Algorithms Probability and Computing

Nicolas Muntwyler

Autumn 2019

Contents

1 Bootstrapping Techniques

We look at 2 approaches for bootstrapping

1.1 MST

Satz 1.1. If the weight function $w : E \to R$ is injective (i.e., no two edges have the same weight), then the graph $G = (v, E)$ contains exactly one MST.

Def. 1.1 (e-min(v)). For every vertex $v \in V$ let $e_{min}(v)$ denote that edge incident to v that has minimum weight.

Def. 1.2 (e-max(C)). For every cycle C let $e_{max}(C)$ denote the edge in C tat has maximum weight.

Def. 1.3 (T-heavy). An edge $e \in E$ is called T-heavy iff $e \notin T$ forms a cycle C with edges of T and all other edges in C have a lower weight.
T is an MST.

Def. 1.4. (Edge-Contraction) Contraction of an edge e = (u,v) means that we "glue" u and v together into a new single vertex and remove loops. The resulting graph is denoted by G/e
An edge contraction can be performed in $O(n)$

Satz 1.2. No edge $e_{max}(C)$ can belong to the MST

Satz 1.3. If all edges are unique then all edges $e_{min}(v)$ for $v \in V$ belong to the MST.

Satz 1.4. $T = (V, E_T), G = (V, E)$
T is MST $\iff$ all edges $e \in E \setminus E_T$ are T-heavy

Satz 1.5. Every edge that is not in MST T is T-heavy

Algorithm 1.1 (Boruvka).
$\forall v \in V$: compute $e_{min}(v)$, insert $e_{min}(v)$ in the MST, contract $e_{min}(v)$ (by removing loops and double edges by keeping only the cheapest edge)
recurse until the graph contains only one vertex.

Runtime of one Boruvka-step: $O(m)$

Algorithm 1.2 (Randomized MST).
Input: (G, $\emptyset$)
Output: MST

Do 3 Boruvka steps (forms G')
Select edges with probability 1/2
Recursively find MSF T of selected edges
From the unselected edges add all not T-heavy edges to T and delete all others (forms G").
recurse G" (until the graph has only one vertex)

Worst-case: $O(min\{m \log(n)), n^2\})$
Expected Runtime : $O(m)$
Remark: The resulting MST consists only of the edges contracted in the Boruvka steps.

1.2 Computing Minimum Cuts in $\tilde{O}(n^2)$

Def. 1.5. Notation $\tilde{O}(\cdot)$ is similar to the Big-O notation, except that we hide not only constants but also log-factors. Thus $O(n^2 \log(n)^3)$ will be $\tilde{O}(n^2)$

Def. 1.6 (Cut). A cut in a graph $G = (V, E)$ is a subset $C \subseteq E$ of edges such that the graph $(V, E \setminus C)$ is disconnected

Def. 1.7 (Minimum Cut). A minimum cut is a cut with the minimum possible number of edges.
Let $\mu(G)$ denote the size of a minimum cut in G.

!!!! We assume that the graph is connected, but allow multiple edges. !!!!

Lemma 1.1. Runtimes:

- Edge contraction in O(n)
- Choose an edge uniformly at random in O(n)
- Find the number of edges connecting two given vertices in O(1)

Satz 1.6. Minimum Cut Observations:

- $\mu(G/e) \geq \mu(G)$ (yep)
- If there exists an minimum cut C in G such that $e \notin C$ then $\mu(G/e) = \mu(G)$

Lemma 1.2. The probability of $\mu(G/e) = \mu(G)$ for a randomly chosen edge is at least $1 - \frac{2}{n}$

Remark: $p_0(n) \leq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{t}{t+2} \cdot \frac{t-1}{t+1} = \frac{t(t-1)}{n(n-1)}$

Satz 1.7 (Important Inequality).

$$1 + x \leq e^x$$

Algorithm 1.3 (Randomized Minimum Cut).
Input: G
Output: $\mu(G)$

Run BasicMinCut(G) around 10n(n-1) times and choose the minimum over all runs

Runtime: $O(n^4)$ (and succeeds with probability: $1 - e^{-20}$)

Algorithm 1.4 (BasicMinCut).
Input: G
Output: potential $\mu(G)$)

while *G has more than 2 vertices* **do**
  pick a random edge e in G
  $G \leftarrow G/e$
end
return the size of the only cut in G

Runtime: $O(n^2)$

We now introduce the bootstrapping:
We observe that the probability of contracting an edge $e \in C$ gets higher and higher. If we falsely contract it we have to begin from the beginning. Now the idea is to have a cutoff, at which we have an other algorithm that returns a minimum cut of the smaller graph with a certain probability. Note that this other Algorithm

can actually be the exact same. This way we go down recursively until a only n $\leq$ 16.
The actual algo works by duplication. We start with G. We duplicate G. In both copies (with n vertexes each) we then contract random edges independently till each has αn vertexes. Then we again duplicate both graphs and continue contracting edges in the now four graphs till they each have $\alpha^2 n$ vertexes. We again duplicate those graphs to get 8 graphs, and so forth.

Algorithm 1.5 (**MinCut(G)**).
Input: G
Output: potential $\mu(G)$

if $n \leq 16$ **then**
  compute $\mu(G)$ by some deterministic method
else
  $t \leftarrow \alpha n + 1$
  $H_1 \leftarrow$ RANDOMCONTRACT(G,t)
  $H_1 \leftarrow$ RANDOMCONTRACT(G,t)
  return min(MinCut(H_1), MinCut(H_2))
end

Runtime: $O(n^2 \log(n))$ with $\alpha = 1/\sqrt{2}$

In order to have a high success probability we repeat MinCut(G) about $O(\log(n)^2)$ times. This gives us the final algo:

Algorithm 1.6 (**MinCut(G)**).
Input: G
Output: $\mu(G)$

Run MinCut(G) $Ca\log(n)^2$ times and choose the minimum over all runs.
(Where C is a suitable constant)

Runtime: $O(n^2 \log(n)^3)$ and succeeds with probability $1 - n^{-a}$

2 Random(ized) Search Trees

Def. 2.1 (**Random Search Trees**). Binary search trees ($\tilde{B_S}$ where S is a grammar, in our case numbers) obtained by inserting elements in random order without rebalancing. The empty tree is denoted by λ.

Def. 2.2 (**Keys**). Keys are distinct real numbers, that get inserted into the tree.

Def. 2.3 (**w(v)**). $w(v)$ denotes the number of nodes in the subtree rooted at v.

Def. 2.4 (**Depth**). The depth or $d(v)$ is defined as:

$$\begin{cases} 0 & \text{if v is the root} \\ 1 + d(u), \text{u parent of v}, & \text{otherwise} \end{cases}$$

Remark: How many to the top.

Def. 2.5 (**Probability of a tree, Probability Distribution of a tree**). The probability of a tree is defined as the probability that it is the result of random insertion of the nodes
We obtain a probability distribution.

Lemma 2.1. The Probability of the tree according to the above distribution is:

$$\prod_{v \in V} \frac{1}{w(v)}$$

Def. 2.6 (**Rank(x)**). Given some finite set $S \subseteq R$, the rank of $x \in R$ in S is:
rk(x) = $rk_S(x) := 1 + |\{y \in S | y \leq x\}|$
Easier: x is the rk(x)-smallest element in S.

Def. 2.7 (**Depth of key with rank i**). $D_n^{(i)}$ is the random variable for the depth of the key of rank i in a random search tree for n keys.
Remark: For $D_n^{(1)}$ you often write: D_n (Depth of key of rank 1 = Depth of smallest key in S)

Satz 2.1 (**Expected Depth of smallest key**).

$$E[D_n] = d_n = H_n - 1 = ln(n) + O(1)$$

To achieve this, we used:

$$\begin{aligned} E[D_n] &= \sum_{i=1}^{n} \underbrace{E[D_n | rk(root) = i]}_{\begin{cases} 0 & \text{if i = 1} \\ 1 + E[D_{i-1}] & \text{otherwise} \end{cases}} \cdot \underbrace{Pr[rk(root) = i]}_{1/n} \\ &= \frac{1}{n} \cdot \sum_{i=2}^{n} 1 + d_{i-1} = d_n \end{aligned} \tag{1}$$

We used the observation, that the depth of the smallest key is 0 if it sis in the root (rk(root) = 1 $\iff$ smallest element is at the root), or it is 1 plus the depth of the smallest key in the left subtree.
And to solve the recursion $d_n = \frac{1}{n} \cdot \sum_{i=2}^{n} 1 + d_{i-1}$, we used the **Gauss-Trick**:

$$\begin{vmatrix} d_n & = \frac{1}{n} \cdot \sum_{i=2}^{n} 1 + d_{i-1} \\ d_{n-1} & = \frac{1}{n-1} \cdot \sum_{i=2}^{n-1} 1 + d_{i-1} \end{vmatrix}$$

and subtract the second line form the first line to get:

$$d_n = \frac{1}{n} + d_{n-1}$$

Which is a nicer recursion that we can solve much more easily to get:

$$d_n = H_n - 1$$

Satz 2.2 (**Expected Overall Depth**).

$$E[\sum_{i=1}^{n} D_n^{(i)}] = E[X_n] = x_n = 2(n+1)H_n - 4n = 2n\ln(n) + O(n)$$

To achieve this we used the same trick as before to get:

$$nx_n = 2(n-1) + (n+1)x_{n-1}$$

To solve this recursion we use the **Multiply-Trick**
We multiply by $\frac{1}{2n(n+1)}$ to get:

$$\underbrace{\frac{x_n}{2(n+1)}}_{=:f_n} = \frac{n-1}{n(n+1)} + \underbrace{\frac{x_{n-1}}{2n}}_{=f_{n-1}}$$

Which is a nicer recursion that we can solve much more easily to get:

$$f_n = H_n - 2 + \frac{2}{n+1} \Longleftrightarrow x_n = 2(n+1)H_n - 4n$$

Satz 2.3 (**Expected Height**).

$$E[\text{Expected Height}] = E[\max_{1\le i\le n} D_n^{(i)}] = E[X_n] = x_n = 4.311\ln(n)$$

The max-operator seems to be very cumbersome. Therefore we get rid of it with the **MachineLearning-Trick**:

$$\max(a_1, a_2, ..., a_n) \approx \log(2^{a_1} + 2^{a_2} + ... + 2^{a_n})$$

To further improve the accuracy we choose not 2 as the basis but 4.311.. . On the way we used all tricks above and the Jensen-Inequality.

Lemma 2.2 (**Jensen's Inequality**). If $f : R \to R$ is a convex function, then f(E[x]) $\le$ E[f(x)].

Example-functions: $f(x) = x^2$, $f(x) = 2^x$

Satz 2.4 (**Tail Estimates**).

$$Pr[X_n \ge \tau\ln(n)] \le n^{\tau(1-\ln(\tau/2)-1)}$$

Which is achieved with the Markov-Inequality.

Def. 2.8 (A_i^j**- Ancestor**).

$$A_i^j := [\text{node j is ancestor of node i}] = \begin{cases} 1 & \text{if node j is ancestor of node i} \\ 0 & \text{otherwise} \end{cases}$$

Satz 2.5 (**Expected Depth of Individual Keys**).

$$E[D_n^{(i)}] = H_i + H_{n-i+1} - 2 \le 2\ln(n)$$

Proof:

$$
\begin{aligned}
E[D_n^{(i)}] &\overset{(1)}{=} E[\sum_{j=1,j\neq i}^{n} A_i^j] \\
&\overset{\text{Lin.}}{=} \sum_{j=1,j\neq i}^{n} E[A_i^j] \\
&\overset{(2)}{=} \sum_{j=1,j\neq i}^{n} Pr[A_i^j = 1] \\
&\overset{(3)}{=} \sum_{j=1,j\neq i}^{n} \frac{1}{|i-j|+1} \\
&= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= \sum_{j=2}^{i} \frac{1}{j} + \sum_{j=2}^{n-i+1} \frac{1}{j} \\
&= H_i + H_{n-i+1} - 2
\end{aligned}
\tag{2}
$$

(1) $A_i^1 + A_i^2 + ... + A_i^n$, which is exactly how many ancestors i has, which is exactly its depth.
(2) $E[A_i^j] = Pr[A_i^j = 1] \cdot 1 + Pr[A_i^j = 0] \cdot 0 = Pr[A_i^j = 1]$
(3) Lemma 3.3

Lemma 2.3. In a random search tree for $n \leq \max i, j$ keys.

$$Pr[A_i^j = 1] = Pr[\text{node j is ancestor of node i}] = \frac{1}{|i-j|+1}$$

Proof-Idea:

$$
\begin{aligned}
Pr[A_i^j = 1] = Pr[A_i^j = 1 | \text{rk(root)} \in \{i..j\}] \cdot Pr[\text{rk(root)} \in \{i..j\}] \\
+ Pr[A_i^j = 1 | \text{rk(root)} \notin \{i..j\}] \cdot Pr[\text{rk(root)} \notin \{i..j\}]
\end{aligned}
$$

And we prove that (wlog $j > i$):

$$Pr[A_i^j = 1 | \text{rk(root)} \in \{i..j\}] = \frac{1}{j-i+1}$$

$$Pr[A_i^j = 1 | \text{rk(root)} \notin \{i..j\}] = \frac{1}{j-i+1}$$

Satz 2.6 (**Expected Comparisons in Quicksort**). Let t_n denote the expected number of comparisons:

$$t_n \overset{(1)(2)}{=} n - 1 + \sum_{i=1}^{n} (t_{i-1} + t_{n-i}) \frac{1}{n} = n - 1 + \frac{2}{n} \sum_{i=1}^{n} t_{i-1} = 2(n+1)H_n - 4n$$

(1): n-1 comparisons in this iteration (compare n-1 elements with your pivot)
(2): Recursion is in style of: [#comparisons | pivot = i]·Pr[pivot = i] = (LeftTree + RightTree) $\cdot \frac{1}{n}$

We now make 2 Observations:

(1): Ever computation of quicksort maps to a search tree (pivot = root)
(2): The number of comparisons in quicksort is exactly the expected Depth of a search tree

The idea for (2) is that $\sum_v d(v) = \sum_v (w(v) - 1)$ where $\sum_v (w(v) - 1)$ is the number of comparisons.

