AADS Exam notes

Caroline Amalie Kierkegaard (qlj556), Mikkel Willén (bmq419),
Isabella Odorico (jkd427) and Fie Hammer (gsr530)

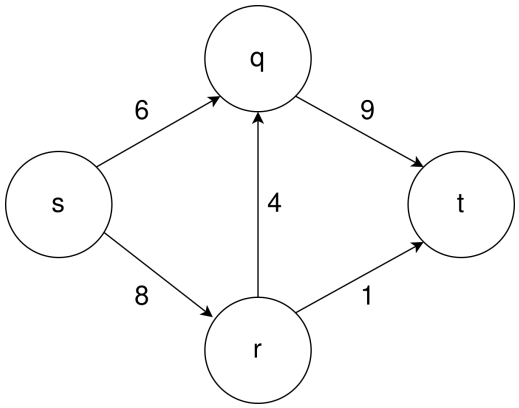
January 22, 2024

Contents

1 Max Flow

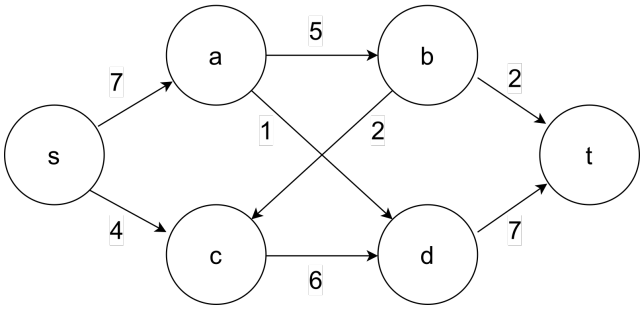
1.1 Disposition

- Introduction
  - Short example of network



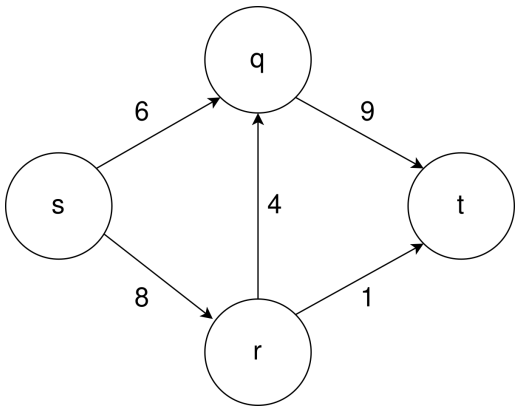
  - Flows and flow networks
  - Maxflow
  - Residual network
  - Augmenting paths
  - Cuts and minimal cuts
- `MAX-FLOW MIN-CUT` proof
- Edmonds-Karp algorithm
  - Short explanation
  - Example

G_0



1.2 Walk-through

1. Flow networks and flow problems are problems where want to transport something from a source node s to a sink node t. Draw the following example:



   A flow network $G = (V, E)$, is a directed graph in which each edge $(u, v) \in E$ has a non-negative `capacity` $c(u, v) \geq 0$. The capacity functions is defined as $c : V \times V \to \mathbb{R}$.

2. A flow is defined as $f : V \times V \to \mathbb{R}$ and satisfy the following two properties:
   Capacity constraint:

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

   Flow conservation:

$$\forall u \in V - \{s, t\}, \;\; \sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$$

   The non-negative quantity $f(u, v)$ is called the flow from vertex u to vertex v. The value $|f|$ of a flow f is defined as:

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

3. A residual network G_f consists of the edges with capacities that represent how we can change the flow on the edges of G. The residual capacity is

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

4. Given a flow network $G = (V, E)$ and a flow f, the residual network of G induced by f is

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

   which means that each of the residual networks edges can admit a flow that is greater than 0.

5. The flow in the residual network, provides options for for adding flow to the original flow network. For a flow f and a residual flow f' in the corresponding residual network, the augmentation of flow f by f' is defined as $f \uparrow f'$ and is a function from $V \times V$ to $\mathbb{R}$ defined as

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

6. A cut (S,T) of a flow network, is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. We then have the net flow $f(S,T)$ across the cut as:

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

and the capacity of the cut:

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

A minimum cut of a network is a cut, where the capacity is the minimum over all cuts of the network.

7. The theorem states:
If f is a flow in a flow network $G = (V,E)$ with source s and sink t, then the following conditions are equivalent:

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

8. **Proof**

   $(1) \Rightarrow (2)$:
   Proof by contradiction. Assume that f is a maximum flow in G but that G_f has an augmenting path p. Then the flow found by augmenting f by f_p is a flow in G with value strictly greater than f, which is a contradiction to the assumption that f is a maximum flow.

9. $(2) \Rightarrow (3)$:
   Lets assume that G_f has no augmenting path, which means G_f contains no path from s to t. We define

$$S = \{v \in V \ : \text{ there exists a path from } s \text{ to } v \text{ in } G_f\}$$

and

$$T = V - S$$

The partition (S,T) must be a cut, since we have $s \in S$ and $t \notin S$, because there is no path from s to t in G_f. Lets now consider a pair of vertices $u \in S$ and $v \in T$. If $(u,v) \in E$, we must have

$$f(u,v) = c(u,v) \tag{1}$$

since if not, we would also have $(u,v) \in E_f$. This would mean that $v \in S$. If $(v,u) \in E$, we must have

$$f(v,u) = 0 \tag{2}$$

since if not, $c_f(u,v) = f(v,u)$ would be positive, which would mean that $(u,v) \in E_f$, which again would mean that $v \in S$. If neither $(u,v) \in E$ or $(v,u) \in E$, then

$$f(u,v) = f(v,u) = 0 \tag{3}$$

and we must then have

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) && \text{flow across the cut} \\ &= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{u \in S} \sum_{v \in T} 0 && \text{1st by 1 and 2nd by 2 and 3} \\ &= c(S,T) && \text{By def, and 0 removed} \end{aligned}$$

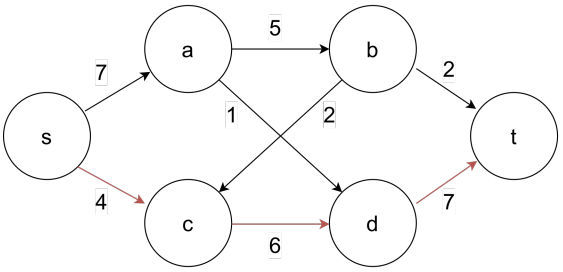
and we have $|f| = f(S,T) = c(S,T)$.

10. (3) ⇒ (1) :
    We have that $|f| \leq c(S,T)$ for all cuts (S,T). $|f| = c(S,T)$ then means that f is a maximum flow.

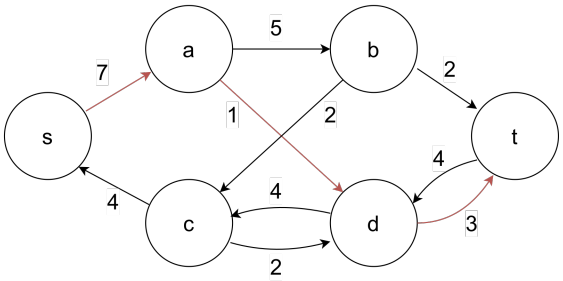
11. The `EDMONDS-KARP` algorithm uses the `FORD-FULKERSON` method, where the chosen augmenting path in the residual graph is always the shortest path with respect to the number of edges in the path.

12. Example:

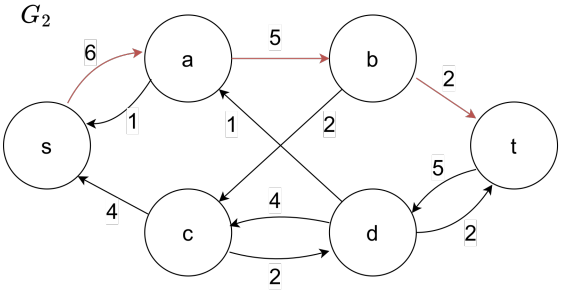
G_0



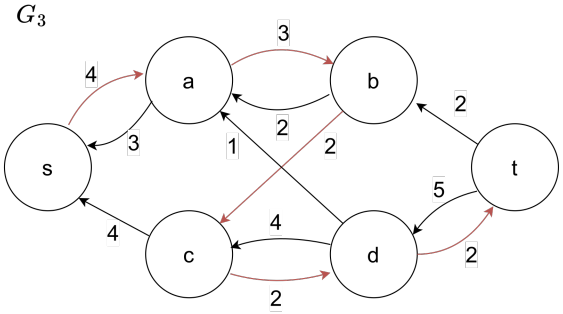
G_1



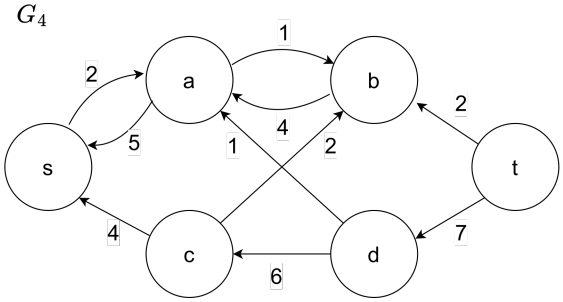
G_2



G_3



G_4



13. Done.

1.3 Extra

1.3.1 `EDMONDS-KARP` running time $O(VE^2)$

1. We now want to analyse the running time of `EDMONDS-KARP` and show, that the running time is $O(VE^2)$
2. We have a graph $G = (V, E)$, with source and sink s and t, and a residual graph G_f. We are going to use forward edges. We are going to divided the edges of the graph into levels, where each level i consists of nodes that have a shortest path from s to it of length i. A forward edges, is an edges where we go from one level to another, defined like this:
$$(u, v) \in E_f \ : \ \delta_f(s, v) = \delta_f(s, u) + 1$$
where $\delta_f(u, v)$ is the shortest path distance in the residual graph from u to v where each edge has weight 1.
3. Lemma 1.: Let G_{f_0} be a residual network obtained at some point in the algorithm and let $f_1, \cdots, f_k$ be the sequence of flows obtained during the next $k \geq 0$ iterations. We then let $d = \delta_{f_0}(s, t)$ and assume that $\delta_{f_i}(s, t) = d$ for $i \in [k-1]$. Then in each of the k iterations, flow is only pushed along forward edges in G_{f_0}. We also have that, $\delta_{f_k}(s, t) \geq d$.
4. We can now proof the desired time bound, which is that `EDMONDS-KARP` runs in $O(VE^2)$ time.
5. Proof: Each iteration of the algorithm takes $O(V + E) = O(E)$ time using breadth-first search in the residual network. We want to show that the number of iterations is $O(VE)$.
6. By Lemma 1., we have that $k \leq |E|$ since G_{f_0} has at most $|E|$ forward edges and each of the k iterations removes at least one such edge. Once a forward edges disappears, it cannot reappear, since otherwise, flow would have been pushed along the reverse of a forward edge, contradicting Lemma 1. Lemma 1 also gives us $\delta_{f_k}(s, t) \geq d$. By maximality of the sequence $f_0, \cdots, f_k$, we cannot have $\delta_{f_k}(s, t) = d$ so $\delta_{f_k}(s, t) \geq d + 1$
7. So we have shown, that every $|E|$ iterations, the distance must have increased by at least 1, and that the distance can never decrease. The distance has an upper bound of $V - 1$, which is the case, where we visit all vertices, and thus we can conclude that the number of iterations is $O(VE)$, for a total runtime of $O(VE^2)$.

