AADS exam

January 2024

Contents

1 Max Flow

1.1 Disposition

What is a flow network?

What is max flow?

Ford-folkerson method
What is a cut?
Max-flow-min-cut theorem

1.1.1 What is a flow network

- Draw flow network

1.1.2 What is max flow

- Define flow - $\sum_{v \in V} f(s, v) - f(v, s)$
- Define max flow as the maximum value of the flow
- define capacity constraint
- define flow conservation

1.1.3 Ford-Fulkerson Method

- go through ford-fulkerson method
- Draw residual network
- define the residual capacity

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

- Show worst case $\mathcal{O}(2 \cdot m) = \mathcal{O}(m)$
- namedrop Edmond-karp

1.1.4 What is a cut?

- Define S as the set reachable from s
- define $T = V \setminus S$
- define min-cut as the cut which minimizes the capacity of the edges.
- define flow across the cut
- say min-cut = max-flow

1.1.5 Max-flow-Min-cut-theorem

- 1 implies 2: Proof by contradiction. If we have a max-flow then we have an augmenting path. Augmenting flow with the path, mean larger flow. Contradicts max flow.
- 2 implies 3: If we have no augmenting paths, then the capacity for all paths must be 0. If there are augmenting paths, when we would be able to reach the vertex from s, thus we would not have a cut. Then by the def. of the flow across the cut, then we know because there are no augmenting paths, that the flow from S to T must be the capacity, and there are 0 flow the other way.
- 3 implies 1: If we have flow across the cut, with the value of the capacity, then we cannot increase the flow because of capacity constraint. Thus we have max-flow.

1.2 Definitions

1.2.1 Flow network

A flow network is a directed graph, with a source s and sink t along with a capacity function $c : V \times V \to \mathbb{R}$, such that:

The capacity is greater than 0 for all edges (no negative edges):

$$c(u, v) \geq 0 | \forall u, v \in V$$

and if there is an edge not in the edge set E, then it has a capacity of 0: $(u, v) \notin E$ then $c(u, v) = 0$

We also assume there are no self-loops and no antiparallel edges. I.e. no edge (v, v) and if there is an edge (u, v) then there is no edge (v, u). If this is the case, then we can remove the self loops, by removing the edge. We can also remove antiparallel edges, by splitting the one edge into two connected by a vertex.

1.2.2 Flow in a flow network

A flow is a function: $f : V \times V \to \mathbb{R}$ such that:

Capacity constraint - no flow is less than 0 or greater than the capacity of the edge:

$$\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$$

Flow conservation - every flow going into a vertex, must also leave the vertex, except for the source and sink:

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

which we can also denote by, the sum of the flow over the edges going into v, must equal the sum of the flow over the edges leaving v:

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

1.2.3 Value of flow

The value of a flow, $|f|$, is the sum of the flow going out of the source, minus the sum of flow going into the source:

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

1.2.4 Residual network

A residual network is a flow network with a capacity function c_f defined by:

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

The edges of the residual network are defined as the set of edges which have a capacity c_f greater than 0:

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

This network can have antiparallel edges, as case 1 ($c(u,v) - f(u,v)$) and 2 ($f(v,u)$) in the capacity function can both be positive, thus creating 2 antiparallel edges.

1.2.5 Ford-Fulkerson Method

The Ford-Fulkerson Method is increasing the flow in some path, while there exists an augmenting path. It does not say how to find augmenting paths, but we know that we have found the max flow, when there are no more augmenting paths left.

1.2.6 Augmenting

Augmenting betyder forstærkende, altså at man forstærker noget. Aka. en augmenting path, vil forstærke/øge flowet.

1.2.7 Augmented flow

If we have a flow f in G and a flow f' in G_f, then we can define the augmented flow as: $f \uparrow f' : V \times V \to \mathbb{R}$

The augmented flow is the flow from G going out of the vertex, plus the flow from G_f going out from the vertex, minus the flow from G_f going into the vertex:

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

1.2.8 Cut

A cut is a partition of the vertices into 2 subsets such that the source and sink are not in the same set.

1.2.9 Net flow across a cut

We can calculate the flow across a cut as the flow going into the sink set, T, minus the flow going into the source set, S:

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

1.2.10 Capacity of flow across a cut

We can define the capacity of the flow across a cut as the sum of the capacity of the edges across the cut from S to T:

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

1.2.11 Net flow value across all cuts (S,T)

We can define the net flow value across all cuts (S,T) as the value of the flow in the network $f(S,T) = |f|$:

proof:
We know that the flow across a cut is:

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

Because of flow conservation, every flow going into u must also go out of u. Thus wee add the double sum of edges going in and out of vertices in S:
$\sum_{u \in S} \sum_{v \in S} (f(u,v) - f(v,u))$ it should be 0 because for every positive term, there should be a negative term, as we eventually add the inverse of the positive edge (flow conservation):

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

Because we have $v \in S$ from the left double sum, and $v \in T$ from the right double sum, then we can combine them to $v \in V$:

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

We can then split the sum into 2 seperate sums, by dividing the set S into a set containing the source and a set containing the rest of S: $\{s\}$ and $S \setminus \{s\}$:

$$= \sum_{u \in \{s\}} \sum_{v \in V} (f(u,v) - f(v,u)) + \sum_{u \in S \setminus \{s\}} \sum_{v \in V} (f(u,v) - f(v,u))$$

We can remove the sum over s:

$$= \sum_{v \in V} (f(s,v) - f(v,s)) + \sum_{u \in S \setminus \{s\}} \sum_{v \in V} (f(u,v) - f(v,u))$$

Then because of flow conservation we can remove the left double sum, as all flow going out of $u \in S \setminus \{s\}$ must also go into $u \in S \setminus \{s\}$. Thus we arrive at the definition of the value of the flow:

$$= \sum_{v \in V} (f(s,v) - f(v,s)) + 0 = |f|$$

1.2.12 Any flow value is upper bounded by the cut capacity

We will prove that for any flow f and any cut (S,T) the value of the flow is less than or equal to the capacity of the cut $c(S,T)$:

Given that the flow value is equal to the flow across the cut:

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

Because we know that the flow must be less than or equal to the capacity of the edge, given the capacity constraint: $f(u,v) \leq c(u,v)$, when we also know that the negative term for the inverse edge: $-f(v,u)$, must be below 0 as per the capacity constraints: $-f(v,u) \leq 0$. Thus we can make the following upper bound:

$$\sum_{u \in S} \sum_{v \in T} (f(u,v) - f(v,u)) \leq \sum_{u \in S} \sum_{v \in T} (c(u,v) - 0) = \sum_{u \in S} \sum_{v \in T} c(u,v) = c(S,T)$$

Thus because of the definition of the capacity of the cut, we get that the flow across the cut is upper bounded by the capacity of the cut.

1.3 Max flow/Min cut Theorem

1.3.1 Theorem

Given a flow f in G, the following 3 statements are equivalent:

1. f is a max flow
2. There is no augmenting path (aka. a path from source to sink in the residual graph G_f)
3. $\exists$ a cut (S,T) such that $|f| = c(S,T)$

1.3.2 Proof: 1 $\Rightarrow$ 2

To prove that 1 $\Rightarrow$ 2, we do so by proof of contradiction: We assume that f is a max flow and there exists an augmenting path p (contradiction of 2). Then we can augment the flow f by the flow f_p produced by augmenting the path p, as per the definition of augmented flow, we get a flow value of:

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

This flow is greater than the max flow, as the existence of an augmented path means an increase in the flow. Thus we reach a contradiction as the max flow is less than the augmented flow:

$$|f \uparrow f_p| > |f|$$

1.3.3 Proof: 2 $\Rightarrow$ 3

If we define the set S to be the set of vertices reachable from s in the residual graph G_f:

$$S = \{v \in V | v \text{ is reachable from } s \text{ in } G_f\}$$

Then we can define the set T, as the set of vertices not reachable from s:

$$T = V \setminus S$$

We know that this partitions the set in 2 subsets of V, with $s \in S$ because s can reach itself from s. We can also say that $t \in T$ because otherwise we would have an augmenting path from s to t, which would contradict the 2. statement from the theorem, which we assume to be true.

Now we define $u \in S$ and $v \in T$. If the edge (u,v) is a part of E, then the flow over this edge would be the same as the capacity of the edge: $f(u,v) = c(u,v)$. If this was not the case, then the capacity for the edge in the residual graph would be above 0, thus it would be an edge of the residual graph. $f(u,v) \neq c(u,v) \Rightarrow c_f(u,v) > 0 \Rightarrow (u,v) \in E_f$. This is not possible as we have defined our set S to be every vertex reachable form s. Thus if this is the case, then by transitivity, both v and u would be reachable from s. Thus both vertices being in S contradicting $v \in T = V \setminus S$.

We also know that if the edge $(u, v) \in E$ then the flow of the inverse edge $f(v, u) = 0$, otherwise we would have an edge in the residual path as $c_f(u, v) > 0$. Thus we reach the same contradiction of the definition of the set $T = V \setminus S$.

Because we have a cut (S, T), then we can calculate the flow across the cut as:

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

Then because of the fact that we have established in the previous paragraph, that the flow $f(u, v) = c(u, v)$ and the flow $f(v, u) = 0$, then we can replace the values of the flow in the double sum:

$$\sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u)) = \sum_{u \in S} \sum_{v \in T} (c(u, v) - 0)$$

Then by applying the definition of the capacity constraint across the cut: $c(S, T)$, we get:

$$\sum_{u \in S} \sum_{v \in T} (c(u, v) - 0) = \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T)$$

Thus we have shown:

$$|f| = f(S,T) = c(S,T)$$

Thereby proving that if there are no augmenting paths, it implies that we have found a cut (S, T) such that the flow value is the same as the capacity of the cut: $|f| = c(S, T)$.

1.3.4 Proof: 3 $\Rightarrow$ 1

If there exists a cut (S, T) such that the flow across the cut is equal to the capacity of the cut, $|f| = c(S, T)$, then it must be the max flow. This is because any other flow f' in the graph would be bounded by the capacity of the cut, as we know from: $|f'| \leq c(S, T) = |f|$. Thus the existence of a cut (S, T) with $|f| = c(S, T)$ implies that f is a max flow.

2 Linear Programming

2.0.1 Linear Programming

- What is a linear programming problem?

- Standard form (with example below)

- Slack form (with example below)
- Simplex algorithm (with example below)
- duality
- converting primal to dual (with example below)
- Weak linear-programming duality

$$\begin{array}{lrcrcr}
\text{Maximize} & 2x_1 & \text{-} & x_2 & & \\
\text{Subject to} & & & & & \\
 & x_1 & + & 2x_2 & \leq & 4 \\
 & 5x_1 & & & \leq & 30 \\
 & x_1 & , & x_2 & \geq & 0
\end{array}$$

2.1 Definition of linear programming

A linear programming problem is defined as a minimization or maximization problem of which we some variables which are subject to some constraints.

Books definition:
We wish to optimize a linear function subject to a set of linear inequalities. That is given a set of real numbers $a_1, ..., a_n$ and a set of variables $x_1, ..., x_n$, we define a linear function f on the variables by:

$$f(x_1, ..., x_n) = a_1x_1 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j$$

Then if b is a real number and f is a linear function, then we have the equation:

$$f(x_1, ..., x_n) = b$$

Which is a linear equality with the following linear inequalities:
The function must be less than or equal to b:

$$f(x_1, ..., x_n) \leq b$$

The function must be greater than or equal to b:

$$f(x_1, ..., x_n) \geq b$$

The term linear constraint denotes the equalities or inequalities.

Note: a linear programming problem cannot be **strictly** less than or greater than b.

2.2 Definitions

2.2.1 Feasible solution

A feasible solution is a solution to the linear program which satisfies all the constraints. I.e. it is a set of values for each variables which are valid given the constraints. **However** it might not be the optimal solution.

2.2.2 Feasible region

The feasible region is the n-dimensional region bounded by the constraints, for which a feasible solution can exists.

2.2.3 Objective function

The objective function is a function under the constraints of the linear program, which we want to maximize in the linear program. This function have a value called **Objective value** which are points within the feasible region.

2.2.4 Optimal solution

An optimal solution is a solution for which the linear function has the maximal value, given the constraints.

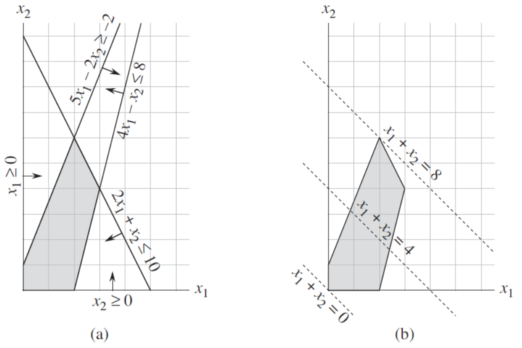


Figure 29.2 **(a)** The linear program given in (29.12)–(29.15). Each constraint is represented by a line and a direction. The intersection of the constraints, which is the feasible region, is shaded. **(b)** The dotted lines show, respectively, the points for which the objective value is 0, 4, and 8. The optimal solution to the linear program is $x_1 = 2$ and $x_2 = 6$ with objective value 8.

2.2.5 Simplex

A simplex is **kinda like** a polygon for which each vertex in the polygon is a point in which the constraints meet in space.
The simplex is bounded by the feasible region.

2.3 Standard and slack form

2.3.1 Standard form

The standard form of a linear program is as follows:
We are given n real numbers: $c_1, ..., c_n$
m real numbers: $b_1, ..., b_m$
And $m \cdot n$ real numbers a_{ij} for $1 \leq i \leq m$ and $1 \leq j \leq n$.
Then we wish to find n real numbers: $x_1, ..., x_n$ such that we can maximize the objective function:

$$\sum_{j=1}^{n} c_j x_j$$

Subject to the constraints:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } 1 \leq i \leq m$$

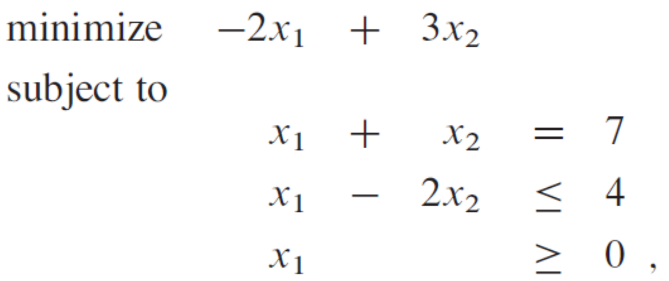
$$x_j \geq 0 \text{ for } 1 \leq j \leq n$$

The last line (constraint) is the non-negativity constraints.