Def. 2.9 (Randomized Search Tree - Treap). We introduce a randomized search tree:
Treap = (search)tree + (min)heap

- defined for sets: $Q \subseteq R \times R$
- $x \in Q$ with $x = (key, prio)$
- Search tree wrt. to keys & min heap wrt. to priorities
- Choose Priorities u.a.r. $\in$ [0,1]

Remark: Since for every newly inserted key a random priority is chosen, the treap is a random search tree for the keys, independently from the insertion order.

Algorithm 2.1 (Insert(x)).

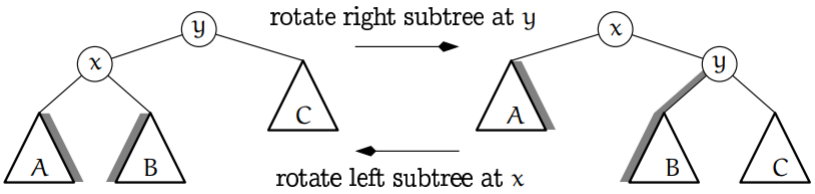
- insert x as a leaf according to the rules of a search tree
- rotate x up until at correct position wrt to its priority

Algorithm 2.2 (Delete(x)).

- rotate x down the tree until it is a leaf (left/rigt rotations)
- remove x

Remark: the operations join() and split() also exist

Rotation:



Lemma 2.4. $\forall T\ \forall x$: The expected number of rotations ≤ 2 (for insertion/deletion)

Lemma 2.5. $\forall T\ \forall x$: Each rotation of x increases: (length of left spine) + (length of right spine), by exactly one.

Lemma 2.6. $\forall n$: in a random search tree for [n] we have:
$\forall j$:

- expected length of left spine: $1 - \frac{1}{j}$
- expected length of right spine: $1 - \frac{1}{n-j+1}$

We use the two latter lemmas to prove the first one.

3 Point Location

Setting: A domain is partitioned. It is assumed that the partition is fixed while there are many queries. A query is often of the sort: "in which partition does the point p lie?"

Def. 3.1 (**Locus approach**). Problem: Given a set S of real numbers and for a given query number find the closest number in S Solution: We partition the domain into regions of equal answers to the query.

3.1 Point/Line Relative to a Convex Polygon

Def. 3.2 (**Convex hull**). The convex hull $conv(P)$ of a finite set P of points in the plane is a bounded convex set. It is bounded by a convex polygon.

Lemma 3.1. The Convex hull can be computed in O(n log n)

Def. 3.3 (**Convex Polygon**). Closed simple piecewise linear curve, that separates the interior of $conv(P)$ from the exterior.
Remark: It can be finitely described by a sequence of its vertices in counterclockwise order.

Lemma 3.2. Given a line l, we can decide in O(1) whether a point x is left/right/above/below of l.

Algorithm 3.1 (**Inside/On/Outside a Convex Polygon C**).
Problem: Given query point q, is q in/on/outside the polygon.
Preprocessing: Sort x-coordinates of the vertices of C. The intervals between two consecutive x-coordinates get associated with the two lines that carry the edges of C in the corresponding x-range of the plane.

1.) Use binary search to locate q.x in the structure. Only continue if q.x is not the biggest or smallest x-coordinate.

2.) q.x lies in some interval and we compare q.y with the two associated lines.

3.) If q.y is below one line and above the other, q is inside the polygon.

Query-Runtime: O(log n) - without preprocessing

Algorithm 3.2 (**A Line Hitting a Convex Polygon**).
Problem: Does line l transect a given polygon C?
Preprocessing: We direct every edge of C in the direction as we pass it moving around C in counterclockwise order. Every such directed edge e has an angel α_e with the x-axis. Compute these angles, and after every angle is computed, sort them. We associate vertex incident to edge of angle α_{i-1} or α_i with the interval $[\alpha_i - 1, \alpha_i]$ (Locus approach)

1.) Compute angles β_1 and β_2 of l (Both direction)

2.) Binary search the structure above to get the associated vertex v and v'

3.) l misses C iff v and v' lie on the same side of l.

Note: l must be between the tangent1: parallel to l, through v and tangent2: parallel to l through v'

Query-Runtime: O(log n) - without preprocessing

3.2 Line Relative to Point Set

Algorithm 3.3 (**Are all points on the same side of a line l**).

Preprocessing: Compute convex hull C

1.) all points are on the same side iff l doesn't intersect C

3.2.1 Reporting the Points Below a Query Line

Let k:= number of points below the query line l.
An optimal solution would be: O(k + logn)

We first try to solve the problem if $P \subseteq conv(P)$ In this case its easy.

Algorithm 3.4 (**Report points below a Query Line l and P $\subseteq$ conv(P)**).

1.) Find vertex p, which is contained in the tangent, which has C above it and is parallel to l

2.) If p if above l stop. (no points)

3.) Start from p and move in clockwise order through the vertices of C and report these points. Until either no more points are left (stop), or we find one that is above l.

4.) Do 3. but counterclockwise

Runtime: $O(k + \log n)$ - without preprocessing

Def. 3.4 (**Onion of P**). Onion of P: $(R_0, V_0), (R_1, V_1), ..., (R_t, V_t)$
R_0 = conv(P), V_0 = vertex set of R_0

R_0 is the convex hull of P. R_1 the convex hull of $P \setminus V_0$, ...

Observations:

- If R_i is completely above a line l, then so are all R_j with $j \in \{i, , t\}$
- $(V_0, V_1, ..., V_t)$ is a Partition of P

We can construct the following algo to determine all lines below l.

Algorithm 3.5 (**Report points below a Query Line l - Onion simple**).

Preprocessing: compute all V_i (Compute the onion)

1.) let i = 0

2.) Apply algo 4.4 with $P = V_i$. If no points are reported; stop.

3.) Repeat step 2 with i=i+1, unless we exhausted all sets.

Runtime: $(k \log n + \log n)$ - without preprocessing

Proof:
Case: Reached a V_j that is completely above l

$$O(\sum_{i=0}^{j-1} k_i + \sum_{i=0}^{j} \log n_i) = O(k + (j+1)\log n) = O((k+1)\log n)$$

(We used, that $k \geq j$)
Case: We exhaust all sets an reach V_t

$$O(\sum_{i=0}^{t} k_i + \sum_{i=0}^{t} \log n_i) = O((k+1)\log n)$$

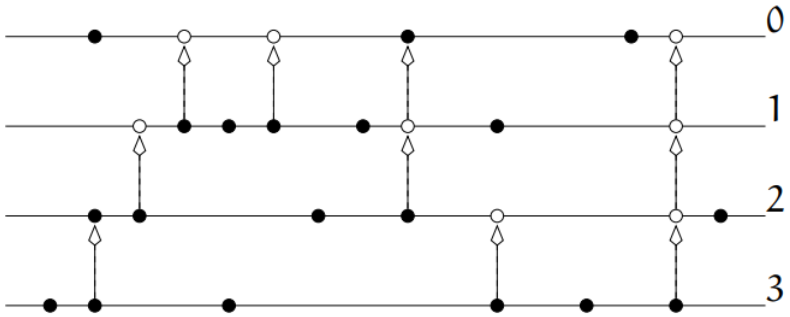
Sadly this runtime is not good enough for us. We want to do better. But where did we lose time? In each new polygon we start over with our binary search for the angle. But we would do better, if we keep the information of the last iteration.

We can do better by putting all angels of the different V_i together in one array and have for each interval t+1 pointers to the respective intervals in the structures of each onionring. But this would take up non-linear storage. Therefore we do something much more clever, called **fractional cascading**:

Def. 3.5 $(S_i, \overline{S_i})$**.** Let for $i \in \{0..t\}$

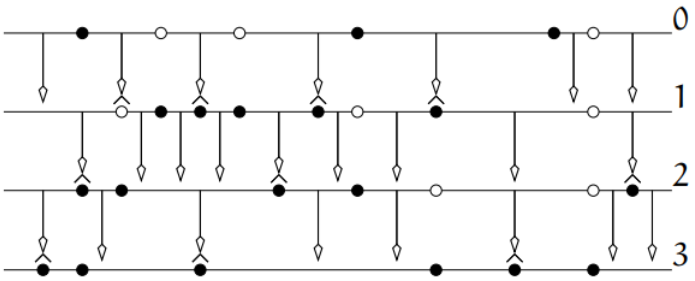
- S_i denote the array of all angels in V_i.
- $\overline{S_i}$ denote the array which contains all elements from S_i and every second element of S_{i+1}. The last set is untouched, so $\overline{S_t} = S_t$

We start at the lowest level S_t and put every second point into S_{t-1} and so on till S_0. The following image is for $t = 3$:



Additionally we add pointers:
For each interval I in S_i we make a pointer to either:

- Interval I' S_{i+1} - (If I is completely contained by I')
- number α in S_{i+1} - (If α (numerically) would be in I)



Now if we need to search the closest number β to a query number x for each S_i, we only have to binary search S_0. Afterwards its always constant time. Because if we know in which interval β must be in S_i we can just follow the pointer to the interval β must be in S_{i+1}.

Algorithm 3.6 (**Report points below a Query Line l - Onion with fractional cascading**).

Preprocessing: compute all V_i (Compute the onion) and setup our structure with the $\overline{S_i}$'s for fractional cascading.

1.) let i = 0

2.) Apply algo 4.4 with $P = V_i$, if no points were reported; stop. Otherwise do step 3 with i=i+1

3.) Apply step 2-3 of algo 4.4 with $P = V_{i+1}$. if no points were reported; stop. Otherwise repeat step 3 with i=i+1

Runtime: $O(k + \log n)$ - without preprocessing
Space: $O(n)$
Preprocessing takes $O(n^2 \log n)$ (could be enhanced to $O(n \log n)$)

Lemma 3.3. $\sum_{i=0}^{t} |\overline{S_i}| \leq 2n$

3.2.2 Counting the Points Below a Query Line

Def. 3.6 (Duality). Let * be the mapping, that maps points to lines and vice-versa:

point p = (a,b) → line p*: y = ax - b

line l : y = ax + b → point l* = (a,-b)

Lemma 3.4. Let p be a point and l a non-vertical line in R^2

(i) (p*)* = p and (l*)* = l

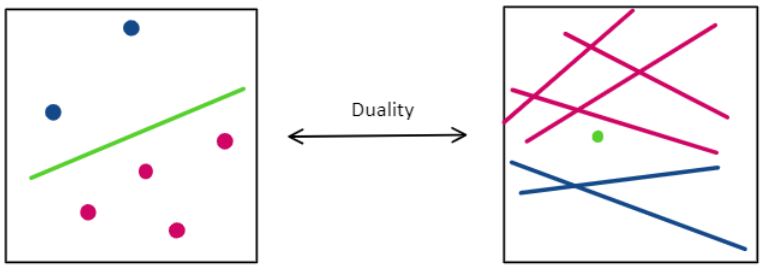
(ii) $p \in l$ iff $l^* \in p^*$

(iii) p lies above l iff l* lies above p*

Remark: This lemma wouldn't hold, if we would choose (a,b) instead of (a,-b)

We now use this lemma as our basis for the algorithm. We note that:

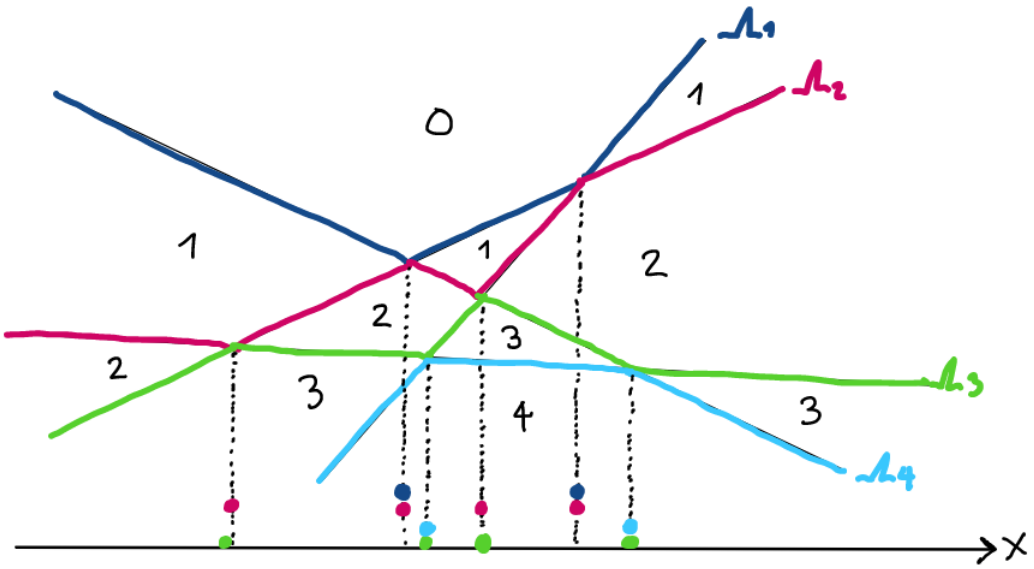
All points p below l are exactly those for which l* is below p*



The idea is to solve our problem by solving an other one, which will however return the same result thanks to the duality. Hence we now want to solve how many lines are above l*. Hence we solve a problem of the type:

"How many lines are above point p"

We solve this in a locus approach. We solve the problem by determining in which partition p lies. The partition tells you how many lines are above you.



Def. 3.7 (k-th level: Λ_k). Λ_k is an x-monotone bi-infinite curve in the plane. All points above this curve have at most k-1 lines above it

In preprocessing we calculate these level and store (intersections, edges, cells) in $O(n^2)$ space, since there exist at most n^2 edges in total and $\binom{n}{2}$ intersection points.

Algorithm 3.7 (Count points below a Query Line l - LevelBasic).
Preprocessing: Compute the structure for the Levels, Λ_k's
Result: Fist level Λ_i in which q is below, to get: i-1

1.) Go through the level in a binary search matter (Find first level in which q is below). In the level do:
2.) Binary search the level for the interval (edge) in which the query point lies in respect to the x-coordinate.
3.) Check for that edge if q is below or above it.

Runtime: $O(\log^2(n))$ (Binary search in a binary search)
Preprocessing: $O(n^2 \log(n))$ but can be done in $O(n^2)$
Space: $O(n^2)$

But once again we can improve this runtime by using fractional cascading. Where we create $|\Lambda_i|$'s from Λ_i's. The Binary search from the first step can be visualized by storing the levels in a balanced tree. At the root we spend logarithmic time to find the interval of this level. Depending on the outcome we then proceed to the left or right child. The leaves hold the number of lines above q.

Till now we began our search form scratch on each node of the tree. But now we do fractional cascading and start coping every second value to the node above. And the again start inserting the new pointers from top to bottom. Afterwards we only need logarithmic time in the first level and afterwards its constant time.

Algorithm 3.8 (Count points below a Query Line l - LevelCool).
Preprocessing: Compute the structure (tree) for the Levels, Λ_k's

1.) Binary search the root for the interval (edge) in which the query point lies in respect to the x-coordinate.
2.) Go to the left child or right child depending on the comparison.
3.) Find corresponding edge in O(1) and do step 2 again. Unless we reached a leaf; return the leaf.

Runtime: $O(\log n)$ - without preprocessing
Preprocessing: $O(n^2 \log(n))$ but can be done in $O(n^2)$
Space: $O(n^2)$

3.3 Planar Point Location - More Examples

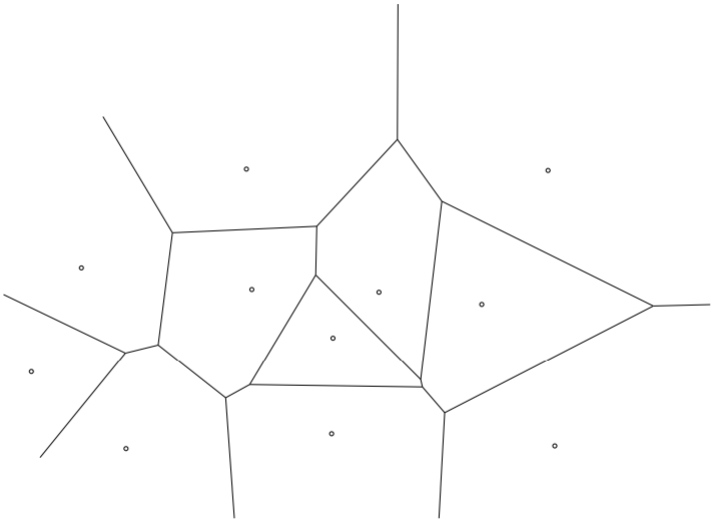
We often can translate a problem into a Problem of the type: Locate a point on a map. Deciding in which partition the point is located.

3.3.1 Point Relative to Convex Polytope

Analogy: Is a point q inside or outside a football.
Solution: Look down on football from heaven (like a projection). Determine in which facet the point lies and check if the facet is above q. If "no", q is not inside the polytope. If "yes" do the analogous with from hell and below. If still yes, then q is inside.
Runtime: $O(\log n)$

3.3.2 Closest Point in the Plane

Problem: Of a set of Points S, which point is closest to our query point q.
Solution: We take the locus approach and ask; for $p \in S$, what are the query points q for which p is the closest points, which gives us a partition into cells: The **Vornoi Diagram**:



The construction of the Vornoi Diagram uses bisectors.
Summing up, the number of cells, edges and vertices of the Vornoi diagram of n point is: O(n)
Why? Cells are trivially O(n). For the edges we argue with the planar Delaunay-diagram of the vornoi diagram, to see, that we have at most 3n-6; O(n) edges. Since edges are O(n), so are the vertices O(n)

We now observe that each cell is bounded by straight line segments. Hence we can just find out which line segment is above q to determine in which cell q must be and hence to which point q is closest. To determine this line segment we use Algo 4.11.
Runtime: $O(\log n)$ - (Preprocessing = Building Vornoi-Diagram + preprocessing of algo 4.11)
Size: $O(n)$

Lemma 3.5. Planar graphs have at most 3n - 6 edges

Def. 3.8 (**Bisector**)**.** The bisector of two points p and p' is the line orthogonal to the segment conv{p,p'} through its midpoint $\frac{p+p'}{2}$

3.4 Trapezoidal Decomposition

3.4.1 1-Dimension

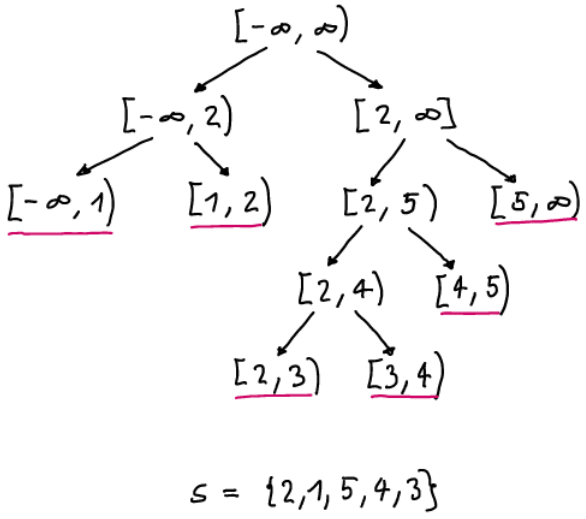
In 1 Dimension a more fitting name would be interval Decomposition.
A set S of n real numbers $(a_1..a_n)$ partitions R into n+1 intervals: $[-\infty, a_1), [a_1, a_2), ..., [a_n, \infty)$

Goal: Given S (not sorted), find interval that contains point q.
How: We could just sort the the $(a_1..a_n)$ but we want to give an other approach that scales to higher dimensions:

We use **Randomized incremental construction** to build a point location structure called the **history graph**:

Algorithm 3.9 (**Building History Graph**)**.**
- For each number s_i do:
- Locating the interval containing s_i, start in the root I_0, and whenever an interval still has children, continue with the one containing s_i
- Append two new children to the interval found

Runtime: $\sum_{i=1}^{n} \log(i) = O(n \log n)$ (log(i) because of below)



Now how much time do we really need to locate a query point q in this structure?
Its proportional to the number of times the interval containing q, changes.

$$x_i := \begin{cases} 1 & \text{interval containing q, changes by inserting } s_i \\ 0 & \text{otherwise} \end{cases}$$

$$x_0 = 1, x_1 = 1$$

X := number of times the interval containing q, changes

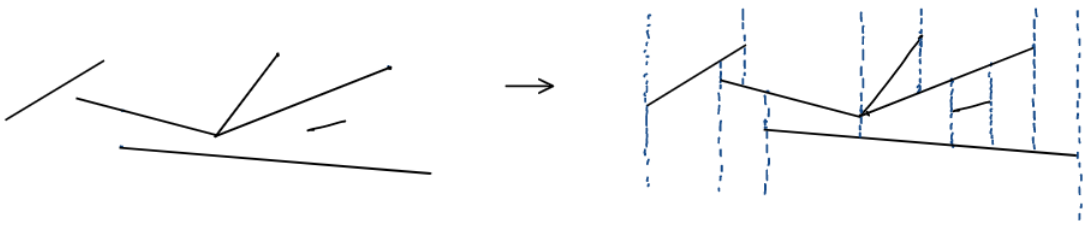
$$E[X] = E[\sum_{i=0}^{n} x_i] = \sum_{i=0}^{n} E[x_i] = 1 + 1 + \sum_{i=2}^{n} E[x_i] = 2 + \sum_{i=2}^{n} Pr[x_i = 1] \overset{(1)}{\leq} 2 + \sum_{i=2}^{n} \frac{2}{i} = 2H_n = O(\log n)$$

(1): To evaluate $Pr[x_i = 1]$ we use the **Backward-Trick:**
Instead of looking in the forward order: inserting s_i, we look at the backwards order: deleting s_i. The probability of deleting exactly s_i is the same as the probability of s_i ending up in this position at the insertion.

In summary we have a structure with expected query time $O(\log n)$ which is built in $O(n \log n)$

3.4.2 2-Dimension (The segment above)

Given: Set S on n segments that are non-crossing and in general position (no two segment endpoints have the same x-coordinate).
Goal: Preprocess S so that any query point $q \in R^2$ the segment above(q) above q can be computed quickly
Similarly to the 1-dimensional case we want to use point location. Hence we want to partition R^2 into cells (trapezoids), where each point in a trapezoid has the same segment above. Such a decomposition would look like this:



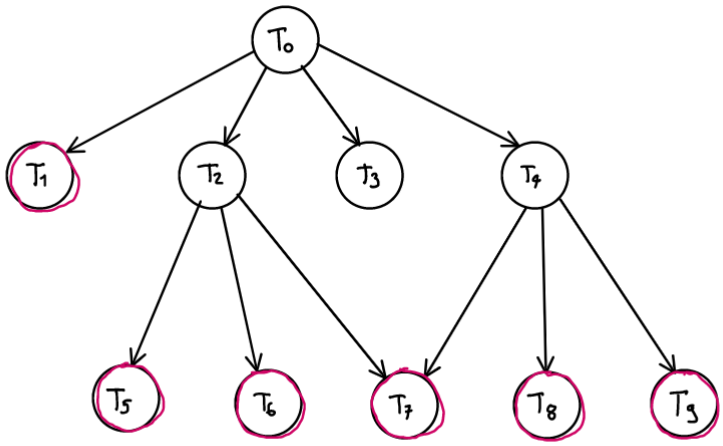
Note that we don't actually compute this. Its just an illustration what we will do later (history graph).

