Algorithms, Probability and Computability
Summary

Lukas Wolf
wolflu@student.ethz.ch

ETH Zurich
HS 2020

Contents

1 Bootstrapping Techniques

Definition 1.1. ***Bootstrapping*** *is the process of building complex systems from simpler ones. We solve a problem by recursively applying the method that we are about to develop. We may improve the time complexity of the algorithm by doing so.*

1.1 Computing MSTs in Linear Time

As we already know from previous lectures, a MST can be computed by Prims's algorithm (together with Fibonacci heaps) in $O(n \log n + m)$ and by Kruskal's algorithm (together with a union find data structure) in $O(m \log n)$.
We first describe some well known facts on MSTs:

- If the weight function $w : E \to \mathbb{R}$ is injective, the MST is unique. We may assume this case from now on (which is not a strong assumption, just add to each edge weight a randomly chosen (small) value ϵ_i and the property will hold with high probability).
- For every vertex $v \in V$, let $e_{min}(v)$ denote that edge incident to v that has minimum weight. Then all such edges belong to the MST.
- For every cycle C let $e_{max}(C)$ denote the edge in C that has maximum weight. Then no such edge can belong to the MST.

Definition 1.2. *Every edge $e \in E$ that does not belong to a given tree $T = (V, E_T)$ (which may be a MST) in the graph $G = (V, E)$ will close a unique cycle with T that consists of e and the tree edges. We call such an edge* ***T-heavy****, iff all other edges in this cycle have a lower weight. Clearly we have:*
T is a MST $\iff$ all edges $e \in E \backslash E_T$ are T-heavy.

There exists an algorithm that finds all T-heavy edges in time $O(m)$ (not covered here). The second observation from above is the basis of Boruvk

Algorithm 1: Boruvka's Algorithm(G)

```
for v ∈ V do
    compute e_min(v)
    insert e_min(v) in the MST and contract e_min(v)
recurse (until the graph contains only one vertex)
```

Every iteration of Boruvka's algorithm can be implemented in $\mathcal{O}(m)$ and reduces the number of vertices by at least a factor of two. Thus the number of iterations is bounded by $\log_2 n$ and we thus obtain a total running time of $\mathcal{O}(m \log n)$. We now want to get rid of the factor $log(n)$.
The problem with Boruvka's algorithm is that we reduce the number of vertices by factors of two, but we have no control on the reduction of the number of edges: we can only guarantee that we reduce m edges to at most $m - n/2$ edges, which, for $m >> n$, is still roughly equal to m.

In order to attack this problem, we first observe that Boruvka's algorithm also works for cases in which the graph is not connected. It then returns minimum spanning trees for each component (a minimum spanning forest MSF).

Algorithm 2: Randomized MST Algorithm(G):

1 Perform three iterations of Boruvka's algorithm (which reduces the number of vertices to at most n/8)
2 In the new graph:
  2.1 Select edges with probability 1/2 and compute recursively (first recursion) an MSF for the graph consisting of the selected edges. Call this forest T
  2.2 Use FindHeavy to find all unselected edges that are not T-heavy to T and delete all other edges.
3 recurse (until the graph contains only one vertex - second recursion)

We then obtain the MSF of G by reversing the three initial Boruvka steps.

Correctness: We only delete edges during Boruvka steps and edges that are T-heavy. By using a subsampling technique, one can show that in the second step, the expected number of edges that are not T-heavy can be bounded by n. The expected running time for the above algorithm is then $C \cdot (n + m)$.

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

From previous lectures we already know two ways to solve this problem. One can use a flow-based algorithm in $O(n^4 \cdot \log n)$ time. We also saw a randomized algorithm that solves the problem with high probability in $O(n^4)$ time. We now improve the latter algorithm to achieve a runtime of $O(n^2 \cdot (\log n)^3)$ This can be abbreviated with the notation $\tilde{O}(n^2)$ (log-polynomial).

1.2.1 Basic version

Assume the graph is connected and we allow multiple edges. We may perform an edge contraction in $O(n)$ time, choose an edge uniformly at random in $O(n)$ time, and find the number of edges connecting two given vertices in $O(1)$ time.

Lemma 1.3. *Let G be a multigraph and e an edge of G. Then $\mu(G\backslash e) \geq \mu(G)$. Moreover, if there exists a minimum cut C in G such that $e \notin C$, the $\mu(G\backslash e) = \mu(G)$.*

The following algorithm repeatedly chooses random edges of the current graph and contracts them, until only two vertices are left:

Algorithm 3: BasicMinCut(G)

```
while G has more than 2 vertices do
    pick a random edge e in G
    G ← G\e
return the size of the only cut in G
```

Lemma 1.4. *Let G be a multigraph with n vertices. Then the probability of $\mu(G) = \mu(G\backslash e)$ for a randomly chosen edge $e \in E(G)$ is at least $1 - \frac{2}{n}$.*

To obtain the true minimum cut value two events must occur: we must choose a non-mincut edge as in above lemma in the first iteration, and must be successful in BasicMinCut(G \ e). This leaves us with the following recurrence:

$$p \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}.$$

Lemma 1.5. *A connected multigraph with n vertices cannot have more than $\binom{n}{2}$ minimum cuts.*

The lemma follows from the insight that finding a minimum cut is independent of finding other cuts and the probability of finding a MinCut cannot be larger than 1.

We can perform **probability amplification**, i.e. run the algorithm N times and return the smallest cut size found, then the failure of N runs in a row is $(1 - \frac{2}{n(n-1)})^N \leq e^{-2N/n(n-1)}$ (with using the well-known inequality $1 + x \leq e^x$). We can therefore bound the failure probability. Altogether we have a randomized minimum cut algorithm with running time $O(n^4)$ and with very small probability of failure. Note that we can get the actual edges of the cut by keeping track of the identity of the edges during contractions, and then from the cut in the final 2-vertex graph we can reconstruct the corresponding cut in the input graph.

1.2.2 Bootstrapping version

For an improved running time we first observe that probability of contracting an edge in the minimum cut increases as the graph shrinks. This suggests that we contract edges until the number of vertices decreases to some suitable threshold value t, and then we use some other, perhaps slower algorithm that guarantees a higher probability of success. In fact we do not even need a slower but more reliable algorithm, we just use the same algorithm, but with a smaller error probability (that we can get by repeated calls).

We start with the graph G and we first duplicate this graph and then contract random edges in both copies independently until some number of vertices αn (for some appropriate constant α). We do this until we arrive at the border case $n = 16$ (16 is rather arbitrary chosen for a handy analysis).

Algorithm 4: Random MinCut(G)

```
if n ≤ 16 then
    compute μ(G) by some deterministic method
else
    t = ⌈αn⌉ + 1
    H_1 = RandomContract(G, t)
    H_2 = RandomContract(G, t)
    return min(MinCut(H_1), MinCut(H_2))
```

Using our assumptions on the representation of the current graph, we know that we can construct the two graphs $H_i, i \in \{1, 2\}$ in $O(n^2)$ time. We thus get the recurrence $t(n) \leq O(n^2) + 2t(\lceil \alpha n \rceil + 1)$. The following lemma can be proven by induction:

Lemma 1.6. *For $\alpha \leq \frac{1}{\sqrt{2}}$ we get $t(n) = O(n^2 \cdot \log n)$.*

The following lemma gives us a bound on the success probability:

Lemma 1.7. *For $\alpha \geq \frac{1}{\sqrt{2}}$ there is a real constant $c > 1$, such that the success probability of MinCut $p(G) \geq \frac{1}{1+log_c n}, \forall n \geq 2$.*

Note that the bounds on α are complementary and therefore our only choice in order to apply both lemmata is $\alpha = \frac{1}{\sqrt{2}}$. We only need $O(\log^2 n)$ repetitions to get a good failure probability, say 10^{-6}. We summarize the achievements:

Theorem 1.8. *Let $a \geq 1$ be a parameter and let $\alpha = \frac{1}{\sqrt{2}}$. The randomized algorithm consisting of $N = C \cdot a \cdot \log^2 n$ repetitions of MinCut, where C is a suitable constant, has running time $O(n^2 \log^2 n)$ and for every n-vertex input graph it computes the minimum cut size correctly with probability at least $1 - n^{-a}$*

2 Random(ized) Search Trees

2.1 Definition

A **search tree** for a given set S of $n \in \mathbb{N}_0$ distinct real numbers, is a binary ordered rooted tree, i.e. there is a distinguished node, called the root, and every node has at most two children. Nodes without children are called leaves. The nodes are labeled by elements in S. In order to specify this labeling, we define: $S^{<x} := \{a \in S | a < x\}$ and $S^{>x} := \{a \in S | a > x\}$, for $x \in \mathbb{R}$ analogously.
The empty tree is denoted by λ, and a search tree can be specified by a **replacement policy** (or grammar), similar to context free grammars. For a tree B_S, one can take an arbitrary $x \in S$ as root, and we recursively put $B_{S^{<x}}$ into the left subtree, and $B_{S^{>x}}$ into the left subtree.
Note that B_S does not stand for just one tree, but for a whole family of trees that can be derived with the above replacement system.

The **depth** of a node, $d(v)$ is defined by: $\begin{cases} d(v) := 0, \text{ if v is the root, and} \\ d(v) := 1 + d(u), \text{ u parent of v.} \end{cases}$
The **height** of a tree is the maximum depth among its nodes.

A slight twist in the definition of search trees leads to our specification of **random search trees**: in the replacement process we now draw the $x \in S$ uniformly at random. The **probability of a tree** is defined as the probability that it is the result of the above randomized process - we obtain a probability distribution on B_S. For $|S| \leq 2$ this is the uniform distribution, but for $|S| \geq 3$ it is not uniform.

Lemma 2.1. *$S \subseteq R$, finite. Given a tree in B_S, we let $w(v)$, v is a node, denote the number of nodes in the subtree rooted at v (including v).*
The probability of the tree according to the above distribution is $\prod_v \frac{1}{w(v)}$, where the product is over all nodes v of the tree.

2.2 Overall (and Average) Depth of Keys

Given some finite set $S \subseteq \mathbb{R}$, the **rank** of $x \in \mathbb{R}$ in S is:

$$rk(x) = rk_S(x) := 1 + |\{y \in S | y < x\}|.$$

In general, if $x \in S$, then x is the rk(x)-smallest element in S. For a tree B_S, we abuse notation by writing rk(v) short for the rank of v's key in S.
For $i, n \in \mathbb{N}, i \leq n$ we denote by $D_n^{(i)}$ the random variable for the **depth** of the key of rank i in a random search tree for n keys. We want to investigate the **expected overall depth** $E[\sum_{i=1}^n D_n^{(i)}]$ in a random search tree. One nth of this quantity is then the expected average depth of the nodes in a random search tree.

Depth of smallest key (rank 1):
Let us write D_n short for $D_n^{(1)}$. For the general case $n \in \mathbb{N}$, we can discriminate between the possible ranks of the root key and using the law of total expectation we get

$$E[D_n] = \sum_{i=1}^n E[D_n | rk(root) = i] \cdot Pr[rk(root) = i],$$

where the expected value is either 0 (if i = 1), or $1 + E[D_{i-1}]$ otherwise. Here we use the observation, that the depth of the smallest key is 0 if it sits in the root, or it is 1 plus the depth of the smallest key within the left subtree.
Writing d_n for $E[D_n], n \in \mathbb{N}$, we get the recurrence:

$$d_n = \begin{cases} 0, \text{ if } n = 1, \text{and} \\ \frac{1}{n} \sum_{i=2}^n (1 + d_{i-1}), \text{ otherwise.} \end{cases}$$

For $n \in \mathbb{N}, n \geq 3$, we make use of a common trick, namely computing $d_n - d_{n-1}$ and thus we obtain:

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

Together with $d_1 = 0$ and $d_2 = \frac{1}{2}$, successive invocation of (2.1) yields

$$d_n = \frac{1}{n} + \frac{1}{n-1} + ... + d_2 = H_n - 1$$

where $H_n := \sum_{i=1}^{n} \frac{1}{i}, n \in \mathbb{N}_0$ is the n-th **harmonic number**. The harmonic numbers are considered the discrete analogue of the natural logarithm. We have an upper bound of

$$H_n \leq 1 + \int_1^x \frac{1}{x} dx = 1 + \ln n, n \geq 1.$$

Also $\ln(n+1) \leq H_n, \forall n \in \mathbb{N}_0$, and $\lim_{n \to \inf}(H_n - \ln n) =: \gamma = 0.57721$, Euler's constant.

Knowing the expected depth of the smallest key, we might wonder now what the chances are that this depth is large (so called **tail estimates**). We can always apply Markov's inequality and get:

$$Pr[D_n \geq \lambda ln(n)] \leq Pr[D_n \geq \lambda(H_n - 1)] \leq \frac{1}{\lambda}.$$

Overall Depth:
For $n \in \mathbb{N}_0$, let $X_n := \sum_{i=1}^{n} D_n^{(i)}$, the random variable for the overall depth of all nodes in a random search tree for n keys. For general $n \geq 1$ we condition on the rank and get,

$$E[X_n] = \sum_{i=1}^{n} E[X_n | rk(root) = i] \cdot Pr[rk(root) = i] = n-1+\frac{1}{n} \cdot 2 \cdot \sum_{i=1}^{n} E[X_{i-1}],$$

since in every node in the two subtrees attached to the root - altogether there are n-1 such nodes - increases its depth by 1, while there is no contribution from the root. Using a similar trick as before we end up with the following theorem:

Theorem 2.2. *$n \in \mathbb{N}_0$. The expected overall depth of a random search tree for n keys is $2(n+1)H_n - 4n = 2nln(n) + O(n)$.*

2.3 Expected Height

We wish to analyze the height of a random search tree for n keys denoted by the random variable $X_n := \max_{i=1}^{n} D_n^{(i)}$. We recall that every binary tree with n nodes has height at least $\lfloor \log n \rfloor$, so this is definitely a lower bound for the expected height. The expected average depth is $2ln(n) + O(1)$, and therefore an even better lower bound. Determining $E[X_n]$ seems hard:

$$E[X_n] \leq \log E[2^{max_{i=1}^{n} D_n^{(i)}}] \leq \log E[\sum_{i=1,\text{i leaf}}^{n} 2^{D_n^{(i)}}],$$

(the first inequality follows from Jensen's inequality) therefore we analyze $Z_n := \sum_{i=1,\text{i leaf}}^{n} 2^{D_n^{(i)}}$ instead. For $n \geq 2$ we get:

$$E[Z_n] = \sum_{i=1}^{n} E[Z_n | rk(root) = 1] \cdot Pr[rk(root) = i],$$

where the expected value equals $2(E[Z_{i-1}] + E[Z_{n-i}]$. For $z_n := E[Z_n]$:

$$z_n = \begin{cases} 0, \text{ if } n = 0, \\ 1, \text{ if n = 1 and} \\ \frac{4}{n}\sum_{i=1}^{n} z_{i-1}, \text{ otherwise.} \end{cases}$$

By solving the recursion we get $E[Z_n] = \frac{(n+3)(n+2)(n+1)}{30}$, and get an upper bound of $3\log n + O(1)$.
Since we can still play with the constant 2 of the exponent, we can redefine it to a constant c, minimize over it and get the following theorem:

Theorem 2.3. *The expected height of a random search tree for n keys is upper bounded by $c \cdot ln(n)$, where $c = 4.311...$ is the unique value greater than 2 which satisfies $(\frac{2e}{c})^c = e$.*

Therefore the constant in the leading term is already tight.