2.3.2 Linear program to standard form

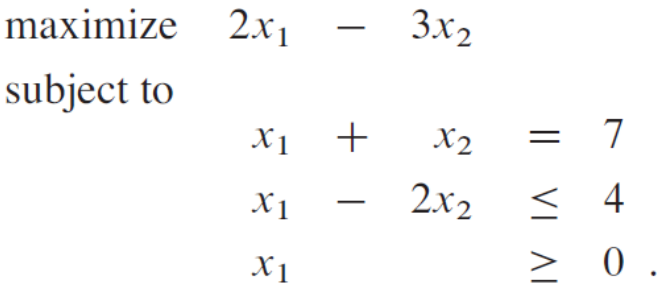
Some linear programs might not be in standard form because of the following reasons:

1. The objective functions is a minimization, where it should be maximization
2. There might be variables without the non-negativity constraint
3. There might be equality constraints, rather than less-than-or-equal-to
4. There might be inequiality constraints of greater-than-or-equal-to, which should be less-than-or-equal-to



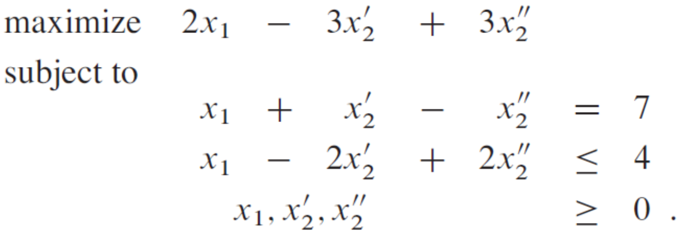
$$\begin{array}{lrcrcl}
\text{minimize} & -2x_1 & + & 3x_2 & & \\
\text{subject to} & & & & & \\
 & x_1 & + & x_2 & = & 7 \\
 & x_1 & - & 2x_2 & \leq & 4 \\
 & x_1 & & & \geq & 0 \ ,
\end{array}$$

1: minimization → maximization
To convert a minimization problem into a maximization problem, we negate every coefficient in the objective function, which then yields a maximization problem. Kinda like multiplying with -1



$$\begin{array}{lrcrcl} \text{maximize} & 2x_1 & - & 3x_2 & & \\ \text{subject to} & & & & & \\ & x_1 & + & x_2 & = & 7 \\ & x_1 & - & 2x_2 & \leq & 4 \\ & x_1 & & & \geq & 0 \ . \end{array}$$

2: negative → non-negative constraints
If there are negative constraints, then we can convert them into non-negative constraints. If we suppose there is a constraint: $x_j \leq 2$, and no non-negativity constraint ($x_j \geq 0$), then valid values for x_j is negative values. However if we replace x_j with x'_j and x''_j such that: $x_j = x'_j - x''_j$ we get that both x'_j and x''_j can be non-negative to equal x_j. Thus when we have replaced the values, we can add x'_j and x''_j to the non-negativity constraints: $x'_j, x''_j \geq 0$.



$$\begin{array}{lrcrcrcl} \text{maximize} & 2x_1 & - & 3x'_2 & + & 3x''_2 & & \\ \text{subject to} & & & & & & & \\ & x_1 & + & x'_2 & - & x''_2 & = & 7 \\ & x_1 & - & 2x'_2 & + & 2x''_2 & \leq & 4 \\ & & & x_1, x'_2, x''_2 & & & \geq & 0 \ . \end{array}$$

3: equality → less-than-or-equal-to
To remove the equality constraints, we can replace the equality constraint with two new constraints, one constraining to be greater-than-or-equal-to and another which constrains to be less-than-or-equal-to. We can do so because if we have a constraint saying $x \leq y$ and $x \geq y$ then it implies that $x = y$.
This will however add greater-than-or-equal-to constraints, which we will remove in the next step.

$$\begin{array}{lrcrcrcr}
\text{maximize} & 2x_1 & - & 3x_2' & + & 3x_2'' & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2' & - & x_2'' & \leq & 7 \\
& x_1 & + & x_2' & - & x_2'' & \geq & 7 \\
& x_1 & - & 2x_2' & + & 2x_2'' & \leq & 4 \\
& & & x_1, x_2', x_2'' & & & \geq & 0 \ .
\end{array}$$

4: greater-than-or-equal-to → less-than-or-equal-to
To replace greater-than-or-equal-to with less-than-or-equal-to, then we can make use of the same trick as in converting the minimization problem into a maximization problem, by negating the right and left hand side of the inequality by multiplying with -1.

$$\begin{array}{lrcrcrcr}
\text{maximize} & 2x_1 & - & 3x_2 & + & 3x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2 & - & x_3 & \leq & 7 \\
& -x_1 & - & x_2 & + & x_3 & \leq & -7 \\
& x_1 & - & 2x_2 & + & 2x_3 & \leq & 4 \\
& & & x_1, x_2, x_3 & & & \geq & 0 \ .
\end{array}$$

2.3.3 Slack form

In slack form we introduce a variable s, called the slack variable. Then we have the inequality constraints from the linear program:

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i$$

which we can rewrite with the slack variable as:

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_j$$

$$s \geq 0$$

We can think of the slack variable as the difference between the left hand side and right hand sides of the inequality of the linear program.

2.3.4 Linear program to slack form

To convert a linear program to a slack form, we first convert it into standard form and then into slack form.

To convert standard form into slack form we can denote the slack variable associated with the i^{th} inequality as x_{n+i} such that:

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j$$

Thus we convert each inequality constraint in the standard form into this, and add the non-negativity constraints of the slack variables.

Thus we get the following linear program in slack form:

$$\begin{array}{lrcrcrcrcr}
\text{maximize} & & & & & 2x_1 & - & 3x_2 & + & 3x_3 \\
\text{subject to} & & & & & & & & & \\
& x_4 & = & 7 & - & x_1 & - & x_2 & + & x_3 \\
& x_5 & = & -7 & + & x_1 & + & x_2 & - & x_3 \\
& x_6 & = & 4 & - & x_1 & + & 2x_2 & - & 2x_3 \\
& \multicolumn{5}{r}{x_1, x_2, x_3, x_4, x_5, x_6} & \geq & 0 & . &
\end{array}$$

The left hand side variables are also called basic variables and the right hand side are called nonbasic variables.

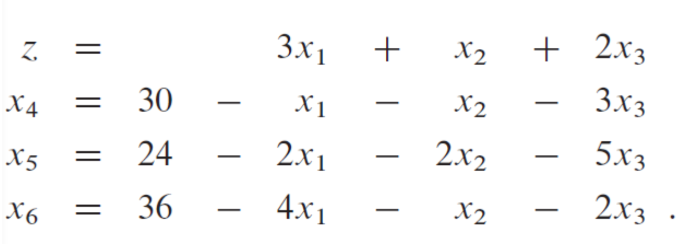
2.4 Simplex algorithm

The basic idea of the simplex algorithm is to start with a linear program in slack form, for which we want to try to increase each nonbasic (left hand side) variable as much as possible. This is kinda like walking the edges of the feasible solution space.

$$\begin{array}{lrcrcrcr}
\text{maximize} & 3x_1 & + & x_2 & + & 2x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2 & + & 3x_3 & \leq & 30 \\
& 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 24 \\
& 4x_1 & + & x_2 & + & 2x_3 & \leq & 36 \\
& \multicolumn{5}{c}{x_1, x_2, x_3} & \geq & 0 \; .
\end{array}$$

We do so by choosing the value in the objective function with the highest coefficient, and them by setting all other variables to 0, we will see that there is a limit to how much we can increase this variable before the non-negativity constraints are broken.

In the following example we will choose x_1 as it has the highest coeffecient in the objective function, and we will switch it with x_6 as it is the constraint for which has the tightest bound for the value of x_1:

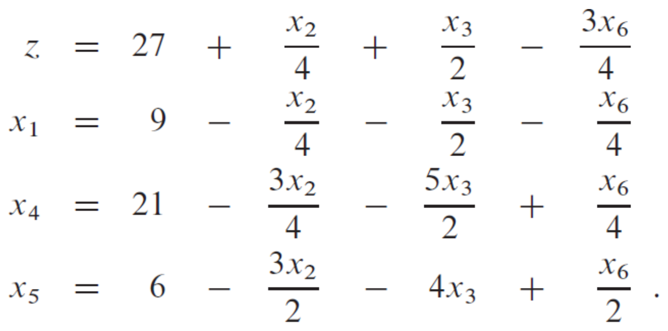


$$\begin{array}{rcrcrcrcr}
z & = & & & 3x_1 & + & x_2 & + & 2x_3 \\
x_4 & = & 30 & - & x_1 & - & x_2 & - & 3x_3 \\
x_5 & = & 24 & - & 2x_1 & - & 2x_2 & - & 5x_3 \\
x_6 & = & 36 & - & 4x_1 & - & x_2 & - & 2x_3 \ .
\end{array}$$

By isolating x_1 in the constraint with x_6 we get:

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$

Which we can replace all other occurrences of x_1 with in the linear program such that we get:



$$\begin{array}{rcrcccccc}
z & = & 27 & + & \dfrac{x_2}{4} & + & \dfrac{x_3}{2} & - & \dfrac{3x_6}{4} \\
x_1 & = & 9 & - & \dfrac{x_2}{4} & - & \dfrac{x_3}{2} & - & \dfrac{x_6}{4} \\
x_4 & = & 21 & - & \dfrac{3x_2}{4} & - & \dfrac{5x_3}{2} & + & \dfrac{x_6}{4} \\
x_5 & = & 6 & - & \dfrac{3x_2}{2} & - & 4x_3 & + & \dfrac{x_6}{2} \ .
\end{array}$$

This operation is called pivoting.

Then we just choose a new variable with the highest coefficient in the objective function. Here we do **not** choose x_6 as it has a negative coefficient, but we choose x_3 instead, and get:

$$
\begin{array}{rcrcrcrcr}
z & = & \dfrac{111}{4} & + & \dfrac{x_2}{16} & - & \dfrac{x_5}{8} & - & \dfrac{11x_6}{16} \\
x_1 & = & \dfrac{33}{4} & - & \dfrac{x_2}{16} & + & \dfrac{x_5}{8} & - & \dfrac{5x_6}{16} \\
x_3 & = & \dfrac{3}{2} & - & \dfrac{3x_2}{8} & - & \dfrac{x_5}{4} & + & \dfrac{x_6}{8} \\
x_4 & = & \dfrac{69}{4} & + & \dfrac{3x_2}{16} & + & \dfrac{5x_5}{8} & - & \dfrac{x_6}{16} \, .
\end{array}
$$

lastly we choose x_2 and we get the following:

$$
\begin{array}{rcrcrcrcr}
z & = & 28 & - & \dfrac{x_3}{6} & - & \dfrac{x_5}{6} & - & \dfrac{2x_6}{3} \\
x_1 & = & 8 & + & \dfrac{x_3}{6} & + & \dfrac{x_5}{6} & - & \dfrac{x_6}{3} \\
x_2 & = & 4 & - & \dfrac{8x_3}{3} & - & \dfrac{2x_5}{3} & + & \dfrac{x_6}{3} \\
x_4 & = & 18 & - & \dfrac{x_3}{2} & + & \dfrac{x_5}{2} \, . & &
\end{array}
$$

We now stop, as each increase to the variables in the objective function, will decrease the objective value (z)

2.5 Duality

2.5.1 linear-programming duality

To proove that a linear program indeed yields the maximum solution, then we can formulate a dual, which is a linear program in which the goal is to minimize and which has an optimal solution which is identical to the original linear program.
We call the original linear program the **primal**.

2.5.2 Dual definition

To formulate a dual for the primal, we define it as:
We want to minimize:

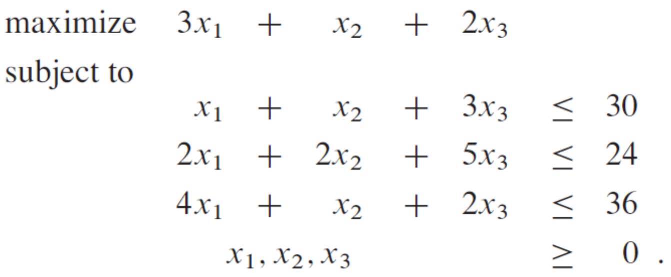
$$\sum_{i=1}^{m} b_i y_i$$

subject to the constraints:

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \text{ for } 1 \leq j \leq n$$

$$y_i \geq 0 \text{ for } 1 \leq i \leq m$$

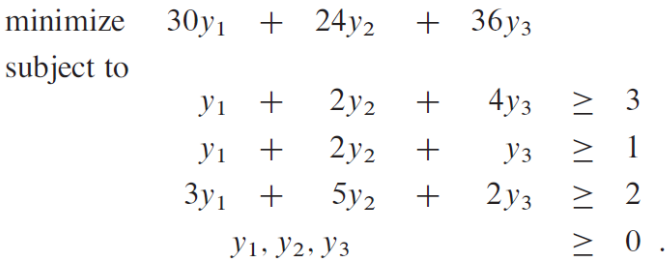
2.5.3 From primal to duel



$$\begin{array}{lrcrcrcl}
\text{maximize} & 3x_1 & + & x_2 & + & 2x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2 & + & 3x_3 & \leq & 30 \\
& 2x_1 & + & 2x_2 & + & 5x_3 & \leq & 24 \\
& 4x_1 & + & x_2 & + & 2x_3 & \leq & 36 \\
& \multicolumn{5}{c}{x_1, x_2, x_3} & \geq & 0 \ .
\end{array}$$

if we want to create a dual from a primal, then we can do so by changing the maximization, to a minimization.

Then transpose the coefficients, such that they become transposed:



$$\begin{array}{lrcrcrcl}
\text{minimize} & 30y_1 & + & 24y_2 & + & 36y_3 & & \\
\text{subject to} & & & & & & & \\
& y_1 & + & 2y_2 & + & 4y_3 & \geq & 3 \\
& y_1 & + & 2y_2 & + & y_3 & \geq & 1 \\
& 3y_1 & + & 5y_2 & + & 2y_3 & \geq & 2 \\
& \multicolumn{5}{c}{y_1, y_2, y_3} & \geq & 0 \ .
\end{array}$$

2.5.4 Weak linear-programming duality proof

let $\overline{x}$ be any feasible solution to the primal and let $\overline{y}$ be any feasible solution to the dual. Then we have:

$$\sum_{j=1}^{n} c_j \overline{x}_j \leq \sum_{i=1}^{m} b_i \overline{y}_i$$

Proof:

We know from the definition of the dual from that:

$$c_j \leq \sum_{i=1}^{m} a_{ij} y_i$$

Which means we can create the inequality by wrapping each term in $\sum_{j=1}^{n}(inner)\overline{x}_j$:

$$\sum_{j=1}^{n}(c_j)\overline{x}_j \leq \sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij}\overline{y}_i\right)\overline{x}_j$$

Then we can swap the sums of the right-hand side:

$$\sum_{j=1}^{n}\left(\sum_{i=1}^{m} a_{ij}\overline{y}_i\right)\overline{x}_j = \sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij}\overline{x}_j\right)\overline{y}_i$$

Then we can use the inequality from the definition of the linear program:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

By wrapping b_i in $\sum_{i=1}^{m}(b_i)\overline{y}_i$:

$$\sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij}\overline{x}_j\right)\overline{y}_i \leq \sum_{i=1}^{m}(b_i)\overline{y}_i$$

Thus we have proven that:

$$\sum_{j=1}^{n} c_j\overline{x}_j \leq \sum_{i=1}^{m} b_i\overline{y}_i$$

2.5.5 Optimal solutions

We can show the optimal solution to the dual and primal to be the solution for which both objective functions are equal.

