Advanced Algorithms - Notes

Jens, Jacob, Bjarke, Nicklas

October 2016

Contents

1 Randomized Algorithms

Randomized Algorithms are simple, fast and easy to implement. We are introduced to two types of randomized algorithms.
Las Vegas algorithms always produce an optimal solution, but the running-time cannot be predicted. We can however measure an expected running-time.
Monte Carlo algorithms has a time-complexity, which we can bound. However it is never certain if an optimal solution is found.

1.1 Randomized Quicksort

Randomized Quicksort is like the common Quicksort algorithm except that the pivot is chosen at random at each recursive step. Whereas Quicksort has a worstcase running-time of $O(n^2)$ directly bound to a certain input, the expected running-time of Randomized Quicksort is independent of the input. The running-time however cannot be predicted and hence Randomized Quicksort is a Las Vegas algorithm.

1.1.1 Analysis with proof

Hn = (Harmonic Number) = $\sum_{i=1}^{n} \frac{1}{i} = \frac{1}{1} + \frac{1}{2} ... \frac{1}{n-1} + \frac{1}{n} \simeq ln(n) + \theta(1)$
(We are expected to know that $Hn \simeq ln(n) + \theta(1)$, but we don't have to understand why)

The expected number of comparisons $E[x] < 2n\ Hn$

When we do Quicksort we apply it to a list of numbers. We label our numbers with i and j, where it holds that $S_{(i)} < S_{(j)}$.
We define indicator $X_{ij} = [S_{(i)}$ is compared with $S_{(j)}]$
$E[x] = E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}] = \sum_{i<j} \Pr[X_{ij}]$

$\Pr[X_{ij}]$ = The probability that $S(i)$ or $S(j)$ is picked first (as pivot) among $S(i), S(i+1), ..., S(j)$
This observation is important and should be explained at the exam. Imagine a list of sorted numbers $[A < S_{(i)} < B < S_{(j)} < C]$, where Randomized Quicksort is applied on a list of the same numbers, but not necessarily in that order. If a number $b \in B$ is picked as the pivot, then it will push $S_{(i)}$ to the left of b and $S_{(j)}$ to the right of b and then $S_{(i)}$ and $S_{(j)}$ will never be compared.

$\Pr[X_{ij}] = \frac{2}{j-i+1}$, because $j - i + 1$ is the size of the set between (and including) $S_{(i)}$ and $S_{(j)}$. The expected number of comparisons is then just summing the probability of comparing a pair for all pairs.

$E[x] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$, lets focus on the last part of the equation.

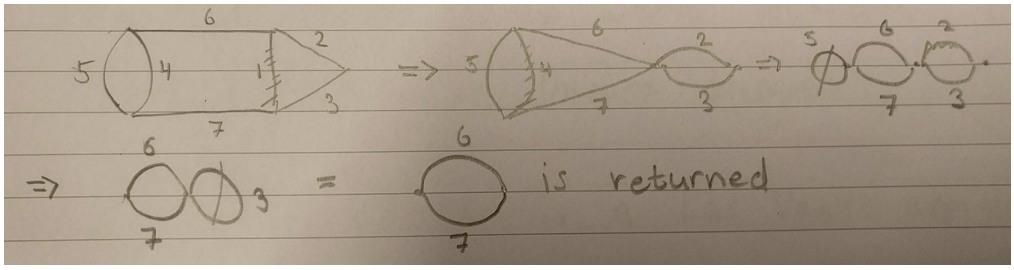
$\sum_{j=i+1}^{n} \frac{2}{j-i+1} = (\frac{2}{2} + \frac{2}{3} ... \frac{2}{n-i+1}) < (\frac{2}{2} + \frac{2}{3} ... \frac{2}{n}) = \sum_{k=1}^{n} \frac{2}{k} = 2Hn$

Summing this over all n gives us, $E[x] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} < n2Hn$

1.2 Randomized Min Cut

For an undirected graph $G(V, E)$ a cut is a set of edges, that divides the graph in two. Multiple cuts might exists, and a Min Cut is a cut of minimal size. For finding a Min Cut, the following Monte Carlo algorithms is introduced.

```
Repeat
    Pick random edge e
    Contract it and remove loops
Until two vertices remain.
Return edges between them.
```



Contracting an edge, means that we merge the two adjacent vertices, a and b, into one vertex, c. All edges connected to either a or b gets connected to c instead. The final set of edges returned represents a cut.
What the algorithm does is simply finding a random cut. We repeat it a number of times and save the cut of minimal size we have found thus far.

1.2.1 Analysis with proof

Let C be a min cut, such that $C \in E$ and $|C| = K$. Our algorithm finds a min cut, if C survives. **Proof: C survives with probability** $2/n(n-1)$

Let $G_i(V_n, E_n)$ be the resulting graph after contracting $e_1 ... e_n$. (Then $G_0 = G$).
Let event $\varepsilon_i = [e_i \notin C]$, then C survives iff $\varepsilon_i \wedge \varepsilon_2 ... \varepsilon_{n-2}$. We call that event ε.

G_1 is the graph after the first contraction. It has $n-1$ nodes.
Also the number of edges must follow, $|E_1| \geq \sum k/2 = nk/2$. This is true, since every node must have degree k or larger. Otherwise you could cut that node of, through all adjacent nodes and then have a min cut of size less than k.

Probability of picking an edge in C in the i'th contraction,
$\Pr[e \in C] = \frac{|C|}{|E_{i-1}|} \leq \frac{k}{n_{i-1} \cdot k/2} = \frac{2}{n_{i-1}} = \frac{2}{n-i+1}$
Probability of not picking an edge in C in the i'th contraction,
$1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}$
Probability of not picking an edge in C in all i'ths contractions,
$\prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdots \frac{4}{6} \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}$

All upper fractions cancels out with the lower fraction two steps to the right. All there is left is the two far left upper fractions $2 \cdot 1$ over two far left lower fractions $n \cdot (n-1)$

So the algorithm fails to find a min cut C with probability,
$1 - \frac{2}{n(n-1)} \leq e^{-\frac{2}{n(n-1)}}$
This holds because of the general rule $\forall x(1 + x \leq e^x)$

If we repeat the algorithm $10n^2$ times then it fails with probability,
$e^{-\frac{2}{n(n-1)}^{10n^2}} < e^{-20}$

The Min Cut algorithm is a Monte Carlo algorithm with One-sided error. We can get a too big C, but never a too small C.

2 Hashing

2.1 Definitions

We're given a hash-function $h : U \to [m]$, where U is a universe of keys (e.g. 64-bit numbers) that we wish to map randomly to a range $[m] = \{0, \cdots, m-1\}$ of hash values. Thus $h(x)$ consists of a random variable for each $x \in U$.

We usually care about 3 things given a hash-function:

- **Space:** The size of the random seed that is necessary to calculate $h(x)$ given x.
- **Speed:** The time it takes to calculate $h(x)$ given x.