1.3.2 Lemma 1. proof

1. Lemma 1.: Let G_{f_0} be a residual network obtained at some point in the algorithm and let $f_1, \cdots, f_k$ be the sequence of flows obtained during the next $k \geq 0$ iterations. We then let $d = \delta_{f_0}(s, t)$ and assume that $\delta_{f_i}(s, t) = d$ for $i \in [k-1]$. Then in each of the k iterations, flow is only pushed along forward edges in G_{f_0}. We also have that, $\delta_{f_k}(s, t) \geq d$.
2. Proof: The proof is by induction on $k \geq 0$. For the base case, we chose $k = 0$, since we then have no iterations and $G_{f_k} = G_{f_0}$.
3. For the induction step, we assume that $k > 0$ and that the claim holds for $k-1$. By the induction hypothesis, this means flow has only been pushed along forward edges so far. We now want to show that the flow is pushed along forward edges in the k'th iteration.
4. The algorithm augments flow along a shortest path p from s to t in residual graph for f_{k-1} to obtain the new residual graph. We know all edges of p are forward edges of G_{f_0} so we must have
$$(u, v) \in p \Rightarrow \delta_{f_0}(s, v) \leq \delta_{f_0}(s, u) + 1$$
since flow has only been pushed along forward edges, so (u, v) is either an edge of G_{f_0} or the reversal of a forward edge of G_{f_0}.

5. We also know that the length of p is δ_{f_0}, which means that it must be the case that for every edge $(u, v) \in p$, we have that $\delta_{f_0}(s, v) = \delta_{f_0}(s, u) + 1$ since there is exactly d edges in p. And since this is the definition of a forward edge, then we have shown that all edges in p are forward edges.

6. The same argument can be made for G_{f_k}, that for any edge $(u, v) \in p$ we have $\delta_{f_0}(s, v) = \delta_{f_0}(s, u) + 1$, which means p must have at least d edges and therefore we have $\delta_{f_k}(s, t) \geq d$ concluding proof of lemma 1.

2 Linear Programming

2.1 Disposition

- Introduction
  - General LP problem
  - Objective function
  - ConStraints
  - Standard form
  - Slack form
- Example Draw example:

$$\begin{array}{lrrrrrr} \min & - & 2x_1 & + & 3x_2 & & \\ \text{s. t} & & x_1 & + & x_2 & = & 7 \\ & & x_1 & - & 2x_2 & \leq & 4 \\ & & & & x_1 & \geq & 0 \end{array}$$

- `SIMPLEX` algorithm
  - Explain duality
  - Proof of weak duality

2.2 Walk-through

1. Linear programming is a mathematical optimization technique used to find the best outcome (maximum or minimum) in a mathematical model with linear relationships. The objective is to optimize a linear objective function, subject to a set of linear equality and inequality constraints.

2. Linear programming is an important subject because many real-life problems can be modelled after a linear program. *Examples to use:* when dealing with transportation and minimizing transportation costs or maximizing the impact of a marketing campaign within a budget, minimising costs or maximising utilization etc.

3. Normally a linear program consists of an objective function which describes a relation between the variables of the problem which we want to minimize or maximize, as well as some form of constraints on the variables.

4. The constraints are linear inequalities or equations that restrict the possible values of the decision variables. They play an important role since they define the limitations or boundaries within which the decision variables must operate. These constraints ensure that the solution to the optimization problem is feasible and realistic within the given context. Constraints can represent limitations on resources, budgetary constraints, production capacities, and other relevant factors. The goal is to find values for the decision variables that satisfy all constraints while optimizing the objective function.

5. If the values satisfy all the constraints, the solution is feasible. If a feasible solution has a min or max objective value it is the optimal solution.

6. We normally write the program in either standard form or slack form, which are two different ways to represent a linear program.

7. In standard form, we are given sets of real numbers $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n$ and $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m$, as well as a matrix of real numbers $\mathbf{a}_{ij}$ for i in $[m]$ and j in $[n]$. We aim to assign values to n variables $x_1, x_2, \ldots, x_n$ in order to maximize the objective function:

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

Subject to the following constraints:

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

$$x_j \geq 0 \quad \text{for } j \in [n]$$

The goal is to maximize the objective function while satisfying the constraints that each linear combination of $a_{ij}x_j$ is less than or equal to b_i, and each x_j is non-negative. Importantly, all inequalities must be of the form $\leq$, and no equalities ($=$) or $\geq$ are allowed.

Let's look at the example and convert the given problem into standard form:
(If not already done draw the example:)

$$\begin{array}{llrcrcl} \min & - & 2x_1 & + & 3x_2 & & \\ \text{s. t} & & x_1 & + & x_2 & = & 7 \\ & & x_1 & - & 2x_2 & \leq & 4 \\ & & & & x_1 & \geq & 0 \end{array}$$

To convert it to standard form, I can first note that min LP is converted to an equivalent max problem by negating the coefficient of the objective function. thus: max $2x_1 - 3x_2$ instead, so:

$$\begin{array}{lrcrcl} \max & 2x_1 & - & 3x_2 & & \\ \text{s. t} & x_1 & + & x_2 & = & 7 \\ & x_1 & - & 2x_2 & \leq & 4 \\ & & & x_1 & \geq & 0 \end{array}$$

Next, all the variables considered as x_j without non-negative constraints are replaced by two non-negative variables noted as $x_j^{'}$ and $x_j^{''}$, and each occurrence of x_j is also replaced by $x_j^{'} - x_j^{''}$ so that it is written as:

$$\begin{array}{lrcrcrl} \max & 2x_1 & - & 3x_2^{'} & + & 3x_2^{''} & \\ \text{s. t} & x_1 & + & x_2{}^{'} & - & x_2^{''} & = 7 \\ & x_1 & - & 2x_2^{'} & + & 2x_2^{''} & \leq 4 \\ & x_1 & , & x_2^{'} & , & x_2^{''} & \geq 0 \end{array}$$

Now, all '=' constraints are replaced by 'opposites' meaning the two inequalities *

$$\begin{array}{lrcrcrl} \max & 2x_1 & - & 3x_2^{'} & + & 3x_2^{''} & \\ \text{s. t} & x_1 & + & x_2{}^{'} & - & x_2^{''} & \leq 7* \\ & x_1 & + & x_2{}^{'} & - & x_2^{''} & \geq 7* \\ & x_1 & - & 2x_2^{'} & + & 2x_2^{''} & \leq 4 \\ & x_1 & , & x_2^{'} & , & x_2^{''} & \geq 0 \end{array}$$

Next, to change the inequality which is not allowed ($\geq$), I turn it around by multiplying both sides by -1. *

$$\begin{array}{lrcrcrl} \max & 2x_1 & - & 3x_2^{'} & + & 3x_2^{''} & \\ \text{s. t} & x_1 & + & x_2{}^{'} & - & x_2^{''} & \leq 7 \\ - & x_1 & - & x_2{}^{'} & + & x_2^{''} & \leq -7* \\ & x_1 & - & 2x_2^{'} & + & 2x_2^{''} & \leq 4 \\ & x_1 & , & x_2^{'} & , & x_2^{''} & \geq 0 \end{array}$$

Now, I can rename the variables for more clarity:

$$\begin{array}{lrcrcrl} \max & 2x_1 & - & 3x_2 & + & 3x_3 & \\ \text{s. t} & x_1 & + & x_2 & - & x_3 & \leq 7 \\ - & x_1 & - & x_2 & + & x_3 & \leq -7 \\ & x_1 & - & 2x_2 & + & 2x_3 & \leq 4 \\ & x_1 & , & x_2 & , & x_3 & \geq 0 \end{array}$$

8. Similarly, in slack form, which we use for the SIMPLEX algorithm, all inequalities that are not non-negativity constraints must be strict equalities. We do this by introducing slack variables and converting them to slack form. Slack variables are what makes the inequalities become equalitites. The slack variable represents the surplus or slack in the constraint. If the original constraint is not fully utilized, the slack variable takes up the remaining value to make the equation equal. For every feasible solution s_1, s_2, s_3 the left hand value is AT MOST the right hand value. This means that there often is a slack between the two.

For instance looking at this: $2x_1 + 3x_2 + x_3 \leq 5$ there is a slack which we can denote by using x_4, If we require that $x_4 \geq 0$ then I can replace my inequality with an equality such that:

$$2x_1 + 3x_2 + x_3 + x_4 = 5$$

This I can also write as:

$$\sum_{j=1}^{n} a_{ij}x_j + x_{n+i} = b_i \quad \text{for } i = 1, 2, ..., m$$

Let's convert this example to slack form. We insert a new variable for each constraint and convert the inequality to equalitites:

$$\begin{array}{lrcrcrcrcrcrcr}
\max & & 2x_1 & - & 3x_2 & + & 3x_3 & & & & & & \\
\text{s. t} & & x_1 & + & x_2 & - & x_3 & + & x_4 & & & = & 7 \\
- & & x_1 & - & x_2 & + & x_3 & & & + & x_5 & = & -7 \\
& & x_1 & - & 2x_2 & + & 2x_3 & & & & & + \; x_6 \; = & 4
\end{array}$$

Now, we can write out the variables as so, by isolating the 3 added slack variables, we move everything to the right side and change the +/- (only for the last three):

$$\begin{array}{rrcrcrcr}
Z = & 0 & + & 2x_1 & - & 3x_2 & + & 3x_3 \\
x_4 = & 7 & - & x_1 & - & x_2 & + & x_3 \\
x_5 = & -7 & + & x_1 & + & x_2 & - & x_3 \\
x_6 = & 4 & - & x_1 & + & 2x_2 & - & 2x_3
\end{array}$$

Looking at this, the slack variables at the left side is the basic and the one on the right is the non-basic. The basic variables are the variables that have a non-zero coefficient in the equations defining the constraints. The values of basic variables are determined by the system of equations, and they play a crucial role in finding a feasible solution. Non-basic are the variables that do not have a non-zero coefficient in the equations defining the constraints. Nonbasic variables are typically set to zero in the process of solving the linear programming problem in the SIMPLEX algorithm.

$$\begin{array}{rrcrcrcr}
Z = & 0 & + & 2x_1 & - & 3x_2 & + & 3x_3 \\
x_4 = & 7 & - & x_1 & - & x_2 & + & x_3 \\
x_5 = & -7 & + & x_1 & + & x_2 & - & x_3 \\
x_6 = & 4 & - & x_1 & + & 2x_2 & - & 2x_3
\end{array}$$

9. SIMPLEX ALGORITHM:
Now, the simplex algorithm continuously changes the basic solution by pivoting variables to and from the basic variables. Pick a variable in the objective function which positively increases the value, and pivot it with the basic variable that bottlenecks how much the non-basic variable can be increased. It starts with an initial feasible solution and iteratively moves towards the optimal solution by pivoting between basic and nonbasic variables. Slack variables help convert inequalities into equations, allowing the simplex method to navigate through feasible solutions efficiently.

