AAD noter

Mia Rahlff Pedersen

November 2024

Contents

1 Max Flows **3**
1.1 Flow Networks . . . 3
1.1.1 Definitions . . . 3
1.2 The Ford-Fulkerson method . . . 3
1.2.1 Residual networks . . . 3
1.2.2 Augmenting paths . . . 4
1.2.3 Cuts of flow networks . . . 4
1.3 The Edmonds-Karp algorithm . . . 5
1.3.1 Example . . . 5

2 Linear programming **6**
2.1 Standard form . . . 6
2.1.1 Converting into Standard form . . . 7
2.2 Slack form . . . 8
2.2.1 Converting into slackform . . . 8
2.3 The simplex method . . . 8
2.3.1 Example . . . 9
2.4 Duality . . . 11
2.4.1 Example . . . 11

3 Randomized algorithms **12**
3.1 Randomized quicksort . . . 12
3.1.1 Lucky and unlucky cases . . . 12
3.1.2 Average case analysis . . . 12
3.2 Randomized minimum cut . . . 13

4 Hashing **13**
4.1 Applications . . . 15
4.1.1 Unordered sets . . . 15
4.1.2 Hashing with chaining . . . 15
4.1.3 Signatures . . . 16
4.2 Multiply-mod-prime . . . 16
4.3 Multiply-shift . . . 17

5 Computational complexity, P, NP, and NP-completeness 17
5.1 Computational problem . . . 17
5.2 Decision problem and languages . . . 17
5.2.1 Verifying a Language . . . 18
5.3 Turing machine . . . 18
5.3.1 Halting problem . . . 19
5.4 Polynomial-time reducability . . . 19
5.5 NP-Complete languages . . . 19
5.5.1 The SAT problem . . . 19
5.5.2 3-CNF . . . 20
5.6 NP-Complete problems . . . 21
5.6.1 The Clique problem . . . 21
5.6.2 The Vertex cover problem . . . 21
5.6.3 The Traveling salesman problem . . . 21
5.6.4 The Ham-cycle problem . . . 21
5.6.5 The subset-sum problem . . . 21

6 Exact exponential algorithms and parameterized complexity 21
6.1 Size of a problem . . . 22

7 van Emde Boas Trees 22
7.1 Preliminary approaches . . . 22
7.1.1 Direct approach . . . 22
7.1.2 Superimposing a tree of constant height/ . . . 22
7.2 Recurse . . . 23
7.2.1 Fix . . . 24
7.2.2 Fix pt.2 . . . 24

8 Polygon triangulation 25
8.1 Triangulation . . . 25
8.2 Partitioning a Polygon into Monotone Pieces . . . 25
8.3 Triangulating a Monotone Polygon . . . 27

9 Approximation algorithms 27
9.1 Vertex-cover . . . 27
9.2 Traveling Salesman . . . 28
9.3 Set cover . . . 29
9.4 MAX-3-SAT . . . 30
9.5 Weighted Vertex Cover . . . 30
9.6 Approximation schemes . . . 32

10 Extra 32
10.1 Log regler . . . 32

1 Max Flows

In the maximum-flow problem, we wish to compute the greatest rate at which we can ship material from the source to the sink without violating any capacity constraints.

1.1 Flow Networks

- A flow network consists of a directed graph G = (V , E), in which every edge has a non-negativity capacity $c(u, v) \geq 0$
- If E contains an edge (u, v), then there is no edge (v, u) in the reverse direction
- If $(u, v) \notin E$ then $c(u, v) = 0$
- We disallow self loops
- For each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$ (s=source, v=vertex and t=sink)

1.1.1 Definitions

Capacity contraint:

$$\text{For all } u, v \in V, \text{ we require } 0 \leq f(u, v) \leq c(u, v)$$

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

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

The value of a flow:

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

Meaning the total flow out of the source minus the flow into the source.

1.2 The Ford-Fulkerson method

1.2.1 Residual networks

Given a flow network G and a flow f, the residual network G_f consists of edges with capacities that represent how we can change the flow on edges of G. An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge.

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

The residual capacities when having a network $G = (V, E)$ with source s and sink t, will be

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

A residual network has the same properties as a flow network, and we can define a flow in the residual network as one that satisfies the definition of a flow, but with respect to capacities cf in the network G_f .

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

If f is a flow in G and f' is a flow in the corresponding residual network G_f ,we define f ↑ f',th eaugmentation of flow f by f' as above.
Note: $f \uparrow f'$ can be read as $f + f'$

1.2.2 Augmenting paths

An Augmenting path p is a simple path from s to t in the residual network.
We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the residual capacity of p, given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$$

An augmenting path refers to a path in a flow network that can accommodate more flow, thereby increasing the total flow value from the source to the sink.

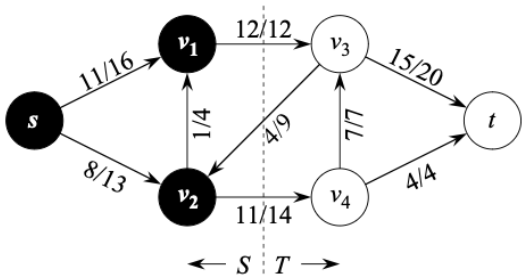
1.2.3 Cuts of flow networks

A flow is maximum if and only if its residual network contains no augmenting path.
If f is a flow, then the net flow $f(S,T)$ across the $cut(S,T)$ is defined to be

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

The capacity of the cut (S,T) is

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



A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network. For capacity, we count only the capacities of edges going from S to T , ignoring edges in the reverse direction. For flow, we consider the flow going from S to T minus the flow going in the reverse direction from T to S.Hence, the flow from **??** will be:

$$12 + 11 - 4 = 19$$

Becuase we consider the flow from S to T (v_1 to v_3 and v_2 to v_4) And also the flow from T to S (v_3 to v_2)
The capacity of the cut will be

