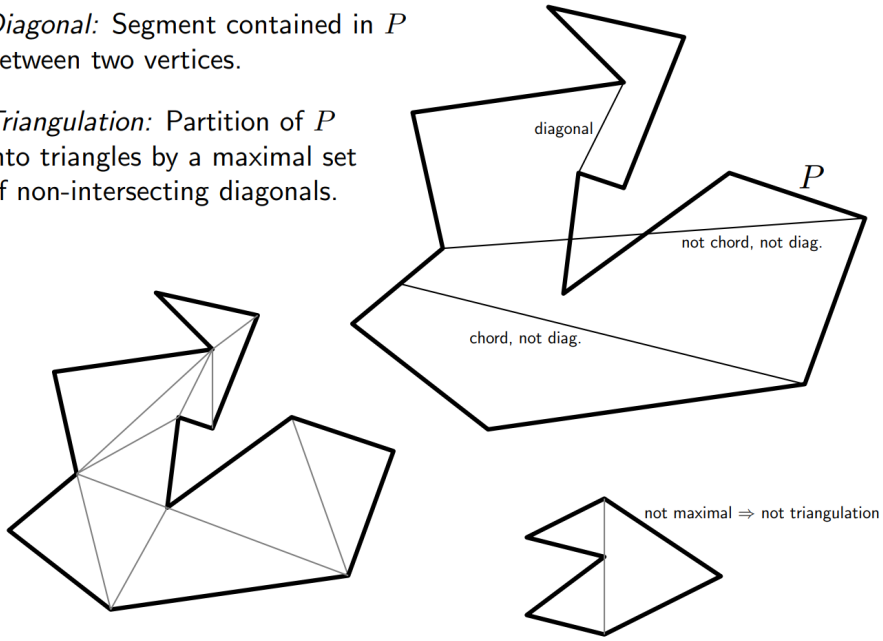
6 Computational Geometry

Art Gallery theorem the floor of $\frac{n}{3}$ is sometimes needed to cover a polygon but its always enough.

What is a triangulation?

Diagonal: Segment contained in P between two vertices.

Triangulation: Partition of P into triangles by a maximal set of non-intersecting diagonals.



Figur 39: Definition of triangulation

Any polygon can be triangulated

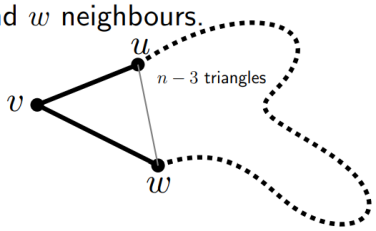
Lemma: A polygon P with n vertices can be triangulated, and any triangulation has $n-2$ triangles using $n-3$ diagonals.

Proof: Induction on n. Base case $n = 3$ is trivial.

Induction step: v leftmost vertex, u and w neighbours.

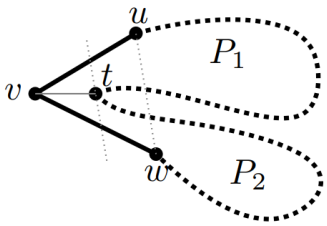
Case 1: uw is a diagonal.

Induction hypothesis ⇒ triangulation with $n-3$ triangles on the other side of uw.