The simplex algorithm is used for solving linear programming problems in various fields such as operations research, economics, finance, and engineering. It is particularly effective for large-scale linear programming problems.

10. The process of SIMPLEX:
    - Select Pivot Element: Choose the first non-basic variable. Determine the pivot row by selecting the smallest non-negative ratio of the right-hand side to the pivot column value.
    - Pivot: Use the pivot element to create a new table by performing row operations to make the pivot column a unit vector and the other entries in the pivot row zero.
    - Update: Repeat the process until the objective row contains non negative coefficients.
    - Solution: Extract the solution from the final table, which provides the optimal values of the decision variables.
11. When does SIMPLEX terminate?
    - When all coefficients in the objective function are negative.
    - When it becomes obvious that LP is unbounded.

    Why does this compute the optimal solution? - Weak duality.
12. Weak duality:
    Understanding the weak duality theorem is crucial in demonstrating how the SIMPLEX algorithm arrives at the optimal solution. This theorem posits that the solution to the dual always serves as an upper bound for the solution of the primal3.

    Dual form:
    The dual form involves minimizing

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

    under the constraint

$$\sum_{i=1}^{m}(a_{ij} y_i) \geq c_j \text{ for } j \in [n], \text{ where } y_i \geq 0 \text{ for } i \in [m]$$

    Weak Duality Theorem Statement:
    In essence, what the weak duality theorem says is that it is always the case that

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

    where $\bar{x}$ is any feasible solution to the primal linear program and $\bar{y}$ is any feasible solution to the dual linear program. This is the case because

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j && \text{by the inequality of the dual constraints} \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i && \text{since this is simply a reordering of the summation terms} \\
&\leq \sum_{i=1}^{m} b_i \bar{y}_i && \text{by the primal inequality constraint}
\end{aligned}$$

13. Done.

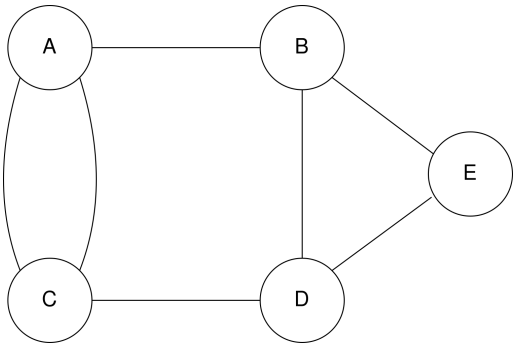
3 Randomized Algorithms

3.1 Disposition

- Introduction
  - What is it effective for?
- Las Vegas algorithms
  - Explanation and guarantees
  - `RAND QUICKSORT` example

$$[3, 5, 2, 1, 4]$$

- Monte Carlo algorithms
  - Explanation and guarantees
  - `MIN-CUT` example



  - Proof and running time

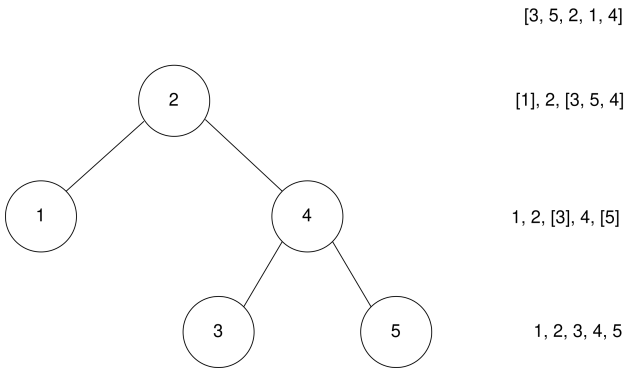
3.2 Walk-through

1. Randomized algorithms are algorithms that utilize randomness in their procedure. They are often the simplest or the fastest alternatives in practical settings, and some problem are practically only solvable using randomization. There are 2 types of randomized algorithms. One is the Las Vegas algorithm, which is a type of algorithm, that always returns the correct result, but where the runtime can vary. The other type is the Monte Carlo algorithm, which is generally fast, but the result of the algorithm is not guaranteed to be the correct result.

2. Since the Las Vegas algorithm always returns the correct result, but has a random runtime, we want to study the running time for this type of algorithm.

3. An example of a Las Vegas algorithm is `RAND QUICKSORT`, which is used for sorting a set of elements. It is similar to the normal `QUICKSORT` algorithm, the only difference is, that the pivot value is chosen at random.

4. Example: We have the following list of elements $[3, 5, 2, 1, 4]$. The first "random" element to be chosen is 2, the next "random" element to be chosen is 4, and we get the following tree T. The

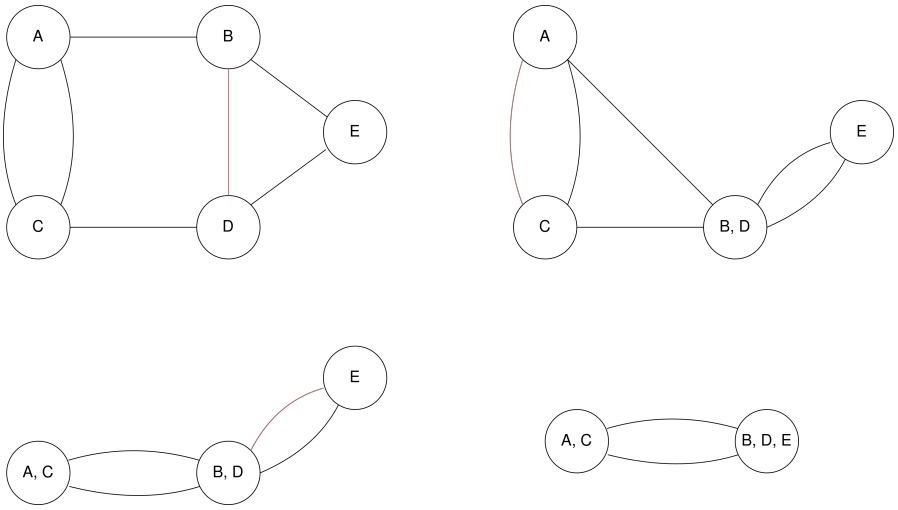


   output of `RAND QUICKSORT` is then the in-order traversal of the tree. For the runtime analysis we are going to use the level-order traversal, which is top-down the left-to-right. $\pi_l = ([2, 1, 4, 3, 5])$.

5. So since the Monte Carlo algorithms are algorithms, that may not return the correct answer, we want to analysis the correctness of the algorithm.

6. An example of a Monte Carlo algorithm is the `MIN-CUT` algorithm, which is an algorithm used to find a cut in a graph with minimum cardinality.

7. Example:



8. We have the following graph G which is an undirected multigraph with 5 vertices. We are going to try to find a min-cut, by picking an edge at random and do a contraction of the edge. We then repeat this, untill there are only two vertices left, where the set of edges between the two vertices being the cut returned.

9. Lets take a look at the probability of failure for this algorithm.

10. We let k be the min-cut size, and then we look at a particular min-cut C with k edges. We know G must have at least $\frac{kn}{2}$ edges, since if it has less, there would be a vertex of degree less than k, which would mean its incident edges would be a min-cut of size less than k.

11. We will bound, the probability that no edge of C is ever contracted, from below, meaning we will bound from below, the probability that edges surviving in the end are exactly the edges in C.

12. We now let $\mathcal{E}_i$ denote the event, that an edge in C is not picked at the i'th step. This is

$$\Pr[\mathcal{E}_1] \geq 1 - \frac{2}{n}$$

for the first step, since the probability of picking an edge in C is at most $\frac{k}{\frac{nk}{2}} = \frac{2}{n}$.

13. Lets assume $\mathcal{E}_1$ occurs, then during the second step there are at least $\frac{k(n-1)}{2}$ edges, so the probability of an edge in C being picked is at most $\frac{2}{n-1}$, and we get the probability

$$\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - \frac{2}{n-1}$$

14. For the i'th step, the number of remaining vertices is $n - i + 1$, which means there are at least $\frac{k(n-i+1)}{2}$ edges remaining, and thus we get the probability

$$\Pr\left[\mathcal{E}_i \mid \bigcap_{j=1}^{i-1} \mathcal{E}_j\right] \geq 1 - \frac{2}{n-i+1}$$

15. If we then look at the probability that no edge of C is ever picked in the entire process, we get

$$\Pr\left[\bigcap_{i=1}^{n-2} \mathcal{E}_i\right] \geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}$$

Thus the probability of discovering a particular min-cut in G is larger than $\frac{2}{n^2}$.

16. We could now try to run the algorithm more time, to reduce the probability of failure. Running the algorithm x times, would give us a probability of failure

$$\left(1 - \frac{2}{n^2}\right)^x$$

which we could then solve for x with some tolerance. If we eg run the algorithm $\frac{n^2}{2}$ times, would the probability that a min-cut is not found in any of the attempts to be

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

which is a significant reduction. Note that the running time increase with the number of repetitions.

17. Done.

3.3 Extra

3.3.1 `RAND-QUICKSORT` runtime

1. We get the expected runtime measuring the number of comparisons. We let the X_{ij} be an indicator variable denoting whether element i and j are compared in the process. The total number of comparisons is then going to be

$$\sum_{i=1}^{n}\sum_{j>i}^{n} X_{ij}$$

and the expected number of comparisons will then be

$$\mathbb{E}\left[\sum_{i=1}^{n}\sum_{j>i}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j>i}^{n}\mathbb{E}\left[X_{ij}\right]$$

by linearity of expectation, and since X_{ij} is an indicator variable, the expectation of it, is equal to the probability p_{ij} that i and j are compared in the process. So we need to figure out what this probability is.

2. To do this, we observe, that only the chosen pivot is compared to other elements. We let S_k denote the k'th smallest element in the set. Looking at the level-order traversal of the tree, we can see, that i and j are only compared if S_i or S_j occurs earlier in the permutation π_l than any element S_l where $i < l < j$. We can see this, since if such an element existed, S_i would be in the left sub-tree of this element S_l, while S_j would be in the right sub-tree of this element S_l.