2.2 Universal Hashing

When we want to generate our random hash function $h : U \to [m]$, we want h to be **universal**, which means that for any given distinct keys $x, y \in U$, when h is picked at random (indenpendentally of x and y), we have a *low collision probability*:

$$Pr[h(x) = h(y)] \leq \tfrac{1}{m}$$

In some appilication we might have a constant c such that we'd have the following bound on the collision probability (Then we say that h is c-universal):

$$Pr[h(x) = h(y)] \leq \tfrac{c}{m}$$

2.2.1 Multiply mod prime

For $p \geq u > m$ we pick a prime number p and two variables $a \in [p]_+ = \{1, 2, \ldots, p-1\}$ and $b \in [p] = \{0, 1, \ldots p-1\}$. The multiply mod prime hash function $h_{a,b} : [u] \to [m]$ is then given by:

$$h_{a,b} = ((ax + b) \mod p) \mod m$$

Given distinct $x, y \in [u] \subseteq [p]$, we want to argue that for random a and b that:

$$\Pr_{a \in [p]_+, b \in [p]}[h_{a,b}(x) = h_{a,b}(y)] \leq \frac{1}{m}$$

Even though we have defined $a > 0$, we will consider all values of $a < p$ including zero, until the end of the proof.

Prime fact: *If p is prime and $\alpha, \beta \in [p]_+$ then $\alpha\beta \not\equiv 0 \mod p$*

Let $x, y \in [p], x \neq y$ be given. For given pair $(a, b) \in [p]^2$, define $(q, r) \in [p]^2$ by:

$$ax + b \mod p = q \tag{2.1}$$
$$ay + b \mod p = r \tag{2.2}$$

Lemma: Equation (2.1) and (2.2) define a 1-1 correspondence between pairs $(a, b) \in [p]^2$ and pairs $(q, r) \in [p]^2$.

Proof: For a given pair $(r, q) \in [p]^2$, we will show that there is at most one pair $(a, b) \in [p]^2$ that statisfies equation (2.1) and (2.2). By subtracting equation (2.1) from equation (2.2) modulo p, we get:

$$(ay + b) - (ax + b) \equiv a(y - x) \equiv r - q \mod p \tag{2.3}$$

We know claim that there is at most one a that satisfies (2.3). By contradiction suppose that there is another a' that satisfies (2.3), and then subtracting equation (2.3) with itself where a is substituted with a' we get:

$$a(y - x) - a'(y - x) \equiv (r - q) - (r - q) \mod p$$
$$(a - a')(y - x) \equiv 0 \mod p$$

But since we know that both $(a-a')$ and $(y-x)$ are non-zero, this contradicts the **Prime fact**. We can now find b that satisfies equation (2.1) and (2.2):

$$b = (q - ax) \mod p$$

We have now proved that for each pair $(q, r) \in [p]^2$, there is at most one pair $(a, b) \in [p]^2$ satisfying equation (2.1) and equation (2.2). This means that there is a 1-1 correspondence between the pairs $(q, r) \in [p]^2$ and $(a, b) \in [p]^2$.

Fact: Since we know $x \neq y$:

$$r = q \Leftrightarrow a = 0$$

Thus if we pick $(a, b) \in [p]_+ \times [p]$, we get $r \neq q$.

We only have $h_{a,b}(x) = h_{a,b}(y)$ iff $q \equiv r \mod p$, and we know that $q \not\equiv r$. This gives us for a given $r \in [p]$, $\lceil p/m \rceil - 1$ different values for q with $q \equiv r \mod p$. We now note that $\lceil p/m \rceil - 1 \leq \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$, thus the number of collisions pairs $(r, q) \in [p]^2, r \not\equiv q$ is bounded by $\frac{p(p-1)}{m}$. The 1-1 correspondence between the pairs (r, q) and (a, b), then implies that there are at most $\frac{p(p-1)}{m}$ collisions pairs $(a, b) \in [p]_+ \times [p]$. Since each of the $p(p-1)$ pairs in $[p]_+ \times [p]$ are equally likely, we conclude that the collision probability is $1/m$.

2.2.2 Multiply-shift

Multiply shift hashes from a w-bit integer to a l-bit integer. We pick a uniformly random odd w-bit integer a:

$$h_a(x) = \lfloor (ax \mod 2^w)/2^{w-l} \rfloor$$

We can think of the function as extracting $w-l, \ldots w-1$ bits from the product ax.

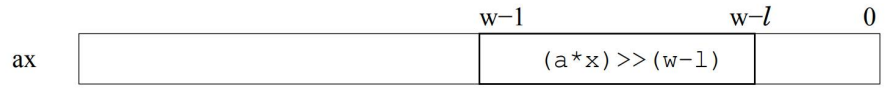


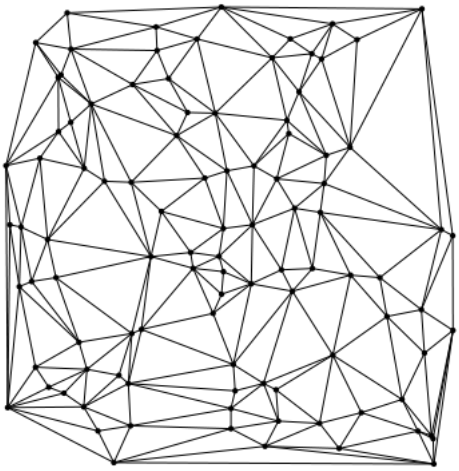
Figure 1: Caption

2.3 Strong Universality

A hash function is stronly universal if the probability is pairwise idependent. Expressed differently we say the every pairwise event $\Pr[h(x) = r \wedge h(y) = q] = 1/m^2$. We can show that a stronly universal hash function also i universal:

$$\Pr[h(x) = h(y)] = \sum_{q \in [m]} \Pr[h(x) = q \wedge h(y) = q] = m/m^2 = 1/m$$

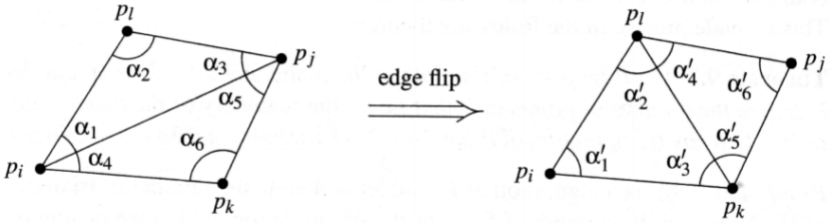
3 Delauney Triangulations



3.1 Problem Definition

Given an input of a set of points, we want to find the Delaunay triangulation, such that the vertices of the triangles matches the points.
There is a lot of ways to turn a set of points into a set of triangles. We want to find a set of triangles, such that we maximize the minimum angle. If the triangles contain no *illegal edges*, we have successfully found a Delaunay triangulation.

An edge is illegal if its counterpart has a larger minimum angle. In the picture below the left triangulation contains an illegal edge, so we prefer the other triangulation.



Observe that all the edges in the Delaunay Triangulation, which is not along the borders, lies in a square. We have the option to flip the edge, which might result in a more angle optimal solution. In the picture above all triangulations contains legal edges.
A Delaunay Triangulation is a set of legal triangulations.