$$12 + 14$$

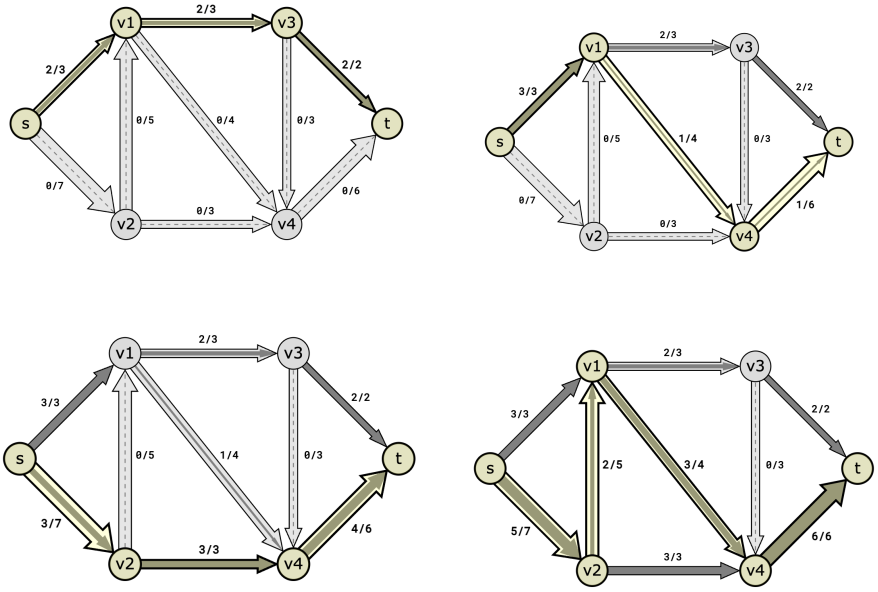
As we only consider the capacities of edges going from S to T.

1.3 The Edmonds-Karp algorithm

We can improve the bound on FORD-FULKERSON by finding the augmenting path p in line 3 with a breadth-first search.

1.3.1 Example

1. we start of by doing a BFS, find an augmented path where flow can be increased. With 2 being the bottleneck
2. we again start of by finding augmenting path, find how much the flow in that path can be increased, and increase the flow along the edges in that path accordingly. We can in this case up the flow by 1.
3. We find the next augmented path. The flow can be increased with 3. Bottleneck being $v_2 \rightarrow v_4$.
4. Last augmented path is found. the flow can be increased by 2.



2 Linear programming

A solution that satisies all constraints, is called a feasible solution.
The set of feasible solution, can be shown graphically and the convex region is called the feasible region.
The problem we wish to maximize is called the objective function.
We call the value of the objective function at a particular point the objective value.

2.1 Standard form

Standard form will be in the following form

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

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

$$x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n. \tag{3}$$

A standard form requires 3 different things

1. Must be a Maximization problem
2. It must have non-negativity constraints

3. Must only consist of less-than-or-equal-to, no equality contraints or greater-than-or-equal-to

.If a linear program has some feasible solutions but does not have a finite optimal objective value, we say that the linear program is unbounded

2.1.1 Converting into Standard form

To turn turn a linear program into standard form, we must have the 3 conditions above met.

1. We start out by turning a minimization problem into a maximization problem, by negating the coefficients in the objective function.

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

   To

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

2. We must then ensure that all variables have non-negativity constraints

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

3. We must then make the equality contraint into a less-than-or-equal-to, by making the constraint both a less-than-or-equal-to and greater-than-or-equal-to

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

4. Lastly we must remove the greater-than-or-equal-to, by negating the greater-than-or-equal-to and turning it into a less-than-or-equal-to

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

2.2 Slack form

The slack form is usefull for the simplex method. It must consist of all equality constraints, exept for the non-negativity contraint. We call the variables on the left-hand side of the equalities basic variables and those on the right-hand side nonbasic variables.

2.2.1 Converting into slackform

1. We start of with $\sum_{j=1}^{n} a_{ij}x_j \leq b_i.$, as a inequality, and introduce s (or x_{n+1} if multiple inequalities) and rewrite the inequality into two contraints

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_j, \quad s \geq 0.$$

2. We then ommit the maximize and subject to, as well as the explicit non-negativity constraints. We use z to denote the objective function.

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

   To

$$\begin{aligned}
&\text{z} = 2x_1 - 3x_2 + 3x_3 \\
&\quad x_4 = 7 - x_1 - x_2 + x_3, \\
&\quad x_5 = -7 + x_1 + x_2 - x_3, \\
&\quad x_6 = 4 - x_1 + 2x_2 - 2x_3
\end{aligned}$$

2.3 The simplex method

1. Convert into slack form

2. Set decision variables $(x_1, x_2, ..x_n)$ to zero and compute the values of slack variables

3. Pick the variable with the highest positive coefficient in the objective function.

4. Compute how much the entering variable can increase before a slack variable becomes zero. The smallest bound determines the leaving variable.

5. Solve for the entering variable in terms of the leaving variable, and Substitute into other equations to update the slack form.

6. Repeat steps 3–5 until all coefficients in the objective function are negative or zero.

7. The values of basic variables give the solution. Slack variables show unused resources.

2.3.1 Example

We solve the following linear program:

$$\text{Maximize } Z = 3x_1 + x_2 + 2x_3$$

subject to:

$$x_1 + x_2 + 3x_3 \leq 30,$$
$$2x_1 + 2x_2 + 5x_3 \leq 24,$$
$$4x_1 + x_2 + 2x_3 \leq 36,$$
$$x_1, x_2, x_3 \geq 0.$$

1. Introduce slack variables x_4, x_5, and x_6 to convert inequalities into equalities:

$$x_4 = 30 - x_1 - x_2 - 3x_3,$$
$$x_5 = 24 - 2x_1 - 2x_2 - 5x_3,$$
$$x_6 = 36 - 4x_1 - x_2 - 2x_3.$$

The objective function remains:

$$Z = 3x_1 + x_2 + 2x_3.$$

2. Set $x_1 = 0$, $x_2 = 0$, $x_3 = 0$. The values of the slack variables are:

$$x_4 = 30, \quad x_5 = 24, \quad x_6 = 36,$$
$$Z = 0.$$

The initial basic feasible solution is:

$$(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 0, 30, 24, 36).$$

3. From the objective function $Z = 3x_1 + x_2 + 2x_3$, choose the variable with the highest positive coefficient in Z: x_1 (coefficient = 3).

4. Compute how much x_1 can increase before a slack variable becomes zero:

$$x_4 = 30 - x_1 \quad \Longrightarrow \quad x_1 \leq 30,$$
$$x_5 = 24 - 2x_1 \quad \Longrightarrow \quad x_1 \leq 12,$$
$$x_6 = 36 - 4x_1 \quad \Longrightarrow \quad x_1 \leq 9.$$

   The smallest bound is $x_1 \leq 9$, so x_6 becomes zero. The leaving variable is x_6.