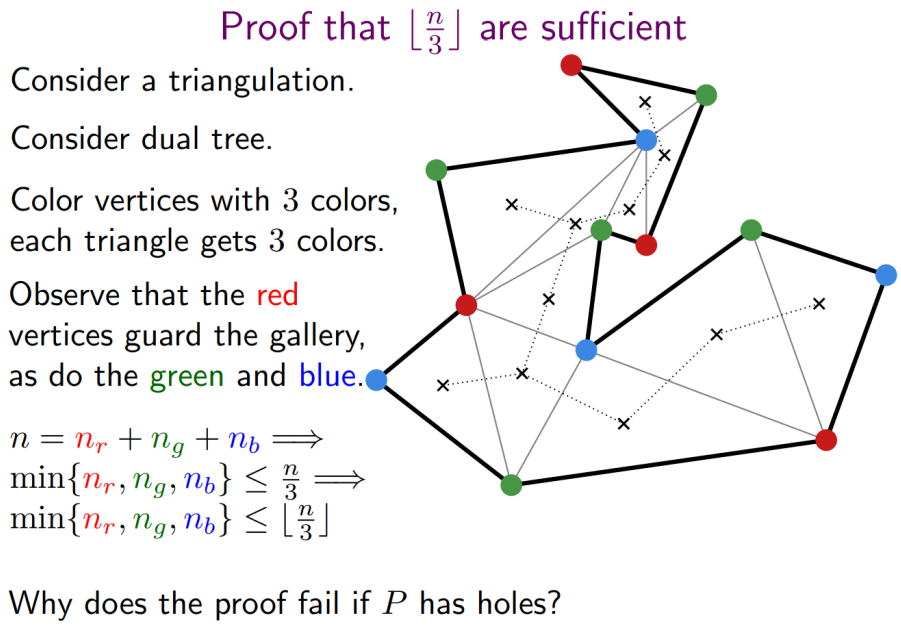


Case 2: uw is no diagonal. Let t be corner in uvw farthest from uw. vt is a diagonal, splits P into P_1 and P_2 with $m_1 < n$ and $m_2 < n$ vertices, $m_1 + m_2 = n + 2$.

Induction hypothesis ⇒
P_1: $m_1 - 2$ triangles.
P_2: $m_2 - 2$ triangles.
In total: $m_1 + m_2 - 4 = n + 2 - 4 = n - 2$.

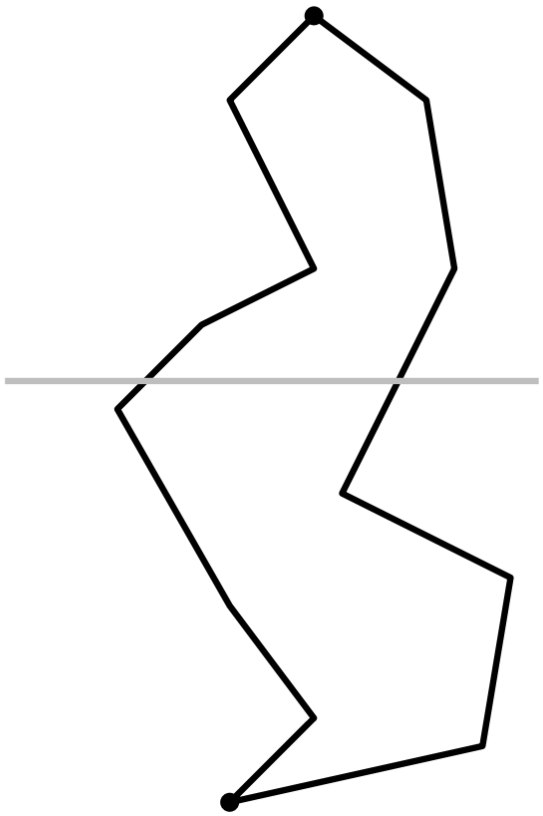


Figur 40: Proof that every polygon can be triangulated



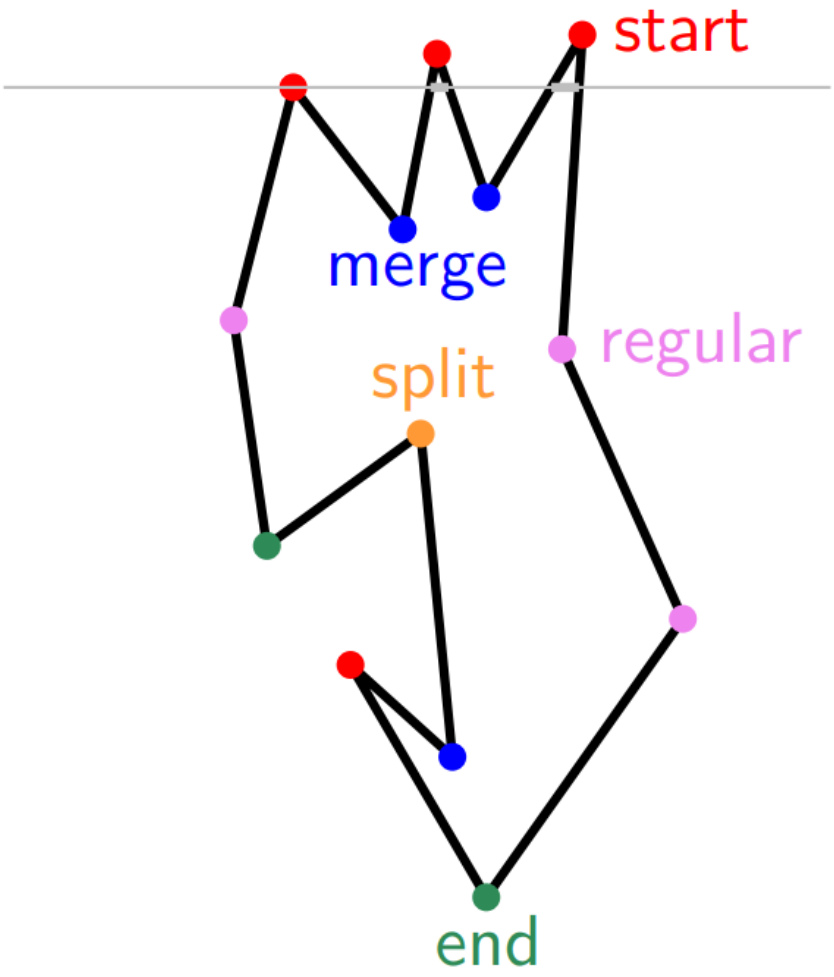
Figur 41: Prof that n over 3 is always enough