Tail estimates
Let us conclude by pointing out that knowing a good estimate for $E[C^{X_n}]$ immediately gives a good tail estimate for X_n via Markov's inequality, namely

$$Pr[X_n \geq \tau ln(n)] = Pr[C^{X_n} \geq C^{\tau ln(n)}] \leq \frac{E[C^{X_n}]}{C^{\tau ln(n)}} \leq n^{2C-1-\tau ln(C)}.$$

The term $2C - 1 - \tau ln(C)$ is minimized for $C = \tau/2$.

Theorem 2.4. $n \in \mathbb{N}, \tau \in \mathbb{R}^+$.

$$Pr[X_n \geq \tau ln(n)] \leq n^{\tau(1-ln(\tau/2))-1};$$

in particular $Pr[X_n \geq 2eln(n)] \leq \frac{1}{n}, (2e = 5.435...)$.

2.4 Expected Depth of Individual Keys

Lemma 2.5. $i, j \in \mathbb{N}$. *In a random search tree for $n \geq maxi, j$ keys, we have*

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

Theorem 2.6. $i, n \in \mathbb{N}, i \leq n$. *Then* $E[D_n^{(i)}] = H_i + H_{n-i+1} - 2 \leq 2\ln 2$.

2.5 Quicksort

Quicksort needs $O(n \log n)$ time on average and is easy to implement. We draw u.a.r. one $x \in S$, called the *pivot* and partition S into sets with elements smaller than x and greater than x, respectively. We then sort both subsets (subtrees) recursively. We now concentrate on the number of comparisons performed and will see the similarities to random search trees.
The comparisons occur in the splitting step, $|S| - 1$ of them. We let $t_n, n \in \mathbb{N}$, stand for the expected number of comparisons made when sorting a set of n keys. Then $t_0 = 0$, and for $n \geq 1$:

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

This is exactly the recurrence we obtained for the expected overall depth of random search trees, and we get the solution for t_n for free.

Theorem 2.7. *$n \in \mathbb{N}_0$. The expected number of comparisons performed by quicksort() for sorting a set of n numbers is $2(n+1)H_n - 4n$.*

It is important to note the difference between deterministic quicksort and randomized quicksort. In the deterministic version where we, e.g., always choose the first element as pivot, we may have bad inputs, such as an already sorted list. In contrast to that, for the randomized version there cannot be any bad inputs.

2.6 Quickselect

Given a set S of keys, suppose we want to know the k-smallest element in S (i.e. the element of rank k in S). This may be done with Quicksort and picking the k-th element in $O(n \log n)$ time. In fact, one of the two recursive calls of Quicksort is always irrelevant. This suggests us the procedure Quickselect.

Let $t(k, n)$ denote the expected number of comparisons among elements in S, with $|S| = n$. In order to get an idea of the asymptotics, assume for the recursive call walways the size of the larger of the two sets and set $t_n := max_{k=1}^{n} t(k, n)$. That is, $t_1 = 0$, and for $n \geq 2$ we get:

$$t(k, n) = n - 1 + \frac{1}{n}\sum_{l=1}^{n} t_{max(l-1,n-l)}.$$

One can now show by induction, that $t(k, n) \leq 4 \cdot n$.

Theorem 2.8. *$k, n \in \mathbb{N}_0, 1 \leq k \leq n$. The expected number of comparisons (among input numbers) performed by quickselect() for determining the element of rank k in a set of n numbers is at most 4n.*

2.7 Randomized Search Trees

2.7.1 Treap Basics

Definition 2.9. *A **Treap** is a coined word that stands for a symbiosis of a binary search tree and a heap. It is defined for sets $Q \subseteq \mathbb{R} \times \mathbb{R}$, with the elements in Q called **items**. The first component of an item x is its **key**, key(x) and the second component is its **priority**, prio(x). Suppose that no two keys in Q are the same, nor are two priorities the same. Then a treap on Q is a binary tree with nodes labeled by Q which is a search tree with respect to the keys, and a min-heap with respect to the priorities. The fact that every set of items allows a treap, and this treap is actually **unique**, becomes evident from the alternative grammar style definition ($Q^{<x} := \{y \in Q | key(y) < key(x)\}$ and analogously for $Q^{>x}$).*

If the priorities are chosen independently and u.a.r. from $[0, 1]$, then the resulting treap is a random search tree for the keys of its items. Therefore, if we maintain for a set of keys a treap, where for every newly inserted key a random priority is chosen, then this treap is a random search tree for the keys, independently from the insertion order. We can also assume all the wonderful properties like expected (and high probability) logarithmic height of the tree. Remains the issue of maintenance of a treap under insertions and deletions.

2.7.2 Treap Insertions and Rotations

To insert a new item x in a treap, we first proceed in a standard binary search tree and insert the new item as a leaf according to its $key(x)$. Next we have to reinstall the heap property: As long as x has smaller priority as its parent item y we perform a rotation at y which makes y a child of x (same as in AVL-tree rotations). This will let x eventually end up at its proper place in the treap.

The running time is proportional to the depth of the new item as a leaf before reinstalling the heap property, plus the number of rotations necessary to get the new item to its place. Both is bounded by the height of the tree and thus expected O(log n).

However, *rotations* are much more costly than just walking down a tree in a search. We can read the number of such rotations from the repaired treap without knowing the priorities (and thus without knowing which rotations were indeed performed). In order to quantify this, we define the *left (right) spine* of a subtree rooted at v as the sequence of nodes on the path from v to the smallest (largest, respectively) key in the subtree. Now associate with a node the sum of the lengths of the right spine in the left child's subtree and of the left spine in the right child's subtree. This number increases exactly by one with every rotation for the node with the new item, and thus this quantity at its final position specifies the necessary number of rotations.

Lemma 2.10. *$n \in \mathbb{N}, j \in [n]$. In a random binary search tree for n keys, the right spine of the left subtree of the node of rank j has expected length $1 - \frac{1}{j}$, and the left spine of the right subtree has expected length $1 - \frac{1}{n-j+1}$.*

This tells us that the expected number of rotations is less than 2, no matter how large the tree is.

2.7.3 Treap Deletions, Splits and Joins

A deletion in a treap is an inverse insertion. First we rotate the item to be removed down the tree until it is a leaf, then we remove it. When we push the item down the tree, we always have a choice of a right or left rotation - which one to select is decided by the priorities. If the left child has the smallest priority of the children, then this child should become the new root of the subtree and we rotate right. Otherwise, we rotate left. Analysis is the same for the insertion.

For a given pivot value s, a *split* of a treap for items Q generates two treaps for items $Q^{<x}$ and items $Q^{>x}$ (we assume that s is not among the keys of the treap). An implementation is easy to describe: first insert an item with key s and priority $-\infty$, which will end up as root. Then remove the root. left and right subtree are the desired trees. The time is again bounded by the height of the tree - the expected number of rotations it not constant in this case, though. The *join* operation takes two treaps with one holding keys all of which are smaller than that of the other one. This can be seen as an inverse split. That is, we generate a new root with an in-between key s and priority $-\infty$, attach the treap with the smaller keys as a left subtree and the other as a right subtree, and then we delete the root item.

Theorem 2.11. *In a randomized search tree (a treap with priorities independently and u.a.r. from* $[0,1]$*) operations find, insert, delete, split and join can be performed in expected time O(log n), n the number of keys currently stored. The expected number of rotations necessary for an insertion or a deletion is always less than 2.*

2.7.4 Random Real Numbers in Treaps

One may be worried that the priorities are real numbers. Since we need to compare these numbers from the interval $[0,1]$, we use a finite binary representation of these. We check whether the available bits suffice to take such a decision. Otherwise we produce extra random bits for the two numbers which extend these sequences until a decision is possible.