5. Solve for x_1 in the equation for x_6:

$$x_1 = 9 - \frac{1}{4}x_2 - \frac{1}{2}x_3 - \frac{1}{4}x_6.$$

   Substitute this into the other equations to update the slack form:

$$x_4 = 21 - 3x_2 - 5x_3 + x_6,$$
$$x_5 = 6 - 3x_2 - 4x_3 + x_6,$$
$$x_1 = 9 - \frac{1}{4}x_2 - \frac{1}{2}x_3 - \frac{1}{4}x_6,$$
$$Z = 27 + x_2 + x_3 - 3x_6.$$

   The basic variables are now x_1, x_4, and x_5.

6. From $Z = 27 + x_2 + x_3 - 3x_6$, choose x_3 (coefficient = 1).

   Compute how much x_3 can increase:

$$x_4 = 21 - 5x_3 \quad \Longrightarrow \quad x_3 \leq \frac{21}{5} = 4.2,$$
$$x_5 = 6 - 4x_3 \quad \Longrightarrow \quad x_3 \leq 1.5.$$

   The smallest bound is $x_3 \leq 1.5$, so x_5 becomes zero. The leaving variable is x_5.

7. Solve for x_3 in the equation for x_5:

$$x_3 = 1.5 - \frac{3}{4}x_2 + \frac{1}{4}x_6.$$

   Substitute this into the other equations to update the slack form:

$$x_4 = 18 - x_2 + x_5,$$
$$x_1 = 8 + x_2 - x_5,$$
$$x_3 = 1.5 - \frac{3}{4}x_2 + \frac{1}{4}x_5,$$
$$Z = 28 - x_2 - x_5.$$

   The basic variables are now x_1, x_3, and x_4.

8. In $Z = 28 - x_2 - x_5$, all coefficients of nonbasic variables (x_2 and x_5) are negative or zero. Thus, the current solution is optimal.

9. The solution is:

$$x_1 = 8, \quad x_2 = 4, \quad x_3 = 0, \quad x_4 = 18, \quad x_5 = 0, \quad x_6 = 0,$$

with objective value:

$$Z = 28.$$

2.4 Duality

Duality enables us to prove that a solution is indeed optimal.
When referring to dual linear programs, we call the original linear program the primal.
To form the dual, we change the maximization to a minimization, exchange the roles of coefficients on the right-hand sides and the objective function, and replace each less-than-or-equal-to by a greater-than-or-equal-to. Also you must consider it a matrix, and take the transpose.

2.4.1 Example

Take original

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

Into dual

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

We see that it is changed into a minimization problem, and the roles of the coefficients is changed. Also notice how the $x_1's$ become y_1, y_2 and y_3(transpose of original)

```
RandQS({s_1, ..., s_n})
  if n = 0:
    return [ ]
  choose random i ∈ {1, ..., n}
  L := {s_j | s_j ≤ s_i and i ≠ j}
  R := {s_j | s_j > s_i}
  return RandQS(L) + [s_i]+RandQS(R)
```

Figure 2: Randomized quicksort pseudo code

3 Randomized algorithms

3.1 Randomized quicksort

Input to the algorithm is a set of n numbers, if the set is empty(Equal to zero, then we return a empty set. Otherwise we choos a random index from 1 to n, and it is uniformly random(P=$\frac{1}{n}$), call the algorithm recursively on L and R, and then append them together with the pivot(randomly chosen n).
It is called a **Las Vegas algortihm** – a algorithm that always returns the correct answer.

3.1.1 Lucky and unlucky cases

Lucky case: s_i is always the median, then $|L| \leq \frac{n}{2}$ and $|R| \leq \frac{n}{2}$, then the running time will be $T(n) = O(n) + 2T(n/2) = O(nlogn)$ which can be further unfolded, see slides.

Unlucky case: s_i is always the minimum, then $|L| = n$ and $|R| = n - 1$, as it is all other elements. Running time will be $T(n) = \Omega(n) + T(n-1) = \Omega(n^2)$.

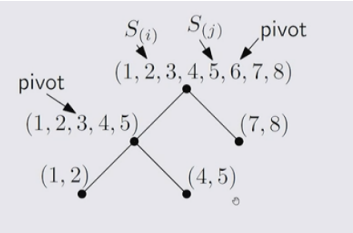
3.1.2 Average case analysis

$\mathbf{E}[X] = O(nlogn)$
Proof: Let $S_1, ..., S_n$ be the numbers in sorted order
Let X_{ij} :=#comparisons between $S_{(i)}$ and $S_{(j)} \in \{0, 1\}$, as if we are comparing two numbers then one must be a pivot and it will not be used again, it will go into the recursive call and not be compared to the pivot again?.

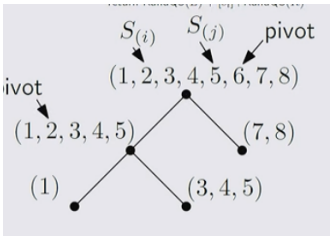
$$\mathbf{E}[X] = \mathbf{E}[\sum_{i=1}^{n} \sum_{j>i} X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} [X_{ij}]$$

Let $p_{ij} := \mathbf{Pr}[S_{(i)}$ and $S_{(j)}$ are compared]
Then $\mathbf{E}[X_{ij}] = 0 \cdot (1 - p_{ij}) + 1 \cdot p_{ij} = p_{ij}$

Oberservation:



(a) $S_{(i)}$ or $S_{(j)}$ not the first pivot among $\{S_{(i)}, ..., S_{(j)}\}$ and will therefore not be compared



(b) $S_{(i)}$ is the first pivot among $\{S_{(i)}, ..., S_{(j)}\}$ and therefore $S_{(i)}$ and $S_{(j)}$ will be compared

Figure 3: Randomized quicksort obervation

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

As the numbers are in sorted order, and will be compared if we choose one of the ends first, therefor there are two ways it can happen. 1) if we choose $S_{(i)}$, 2) if we choose $S_{(j)}$.

Giving us:

$$\mathbf{E}[X] = \mathbf{E}[\sum_{i=1}^{n}\sum_{j>i} X_{ij}] = \sum_{i=1}^{n}\sum_{j>i}[X_{ij}] = \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1}$$

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

$$2nO(logn) = O(nlogn)$$

3.2 Randomized minimum cut

Given: Graph $G = (V, E), |V| \geq 2$
Find: Min. set of edges $C \subseteq E$ s.t. G C is disconnected. **Def.:** $\lambda(G) := |C|$(*Edge connectivity* in G) Min-cut will be 4&1 and also 6&7 Algorithm will firstly choose c, d in the example graph(4), and so we will remove that edge(**??**). Now it chooses edge 1, merging a and e(**??**). Now choosing edge 3, merging a, b, e, giving us a self-loop(**??**) which we will remove(**??**). So we will now return edge {4,5,7}, as we only have 2 vertices left.