3.2 The Algorithm

The pseudocode has detail left out. The sections are labeled, and will be explained.
The input is a set of vertices V of size n. There are no edges as input. The algorithm has an expected running-time of $O_e(n \cdot lg(n))$

```
Algorithm: DelaunayTriangulation
(1) Create big triangle covering all points

    while unpicked vertices remain:  ( n-1 times )
        Pick a random unpicked vertex v
    (2) Find current triangle t that v lies in   ( O(lg(n)) )
        Add 3 edges from v to all vertices of t
    (3) Check if surrounding triangles has gone illegal and fix it

(4) Remove initial big triangle
```

The algorithm is an *incremental algorithm*. This means we got a Delaunay Triangulation (DT), then we add a point to it, which might mess it up, but then we repair it.

(1) We start by adding a big triangle covering all vertices. The triangle is created, such that the top vertex is the vertex in V with the highest y-coordinate, which is found in $O(n)$ time. The two remaining vertices, which we will deal with in the end, are placed such that the big triangle covers all points. (always possible)

Now we got a Delaunay triangulation, which is just a single triangle. Then we add the remaining $n-1$ vertices - handling one at a time - where we will "repair" it every time.

We will deal with (2) later and focus on (3). Now just imagine we have found the triangle t, which out vertex v lies in.
(The case (5) where v does not lie in a triangle, by lying directly on an edge, is explained later).

When we add v there are 3 edges ($(v_1, v_2), (v_2, v_3)$ and (v_3, v_1)) that might have become illegal, which is the 3 edges of t.
To handle this we use a underlying function *LegalizeEdge*, which does nothing if the edge is legal, and fixes the edge if it is illegal.

We call LegalizeEdge(v,(v_1, v_2)), LegalizeEdge(v,(v_2, v_3)) and LegalizeEdge(v,(v_3, v_1)), where the order does not change the outcome.

3.2.1 Legalize Edge

```
LegalizeEdge(v, (vi,vj) )
let e = (vi,vj)
if (e is illegal) then
   let (vi,vj,vk) be the triangle adjacent to triangle (vi,vj,v)
   flip e  (which will now connect v and vk)
   LegalizeEdge(v, (vi, vk))
   legalizeEdge(v, (vj, vk))
```

Observe two things.

- LegalizeEdge calls itself recursively. Whenever an edge is flipped it must fix two adjacent potentially illegal triangles.
- Input v always remain the first input. All edges flipped will connect v.

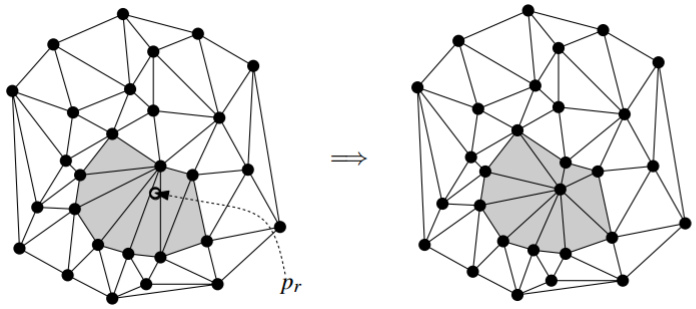
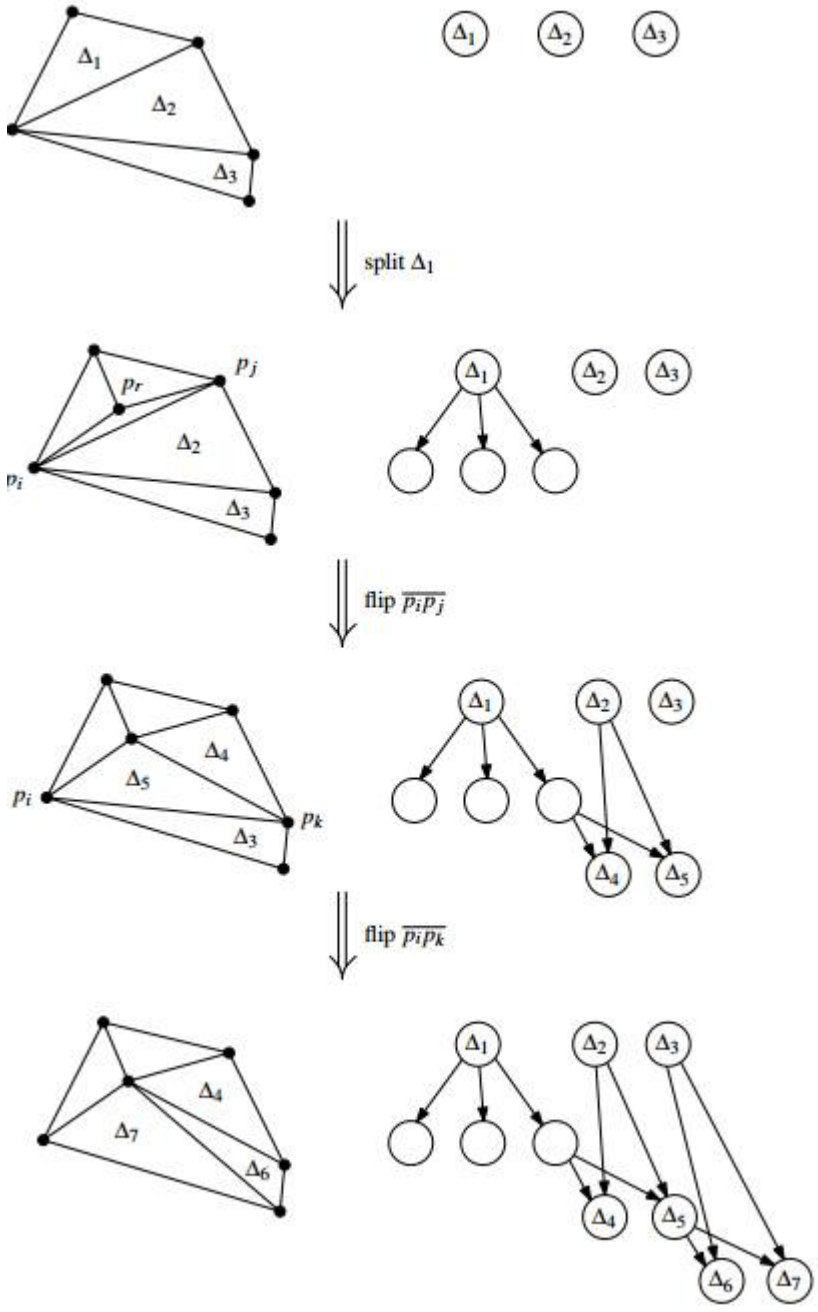


Figure 9.8
All edges created are incident to p_r

3.2.2 Locating the triangle using a Data Struture

On the next page we see a datastrucutre. Every node denotes the area the corresponding triangle covers. The node will always cover this area - even if the triangle is destroyed. We see that $\triangle_1$ is divided in 3 and now points to 3 (unnamed triangle), where we will call the third one $\triangle_c$.
Next, due to edge flipping, $\triangle_2$ is disbanded, and two new triangles ermeges - $\triangle_3$ and $\triangle_4$. These two triangles are both placed upon areas denoted by nodes of $\triangle_c$ and $\triangle_2$. So both $\triangle_c$ and $\triangle_2$ gets pointers to $\triangle_3$ and $\triangle_4$. Another edge flip occours and a similiar process will be left unexplained.