3 Point Location

In this chapter we concentrate on the so-called *locus approach*, where the domain is partitioned into regions of equal answers. Often we will pre-process some data first, and then allow to locate queries very fast in the data structure.

3.1 Point/Line Relative to a Convex Polygon

The convex hull conv(P) of a finite set P of points in the plane is a bounded convex set, and if it is not empty, a point, or a line, it is bounded by a convex polygon. The polygon can be described by a finite sequence of vertices, e.g. in counter-clockwise order.

3.1.1 Inside/On/Outside a Convex Polygon

To decide whether a query point q lies inside, on, or outside of a convex polygon C can be answered easily: We sort the x-coordinates of the vertices of C and prepare them for binary search (in a linear array). The intervals between two consecutive x-coordinates are associated with two lines that carry the edges of C in the corresponding x-range of the plane. For a query point q, we first

locate its x-coordinate x_q in this structure. If x_q is smaller than the smallest x-coordinate of vertices, q is clearly outside of C; same if it is larger than the largest coordinate. Otherwise, x_q lies in some interval and we can compare q with the two associated lines (above and below) to decide about the answer to the query. Summing up, the structure needs $O(n)$ space and $O(\log n)$ query time. If the vertices of C are provided in sorted order along C, building the structure takes linear time as well.

3.1.2 A Line Hitting a Convex Polygon

We observe that a line l intersects with a convex polygon C iff it lies between (or coincides with one of) the two tangents to C parallel to l. Such a tangent is determined by a vertex of C. Thus, we prepare for queries as follows: we direct every edge of C in the direction as we pass it moving around C in counter-clockwise order. Every such directed edge e has an angle α_e in the range $[0, 2\pi)$ with the x-axis. Note that if e and the next edge e' share vertex v, then all directed tangents which have C to their left and have an angle in the range from α_e and $\alpha_{e'}$, touch C in vertex v. Hence, we can store all angles $\alpha_0, \alpha_1, ..., \alpha_{n-1}$ sorted in an array. Given a query line l, we get two angles β and β' - depending on which direction we assign to l - which we locate in our array, therefore getting two vertices v and v'. The vertices v and v' associated to the found intervals determine the tangents parallel to l. l misses C iff both v and v' lie on the same side of l. Again, this results in an optimal structure with linear storage and preprocessing time, and with logarithmic query time.

3.2 Line Relative to Point Set

To decide whether a query line has a set of points P all on the same side, we can compute the convex hull of P in $O(n \log n)$ time, and are back in known territory.

3.2.1 Reporting the Points Below a Query Line

If the points in P are the vertices of a convex polygon C, a structure with optimal query time of $O(k + \log n)$ for reporting the k points below a non-vertical query line l is relatively easy to derive: In logarithmic time, we find a vertex p of C which is contained in the tangent which has C above and is parallel to the query line l. If p is above or on l we are done, since no other point can be below l. Otherwise, starting from p we first move in clockwise order through the vertices of C and report them (as "below l") until we either have exhausted all vertices, or we meet the first vertex not below l. In the latter case we repeat the same procedure in counter-clockwise order. In order to handle general sets P, we first define a new structure. The *onion* of a general set P is the sequence

$$(R_0, V_0), (R_1, V_1), ..., (R_t, V_t).$$

If P is empty, the sequence is empty. Otherwise let $R_0 := conv(P)$ and let V_0 be

the vertex set of R_0. $(R_1, V_1), ...$ is recursively defined to be the onion of $P \setminus V_0$. We observe that

- $R_t \subset R_{t-1} \subset R_0$; hence, if R_i is completely above or on a line l, then all $R_j, j \in \{i, ..., t\}$ are above or on the line l, and so are all $V_j, j \in \{1, ..., t\}$.

Each V_i is either the vertex set of a convex polygon or $|V_i| \leq 2$. In either case we can preprocess V_i so that the points in V_i below a query line can be reported in time $O(k_i + \log n_i)$, k_i the number of points reported and $n_i = |V_i|$. Now we can start querying with a line l in the structure for V_0, report all points in V_0 that are below l, then proceed with further structures until either (i) we have reached a V_j that is completely above or on l or (ii) we get to the situation that we have exhausted all sets (i.e. have reached V_t). In case (i) this takes time

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

We use $j \leq k$ here, since each k_i is at least 1. In the case (ii) we take time

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

We note that this is no longer optimal, for instance for $k = \lceil \log n \rceil$, we have $O((\log n)^2)$.

3.2.2 Improvement: Searching in Many Lists

The problem is that whenever we handle a new convex polygon V_i in the structure, we locate a real number β in a linear array. In each step, we throw away all the information from previous iterations. For a better solution we first enhance the sets S_i with extra information, resulting in sets $\bar{S}_i$. The "last" set S_t is left untouched, so $\bar{S}_t = S_t$. Otherwise, $\forall i = t-1, t-2, ..., 0$, to obtain $\bar{S}_i$ we add to S_i every other number from $\bar{S}_{i+1}$ in its sorted order starting with the second number.
Each of these $\bar{S}_i$'s are stored in a linear array. We observe that each interval in $\bar{S}_i$ is either entirely contained in $\bar{S}_{i+1}$ or contains exactly one number from it. Hence, $|\bar{S}_i| \leq |S_i| + \frac{|\bar{S}_{i+1}|}{2}$. In the former case we add a pointer from intervall I to I', in the latter a pointer from I to a, where a is the single number from $\bar{S}_{i+1}$. Once we have located a number x in array $\bar{S}_i$, we can locate it in $\bar{S}_{i+1}$ in O(1) time.
Summing up, if $\bar{S}_j$ is the last structure we have to search in, the overall query time went down to

$$O(k + \log n + j) = O(k + \log n),$$

which is optimal. We have the $\log n$ factor for the location in the first array.

Lemma 3.1. *The overall size of the structure is bounded by* $O(\sum_{i=0}^{t} \bar{n_i}), \bar{n_i} = |\bar{S}_i|$. *One can prove by induction that the following holds:*

$$\sum_{i=0}^{t} \bar{n_i} \leq 2n.$$

Theorem 3.2. *A set P of n points in the plane can be preprocessed in time $O(n^2 \log n)$ and linear storage so that the set of points below a query line can be reported in time $O(k + \log n)$, k the number of points below the query line.*

Better solutions with time $O(n \log n)$ exist but are not discussed in this lecture.

The method of cascading some elements of one list to another list is called *fractional cascading* and has numerous applications.

3.2.3 Duality

A point and a (non-vertical) line can both be written as a pair $(a, b) \in \mathbb{R}^2$, i.e. lines and points are just different interpretations of the same 'thing', namely $\mathbb{R}^2$. Duality exploits this fact by mapping points to lines and lines to points. Let * be the mapping that maps points to non-vertical lines, and non-vertical lines to points by

$$\begin{aligned} \text{point } p = (a, b) &\longrightarrow \text{line } p* : y = ax - b \\ \text{line } l : y = ax + b &\longrightarrow \text{point } l* = (a, -b) \end{aligned}$$

Note that the duality * preserves *incidents*, as a point p lies on line l iff point l* lies on line p*.