4 Hashing

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

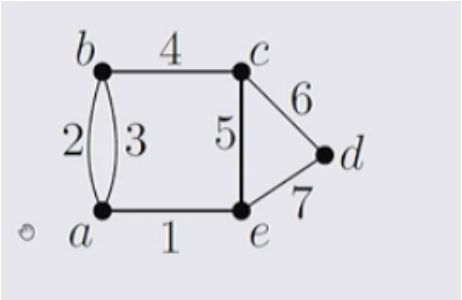


Figure 4: Randomized mincut example

```
RandMinCut(G)
  if G not connected
    return ∅
  while |V(G)| > 2
    contract random edge e ∈ E(G)
    remove self-loops
  return E(G)
```

Figure 5: Randomized min-cut pseudo code

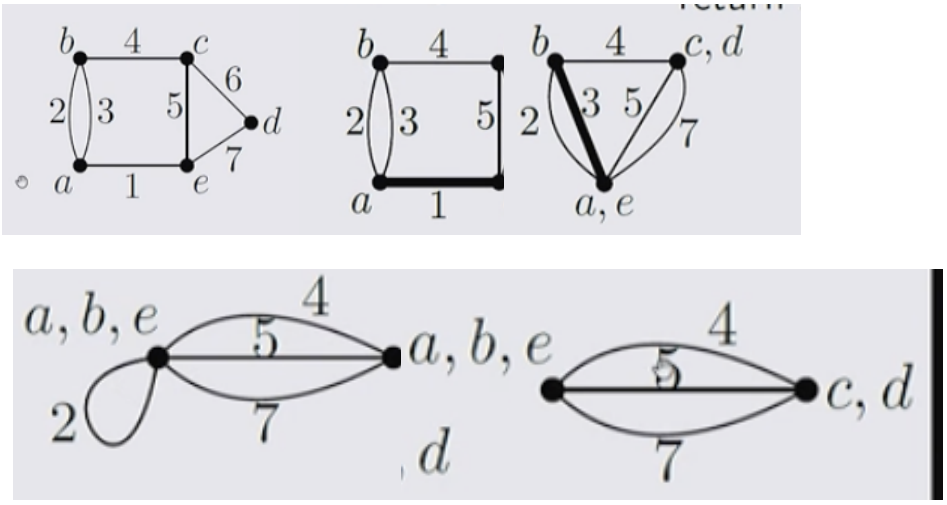


Figure 6: Mincut algorithm example

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

The total number of possible functions is $m^{|U|}$(m to the size of the universe), meaning it reuires $|U|log_2 m$ bits to represent.
Universal: a random hash function $h : U \to [m]$ is universal if, for all $x \neq y \in U : \Pr[h(x) = h(y)] \leq \frac{1}{m}$, meaning that the probability that two distinct values are the same is equal to at most $1/m$.
c-approximately universal:a random hash function $h : U \to [m]$ is c-approximately universal if, for all $x \neq y \in U : \Pr[h(x) = h(y)] \leq \frac{c}{m}$, for some constant c.
Strongly universal: a random hash function $h : U \to [m]$ is strongly universal if for all $x \neq y \in U$, and $q, r \in [m] : \Pr[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}$, meaning for two distinct variables in the universe and two elemts we are hashing into, the probability that x hashes to q and y hashes to r is exactly $1/m^2$. Equvalently:

- Each key is hashed uniformly into $[m]$, means for any x and q the probability that x hashes to q is $1/m^2$ and each key hash independantly
- Any two distinct keys hash independently

c-approximately strongly universal: a random hash function $h : U \to [m]$ is c-approximately strongly universal if for all $x \neq y \in U$, and $q, r \in [m] : \Pr[h(x) = q \wedge h(y) = r] = \frac{c}{m^2}$, meaning for two distinct variables in the universe and two elemts we are hashing into, the probability that x hashes to q and y hashes to r is exactly $1/m^2$. Equvalently:

- Each key is hashed c-approximately uniformly into $[m]$, means for any x and q the probability that x hashes to q is $1/m^2$ and each key hash independantly
- Any two distinct keys hash independently

4.1 Applications

4.1.1 Unordered sets

Maintain a set S of at most n keys from some unorderes universe U, under

- $Insert(x, S)$ Insert key x into S
- $Delete(x, S)$ Delete key x from S
- $Member(x, S)$ Return $x \in S$

This couldbe stored in some sort of balanced tree to store S, but each operation would usually take $o(log\ n)$.

4.1.2 Hashing with chaining

Idea is to pick $m \geq n$ and a $universalh : U \to [m]$. Store array L, where

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

Then $x \in S \Leftrightarrow x \in L[h(x)]$
Theorem

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

Proof

$$\begin{aligned}
\mathbf{E}[|L[h(x)]|] &= \mathbf{E}[I\{y \in S | h(y) = h(x)\}|] \\
&= \mathbf{E}[\sum_{y \in S}[h(x) = h(y)]] \\
&= \sum_{y \in S} \mathbf{E}[h(x) = h(y)] \\
&= \sum_{y \in S} Pr[h(x) = h(y)] \\
&\leq |S| \frac{1}{m} = \frac{n}{m} \leq 1
\end{aligned}$$

as n is a "chosen" upperbound.

4.1.3 Signatures

Assign a unique "signature" to each $x \in S \subseteq U, \quad |S| = n$
Solution: Use universal hash function $s : U \to [n^3]$. Then by union bound

$$\begin{aligned}
Pr[\exists\{x, y\} \subseteq S | s(x) = s(y)] &\leq \sum_{\{x,y\} \subseteq S} Pr[s(x) = s(y)] \\
&\leq \frac{\binom{n}{2}}{n^3} \\
&< \frac{1}{2n}
\end{aligned}$$

Thus we with high probability have no collisions.
The reason for $\binom{n}{2} = \frac{n(n-1)}{2}$, is that the possibility of choosing two The number of ways to choose 2 elements from n elements is a binomial coefficient.