The data structure is used, so whenever a point is added, we will in constant time see, which of the 3 triangles (potentially former) triangles in lies in. We continue this approach all the way down. When a leaf is reached, we know that this triangle still exists and therefore is the triangle our point lies in. The data structure used $O(n)$ space and has an expected height of $O_e(lg(n))$


Δ1
Δ2
Δ3
split Δ1
pr
pj
pi
flip $\overline{p_i p_j}$
Δ4
Δ5
pk
flip $\overline{p_i p_k}$
Δ7
Δ6

4 Maximum Flow

4.1 Definition of the problem

We have a flow network, which is a directed graph $G(V, E)$ - every edge (u, v) has a non-negative capacity $c(u, v)$. We have a source vertex, s, and a sink vertex t. A flow must have the two properties

- **Capacity constraint** - For all $u, v \in V$ we require $0 \leq f(u, v) \leq c(u, v)$. Flows between two connected vertices cannot be negative and must not exceed the capacity.
- **Flow conservation** - For all $u \in V - s, t$, we require

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

  The amount of flow entering a vertex must equal the flow going out - except for the source and sink vertices s and t.

Having a flow f we define the value of the flow $|f|$ as,

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

That is, the total flow out of the source minus the flow into the source. In the maximum-flow problem, we want to find a flow of maximum value.

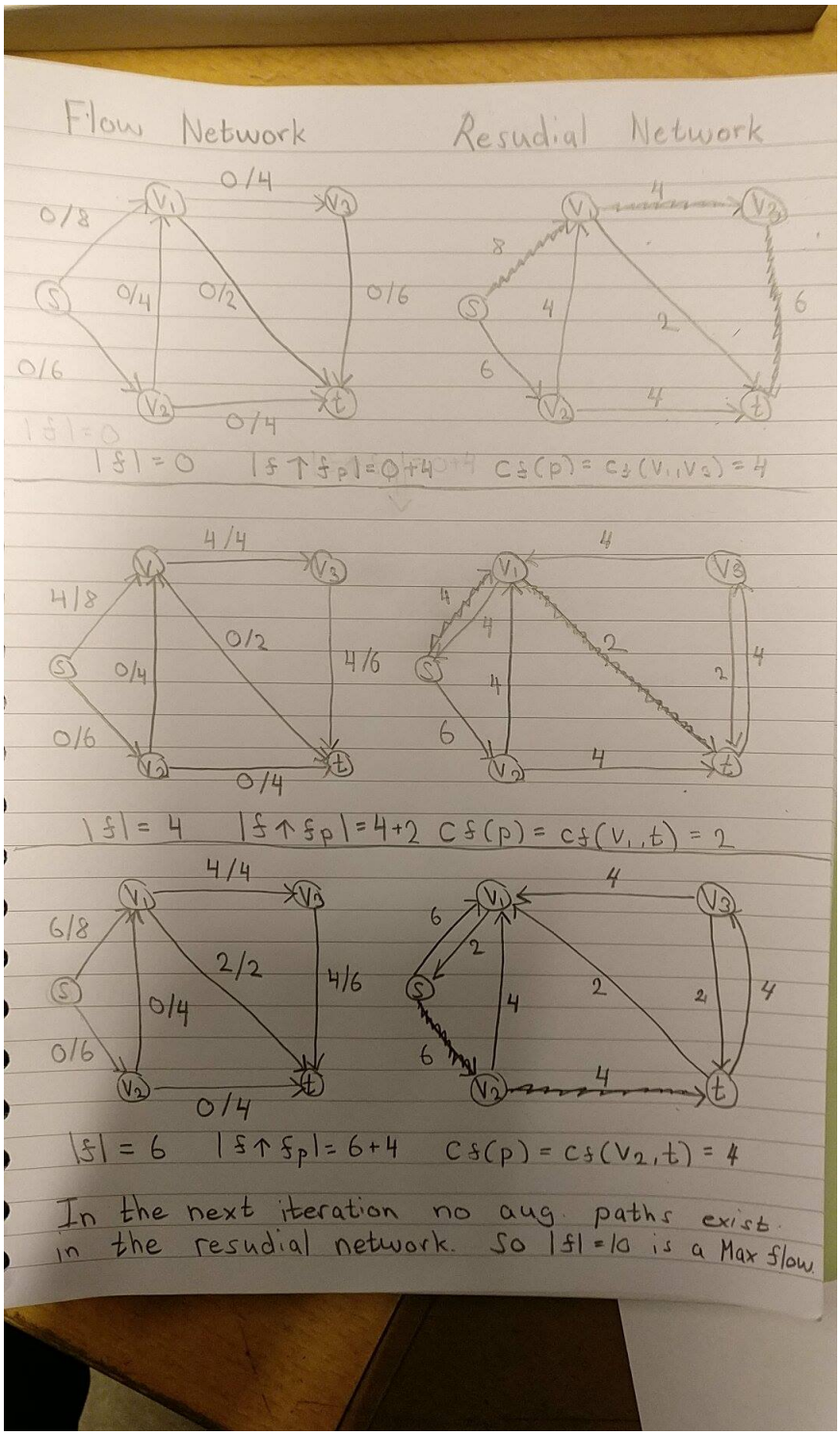
4.2 The Ford-Fulkerson method

```
f = 0
while (exist an augmenting path p from s to t) then
   augment f with a max flow along p
output f
```

For every iteration in the while loop we create a residual network $G_f(V, E_f)$, which purpose is to help us find an augmenting path and the value of the associated flow. The capacities in the residential network is defined such that,

- if $(u, v) \in E$ then $c_f(u, v) = c(u, v) - f(u, v)$
- if $(v, u) \in E$ then $c_f(u, v) = f(v, u)$
- otherwise $c_f(u, v) = 0$

We add an edge (u, v) to E_f whenever the corresponding capacity $c_f(u, v)$is not zero. Below we see three iterations of the Ford-Fukerson method.


Flow Network
Resudial Network
0/4
0/8
0/4
0/2
0/6
0/6
0/4
|f| = 0
|f| = 0 |f ↑ fp| = 0+4 Cf(p) = Cf(V1,V2) = 4
4/4
4/8
0/2
4/6
0/4
0/6
0/4
|f| = 4 |f ↑ fp| = 4+2 Cf(p) = Cf(V1,t) = 2
4/4
6/8
2/2
4/6
0/4
0/6
0/4
|f| = 6 |f ↑ fp| = 6+4 Cf(p) = Cf(V2,t) = 4
In the next iteration no aug paths exist.
in the resudial network. So |f| = 10 is a Max flow.

4.3 Main Proof

Given flow f in G_f the following three statements are equivalent.

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

We show it by (1) $\Rightarrow$ (2) and (2) $\Rightarrow$ (3) and (3) $\Rightarrow$ (1)

(1) implies (2)

Assume for contradiction that f is a max flow.
However if an augmenting path is in G_f, then $|f \uparrow f_p| > |f|$ and then f cannot be a max flow. Contradiction.

(2) implies (3) (hard one)

We assume that there is not augmenting path in G_f.
We define $S = \{v \in V | \exists \text{ path } s \to v \in G_f\}$ and $T = V \setminus S$. In other words S is the set of vertices reachable by s in G_f and T is the remaining vertices.
Clearly $s \in S$ - and $t \in T$, since we have no augmenting paths.