3. Since any of the elements S_i, S_j and all elements between them is equally likely to be the first of these elements, the probability that the first element is either S_i or S_j is

$$\frac{2}{j-i+1}$$

which means that

$$p_{ij} = \frac{2}{j-i+1}$$

Inserting this we get

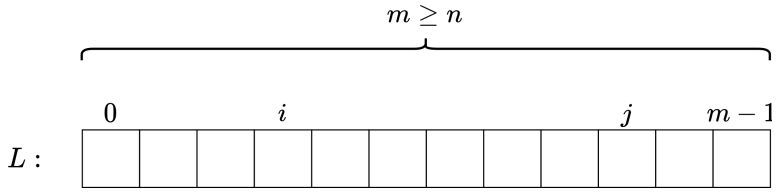
$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j>i}^{n} p_{ij} &= \sum_{i=1}^{n}\sum_{j>i}^{n}\frac{2}{j-i+1} && \text{inserting}\\
&\leq \sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{2}{k} && \text{since } j \geq n-i+1\\
&\leq 2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k} && \text{since } n \geq n-i+1\\
&= 2n\sum_{k=1}^{n}\frac{1}{k} && \text{by def.}\\
&\leq 2n\log n && \text{n'th harmonic number, it is } \leq \ln n + \Theta(1)
\end{aligned}$$

So the expected running time of `RAND QUICKSORT` is $O(n \log n)$

4 Hashing

4.1 Disposition

- Introduction
  - Motivation
  - Explanation
- Types of hashing
  - Universal hashing
  - C-Approximate universal hashing
  - Strong universal hashing
- Hashing with chaining
  - Example



  - Expected running time
- Signatures

4.2 Walk-through

1. **Motivation**: Hashing is a way in which we can map keys to values. It is used in data structures such as hash tables, which allow for efficient search, insertion, and deletion operations. Hashing is particularly useful when dealing with large sets of data, where other techniques such as linear search may become impractical. The idea behind hashing is that the universe U of keys that we wish to map is much too large to contain in memory. Hence we map it to a smaller range $m = \{0, 1, \ldots, m-1\}$ of hash values.

2. **Explanation**: A hash function h is a function from the universe U to $[m]$, where $h(x)$ is a random variable. Since the universe U is much larger than $[m]$ we are not able to map all values of the universe to a unique hash value, and thus we might have collisions between the hash values of keys from the universe U.

3. **Types of hashing:** Some hash functions have properties, which guarantee a low collision probability, and might therefore be preferable to others.

4. **Universal hashing** Universal hashing is a way to minimize collision. In order to be considered universal, the probability of two distinct values, x and y, being the same, must be lower than or equal to $\frac{1}{m}$

$$h : U \to [m] \text{ is universal if } \forall x \neq y \in U : \Pr[h(x) = h(y)] \leq \frac{1}{m}$$

5. **C-approximately universal hashing:** A more relaxed approach to the universal function is the C-approximately function, which remains equal to the universal hashing, except we expect the probability of collision to be lower than or equal to $\frac{c}{m}$

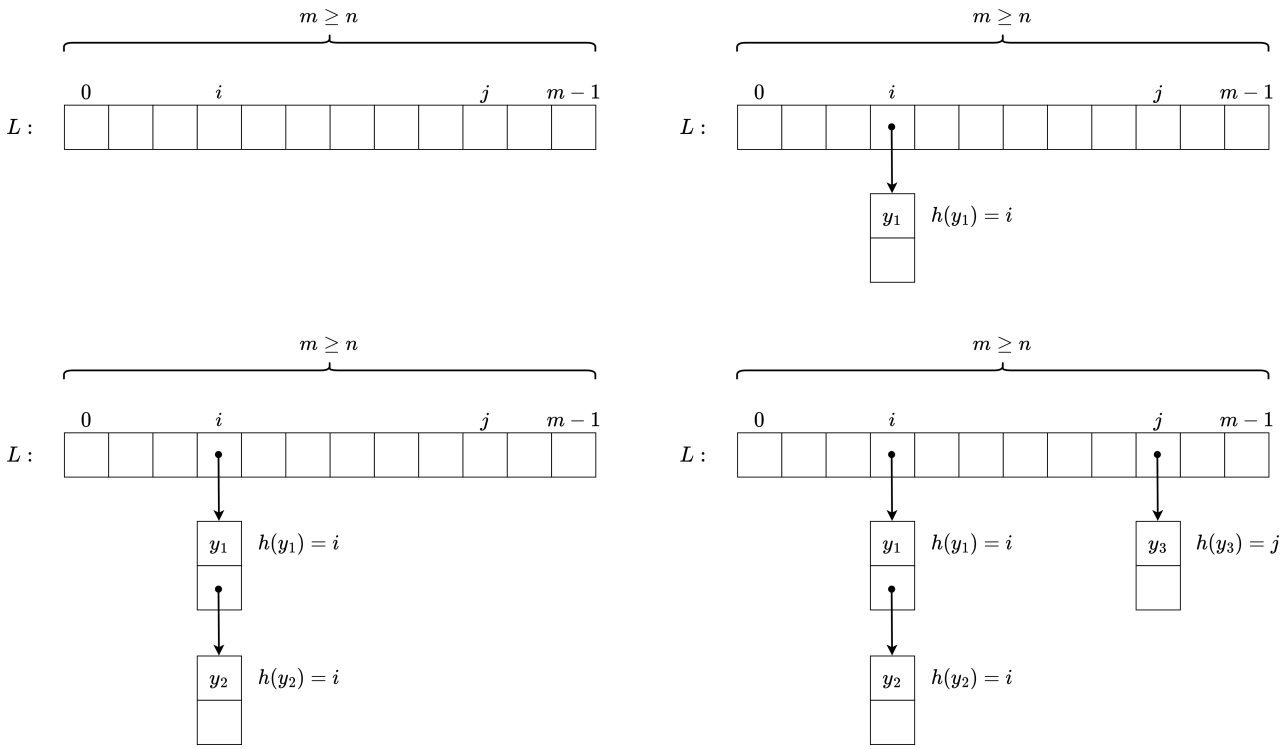
$$h : U \to [m] \text{ is c-approximately universal if } \forall x \neq y \in U : \Pr[h(x) = h(y)] \leq \frac{c}{m}$$

6. **Strong universal hashing** A strong universal hashing function is a universal function that also can ensure that the probability of two keys independently mapping to the same value is equal to $\frac{1}{m^2}$. Also called 2-independent hashing.

$$h : U \to [m] \text{ is strongly universal if } \forall x \neq y \in U \text{ and } q, r \in [m] : \Pr[h(x) = q \wedge h(y) = r] \leq \frac{1}{m^2}$$

7. **Hashing with chaining** Hash tables with chaining is one of the most classic applications of universal hashing. We have a set $S \subseteq U$ of keys that we wish to store and then be able to find in constant time.

8. **Example (chaining)** We let $n = |S|$ and $m \geq n$. Then we pick a universal hash function, and create an array L of m linked lists, so that all hash values, we have a place in L to store them.



9. **Expected running time:** We can store and look up in constant time using a hash table with chaining. The proof can be divide into two parts; one where $x \in S$ and one where $x \notin S$. We show the latter part below.

   We claim that:

   $$\text{for } x \notin S,\ \mathop{\mathbb{E}}_{h}[|L[h(x)]|] \leq 1$$

   The length $|L(h(x)|$ denotes the length of the linked list in $L[h(x)]$. The length of the linked list, must be the sum of all elements in S hashing to the same value as x, so we get

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

   We can then use the linearity of expectation to move the expectation inside the sum:

   $$= \sum_{y \in S} \mathop{\mathbb{E}}_{h}[h(y) = h(x)]$$

   Since $h(y) = h(x)$ is an indicator variable (because it denotes if they collide or not), the expectation of it is equal to the probability that they hash to the same value. So we can rewrite it as:

   $$= \sum_{y \in S} \Pr[h(x) = h(y)]$$

   Since we picked a universal hash function, we know the probability of a collision is at most $\frac{1}{m}$ for each element in the set S, and we therefore get:

   $$\leq |S| \cdot \frac{1}{m} = \frac{n}{m} \leq 1$$

   The other part of the proof, where $x \in S$ is pretty similar. The probability of collision is the same, but the expectation of the list length must be 1 greater, since we $x \in S$ so i must already have hashed to the value at $L[h(x)]$, so we get:

   $$1 + \frac{n}{m} \leq 1 + 1 = 2$$

10. Another use of hashing is signatures, where we assign a unique signature $s(x)$ to each key. So we want $s(x) \neq s(y)$ for all distinct keys $x, y \in S$. To do this, we pick a universal hash function $s : U \to [n^3]$. The probability of a collision is then

$$\begin{aligned}
\Pr_s[\exists\{x,y\} \subseteq S \,:\, s(x) = s(y)] &\leq \sum_{\{x,y\}\subseteq S} \Pr_s[s(x) = s(y)] && \text{union bound} \\
&\leq \frac{\binom{n}{2}}{n^3} && \text{number of possibilities div number of hashes} \\
&= \frac{1}{2n}
\end{aligned}$$

11. Chaining and signatures can also be combined, where we let the list $L[i]$ store the signatures $s(x)$ of the keys that hash to i, so

$$L[i] = \{s(x) \mid x \in X,\ h(x) = i\}$$

Then we can check if x is in the table, by checking if $s(x)$ is in $L[h(x)]$.

4.3 Extra

4.3.1 MULTIPLY-SHIFT 2-universal proof

This proof is NOT part of the curriculum. `MULTIPLY-SHIFT` addresses hashing from w-bit integers to l-bit integers.
`MULTIPLY-SHIFT` is 2-universal, which means, that for $x \neq y$ and a uniformly random odd w-bit integer a, we have

$$\Pr[h(x) = h(y)] \leq \frac{2}{2^l} = \frac{2}{m}$$

1. Think of the bits representing a number, as indexed with bit 0 as the least significant bit. The idea is to extract the l most significant bits of $ax \mod 2^w$.

2. We have that $h_a(x) = h_a(y)$ if and only if ax and $ay = ax + a(y-x)$ have the same bits on indices $w-l, \cdots, w-1$. This requires that the bits $w-l, \cdots, w-1$ if $a(y-x)$ are either all 0s or all 1s. We can see this, since if we get no carry from bits $0, \cdots, w-l$ when we add $a(y-x)$ to ax, then $h_a(x) = h_a(y)$ exactly when bits $w-l, \cdots, w-1$ of $a(y-x)$ are all 0s.

3. Similarly, if we get a carry, then $h_a(x) = h_a(y)$ exactly when the bits $w-l, \cdots, w-1$ of $a(y-x)$ are all 1s, since the carry is then carried all the way up, leaving 0s behind. We can then conclude, that it is enough to show that the probability of the bits being all 0s or 1s is at most $\frac{2}{2^l}$.