4.2 Multiply-mod-prime

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

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

This is NOT a random hash function
Choose $a, b \in [p]$ independently and uniformly at random and let $h(x) := h^m_{a,b}(x)$
Then $h : U \to [m]$ is a 2-approximately strongly universal hash function.

4.3 Multiply-shift

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

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

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

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

P: Problems we can solve in polynomial time
NP: Problems we can verufy in polynomial time.

5.1 Computational problem

Represent objects as strings in some alphabet Σ
Σ Can be

- 0 or 1
- a-z,A-Z,0-9

or something else, chosen convieniently. As long as it is finite,
Sting: $s \in \Sigma^*$ - sequence of 0 or more characters from Σ

5.2 Decision problem and languages

Consider function $f : \Sigma^* \to \{0, 1\}$, where 0 ="no" and 1 ="yes". Examples of this could be:

- Is there a path in G from s to t?
- Is the given formula satisfiable?
- Is there a prime factor of N of size at most U?

An algorithm solves the problem in time $O(T(n))$, if for any instance of length n, the algorithm returns a solution (0 or 1) in time $O(T(n))$
If $T(n) = O(n^k)$ for som constant k, the problem is polynomial solvable
Language: $L \subseteq \Sigma^*$, $L = \{x \in \Sigma^* | f(x) = yes\}$

5.2.1 Verifying a Language

A verification algorithm is a algorithm A taking two arguments $x, y \in \{0,1\}^*$ where y is the certificate. A verifies a string x if there is a certificate y such that $A(x, y) = 1$.
THe language verified

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

5.3 Turing machine

Definition:

- Fixed alphabet Σ(Finite size)
- Program Q (Set of states of the Turing machine)
- Tapes
  - Input tape contains input, read-only
  - Work/Output tape

  Tapes have starting point/position, but no end

At each step of the turing machine:

1. Reads symbol at current position on all tapes
2. Writes symbol to work tapes
3. Move tape head left or right one step (or stand still)
4. Jump to new state $q \in Q$

Step 2-4 depend on the current state of q and the symbols read on tapes in step 1.
Special state: $q_{halt} \rightarrow TuringMachinestops$
Running time: *# steps before reching* q_{halt}
Example:
TM that decides whether #1 is in binary string odd

- **Always:** Read symbol s on input tape, move tape to the right
- q_{start}
  - if $S = 0$ go to q_{start}
  - if $S = 1$ got to q_{odd}
  - if $S = EOF$ write 0 on output and go to q_{halt}
- q_{odd}
  - if $S = 0$ go to q_{odd}
  - if $S = 1$ got to q_{start}
  - if $S = EOF$ write 1 on output and go to q_{halt}

5.3.1 Halting problem

Consider following language

$$\text{HALT} = \{\langle M, x\rangle | \text{Turing machine M halts on input x}\}$$

If M or x is not valid, then $\langle M, x\rangle$ is a no instance
Theorem: The language Halt is not decidable by any turing machine
Proof:Suppose that H is a TM, that decides halt. We can construct another TM $H\prime$ that simulates H as a subroutine, then we can feed $H\prime$ to H as a suitable input.

5.4 Polynomial-time reducability

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

$$x \in L_1 \leftrightarrow f(x) \in L_2$$

We write it as $L_1 \leq_p L_2$. And if that is the case, L_1 is in a sense no harder to solve than L_2, more precisely

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

5.5 NP-Complete languages

Language L is NP complete if

1. L$\in$ NP and
2. L$\prime \leq_P L$ for every $L \in NP$(For every problem in NP, there is a reduction in polynomial time from $L\prime$ to L.)

L is NP-Hard if property 2 holds

5.5.1 The SAT problem

A boolean formula ϕ consists of boolean variables $x_1, .., x_n$. Boolean connectives $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, parentheses(and)$
Example:$\phi = (x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$
A satisfiable assignment for a boolean formula ϕ is an assignment of 0/1 values to variables that make ϕ evaluate to 1
ϕ is satisfiable is there exists a sattisfiable assignment to ϕ(making it evaluate to 1)
Definition of SAT:

$$\texttt{SAT} = \{\langle\phi\rangle | \phi \text{ Is a satisfiable boolean formula}\}$$

To show that SAT $\in NPC$ we follow "recipe"

1. We start of by showing that $\texttt{SAT} \in NP$ by constructing a verification algorithm A taking inputs x and y.
   x is as boolean formula ϕ and y is the certificate. With A returning 1 if y defines a satisfiable assignment for ϕ, 0 otherwise. This can easily be done in polynomial time, making $\texttt{SAT} \in NP$

2. Showing $\texttt{Circut-SAT} \leq_p \texttt{SAT}$. Given a circut C, we tranform it into a boolean functionϕ by associating a variable x_i with each wire of C and let x_m be the output wire variable. Giving us, as an example:

$$\phi_1 = (x_4 \leftrightarrow \neg x_3)$$
$$\phi_2 = (x_5 \leftrightarrow (x_1 \vee x_2))$$
$$\phi_3 = (x_6 \leftrightarrow \neg x_4)$$
$$\phi_4 = (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\phi_5 = (x_8 \leftrightarrow (x_5 \vee x_6))$$
$$\phi_6 = (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\phi_7 = (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))$$

   If $\phi_1, .., \phi_n$ is subformulas we define ϕ to be $x_m \wedge \phi_1 \wedge \phi_2 \wedge ... \wedge \phi_k$ giving us following formula

$$\phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2))$$
$$\wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).$$

   Which can be constructed in polynomial time . C is satisfiabel if and only if ϕ is satisfiable

$$\langle C \rangle \in \texttt{CIRCUT-SAT} \Leftrightarrow \langle \phi \rangle \in \texttt{SAT}$$

5.5.2 3-CNF

Must be the OR of three different clauses, each having three literals in them. Must be distinct.

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

We can define

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

And show that it is NP complete doing almost the same as for the SAT problem, see slides for poof.

5.6 NP-Complete problems

Let L and $L\prime$ be two languages, with $L\prime \in NPC$.
If $L\prime \leq_p L$ then L is NP-Hard, if in addition $L \in NP$ then L is NP-Complete
General technique to show NP-Completeness of a language L:

- Show that $L \in NP$(Show that it can be verified in polynomial time)
- Pick another language $L\prime$ that is known to be NP-complete
- Show that $L\prime \leq_p L$, show that there is a polynomial time computible function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$

$$x \in L\prime \leftrightarrow f(x) \in L$$

5.6.1 The Clique problem

5.6.2 The Vertex cover problem

5.6.3 The Traveling salesman problem

5.6.4 The Ham-cycle problem

An undirecthed graph is hamiltonian if it contains a simple cycle containing every vertex of G. Cannot visit the same vertex more than once.
We can define the problem as

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

Meaning we give G as input and need to decide if it is hamiltonian or not.
Verifying `HAM-CYCLE`:
We consider an algorithm A_{Ham} taking two parameters $\langle G\rangle$ and $\langle C\rangle$. Cheking that $\langle G\rangle$ defines an undirected graph, and that $\langle C\rangle$ encodes a simple cycle. containing every vertex exaclty once. If it does, it outputs 1, otherwise 0.