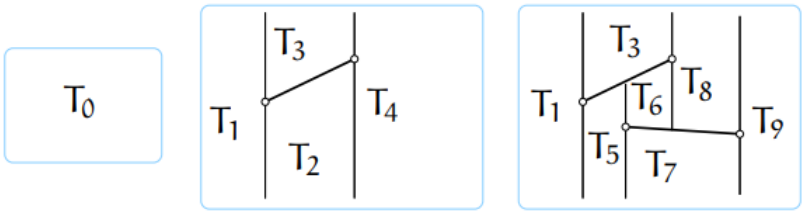
In such a structure we only need to determine the trapezoid to determine the segment above. In order to locate a query point $q \in R^2$ in the trapezoid decomposition ($\mathcal{T}(S)$) we again build a point location structure that we again call history graph using RIC. Since we have O(n) trapezoids the time complexity stays the same.

Algorithm 3.10 (Building History Graph 2D).

- Insert one segment s_i after the other:
- locate one endpoint p of segment s_i
- Starting from p, "walk" along s_i from one trapezoid to the next. All encountered trapezoids are destroyed and new ones are being generated.

Runtime: $O(n \log n)$ (why exactly is not said)





Satz 3.1. An edge in the history graph is drawn from T to T' if:

- $T \in \mathcal{T}(S_{i-1}) \setminus \mathcal{T}(S_i)$
- $T' \in \mathcal{T}(S_i) \setminus \mathcal{T}(S_{i-1})$
- $T \cap T' \neq \emptyset$

are all fulfilled.

So we have preprocessed this history graph. But can we really locate q in the desired $O(\log n)$ time in the history graph. In order to do so, we need, that:

(i) Size of history Graph (vertices and edges) stays linear, hence O(n)

(ii) Time from parent node to the desired child is Constant

(iii) The length of the path from the root to the desired leaf is expected to be logarithmic

Lemma 3.6. Given $T \in \mathcal{T}(S_{i-1}) \setminus \mathcal{T}(S_i)$, the number of trapezoids in $\mathcal{T}(S_i)$ overlapping T is at most 4
In other words: we have at most 4 outgoing edges from each node.

Proof: trivial

This means that we proceed in constant time to the next node. Hence we solved (ii)

Also the graph is linear in the number of its vertices (i.e. trapezoids), since there are at most four times as many edges. The number of trapezoids is in O(n) (Lemma 4.8). Hence the graph is linear to n. Hence we solved (i).

Lemma 3.7. Given $T \in \mathcal{T}(S)$, there exists a set $S' \subseteq S$ of at most 4 segments, such that $T \in \mathcal{T}(S')$

Proof: If $T \in \mathcal{T}(S)$, there are at most 4 segments in S whose removal lets T disappear (in a backward process)

With the backwards analysis we see: Pr[trapezoidal changes in step i]$\leq \frac{4}{i}$
The length of the path to q is proportional to the number of times the trapezoidal containing q, changes.
Pr[number of times the trapezoidal containing q, changes] $= \sum_{i=1}^{n} \frac{4}{i} = 4H_n = O(\log n)$
Hence we also solved (iii)

Lemma 3.8. $|\mathcal{T}(S)| = O(n)$, where n is the number of segments.

Proof: We use the Backward trick again together with Induction:

Base case (n=0): $|\mathcal{T}(S_0)| = 0 = n = O(n)$

Induction hypothesis: $|\mathcal{T}(S_i)| = O(i)$

Induction step (i → i+1): First note:
(1) After I.H we have O(i) trapezoids in $\mathcal{T}(S_i)$
(2) Any trapezoid $T \in \mathcal{T}(S_{i+1})$ disappears (backward process) because of removal of random segment in S_{i+1} with probability at most $\frac{1}{i+1}$.(A trapezoid is defined by at most 4 segments)

Because of (2) and (1) the expected number of trapezoids removed in the backward process from S_{i+1} to S_i is at most $\frac{4}{i} \cdot O(i) = O(1)$
Hence the overall expected number of trapezoids ever removed in the backwards process from $\mathcal{T}(S_{i+1})$ is: (i+1)·O(1) = O(i+1)
Since the number of ever removed trapezoids is equal to the number of trapezoids every added, the proof is complete.

Remark: We can prove this also with the use of planar graphs and plane embedding.

Since we showed (i),(ii) and (iii) are fulfilled we can locate q in $O(\log n)$, which gives us the algorithm for finding the segment above:

Algorithm 3.11 (**Segment above**).
Preprocessing: Compute History Graph 2D

- Find the trapezoid containing q in the history Graph
- Return the segment associated with that trapezoid

Runtime: $O(\log n)$
Preprocessing: $O(n \log n)$

4 Linear Programming

4.1 Basic Setting

Given a system of linear equations and non-strict inequalities as well as a vector $c \in R^n$, we want to find an $x^* \in R^n$, that fulfills the linear system and maximizes c^Tx*.
Notethat :c$^T x = c_1x_1 + c_2x_2 + ... + c_nx_n$

The function $c^T x$ is called the **objective function**.
The linear equations and non-strict inequalities are called the **constraints**.
If x fulfills all constraints in the linear system, we call x a **feasible solution**.
If x is feasible and the optimal solution, we call x the **optimum**.
If an LP has no feasible solution, we call it **infeasible**
If an LP is feasible but not an optimal solution, we call it **unbounded**

Possible set of solution of the LP: 1,0 or ∞ solutions

Satz 4.1. Every linear system can be translated to:

$$\text{maximize } c^T x \text{ subject to } Ax \leq b$$

$$\text{maximize } c^T x \text{ subject to } Ax = b, x \geq 0$$

For the first point, we achieved this by changing equations to two inequalities (each direction), and when needed we reverse the direction of the inequalities by changing the sign.

A linear program is efficiently solvable, both in theory and in practice.

4.2 Direct applications

Here we formulate a problem as a linear program. This can not be done with any problem, but for some it works:

- Diet problem
- Ice cream all year round
- Line fitting
- Cutting paper rolls

We often have to apply little tricks to get a linear system. For example exchanging $|x - y|$ with $x - y$ and some constraints.

4.3 Geometry of LP

We first denote some geometrics:

- A **hyperplan** in R^n is an affine subspace of dimension n-1. In other words it is a set of the form: $\{x \in R^n : a_1x_1 + ... + a_nx_n = b\}$, where $a_1, ..., a_n$ are not all 0.
- A (closed) **halfspace** in R^n is a set of the from $\{x \in R^n : a_1x_1 + ... + a_nx_n \leq b\}$. again wit at least one a_i nonzero.
- A **convex polyhedron** in R^n is the intersection of finitely many closed half-spaces.
- A convex **polytope** in R^n is a bounded convex polyhedron. (Named polygon in R^2, polytope in R^3).

The set of feasible solution of an LP is defined by a convex polyhedron.
A **basic feasible solution (BFS)** of a linear program is a feasible solution x for which n linearly independent constrains hold with equality.

Since we have linearly independent constraints, only one solution has to exist. From the m constraints choosing n linearly independent constraints yields at most $\binom{n}{m}$ possibilities. Therefore at most $\binom{n}{m}$ BFS exist. Note that

in order for such a solution be feasible it also has to fulfill the other m-n constraints. We know without proof that:

BFS's correspond to vertices of the polyhedron

Satz 4.2 (**Theorem 4.1**). If the following holds:

- LP is of the form: maximize $c^T x$ subject to Ax = b, x $\leq$ 0
- $P \neq \emptyset$, where $P \subseteq R^n$ is a convex polyhedron of all feasible solutions.
- Objective function $c^T x$ is bounded from above on P.

Then there exists a basic feasible solution that is optimal.

Hence there always exists a feasible solution. Except if there are no solutions at all or $c^T x$ is unbounded.

4.4 Bounds on solutions

We want to show that we can find solutions whose size is polynomial (actually linear) in the size of the LP. In order to speak of theoretically efficient, i.e. polynomial algorithms, we need to measure the **size** of the given linear program. The size of a LP is the number of bits to write all coefficients.
Since an LP L can be written as "maximize $c^T x$ subject to $Ax \leq b$" we have: <L>:= <A> + <b> + <c>

Satz 4.3 (**Theorem 4.2**). If an LP (in educational form) with rational coefficients has a feasible solution, then it has a feasible solution $\tilde{x} \in R$ with for every j: $<\tilde{x_j}>$ = O(<L>)
This also holds form optimal solutions.

In order to prove this theorem, we needed the following lemma:

Lemma 4.1. For a rational matrix A, <det(A)> = O(<A>)

4.5 Duality and the Farkas lemma

We want to show, that its easy to:

(1) Convince someone that you have solved Ax=b

(2) Convince someone that Ax=b is unsolvable

Remember that (2) is not easy for an NP-hard problem. So in the following we give methods to have easy certificates for unsolvability of systems of linear inequalities (2).

Lemma 4.2. A system $Ax = b$ of linear equations is unsolvable iff
there exists y such that $A^T y = 0$ and $b^T y = 1$

Lemma 4.3 (**Farkas lemma I**). A system Ax$\leq$b of linear inequalities is unsolvable iff
there exists $y \geq 0$ such that $A^T y = 0$ and $b^T y < 0$

Lemma 4.4 (**Farkas lemma II**). A system $Ax = b$ of linear equations has no nonnegative solution iff
there exists y such that $A^T y \geq 0$ and $b^T y < 0$

Lemma 4.5 (**Farkas lemma III**). A system $Ax \leq b$ of linear inequalities has no nonnegative solution iff
there exists $y \geq 0$ such that $A^T y \geq 0$ and $b^T y < 0$

Note that all Farkas lemma imply each other

We know want to show that the same for the question "maximize $c^T x$ subject to Ax$\leq$b and $x \geq 0$. This question is the same as "Is the optimal value of the LP (P) greater or equal to some given number γ.
As before we want to show:

(1) Convince someone that there exists an $\tilde{x}$ with $c^T\tilde{x} \geq \gamma$

(2) Convince someone that there exists no $\tilde{x}$ with $c^T\tilde{x} \geq \gamma$

For (1) we just give an $\tilde{x}$ such that $c^T\tilde{x} \geq \gamma$.
For (2) we introduce the duality:

Satz 4.4 (**Strong Duality Theorem** - Theorem 4.6)**.** Let:
(P) maximize $c^T x$ subject to $Ax \leq b$ and $x \geq 0$
(D) minimize $b^T y$ subject to $A^T y \leq c$ and $y \geq 0$
Then one of the following possibilities occurs:

- Neither (P) nor (D) has a feasible solution.
- (P) is unbounded and (D) has no feasible solution.
- (P) has no feasible solution and (D) is unbounded.
- Both (P) and (D) have feasible solution. Then both have an optimal solution, and if x^* is an optimal solution of (P) and y^* is an optimal solution of (D), then

$$c^T x^* = b^T y^*$$

Which ever case happens we can easily give a NO-certificate for (2). Specifically in the last case we give y.

4.6 The ellipsoid method

The ellipsoid method solves linear programs in provably polynomial time: O(p(<L>)).
However in practise we use either the interior point method or the simplex method.

How to find an optimal solution to an LP P in polynomial time:

1.) Transforming the LP P, into a dual LP D. Where it holds that: Any feasible solution of D is an optimal solution for P.

2.) Find a feasible solution for D = $Ax \leq b, x \geq 0, c^T x \geq b^T y, A^T y \geq c, y \geq 0$. (How? →Blackbox)

Note that instead of this we can have an algorithm that uses a Blackbox (returning that an LP is feasible or not) combined with a binary search (ex.3 in kw45). But in both ways we have this Blackbox that finds us a feasible solution of an LP. This Blackbox is our ellipsoid method.

How the ellipsoid method works:
Remember that we only need to find one feasible solution and that the set of all feasible solution build the convex polyhedron. Hence the ellipsoid method just tries to locate that polyhedron P and pick an $y \in P$.
To do that we firstly show how to do that for a relaxed Problem, and then show how you can transform the general case to the relaxed problem.

Def. 4.1. Relaxed Problem 4.7). Together with the matrix A and vector b we are given rational numbers $R > \epsilon > 0$. We assume that the polyhedron P is contained in the ball B(0,R) centered at 0 with radius R. If P contains a ball of radius ϵ, then the algorithm has to return a point $y \in P$. However, if P contains no ball of radius ϵ, then the algorithm may return either some $y \in P$, or the answer NO SOLUTION.

Then the ellipsoid method solved it with the following algorithm:

Algorithm 4.1 (**Ellipsoid method**).

1.) Set k=0 and $E_0 = B(0, R)$

2.) Let s_k be the center of the current ellipsoid E_k. If s_k satisfies all inequalities of the system $Ax \leq b$, return s_k as solution; stop.

3.) Otherwise, choose an inequality of the system that is violated by s_k. Let is be the i-th inequality; so we have $a_i^T s_k > b$. Compute E_{k+1} as an ellipsoid containing the set $E_k \cap \{x \in R^n : a_i^T x \leq b\}$ and such that
volE_{k+1} (volume of E_{k+1} is substantially smaller than volE_k.