3 Random Algorithms

3.1 Disposition

- Motivation
- Las Vegas and Monte Carlo
- Randomized Quicksort
- Randomized Min-Cut

3.1.1 Motivation

Might be simpler to implement, but harder to analyse. Might be faster, in expected time but have weaker guarantees.

3.1.2 Las Vegas and Monte Carlo

Kig på rækkefølgen af quicksort og min-cut, det svarer til rækkefølgen der er skrevet ned.

Las Vegas: expected time Monte Carlo: Probability of error

3.1.3

3.2 Randomized Quicksort

The main idea is to pick the pivot element **uniformly at random**.

3.2.1 Finding the probability of 2 elements being compared

The probability of two elements at index i, j in the final sorted array: $[S_{(1)}, ..., S_{(n)}] :=$ RandQS(S)

Can be defined as when $i < j$, X_{ij} is the number of times $S_{(i)}$ and $S_{(j)}$ are compared. This means that:

$$\#Comparisons = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} = \sum_{1 \leq i < j \leq n} X_{ij} = \sum_{i<j} X_{ij}$$

This means the expected number of comparisons are:

$$\mathbb{E}[\#Comparisons] = \mathbb{E}\left[\sum_{i<j} X_{ij}\right]$$

Linearity of expectation ($\mathbb{E}[A + B] = \mathbb{E}[A] + \mathbb{E}[B]$) makes it

$$\mathbb{E}[\#Comparisons] = \sum_{i<j} \mathbb{E}[X_{ij}]$$

Because X_{ij} can be true/1 (i, j are compared) or false/0 (not compared), then we say $X_{ij} \in \{0, 1\}$ is an indicator variable for the event that i, j are compared.

This means we can replace $\mathbb{E}[X_{ij}]$ with the probability p_{ij} of the event being true:

$$\mathbb{E}[X_{ij}] = \sum_{x \in \{0,1\}} Pr[X_{ij} = x] \cdot x = (1 - p_{ij}) \cdot 0 + p_{ij} \cdot 1 = p_{ij}$$

We can calculate this probability as we know that the elements are only compared once iff. the pivot chosen, c, is i or j ($c \in \{i, j\}$).

We also know that if the pivot chosen is to the left or to the right of both elements, $c < i$ or $j < c$ then the elements are not compared at the current step, but might be in a recursive step.

Lastly if the pivot is between the elements, $i < c < j$ then they are separated and thus not compared.

We can calculate the probability of p_{ij} as the probability of picking $S_(i), S_(j)$ given we pick the pivot uniformly at random (u.a.r.):

$$p_{ij} = Pr[c \in \{i, j\} | c \in \{i, i+1, ..., j\} \text{ u.a.r. }]$$

When the first element i is picked, then we have $j + 1 - i$ elements left to pick from, thus when we pick 2 elements we get:

$$p_{ij} = Pr[c \in \{i, j\} | c \in \{i, i+1, ..., j\} \text{ u.a.r. }] = \frac{2}{|\{i, i+1, ..., j\}|} = \frac{2}{j+1-i}$$

This means the expected number of comparisons is:

$$\mathbb{E}[\#Comparisons] = \sum_{i<j} p_{ij} = \sum_{i<j} \frac{2}{j+1-i}$$

3.2.2 Number of comparisons analysis

To make it into big O notation we can simplify the expression via. the following steps:

1. expand $\sum_{i<j}$

$$\mathbb{E}[\#Comparisons] = \sum_{i<j} \frac{2}{j+1-i} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j+1-i}$$

2. replace $j + 1 - i$ with k such that $k = j + 1 - i$

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j+1-i} = \sum_{i=1}^{n-1} \sum_{k=2}^{n+1-i} \frac{2}{k}$$

3. bound it by double $\sum^n$

$$\sum_{i=1}^{n-1} \sum_{k=2}^{n+1-i} \frac{2}{k} < \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k}$$

4. remove first sum

$$\sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} = n \sum_{k=2}^{n} \frac{2}{k}$$

5. move 2 outside fraction

$$n \sum_{k=2}^{n} \frac{2}{k} = 2n \sum_{k=2}^{n} \frac{1}{k}$$

6. Make sum of k start at 1

$$2n\sum_{k=2}^{n}\frac{1}{k} = 2n\left(\left(\sum_{k=1}^{n}\frac{1}{k}\right) - 1\right)$$

7. Replace sum with Harmonic number, H_n

$$2n\left(\left(\sum_{k=1}^{n}\frac{1}{k}\right) - 1\right) = 2n\left(H_n - 1\right)$$

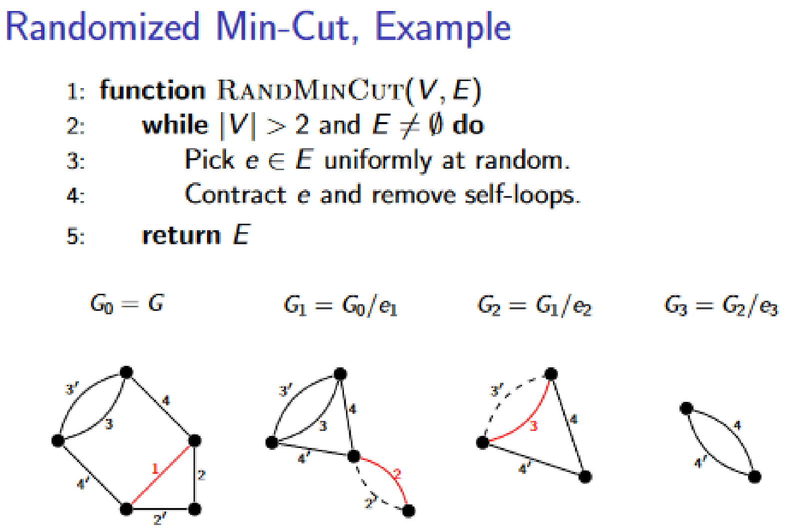
8. We know this to be bounded by:

$$2n\left(H_n - 1\right) \leq 2n \cdot H_n = 2n\int_{1}^{n}\frac{1}{x}dx = 2n\ln n \in \mathcal{O}(n\log n)$$

Thus the expected number of comparisons is $\mathcal{O}(n\log n)$.

3.3 Randomized min-cut

Pick uniformly at random an edge in G and *contract* that edge. i.e. merge the two nodes from either end into 1 node, and remove the edge. Repeat until only two vertices or no more edges.



Randomized Min-Cut, Example

```
function RandMinCut($V, E$)
    while $|V| > 2$ and $E \neq \emptyset$ do
        Pick $e \in E$ uniformly at random.
        Contract $e$ and remove self-loops.
    return $E$
```

The probability of the randomized min-cut algorithm returns the min-cut is the probability that the randomized min-cut algorithm returns the correct two "*contracted*" vertices which edges correspond to the min-cut, C.

This probability is greater than or equal to picking any two random vertices:

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

If we define ε_i to be the event that in the i^{th} event, the chosen edge e_i is not a part of the min-cut, $e_i \notin C$.

This means that we return the correct C iff. ε_1 to ε_{n-2} happened: $\varepsilon_1 \cap ... \cap \varepsilon_{n-2}$. In other words the probability of C is preserved in all steps, is the probability of step 1 not choosing $e_1 \in C$ along with step 2, given step 1 etc.: $\varepsilon_1 \cap ... \cap \varepsilon_{n-2}$.

Thus we need to show:

$$Pr[C] = Pr[\varepsilon_1 \cap ... \cap \varepsilon_{n-2}] \geq \frac{2}{n(n-1)}$$

We can define the **conditional probability** of ε_2 given ε_1 as:

$$Pr[\varepsilon_2 | \varepsilon_1] = \frac{Pr[\varepsilon_2 \cap \varepsilon_1]}{Pr[\varepsilon_1]}$$

We can isolate $Pr[\varepsilon_1 \cap \varepsilon_2]$:

$$Pr[\varepsilon_2 \cap \varepsilon_1] = Pr[\varepsilon_1] \cdot Pr[\varepsilon_2 | \varepsilon_1]$$

To get the probability that each event does not remove an edge in C, i.e. all events are successful given the previous events:

$$Pr[\cap_{i=1}^{k} \varepsilon_i] = Pr[\varepsilon_1] \cdot Pr[\varepsilon_2 | \varepsilon_1] \cdots Pr[\varepsilon_k | \cap_{i=1}^{k-1} \varepsilon_i]$$

Thus we can define the probability of a specific min-cut C being returned as:

$$Pr[\text{Specific } C] = Pr[\varepsilon_1 \cap \cdots \cap \varepsilon_{n-2}]$$

$$= Pr[\varepsilon_1] \cdot Pr[\varepsilon_2 | \varepsilon_1] \cdots Pr[\varepsilon_{n-2} | \varepsilon_1 \cap \cdots \cap \varepsilon_{n-3}]$$

$$= \prod_{i=1}^{n-2} p_i$$

where $p_i = Pr[\varepsilon_i | \varepsilon_1 \cap \cdots \cap \varepsilon_{i-1}]$

We know that in the i^{th} step we have a $n_i = n - i$ vertices, because in each step we contract 2 vertices into 1 (aka. reduce number of vertices by 1).

We also know that contracting an edge does not decrease the min-cut size, as a cut in the i^{th} step is also a cut in the previous step, $i - 1$. Thus we can bound the cut, $\lambda(G_i)$ to be greater than to equal to the size of the min-cut $\lambda(G_i) \geq |C|$.

Because we can create a cut by removing all incident edges to a vertex v. i.e. if we remove all edges for a vertex, then we can create a cut. This means that every vertex in the i^{th} step has a degree $d_i(v)$ of at least the min-cut: $d_i(v) \geq |C|$.

Putting it all together we get:

$$d_i(v) \geq \lambda(G_i) \geq |C|$$

We can then define the sum of the incident edges over all vertices in the i^{th} step as:

$$\sum_{v \in V_i} d_i(v) \geq n_i|C|$$

However because we count every edge twice in $d_i(v)$ then we can define the sum of the degree of edges in the i^{th} step as:

$$|E_i| = \frac{1}{2} \sum_{v \in V_i} d_i(v) \geq \frac{1}{2} n_i|C|$$

We can define the probability of picking an edge not in C in the i^th step as p_i. Then we can define the probability of picking an edge in C in the i^{th} step, given that no previous steps have picked an edge in C as:

$$1 - p_i = Pr[\text{ uniformly random } e \in E_{i-1} \text{ is in } C| \cap_{j=1}^{i-1} \varepsilon_j] = \frac{|C|}{E_{i-1}}$$

We know that $|E_{i-1} \geq \frac{1}{2}n_{i-1}|C|$. Thus we can make the bound:

$$\frac{|C|}{E_{i-1}} \leq \frac{|C|}{\frac{1}{2}n_{i-1}|C|} = \frac{2|C|}{n_{i-1}|C|} = \frac{2}{n_{i-1}}$$

Because we know that in the i^{th} step we have $n_i = n - i$ vertices we get:

$$\frac{2}{n_{i-1}} = \frac{2}{n-(i-1)} = \frac{2}{n-i+1}$$

Thus we get:

$$1 - p_i \leq \frac{2}{n-(i-1)} = \frac{2}{n-i+1}$$

Which implies that the probability of not picking an edge in the i^{th} step from C is:

$$p_i \geq 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}$$

Lastly the probability that the min-cut C is returned is then:

$$Pr[C \text{ is returned}] = \prod_{i=1}^{n-2} p_i$$

where $p_i = Pr[\varepsilon_i | \varepsilon_1 \cap \cdots \cap \varepsilon_{i-1}]$

And with the bound we just found:

$$Pr[C \text{ is returned}] = \prod_{i=1}^{n-2} p_i \geq \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1}$$

Unrolling the product we get:

$$\prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} = \frac{\textbf{n-2}}{n} \cdot \frac{\textbf{n-3}}{n-1} \cdot \frac{\textbf{n-4}}{\textbf{n-2}} \cdots \frac{\textbf{3}}{\textbf{5}} \cdot \frac{2}{\textbf{4}} \cdot \frac{1}{\textbf{3}}$$

As we see all the **bold** terms cancel out, and we are left with:

$$Pr[C \text{ is returned}] \geq \frac{\textbf{n-2}}{n} \cdot \frac{\textbf{n-3}}{n-1} \cdot \frac{\textbf{n-4}}{\textbf{n-2}} \cdots \frac{\textbf{3}}{\textbf{5}} \cdot \frac{2}{\textbf{4}} \cdot \frac{1}{\textbf{3}} = \frac{2}{n(n-1)}$$

4 Hashing

4.0.1 Disposition

- Hash functions
- Universal hashing
- Hash table with chaining
- Strong universality
- Fast hash functions with Multiply shift

4.1 Definitions

4.1.1 Hash function

A hash function is a function which maps a universe U of keys to a number between 0 and a positive number m

A *random hash function* h is a randomly chosen hash function from a family of functions which map $U \to [m]$.
We can also say that it is a function h such that $\forall x \in U$, $h(x) \in [m]$ is a random variable ($h(x)$ is a random variable).

4.1.2 What we care about in hash functions

We care about the space needed to represent h.
We care about the time complexity of $h(x)$, as we want fast hash functions.
We care about the properties of the random variable.

4.1.3 Truly random hash function:

A hash function is truly random if the variable $h(x)$ is independent and uniform in $[m]$, i.e. it has an equal probability to hash to any value, and does not depend on previous hashes.
This is not feasible as there are $m^{|U|}$ possible functions which map U to $[m]$. This is because each U has m possible positions to map to, thus it becomes $m^{|U|}$. This means that we need at least $log_2(m^{|U|}) = |U|log_2m$ bits to store/represent the hash function we have picked.