5.6.5 The subset-sum problem

Given a set S of positive integers and given integer target $t > 0$

6 Exact exponential algorithms and parameterized complexity

Exact exponential algorithms: For all instanses find an exact solution in exponential time
Parameterized algorithms: An algorithm running in polynomial time, and find an exact solution for a subset of instances
Approximation algorithms: An algorithm that in polynomial time finds, for all instances, an approximate solution.
$\mathcal{O}^*$ **Notation:** For functions f and g we write $f(n) = \mathcal{O}^*(g(n)) iff f(n) = O(g(n)poly(n))$, where $poly(n)$ is a polynomial. For example, for $f(n) = 2^n n^2$ and $g(n) = 2^n$, $f(n) = \mathcal{O}^*(g(n))$

6.1 Size of a problem

Problem	certificate size	brute-force time
SAT, MIS	$m(x) = n$	$T(n) = \mathcal{O}^*(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) = \mathcal{O}^*(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) = \mathcal{O}^*(n^k)$
Vertex coloring	$m(x) = \log_2(n^n)$	$T(n) = \mathcal{O}^*(n^n)$

7 van Emde Boas Trees

The set $\{0, 1, 2, .., u-1\}$ the universe of values that can be stored and u the universe size. We assume that u is an exact power of 2 (2^k for some $k \geq 1$)

7.1 Preliminary approaches

7.1.1 Direct approach

We store dynamic-sets as a bit vector. To store the universe, we maintain an array $A[0, .., u-1]$ of u bits. The entry $A[x]$ holds a 1 if the value x is in the dynamic set, and it holds a 0 otherwise.
$Insert, Delete, Member$ takes $\mathcal{O}(1)$ time
$Minimum, Maximum, Succesor, Predessesor$ takes worst case $\Theta(u)$ time, as we may have to scan through $\Theta(u)$ elements.

7.1.2 Superimposing a tree of constant height/

The bit stored in an internal node is the logical-or of its two children.
for tree of constant height, we split the universe into $\sqrt{u}$ clusters with size $\sqrt{u}$

- *Minimum:* Start at root and always go the left-most node containing a 1
- *Maximum:* Start at root and always go to the right-most node containing a 1

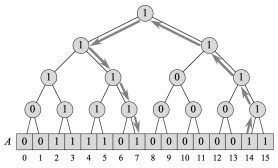


Figure 20.1 A binary tree of bits superimposed on top of a bit vector representing the set $\{2, 3, 4, 5, 7, 14, 15\}$ when $u = 16$. Each internal node contains a 1 if and only if some leaf in its subtree contains a 1. The arrows show the path followed to determine the predecessor of 14 in the set.

Figure 7: Superimposing a binary tree

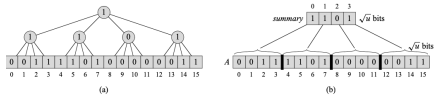


Figure 8: Superimposing a tree of constant height

- *Successor:* Look inside the cluster, if there is not an answer inside the cluster, go up to the summary vector and find the next cluster to have something in it, and then look inside that cluster
- *Pressessor:* Start at the leaf indexed by x, and head up toward the root until we enter a node from the right and this node has a 1 in its left child z. Then head down through node z always taking the rightmost node containing a 1.(Reverse form successor)
- *Insert:* We store a 1 in each node on the simple path from the appropriate leaf up to the root
- *Delete:* We go from the appropriate leaf up to the root, recomputing the bit in each internal node on the path as the logical-or of its two children.

If $x = i\sqrt{u} + j$ with i being the cluster number and $j(0 \leq j < \sqrt{u})$ being the position of x within that cluster.

$$High(x) = \lfloor x/\sqrt{u} \rfloor \tag{4}$$

$$Lox(x) = x \mod \sqrt{u} \tag{5}$$

$$Index(i, j) = i\sqrt{u} + j \tag{6}$$

High of x corresponds to the high half of the bits, and low of x corresponds to the bottom half of the bits.

7.2 Recurse

Recurse : $V = size - u$

- $V.clusters[i] = size - \sqrt{u}$
- $V.summary = size - \sqrt{u}$

Operations:

- *Insert(V,x):* Insert($V.cluster[high(x)], low(x)$) and Insert($V.summary, high(x)$) - Meaning we insert low(x) into high(x) and recusively update summary to maintain the name of the cluster(high(x)), so that it is not empty in the summary structure?
  $T(u) = 2T\sqrt{u} + \mathcal{O}(1) = \mathcal{O}(\log u)$

- *Successor(V,x):* Remeber
  1. Look into high(x)
  2. look at $V.summary$ for next 1(successor)
  3. look for first 1 in that cluster.

  could be written as

$$
\begin{aligned}
&i = high(x)\\
&j = Successor(V.Cluster[i], low(x))\\
&if\ j = \infty :\\
&i = successor(V.summary, i)\\
&j = successor(V.cluster[i], -\infty)\\
&Return\ index(i, j)
\end{aligned}
$$

7.2.1 Fix

To get better time we can store the minimum. $V.min$(When inserting). We could also be storing the maximum $V.max$(When inserting).We could then compare low(x) to the max in cluster[i], because if there is a successor it has to be within that cluster(because if low is greater than max, it has to be in the next cluster).

$$
\begin{aligned}
&i = high(x)\\
&if\ low(x) < V.cluster[i].max :\\
&j = successor(V.cluster[i], low(x)\\
&else :\\
&i = successor(V.summary, high(x))\\
&j = successor(V.cluster[i].min\\
&ReturnIndex(i, j)
\end{aligned}
$$

This now takes $\mathcal{O}(\log \log u)$

7.2.2 Fix pt.2

Don't store the min recusively.