We need to show that $f(S,T) = c(S,T)$ (since $f(S,T) = |f|$)
Consider a pair of vertices $(u,v) \in S \times T$ - then 3 cases are possible:

Case 1 - $(u,v) \in E$
$C_f = 0$ since otherwise $\exists$path $s \to u \to v$ in G_f implying that $v \in S$
So $f(u,v) = c(u,v)$.
Case 2 - $(v,u) \in E$
$f(v,u) = 0$ - because a positive flow would cause an augmenting path from v to u, contradicting our initial assumption.
Case 3 - $(u,v) \notin E$ **and** $(v,u) \notin E$
If no edge is present, then $f(u,v) = f(v,u) = 0$.

$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - f(v,u) = c(S,T)$
Because we stablished that $f(v,u)$ is always 0, and $f(u,v) = c(u,v)$, when u and v is connected by an edge, and 0 otherwise.

(3) implies (1)

Let (S,T) be a cut such that $|f| = C(S,T)$. Then let f^* be a max flow in G. We then have $|f^*| \leq c(S,T) = |f|$, which indicates that $|f|$ is a max flow.

4.4 Time-Complexity with proofs

The Ford Fulkerson is a method, and not a direct algorithm. The time-complexity depends on our strategy of finding the augmenting paths.

We can find augmenting paths in the residual graph by either using breadth first or depth first - both techniques takes linear time $O(E)$.
If we assume all capacities in G are integers and we denote the max flow by $|f^*|$, then we can bound the time-complexity to $O(|f^*|E)$.
A singe iteration takes $O(E)$ time, and we do at most $|f^*|$ iterations, because we increment the current flow by at least 1 for every iteration.

Edmonds-karp algorithm with breadth-first search

The Edmonds-karps algorithm is simply using Ford Fulkersons method as descriped. When finding the augmenting path, we will however explicitly use the breadth-first search, which always ensure a shortest path in the resudial network.

Edmonds-karp has $O(VE^2)$, which we will prove below in two steps.

(1) - First we prove that the the shortest distance to every vertex in the residual network can never decrease.
Formally: **for every** $v \in V$**,** $d_{G_f}(s, v)$ **can never decrease**

Assume for contradiction that there exists a flow f at some point in the algorithm, where f' is the flow in the next iteration and $d_{G_{f'}}(s, v) < d_{G_f}(s, v)$ for some vertex $v \in V$, which is not s.

Consider a shortest path $s \to u \mapsto v$ in $G_{f'}$.
(Here $s \to u$ denotes some path from s to u, but v is coming directly after u) So $d_{G_{f'}}(s, v) = d_{G_{f'}}(s, u) + 1$. We want to show that $d_{G_{f'}}(s, u) \geq d_{G_f}(s, u)$ cannot be true by going through 2 following cases.

...

(2) - The total number of flow augmentations is $O(VE)$
An edge (u, v) is *critical* on an augmenting path if it is the weakest link, such that the flow through the path is equal to (u, v). On an augmenting path always one edge is critical.
We show that **each edge $e \in E$ can be critical at most $|V|/2$ times**.

Let u and v be vertices in V connected by an edge (u, v). Since augmenting paths are shortest paths, when (u, v) is critical the first time on flow f, then

$$\delta_f(s, v) = \delta_f(s, u) + 1$$

The edge was critical, so the capacity was filled, so the edge disapears in the residual network.
(u, v) only reappears in the residual network if (v, u) appears in an augmenting path. Lets say that happens on the flow f', then

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$$

(node that the direction has switched)
Now we use prove **(1)** and use that $\delta_f(s, v) \leq \delta_{f'}(s, v)$

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2$$

So the distance to u increases at least by 2 from whenever (u, v) becomes critical again. In the worst case the distance to u starts by being 0 and ends up being $|V| - 2$. (minus 2, since path $s \to u$ cannot be incremented by s, t, u)
Since the distance to it increases by at least two, the edge can at most be critical $(|V| - 2)/2 = |V|/2 - 1$ times more, after it became critical the first time yielding a totalt of $|V|$ times.
There is $O(E)$ pairs of vertices with an edge between them, where edge each is critical at mose $|V|$ times, and since every augmenting path has at least one critical edge, we can at most have $O(VE)$ augmenting paths.

The algorithm do at most $O(VE)$ augmenting paths and each path takes $O(E)$ time to find, so it yields a final time complexity of $O(VE^2)$

5 Fibonacci Heaps

A Fibonacci heap is a collection of rooted trees, each with the minimum heap property. There exists circular doubly-linked lists for each level of the trees of the forest, and also for the root nodes. A pointer, H.min, exists, pointing to the minimum root node.

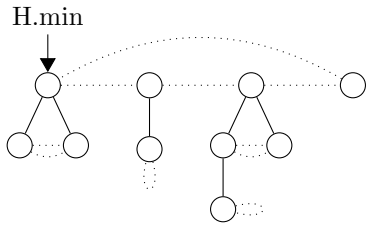


Figure 2: An example of a Fibonacci Heap data structure. The dotted lines represent pointers in each direction, while the solid lines represent pointers in one direction.

5.1 Maximum Degree

In the amortized analyses we perform, we assume that we know an upper bound $D(n)$ of the degree of any node in an m-node Fibonacci heap.

This wont be proven, but the bound is $D(n) = O(\lg n)$.

5.2 Potential function for amortized analysis

In order to perform an amortized analysis of the Fibonacci heap, we make use of the potential method, in which we store a "potential" in the data structure that can be used to outweigh later operations.

To do this, we need to define a potential function $\Phi(H)$ for a Fibonacci heap H, that defines the potential for that heap.

- Given a Fibonacci heap H, we denote the number of trees in the heap as $t(H)$ and the number of marked nodes as $m(H)$. We can then define the potential of the heap H as

$$\Phi(H) = t(H) + 2m(H). \tag{5.1}$$

- We assume that a unit of potential can 'pay' for a constant amount of work on the heap, where the constant is suitably big to be able to cover for any of the constant-time pieces of work that we can come upon.

5.3 Analysis

Let H_i be the heap after the ith update, with $i = 1, \cdots, t$. Associate $\Phi(H_i)$ to each H_i such that

$$\Phi(H_i) \geq \Phi(H_0) \text{ for } i = 0, \ldots, t.$$

Let c_i be the actual cost of the ith operation. Define the *amortized* cost of the ith operation

$$\hat{c}_i = c_i \Phi(H_i) - \Phi(H_0).$$

5.4 Insert

Insertion $\Theta(1)$: In the following, consider the ith update and let $H = H_{i-1}$ and $H' = H_i$.

INSERT(H, x): Add x to the root list, and update H.min accordingly.

Actual cost $c_i = O(1)$.

$$\begin{aligned} \Phi(H) &= t(H) + 2m(H) \\ \Phi(H') &= (t(H) + 1) + 2m(H)) \\ \hat{c}_i &= O(1) + \Phi(H') - \Phi(H) \\ &= O(1) + 1 \end{aligned}$$

5.5 Decrease-Key

5.6 Extract-Min

1. First all children are removed from the minimum node.
2. These children are put in the "root list" of the heap.
3. The minimum node is removed.
4. In the end the CONSOLIDATE function is run on the "root list", until there is at most *one* root of each degree.