4. To do this, we use the fact that any odd number z is relatively prime to any power of two, meaning

   $$\text{Fact 1: if } \alpha \text{ is odd and } \beta \in [2^q]_+ \text{ then } \alpha\beta \not\equiv 0 \pmod{2^q}$$

   We then define b such that $a = 1 + 2b$. We know b is uniformly distributed in $[2^{w-1}]$, since a is chosen uniformly at random. We also define z to be the odd number satisfying $(y-x) = z2^i$. We can then rewrite

   $$a(y-x) = (1+2b)(y-x) = z2^i + 2bz2^i = z2^i + bz2^{i+1}$$

5. We know want to show that $bz \mod 2^{w-1}$ is uniformly distributed in $[w^{w-1}]$. There is a 1 to 1 between $b \in [w^{w-1}]$ and the product $bz \mod 2^{w-1}$. If there were not, there would be a b' such that

   $$b'z \equiv bz \pmod{2^{w-1}} \Leftrightarrow z(b-b') \equiv 0 \pmod{2^{w-1}}$$

   which is a contradiction to fact 1. This means the uniform distribution on b implies that $bz \pmod{w^{w-1}}$ is uniformly distributed.

6. We can now conclude, that $a(y-x)$ has 0 in bits $0, \cdots, i-1$, because of shifting, it has 1 at index i, because z is odd and has a 1 at bit 0, which is then shifted to the i'th index, and finally that it has a uniform distribution on bits $i+1, \cdots, i+w-1$, since bz is uniformly distributed and shifted to index 2^{i+1}.

7. If $i \geq w-l$, then $h_a(x) \neq h_a(y)$, since ax and ay are different in bit i. If $i < w-l$, then, because of carried, we could have $h_a(x) = h_a(y)$ if bits $w-l, \cdots, w-1$ of $a(y-x)$ are either 0s or 1s. Because of the uniform distribution, either event have a probability of $\frac{1}{w^l}$ of happening, which gives a combined probability of $\frac{2}{2^l}$, which concludes the proof.

5 NP Completeness

5.1 Disposition

- Introduction
  - Decision problems
  - Polynomial time
  - Languages
  - Encoding
- Classes
  - P and NP
  - co-NP
  - NP-Hard
  - NP-Completeness
- `CIRCUIT-SAT`
  - `SAT` in NP
  - `CIRCUIT-SAT` is reducible to `SAT`
  - Example of `CIRCUIT-SAT`

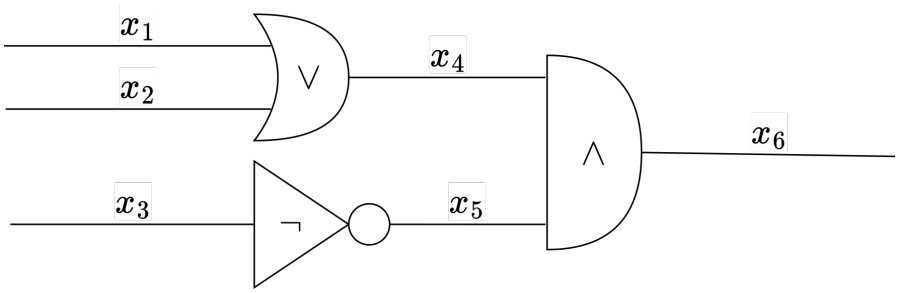


Figure 1: Example of circuit

  - `SAT` is NP-Complete

5.2 Walk-through

1. Intro: This topic is about problems and the complexity of these problems.

2. Decision problems is "Yes" or "No" problems. This means that the problem consist of a yes-instance, which we can map to 1 and a no-instance, which we can map to 0.

3. Polynomial time (definition): An algorithm solves a problem in polynomial time if for any instance of size n, the algorithm return a solution (0 or 1) in time $O(n^k)$, where k is some constant.

4. Language: An alphabet is a finite set Σ of symbols. A language L over alphabet Σ is a set of strings of symbols from Σ.

5. Encoding: We can say that instances of decision problems are encoded as binary strings and as mentioned we map the solution to 1 or 0.

6. Classes: A complexity class is a set of computational problems of related resource-based complexity. The two most commonly analyzed resources are time and memory.

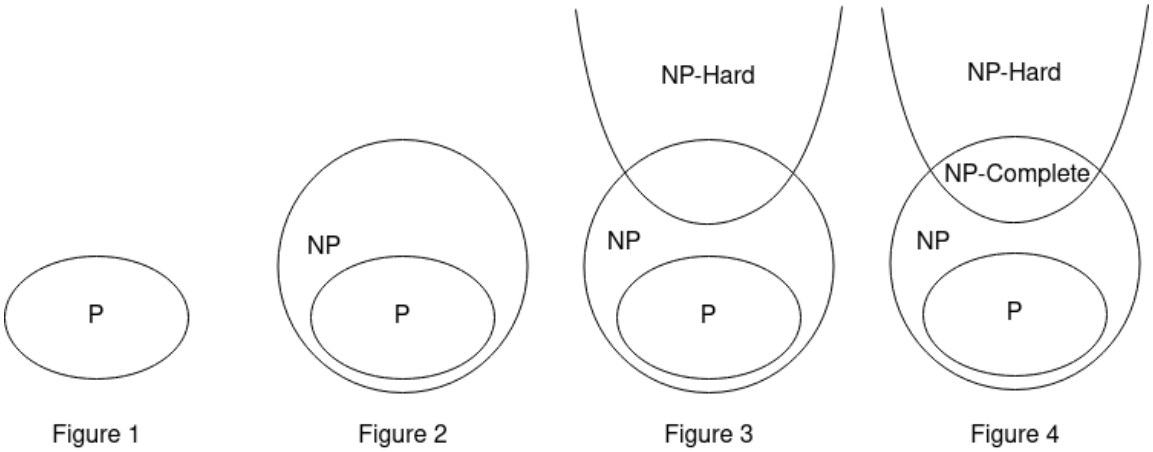


Figure 2: Drawings of the classes for the blackboard

7. P: The complexity class P contains all decision problems that can be solved by a deterministic Turing machine using polynomial time. Draw figure 1.

8. NP: NP is the set of decision problems verifiable in polynomial time by a deterministic Turing machine. They are not necessarily solvable in polynomial time. Draw figure 2.

9. Co-np: A decision problem X is a member of co-NP if and only if its complement X is in the complexity class NP.

10. NP-Hard: NP-Hard is a class of problems that are informally "at least as hard as the hardest problems in NP". A more precise definition is: a problem X is NP-hard when every problem Y in NP can be reduced in polynomial time to X, which means that we can use X's solution to solve Y in polynomial time. Draw figure 3.

11. Bonus about NP-hard: So finding a polynomial time algorithm to solve any NP-hard problem would give polynomial time algorithms for all the problems in NP. But since we suspect that P $\neq$ NP, it is unlikely that such an algorithm exists. The drawing I have made is valid under that assumption.

12. NP-Complete: This leads us to the complexity class NP-Complete. Draw figure 4. An NP-Complete problem is both in NP and NP-hard. This means that the correctness of a solution can be verified in polynomial time, but also that the problem can be used to simulate every other problem for which we can verify quickly that a solution is correct. In this sense, NP-complete problems are the hardest of the problems to which solutions can be verified quickly. If we could find solutions of some NP-complete problem quickly, we could quickly find the solutions of every other problem to which a given solution can be easily verified.

13. `SAT`: An example of a NP-Complete problem is the Boolean satisfiability problem (`SAT`). The problem is, given a formula, to check whether it is satisfiable - yes or no. A formula is satisfiable if it can be made TRUE by assigning appropriate logical values to its variables.
I will now show that `SAT` is NP-Complete. So we have to

    - show that `SAT` $\in$ NP.
    - pick another language known to be NP-Complete. Here, `CIRCUIT-SAT`.
    - show that `CIRCUIT-SAT` $\leq_P$ `SAT`. (show that `CIRCUIT-SAT` is polynomial-time reducible to `SAT`)

14. `SAT` $\in$ NP:

    - To show that `SAT` $\in$ NP, I construct a verification algorithm A that takes x and y as inputs.
    - x is a boolean formula ϕ and y is an assignment of values to variables of ϕ.
    - A returns 1 if y defines a satisfying assignment for x and returns 0 otherwise.
    - We can easily make this A run in polynomial time.
    - Thus, `SAT` $\in$ NP.

15. `CIRCUIT-SAT` $\leq_P$ `SAT`

    - Given a circuit C, I transform it into a boolean function ϕ as follows. Let's first draw a circuit. Example:

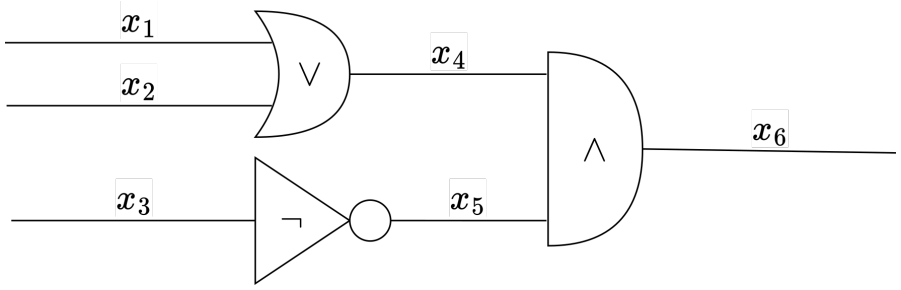


Figure 3: Example of circuit

    - Then I associate a variable x_i with each wire of C for $i = 1, ..., m$. Let x_m be the output wire variable. So x_6 in my example.
    - We can view each gate of C as a function, mapping the values on its input wires to the value on its output wire.
    - I construct a sub-formula for each of these functions.

$$\phi_1 = (x_4 \leftrightarrow x_1 \vee x_2)$$
$$\phi_2 = (x_5 \leftrightarrow \neg x_3)$$
$$\phi_3 = (x_6 \leftrightarrow x_4 \wedge x_5)$$

    - I define the full boolean formula to be $x_6 \wedge \phi_1 \wedge \phi_2 \wedge \phi_3$.

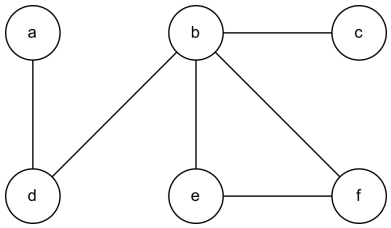
- ϕ can be constructed in polynomial time.
- This shows that, C is satisfiable if and only if ϕ is satisfiable.
- Thus, I have shown that `SAT` is NP-Complete.

16. Done.

6 Exact Exponential Algorithms and Parameterized Complexity

6.1 Disposition

- Introduction
  - Exact exponential algorithms
  - Parameterized complexity
- Exact exponential algorithms
  - Exact `TSP` via dynamic programming
  - Running time and space
- Parameterized problems
  - `K-VERTEX-COVER`



  - Proof of running time

6.2 Walk-through

1. When we solve problems, we usually want algorithms that

   (a) finds a exact solution
   (b) for all inputs
   (c) in polynomial time.