4.) If volE_{k+1} is smaller than the volume of a ball of radius ϵ, return NO SOLUTION; stop. Otherwise, increase k by 1 and continue with step 2.

No in order to have proved that we can find an optimal solution for every linear program we still need to show 3 things:

1.) Show that we only need polynomial many iterations

2.) Show that every iteration can be implemented in polynomial time (Issue A)

3.) Show that we can transform the general case into the relaxed problem 4.7, in polynomial time (Issue B)

To solve 1.) we just use the following lemma, which bounds the number of iterations.

Lemma 4.6. Let E be an n-dimensional ellipsoid in R^n with center s, and let H be a closed half-space whose interior does not contain s. Then there exists an ellipsoid E', given by an explicit formula that contains $E \cap H$ and satisfies:

$$\text{vol}E' \leq \rho \cdot \text{vol}E, \qquad \text{where } \rho = \rho(n) := e^{-1/(2n+2)}$$

To prove the lemma we used:

- an n-dimensional unit ball is a set of the form B = $\{x \in R^n : ||x|| \leq 1$
- An n-dimensional ellipsoid is a set of the form E = $\{\text{Mx} + \text{s} : \text{x} \in B^n\}$
- vol(E)·|det(M)|

To solve 2.) we only need to get an other ellipsoid by explicit formula. Sadly it would contain roots in the new Matrix M, hence we need to argue that we can round results and still get a correct ellipsoid. This is tedious and wasn't covered.

To solve 3.) Let $Ax \leq b$ be the system we want to check for feasibility. (Let φ = <A> + <b>).
Then we transform the old system to a new system $\hat{A}x \leq \hat{b}$ that fulfills:

(i) $\hat{P} \subset B(0, R)$ where <R> is polynomial in φ

(ii) If $P \neq \emptyset$, then $\hat{P}$ contains an ϵ-ball with $\epsilon > 0$ and $<\epsilon>$ polynomial in φ

(iii) If $P = \emptyset$, then $\hat{P} = \emptyset$

The new system $\hat{A}x \leq \hat{b}$ is: $\quad Ax \leq b + \eta 1, \quad -K - \eta \leq x_j \leq K + \eta$

This system fulfills (i)-(iii). To see that we used Theorem 4.2 and the following lemma:

Lemma 4.7. Let $Ax \leq b$ be a system of inequalities with rational coefficients and encoding size φ, let P be its polyhedron of feasible solutions, and let P_η be the polyhedron of the system $Ax \leq b + \eta 1$. For $\eta = 2^{-C_2 \varphi}$, with a sufficiently large constant C_2 the following hold:

(a) If $P \neq \emptyset$, then P_η contains an ϵ-ball for $\epsilon = \eta / 2^\varphi$

(b) If $P = \emptyset$, then $P_\eta = \emptyset$

Note that we choose $R = (K + \eta)\sqrt{n}$ and $\epsilon = \eta / 2^\varphi$. And the encoding size will be at most O(nφ).

Lastly we note, that in order for the ellipsoid method to run in polynomial time, we need a **polynomial separation oracle**, that tells if,and if so return at least one, constraint that got violated.

4.7 Traveling Salesman

Def. 4.2 (**Tour, Hamilton cycle**). A tour τ for a graph $G = (V, E)$ with costs $c_e \in R, e \in E$ is a subset E_τ of E, such that:

(i) The graph (V, E_τ) is connected

(ii) every vertex is incident to exactly two edges in E_τ

Def. 4.3 (**Boundary**).

$$\delta(S) := \{e \in E : |e \cap S| = 1\} = \text{edge set of cut (S, } V \setminus S)$$

$$\delta(v) := \text{set of edges incident to v}$$

We construct the following linear program (**Subtour LP**):

- min $c^T x$
- subject to:

$$\begin{aligned} &\textstyle\sum_{e\in\delta(v)} x_e = 2 && \text{for all } v \in V \\ &\textstyle\sum_{e\in\delta(S)} x_e \geq 2 && \text{for all } S \subseteq V \text{ with } \emptyset \neq S \neq V \\ &1 \geq x_e \geq 0, && \text{for all } e \in E \end{aligned}$$

We note that for $\tilde{x}$:= optimal solution for this LP and the triangle inequality:

$$c^T \tilde{x} \leq \text{OPT}_{tour} \leq \frac{3}{2} c^T \tilde{x}$$

We also note that for this problem exponentially many constraints exist, but with the min-cut algorithm we have a separation oracle that runs in polynomial time.

4.8 Minimum Spanning Tree

Def. 4.4. A spanning tree of graph G = (V,E) is a subgraph T = (V,E'), $E' \subseteq E$, such that:

(i) connected

(ii) no cycles

We construct the following linear program (**Loose Spanning Tree**):

- min $c^T x$
- subject to:

$$\begin{aligned} &\textstyle\sum_{e\in E} x_e = n - 1 \\ &\textstyle\sum_{e\in\delta(S)} x_e \geq 1 && \text{for all } S \subseteq V \text{ with } \emptyset \neq S \neq V \\ &1 \geq x_e \geq 0, && \text{for all } e \in E \end{aligned}$$

After a closet look, we will note that for some Graphs the loose spanning tree form our LP has weight at most half of the actual MST.
This comes from the fact that we can assign edges an $x_e = 1$, that have 0 cost, in order to manipulate our subject to functions.

However the following LP does better (**Tight linear Spanning Tree**):

- min $c^T x$
- subject to:

$$\begin{aligned} &\textstyle\sum_{e\in E} x_e = n - 1 \\ &\textstyle\sum_{e\in E\cap\binom{S}{2}} x_e \leq |S| - 1 && \text{for all } S \subseteq V \text{ with } \emptyset \neq S \neq V \\ &1 \geq x_e \geq 0, && \text{for all } e \in E \end{aligned}$$

Satz 4.5 (**Theorem 4.11**). Every basic feasible solution of the LP above is integral. Therefore the value is equal to the cost of the MST for every cost vector c.

4.9 Back to the Subtour LP

We want to figure out how big the integrality gap is.

Def. 4.5 (**Integrality gap**). Let:
x^* be the optimal integral solution
$\tilde{x}$ be the optimal solution of the LP

$$\text{Integrality gap} = \frac{c^T x^*}{c^T \tilde{x}}$$

Not that the gap is always ≥ 1.

Def. 4.6 (**Walk**). A walk is a sequence of (not necessarily distinct) vertices $(v_0, v_1, ..., v_l)$ with consecutive vertices adjacent in G. The walk is closed if $v_0 = v_l$. The walk has length l.

Lemma 4.8. Let the graphic metric c on $\binom{V}{2}$ be induced by the connected graph G = (V,E). Then the optimal tour of G' = (V, $\binom{V}{2}$) with costs c, equals the length of the shortest closed walk in G visiting all vertices at least once.

Note that the graph G' = (V, $\binom{V}{2}$) is graph G but you can also use non present edges with cost according to the triangle inequality.

Proof:
We prove this lemma by showing:

(1) optimal tour in G' $\leq$ cost of shortest closest walk in G of all vertices

(2) optimal tour in G' $\geq$ cost of shortest closest walk in G of all vertices

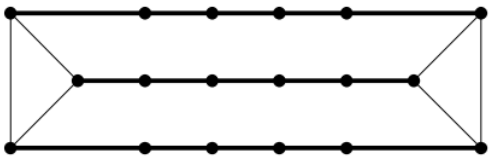
(1): Given a walk in G we can create a tour in G', which has at most the same cost
(2): Given a tour in G', we can create a walk in G, that has at most the same cost.

Lemma 4.9. Let n:=3k (the number of vertices of G_k and G'_k):

(i) The Subtour LP on G'_k has a feasible solution with value n

(ii) Every closed walk in G_k visiting all vertices has length at least $\frac{4}{3}n - O(1)$ and therefore every tour in G'_k has cost at least $\frac{4}{3}n - O(1)$.

The Graph G_6 looks like this:



Proof:

(i) We choose $x_e = 1/2$ for the six edges in the 3-cliques and $x_e = 1$ for the path edges (from G). For all other edges in G'_k we choose $x_e = 0$. You can easily see that that way the constraints hold and that $c^T x = \sum_e x_e = n$.

(ii) A formal prove uses that on cuts we need an even number of edges that cross it, since we have a tour. But informally we have to walk one of the 3 paths one more time in order to get to the beginning. Hence we have n + $\frac{n}{3} = \frac{4}{3}n$

From the lemma we can conclude that for some graphs (namely G_k) the integrality gap is at least: $\frac{(ii)}{(i)} = \frac{\frac{4}{3}n}{n} = \frac{4}{3}$. But for some other graphs it may be worse. Is there an upper limit?

4.10 Subtour LP versus Tight Spanning Tree LP

We now want to check if there is an upper bound. For that we will relate our LP at hand with the tight spanning tree LP.
Why? Because we know that every MST can be extended to a tour with cost at most twice the MST (double all edges). But this was for not LP's and hence we want to translate that thought onto our subtour LP.

LP subtour := optimal solution for the Subtour LP
comb. subtour := optimal combinatorial tour (ground truth, best possible in the Graph).
OPT_{MST} := optimal combinatorial MST in the Graph

Then we can derive:

$$\frac{\text{comb. subtour}}{\text{LP subtour}} \overset{(1)}{\leq} \frac{2 \cdot \text{OPT}_{MST}}{\text{LP subtour}} \overset{(2)}{\leq} \frac{2 \cdot \text{OPT}_{MST}}{\frac{n}{n-1}\text{OPT}_{MST}} \overset{(3)}{\leq} \frac{2 \cdot \text{OPT}_{MST}}{\text{OPT}_{MST}} = 2$$

(1) From AW. Every tour is maximal 2 times the MST (double every edge)
(2) Satz 4.6
(3) simple analysis

Hence we have found an upper bound, namely 2. We now just need to introduce and prove Satz 4.6. For this we need the following lemma:

Lemma 4.10. For a given graph G = (V,E), if $x \in R^E$ is a feasible solution of the Subtour LP, then $\frac{n-1}{n}x$ is a feasible solution of the Tight Spanning Tree LP.

Proof:

(i) $\sum_{e \in E} \frac{n-1}{n} x_e = \frac{n-1}{n} \sum_{e \in E} = \frac{n-1}{n} \frac{1}{2} \sum_{v \in V} \sum_{e \in \delta(v)} x_e = \frac{n-1}{n} \frac{1}{2} 2n = n - 1$

(ii) $\sum_{e \in E \cap \binom{S}{2}} x_e = \frac{1}{2}(\sum_{v \in S} \sum_{e \in \delta(v)} x_e - \sum_{e \in (S)} x_e) \leq \frac{1}{2}(|S|2 - 2) = |S| - 1$

(iii) $1 \geq x_e \geq 0 \Longrightarrow 1 \geq \frac{n-1}{n} x_e \geq 0$

Satz 4.6. Given a graph G, if $\tilde{x}$ is an optimal solution of the Subtour LP and OPT_{MST} is the cost of the minimum spanning tree, then $c^T\tilde{x} \geq \frac{n}{n-1}\text{OPT}_{MST}$

Proof:
We have $c^T\tilde{x}$ as the optimal solution to the subtour LP. Hence $\tilde{x}$ is also feasible solution to the subtour LP. With Lemma 4.10: $\frac{n-1}{n}\tilde{x}$ is a feasible solution of the tight spanning Tree LP. Since we minimize $c^T x$ in the tight spanning tree LP we have that $c^T \frac{n-1}{n}\tilde{x}$ is at least the optimal solution of the tight spanning tree LP. Since the optimal solution of the tight spanning tree LP is integral it is also the combinatorial best solution hence OPT_{MST}. With that we have:

$$c^T \frac{n-1}{n}\tilde{x} \leq \text{OPT}_{MST}$$

Which is nothing else than:

$$c^T\tilde{x} \leq \frac{n}{n-1}\text{OPT}_{MST}$$

Satz 4.7 (Theorem 4.17)**.** If G is a complete graph with costs satisfying the triangle inequality, then the integrality gap for the subtour LP is $\in [\frac{4}{3}, \frac{3}{2}]$

So why $\frac{3}{2}$ and not the 2 form above. Well we can do better: We know that:

$$\text{OPT}_{\text{tour}} \leq \text{OPT}_{\text{MST}} + \text{OPT}_{\text{matching(U)}}$$

You can show that:

(i) $\text{OPT}_{\text{tour}} \leq c^T\tilde{x}$

(ii) $\text{OPT}_{\text{matching(U)}} \leq \frac{1}{2}c^T\tilde{x}$:

And with that we get:

$$\frac{\text{comb. subtour}}{c^T \tilde{x}} \leq \frac{\text{OPT}_{\text{MST}} + \text{OPT}_{\text{matching(U)}}}{c^T \tilde{x}} \leq \frac{3}{2}$$

Note that this theorem holds for triangle inequality TSP. If we induce the graphic metric (weight = min distance) we have some even better upper bound, namely 1.4.