Extract-Min(H), $\Theta(n)$**:**

$$\begin{aligned}
\Phi(H) &= t(H) + 2m(H) \\
\Phi(H') &= (D(n) + 1) + 2m(H) \\
\hat{c}_i &= O(D(n) + t(H) + \Phi(H') - \Phi(H) \\
&= O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\
&= O(D(n)) + O(t(H)) - t(H) \\
&= O(D(n))
\end{aligned}$$

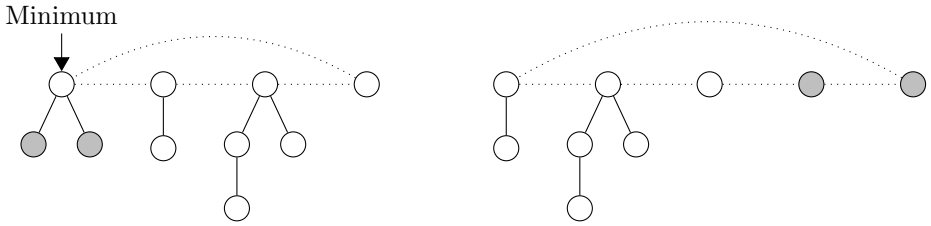


Figure 3: Initial steps of ExtractMin

6 NP-Completeness

6.1 The complexity class NP

The complexity class NP is the class of languages that can be verified with a polynomial time algorithm A.

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

What we know:

- P $\subseteq$ NP $\cap$ co-NP

What we don't know:

1. P = NP - probably not!
2. NP = co-NP,

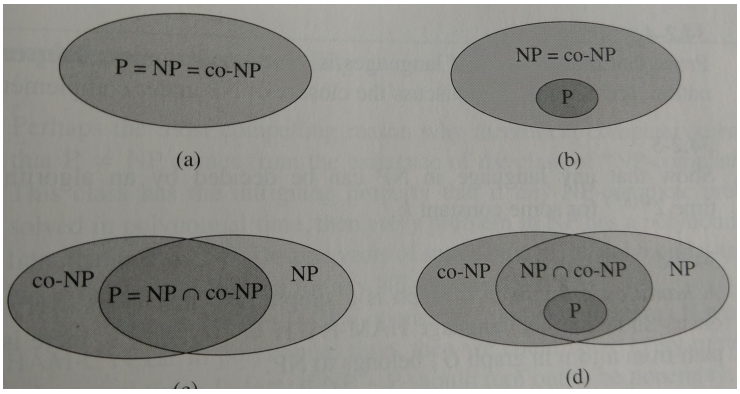


Figure 4: NP complexity class relations

6.2 Polynomial time reduction

We say that a language L_1 is polynomial time reducible if there exist a language L_2, written $L_1 \leq L_2$[1] if there exist a polynomial time computable function $f : \{0,1\}* \to \{0,1\}*$ such that for all $x \in \{0,1\}*$, $x \in L_1$ iff $f(x) \in L_2$.

[1] $\leq$ can be read as, to determine if $(x) \in L_2$ is at least as hard to determine if $x \in L_1$

6.3 Definition

A language $L \subseteq \{0,1\}*$ is NP-complete if:

1. $L \in NP$, and
2. $L' \leq_p L$ for every $L' \in NP$

If a language L only satisfies property 2, but not neccesarily property 1, we say that L is NP-hard. We define NPC to be the class of NP-complete problems.

6.3.1 Proof: If any NP-complete problem is polynomial time solveable then $P = NP$

We first introduce and prove the following lemma:
Lemma P
If $L_1, L_2 \subseteq \{0,1\}*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P$ implies $L_1 \in P$.

Proof
Let A_2 be a polynomial-time algorithm that decides L_2, and let F be a polynomial time reduction algorithm that computes the reduction function f. We shall construct a polynomial time algorithm A_1 that decides L_1.

For a given $x \in \{0,1\}*$, algorithm A_1 uses F to transform x into $f(x)$, and then uses A_2 to test whether $f(x) \in L_2$. Algorithm A_1 takes the output from A_2, and uses it as its own output. Since both F and A_2 runs in polynomial time, A_1 also runs in polynomial time.

Lemma NP
If any NP-complete is polynomial-time solvable, then $P = NP$. And if any problem in NP is NOT polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Proof of the first statement
Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \leq_p L$ by definition. Thus from **lemma P**, we also have $L' \in P$.

Proof of the second statement
Suppose that $L' \in NP$ and L' is NOT polynomial-time solvable. Then by point 2 ($L' \leq_p L$ for every $L' \in NP$) in the definition of NP-completeness and **lemma P** no NP-complete problem is polynomial-time solvable.

6.4 Proving NP-completeness by reduction

Lemma
If L is a language such that $L' \leq_p L$ for some $L' \in NPC$, then L is NP-hard. If, in addition, $L \in NP$, then $L \in NPC$.

Proof
Since L' is NP-complete, for all $L'' \in NP$, we have $L'' \leq_p L'$. By supposition, $L' \leq_p L$, and thus by transitivity, we have $L'' \leq_p L$, which shows that L is NP-hard. If in addition $L \in NP$, then by definition we have $L \in NPC$.

Thus the step to prove by reduction from an NP-complete language L' that the langauge L is NP-complete is the following:

1. Prove $L \in NP$
2. Select a known NP-complete language L'
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}*$ of L' to and instance $f(x)$ of L
4. Prove that the function f satisfies $x \in L'$ iff $f(x) \in L$ for all $x \in \{0,1\}*$
5. Prove that the algorithm computing f runs in polynomial time.

6.5 Reduction examples

6.5.1 SAT is NP-complete

The SAT problem is: Given a boolean formula Q, is Q satisfiable for some value assignments of the variables.

SAT is in NP We can construct a verifier A, that verifies $x \in SAT$ with a certificate y consisting of value assignments for each variable in x. Algorithm A then substitutes every variable in x with the corresponding value, and then evaluates the expression.

SAT is NP-hard We will use the fact that CIRCUIT-SAT problem is NP-complete, by reducing CIRCUIT-SAT to SAT, i.e. $CIRCUIT - SAT \leq_p SAT$.

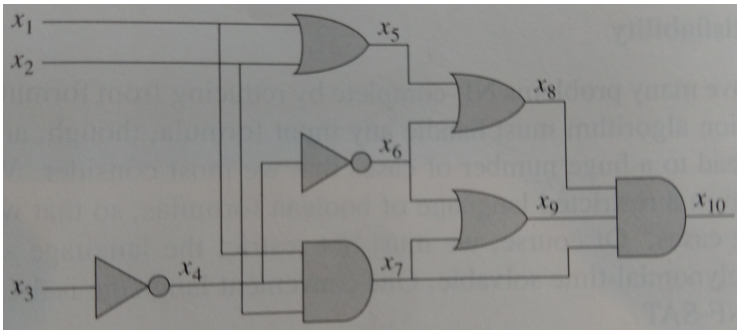


Figure 5: Diagram of a circuit

To reduce CIRCUIT-SAT to SAT, we assign a variable x_i to each wire. For each wire we construct a sub-formula that correspond to the truth condition of the wire. Example from the diagram in Figure 5:

$$\begin{aligned}\phi = x_{10} &\wedge (x_5 \leftrightarrow x_1 \vee x_2) \\ &\wedge (x_4 \leftrightarrow \neg x_3) \\ &\wedge (x_6 \leftrightarrow \neg x_4) \\ &\wedge (x_7 \leftrightarrow (x_1 \wedge x_4 \wedge x_2)) \\ &\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ &\wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ &\wedge (x_10 \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\end{aligned}$$

since we can make the following transformation by looping through every gate g, and for every gate consider every wire w connected to the gate, we can make the transformation in $O(gw)$ time, which is polynomial.