Observation 3.3. *Let p be a point and l a non-vertical line in $\mathbb{R}^2$. Then,*

- $(p*)* = p$ *and* $(l*)* = l$,
- $p \in l$ *iff* $l* \in p*$, *and,*
- *p lies above l iff l* lies above p*.*

3.2.4 Counting the Points Below a Query Line

If we only want to count the points below a line, we would like to get rid of the k in our runtime. By duality, we can transform the problem to the problem of counting lines from a given set L of non-vertical lines above a query point q. That, on the other hand, is a point location problem: The plane is partitioned into regions of equal answers $i, i \in \{0, ..., n\}$, and what is left is to locate q in these regions.
For $k \in [k]$ the *k-level*, Λ_k, of a set of L of non-vertical lines is the set of all points in $\mathbb{R}^2$ which have at most k-1 lines above and at most n-k lines below. Hence, such points lie on at least one of the lines in L. It is easy to see that every vertical line intersects the k-level in exactly one point. Given a point q and a bi-infinite x-monotone curve C, we write $C \succ q (q \prec C)$, if q lies below (above, resp.) the curve C.

Observation 3.4. *For $k \in [n]$, the k-level Λ_k is an x-monotone bi-infinite curve in the plane. If $1 \leq i \leq j \leq n$, every point on the i-level is either on or above the j-level.*

The number of lines in L above a point q is $\min\{k|q \succeq \Lambda_k\} - 1$, *with the convention that* Λ_{n+1} *is a symbolic curve with* $q \succeq \Lambda_{n+1}$ *for all* $q \in \mathbb{R}^2$.

So the question of how many lines are above a query point q can be resolved by locating q among the $\Lambda'_k s$, which can be done by binary search with $O(\log n)$ comparisons with these curves. Remains the issue how we efficiently compare q to such curves.

The Complexity of a Line Arrangement
A set L of n lines in the plane partitions $\mathbb{R}^2$ into areas of various dimensions: vertices (of dimension 0), edges (dimension 1), and cells (dimension 2). The vertices, edges and cells, they are called the *faces*, with their incidence structure are called the *line arrangement* (of L). Two faces are called *incident* if one is contained in the relative closure of the other.

Lemma 3.5. $n \in \mathbb{N}_0$. *An arrangement of n lines has at most* $\binom{n}{2}$ *vertices, at most* n^2 *edges, and at most* $\binom{n+1}{2} + 1$ *cells. If no three lines intersect in a common point and no two lines are parallel, all of these bounds are attained.*

We define the complexity m_k of Λ_k as the number of edges from the arrangement of L in Λ_k; hence, the number of vertices incident to these edges is $m_k - 1$. Clearly, the k-level can be stored with $O(m_k)$ space, and we can decide for a given point q whether it lies below the level in $(\log m_k) = O(\log n)$ time (note that $m_k \leq n^2$ for sure): Namely, we sort the vertices of Λ_k by their x-coordinate, and then locate q's x-coordinate in this sequence, thus finding an edge e where q's x-coordinate is between two endpoints of e. Now we can simple compare q to this edge. This gives us an intermediate result where we can store the n lines in the plane with $O(n^2)$ space, so that the number of lines above a query point can be determined in time $O((\log n)^2)$.

To improve the runtime, we again call for fractional cascading, where we (at least conceptually) store the levels of an arrangement in a balanced tree for locating q among them. That is, we are able to cut down the query time to $O(\log n)$ without asymptotic increase in space. We can state the result in the following primal setting:

Theorem 3.6. *A set P of n points in the plane can be preprocessed with* $O(n^2)$ *storage so that the number of points below a non-vertical line can be computed in* $O(\log n)$.

3.3 Planar Point Location - More Examples

We have seen that the problem of counting the points (from a given set) below a query line can be translated to the problem of locating a point in a subdivision (partition) of the planee. The latter appears again and again. An obvious occurence is that of locating a point in a map.

3.3.1 Point Relative to Convex Polytope

Suppose we want to decide for a convex polytope $P \subseteq \mathbb{R}^3$ with $n \geq 4$ vertices whether it contains a query point $q \in \mathbb{R}^3$. As we have seen in AnW, the boundary of such a polytope decomposes into the n vertices, at most $3n - 6$ edges, and at most $2n-4$ facets. We can store the planes carrying the facets, and compare a query point with each of them to decide whether it lies in the polytope or not. This takes time and space $O(n)$. Point location after preprocessing allows improvement to $O(\log n)$ query time and linear space.
To this end, we can split the boundary of P into an upper and a lower part. To define these parts, note that any vertical line that meets the polytope intersects it in a vertical line segment (possibly a single point). The collection of topmost points of these segments forms the upper boundary, the bottommost form the lower boundary. Note that there are parts of the boundary that appear in both, and also some that appear in neither one.
The vertical projection of facets, edges, and vertices of the lower boundary gives a subdivision of a convex polygon in the plane. If a query point q projects to some point outside this polygon, it lies outside the polytope P. Otherwise it lies in the projection of some facet of the lower boundary of P. If this facet is determined by point location, we can compare q with this plane: If q is below the plane, q is outside. Otherwise we still have to decide whether it is also below the upper boundary of P in an analogous fashion.

3.3.2 Closest Point in the Plane - the Post Office Problem

We want to preprocess a set S of n points in such a way that for a query point q the point closest to q in S (may not be unique) is delivered. We follow the locus approach for this problem. Note that for two points p and p', the *bisector* b of p and p' separates between the two areas closer to p and p', respectively. Let us denote by $h(p, p')$ the open halfplane determined by b containing p. Then the set of points $V_S(p)$ closer to p than to any other point in S can be written as

$$V_S(p) := \bigcap_{p' \in S \setminus \{p\}} h(p, p').$$

It is called the *Voroni cell* of p (w.r.t. S). A Voroni cell is a convex set with a piecewise linear boundary. The collection of all Voroni cells results in the Voroni diagram of S. Point location in this structure solves closest neighbor queries.
We can show that the number of cells, edges, and vertices of the Voroni diagram of n points is $O(n)$.

3.4 Trapezoidal Decomposition

3.4.1 Some Definitions and Assumptions

A *segment* s in the plane is the convex hull of two points p_1 and p_2 in $\mathbb{R}^2$. The segment s deprived of its endpoints p_1, p_2 is called the *relative interior* of s.
A set S of segments is called *non-crossing*, if every segment in S is disjoint from

the relative interiors of the other segments. So segments may intersect only in their respective endpoints. P(S) is the set of endpoints of the segments of S. We call S in *general position*, if no two endpoints have the same x-coordinate. The set of relative interiors of the segments in S is denoted by S°. If S is non-crossing, for every point p in $\bigcap_{s \in S} s$ there is a unique element f in $P(S) \cup S^\circ$ with $p \in f$.
Furtheron we will assume that S is a set of n non-crossing segments in general position in the plane. A vertical upward ray emanating at a point q is either disjoint from S, or there is a first point p where it meets a segment (may be q itself). In the former case we define $above(q) =: \top$ (representing a symbolic segment above everything), or in the latter case above(q) is the unique f with $p \in f \in P(S) \cup S^\circ$.
Our goal is to preprocess S so that for any query point q the endpoint or segment *above(q)* above q can be computed quickly. We make simplifying assumptions, i.e. q won't lie on a segment and shares its x-coordinate with none of the endpoints in P(S).

3.4.2 Trapezoidal Decomposition

For every endpoint p in P(S) we consider two vertical extensions, one from p upward until the first segment in S° is met (or it extends to infinity), and one downward until the first segment in S° is met If neither case is true, we let it extend to infinity. The set of connected components of $\mathbb{R}^2$ without all segments in S and without all vertical (upward and downward) extensions from points in P(S) is called *trapezoidal decomposition of S* (including artifacts like triangles, infinite trapezoidal slabs, halfplanes, or even the whole plane if $S = \emptyset$), denoted by $\mathbb{T}(S)$. Trapezoids are convenient since they have constant size descriptions and we do not pay for this with an increase in the number of regions, as the following lemma shows:

Lemma 3.7. *For a set S of n non-crossing segments in general position in* $\mathbb{R}^2$ *we have* $|\mathbb{T}(S)| = O(n)$. *(Proof by number of edges of a planar graph).*

In fact, it can be shown that there are at most $3n + 1$ trapezoids in a planar graph.