Y-monotone polygon is a polygone has a top vertex a botom vertex and two y-monotone chains between the top and botom, so the idea is to split the polygone into different y monitone poligones and then split those. See example below:



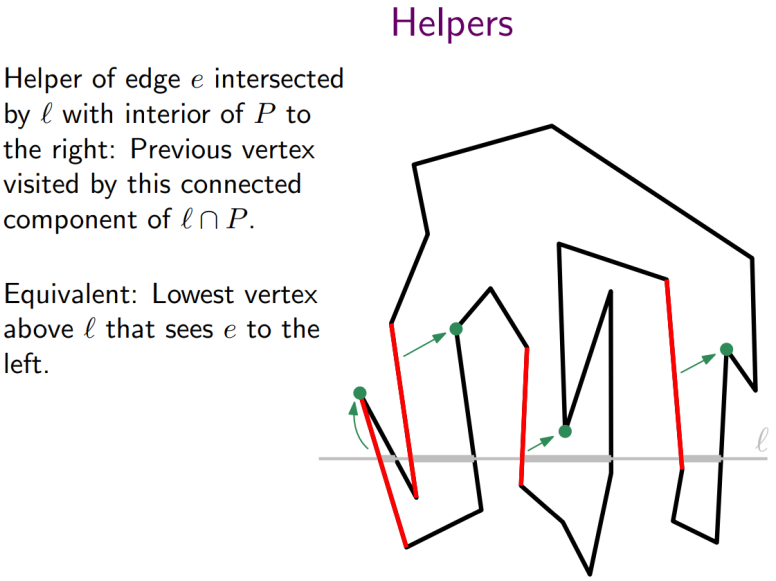
Figur 42: y monotone polygone

Vertex types There are 5 different vertex types. Start, stop, split, merge and regular. They can be seen on the illustration below:

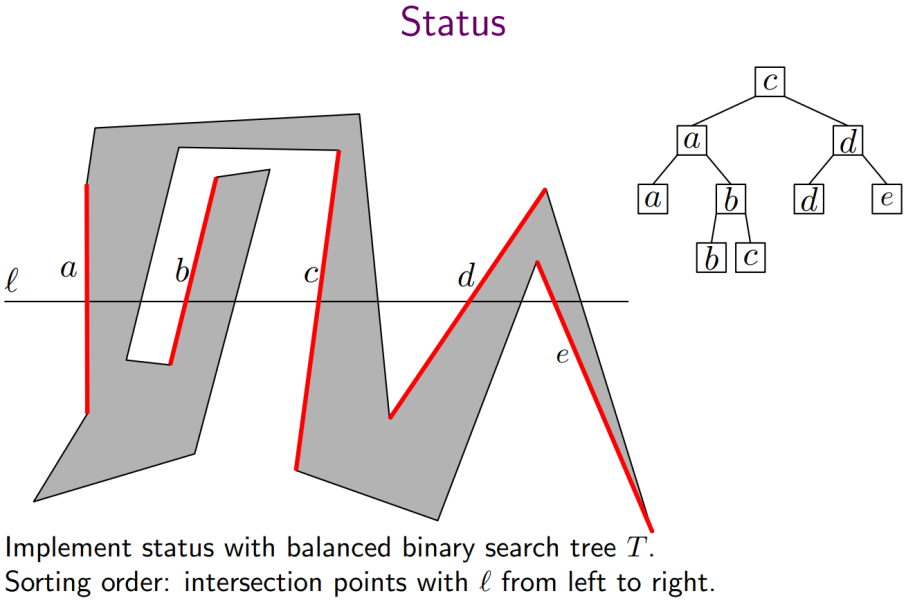


Figur 43: Vertex types

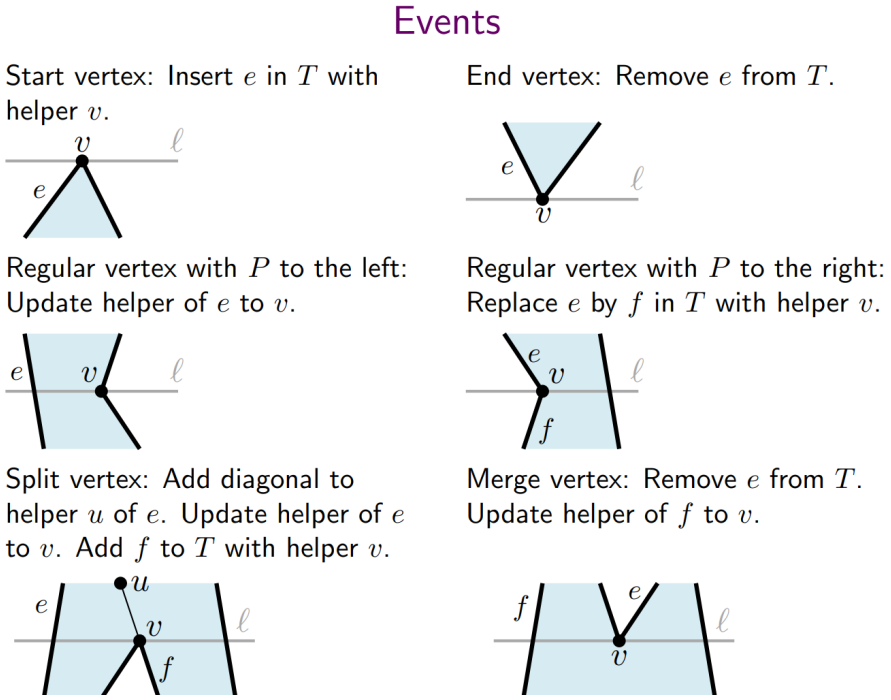
Sweep line Is when we have a line going from the top and down and then from the bottom up. We know that if we dont have any split or merge vertex then the polygone is y-monotone. After doing the sweep down and removing split vertexes, we flip the graph and simply do the same upside down, now our merge vertexes will be the new split vertexes.



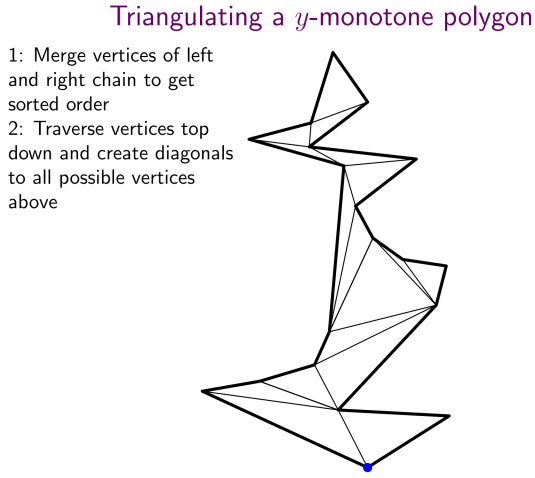
Figur 44: Helpers triangulation



Figur 45: Status tree T



Figur 46: Events on sweep



Figur 47: Triangulate a monotone polygon

7 Approximation algorithms

Def.: An algorithm for an optimization problem has *approximation ratio* $\rho(n)$ if for every input of size n,

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n).$$

C^* :=cost(opt. sol.) — minimization problem

C :=cost(produced sol.) — maximization problem

Figur 48: Definition of approximation algorithm

Approx-vertex-cover We approximate a vertex cover by choosing an edge in the graph and add both the endpoints to the graph, then we remove all the edges the vertexes touch from the graph and keep going. This is a 2-approximation algorithm as the worst case is it is twice as bad as the optimal solution.

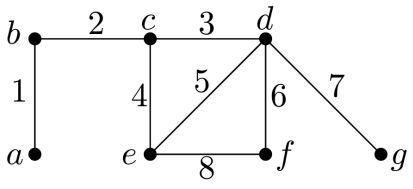
Vertex Cover

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 $v \in V'$.