When the circuit have a satisfying assignment, then each wire have a well-defined value and the output is 1. Which means that if we assign the wire values to each variable in ϕ , each clause will evaluate to one, since all clauses are in conjunction ϕ wille evaluate to 1.

7 Exact Exponential Algorithms

Exact Expeonential Algorithms belong to a paradigm, which is applied to problems that have no known polynomial time solution, in particular, problems in NP, and must use exponential time in order to find a solution.

7.1 Travelling Salesman Problem (TSP)

The problem: Given a set of n cities, and for every pair of cities (c_i, c_j) a distance $d(c_i, c_j)$, find the shortest route, starting in a city c_1, passing through all cities exactly once, and ending in the starting city c_1.

A naive solution: Assume c_1 to be the starting city. Compute all permutations of the cities, and return the permutation with the lowest cost. This would result in a time complexity of $\Omega(n!)$.

Using dynamic programming: Compute for every pair (S, c_i), where $S = \{c_2, c_3, \ldots, c_n\}$ and $c_i \in S$, a value $\mathrm{OPT}[S, c_i]$, which is the minimal length of a tour, which starts in c_1, goes through all cities in S exactly once and ends in c_i.

For $|S| > 1$ we define

$$\mathrm{OPT}[S, c_i] = \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_j] + d(c_j, c_i) \text{ where } c_j \in S \setminus \{c_i\}\}.$$

With this, we can define the optimal tour in S ending in c_i as

$$\mathrm{OPT}[S, c_1] = \mathrm{OPT}[S \setminus \{c_i\}, c_j] + d(c_i, c_j).$$

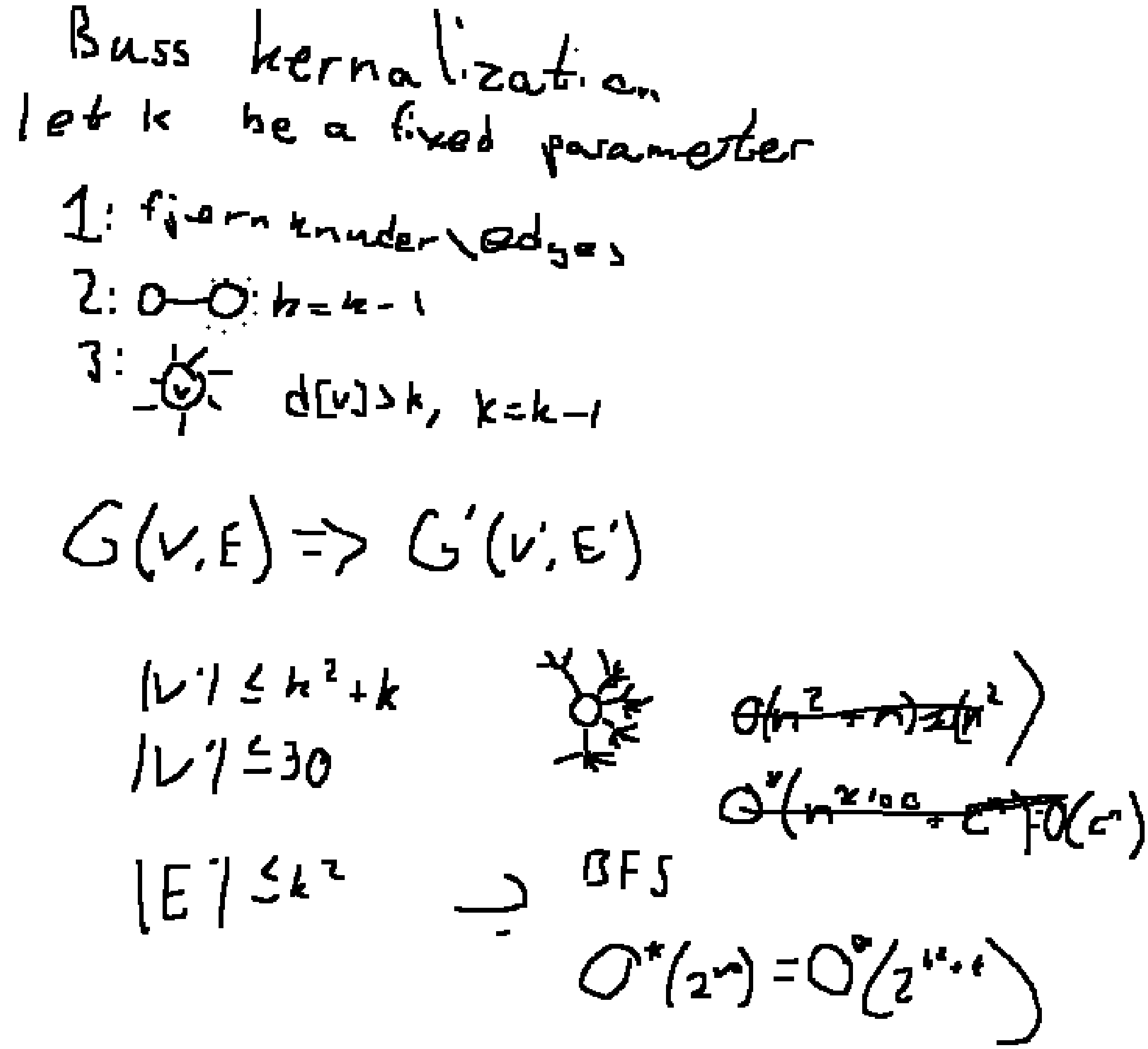
Finding the minimum over all $\{2, 3, \ldots, n\}$, we can establish the following algorithm.

1. Compute $\mathrm{OPT}[S', c_i]$ for all S' with $|S'| = 2$.
2. Repeat step 1, with increasing cardinality subsets until $S' = S$.
3. Return $\min\{\mathrm{OPT}[\{c_2, c_3, \ldots, c_n\}, c_i] + d(c_i, c_j)\}$ where $i \in \{2, 3, \ldots, n\}$

Analysis of time complexity:

7.2 Vertex Cover

There exists a algorithm with k-parameterization, some hard notes:



8 Approximation Algorithms

We say that a randomized algorithm for a problem has an approximation ratio of $\rho(n)$ if, for any input of size n, the *expected* cost C of a solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

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

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-approximation algoroithm. These notions apply to both cost-minimization and cost-maximization problems.

When we solve a maximization-problem we get that the approximation ratio is ≤ 1, since $\frac{C}{C^*} \leq 1$, and when we solve a minimization-problem we get the approximation ratio is $\frac{C^*}{C} \geq 1$.

8.1 MAX-3-CNF

(In this algorithm the cost C is our clauses, and we want to *maximize* the number of clauses to be satisfied, we have 2^3, 8 possible clauses total).