4.1.4 Universal Hash function

A universal hash function is said to be universal iff. for any given distinct keys $x, y \in U$, when we pick h at random, we have a collision probability of:

$$Pr_h[h(x) = h(y)] \leq \frac{1}{m}$$

4.1.5 c-approximately universal hash function

A c-approximately universal hash function is a random hash function which has some probability, $\frac{c}{m}$, of mapping 2 distinct elements to the same value:

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

4.1.6 Strongly universal Hash function

A universal hash function is said to be strongly universal iff. the probability for any pairwise event is $\frac{1}{m^2}$.
If $h(x)$ and $h(y)$ is independent, then the probability of both hashing to specific values is:

$$Pr[h(x) = q \wedge h(y) = r]$$

Because they are independent, then this is the same as multiplying the probability of the event $h(x) = q$ with the probability of $h(y) = r$:

$$Pr[h(x) = q \wedge h(y) = r] = Pr[h(x) = q] \cdot Pr[h(y) = r]$$

Furthermore because we know that they are uniform in $[m]$, then we get:

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

4.1.7 c-approximately STRONGLY universal hash function

A c-approximately strongly universal hash function is a random hash function which hash each key c-approximately uniformly into $[m]$:

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

Any two keys must also hash independently:

$$\forall x \neq y \in U, q, r \in [m] : Pr_h[h(x) = q \wedge h(y) = r] \leq \frac{c^2}{m^2}$$

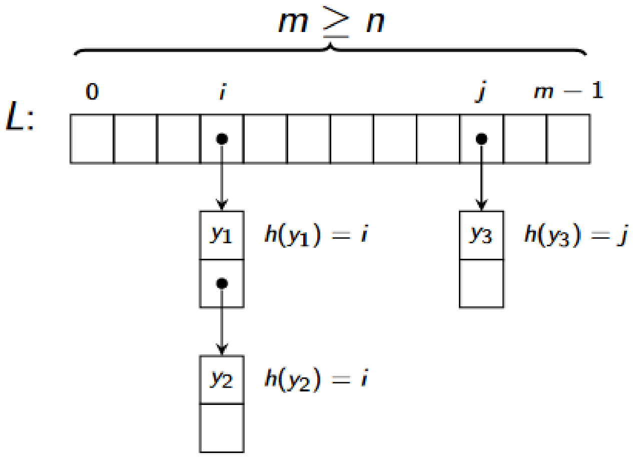
Proof:

If we know that h is c-approximately strongly universal, then we know that the probability of hashing to a specific value, q, is:

$$Pr[h(x) = q] \leq \frac{c}{m}$$

Thus we know that the upper bound on the pairwise event probability must be:

$$Pr[h(x) = q \wedge h(y) = r] = Pr[h(x) = q] \cdot Pr[h(y) = r]$$
$$\leq \frac{c}{m} \cdot \frac{c}{m} = \frac{c^2}{m^2}$$



4.2 Hashing with chaining

4.2.1 Unordered sets

If we want to have a data structure which can maintain inserts, deletes and member queries on an unordered set, then we can do so with a time complexity of $\mathcal{O}(1)$ for all operations with a hash table with chaining.

A van Emde Boa tree works on ordered sets and supports the operations in $\mathcal{O}(\log \log U)$

4.2.2 Hashing with chaining

The idea with hashing with chaining is to create an array L, of length m, such that $m \geq n$, and a universal hash function: $h : U \to [m]$.
Every index of L is a linked list over the set of elements which share the same hash value according to the index: $L[i] =$ Linked list over $\{y \in S | h(y) = i\}$

Each operation on the set takes constant time to index the array + the time it takes to get to the element stored in the linked list at the index: $\mathcal{O}(|L[h(x)]| + 1)$.

4.2.3 Estimated size of a linked list

For elements which we have not stored yet, we can estimate the size of the linked list at the index for the element as:

$$x \notin S, \mathbb{E}_h\left[|L[h(x)]|\right] \leq 1$$

Proof: We can show that it is less than or equal to 1 via. the following:

We can replace the inner part of the estimation with the set S of elements stored which have the same "hash value" as the element x:

$$\mathbb{E}_h\left[|L[h(x)]|\right] = \mathbb{E}_h\left[|y \in S | h(y) = h(x)|\right]$$

The size of the set is the same as the sum over the elements $y \in S$ where $h(y) = h(x)$ is true, or 1 in Iverson Bracket notation: $[h(y) = h(x)]$:

$$= \mathbb{E}_h\left[\sum_{y \in S}[h(y) = h(x)]\right]$$

We can rewrite this with Linearity of expectation which says:
$\mathbb{E}_h\left[\sum_i X_i\right] = \sum_i \mathbb{E}_h\left[X_i\right]$. Thus we get:

$$\mathbb{E}_h\left[\sum_{y \in S}[h(y) = h(x)]\right] = \sum_{y \in S}\mathbb{E}_h\left[[h(y) = h(x)]\right]$$

Now because we can use the expectation of an indicator variable: $\mathbb{E}_h\left[X\right] = Pr[X = 1]$. We can rewrite the expression to use the probability of a collision:

$$\sum_{y \in S}\mathbb{E}_h\left[[h(y) = h(x)]\right] = \sum_{y \in S} Pr_h[h(y) = h(x)]$$

Then because we know that we have a universal hash function with collision probability of $\frac{1}{m}$, and we store $|S| = n$ elements, which gives us n elements which can have the possibility of having the same hash value, we get:

$$\sum_{y \in S} Pr_h[h(y) = h(x)] \leq \frac{|S|}{m} = \frac{n}{m} \leq 1$$

4.3 Multiply-mod-prime

The Multiply-mod-prime hashing scheme is defined by having a universe of $U = [u]$, then picking a prime $p \geq u$. Then it holds that for $a, b \in [p]$ which are picked uniformly at random and $m < u$, a hash function is defined as:

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

This hash function is 2-approximately **strongly** universal.

4.4 Multiply-shift

Multiply-shift scheme is a hashing scheme which makes use of operations which are fast on computers like right shift, and overflow characteristics of multiply operations (a x-bit number cannot overflow past the x'th bit, thus removing the need to modulus some number to remove the overflow).

4.4.1 2-approximately universal multiply-shift

We define a universe $U = [2^w]$ and $m = 2^l$, then for any odd number $a \in [2^w]$ we can define the hash function:

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

Then if we chose uniformly at random an odd $a \in [2^w]$, then the hash function is 2-approximate universal.

4.4.2 Implementation:

If we chose a universe of $U = [2^{64}]$, then if we make sure to only work with unsigned integers of 64 bits, then we do not need to use the modulus operator, as the overflow is already discarded. Now the only thing that is left to do is to divide ax by 2^{64-l}, this can be done by rightshifting by $64 - l$, thus we get the following pseodocode:

```
fun(a: u64, x: u64, l: u64): u64 {
    return (a * x) >> (64 - l)
}
```

4.4.3 strongly universal multiply-shift

We define a universe $U = [2^w]$ and $m = 2^l$, and then pick $\overline{w} \geq w + l - 1$. Then for any pair $(a, b) \in [2^{\overline{w}}]^2$ we can define the hash function:

$$h_{a,b}(x) := \left\lfloor \frac{(ax + b) \bmod 2^{\overline{w}}}{2^{\overline{w}-l}} \right\rfloor$$

If we chose uniformly at random $a, b \in [2^{\overline{w}}]^2$, then the hash function is strongly universal.

4.4.4 Implementation:

Now because we are restricted in $l \leq w$, then we can chose $w = 32$ and $\overline{w} = 64$, such that $U = [2^{32}]$. Then using the same tricks as before we get the following pseodocode:

```
fun(a: u64, b: u64, x: u32, l: u32): u32 {
    return (a * x + b) >> (64 - l)
}
```

4.5 Applications

4.5.1 universal hashing - signatures

If we want to assign a signature to each $x \in S \subseteq U$, then we can use a hash function $s : U \rightarrow [n^3]$, where $n = |S|$.

Proof: The probability that two elements have the same signature, when signed is:

$$Pr_s[\exists\{x, y\} \subseteq S | s(x) = s(y)]$$

This is the same as the sum of the probability that 2 elements drawn from the set S has the same signature:

$$\leq \sum_{\{x,y\} \subseteq S} Pr_s[s(x) = s(y)]$$

The number of possible sets of 2 which we can make from S is: $\binom{n}{2}$. Thus we get:

$$\leq \frac{\binom{n}{2}}{n^3} = \frac{1}{2n}$$

As we can see there is a "high probability" of having **no** collisions.

4.5.2 strongly universal hashing - Coordinated sampling

Coordinated sampling is the idea that a bunch of agents can observe a subset of events from a universe, and can only store a subset S_i of the observed events. We denote $A_i \subseteq U$ as the event observed by agent i. Thus $S_i \subseteq A_i \subseteq U$
If every agent just picks a random sample of events to store, and then make a decision based on the stored events, then there is little probability of them making the same decision, as the samples are incomparable.

However if all agents make the same decision about what events to store, then we get that the intersection or union of the stored samples of the events from 2 different agents is a sample of the intersection or union of the events. Thus it holds that:

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

and

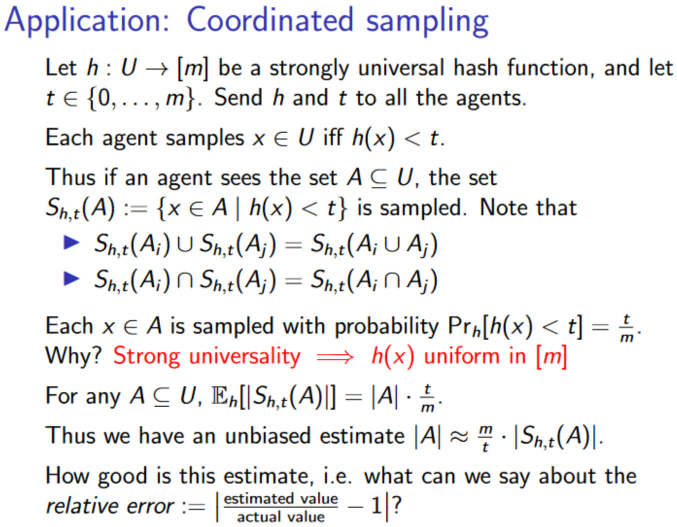
$$S_i \cap S_j \text{ is a sample of } A_i \cap A_j$$

To get this behaviour, the agents can decide to only store elements which hash to a value which is less than a given threshold t. Thus each agent has the same threshold $t \leq m$.

Thus if we use a strongly universal hash function, then we get that the probability of an element $x \in A$ is sampled is:

$$Pr_h[h(x) < t] = \frac{t}{m}$$

For the rest of the proof, check the slides:



Application: Coordinated sampling

Let $h : U \to [m]$ be a strongly universal hash function, and let $t \in \{0, \dots, m\}$. Send h and t to all the agents.

Each agent samples $x \in U$ iff $h(x) < t$.

Thus if an agent sees the set $A \subseteq U$, the set $S_{h,t}(A) := \{x \in A \mid h(x) < t\}$ is sampled. Note that

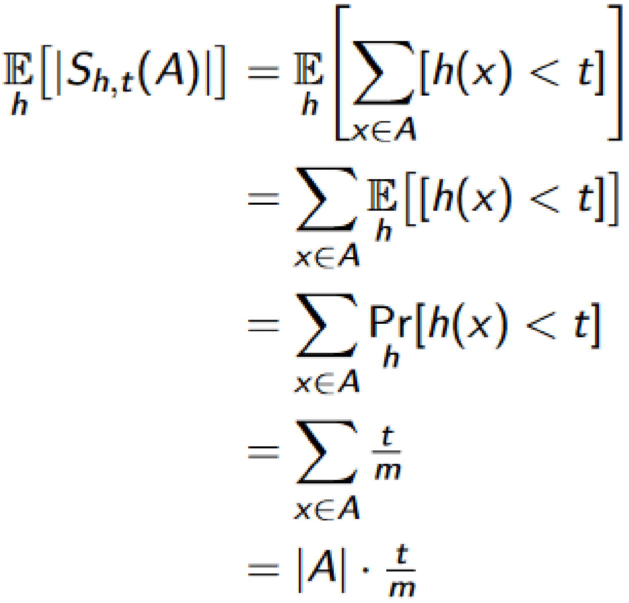
- $S_{h,t}(A_i) \cup S_{h,t}(A_j) = S_{h,t}(A_i \cup A_j)$
- $S_{h,t}(A_i) \cap S_{h,t}(A_j) = S_{h,t}(A_i \cap A_j)$

Each $x \in A$ is sampled with probability $\Pr_h[h(x) < t] = \frac{t}{m}$.
Why? Strong universality $\implies$ $h(x)$ uniform in $[m]$

For any $A \subseteq U$, $\mathbb{E}_h[|S_{h,t}(A)|] = |A| \cdot \frac{t}{m}$.

Thus we have an unbiased estimate $|A| \approx \frac{m}{t} \cdot |S_{h,t}(A)|$.

How good is this estimate, i.e. what can we say about the *relative error* $:= \left|\frac{\text{estimated value}}{\text{actual value}} - 1\right|$?



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

Concentration bound

Lemma

Let $X = \sum_{a\in A} X_a$ where the X_a are pairwise independent 0–1 variables. Let $\mu = \mathbb{E}[X]$. Then $\mathrm{Var}[X] \le \mu$, and for any $q > 0$,

$$\Pr[|X-\mu| \ge q\sqrt{\mu}] \le \tfrac{1}{q^2}$$

Proof (not curriculum).

For $a \in A$ let $p_a = \Pr[X_a = 1]$. Then $p_a = \mathbb{E}[X_a]$ and

$$\begin{aligned}\mathrm{Var}[X_a] &= \mathbb{E}[(X_a-p_a)^2] = (1-p_a)(0-p_a)^2 + p_a(1-p_a)^2\\ &= (p_a^2 + p_a(1-p_a))(1-p_a) = p_a(1-p_a) \le p_a\\ \mathrm{Var}[X] &= \mathrm{Var}\Big[\sum_{a\in A} X_a\Big] = \sum_{a\in A}\mathrm{Var}[X_a] \le \sum_{a\in A} p_a = \mu\end{aligned}$$

Finally, since $\sigma_X = \sqrt{\mathrm{Var}[X]} \le \sqrt{\mu}$ we get:

$$\begin{aligned}\Pr[|X-\mu| \ge q\sqrt{\mu}] &\le \Pr[|X-\mu| \ge q\sigma_X]\\ &\le \tfrac{1}{q^2} \qquad \text{(Chebyshev's ineq.)} \ \square\end{aligned}$$

Application: Coordinated sampling

Let's apply this lemma to the estimate $|A| \approx \frac{m}{t}|S_{h,t}(A)|$ from our coordinated sampling.
Let $X = |S_{h,t}(A)|$ and for $a \in A$ let $X_a = [h(a) < t]$. Then $X = \sum_{a\in A} X_a$ and for any $a, b \in A$, X_a and X_b are independent. Also, let $\mu = \mathbb{E}_h[X] = \frac{t}{m}|A|$.
Then for any $q > 0$,

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

We needed strong universality in two places for this to work.
Where? h must be uniform to get unbiased estimate, and pairwise independent for the lemma.