   Unfortunately, some problems are hard and we must loosen a bit up on some of the constraints.

   - We might instead of an exact solution, find an approximation with an approximation algorithm but that is not my topic for this exam.
   - Instead of using polynomial time, we might allow exponential time and that is called exact exponential algorithms.
   - Instead of making an algorithm work for all inputs, we might allow only inputs of small fixed values of some parameter. That is called parameterized algorithms.

   The last two kinds is today's topic.

2. Travelling salesman: There is a hard problem called the travelling salesman problem, for which we will settle for an exponential time algorithm in order to get the exact solution for all inputs. The problem is, that a salesman has to visit n cities exactly once and return to where he started. Given a set of distinct cities $\{c_1, c_2, \cdots, c_n\}$. Between each of the cities, there is a distance, we will denote as $d(c_i, c_j)$. The salesman wants to travel the shortest amount, meaning, we minimize the total length, the salesman has to travel.

3. The naive brute force approach to this problem is to enumerate all possible permutations of cities, of which there are $n!$ and find the shortest. Using dynamic programming, we can obtain a much faster algorithm.

4. Idea: For every pair (S, c_i), where S is a nonempty subset of the cities, not including c_1, and $c_i \in S$, the algorithm computes the value $\mathrm{OPT}[S, c_i]$. This value is the minimum length of a tour, starting in c_1, visiting all cities of S and ends in c_i.

5. We compute the values of $OPT[S, c_i]$ in order of increasing cardinality of S. In the case where S only contains 1 city, $OPT[S, c_i] = d(c_1, c_i)$. For cases where S contains more than 1 city, we can express $OPT[S, c_i]$ in terms of subsets of S:

   $$OPT[S, c_i] = \min\{OPT[S \setminus c_i, c_j] + d(c_j, c_i) \ : \ c_j \in S \setminus c_i\} \tag{4}$$

   It follows that, if in some optimal tour in S terminating in c_i and with c_j immediately preceding c_i then

   $$OPT[S, c_i] = OPT[S \setminus c_i, c_j] + d(c_j, c_i)$$

   This means we obtain (4), buy taking the minimum over all cities that can precede c_i. This also means, that the value OPT of the optimal solution is the minimum of

   $$OPT[\{c_2, c_3, \cdots, c_n\}, c_i] + d(c_i, c_1)$$

   where the minimum is taken over all indices $i \in \{2, 3, \cdots, n\}$.

6. Running time and space: The amount of steps required to perform (4) for a fixed set S of size k and with all vertices $c_i \in S$ is $O(k^2)$. (4) is computed for every subset S of cities in the algorithm, of which there are

   $$\sum_{k=1}^{n-1} O\left(\binom{n}{k}\right)$$

This means that is takes

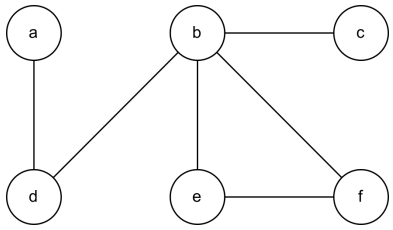
$$\sum_{k=1}^{n-1} O\left(\binom{n}{k} k^2\right) = O(n^2 2^n)$$

time to compute OPT in total. This is a quite significant improvement to from the trivial enumeration algorithm taking $O(n!n)$ time, to this algorithm.

7. It is worth noting, that this algorithm is exponential in space as well. The algorithm uses $O(2^n)$ memory, since we have to keep all the values of $OPT(S, c_i)$.

8. Parameterized problems: Now I will move on the parameterized problems.

9. `K-VERTEX-COVER` via bounded search tree: The "Bar fight prevention" problem also known as the `K-VERTEX-COVER` problem, is a hard problem that seemingly cannot be solved exactly in polynomial time for all inputs. Therefore we decide to only solve it for inputs of a fixed value of some parameter. The problem is:
You are a bouncer that wants to prevent fights at a bar. Denote a graph G of n nodes, where nodes represent bar guests and edges are fights, for an example guest a will fight guest b if they share an edge. You can at most reject k guests from the bar. Thus, this problems translates to: Is there a vertex cover of G of size at most k?



Trying all possibilities would result in the running time $O(2^n)$. But by restricting the parameter k, we can find a better solution. I would like to present the algorithm called `K-VERTEX-COVER` via bounded search tree.

- You can always add a node with degree $d \geq k+1$ to your vertex cover, since if you did not, you would have to add all $k+1$ neighbours to the vertex cover and that would break the cover size of k.
- So after adding those, all the remaining nodes will have at most degree k.
- We know that every edge has to be covered. The only way to do this is to include one of it's endpoints in the cover.
- Thus, we can do the following: For a given edge $\{u, v\}$, try adding u to the cover and run the algorithm recursively to check whether the remaining edges can be covered using at most $k-1$ vertices.
- If this succeeds, we have a solution.
- If not, we replace u by v and does the same.
- If this also fails, then we are guaranteed that no vertex cover of size k exists.

10. Proof of running time: For each recursive call, we decrease k by 1. When k reaches 0, all the algorithm has to do is to check if there are remaining edges in the graph not covered by the proposed vertex cover. Each call spawn two recursive calls, which we do at most k times, giving a total of at most 2^k recursive calls. Let m be the number of edges. Each call runs in linear time $O(n+m)$, since we have to check if all edges are covered.

    We know something about the number of edges m. Since each node has at most k edges

(because we removed nodes with more edges), there is at most $\frac{nk}{2}$ edges in the graph. Thus $m = O(nk)$ and $O(n+m) = O(nk)$.

The total runtime is thus $O\left(2^k nk\right)$.

11. Done.

6.3 Extra

6.3.1 Dynamic programming

Comments about dynamic programming (just good to know):

- Dynamic programming requires "Optimal Substructure" but subproblems may be overlapping.
- Instead of recursively solving smaller disjoint subproblems, "Dynamic Programming" solves all smaller subproblems in order of increasing size.
- A hybrid idea called "Memoization" does the same by using recursion, but caching results so each subproblem is only solved once.
- In our TSP example the original problem does not have the optimal substructure property (A piece of an optimal tour does not have to be an optimal tour of some subgraph). The trick is to notice that the problem of computing $OPT[\{c_2, ..., c_n\}, c_i]$ does have the property, and that TSP can be solved once we know that for all c_i.

6.3.2 FPT and XP

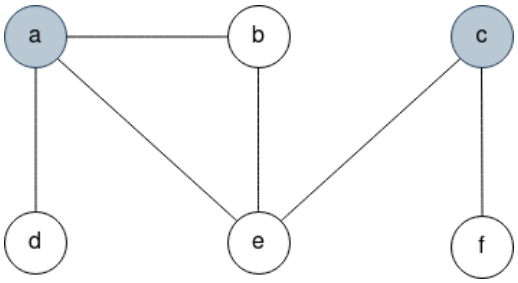
Fixed parameter tractable (FPT) VS Slice-wise polynomial (XP). FPT vs XP: As just seen, an important feature of the Bar Fight Prevention problem is the existence of the parameter k. In this case k is the maximum solution size, but other problems may have different parameters.
A parameterized problem can have some properties that I would like to mention

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

7 Approximation Algorithms

7.1 Disposition

- Introduction
  - Motivation
  - Approximation ratio
- `VERTEX-COVER`
  - Example
  - Draw example:

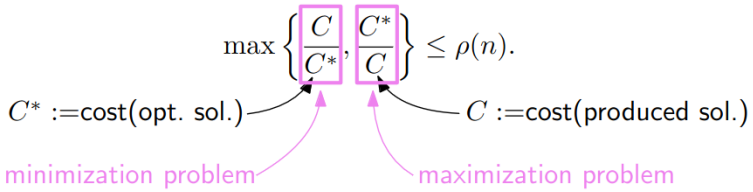


  - `APPROX-VERTEX-COVER`'s running time
  - Proof of 2-approximation
- `3-SAT`
  - `MAX-3-SAT`
  - Proof of $\frac{8}{7}$-approximation

7.2 Walk-through

1. Motivation: As we know from NP-completeness, some problems are harder to solve than others. There exists problems that we cannot expect to solve in polynomial time, but nevertheless we still might want to solve them - and also in reasonable time.
   So, one way of dealing with these problems is to make do with near optimal solutions instead of exact optimal solutions. This is the purpose of approximation algorithms. We make a trade off between the exactness of the solution, with the running time of the solution. Thus, approximation algorithms are faster, but inaccurate to some extent.

2. Approximation ratio: In order to give a bound on how far away from the optimal solution an approximation is, we use the concept of $\rho(n)$-approximations. That means, that for some input of size n, the approximated solution is at most some factor $\rho(n)$ worse than the optimal solution. We can only approximate solutions for optimization problems, and not decision problems, btw. We measure this by



3. Now, I will give some examples of approximation algorithms for the two problems `VERTEX-COVER` and `3-SAT`.

4. Draw example:

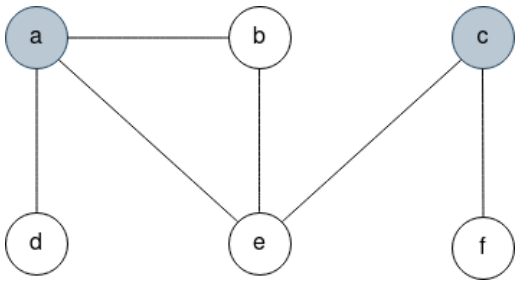


Figure 4: Example of graph

5. `VERTEX-COVER` (definition): Let $G = (V, E)$ be a graph. A set $V' \subseteq V$ of vertices is a vertex cover if for all $uv \in E$, we have $u \in V'$ or $v \in V'$. Colour the vertices on the graph. Finding a `MINIMUM-VERTEX-COVER` is an NP-hard problem, and therefore we do not know an polynomial time solution for it. Hence, an approximation algorithm has been made.

6. `APPROX-VERTEX-COVER`: is a 2-approximation algorithm and I would like to show that. This means I have to show 2 things:

   - That the algorithm is fast, meaning it runs in polynomial time
   - and that the approximated solution is at most twice as bad as the optimal solution.

7. Running time: The algorithms works by choosing an edge from the set of edges $E(G)$ in the graph. The two vertices u, v of the edge will be added to the cover C and all edges incident on u or v will be removed from $E(G)$. When no more edges are left in $E(G)$, the cover will

be returned. This has a running time of $O(|V| + |E|)$, since we go trough all the edges and potentially adds all vertices to the cover.