5 Randomized Algebraic Algorithms

In this chapter we consider algorithms about probabilistic checking.

5.1 Checking Matrix Multiplication

We want to check if $A \cdot B = C$, where A, B, C are $n * n$ matrices.

Algorithm 5.1 (**Naive Checker**).
Randomly check l entries of C by calculating the exact result of that entry.

Runtime: Calculating 1 entry: O(n). Hence for l entries: O(nl).
Success Prob.: Pr[Detect a wrong matrix multiplication] $\geq \frac{l}{n^2}$

This is pretty low success probability. We want to improve:

Algorithm 5.2 (**Freivald's Algorithm**).
1.) Pick random vector $x \in \{0,1\}^n$

2.) Compute Cx and ABx

3.) If the results agree return YES, otherwise NO.

Runtime: Since we can parenthesize A(Bx) we get: $O(n^2)$
Success prob.: Pr[Detect a wrong matrix multiplication] $\geq \frac{1}{2}$

Proof:

$$\begin{aligned} Pr[\text{Detect a wrong matrix multiplication}] &= Pr[Cx \neq ABx | C \neq AB] \\ &= Pr[(C-AB)x \neq 0 | C \neq AB] \\ &= Pr[Dx \neq 0 | D \neq 0] \\ &\overset{(1)}{=} 1 - \underbrace{Pr[Dx = 0 | D \neq 0]}_{\leq \frac{1}{2}} \\ &\geq 1/2 \end{aligned} \tag{3}$$

(1): With (y:= Dx) we have: $Pr[y = 0 | D \neq 0] \leq Pr[y_i = 0 | D_{ij} \neq 0]$. This holds because D needs to be nonzero at least for one element. Let that element be at D_{ij}. In order for y to be 0 every element of y, hence also y_i must be 0. Its also less than, because we don't even consider that the other y_j need also to be nonzero.
$Pr[y_i = 0 | D_{ij} \neq 0] \leq \frac{1}{2}$. This holds because $y_i = d_{i1}x_1 + ... + d_{in}x_n = d_{ij}x_j + S$.
Whatever S is, in one case ($x_n = 1$) we add a nonzero number to S and in the other case ($x_j = 0$) we don't. Therefore we get 2 different numbers, hence not both can be 0. Hence we can only hope that we get a zero in say the first case (we probably wont get a 0 in both cases, but certainly not in both).
Since both cases have chance of 1/2 we only have a chance of less that 1/2 to get a zero.

5.2 Is a Polynomial Identically Zero

This is not an easy task if we only have a Blackbox that evaluates our Polynom. (You dont know the roots). However we know that a polynomial with one variable has either:

- 0 everywhere
- 0 at d points. Where these points are the roots and d the degree.

Lemma 5.1. Polynomial p with one variable with d+1 points equal to 0, then p is identically 0.

We will now see that this scales to multivariable polynomials:

The following is a multivariable polynomial:

$$3x_1^6x_2^2x_5 - x_3^2x_4^5 + 2x_5^2 - 17$$

- 5 Varibales
- Degree 9

We allow the coefficients of the polynomial to be from an arbitrary field $\mathbb{F}$ (Often $\mathbb{F} = \mathbb{Q}$).
We denote $\mathbb{F}[x_1, x_2, ..., x_n]$ as the ring of all polynomials in the variables $x_1, ..., x_n$ with coefficients in $\mathbb{F}$.

Lemma 5.2 (**Schwartz-Zippel theorem**). Let $P(x_1, ..., x_n) \in \mathrm{F}[x_1, ..., x_n]$ be a (nonzero) polynomial of degree $d \geq 0$, ans so let $S \in \mathbb{F}$ be a finite set. Then the number of n-tuples $(r_1, ..., r_n) \in S^n$ with $p(r_1, ..., r_n) = 0$ is at most $d|S|^{n-1}$.
In other words, if $r_1, ..., r_n \in S$ are chosen independently and uniformly at random, then the probability of $p(r_1, ..., r_n) = 0$ is at most $\frac{d}{|S|}$.

In my words: if $Pr[p \neq 0]$ then $Pr[p(r) = 0] \geq \frac{d}{|S|}$
(r is random point, and $p \neq 0$ means, that not all coefficients are 0)

This is more or less equivalent to the lemma 5.1 but just for polynomials with multiple variables. Intuitively this also makes sense. Because if we don't have the 0 polynomial then its only 0 at the roots, of which only d exists. So from all points ($= |S|$) we can only choose d, to get 0.

The Proof uses induction on the number of variables.

5.3 Testing for Perfect Bipartite Matchings

We call Perfect Matchings: PM.
Actually computing a maximum matching can be done deterministic in $O(m\sqrt{n})$.
However we will see that we can do much better with randomization:

Algorithm 5.3 (**Find bipartite perfect Matching**).

1.) Take a random edge e = (u,v) in G. (No edges anymore: go to step 4)

2.) If G' = G-{u,v} has a PM (run Algo 5.4) then add edge e to PM M, and repeat with step 1 on G'

3.) otherwise repeat step 1 with on G-e

4.) return M

For this algo to work we now need to figure out how we can test a Graph for a bipartite Matching:

Let G be bipartite with $V = \{u_1, ..., u_n\} \cup \{v_1, ..., v_n\}$.
Let S_n be the set of all permutations of $\{1, ..., n\}$.
Let $\pi \in S_n$.
A perfect Matching PM = $\{\{u_1, v_{\pi(1)}\}, ..., \{u_n, v_{\pi(n)}\}\}$

Algorithm 5.4 (**Test bipartite Graph for perfect Matching**).

1.) construct matrix A:= $\begin{cases} \mathrm{x}_{ij} & \text{if } u_i, v_j \in E(G) \\ 0 & \text{otherwise} \end{cases}$

2.) compute det(A)

3.) if det(A) $\neq 0$: return "YES"

4.) else return "NO"

Note that this algo need to be repeated to work, because:

Satz 5.1. (Theorem)

(i) If $\nexists$PM, then Pr["NO"] = 1

(ii) If ∃PM, then Pr["YES"] $\geq \frac{1}{2}$

Proof:

(i) $\nexists$PM $\overset{\text{lemma 5.3}}{\Rightarrow}$ $\det(A) = 0$

(ii) Since x_{ij} are variables, det(A) is the sum of polynomials of degree at most n. Hence det(A) is itself a polynomial with degree at most n. We now define a set S of values that the variables live in (Most often GP(p) with: 2n $\leq$ p $\leq$ 4n). Thanks to lemma 5.4 we have that the polynomial det(A)$\neq$ 0. Therefore we can apply Schwartz-Zippel to get: $Pr[det(A) \neq 0] \geq \frac{deg(det(A))}{|S|} \geq \frac{n}{2n} = \frac{1}{2}$.
Hence we output YES (since $det(A) \neq 0$)

Satz 5.2.

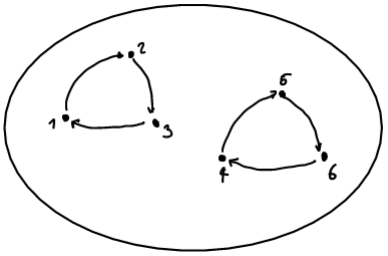
$$det(A) = \sum_{\substack{\pi \text{permutation} \\ \pi:[n]\to[n]}} sign(\pi) \underbrace{A_{1\pi(1)} \cdot A_{2\pi(2)} \cdot \ldots \cdot A_{n\pi(n)}}_{A_\pi}$$

Lemma 5.3. ∃PM $\iff$ polynomial det(A) is not identically zero.

In the proof we use: $A_\pi \neq 0 \iff \{\{u_i, v_{\pi(i)}\} | i \in [n]\}$ is PM in G

5.4 Perfect Matchings in General Graphs

Sadly we can't do the same as for bipartite Graphs. This is because lemma 5.3 doesn't work anymore:



$$\pi : 1 \to 2, 2 \to 3, 3 \to 1, 4 \to 5, 5 \to 6, 6 \to 4$$

A_π is a nonzero term and hence det(A) $\neq$ 0 but G must not necessarily have a PM. (drawn edges could be the only ones in G) (violates lemma 5.3)

To solve this issues we do the same as for bipartite graphs but we just use a different Matrix (**Tutte Matrix**):

$$B := \begin{cases} \mathrm{x}_{ij} & \text{if } u_i, v_j \in E(G) \text{ and } i < j \\ \text{-x}_{ij} & \text{if } u_i, v_j \in E(G) \text{ and } j < i \\ 0 & \text{otherwise} \end{cases}$$

Since we use the same algorithm, we want to prove Satz 5.1 again. To do that we need to prove:

Lemma 5.4. ∃PM $\iff$ polynomial det(B) is not identically zero.

Proof:

$\Longrightarrow$

$\Longleftarrow$

6 Parallel Algorithms

We often use models to talk about parallel algorithms. We often use one of these two models:

Def. 6.1 (PRAM-model). .
PRAM = Parallel Random Access Machines = Number of RAM machines, who can access shared memory.

Def. 6.2 (MPC-model). .
MPC = Massive Parallel computation
A more modern model for parallelism.

6.1 Warm Up: Adding Two Numbers

Carry-Ripple (how we intuitively do it) uses O(n) gates and even with many friends we wouldn't be faster. Note that we need O(1) many computations for each s_i (s_i i-th bit of the result/sum).
Remember that $s_i = a_i + b_i + c_{i-1}$

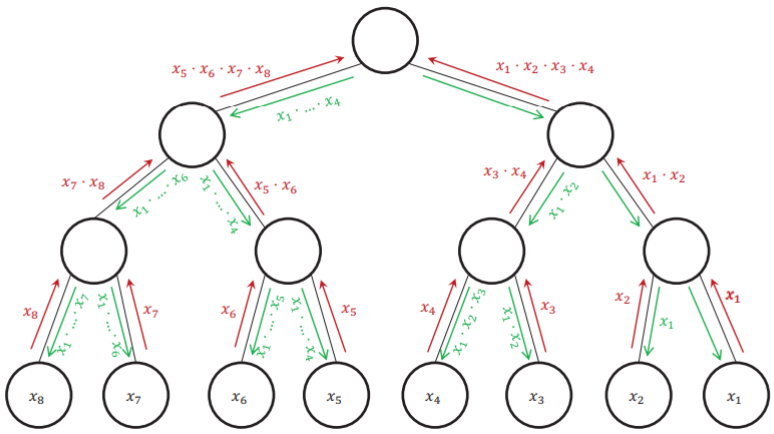
Carry look-ahead:
We compute the carry-bits more intelligent: $c_{i-1} = x_{i-1} * x_{i-2} * \cdots * x_1$. $(c_0 = k)$

$$x_i := \begin{cases} k & \text{if } a_i = 0 \wedge b_i = 0 \\ g & \text{if } a_i = 1 \wedge b_i = 1 \\ p & \text{otherwise} \end{cases}$$

*	k	g	p
k	k	k	k
g	g	g	g
p	k	g	p

Note that when we want to calculate $s_i = a_i + b_i + c_{i-1}$ we have that $c_{i-1} = k \stackrel{\wedge}{=} 0$ and $c_{i-1} = g \stackrel{\wedge}{=} 1$. Also note that we never have $c_{i-1} = p$.

Now for s_i we only need a_i, b_i and c_{i-1}. To get c_{i-1} we only have to have the product of $x_{i-1} * x_{i-2} * \cdots * x_1$, and from PP we know we can use a binary structure with an up and down pass in order to get the multiplication in O(log n) time to the right leaf:



Therefore we get an $O(\log n)$ runtime, because processor p_i can calculate s_i independent from all other. (Note that we do O(n) work).

6.2 Models and Basic Concepts

Circuit model and PRAM model:

6.2.1 Circuit-Model

Def. 6.3. The NC(i) class denotes the set of all decision problems that can be decided by a Boolean circuit with poly(n) gates of at most two inputs and depth at most $O(\log^i n)$, where n denotes the input size (= number of input bits to the circuit)

Def. 6.4. A problem is in RNC(k) if there exists a randomized circuit with depth $O(\log^k(n))$ and size poly(n) which outputs the correct sollution with probability at least 2/3.

Def. 6.5. The AC(i) class denotes the set of all decision problems that can be decided by a Boolean circuit with poly(n) gates of potentially unbounded fan-in an d depth at most $O(\log^i n)$.

Lemma 6.1.

$$NC(k) \subseteq AC(k) \subseteq NC(k+1)$$

Note that every circuit takes one time step time to complete.