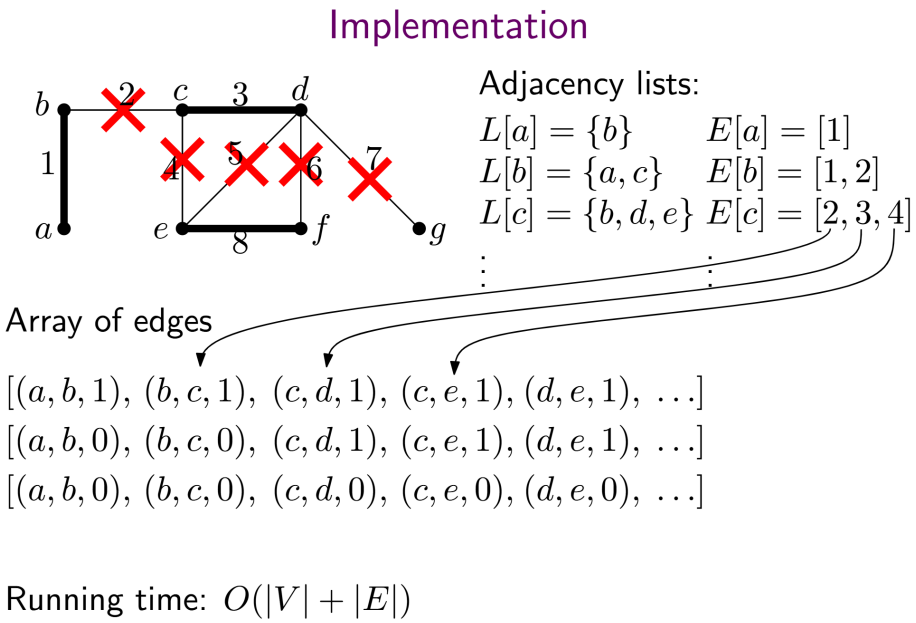
NP-hard!

```
APPROX-VERTEX-COVER(G)
  C := ∅
  while E(G) ≠ ∅
    choose uv ∈ E(G)
    C := C ∪ {u, v}
    remove all edges incident on u or v from E(G)
  return C
```

Exercise:



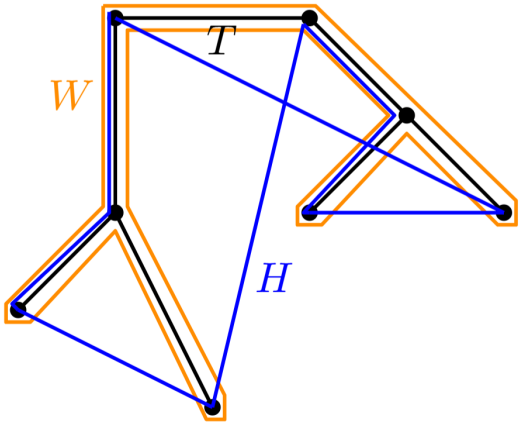
Figur 49: Approx vertex cover definition



Figur 50: Approx vertex cover example

Approx traveling Salesman Here we have to assume the triangle inequallity, that is $c(uw) \leq c(uv) + c(vw)$ (meaning it is at least as fast to go directly.)
To solve the problem we make a Minimum spanning tree and then we make a euler tour of T (visit every edge at least once) and then we do that again but skipping the ones we have allready visited.

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 51: Approx TSM problem example

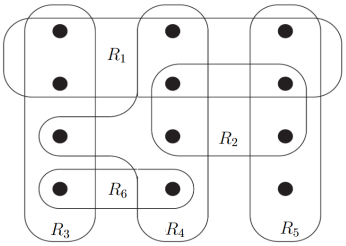
Set cover Find the least amount of sets where it fits the whole set:

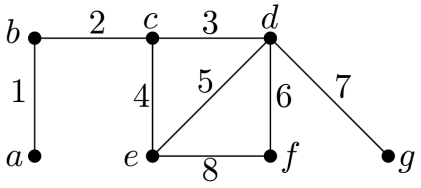
Set Cover

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.

Exercise: Show that vertex cover is a special case.