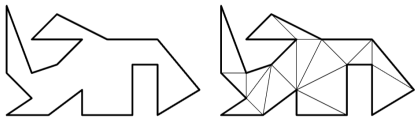
$$
\begin{aligned}
&Insert(V, x) :\\
&if\ V.min = None : V.min = V.max = x\\
&if\ x < V.min : swap\ x \leftrightarrow V.min\\
&if\ x > V.max : V.max = x\\
&if\ V.Cluster[high(x)].min = None : Insert(V.summary, high(x))\\
&Insert(V.cluster[high(x)], low(x))
\end{aligned}
$$

This now runs in $\mathcal{O}(\log \log u)$

8 Polygon triangulation

8.1 Triangulation

Trianulation, is the decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals, example(**??** We restrict ourselves to regions that



are simple polygons, that is, regions enclosed by a single closed polygonal chain that does not intersect itself. Thus we do not allow regions with holes.
Lemma: A polygon P with n vertices can be triangulated, and any triangulation has $n-2$ triangles using $n-3$ diagonals.
Theorem: $\lfloor \frac{n}{3} \rfloor$ is sufficient
To find such a subset we assign each vertex of P a color: white, gray, or black. The coloring will be such that any two vertices connected by an edge or a diagonal have different colors. This is called a 3-coloring of a triangulated polygon. In a 3-coloring of a triangulated polygon, every triangle has a white, a gray, and a black vertex

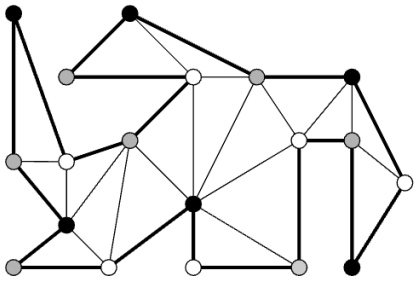
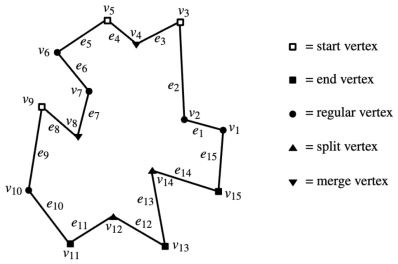


Figure 9: 3-coloring example

8.2 Partitioning a Polygon into Monotone Pieces

Definition: A polygon that is monotone with respect to the y-axis is called y-monotone. If we walk from a topmost to a bottommost vertex along the left (or the right) boundary chain, then we always move downwards or horizontally, never upwards.
Strategy: Partition P into y-monotone pieces, and then triangulate the piece.

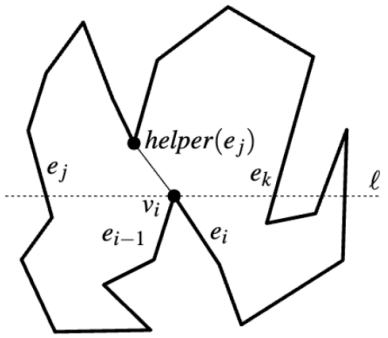
Partition is done by getting rid of turn vertices(Vertices that shift downwards to upwards or vice versa: Start,end, split and merge are all turn vertices**??**)



- ***Start vertex:***two neighbors lie below it and the interior angle at v is less than π
- ***Split vertex:***Two neighbors lie below it and the interior angle at v is greater than π
- ***End vertex:***two neighbors lie above it and the interior angle at v is less than π
- ***Merge vertex:*** two neighbors lie above it and the interior angle at v is greater thanπ
- ***Regular vertex:***has one of its neighbors above it, and the other neighbor below it.

Idea: Use horizontal sweep line going down(if two vertices has same y-coordinate, then the leftmost point has priority). The goal of the sweep is to add diagonals from each split vertex to a vertex lying above it.

Helper: Let e_j be the edge immediately to the left of v_i on the sweep line and let e_k be the edge immediately to the right of vi on the sweep line. Then helper is defined as the lowest vertex above the sweep line such that the horizontal segment connecting the vertex to e_j lies inside $\mathcal{P}$.



Handling merge vertices: Suppose the sweep line reaches a merge vertex v_i. Let e_j and e_k be the edges immediately to the right and to the left of v_i on the sweep line, respectively. Observe that v_i becomes the new helper of e_j when we reach it.We would like to connect v i to the highest vertex below the sweep line in between e_j and e_k.

8.3 Triangulating a Monotone Polygon

Let $\mathcal{P}$ be a strictly(not contain horizontal edges) $y-$monotone polygon with n vertices.
The algorithm handles the vertices in order of decreasing y-coordinate. If two not yet triangulated triangles split off vertices have the same y-coordinate, then the leftmost one is handled first
When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible.
Using a stack to make diagonals:

9 Approximation algorithms

Definition: An Algorithm for an optimization problem has approximation ratio $\rho(n)$ if for every input of size n

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

Where C^* is the optimal solution and C is the produced solution.
The $\frac{C}{C^*}$ part is a minimization problem, and $\frac{C^*}{C}$ a maximization problem.

9.1 Vertex-cover

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

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

Figure 10: Pseudocode for Approximation Vertex Cover

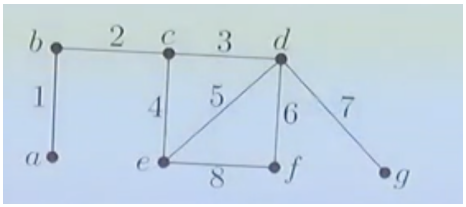


Figure 11: Example graph

1. We start of by choosing a and be, so we will have $C = \{a, b\}$ and remove all edges incident to a and b.
2. We choose c and d ($C = \{a, b, c, d\}$) and remove all edges incident.
3. Then we choose e and f ($C = \{a, b, c, d, e, f\}$) and remove the last remaining edge

Theorem: The APPROX-VERTEX-COVER is a 2-approximation algorithm
Proof: Let C^* be an optimal cover. Let $A \subset E$ be the edges chosen by the algorithm. An endpoint in of each $uv \in A$ must be in the optimal cover(C^*). SInce it is a cover it must need all the edges. When we choose an edge we remove all the outgoing edges, therefor they never share an endpoint, giving us:

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

As A must be equal to half the number of vertices we put in our C.