Def. 6.6.

$$NC = \bigcup_i NC(i)$$

$$AC = \bigcup_i AC(i)$$

6.2.2 PRAM-Model

We have p PRAM processors with local memory and shared memory. For the shared memory we have 4 variations:

- EREW (Exclusive Read Exclusive Write)
- ERCW
- CREW
- CRCW (Concurrent Read Concurrent Write)

In each time step we can either write/read to local memory or to global. We also have classes. For example:

Def. 6.7. CRCW(k) denotes the set of decision problems that can be computed by the corresponding version of the PRAM-model (in this case CRCW) with poly(n) processors and in $O(\log^k n)$ time steps.

Lemma 6.2.

$$EREW(k) \subseteq CRCW(k) \subseteq EREW(k+1)$$

6.2.3 PRAM vs Circuits

PRAM can simulate Circuits and vice versa. We aslo see that

$$NC = EREW$$

(with $EREW = \bigcup_i EREW(i)$)

6.2.4 Basic Problems

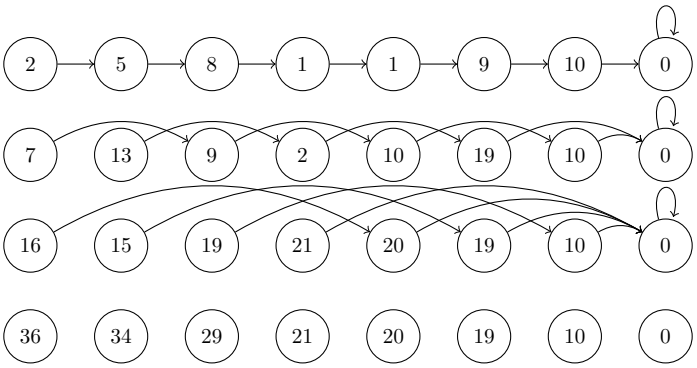
Parallel Prefix
Exactly what we did in the carry look-ahead.
$O(\log n)$ depth and $O(n)$ work

List Ranking
Input: Linked list represented by a content array c[1..n] and successor pointer array s[1..n].
Goal: Compute all suffix sums, from any starting point.
How: To do this we use O(log n) iterations and in each step we do:

(1) In parallel, for each $i \in \{1, ..., n\}$ set $c(i) = c(i) + c(s(i))$

(2) In parallel, for each $i \in \{1, ..., n\}$ set $s(i) = s(s(i))$ (= **Pointer Jumping**)

Here an example that uses 3 iterations (log(8)).



This algorithm uses O(log n) time and O(n log n) work.

6.2.5 Work-Efficient Parallel Algorithms

Def. 6.8. Depth of computation = Number of rounds
Work = Summation of the computations done over all rounds

Satz 6.1 (**Brent's Principle**). If an algorithm does x computations in total and has depth t, then using p processors, this algorithm can be run in: $\frac{x}{p} + t$ time.

Def. 6.9 (**work-efficient**). An algorithm is work-efficient if the amount of work in the parallel setting is proportional to the amount of work for the sequential counterpart.

6.3 Lists and trees

We already have an algo for list ranking that reacquires $O(n \log n)$ work. Now we want one that has only O(n) work. We realize that the only difference between "Parallel Prefix" and "List Ranking", is how the elements are stored (i.e. in an array and in linked list respectively). The problem with the linked list is that we can't do the **Binary tree trick** as in parallel prefix:
Reducing the problem to compute prefix sum on the n/2 elements at even positions on the array in O(1) time and O(n) work.

Sadly we can't do that in list ranking since an element doesn't know if its an even or odd element. Note that this even and odd could also be changed to something other. We just need to reduce that number of elements. And this is exactly the clue we needed. We do the **Subset S Trick:**
We

Algorithm 6.1 (**(List Ranking)**).

1.) From start linked list L we build a smaller list L' with the S-Trick. The elements are the elements from S but the value is the sum of: the elements starting (and including) itself till the right before the next element in S.

2.) If size of list L' is greater that $O(\frac{n}{\log n})$ repeat step 1.

3.) Otherwise apply the other list ranking algo on L'.

4.) Reinsert deleted elements in iteration i in step 1 and determine its prefix.

5.) Repeat step 4 till i = 1

Runtime: $O(\log n \cdot \log \log n)$
Work: $O(n)$

Proof of Runtime:
Fist we note that step 3 (which has $|L'| \leq O(\frac{n}{\log n})$) we need with the other algo: $O(\log(\frac{n}{\log n})) = O(\log n)$ time.
Hence we only need to show that for the recursively steps we only need $O(\log n \cdot \log\log n)$ time.
We note that we only need $O(\log\log n)$ recursive calls. Since if in each iteration $|L'| = |S| = c \cdot |L| \leq |L|$ we have that if x= iteration we need to fulfill:

$$\log n \leq c^x n$$

Which gets us that $x \leq O(-\log_c \log n) = O(\log\log n)$. Now in each iteration we need to:

(1) Choose the elements of S

(2) Compact the linked list and build the new linked list.

For (1) we apply the **Coin-Trick**: Each element tosses a coin, which lands on head or tail. Now all elements are in S except from those that have a head itself and its successor a tail. This can be done in O(1) time. The probability that $c < 1$ in $|L'| = |S| = c|L|$ is almost one (do it with c = 15/16 and chernoff bound.)
For (2): We use the Parallel-Prefix algo with initial values: $0 \in$ I, $1 \in$ S. We can do this because the numbering doesn't need to be monotonic. Afterwards its easy to delete the items from I and update the values from the values in I. (in small time)

6.3.1 The Euler Tour Technique

The tree is given as the adjacency lists for each vertex.

Rooting the Tree
Problem: Determine parent[v] for each vertex except the root r.

Algorithm 6.2 (**(Tree Rooting)**).

1.) Replace each tree-edge u,v with two directed edges <u,v> and <v,u>.

2.) "Create/Have" Linked list LS (Euler path), that start at the root and follows the eulerian tour.

3.) Assign each arc weight 1 and compute the prefix sum on LS with the list ranking algorithm. Each arc <u,v> gets a number $\eta(< u, v >)$.

4.) For each edge u,v we have: u is parent if $\eta(< u, v >) < \eta(< v, u >)$. Otherwise v is parent.

Step 1 ensures that an Euler tour exists. If we are at node v with adjacency List L[u], degree d and we came from $< u_i, v >$, the next edge in the eulerian tour is: $< v, u_{(i+1) \mod d} >$.
Since the euler tour would be a circle we delete the last edge that would go to the root. This way we create a euler path.

Runtime: step 1 needs O(1), step 2 needs O(0), step 3 need O(log n), step 4 needs O(1). $\Longrightarrow O(\log n)$.
Work: step 3 needs $O(n)$ and step 4 needs $O(n)$, $\Longrightarrow O(\log n)$.

Computing Pre-Order
Problem: Order the Depth-First Search traverses the vertices.

Algorithm 6.3 (**(Pre-Order)**).

1.) Apply the Tree rooting algo to find parents.

2.) Assign arc <parent(v),v> value 1 and arc <v,parent(v)> value 0.

3.) Compute prefix sum on Euler path with list ranking

Runtime: $O(\log n)$
Work: $O(n)$

Computing Depth
We have depth(r) = 0 and depth(v) = depth(parent(v))+1

Algorithm 6.4 (**(Depth)**).

1.) Apply the Tree rooting algo to find parents.

2.) Assign arc <parent(v),v> value 1 and arc <v,parent(v)> value -1.

3.) Compute prefix sum on Euler path with list ranking

Runtime: $O(\log n)$
Work: $O(n)$

6.4 Merging and Sorting

6.4.1 Merging

The sorting part in Merge sort can be done independently. We therefore need to only examine the merging step.
Input: A[1..n/2] and B[1..n/2] both sorted.
Output: C[1..n] also sorted.

Basic Merging
Idea: We insert item A[i] at C[i+j] where in array B exactly j items are smaller than A[i].
For this to work we need to do a binary search for each item of A. Although this can be done completely in parallel. So for 1 merging step we need O(binary search time). However the work will be more.
Runtime: $\sum_{i=1}^{\log n} \log(\frac{n}{2^i} = \log^2(n)$
Work: $\sum_{i=1}^{\log n} 2^i \cdot O(\frac{n}{2^i} \cdot \log(\frac{n}{2^i}) = O(n \log^2(n))$

Improved Merging
Input: Sorted Array A[1..n] and B[1..m]
Idea: We use the **Fencepost-Trick:**
We choose $\sqrt{n}$ evenly-spaced out fenceposts $A[\alpha_1], ..., A[\alpha_{\sqrt{n}}]$ with $\alpha_i = (i-1)\sqrt{n}$. Similarly we choose $\sqrt{m}$ evenly spaced-out fenceposts for B.
We then search for each α_i the j such that $B[\beta_j] \leq A[\alpha_i] \leq B[\beta_{j+1}]$. This can be done in O(1) with O(n+m) processors in the CREW model (We performed all $\sqrt{nm}$ comparisons between the fenceposts).
Then We compare $A[\alpha_i]$ to all $\sqrt{m}$ elements in $B[\beta_j,..\beta_{j+1}]$ (Also in O(1) in the CREW model). With that we have the exact rank of each fencepost $A[\alpha_i]$. To sort the elements in between, we have to recuse on the $\sqrt{n}$ sub-intervals of A with $\sqrt{n}$ elements each. Since we are in the CREW model this doesn't give us a recursion of $\sqrt{n} \cdot T(\sqrt{n})$ but only $T(\sqrt{n})$. So in total we get:

$$\mathrm{T(n)} = \mathrm{O(1)} + \mathrm{T}(\sqrt{n}) = \mathrm{O}(\log \log n)$$

Note that the number of elements in B don't have to change in the recursion and it could be that in the last recursion step it boils down to finding one item of A in the array of B of length m. But since we are in the CREW model this takes O(1) time.
Runtime: $\mathrm{O}(\log \log n)$
Work: $\mathrm{O}(n \log \log n)$

Merge sort
Putting this merging steps together our merge sort algo has:
Runtime: $\mathrm{O}(\log n \log \log n)$
Work: $\mathrm{O}(n \log n \log \log n)$

However if we pipeline the merges we lose the $\log \log n$ factor to get:
Runtime: $\mathrm{O}(\log n)$
Work: $\mathrm{O}(n \log n)$

6.4.2 Quick Sort

Basic Quicksort
Basic principle is to break an unsorted array A[1..n] into B[1..j] and B'[1..(n-j)] according to a pivot A[k]

Algorithm 6.5 (**(Basic Quicksort split)**).

1.) Determine pivot A[k]

2.) For each element A[i] set x[i] to 1 if A[i] $\leq$ A[k] and 0 otherwise

3.) Compute prefix sum on x(k)

4.) Set B[x(i)] = A[i]

5.) For each element A[i] set x[i] to 1 if A[i] > A[k] and 0 otherwise

6.) Compute prefix sum on x'(k)

7.) Set B'[x'(i)] = A[i]

Runtime: O($\log n$)
Work: O(n)

Algorithm 6.6 (**(Basic Quicksort)**).

1.) Apply Basic Quicksort split

2.) Recurse until we only have array of size 1

We claim that we only need $O(\log n)$ recursion steps.

Proof:
Let X := be the number of successes (:= pivot is in $[\frac{1}{4}n, \frac{3}{4}n]$) We have that Pr[success] = $\frac{1}{2}$. Hence E[X]= $\frac{1}{2}$Depth.
Further we choose $O(\log n) \leq 20 \cdot \log n$. Therefore E[X] = 10log n
If we have only successes the worst case of the amount of recursion levels is: $\log_{\frac{4}{3}}$.Now:
$Pr[depth \geq 20 \log n] \leq Pr[X \leq \log_{\frac{4}{3}}] \leq Pr[X \leq 2.5 \log n] = Pr[X \leq (1 - 0.75)10 \log] \leq e^{-\frac{1}{2}0.75^2 10 \log n} = e^{-2.8125 \log n} < 2^{-4 \log n} \leq \frac{1}{n^4}$.
This was for one branch. We can take the Union bound of all branches to get $\leq 1 - \frac{1}{n^3}$ probability that we have $O(\log n)$ depth.

Runtime: O($\log^2 n$)
Work: O($n \log^2 n$)

Improved Quick Sort
Idea: Use multiple pivots

Algorithm 6.7 (**(Improved Quicksort)**).

1.) Pick $\sqrt{n}$ pivots randomly

2.) Sort pivots

3.) Use pivots as splitters and insert all other elements in the appropriate interval [p_i,p_{i+1}]

4.) Recuse on each subproblem between two splitters. Unless:

5.) If the size is under Olog n) solve it deterministicly.