While a SAT-Solver solves whether a given formula Φ is satisfiable the

`MAX-3-CNF` or `MAX-3SAT` is a randomized approximate algorithm that solves the maximization problem:

> Given a 3-CNF formula Φ (with at most 3 variables per clause), finds an assignment that satisfies the largest number of clauses.

This decision version of the `MAX-3-CNF` algorithm is NP-Complete. If we were to satisfy **all** clauses in a given formula Φ it'd be NP-Hard. Thus we propose an algorithm which satisfies $\geq \frac{7}{8}$ of clauses. This is a

$$\max\left(\frac{7}{8}, \frac{8}{7}\right) \text{ thus a: } \frac{8}{7} - \text{Approximation algorithm.}$$

8.1.1 Algorithm

The `MAX-3-CNF` algorithm contains:

Definitions & Assumptions:
a 3-CNF formula in (**Conjunctive normal form**),
n boolean variables: $x_1, \cdots, x_n \in \{\text{TRUE, FALSE}\}$,
literals: variables and negation of variables: $x_1, \cdots, x_n, \neg x_1, \cdots, \neg x_n$.
clauses: each clause is a **disjunction ('OR')** of 3 different literals
e.g. $(x_1 \vee \neg x_2 \vee \neg x_3)$,
formula: **conjunction ('AND')** of m clauses,
a deciding whether a 3-CNF formula is satisfiable or NP-Hard.

Example:
$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1, \vee \neg x_2 \vee x_3)$ is satisfiable with the assignment: $\{x_1 = TRUE, x_2 = FALSE, x_3 = TRUE\}$, since the first clause is satisfied by x_1, and the second clause is satisfied by $\neg x_2$.

Example of a CNF-formula not satisfiable: $(x_1) \wedge (\neg x_1)$.

Example of a 3-CNF-formula not satisfiable: $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$.

The algorithm:

For each variable x, let x = TRUE with probability $\frac{1}{2}$ and x = FALSE with probability $\frac{1}{2}$, independently.

Proof:
Suppose that we have independently set each variable to 1 with probability $\frac{1}{2}$ and to 0 with probability $\frac{1}{2}$. For $i = 1, 2 \cdots, n$, we define the indicator random variable

$$Y_i = \begin{cases} 1, \text{if i'th clause is satisfied} \\ 0, \text{if i'th clause is not satisfied} \end{cases} \tag{8.1}$$

A clause is not satisfied only if all three of its literals are set to 0, and thus $\Pr[Y_i = 0] = \frac{1}{2}^3 = \frac{1}{8}$, and thus we must have that the probability of the i'th clause to be satisfied is $\Pr[Y_i = 1] = 1 - \Pr[Y_i = 0] = \frac{7}{8}$.
Since $Pr[i] = 1 * Pr[i] + 0 * Pr[\neg A] = E[I[i]] = E[Y_i]$, we have that $E[Y_i] = \frac{7}{8}$.

Now let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Y_m$, now we have:

$$E[Y] = E[\sum_{i=1}^{m} Y_i] = \sum_{i=1}^{m} E[Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7*m}{8}$$

thus we have prooved that we have an approximation ratio of $\frac{m}{7*m/8}$ at most, hence we have an $\frac{8}{7}$ approximation algorithm.

8.2 APPROX-VERTEX-COVER

The vertex-cover problem is proven to be NP-Complete. A **vertex-cover** of an undirected graph $G = (V, E)$ is a subset $V' \subset V$ such that if (u, v) is an edge in G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover thus is the number of vertices in it.

The **vertex-cover-problem** is to find a vertex cover of minimum size in a given undirected graph, thus it is a minimization algorithm with approximation ratio of

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

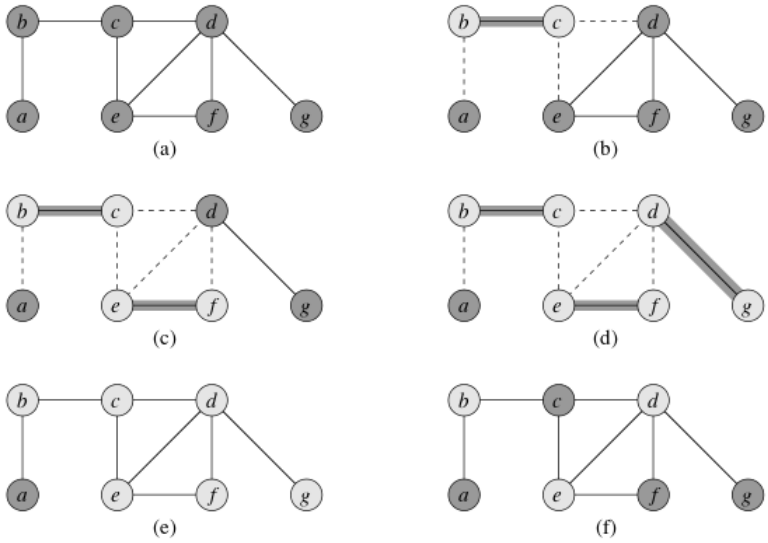
where C^* is the optimal vertex cover. This problem is an optimization-version of the NP-complete decision problem.

The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

8.2.1 Algorithm

```
APPROX-VERTEX-COVER(G)
1  C ← ∅
2  E' ← E[G]
3  while E' ≠ ∅
4       do let (u, v) be an arbitrary edge of E'
5          C ← C ∪ {u, v}
6          remove from E' every edge incident on either u or v
7  return C
```

Example:



(a): The input graph.
(b): The edge (b, c) is picked arbitrarily, vertices b and c is added to the set C containing the vertex cover being created. Edges (b, a), (c, e), (c, d) are removed from E' since they are already covered by some vertex in set C.
(c): The edge (e, f) is chosen, thus the vertices e and f are added to C. Edges (e, d), (f, d) are removed from E'.
(d): Edge (d, g) is picked.
(e): The set C, which is the vertex cover produced by the algorithm, contains the six vertices {b,c,d,e,f,g}.
(f): This picture shows the optimal vertex cover {b,d,e} but keep in mind, our algorithm assured that the size of our approximated vertex cover is no more than twice the amount of vertices contained in the optimal vertex cover.

The running time of the algorithm is O(V + E) using an adjacency list to represent E'.

Proof:
The set C of vertices that is returned by `APPROX-VERTEX-COVER` is a vertex cover, since the algorithm loops until every edge in E[G] has been covered by some vertex in C.

Now lets proove that the vertex cover the approximation algorithm returns is at most, twice the size of the optimal solution.
Let A be the set of vertices that were arbitrarily picked (line 4). In order to cover the edges in A, any vertex cover, in particular an optimal cover C^*, must include at least one endpoint of each edge in A.
No two edges in A share an endpoint, since once an edge is picked, all other edges that are incident on its endpoints are deleted from E'.
Thus, no two edges in A are covered by the same vertex from C^*, and since each edge (u, v) in A must have either u or v $\in C^*$, we have the lower bound:

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

on the size of the optimal vertex cover. Each edge picked by our algorithm is an edge which neither of its endpoints is already in C, thus we have the upper bound on the size of the vertex cover returned:

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

and thus we have that:

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

prooving that the size of C, the vertex cover returned by the approximation algorithm is no more than twice the sice of the optimal vertex cover.