9.2 Traveling Salesman

Definition: Given a complete undirected graph $G = (V, E)$. For all $u, v \in V$, we are given $c(uv) = \{0, 1, ..\}$. Goal is to find minimum weight cycle through all vertices.
Theorem: APPROX-TSP is a 2−approximation algorithm
Proof: Let H^* be an optimal solution. We have that $c(H) \leq c(H*)$, as T is the total cost of connecting all vertices and H^* is the smallest wights cycle. Then we have $c(W) = 2c(H)$ as the euler tour visits each vertex twice. Lastly we have $c(H) \leq c(W)$, giving us $c(H) \leq 2c(H^*)$

APPROX-TSP(G, c)
  Find MST T
  Make Euler tour W using each edge of T twice
  Shortcut W to H by skipping duplicates
  Return H

Figure 12: TSP algorithm

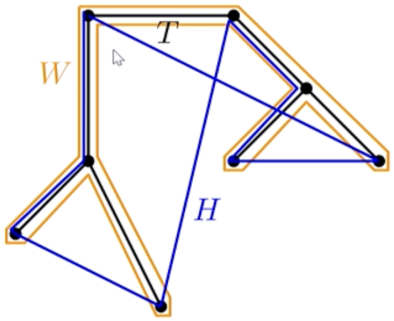


Figure 13: Approx-TSP example tour

9.3 Set cover

Input: Pair(x,f) where x is a finite set and $f \subseteq \mathcal{P}(x)$ is a family of subsets of x
Goal: find $\mathcal{C} \subseteq f$ covering x, i.e $\bigcup_{S \in \mathcal{C}} S = x$ with $|C|$ minimum We can see an

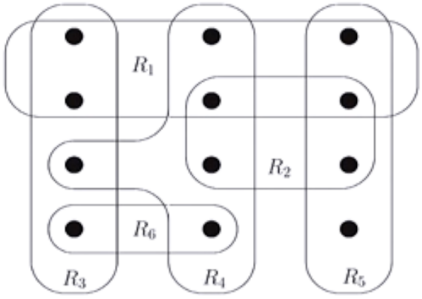


Figure 14: Set-Cover example

example in figure 14, where we have 6 different sets, and want to cover all of the dots in the set.

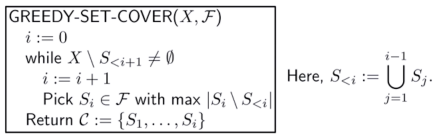


Figure 15: set cover algorithm

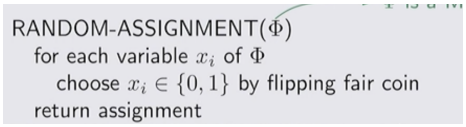


Figure 16: Max-3-sat example algorithm

9.4 MAX-3-SAT

Theorem: Random assignment is a $\frac{8}{7}$-approximation algorithm
Proof: We let $\Phi = C_1 \wedge C_2 \wedge ... \wedge C_n$ and consider $C_i = l_1 \vee l_2 \vee l_3$. For C_i to not be true we must have that $C_i = \neg l_1 \vee \neg l_2 \vee \neg l_3$
The probabilities for this is

$$\mathbf{Pr}[\neg C_i] = \mathbf{Pr}[\neg l_1] \cdot [\neg l_2] \cdot [\neg l_2] = \frac{1}{2}^3 = \frac{1}{8}$$

Meaning the probability that it is true is

$$\mathbf{Pr}[C_i] = 1 - \mathbf{Pr}[\neg C_i] = 1 - \frac{1}{8} = \frac{7}{8}$$

We define X to be the number of satisfied clauses

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

$$\mathbf{E}[X] = E\left[\sum_{i=1}^{n} [C_i]\right] = \sum_{i=1}^{n} \mathbf{E}[C_i] = \sum_{i=1}^{n} \frac{7}{8} = \frac{7n}{8}$$

This gives us an approximation ratio of

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

9.5 Weighted Vertex Cover

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

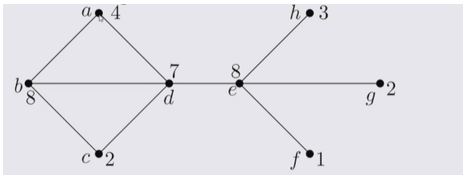


Figure 17: Weighted vertex cover example

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

Figure 18: Weighted vertex cover algorithm

Now: We are given weight $w(v) > 0$ for each $v \in V$
Goal: Find a Vertex cover with minimum

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

For the example, the minimum unweighted vertex cover will be $\{b, d, e\}$, and the weighted will be $\{a, c, d, f, g, h\}$
Can be made a 0-1 program:
0-1-integer program (IP):

$$x_v \in \{0, 1\},\ \forall v \in V \quad (x_v = 1 \iff v \in C)$$

$$x_u + x_v \geq 1,\ \forall uv \in E \quad (\text{edge } uv \text{ covered})$$

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

Which can be relaxed to a linear program replace $x_v \in \{0, 1\}$ with $0 \leq x_v \leq 1$. Result:

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

$$x_u + x_v \geq 1,\ \forall uv \in E$$

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

Theorem: The weighted vertex cover algorithm is a polynomial-time 2-approximation algorithm.
Proof: The linear program runs in polynomial-time. Also $uv \in E \implies \bar{x}_u + \bar{x}_v \geq 1 \implies \bar{x}_u \geq \frac{1}{2} \vee \bar{x}_v \geq \frac{1}{2} \implies u \in C \vee v \in C$. We now have that $w(C) = \sum_{v \in C} w(v) = \sum_{v \in V} w(v)[v \in C] = \sum_{v \in V} w(v) \left[\bar{x}_v \geq \frac{1}{2}\right] \leq \sum_{v \in V} w(v) \left[0 \leq 2\bar{x}_v\right]$.

9.6 Approximation schemes

Plynomial-time Approximation Scheme(PTAS): Approximation algorithm that takes instance I of an optimization problem P and $\varepsilon > 0$ as input. For any fixed ε works as $(1+\varepsilon)$-approximation algorithm for P
Fully polynomial-time approximation scheme (FPTAS): PTAS with runtime polynomial in $1/\varepsilon$ and the size of I.

10 Extra

10.1 Log regler

Vi siger

$\log_{10}(200) = 2,3$ fordi $200 = 10^{2,3}$

Vi kan gøre det mere generelt:

Hvis $y = 10^x$ så er $\log_{10}(y) = x$

eller sagt på en anden måde

$\log_{10}(10^x) = x$

Med ord ville man sige

1. $\log(a \cdot b) = \log(a) + \log(b)$
2. $\log\left(\frac{a}{b}\right) = \log(a) - \log(b)$
3. $\log(a^x) = x \cdot \log(a)$