Computation and work:
step1: $O(1)$ depth, $O(\sqrt{n})$ work
step2: $O(\log n)$ depth, $O(n)$ work
step3: $O(\log n)$ depth, $O(n \log n)$ work
step5: $O(\log n)$ depth, $O(\log^2 n)$ work

A detailed analysis with probability, results in that:

Runtime: O($\log n$) (probability: $\geq 1 - \frac{1}{n^2}$)
Work: O($n \log n$) (probability: $\geq 1 - \frac{1}{n^2}$)

6.5 Connected Components

We look at undirected Graphs, given as lists of adjacency linked list L(v) for all v.
Output should be an component identifier D(v) for all v.

Isolated vertices can be identified in O(1) using O(n) work and remove them in O($\log n$) using O(m) work.

We assume CRCW model.

6.5.1 Basic Deterministic Algorithm

Algorithm 6.8 (**(Basic Deterministic)**).

1.) Each vertex v_i is its own Fragment F_i rooted at $r_i (= v_i)$ and $D(v_i) = r_i$

2.) For 1 to $\log n$ do:

3.) Compute the proposed merge $p(r_i)$ for each Fragment F_i (We get directed edges)

4.) Merge the connected Fragments and reshape them into a star-shape

Notes:
step1: Each Fragment has a root r_i. At the start $r_i = v_i$ (because each vertex is its own Fragment).
step3: The proposed merge $p(r_i)$ for Fragment F_i is the minimum root node r_k such that there is an edge from our Fragment F_i to F_k. We then would have $p(r_i) = r_k$. If there is no connected component we propose ourself: $p(r_i) = r_i$.
step4: 1.) For each node remove self-loop and set $D(r_i) = p(v_i)$. 2.) Set $D(v) = D(D(v)$ for all nodes 3.) Repeat step 2 $\log n$ times. 4.) D(v) = $\min\{D(v), p(D(v))$
Note that the proposed merges (seen as edges) could form a cycle (if you are each others minimums), but only a cycle of length at most 2.

Computation and work:
step3: $O(\log n)$ depth and $O(m)$ work
step4: $O(\log n)$ depth and $O(m \log n)$ work

We get:
Runtime: O($\log^2 n$)
Work: O($m \log^2 n$)

6.5.2 Randomized Algorithm

Algorithm 6.9 (**(Basic Deterministic)**).

1.) Each vertex v_i is its own Fragment F_i rooted at $r_i (= v_i)$ and $D(v_i) = r_i$

2.) For 1 to $\log n$ do:

3.) Compute the proposed merge $p(r_i)$ for each Fragment F_i (We get directed edges)

4.) Flip a coin for each Fragment an only keep proposed edges that go from a head fragment to a tail fragment.

5.) Merge the connected Fragments and reshape them into a star-shape

step3: Now instead of just having the minimum r_k of other connected Fragment F_k to be our proposed merge edge, we now propose all possible merges:
For all edges (u,v) with D(v)≠D(u) we propose p(D(v)) = D(u).
step5: Now takes only O(1) since in step4 we made sure the longest chain of pointers is 2.

Runtime: $O(\log^2 n)$ (probability $\geq 1 - \frac{1}{n^2}$)
Work: $O(m \log^2 n)$ (probability $\geq 1 - \frac{1}{n^2}$)

6.6 (Bipartite) Perfect Matching

Lemma 6.3. Matrix multiplication of two $n * n$ Matrices can be done in $O(\log n)$ using $O(n^3)$ work
Same for: Inverse, determinant and adjoin

Given bipartite Graph G=(V,U,E) who has a perfect matching. (If not we simply use algo from chapter 5 with determinant to check).
Now we can't just parallelize algo 5.3. (Will only work if we have a unique perfect matching). However will use this.

Idea: Assign weights, such that the minimum-weight perfect matching is unique.

Lemma 6.4. Isolation Lemma For each edge $e \in E$ pick a random weight $w(e) \in \{1, 2, ..., m\}$.
With probability at least $\frac{1}{2}$, there is a unique min-weight perfect matching.

Proof:
If we don't have a unique min-weight set, there exists at least one edge e, that is ambiguous. Meaning there exits a min-weight set with e in it and there exists a min-weight set without e. Now the probability of e being ambiguous is $\frac{1}{2m}$. Since we have m edges we get the probability, that no edge is ambiguous (hence we have a unique min-weight set) is $m \cdot \frac{1}{2m} = \frac{1}{2}$.
Probability for e being ambiguous: Let W be the weight of a minimum weight set containing e (weight of all edges except e) and W' the weight of a minimum weight set that does not contain e. Let Now in order for them to be the same weight we need W+$w(e)$=W' hence $w(e)$ = W-W'. There will only be one number in $\{1, ..., 2m\}$ that fullfills this. The chance of uniformly choosing that value is: $\frac{1}{2m}$.

Algorithm 6.10 ((Perfect Matching)).

1.) Set each edge a weight $w(e)$ u.a.r from $\{1, 2, ..., 2m\}$

2.) Define $n * n$ matrix A with $a_{ij} = 2^{w(e_{ij})}$ if there is an edge $e = (u_i, v_i)$, 0 otherwise.

3.) Compute det(A)

4.) Define w := heighes power of 2 that diveds det(A)

5.) Compute adj(A) (To get all det(A_{ij}) witch A_{ij} = A but i-th row and j-th column removed)

6.) For each edge $e_{ij} = (u_i, v_j)$ do:

7.) If $\frac{det(A_{ij}) \cdot 2^{w(e_{ij})}}{2^w}$ is odd, we include e_{ij} in output matching.

With the two lemmas below we can show correctness:

Lemma 6.5. If we have a unique minimum weight perfect Matching with weight w. Then the highest power of 2 that divides det(A) is 2^w.

Lemma 6.6. If we have a unique minimum weight perfect Matching M with weight w. Then edge $e_{ij} = (u_i, v_j)$ is in M iff $\frac{det(A_{ij}) \cdot 2^{w(e_{ij})}}{2^w}$ is odd

Runtime: $O(\log^2 n)$
Work: polynomial

6.7 Massively Parallel Computation (MPC)

The PRAM model is far from the reality. Not every process has the same speed, the assumption that a memory access takes unit time is false, etc.. Therefore we look at a more coarse-grained model called MPC:

Def. 6.10. MPC A System composed of M number of machines, each of which has a memory size of S words. Input size is N. Where S = N^ϵ with $\epsilon < 1$. The input is distributed equally to all machines.
We have rounds. Each round, each machine can send at most S words and receive at most S words.

6.8 MPC: Sorting

Goal: Each machine should know the rank (in the sorted list) of each of the items that it *initially* held.
Input: $S = n^\epsilon$, where n is the number of elements we need sorted over all.

Algorithm 6.11 ((QucikSort in MPC)).

1.) Pivots: Each machine marks each of its elements with probability $p = \frac{n^{\frac{\epsilon}{2}}}{2n}$.

2.) Gather all marked elements (pivots) in one leader machine r, which sorts them.

3.) The leader machine r then broadcasts the sorted list to all other machines

4.) We then do a convergecast to r, such that r knows total items between p_i and p_{i+1} (pivots).

5.) For each interval $[p_i, p_{i+1}]$, r assigns a number of machines that are responsible for sorting these elements.

6.) Machine r broadcasts its result from step 5.

7.) Each machine sends each of the items it hold to a random one of the machine responsible for the related subproblem.
(Remember for each item, from which machine it initially came)

8.) Recuse on each subproblem unless its smaller than n^ϵ, then we sort in one machine.

Complexity
Broadcasting and Convergasting only require $O(\frac{1}{\epsilon})$ rounds. (In practise this is bout 3 rounds).
So each iteration of the algo needs $O(\frac{1}{\epsilon})$ rounds. With lemma below we get that the whole algorithm needs $O(\frac{1}{\epsilon^2})$ rounds, which is constant.

Lemma 6.7. With probability $1 - \frac{1}{n^10}$, after $O(\frac{1}{\epsilon})$ iterations of the above algorithm, the output is sorted

6.9 MPC: Connected Components

Given is a graph G as list of edges.
We assume we have $S = n^{1+\epsilon}$ where n is the number of vertices. However m >> n.
Goal: Find maximal Forest of G (Knowing forest is better than just components).

Algorithm 6.12 ((Components in MPC)).

1.) Send all edges to the first $\frac{2m}{n^{1+\epsilon}}$ machines at random.

2.) Each machines calculates a maximal local forest and discards the other edges that aren't in the Forest.

3.) Repeat until we only have $n^{1+\epsilon}$ edges. Then we can just put all of them in one machine and calculate maximal forest.

Step 1 works because: E[edges going to each machine $= \frac{m}{2m/n^{1+\epsilon}} = \frac{n^{1+\epsilon}}{2}$.
Also after each iteration we reduce the number of edges by $\frac{n^\epsilon}{2}$, because a forest can hold at most n-1 edges and only $\frac{2m}{n^{1+\epsilon}}$ machines have edges.
With that we can conclude that oafter$O(\frac{1}{\epsilon})$ repetitions that algo terminates. Hence constant number of rounds.

6.10 MPC: Maximal Matching

Maximal Matching: you can't add anymore edges to it.

Lemma 6.8. Any maximal Matching M has at least $\frac{|M^*|}{2}$ edges.
$|M^*|$ is the number of edges of any maximum matching M^*.

Def. 6.11. A maximum matching is the matching with absolutely largest cardinality possible.

Goal: Each machine should know which of its edges belong to M.

Algorithm 6.13 ((Maximal Matching in MPC)).

1.) Mark each edge with probability $p = \frac{n^{1+\epsilon}}{2m}$

2.) Move all marked edges to one machine

3.) Compute maximal matching among marked edges

4.) Announce all matched vertices to all machines

5.) On every machine delete edges associated with these vertices.

6.) Repeat until only edges in the matching remain.

Lemma 6.9. After each iterating we have at most $\frac{2m}{\epsilon}$ edges with probability $1 - exp(-O(n))$.

And with that we show that in $O(\frac{1}{\epsilon})$ iterations (hence constant) the algorithm terminates.

7 Notation

$log(x) = log_2(x)$
$ln(x) = log_e(x)$

8 Math

- $\sum_{i=1}^{\log n} \log(\frac{n}{2^i} = sum_{i=1}^{\log n}(\log(n) - i) = \log^2(n)$
- $\sum_{i=1}^{\log n} 2^i \cdot O(\frac{n}{2^i} \cdot \log(\frac{n}{2^i}) = \sum_{i=1}^{\log n} O(n(\log(n-i)) = O(n \log^2(n))$
- $O(\sqrt{nm}) = O(n+m)$
- $T(n) = O(n \log n) + \underbrace{T(\sqrt{n}) + ... + T(\sqrt{n})}_{\sqrt{n}}$
- If $E[X] \leq c$ Then with Markov: $Pr[X \geq 1] \leq c$
- $det(A) = \sum_{\sigma \in S_n} sign(\sigma) \cdot value(\sigma)$
- $value(\sigma) = \prod_{i=1}^{n} a_{i,\sigma(i)}$

9 Additional Notes

- $\text{sign}(\pi) = -1^{\text{number of even cycles}}$
- LP-Trick: Epsilon
- LP-Trick: have (A -A)^{T}y $\geq (c - c)^T$. This imlies $A^T y \leq c$ and $-A^T y \leq -c$. This implies $A^T y = c$
- LP-Trick: Want x ≥ 0. Replace x with $y2 - y1$, A with $(A - A)$ and c with $(c - c)^T$
- LP-Trick: Want minimize $c^T x$ instead of maximize $c^T x$. Just do minimize $-c^T x$.
- LP-Trick: Wnat $Ax \leq b$ and have $Ax = b$, just do: $Ax \leq b$ and $-Ax \leq -b$.
- Pfaffian orientation: Every nice cycle has an odd orientation
- Lemma: If every finite face has an odd number of clockwise edges, then all nice cycles are oddly oriented
- Nice cylce C: if vertices removed of C, then graph without C has a PM.
- Schwartz-Zippel: For polynom of degree at most d, with n variables and each variable is $\in S$. The number of polynoms that result in 0, are $d \cdot |S|^{n-1}$.
- Remember in S trick take the nonzero element as your S + $a_j r_j$.
- Existence of PM iff det(tutte) $\neq 0$.
- Number of PM in planar graph = $\sqrt{det(skewMatrix)}$
- For bipartite Graph: det(Adjacency Matrix) is odd, if number of PM is odd
- For bipartite Graph: det(Adjacency Matrix) is even, if number of PM is even (careful 0 is also even. So 0 PM and 2 PM can both result in det(Adj.Matrix) = 0)
- Adjacency Matrix:
$$B := \begin{cases} 1 & \text{if } u_i, v_j \in E(G) \\ 0 & \text{otherwise} \end{cases}$$
- Skew Matrix: Tutte matrix but with 1, -1
- You are an ancestor of yourself