5 NP

5.1 Disposition

- Languages, decision problem, verification
- P. NP Class definition
- Reductions
- NP-Hard, NP-Complete definitions
- Clique $\in$ NPC
- TSP $\in$ NPC

5.1.1 3-CNF-SAT example

$$\varnothing = (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3)$$

5.2 Definitions

5.2.1 Languages

An **alphabet** is a finite set $\sum$ of symbols.

A language L over $\sum$ is a set of strings of symbols from $\sum$. It can also contain the empty string ϵ.
If there are no strings in the language, not even the empty string, we say it is empty $\varnothing$.
The language containing all strings made from $\sum$ is denoted by $\sum^*$.
Any language over $\sum$ is a subset of $\sum^*$.

An example of an language is:
if we have the alphabet: $\sum = \{a, b\}$, then we can make the language: $L = \{\epsilon, a, b, ab, ba, aaab, ...\}$.

The **complement** of L is denoted by $\overline{L}$ and is defined by: $\overline{L} = \sum^* - L$, i.e. the strings composed of the alphabet which are not in L.

5.2.2 Decision problems

A decision problem is some problem, that when given a problem outputs a yes or a no for if the problem is true or false. i.e. we want problems like is there a clique of 5? **NOT:** What is the largest clique?

You can think of a decision problem as a function which takes a language and outputs yes/1 or no/0.

5.2.3 Reductions

We can say that a language L_1 can be reduced to another language L_2 if there exists an efficient (polynomial time algorithm) which can compute a function $f : \sum^* \rightarrow \sum^*$ such that:

Every string x which is in L_1, is also in L_2:

$$\forall x \in L_1 \Rightarrow f(x) \in L_2$$

The inverse of the above:

$$\forall x \notin L_1 \Rightarrow f(x) \notin L_2$$

We can use this to say the following:
If there exists an efficient algorithm A to solve L_2, and there exists an efficient reduction f such that $L_1 \leq L_2$, then we can solve L_1 by reducing to L_2 and then applying A. i.e. $A(f(x))$ is a solution to L_1.

We can also say that if we know a reduction is possible and that L_1 is a hard problem, then we know that L_2 must be at least as hard as L_1.

5.3 Complexity Classes

5.3.1 P

P is a class of problems which can be **solved** efficiently, i.e. with a polynomial time algorithm.

5.3.2 NP

Problems for which solutions/certificate can be **verified** in polynomial time. Formally we can say that a language L is in NP if there exists a polynomial time verification algorithm A, which takes to inputs, x, y such that:

$$x \in L \Leftrightarrow \exists y \in \sum^{*} \text{ and the length of } y, |y| = \mathcal{O}(|x|^c) : A(x, y) = 1$$

where y is a certificate for x.

We can say that P is a subset of NP as we can verify all problems in P in polynomial time by running the algorithm to solve P. i.e. P $\subset NP$

5.3.3 co-NP

The definition of co-NP or compliment NP is defined by the set of languages for which the compliment is in NP. i.e. $L \in$ co-NP iff. $\overline{L} \in$ NP:

$$\text{co-NP} = \{L : \overline{L} \in NP\}$$

Now because P is closed under complement, which means that if $L \in P \Rightarrow \overline{L} \in P$ and because $P \subset NP$, then we get P $\subset$ co-NP

5.3.4 NP-complete

NP-complete is the set of languages for which the language is in NP:

$$L \in NP$$

and is at least as hard as every other language is NP, i.e. can be reduced to any other language in NP:

$$L' \leq_P L \text{ for every } L' \in NP$$

5.3.5 NP-hard

NP hard languages are languages which are at least as hard as any other language in NP. i.e. they can be reduced to any other language in NP, but might not itself be in NP.

$$L' \leq_P L \text{ for every } L' \in NP$$

In other words, NP-hard problems cannot be verified in polynomial time, but can be reduced in polynomial time to a problem which can be verified in polynomial time.

5.4 NP-completeness proofs

5.4.1 How to prove NP-completeness

To prove that $L \leq NPC$ we need to prove the following:

$$L \in NP$$

$$L' \leq L \text{ for every } L' \in NP$$

A guide to do so is as follows:

1. Prove $L \in NP$.
2. Select a known NP-complete language L' to use for the reduction.
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.

4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.

5. Prove that the algorithm computing f runs in polynomial time.

5.4.2 Clique $\in$ NP-complete

The Clique problem is a problem of finding a clique (subset of vertices which are fully connected with edges) within a graph. We can formulate a desition problem by asking if a given graph has a clique of size k. Thus the formal deffinition becomes:

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

CLIQUE $\in$ NP:
To show that CLIQUE is in NP we show that for a given graph, we can use a subset of the vertices corresponding to the clique, as a certificate. We accept the certificate if the size of the vertex set is equal to k. Furthermore we check that each vertex pair of the vertex set, has a corresponding edge in the graph.

3-CNF-SAT $\leq_P$ CLIQUE:
Given an instance of 3-CNF-SAT with k boolean formulas such that: $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$. For each clause C_r, $1 \leq r \leq k$, we have 3 literals l_1^r, l_2^r, l_3^r. We will construct the graph such that the formula is only satisfying iff. G has a clique of size k. We can do so by adding a vertex v_i^r, to V, for each literal l_i^r. Then for each v_i^r in C_r we add an edge to a vertex v_j^s if $s \neq r$ and if the two literals are not the negation of each-other.

3-CNF-SAT $\Rightarrow$ CLIQUE:
We know that for the boolean formula to be a satisfying assignment, then at least one literal must be true from each clause. If we pick one true literal from each clause we get a subset of the vertices, and this subset is of size k and has a clique of this size. We know this as by the construction, each vertex pair which are not composed of the same literal negated and non-negated, and in the same clause gets an edge.

3-CNF-SAT $\Leftarrow$ CLIQUE:
We can also show that a k clique will be a satisfying assignment to the k-clauses in the boolean formula for 3-CNF-SAT. We know that there are no edges for vertices of the same clause, which makes sure that the clique connects all other clauses. We also know that if a vertex is in the k-clique subset of vertices, then we know that it does not have a corresponding negative **and** non-negative literal in the vertex set, as no such edges are allowed. Thus it would not be part of the clique. Thus we can see that a clique of size k is a satisfying solution to the size k boolean formula.

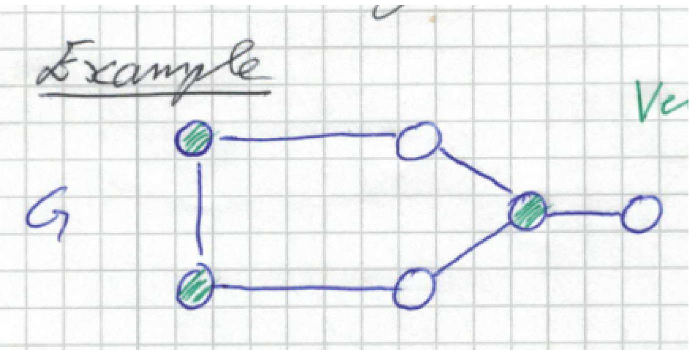
5.4.3 Vertex-Cover $\in$ NP-complete

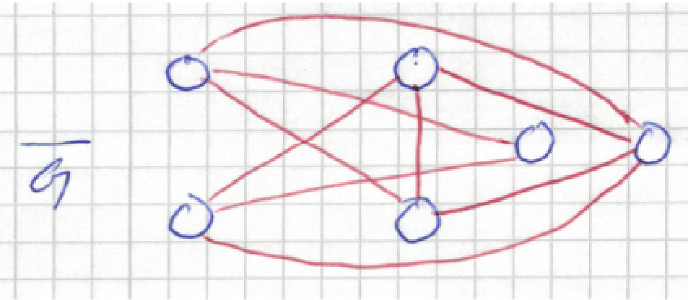
The vertex cover problem, is the problem of finding a set of vertices, for which all incident edges to the vertices are covered. The decision problem is the problem of finding such a cover of size k. We can formulate this as:

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

VERTEX-COVER $\in$ NP:
If we are given a graph G and a size of the cover k, then we can use a certificate of $V' \subseteq V$ for which we can check if it is a vertex cover of size k. We start by checking that $|V'| = k$. Then we can go through each edge (u, v) of the graph and then remove it if it has a corresponding u or v in V'. If all edges are removed, then we know that we have a vertex-cover of size k. This is all doable in polynomial time, thus VERTEX-COVER $\in$ NP.

CLIQUE $\leq_P$ VERTEX-COVER:
We construct a complement graph $\overline{G}$ which is composed of the same vertices, but with an edge set $V \times V \setminus E$. i.e. if an edge is in G, then it is not in $\overline{G}$ and vice versa.
Then we say the if there is a vertex cover of size k in G, then there is a clique of size $|V| - k$ in $\overline{G}$.





CLIQUE $\Rightarrow$ VERTEX-COVER:
If we have a Clique $c \subseteq V$ in G of size k, then we know that the edges of the clique are not edges in $\overline{G}$. Then we also know that all edges in $\overline{G}$ has an endpoint in the vertices not contained within the clique, $V \setminus c$. Thus we know that this is a vertex cover of size $|V| - k$ **CLIQUE $\Leftarrow$ VERTEX-COVER:**
If we have a vertex cover c^* in $\overline{G}$, then the set $V \setminus c^*$ does not have any edges in $\overline{G}$. This means that all edges which end in $V \setminus c^*$ are in G. Thus if c^* is a vertex-cover of size $|V| - k$, then $V \setminus c^*$ is a clique in G of size $|V| - (|V| - k) - k = k$.

6 Exact exponential and parameterized algorithms

6.1 Disposition

- Motivation
- Exponential and Parameterized algorithms
- Kernelization
- k-vertex-cover parameterized
- FTP vs XP
- Brute force exponential algorithm solution
- TSP with dynamic programming???

6.2 Introduction

6.2.1 Goals of algorithms

We usually want algorithms that:

1. runs in polynomial time
2. run on all inputs
3. finds an exact solution

This is now feasible for all problems, thus we can relax some of these constraints.

Exact exponential algorithms:
Allows exponential time - relaxes 1.

Parameterized algorithms:
Only small inputs of some parameter can be solved, i.e. big inputs take too long. - relaxes 2.

Approximation algorithms:
Approximates a solution - relaxes 3.

6.2.2 Brute force NP algorithm

We know that if we have a problem in NP, then we can check the certificate in polynomial time via a function $m(x) = \mathcal{O}(x^c)$.
We can use this to brute force a solution by trying all potential certificates that can be verified in polynomial time, and then checking if any one of them is a solution to the problem instance.

The certificate is a bit string of a most 2 to the power of a polynomial: $2^{\mathcal{O}(x^c)}$ with x being the problem instance. We also know that we can verify a certificate in polynomial time $\mathcal{O}(x^c)$. Thus we can brute force any NP problem in: $\mathcal{O}(2^{(m(x))} \cdot x^c)$.

6.2.3 $\mathcal{O}^*(\cdot)$ notation

When comparing the runtime of exact exponential algortihms the biggest factor by far is the basis for the exponential, i.e. the a in $\mathcal{O}(a^n)$. This is because if you have any exponential multiplied with a polynomial expression, then a larger exponential base will always be bigger in terms of big O notation:

$$0 < a < b, c \in \mathbb{R} : \mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n)$$

Thus we can define the notation $\mathcal{O}^*(\cdot)$ as a notation which ignores any polynomial factor:

$$0 < a < b, c \in \mathbb{R} : \mathcal{O}(a^n) \subset \mathcal{O}^*(a^n) \subset \mathcal{O}(b^n)$$

Using this notation the brute force solution goes from $\mathcal{O}(2^{(m(x))} \cdot x^c)$ to $\mathcal{O}^*(2^{(m(x))})$.

6.3 Exact exponential algorithms

6.3.1 TSP with dynamic programming

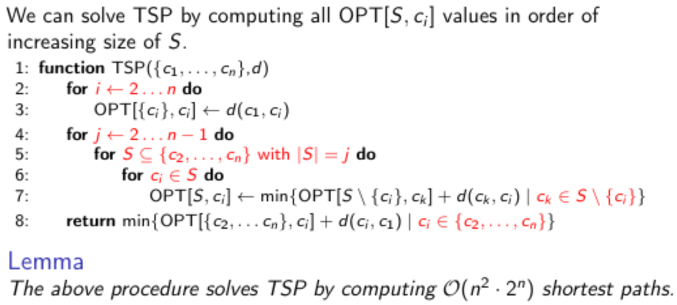
The traveling salesman problem is the problem of finding the tour through n cities with distances, which has the smallest total distance.

We can formulate the problem as the problem of finding a permutation of the cities $C_1, ..., C_n$ for which we minimize the distance $d(C_i, C_j)$ such that we minimize the distance traveled in the tour: $d(C_n, C_1) + \sum_{i=1}^{n-1} d(C_i, C_i + 1)$.

The idea is to memoization such that for all subsets of $\{c_2, .., c_n\}$ and $C_i \in S$ we can store the optimal distance to C_i through the set $S \cup C_1$, such that we start in C_1, then visits all cities in S and end in C_i. We store this value in $\text{OPT}[S, C_i]$ such that:

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

We can see that this recursively finds the shortest path by finding the shortest path to c_k and adding the distance $d(c_k, c_i)$ which gives the shortest path to c_i.