8. Proof of 2-approximation: Let A denote the set of edges that are chosen by the algorithm from the set of edges. Any vertex cover, including the optimal cover $C^\star$, must cover the edges in A, meaning that for each edge in A, either of its endpoints must be in the vertex cover. Since no two edges in A share endpoints, at least one node for each edge must be in a vertex cover, and we have a lower bound of

$$|A| \leq |C^*|$$

Because we for each added edge add max 2 vertices to the cover, we have that

$$|C| = 2|A|$$

So we can now bound the approximated solution by

$$|C| = 2|A| \leq 2\,|C^*|\,,$$

which can be rewritten as

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

9. `3-SAT`: Now, lets talk about another problem, which to our knowledge cannot be solved in polynomial time. The 3-satisfiability problem, which is the same at the boolean satisfiability problem, but with the restriction that each clause is limited to at most 3 literals. The goal is to satisfy the whole formula by satisfying each clause.

10. `MAX-3-SAT`: There have been created an approximation algorithm called `MAX-3-SAT`, which takes the same input as `3-SAT`, but the goal is instead to return the assignment of the variables that maximise the number of clauses evaluating to 1. The problem has been turned into an optimisation problem.

11. Proof of $\frac{8}{7}$-approximation: By randomly flipping a coin for each variable, we get an equal chance of it being 1 or 0. This will yield a randomized $\frac{8}{7}$-approximation algorithm. I would like to prove that now:

    - Suppose that we have independently flipped a coin for each variable. For $i = 1, 2, \ldots, m$, we define the indicator random variable

    $$Y_i = \begin{cases} 1 & \text{if clause } i \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}$$

    so that $Y_i = 1$ as long as we have set at least one of the literals in the i'th clause to 1.

    - Since no literal appears more than once in each clause, and since we have assumed that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent.

    - A clause is not satisfied only if all three of its literals are set to 0 , and so $\Pr[\text{clause } i \text{ is not satisfied}] = \left(\frac{1}{2}\right)^3 = \frac{1}{8}$. Thus we have $\mathbb{E}[Y_i] = 1 - \frac{1}{8} = \frac{7}{8}$.

- Let $Y = Y_1 + Y_2 + \ldots + Y_m$. Then we have

$$\begin{aligned}\mathbb{E}[Y] &= \mathbb{E}\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} \mathbb{E}[Y_i] \qquad \text{by linearity of expectation} \\ &= \sum_{i=1}^{m} \frac{7}{8} \\ &= \frac{7m}{8}.\end{aligned}$$

- Clearly, m is an upper bound on the number of satisfied clauses, since m is the number of clauses, and thus we have $C^* \leq m$.
- We have also just shown that $C = \frac{7m}{8}$ in expectation. Hence the approximation ratio is at most

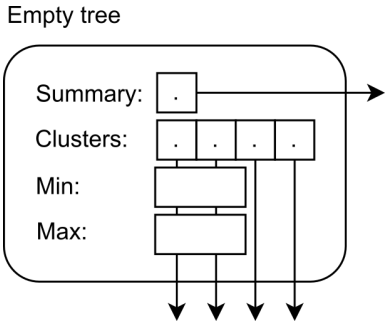
$$\frac{C^*}{C} \leq \frac{m}{\frac{7m}{8}} = \frac{8}{7}.$$

12. Done.

8 van Emde Boas Trees

8.1 Disposition

- Introduction
  - Motivation
- Construction of a vEB tree
  - Example of vEB tree: `INSERT` {1,2,7,4,5,6}
  - Operations on the vEB tree



- Proof of running time

8.2 Walk-through

1. Motivation: van Emde Boas trees are an interesting data structure because of the bound on the running time of the operations. When you know the range of the set of keys you want to sort, you can use this information to construct a sorting algorithm that runs asymptotically faster than the usual $O(n \log n)$ (i.e. counting sort in linear time). The same idea on data structures results in the van Emde Boas trees-structure. van Emde Boas trees support the operations: `SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR` and `PREDECESSOR` all in $O(\lg \lg u)$ time, where u is the size of the universe of keys. This universe is any exact power of 2.
   The catch is of course that the keys must be in the range 0 to $u - 1$ and no duplicates are allowed.

2. Helpful definitions: We will allow the universe size u to be any exact power of 2 , and when $\sqrt{u}$ is not an integer, then we will divide the $\lg u$ bits of a number into the most significant $\left\lceil \frac{(\lg u)}{2} \right\rceil$ bits and the least significant $\left\lfloor \frac{(\lg u)}{2} \right\rfloor$ bits. For convenience, we denote the resulting numbers as the upper and lower square root of u.

$$\sqrt[\uparrow]{u} = 2^{\left\lceil \frac{(\lg u)}{2} \right\rceil}$$
$$\sqrt[\downarrow]{u} = 2^{\left\lfloor \frac{(\lg u)}{2} \right\rfloor}$$

   We define some helpful functions. We can use high to get the number on the most significant bits of the number x. Low tells us the number on the least significant bits. We can use the index-function to get the full number, when we have the number on the most and least significant bits.

$$\begin{aligned} \text{high}(x) &= \left\lfloor \frac{x}{\sqrt[\downarrow]{u}} \right\rfloor, \\ \text{low}(x) &= x \bmod \sqrt[\downarrow]{u}, \\ \text{index}(x, y) &= x\sqrt[\downarrow]{u} + y \end{aligned}$$

3. Construction: A van Emde Boas tree ($vEB(u)$) contains

   - the universe size u (who is an exact power of 2)
   - the elements min and max
   - a pointer called *summary* to a $vEB(\sqrt[\uparrow]{u})$ tree
   - and an array $cluster[0...\sqrt[\uparrow]{u}]$ of $\sqrt[\uparrow]{u}$ pointers to the $vEB(\sqrt[\downarrow]{u})$ trees.

   Now let's construct a tree (see the drawings for how to draw the example):

   (a) Let's start with an empty vEB tree. If we know that a vEB tree is empty, we can insert into it in constant time, by only updating its min and max properties. Similarly, if the tree contains exactly 1 element, we can delete in constant time. These properties will allow us to cut short the chain of recursive calls.

   (b) We can also easily tell if a vEB tree contains 0, 1, or at least 2 elements in constant time.

   (c) I add 1 and 2 to the tree as min and max. A very nice property is that the the `MINIMUM` and `MAXIMUM` operations do not need to recurse, for they can just return the value of min or max.

   (d) Let's insert 7.

   (e) Let's insert 4.

   (f) Let's insert 5.

(g) Let's insert 6.

(h) Let's use the `SUCCESSOR` operation. The operation can avoid making a recursive call to determine whether the successor of a value x lies within high(x). That is because x's successor lies within its cluster if and only if x is strictly less than the max attribute of its cluster. A symmetric argument holds for `PREDECESSOR` and min.

(i) Find successor of 5.

(j) Find successor of 2.

4. Running time: The recursive procedures that implement the vEB-tree operations will all have running times characterized by the recurrence:

$$T(u) \leq T(\sqrt[\updownarrow]{u}) + O(1)$$

The trees shrink by a factor of $\sqrt[\updownarrow]{u}$ for each level. So when an operation traverses through the data structure, it will spend a constant amount of time at each level and then recurse to the level below. Thus, will the recurrence characterize the running time of the operation. We now want to show that this running time is $O(\lg \lg u)$ Letting $m = \lg u$, so that $u = 2^m$ we rewrite as

$$T(2^m) \leq T\left(2^{\lceil \frac{m}{2} \rceil}\right) + O(1).$$

Noting that $\left\lceil \frac{m}{2} \right\rceil \leq \frac{2m}{3}$ for all $m \geq 2$, we have

$$T(2^m) \leq T\left(2^{\frac{2m}{3}}\right) + O(1).$$

Now we rename $S(m) = T(2^m)$, giving us the new recurrence

$$S(m) \leq S\left(\frac{2m}{3}\right) + O(1),$$

which, by case 2 of the master method, has the solution $S(m) = O(\lg m)$.

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$
$$S(m) \leq S\left(\frac{2m}{3}\right) + O(1)$$