3.4.3 The History Graph

For some ordering $\sigma = s_1, s_2, ..., s_n$ of S let $S_i := \{s_1, s_2, ..., s_i\}$. The history graph of S (w.r.t. σ) has vertex set $\bigcup_{i=0}^{n} \mathbb{T}(S_i)$. There is a directed edge from T to T' if there is an $i \in [n]$ such that $T \in \mathbb{T}(S_{i-1}) \setminus \mathbb{T}(S_i), T' \in \mathbb{T}(S_i) \setminus \mathbb{T}(S_{i-1})$, and $T \cap T' \neq \emptyset$. The source of the graph is $T_0 := \mathbb{R}^2$, and the sinks are the trapezoids in $\mathbb{T}(S)$.
We generate $\mathbb{T}(S)$ incrementally, adding one segment after the other. Whenever a trapezoid T disappears, we mark it as 'destroyed', and equip it with pointers to the new trapezoids that intersect the area of T. Location of a point q starts in the source and we proceed to the unique successor which contains q until we

find a leaf/sink (which then contains q). Let's take a look at the quality of the structure:

Observation 3.8. *Given $T \in \mathbb{T}(S_{i-1}) \setminus \mathbb{T}(S_i)$, the number of trapezoids in $\mathbb{T}(S_i)$ overlapping with T is at most 4.*

That is we can proceed in constant time to the successor trapezoid containing q. Roughly speaking, a trapezoid in $\mathbb{T}(S)$ is determined by four segments in S, two responsible for the upper and lower boundary resp., and two for left and right, resp., delimiting vertical extension.

Observation 3.9. *Given $T \in \mathbb{T}(S)$, there is a set $S' \subseteq S$ (not necessarily unique) of at most 4 segments, such that $T \in \mathbb{T}(S')$.*

That is, if $T \in \mathbb{T}(S)$, there are at most 4 segments in S whose removal lets T disappear (in a backwards process). A backward analysis argument shows that the expected number of trapezoids in $\mathbb{T}(S)$ containing q is at most $4H_n = O(\log n)$.
Similarly, the expected number of trapezoids ever removed in the backward process is $O(n)$.

Theorem 3.10. *Let S be a set of n non-crossing segments in general position in the plane. Assume a u.a.r. random ordering of the segments in S. Then the history graph has expected size $O(n)$. For any fixed $q \in \mathbb{R}^2$, the expected time for locating q (with the history graph structure) in $\mathbb{T}(S)$ is $O(\log n)$.*

The history graph can be computed in expected $O(n \log n)$. The general underlying principle, *randomized incremental construction* has numerous applications including computations of convex hulls or Voronoi diagrams.

4 Linear Programming

4.1 Basic Setting

LP is a powerful tool and extremely important in practice, especially for solving optimization problems. In a general LP we want to find a vector $x \in \mathbb{R}^n$ maximizing (or minimizing) the value of a given linear function among all vectors $x \in \mathbb{R}^n$ that satisfy a given system of linear equations and non-strict linear inequalities. The linear function to be maximized (minimized) is called the *objective function*. It has the form

$$c^T x = c_1 x_1 + ... + c_n x_n.$$

The linear equations and inequalities in the LP are called the *constraints*. Every vector x satisfying all constraints of a given LP is a *feasible solution*. Each x that gives the maximum possible value of $c^T x$ among all feasible x is called an *optimal solution*. A linear program may in general have a single optimal solution, infinitely many optimal solutions, or none at all.

Matrix Notation
Every linear program can also be converted into the following *equational form*

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

Note that we can convert a LP $Ax = b$ it to a system $Ax + \epsilon = b$, and replace each x_i by $x_i' = x_i''$. Therefore we obtain an equational form LP with $2n + m$ variables and $2n + 2m$ constraints. That is, all variables are required to be nonnegative, and besides this, there are only equality constraints.

On solving LPs
A linear program is efficiently solvable both in theory and in practice (many software packages available). In theory, algorithms have been developed that provably solve each linear program in time bounded by a certain polynomial function of the input size. The input size is measured as the total number of bits needed to write down all coefficients in the objective function and in the constraints. In practice, a number of software packages are available.

4.2 Direct Applications

Here we demonstrate a few tricks for reformulating problems that do not look like linear programs at first sight.

4.2.1 Ice Cream Manufacturer

An ice cream manufacturer needs to set up a production plan for the next year, based on the monthly sales. We introduce a cost for a change in the produced amount (e.g. workers have to be hire or laid off). We also introduce a cost for storing the ice cream if we have a surplus. We now search for a compromise minimizing the total cost resulting both from changes in production and from storage of surpluses.

Let us denote the demand in month i by $d_i \geq 0$. Then we introduce a nonnegative variable x_i for the production in month i and another nonnegative variable s_i for the total surplus in the store at the end of month i. To meet demand in month i, we use the production in month i and the surplus of the last month $i - 1$:

$$x_i + s_{i-1} > d_i, \forall i \in \{1, 2, ..., 12\}.$$

The quantity $x_i + s_{i-1} - d_i$ is exactly the surplus after month i, and thus we have

$$s_i + s_{i-1} - s_i = d_i, \forall i \in \{1, 2, ..., 12\}$$

We set $s_0 = s_{12} = 0$, assuming no surplus at the start and end of the year.

Among all nonnegative solutions of these equations and inequalitites we are

looking for the one that minimizes the total cost. We assume change of production cost of 50 and storage cost of 20 per unit ice cream. This gives us the total cost expressed by

$$50 \sum_{i=1}^{12} |x_i - x_{i-1}| + 20 \sum_{i=1}^{12} s_i,$$

where we set $x_0 = 0$ (history can easily be taken into account). Since this cost function is not linear, we have to make it linear with the following trick:
Note that the change in production is either an increase or decrease. Let us introduce a nonnegative variable y_i for the increase in month $i-1$ to month i, and a nonnegative variable z_i for the decrease. Then

$$x_i - x_{i-1} = y_i - z_i,$$

which leaves us with the following linear program:
We minimize $50 \sum_{i=1}^{12} y_i + 50 \sum_{i=1}^{12} z_i + 20 \sum_{i=1}^{12} s_i$ with subject to:

$$\begin{gathered} x_i + s_{i-1} - s_i = d_i \\ x_i - x_{i-1} = y_i - z_i \\ x_0 = s_0 = s_{12} = 0 \\ x_i, s_i, y_i, z_i > 0, \forall i \in \{1, 2, ..., 12\}. \end{gathered}$$

We have to note that one of y_i*, z_i* has to be zero for all i, otherwise we could decrease both and obtain a better solution. This implies that $y_i * + z_i* = |x_i - x_{i-1}|$, as required.

4.2.2 Fitting a Line

Suppose that we have some data points $(x_1, y_1), ..., (x_n, y_n)$, and we would like to fit a line approximating the dependence of y on x.
There is no unique way to solve this and several. criteria are commonly used, the most popular method though is *least squares*. We seek a line with equation $y = ax + b$ minimizing the expression

$$\sum_{i=1}^{n} (ax_i + b - y_i)^2.$$

In words, for every point we take its vertical distance from the line, square it, and sum these "squares of errors".
This method may not always be suitable. For instance to be less sensitive to a small number of outliers, we can also minimize the sum of absolute values of all errors. We can use a similar trick to the one we saw in the example before to capture this apparently nonlinear optimization problem by a linear program. We now minimize the sum over the auxiliary error variables $e_i, i \in \{1, ..., n\}$, with subject to the following contraints:

$$\begin{gathered} e_i \geq ax_i + b - y_i \\ e_i \leq -(ax_i + b - y_i) \end{gathered}$$

The constraints guarantee that

$$e_i < \max(ax_i + b - y_i, -(ax_i + b - y_i)) = |ax_i + b - y_i|.$$

In an optimal solution, each of these inequalities has to satisfied with equality, otherwise we could decrease the corresponding e_i.