We can solve TSP by computing all OPT$[S, c_i]$ values in order of increasing size of S.

```
1: function TSP($\{c_1, \ldots, c_n\}, d$)
2:     for $i \leftarrow 2 \ldots n$ do
3:         OPT$[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
4:     for $j \leftarrow 2 \ldots n-1$ do
5:         for $S \subseteq \{c_2, \ldots, c_n\}$ with $|S| = j$ do
6:             for $c_i \in S$ do
7:                 OPT$[S, c_i] \leftarrow \min\{\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
8:     return $\min\{\text{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$
```

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof of lemma: From the code above, we can prove the running time. We see that in line 4-5 we make the permutations, for which there are the sum of 2 to $n-1$ lengths to make permutations from a set of size n. And because we do not include c_1, then we start at 2 (line 4). For each subset of length j, we can make $\binom{n-1}{j}$ combinations (line 5). Thus for lines 4-5 the number of sets we produce is:

$$\sum_{j=2}^{n-1} \binom{n-1}{j}$$

Then in line 6 we go trough each city c_i in the set S, which is the size of j. Then in line 7, we find the minimum path to that city by making $j-1$ lookups in OPT. Thus for lines 6-7 we get:

$$\sum_{i=1}^{j} (j-1)$$

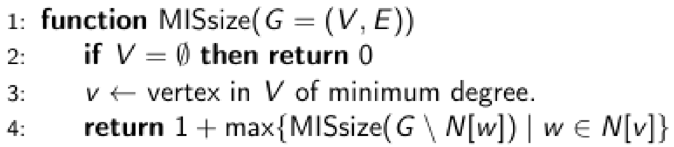
When we combine the number of sets with the number of operations we do per set we get:

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \cdot \sum_{i=1}^{j} (j-1)$$

We can upper bound the operations pr set to be n^2 as the max value of j is $n-1$:

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \cdot \sum_{i=1}^{j} (j-1) \leq \sum_{j=2}^{n-1} \binom{n-1}{j} \cdot n^2$$

Furthermore we can upper bound the number of sets produced to be the sum from $j = 1$ to n and $\binom{n}{j}$:



```
1: function MISsize($G = (V, E)$)
2:     if $V = \emptyset$ then return 0
3:     $v \leftarrow$ vertex in $V$ of minimum degree.
4:     return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$
```

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \cdot n^2 \leq \sum_{j=1}^{n} \binom{n}{j} \cdot n^2 = 2^n \cdot n^2$$

Thus we get that the running time is: $\mathcal{O}(2^n \cdot n^2)$ or $\mathcal{O}^*(2^n)$

6.3.2 Maximum Independent Set

The problem of finding the maximum independent set, is defined as given an undirected graph, we must find the largest set $I \subseteq V$ such that each edge has at most one endpoint in I.

Naive apporach:
Try all subsets of V. There are $2^{|V|}$ subsets, thus it would take $\mathcal{O}^*(2^{|V|})$.

However if we notice that the closed neighborhood of any vertex $v \in V$ intersected with I cannot be empty: $N[v] \cap I \neq \varnothing$.
The definition of the closed neighborhood is as follows:

$$N[v] := \{v\} \cup \{w \in V | (v, w) \in E\}$$

This is because if it where empty, then there would exists an edge which was not covered by I, as the intersection would be empty meaning that a vertex in the closed neighborhood of v should be included in I, as there is an edge which does not have an endpoint in I unless v is included in I.

With this observation we can develop the following algorithm:
The algorithm takes the smallest degree vertex (line 3), and then goes through each vertex w in the closed neighborhood and tries to find the maximum independent set of the graph minus the closed neighborhood of w, as all the edges are covered by w.

We can show the maximum number of sub-problems (recursive calls) $T(n)$, for a graph with n vertices to be:
When there are no vertices in the graph, then there is 1 recursive call:

$$T(0) = 1$$

When there are n vertices in the graph, then we have 1 + the number of recursive calls. We make 1 recursive call for each vertex in the closed neighborhood to v. This gives us the sum over $w \in N[v]$. Then for each recursive call, we remove all the neighbors of w, which is a set of size $d(w) + 1$(degree of w + 1 as w is a part of $N[w]$). Thus we get:

$$T(n) \leq 1 + \sum_{w \in N[v]} T(n - (d(w) + 1))$$

We can upper bound this by the size of the closed neighborhood of v, as v is chosen to be the vertex with the smallest degree. The closed neighborhood of: v has a size of $|N[v]| = d(v) + |\{v\}| = d(v) + 1$. Thus we replace the sum over the closed neighborhood:

$$= 1 + (d(v) + 1) \cdot T(n - (d(w) + 1))$$

We can the upper bound this by removing $d(v)$ vertices in each recursive call instead of $d(w)$, as we know that $d(v) < d(w)$:

$$\leq 1 + (d(v) + 1) \cdot T(n - (d(v) + 1))$$

Then we can replace the expression $d(v) + 1$ with s, and we get:

$$\leq 1 + s \cdot T(n - s)$$

There is a proof in the slides which assume that s is static in each recursion, however it changes with each recursion. The proof for this is in assignment 4, ex. 2.

6.4 Parameterized problems

6.4.1 Kernelization

The idea is to pre-process the problem instance, such that we can reduce the problem such that the size of the smaller problem ideally depends on a parameter k such that $k < n$. The smaller problem is called a kernel, thus we call the process of finding it kernelization.

6.4.2 k-vertex cover

The k-vertex cover problem is asking if there exists a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has an endpoint in C.

By some clever tricks we use kernelization to reduce the problem though the following steps:

We can drop every vertex with a degree of 0, as no edge has an endpoint in such a vertex.

We can see that we must include v in the vertex set, if $d(v) > k$ as to cover all edges, we must include v in the set. Thus we can remove v and reduce k by 1 for each v removed.

If we have that the degrees of any vertex is at most k and the number of edges is $|E| > k^2$, then we know that we cannot solve the problem, as we cover at most k edges per vertex. Thus we chose k vertecies each covering k edges without overlap, we would not be able to cover more than k^2 edges, thus not being able to solve the problem.

Doing this kernelization we get that the number of vertecies left in the reduced graph is the sum of 1 over the vertecies:

$$|V| = \sum_{v \in V} 1$$

This is bounded by the degree of each vertex. Now the sum of the degree over $v \in V$ is $2|E|$ because each $d(v)$ is the number of edges incident to v, then they must be counted again when we reach the neighbors of v. Thus we get:

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

And we know that this is then bounded by the maximum number of edges of k^2, which gives us:

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

However we can do better. If we notice that when the closed neighborhood of v only consists of 2 elements $N[v] = \{v, w\}$, then we know that we must include v or w, but not both to cover the edge (v, w). Knowing this we can drop both vertecies from the graph and then reduce k by 1. This makes the **mimimum** degree of each vertex to be 2. Thus we can get the following bound:

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

The running time to produce the kernel is $\mathcal{O}(|V| + |E|)$. We know that there are $\binom{k^2}{k}$ possible solutions, and that we can find a solution in $\mathcal{O}(k^2)$ thus we get that the running time for kernelization + finding a solution is: $\mathcal{O}(|V| + |E| + \binom{k^2}{k} k^2) \subseteq \mathcal{O}(|V| + |E| + k^{2k+2})$

6.4.3 Improving k-vertex cover with Bounded search trees

We know that when we have an edge, at least one of the endpoint must be in the solution and the other must be rejected from the graph, as it is covered by the other. The idea of a bounded search tree is to pick any edge, and then try to solve the sub-graph with the removed vertex. Do this with a recursion depth of at most k and you get that we have at most 2^k subproblems. If we start by kernelizing the graph like in the previous section. Then we get that there are at most k^2 edges, $|E| \leq k^2$. With the time it takes to kernelize, we get a running time of:

$$\mathcal{O}(|E| + |V| + k^2 \cdot 2^k)$$

6.5 FPT & XP

6.5.1 Fixed Parameter Tractable (FPT)

FPT is defined by algorithms with a running time of $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$.

6.5.2 Slice-wise Polynomial (XP)

XP is defined by algorithms with a running time of $f(k) \cdot n^{g(k)}$, for some functions f, g.
Thus if $g(k) = c$ then $FPT \subset XP$.

k-clique is XP:
If we use the simple brute force algorithm to solve k-clique problem, then there are $\binom{n}{k} \leq n^k$ possible k-sized subsets of vertecies. We can check if a subset is a clique in $\mathcal{O}(k^2)$ time. Thus we get a total running time of $\mathcal{O}(k^2 \cdot n^k)$ which is in XP, if $f(k) = k^2$ and $g(k) = k$ for the definition of XP: $f(k) \cdot n^{g(k)}$.

7 Approximation Algorithms

7.1 Disposition

- Motivation
- approximation ratio
- Minimum vertex cover 2-approxmiate proof
- maximum 3-CNF with random assignments
- 8/7 approximation proof

7.2 Introduction

7.2.1 Why Approximation Algorithms?

The goals of algorithms is to:

1. have a reasonable time complexity

2. Work on all inputs

3. give precise results

Approximation algorithms relaxes the last part, as they approximate to some results, usually to reduce the time complexity.
This enables us to work on optimization problems, as in contrast to decision problems, as we can try and approximate the optimal solution.

7.2.2 Approximation Ratio

We can define the approximation ratio $\rho(n)$ as:

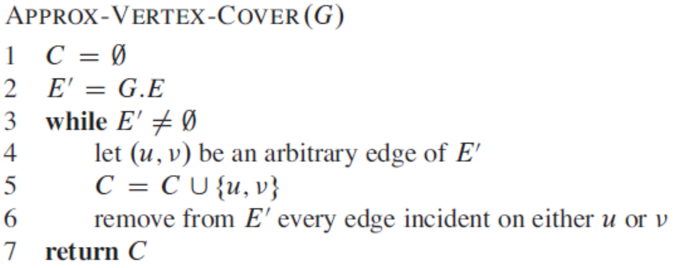
$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n)$$

where C is the approximated solution
C^* is the optimal solution.

When we have a minimization problem, the C^* becomes the smallest, thus the left side is the largest fraction. It's the right side for maximization problems.

7.3 Minimization problems

7.4 Minimum vertex cover



```
APPROX-VERTEX-COVER($G$)
1  $C = \emptyset$
2  $E' = G.E$
3  while $E' \neq \emptyset$
4      let $(u, v)$ be an arbitrary edge of $E'$
5      $C = C \cup \{u, v\}$
6      remove from $E'$ every edge incident on either $u$ or $v$
7  return $C$
```

We can see in the pseudo-code that we pick any edge of the graph, then add the vertices to out solution set C and then remove all edges connected to u and v. repeat until no more edges.
This means we have a running time of $\mathcal{O}(|E| + |V|)$, as we need to repeat until no more edges, and because we need to keep track of vertices.

7.4.1 Proof 2-approximation

We let A denote the set of edges chosen in line 4, then we know that because we remove all other edges incident to the endpoint, that those edges cannot be chosen at some other step. From this we know that every edges, with endpoint

$(u, v) \in A$ then u or v must be a part of the optimal vertex cover C^*.
Thus we know that the optimal solution is equal or greater than the size of A:

$$|C^*| \geq |A|$$

And because we always chose 2 vertices for every picked edge, we know that our solution is twice as large as A:

$$|C| = 2 \cdot |A|$$

or we can write it as:

$$\frac{|C|}{2} = |A|$$

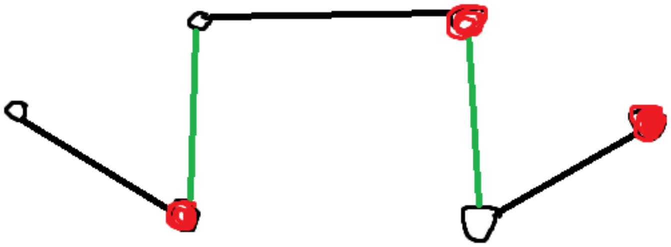
Thus we can replace a with $\frac{|C|}{2}$:

$$|C^*| \geq \frac{|C|}{2} = |A|$$

From this we get that the approximated solution is a 2-approximate solution:

$$2 \geq \frac{|C|}{|C^*|}$$

7.4.2 Exstra: Why $|C^*| \geq |A|$ and not $|C^*| = |A|$



If we have the above graph, and choose A to be the green edges, then we have $|A| = 2$. However to cover all the edges in the graph, we must have at least 3 vertices, like the red ones. Thus we have an optimal solution of size 3, and we get:

$$3 = |C^*| \geq |A| = 2$$

7.5 Traveling salesman problem with triangle inequality

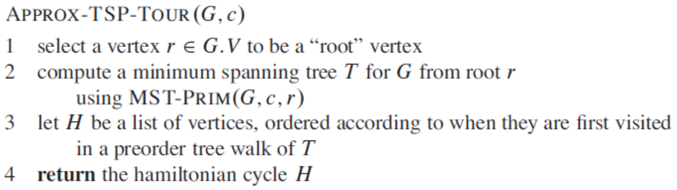
In the traveling salesman problem, the goal is to find a cycle through all vertices which has the lowest cost.
We assume that the triangle inequality holds for the cost function. I.e. when

we calculate the cost of going from one vertex to another, then the following holds:

$$c(u, w) \leq c(u, v) + c(v, w)$$

This can be interpreted as the route directly from u to w costs less-than-or-equal-to going through v in the way to w.

7.5.1 Algorithm



```
APPROX-TSP-TOUR$(G, c)$
select a vertex $r \in G.V$ to be a "root" vertex
compute a minimum spanning tree $T$ for $G$ from root $r$
    using MST-PRIM$(G, c, r)$
let $H$ be a list of vertices, ordered according to when they are first visited
    in a preorder tree walk of $T$
return the hamiltonian cycle $H$
```

The algorithm works by creating a minimum spanning tree from any vertex as root.
Then we walk the spanning tree, and make a list ordered by when each vertex is visited. However if we meet a vertex we have already seen, then we skip that vertex. Lastly we make sure that we end in the starting vertex.
This list of vertices is a cycle through all the vertices, and thus an approximate solution for TSP.

The running time of this is the cost to create the minimum spanning tree + making the list of vertices: $\Theta(|V|^2)$

7.5.2 Proof of 2-approximation

We know that the minimum spanning tree is not a cycle, and though it visits every vertex it does not constitute at solution. Thus we know that the cost of the edges in the minimum spanning tree, $c(T)$ must be less than the cost of the optimal solution $c(H^*)$.

$$c(T) \leq c(H^*)$$

If we walk every edge in the minimum spanning tree twice, then we get a cycle through every edge twice. Thus we can conclude that the cost of the full walk $c(W)$ is equal to twice the cost of the edges in the minimum spanning tree:

$$c(W) = 2 \cdot c(T)$$

This means that it is less than or equal to twice the cost of the optimal path:

$$c(W) = 2 \cdot c(T) \leq 2 \cdot c(H^*)$$

However the resulting path H does not walk through every edge twice, it takes shortcuts (Thus triangle inequality applies), which means we know that the cost of the resulting path is less than or equal to the walk though the spanning tree edges:

$$c(H) \leq c(W)$$

From this we know that the resulting path is less than or equal to twice the cost of the optimal path as

$$c(H) \leq c(W) = 2 \cdot c(T) \leq 2 \cdot c(H^*)$$

Or simply:

$$c(H) \leq 2 \cdot c(H^*)$$

Thus in the end, if $C = c(H)$ and $C^* = c(H^*)$

$$C \leq 2 \cdot C^*$$

Which is:

$$\frac{C}{C^*} \leq 2$$

And we have shown that it is 2-approximate.

7.6 MISSING: Subset sum

7.7 Weighted vertex cover

7.7.1 Problem

You are given a graph, with each vertex having a positive weight associated with the vertex, $w(v)$. Then the problem is finding a vertex cover C, which has the minimum weight, such that:

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

We can formulate the problem as a 0-1-integer linear programming problem, such that each vertex has a variable x_v, which can be 0 or 1. If it is 1, then we say it is a part of the cover C:

$$x_v \in \{0, 1\}, \forall v \in V$$

$$x_v = 1 \Leftrightarrow v \in C$$

We must also include at least one vertex from each edge, otherwise we would not have a cover. Thus the sum of the two vertices of an edge must be greater than or equal to 1, to constitute that one of them is included:

$$x_u + x_v \geq 1, \forall u, v \in E$$

And the objective function which we want to minimize is the sum of the weights:

$$\text{minimize} \sum_{v \in V} w(v) \cdot x_v$$

However integer linear programming problems are still NP-complete.