Since

$$a = b^d$$
$$1 = 3^0$$

Then we have case 2

$$O(n^d \lg n)$$
$$O(1 \lg m)$$

Thus we have

$$T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u).$$

5. Done.

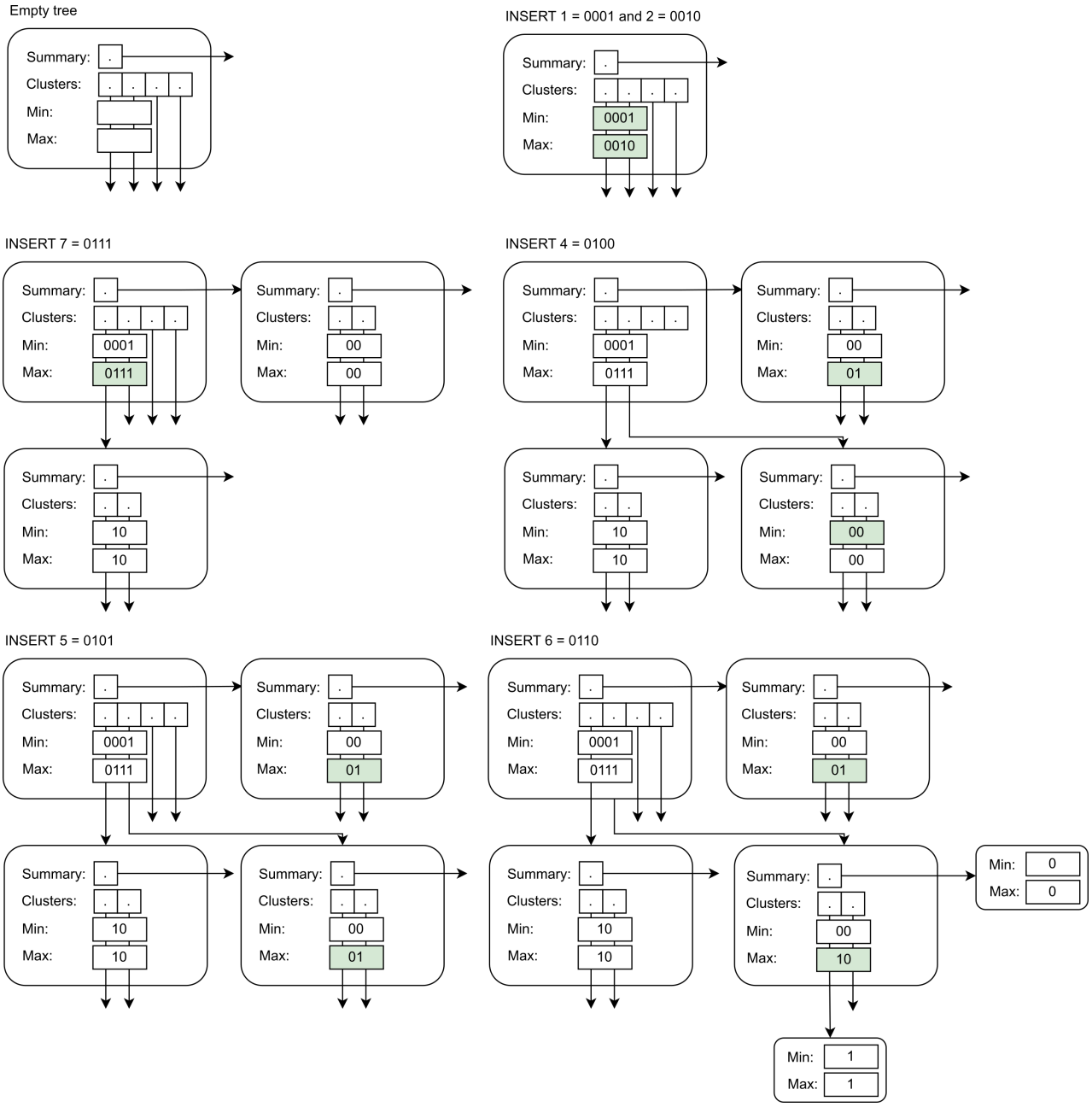


Figure 5: Example of `vEB` tree and operations

8.3 Extra

8.3.1 Master theorem

Let $T(n)$ be a monotonically increasing function that satisfies

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$
$$T(1) = c$$

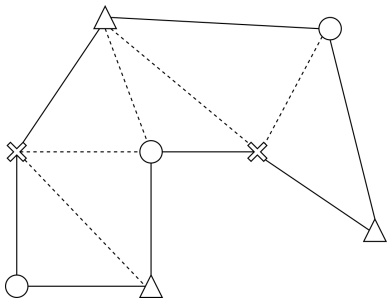
where $a \geq 1, b \geq 2, c > 0$. If $f(n) \in \Theta\left(n^d\right)$ where $d \geq 0$, then

$$T(n) = \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log n\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_b a}\right) & \text{if } a > b^d \end{cases}$$

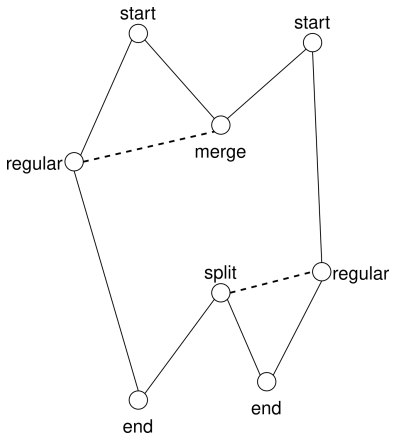
9 Polygon Triangulation

9.1 Disposition

- Introduction
  - Art Gallery Problem
- Triangulation of simple polygon
  - Induction proof
- Art Gallery Theorem
- 3-coloring



- y-monotone polygons
  - Partitioning a polygon into monotone pieces
  - Triangulating a monotone polygon



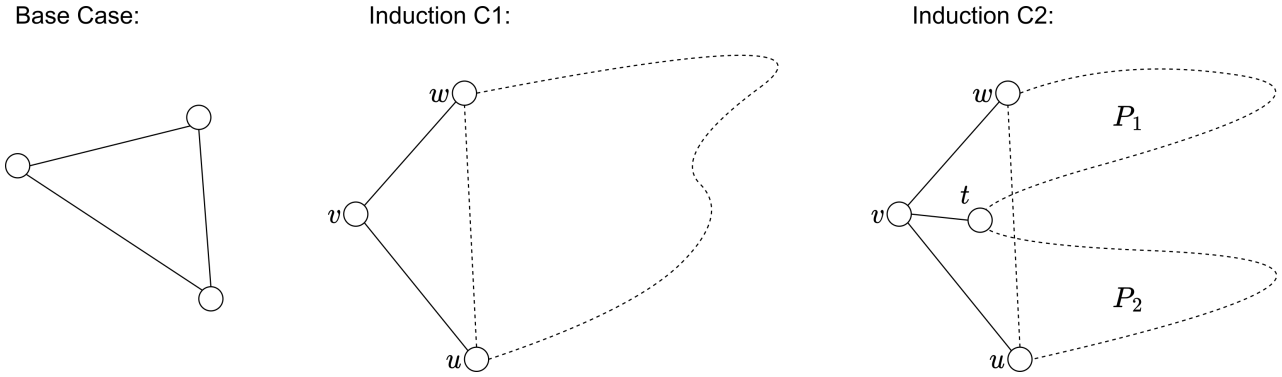
9.2 Walk-through

1. Art Gallery Problem: To protect an art gallery from thieves we want to place cameras in the gallery. Every part of the gallery must be visible to at least one of the cameras, but we want to use as few cameras as possible for cost effectiveness. This problem is called the *Art Gallery Problem*: How many cameras are needed to cover a given art gallery (a polygon with n vertices)? And how do we decide where to place them? This is where polygon triangulation becomes useful.

2. Triangulation of a simple polygon: Let's specify our art gallery problem a bit more. The floor plan of a gallery is a polygon P with n vertices. We restrict it to be a simple polygon, which means that we do not allow holes in the polygon. So how many cameras do we need to guard a simple polygon? We decompose P into pieces that are easy to guard: triangles! We do this by drawing diagonals between pairs of vertices. A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals is called a triangulation. A triangulation is not necessarily unique.

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

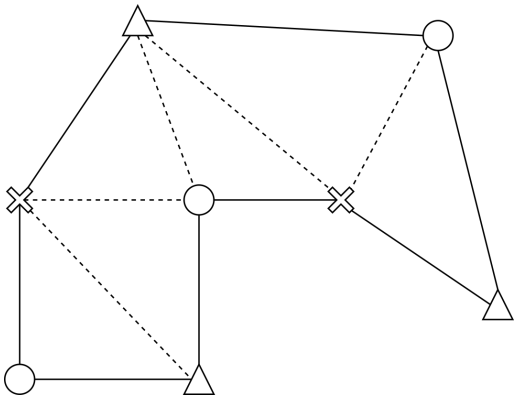
4. Proof: Induction on n.



   - Base case: $n = 3$ is trivial.
   - Induction step: v is the leftmost vertex. u and w is the neighbours.
   - Case 1: uw is a diagonal. Induction hypothesis means that there exists a triangulation with $n-3$ triangles on the other side of uw. See drawing of case 1.
   - Case 2: uw is not a diagonal. Let t be the corner in uvw farthest from uw. vt is a diagonal, that splits P into P_1 and P_2 with $m_1 < n$ and $m_2 < n$ vertices. $m_1 + m_2 = n + 2$, because every vertex of P occurs in exactly one of the two sub-polygons, except from v and t, which occur in both sub-polygons.
     The induction hypothesis means that P_1 has $m_1 - 2$ triangles and P_2 also has $m_2 - 2$ triangles. So in total $m_1 + m_2 - 4 = n + 2 - 4 = n - 2$.

5. Art Gallery Theorem: So the lemma implies that any simple polygon can be guarded with $n-2$ cameras. But that might seem a bit overkill. The Art Gallery Theorem states that: for simple polygon with n vertices $\lfloor n/3 \rfloor$ cameras are occasionally necessary and ALWAYS sufficient. We would like to prove that now.

6. 3-coloring: We can select a subset of vertices, such that any triangle in the triangulation T has at least one selected vertex and then place the camera at the selected vertices. To find such a subset we assign each vertex of P a colour or symbol: cross, circle and triangle. Or in this case symbols circle, cross and triangle.

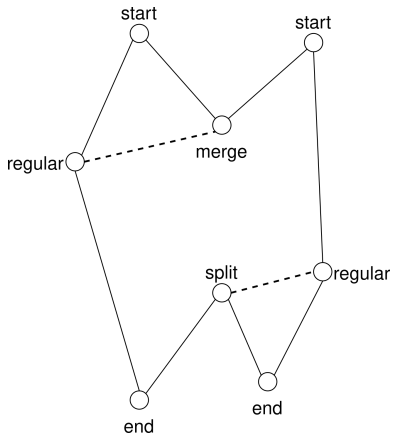


Then we can place the cameras at all the "cross" vertices and we will have guarded the whole polygon. By choosing the smallest colour class, we can guard P using $\lfloor \frac{n}{3} \rfloor$ cameras.
But does a 3-coloring always exist? Yes! We look at the dual graph of the triangulation. The dual graph G has a node for very triangle in T and an edge over each diagonal. This means that removing any edge will split the graph in two, and thus G is a tree. We can traverse through the tree and always give the node a new colour (might need expanding on).

7. Now we know that $\lfloor \frac{n}{3} \rfloor$ cameras are always sufficient. But we still need to compute the camera positions. We need a fast algorithm for triangulating a simple polygon. We can do that by partitioning the polygon into monotone pieces and then triangulate the monotone polygons.

8. Partitioning a polygon into monotone pieces: A polygon that is monotone with the respect to the y-axis is called y-monotone. If we walk from a topmost to a bottom-most vertex along one of the side of the polygon, we would always move downwards or horizontally - never upwards. Each vertex has a special property. It is either a

   - start,
   - merge,
   - regular,
   - split,
   - or end vertex.



There is a more advanced algorithm for how to deal with these vertices, but this is the gist of it:

- Sweep trough the polygon from top to bottom and when you meet either a
- Merge vertex: Make a diagonal to the next visited vertex.

- Split vertex: Make a diagonal to the previously visited vertex.

The polygon is now split into monotone pieces.

9. Triangulating a monotone polygon: After decomposing the polygon into y-monotone polygons, we triangulate it. This is done with an algorithm that pushes and pops the vertices and the running time is $O(n \lg n)$.

,

9.3 Extra

9.3.1 TRIANGULATE-MONOTONE-POLYGON

Input: A strictly y-monotone polygon P stored in a doubly-connected edge list D.

Output: A triangulation of P stored in the doubly-connected edge list D.

1. Merge the vertices on the left chain and the vertices on the right cain of P into one sequence, sorted on decreasing y-coordinate. The left chain is vertices to the left of a line between the start and end vertices, and the right chain are vertices on the right.
   If two vertices have the same y-coordinate, then the leftmost one comes first.
   Let $u_1, \cdots, u_n$ denote the sorted sequence.
2. Initialise an empty stack S and push u_1 and u_2 onto it.
3. **for** $j = 1$ to $n - 1$
4. **if** u_j and the vertex on top of S are on different chains **then**
5. Pop all vertices from S.
6. Insert into D a diagonal from u_j to each popped vertex, except the last one.
7. Push u_{j-1} and u_j onto S.
8. **else** Pop one vertex from S.
9. Pop the other vertices from S as long as the diagonals from u_j to them are inside P. Insert these diagonal into D. Push the last vertex that has been popped back onto S.
10. Push u_j onto S.
11. Add diagonals from u_n to all stack vertices except the first and the last one.