4.3 Geometry of Linear Programs

4.3.1 Convex sets and Polyhedra

We recall some geometric notations.

- A set $C \subset \mathbb{R}^n$ is *convex*, if C contains the segment connecting every two of its points. That is, for every $x, y \in C$ and every $t \in [0, 1]$ we have $(1-t)x + ty \in C$.
- A **hyperplane** in $\mathbb{R}^n$ is an affine subspace of dimension $n-1$. In other words, t is a set of the form $\{x \in \mathbb{R}^n : a_1x_1 + ... + a_nx_n = b\}$, where $a_1, a_2, ..., a_n$ are not all 0.
- A (closed) *half-space* in $\mathbb{R}^n$ is a set of the form $\{x \in \mathbb{R}^n : a_1x_1 + ... + a_nx_n \leq b\}$, again with at least one a_i nonzero.
- A *convex polyhedron* in $\mathbb{R}^n$ is the intersection of finitely many closed half-spaces.

A convex polytope in $\mathbb{R}^n$ can be defined as a bounded convex polyhedron (i.e. one contained in a sufficiently large ball). Arbitrary convex polyhedra need not be bounded; an example is a single half space.

4.3.2 Basic feasible solutions

The set of all feasible solutions of a linear program is, more or less by definition, a convex polyhedron in $\mathbb{R}^n$. A *basic feasible solution* of a linear program is a feasible solution x for which n linearly independent constraints hold with equality. We note that for every n-tuple of linearly independent constraints, there is at most one point satisfying all of them with equality. Consequently, each linear program has only finitely many basic feasible solutions, namely at most $\binom{m}{n}$.
Geometrically, basic feasible solutions correspond to the vertices of the polyhedron P. A point $v \in P$ is called a *vertex*, if there is a linear function whose maximum over P is attained in v and nowhere else. That is, there is $c \in \mathbb{R}^n$ with $c^Tv > c^Tx$ for all $x \in P \setminus \{v\}$.
An arbitrary linear program may not have any basic feasible solutions at all (consider a single inequality constraint in $\mathbb{R}^2$). However, linear programs in equational form have "enough" basic feasible solutions.

Theorem 4.1. *Let us consider a linear program in equational form*

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

and let $P \subset \mathbb{R}^n$ be the convex polyhedron of all feasible solutions. If $P \neq \emptyset$ and the objective function c^Tx is bounded from above on P, then there exists a basic feasible solution that is optimal, and in particular, the linear program has an optimal solution.

This tells us that an optimal solution only does not exist, if there are no feasible solutions at all, or if the objective function is unbounded from above. The theorem is a consequence of the following statement:

> If the objective function is bounded from above, then for every feasible solution x there exists a basic feasible solution $\tilde{x}$ with $c^T\tilde{x} \geq c^T x$.

4.4 Bounds on solutions

4.4.1 Encoding of Linear Progams

For algorithmic purposes, we will consider only programs in which all of the coefficients are rational numbers (s.t. they have a finite encoding). The size of a linear program is the total number of bits needed to write all of the coefficients. More formally, the encoding size of an integer z is

$$\langle z \rangle := \lceil \log_2(|z| + 1) \rceil + 1$$

which is the length for a standard binary encoding plus one bit for the sign. Immediate useful facts are

$$\langle xy \rangle < \langle x \rangle + \langle y \rangle \text{ and } |x| < 2^{\langle x \rangle}$$

For a rational number $r = \frac{p}{q}$ with p and q relatively prime we have $\langle r \rangle := \langle p \rangle + \langle q \rangle$. For a rational matrix A is is the sum over the encoding sizes of all its entries. The same holds for a rational vector.
Then for a linear program L, the encoding size is the sum of encoding sizes of the vectors and matrices involved.

4.4.2 Bounds on optimal solutions

Theorem 4.2. *If a linear program L with rational coefficients has a feasible solution, then it also has a rational feasible solution $\tilde{x}$ with $\langle \tilde{x} \rangle = \mathcal{O}(\langle L \rangle)$ for every j (consequently, $\tilde{x}$ is contained in the cube $[-K, K]^n$ with $K \leq 2^{\mathcal{O}(\langle L \rangle)}$). A similar statement holds for optimal solutions.*

That means the encoding size of a feasible solution is at most the encoding size of the hole program, which might seam pretty weak at first sight. It it not that bad though, since a solution could also blow up exponentially. We note that actually $\langle \tilde{x_j} \rangle = \mathcal{O}(\langle L \rangle - \langle c \rangle)$, where c is the coefficient vector of the objective function.
In order to proof the theorem, we need bounds on the encoding size of the determinant of a rational matrix.

Lemma 4.3. *For a rational $n \times n$ matrix A we have $\langle \det(A) \rangle = \mathcal{O}(\langle A \rangle)$.*

The proofs of the above two statements can be found on p.93 of the script.

4.5 Duality and the Farkas lemma

We begin with a simpler version of the duality of linear programming, called the Farkas lemma.

4.5.1 The Farkas lemma

One can easily check that a given solution it correct. If we want to show that a system has no solution, one way is to write down a linear combination of the equations that is obviously inconsistent. Namely, if A is an $m \times n$ matrix and we exhibit a vector $y \in \mathbb{R}^n$, s.t. $y^T A = 0$ and $y^T b = 1$, then it is clear that $Ax = b$ has no solution.
Thus for the decision problem "does a given system of linear equations have a solution" there are easy certificates both for the YES and NO answers. The Farkas lemma provides easy certificates for unsolvability of systems of linear inequalities.

Lemma 4.4. *(Farkas lemma I). A system $Ax \leq b$ of linear inequalities is **unsolvable** if and only if there exists $y > 0$ such that $A^T y = 0$ and $b^T y < 0$.*

Lemma 4.5. *(Farkas lemma II). A system $Ax = b$ of linear equations has **no nonnegative** solution if and only if there exists y such that $A^T y > 0$ and $b^T y < 0$.*
*(Farkas lemma III). A system $Ax < b$ has **no nonnegative** solution if and only if there exists $y \geq 0$ such that $A^T y \geq 0$ and $b^T y < 0$.*

One can show that the three variants of the Farkas lemma are mutually equivalent. The geometric meaning of the lemma II is shown on p. 95 in the script.

4.5.2 The strong duality theorem

We now let **(P)** denote the linear program of the form

$$\text{maximize } c^T x \text{ subject to } Ax < b \text{ and } x > 0.$$

Similar to the Farkas lemma, we can also interpret the duality of linear programming as the existence of easy certificates for NO answers - but what is the question? Well, we ask "Is the optimal value of (P) greater or equal to some given number γ?".
A YES answer has an obvious certificate, namely a feasible solution $\tilde{x}$ with $c^T \tilde{x} > \gamma$. How can we hope to certify NO?
Suppose that we can make a nonnegative linear combination of the rows of the matrix A in which every coefficient is at least as large as the corresponding coefficient of c; i.e., we have a vector $y > 0$ with $y^T A \geq c$. Then the corresponding linear combination of the right-hand sides i.e., $y^T b$, is an upper bound for the maximum of $c^T x$. Indeed, we have $c^T x \leq (y^T A)x = y^T(Ax) < y^T b$, where the first inequality relies on $x > 0$ and the second on $y \geq 0$.
Now the task of finding the best y as above, i.e. one that gives the smallest upper bound on $c^T x$, can be written as a linear program **(D)**:

$$\text{minimize } b^T y \text{ subject t do } A^T y \geq c \text{ and } y \geq 0.$$

This is called the *dual* of the linear program (P).
By the previous considerations, we see that the minimum of (D) is equal to the maximum of (P), assuming (P) feasible and bounded. In this respect, the optimum of (D) is a perfect NO certificate.