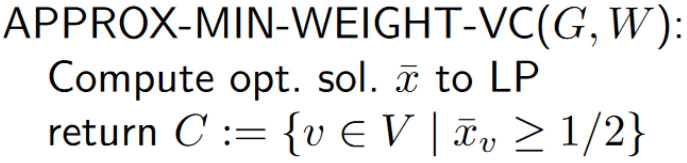
We can relax the problem by replacing the constraint of $x_v \in \{0, 1\}$ with the constraint $0 \leq x_v \leq 1$, which makes it a linear programming problem, which is solvable with the constraints:

$$0 \leq x_v \leq 1, \forall v \in V$$

$$x_u + x_v \geq 1, \forall u, v \in E$$

Note: that this relaxation makes the solution **smaller than or equal to** the non-relaxed integer linear program.

We can then develop an algorithm which computes the assignments of the variables to the linear program, and then creates a set of vertices which have a value higher than $\frac{1}{2}$:



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

There is nothing in the syllabus about a polynomial linear programming algorithm, but it exists. So the running time of the algorithm is the time it takes to solve the linear program + checking values of the vertices: $\mathcal{O}(PolyLP(V, E) + |V|)$

7.7.2 proof it is 2-approximate

We define a variable z^* to be the sum of the values $\overline{x}_v$ assigned to the variables in the linear program multiplied by the weight of the assiciated vertex:

$$z^* := \sum_{v \in V} \overline{x}_v \cdot w(v)$$

Not because the relaxed linear programming problem is smaller than the integer linear program problem, then we say that z^* is less than or equal to the weight of the minimum vertex cover C^*:

$$z^* \leq w(C^*)$$

Now we see that the weight of the approximate solution is:

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

We can write this as a sum over the vertices in V, using the Iverson bracket notation, with $[v \in C]$ being 1 if v is a part of C, zero otherwise:

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

Which we can replace $[v \in C]$ with $[\overline{x}_v \geq \frac{1}{2}]$ as this is the definition of being included in the solution set, as per the algorithm.

$$w(C) = \sum_{v \in C} w(v) = \sum_{v \in V} w(v)[v \in C] = \sum_{v \in V} w(v)[\overline{x}_v \geq \frac{1}{2}]$$

Now we see that:

$$[\overline{x}_v \geq \frac{1}{2}] \leq 2\overline{x}_v$$

We can show this as if $\overline{x}_v = 0$ then we get:

$$[0 \geq \frac{1}{2}] = 0 \leq 2 \cdot \overline{x}_v = 2 \cdot 0 = 0$$

And when $\overline{x}_v = 1$ then we get:

$$[1 \geq \frac{1}{2}] = 1 \leq 2 \cdot \overline{x}_v = 2 \cdot 1 = 2$$

Thus we know that it holds and we can use this to write the inequality:

$$w(C) = \sum_{v \in V} w(v)[\overline{x}_v \geq \frac{1}{2}] \leq \sum_{v \in V} w(v) \cdot 2\overline{x}_v$$

Because every term of the sum has 2 added, we can move it outside the sum:

$$\sum_{v \in V} w(v) \cdot 2\overline{x}_v = 2 \cdot \sum_{v \in V} w(v) \cdot \overline{x}_v$$

Then we can replace the sum with z^*:

$$2 \cdot \sum_{v \in V} w(v) \cdot \overline{x}_v = 2 \cdot z^*$$

and because we know it is bounded by $z^* \leq w(C^*)$ we get:

$$2 \cdot z^* \leq 2 \cdot w(C^*)$$

Thus we get:

$$w(C) \leq 2 \cdot w(C^*)$$

Which we can rewrite as:

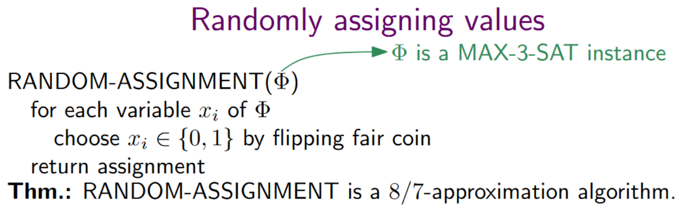
$$\frac{w(C)}{w(C^*)} \leq 2$$

Thus we get that it is an 2-approximation.

7.8 Maximization problems

7.9 Maximum 3-CNF with random assignments

The idea is for each variable we flip a coin, and assign it to true/1 or false/0. Then we return the assignments.



Randomly assigning values

Φ is a MAX-3-SAT instance

```
RANDOM-ASSIGNMENT($\Phi$)
  for each variable $x_i$ of $\Phi$
    choose $x_i \in \{0,1\}$ by flipping fair coin
  return assignment
```

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

This algorithms runs in $\mathcal{O}(n)$, where n is the number of variables.

7.10 Proof of 8/7-approximation

If we choose an assignment of a variable with probability of the variable being false: $\frac{1}{2}$. And each clause has 3 literals l_1, l_2, l_3. Then we know that the probability of a single clause being false is:

$$Pr[C_i = false] = Pr[l_1 = false] \cdot Pr[l_1 = false] \cdot Pr[l_1 = false] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \left(\frac{1}{2}\right)^3 = \frac{1}{8}$$

Thus the probability of a clause being true is:

$$Pr[C_i = true] = 1 - Pr[l_1 = false] \cdot Pr[l_1 = false] \cdot Pr[l_1 = false] = 1 - \frac{1}{8} = \frac{7}{8}$$

The number of satisfied clauses can be expressed as the sum over the Iverson bracket ($[C_i]$ is 1, when C_i is true, and 0 otherwise) of the clause being true. We call this stochastic variable X:

$$X = \sum_{i=1}^{n} [C_i]$$

Then the expected number of clauses being true is:

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

Then by linearity of expectation:

$$\mathbb{E}\left[\sum_{i=1}^{n} [C_i]\right] = \sum_{i=1}^{n} \mathbb{E}[C_i]$$

Then we can replace $\mathbb{E}[C_i]$ with the probability of C_i being true:

$$\sum_{i=1}^{n} \mathbb{E}[C_i] = \sum_{i=1}^{n} \frac{7}{8}$$

Then replace the sum, by n:

$$\sum_{i=1}^{n} \frac{7}{8} = n \cdot \frac{7}{8}$$

Thus the expected number of true clauses to be $n \cdot \frac{7}{8}$ which is the expected output of the algorithm.

$$C = n \cdot \frac{7}{8}$$

We know that the maximum number of true clauses is n. Thus we get an approximation of:

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

Thus we get that the algorithm is $\frac{8}{7}$-approximate

8 van Emde Boas Trees

8.1 Disposition

- Motivation: Ordered sets, Predecessor/successor
- van Emde Boas Tree structure (drawing)
- Running time analysis
- Space analysis
- improvements (to space complexity)

8.2 Notes

HUSK: Det er $\mathcal{O}(loglog|U|)$ ikke $\mathcal{O}(loglogN)$. HUSK: $w = log|U|$ pgs. $U = [2^w]$ og running time er: $\mathcal{O}(logw) = \mathcal{O}(loglog|U|)$ If we have a set of distinct keys $0 - u - 1$ then we can store them in van Emde Boas tree, S.

Operations are:

- empty(S), min(S), max(S): $\mathcal{O}(1)$
- member(x,S), predecessor(x,S), successor(x,S), insert(x,S), delete(x,S): $\mathcal{O}(loglog|U|)$

Master theorem siger at running time for 2x recursion bliver $\mathcal{O}(logw)$

8.3 van Emde Boas Tree data structure

8.3.1 Overview

The van Emde Boas tree can store integers from the universe $U = [2^w]$. Such that we can make efficient operations on the subset of integers.
The data structure supports the following operations:

- Empty, min, max: $\mathcal{O}(1)$
- member: $\mathcal{O}(\log \log U)$
- predecessor, successor: $\mathcal{O}(\log \log U)$
- insert, delete: $\mathcal{O}(\log \log U)$

8.3.2 Structure

The data structure is a recursive data structure, which has at each recursive level 4 fields.

Min/Max:
A place to store the minimum and maximum values. These are the minimum and maximum elements of the set. These values are only stored in these fields and not in any sub structures.
This enables us to query a structure for the min or max elements in: $\mathcal{O}(1)$.
If the minimum value is higher than the maximum value, then we know that the structure is empty. This trick enables us to check if the set is empty in: $\mathcal{O}(1)$.

Summary:
The structure has a part of the tree called the summary. This part is used to store the higher part of the binary representation of the element. i.e. the upper part of the bit-string.
The summary is a pointer to another sub-structure which is another van Emde Boas tree, **with the size of half the universe**, i.e. if the size of the universe is 2^w, then the summary has a size of $2^{\lceil \frac{w}{2} \rceil}$.
Thus we store the upper part of the bit-string-representation of the elements, in the summary.

Clusters:
Each structure also has a part we call the clusters. A cluster is a van Emde Boas Tree sub-structure which stores the lower part of the bit-string. There are $\log U = w$ number of clusters stored in an array. Each storing half, the size of the universe of the parent structure.
To know what cluster to store an element, we can index the array with the upper part of the bit-string, as the index. Thus all elements stored in a specific cluster all share the same bit-prefix.

8.3.3 Recursion depth proof

We must prove that the recursion depth of vEB when used on a universe $U = [2^w]$ is: $\lceil \log w \rceil = \mathcal{O}(\log w) = \mathcal{O}(\log \log |U|)$.

Proof:

We define the depth to be $d(w)$ when the size of the universe is some power of 2, i.e. $|U| = 2^w$.
We will then prove by induction, that $d(w) = \lceil \log_2 w \rceil$

The base case is $w = 1$.
We know there is no recursion when $w = 1$, so we get that $d(1) = 0 = \lceil \log_2(1) \rceil = \lceil 0 \rceil$.

For the induction case, we suppose that the formula, $d(w') = \lceil \log_2 w' \rceil$ holds for all $w' > 1$.
If we pick w' to be:

$$w' = \lceil \frac{w}{2} \rceil$$

Now because we know that with each recursion the size of the universe is halved, we get that the recursion depth of $d(w)$ is 1 more than the recursion depth of $d(w')$:

$$d(w) = 1 + d(w')$$

Then we can replace the depth function, $d(w)$ with $\lceil \log_2 w \rceil$:

$$d(w) = 1 + d(w') = \lceil \log_2 w \rceil = 1 + \lceil \log_2 w' \rceil$$

And then we can move the "1+" into the $\log_2$ by multiplying the inner with the log base, i.e. 2:

$$d(w) = 1 + d(w') = \lceil \log_2 w \rceil = 1 + \lceil \log_2 w' \rceil = \lceil \log_2 2 \cdot w' \rceil$$

Show $\lceil \log_2 2{\cdot}w' \rceil = \lceil \log_2 w \rceil$: Now we need to prove that $\lceil \log_2 2{\cdot}w' \rceil = \lceil \log_2 w \rceil$:

When w is even, then we know that:

$$2w' = w$$

However when w is odd, then we know that because we defined w' to be the ceiling of w over 2, then we get that it is 1 more than 2:

$$2w' = w + 1$$

or we can say:

$$2w' - 1 = w$$

Now because we have defined w' to be larger than 1 and it is odd, then we know that for all $w' \geq 3$ the ceiling of the $\log 2w'$ is the same as the ceiling of $\log 2w' - 1$:

$$\lceil \log_2 2 \cdot w' \rceil = \lceil \log_2 2 \cdot w' - 1 \rceil = \lceil \log_2 w \rceil$$

8.3.4 Operations

We already know how to do the operations Empty, Min and Max with a time complexity of $\mathcal{O}(1)$.

Predecessor:
To get the predecessor to an element, we can check if the element is larger than the maximal element. If so, then we return that element.
If not, then we check if the cluster stored according to the bit-prefix is empty. If it is not AND the minimum element of the cluster is larger than our element, then we recursively search the cluster.
If it is empty or has a larger minimum element, then we check the summary for the predecessor, this returns the index for the cluster where the predecessor is stored as the max, because it does not share the same prefix, then the predecessor must be the max value.

Because we only do recursions in one sub-structure, and each substructure halves the number of bits, then we get a running time which is the depth of the tree, which is $\mathcal{O}(d(w)) = \mathcal{O}(\log\log |U|)$.

Successor:
Like predecessor, but inverted in terms of min and max. Same running time.

Insert:
To insert an element we check if the structure is empty, if so, then we insert the element into the min and max.
If there is only 1 element in the structure, i.e. min=max, then we replace the appropriate min or max.

If the inserting element is smaller than the min or larger than the maximum element, then we switch the two elements, such that we store the originally inserted element in the min or max, and then try to recursivly insert the previously min/max element.

If it is in-between the min and max, then we check if the cluster at the bit-prefix index is empty. If it is empty, then we insert it into that cluster, and **because it is empty**, then it is a constant time operation $\mathcal{O}(1)$. Then we also insert the bit-prefix into the summary.
If it is not empty, then we insert just insert it into the cluster.

Thus we get that we only make 1 recursive call at each sub-structure. And because each substructure contains half the bits of the universe, then we get a running time of: $\mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$

8.3.5 Space complexity

The original structre does not contain null-pointers, thus it used $\mathcal{O}(|U|)$ space. However if we use null-pointers and hashtables, then we do not need to store empty substructures. Then we only store the elements in the set, and because each element inserted makes $\mathcal{O}(\log\log|U|)$ updates, then we get that our space is the number of element stored, with the places needed to be updated: $\mathcal{O}(n \cdot \log\log|U|)$.

However if we use hash tables with chaining, then we get an expected time of $\mathcal{O}(\log\log|U|)$ instead of a worst case time for all operations, as there could be collisions.

9 Polygon Triangulation

9.1 Disposition

- What is triangulation?
- Proof any triangle can be triangulated
- Proof: art gallery problem
- Walkthrough of y-monotone algorithm
- walkthrough of y-monotone triangulation

9.2 Definitions

9.2.1 Diagonal

A diagonal is a line connecting two vertices of a polygon. This line must be inside the polygon, and must be fully inside the polygon and connect to a vertex on either end.

9.2.2 Chord

A chord is a line which is inside a polygon and connects a vertex to a wall of the polygon.

9.2.3 Triangulation

Triangulation is a partitioning of a polygon into triangles such that no partitioning line overlaps another, and such that the polygon cannot be partitioned any more.