$X := \{1, 2, \ldots, 8\}$
$\mathcal{F} := \{\{1\}, \{1, 2\}, \{2, 3, 4\}, \{3, 5, 6, 7\}, \{4, 5, 8\}, \{6, 8\}, \{7\}\}$



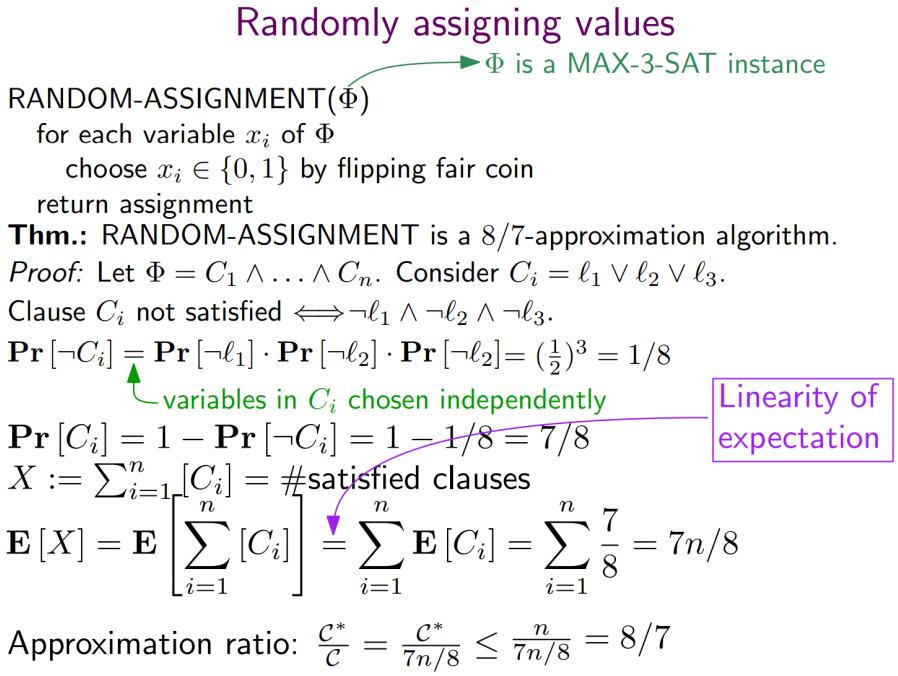


$X := E$
$\mathcal{F} := \{E(v) \mid v \in V\}$
$E(v) := \{uv \in E \mid u \in V\}$

Figur 52: Set cover example

Greedy set cover Here we cover the most amount of elements that are not covered already.

Randommised 3 sat



Figur 53: Randomized 3 sat

Minimum weighted vertex cover is where we find a vertex cover with the minimum amount of weight, so we have put weights on each vertex. This can be made into a linear programmingg problem, however it is still NP complete.

Weighted Vertex Cover

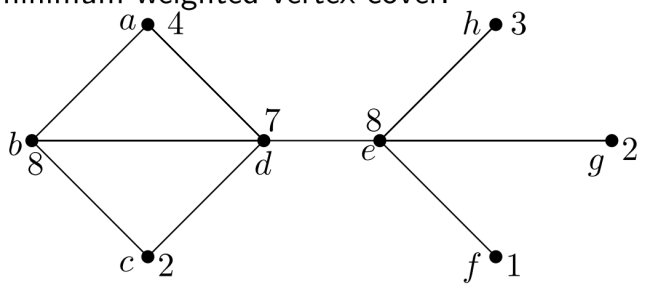
Def.: Let $G = (V, E)$ be a graph. A set $V' \subseteq V$ of vertices is a *vertex cover* if for all $uv \in E$, we have $u \in V'$ or $v \in V'$.

Now: We are given weight $w(v) > 0$ for each $v \in V$.

Goal: Find vertex cover C with minimum

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

Exercise: Find minimum (unweighted) vertex cover and then minimum weighted vertex cover.



Figur 54: Weighted vertex cover

Subset sum approximation We want to find a sum of numbers where the cost is as close to t as possible without exceeding it.

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 $\sum_{x \in U} x$.

Example: $S = \{1, 4, 5\}$, $t = 8$.

NP-complete to decide if $\exists U \subset S : \sum_{x \in U} x = t$.

Abstract exact alg.:

for $k = 1, 2, \ldots, n$
  compute $L_k := \left\{ \sum_{x \in U} x \mid U \subset \{x_1, \ldots, x_k\} \wedge \sum_{x \in U} x \leq t \right\}$.
return $\max L_n$

Note: $L_k \subset L_{k-1} \cup (L_{k-1} + x_k)$.

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 55: Subset-sum