9.3 Proof: Any polygon can be triangulated

9.3.1 Lemma

Any polygon with n vertices can be triangulated with $n-2$ triangles usinng $n-3$ diagnoals.

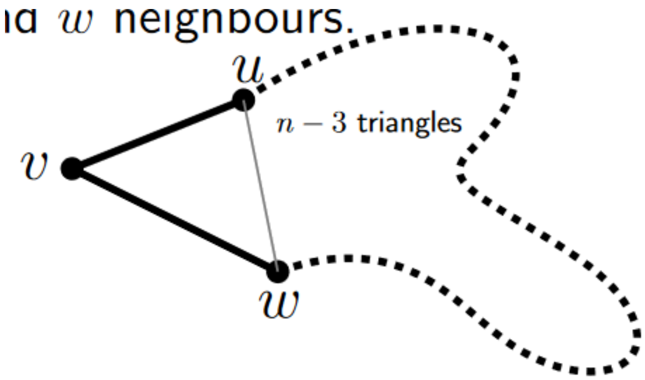
9.3.2 Proof

To prove it, we make use of induction.

Base case:
The base case is when $n = 3$, then we know we have a single triangle and we did not use any diagonals to partition the polygon.

Inductive step:
In the inductive step we let v be the leftmost vertex of the polygon and u and w be the neighbours to v.
Then from this we know that there are two cases. Either u, w is a diagonal in the triangulation, or it is not a diagonal.

u, w **is a diagonal:**
If u, w is a diagonal, then we can make a triangle, which removes v from the polygon and we have $n-1$ vertices left to be triangulated into $n-1-2$ triangles using $n-1-3$ diagonals.

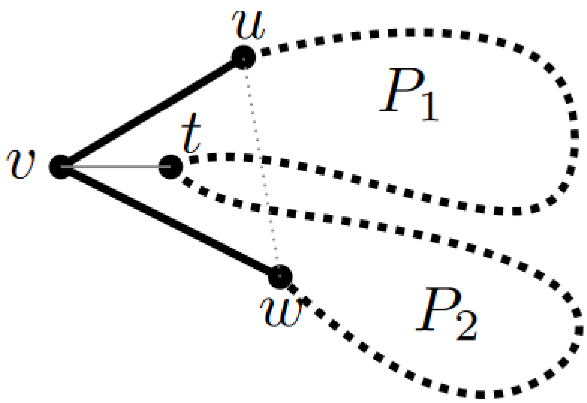


u, w **is NOT a diagonal:**
If u, w is not a diagonal, then we pick t which is the corner farthest from u, w and make a diagonal v, t. This splits the polygon into two polygons P_1 and P_2. Because we split v and t into two new vertices, then we have not removed any vertices, only added 2 new, but we know that the number of vertices V of the polygons must be:

$$|V_1| < n$$

and

$$|V_2| < n$$



We also know that the total number of vertices with the 2 split vertices is:

$$|V_1| + |V_2| = n + 2$$

Thus we know that there must be $|V_1| - 2$ triangles in P_1 with $|V_1| - 3$ diagonals. There must also be $|V_2| - 2$ triangles in P_2 with $|V_2| - 3$ diagonals.

Thus we get the total number of triangles in both polygons to be, when we split 2 vertices into 2 (adding 2 more to n):

$$|V_1| - 2 + |V_2| - 2 = |V_1| + |V_2| - 4 = n + 2 - 4 = n - 2$$

EXSTRA: And when we have added 1 diagonal + added 2 more vertices, then the number of diagonals are:

$$|V_1| - 3 + |V_2| - 3 + 1 = |V_1| + |V_2| - 6 + 1 = n + 2 - 6 + 1 = n - 3$$

9.4 Art gallery problem

The art gallery problem is defined as: How many guards are needed to cover a gallery of n vertices?

More formally: Given a set of points in cyclic order of each corner of a polygon, What is the minimum number of guards needed to cover every part of the polygon?

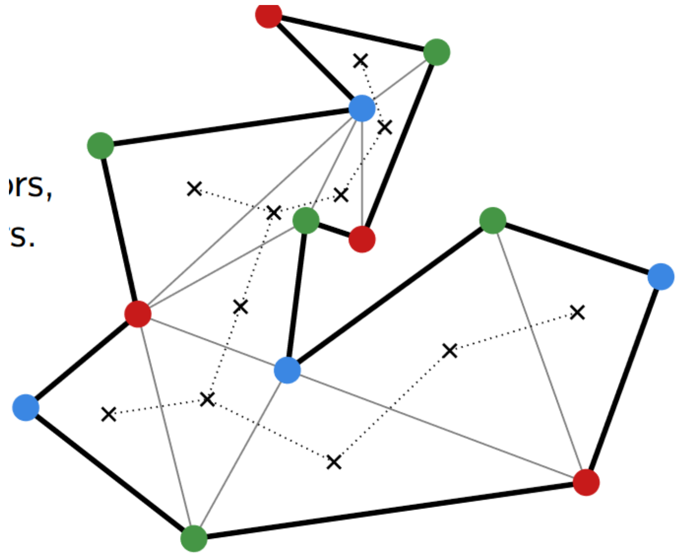
9.4.1 Art gallery theorem

The art gallery theorem says that you can always cover a polygon of n vertices with the following number of guards:

$$\#\text{guards to cover any polygon} = \lfloor \frac{n}{3} \rfloor$$

9.4.2 Proof that $\lfloor \frac{n}{3} \rfloor$ is sufficient

Start by triangulating a polygon without holes.
Then we can create a tree, where each vertex is a triangle, and each connecting triangle corresponds to a branch.
Then we can pick any vertex of the tree as the root, and then walk the tree coloring each triangles corners using 3 colors.



We can see that if we pick any color to be the guards, then we cover the polygon.

We also notice that because we use 3 colors, then we get:

$$n = n_r + n_g + n_b$$

where n_r is one color, n_g is another and n_b is the last color.

Because each triangle has all 3 colors, and a triangle shares at least 1 corner with the vertices, then we get that a single color colors at least $\frac{n}{3}$ of the vertices:

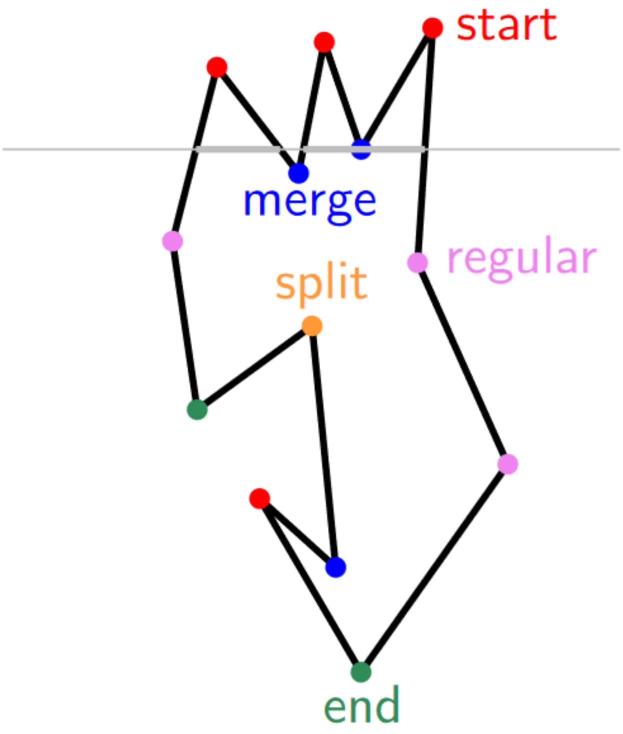
$$\min \{n_r + n_g + n_b\} \leq \frac{n}{3}$$

However we also know that the number of colored vertices must be an integer, thus we can use integer division (flooring the result):

$$\min\{n_r + n_g + n_b\} \leq \lfloor \frac{n}{3} \rfloor$$

Thus we have proven that it is always sufficient to use $\lfloor \frac{n}{3} \rfloor$ guards.

9.5 Types of vertices in a polygon



In the figure there are 4 types of vertices.
Start: a vertex with walls in a $\wedge$ shape.

End: a vertex with walls in a $\vee$ shape.

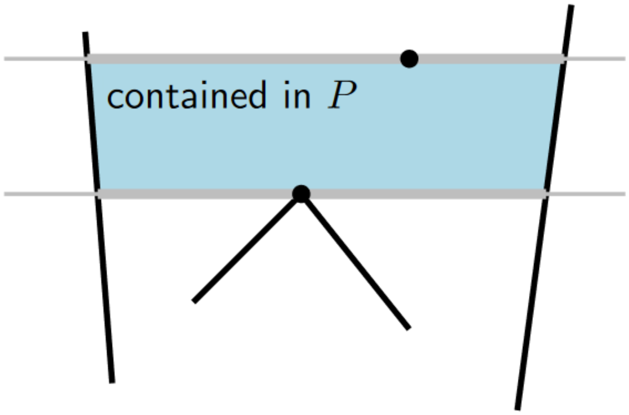
Merge: a vertex for which 2 line-segments (gray line in the figure) merge

Split: a vertex for which a line segment splits into two.

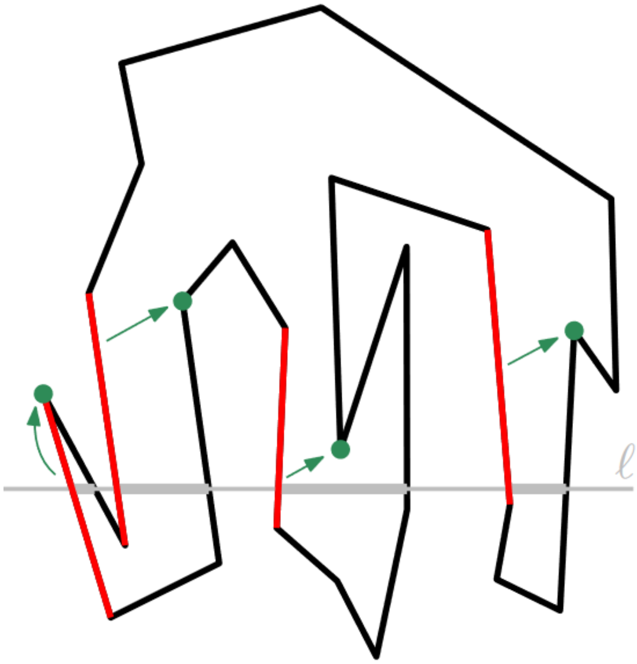
9.6 Y-monotone algorithm

To make a polygon into multiple y-monotone polygons, we want to partition the polygon into smaller polygons such that each sub-polygon does not contain any merge or split vertices.

To do so we can think of a line going from the top of the polygon to the bottom. Then each split vertex we come across we create a valid diagonal to the lowest point within the area of the polygon and the line.



To keep track of the valid areas to place such a diagonal, we can make use of a helper edge. A helper edge is an edge which intersects the line and has the interior of the polygon to the right. Then any valid diagonals must be made to the lowest vertex which has the helper edge to the right.



We can store the edges in a balanced binary tree, such that when we reach a split vertex, we can find the left edge in $O(\log n)$. Furthermore we can insert/delete helper edges in $\mathcal{O}(\log n)$.

Now we can define the operations to add or remove edges from the balanced binary search tree.
start vertex: we insert the left edge.

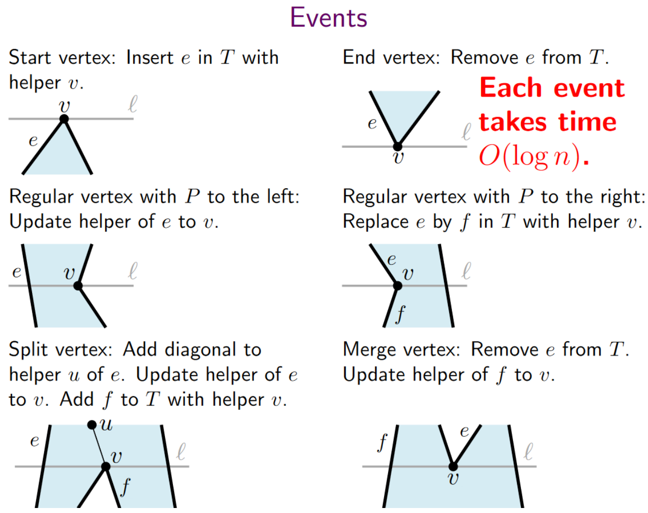
End vertex: remove left edge.

regular vertex with P to the LEFT: update the lowest vertex to the edge to be the new vertex.

regular vertex with P to the RIGHT: remove the edge above the vertex and insert the edge below.

Split vertex: add a diagonal to the vertex associated with the left helper edge. Add the right edge.

Merge vertex: remove the right edge. Update the vertex associated with the helper edge to the left.



Events

Start vertex: Insert e in T with helper v.

End vertex: Remove e from T.

Each event takes time $O(\log n)$**.**

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.

When we do this from top to bottom and bottom to top, then we have partitioned the polygon into y-monotone polygons.

When we do one sweep, we get at most 2 extra vertices per diagonal, and we know that there can be at most $n-3$ diagonals. This means that the maximum number of vertices after one sweep is:

$$n + 2(n-3) = 3n - 6$$

Thus the can calculate the time complexity:
We must sort the points, before starting the algortihm: $\mathcal{O}(n \log n)$
Then we must handle n events (helper edge manipulation) with a time complexity of $\log n$ per event: $/(n \log n)$
Then we must to it again, but from bottom to top, where we have $\leq 3n - 6$ vertices: $/(3n \log(3n))$
Thus we get a total complexity of:

$$n \log n + n \log n + 3n \log(3n)$$

And in $\mathcal{O}$:

$$\mathcal{O}(n \log n)$$

9.7 triangulation of y-monotone polygon

The general idea is to merge vertices of the left and right side of the top-most vertex.

Then go through vertices from the top down and create diagonals from the current to all possible vertices above the current.

We can make use of a stack to keep track of vertices when going top down through the vertices. We start by pushing the top-most vertex to the stack.

When going though the vertices there are two cases.
New vertex on the same side: then pop vertex, try and make diagonal, if not possible then add last popped vertex to the stack and push the new vertex.

New vertex on the opposite side: then pop every vertex and push the first vertex to the stack. Then push the new vertex to the stack.

Because we push at least 2 vertices every time we make a diagonal, then we get that we can make at most $2n$ pushes, pops and diagonal checks in total. Thus we get a linear time complexity for this algorithm: $\mathcal{O}(n)$.

9.8 Total time complexity

If triangulating a y-monotone polygon is the last step in the pipeline of triangulating a polygon, then we get that the total time complexity of this is the time it takes to make it y-monotone + triangulation for all the y-monotone polygons. And be cause the last step is linear, then we get a total time complexity of $\mathcal{O}(n \log n)$