Theorem 4.6. *(Strong duality theorem). For the linear programs (P) and (D) as above, exactly one of the following possibilities occurs:*

1. *Neither (P) nor (D) has a feasible solution.*
2. *(P) is unbounded and (D) has no feasible solution.*
3. *(P) has no feasible solution and (D) is unbounded.*
4. *Both (P) and (D) have a 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*

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

The duality theorem easily implies several significant min-max theorems such as the maxflow-mincut theorem, König's theorem about matchings in bipartite graphs, Hall's marriage theorem, and others.
The duality theorem is valid for each linear program, not only for one of the form (P); we have only to construct the dual linear program properly

4.6 The ellipsoid method

The ellipsoid method is an algorithm for certain nonlinear optimization problems. However, it has never been of practical interest for linear programming.

A **polynomial LP algorithm** is an algorithm for LP that finds a correct solution for every LP L with rational coefficients in at most $p(< L >)$ steps.
There are basically two classes of methods used in practice, each with many variations.
The *simplex method* is a basic strategy to go along the edges of the polyhedron of feasible solutions, from one basic feasible solution to another, in such a way that the value of the objective function improves each step.
The *interior point methods* start in the interior of the feasible polyhedron and proceed towards an optimal solution, in discrete steps but guided by certan smooth, analytically defined curve inside the polyhedron.
For very large (and usually sparse) LPs, interior point seems to be the winner.

4.6.1 The relaxed problem and the algorithm

The ellipsoid method does not directly solve a LP, but rather it seeks a solution of a system of linear inequalities $Ax \leq b$, if one exists - this is called the **feasibility problem**. A polynomial algorithm for the feasibility problem allows us

to solve a general LP in polynomial time. An elegant way of seeing this is via the strong duality theorem. For the linear program "maximize $c^T x$ subject to $Ax \leq b, x \geq 0$", we set up the following system of inequalities with vectors x and y:

$$Ax \leq b, x \geq 0, c^T x \geq b^T y, A^T y \geq c, y \geq 0.$$

By strong duality, this system is feasible exactly if the original LP has an optimal solution, and if $(\tilde{x}, \tilde{y})$ is a feasible solution of the system, then $\tilde{x}$ is an optimum of the original LP.

Let $P = \{x \in \mathbb{R}^n : Ax < b\}$ be the polyhedron consisting of all solutions of the considered system of inequalities; we thus want to find a point $y \in P$, or conclude that $P \neq 0$. Then the relaxed feasibility problem can be described as follows:
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 $y \in P$, or the answer NO SOLUTION.

The ellipsoid algorithm for the above problem generates a sequence of ellipsoids $E_0, E_1, ..., E_t$, each of them guaranteed to contain all of P. A rough outline is as follows:

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

A crucial geometric fact is captured by the following lemma:

Lemma 4.7. *Let E be an n-dimensional ellipsoid in $\mathbb{R}^n$ 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*

$$volE' < \rho \cdot volE, \text{ where } \rho = \rho(n) := e^{-1/(2n+2)}.$$

Therefore it is easy to bound the number of iterations of the above algorithm. We get an upper bound of $\lceil n(2n+2)\ln(R/\epsilon) \rceil$ on the maximum number of iterations. In order to obtain a polynomial time algorithm we have to show the following two properties: (A) that the iterations in the above algorithm can

be implemented in polynomial time, and (B), that given the algorithm one can obtain a polynomial algorithm for deciding the feasibility of an arbitrary system $Ax \leq b$.

4.6.2 Geometry of ellipsoids

Shows that the ellipsoid always exists and that the volume shrinks with a lower bound, p.103ff in script (also covers issue (A)).

4.6.3 Solving LPs using the relaxed feasibility problem

We now explain how the algorithm for the relaxed feasibility problem can be used to decide the feasibility of an arbitrary system $Ax \leq b$, while preserving polynomial running time. Let us write $\phi = \langle A \rangle + \langle b \rangle$ for the encoding size of the considered system. We will construct a new system $\hat{A}x < \hat{b}$, whose encoding size is polynomial in ϕ, and such that

- $\hat{P} \subset B(0, R)$, where $\langle R \rangle$ is polynomial in ϕ.
- If $P \neq \emptyset$, then $\hat{P}$ contains an ϵ-ball, with $\epsilon > 0$ and $\langle \epsilon \rangle$ polynomial in ϕ.
- If $P = \emptyset$, then $\hat{P} = \emptyset$

Making the polyhedron bounded is easy using Theorem 4.2 (bounded encoding size). Next we have to take care of "thickening" the polyhedron so that, if it was nonempty, it will contain an ϵ-ball. The nontrivial part is to make sure that thickening an empty polyhedron cannot create a nonempty one.

Lemma 4.8. *Let $Ax < b$ be a system of inequalities with rational coefficients and encoding size ϕ, let P be its polyhedron of feasible solutions, and let P_μ be the polyhedron of the system $Ax \leq b + \eta 1$, where 1 is the vector of all 1s. For $\eta = 2^{-C_2\phi}$, with a sufficiently large constant C_2, the following hold.*

- *(a) If $P \neq 0$, then P_η contains an ϵ-ball for $\epsilon = \eta/2^\phi$.*
- *(b) If $P = 0$, then $P_\eta = \emptyset$.*

Note that the algorithm so far only decides whether the system $Ax \leq b$ has a solution, but it does not necessarily provide one, since it only provides a solution of the auxiliary system. Finding one is left as an exercise.

4.7 Traveling Salesman

Given a graph $G = (V, E)$ with $c_e \in \mathbb{R}, e \in E$, a **tour** t is a spanning cycle (also called Hamiltonian cycle), formally a subset E_t of E such that the graph (V, E_t) is connected and every vertex is incident to exactly two edges in E_t. The cost of tour t is defined as $\sum_{e \in E_t} c_e$. An optimal tour in G is a tour of minimal cost. The task of computing an optimal tour for a given graph is called the *travelling salesman problem* (TSP) (a classical NP-complete problem).
For $S \subset V$, let $\delta(S) := \{e \in E : |e \cap S| = 1\}$, sometimes called the *boundary*

of S or the edge set of the cut $(S, V \setminus S)$. With $\delta(v)$ we mean the set of edges incident to v. With this we can specify the characteristic vectors of edge sets of tours by the following constraints.

$$x \in \{0,1\}^E$$
$$\sum_{e \in \delta(v)} x_e = 2, \forall v \in V, \text{ and}$$
$$\sum_{e \in \delta(S)} x_e >= 1, \forall S \subseteq V, \emptyset \neq S \neq V.$$

Therefore every nontrivial cut must contain at least one edge (connected graph). For every edge set of a tour, every such cut must indeed have at least two edges (cut must have even size). Therefore we can substitute and end up with

$$\sum_{e \in \delta(S)} x_e \geq 2, \forall S \subseteq V, \emptyset \neq S \neq V.$$

While these constraints are equivalent in the integer program, the resulting LP relaxation of the TSP is more constrained and therefore its optimal solution is hopefully closer to the integer solution. Consider the **Subtour LP**:

$$\min c^T x \text{ subject to:}$$
$$\sum_{e \in \delta(v)} x_e = 2, \forall v \in V,$$
$$\sum_{e \in \delta(S)} x_e > 2, \forall S \subset V, \emptyset \neq S \neq V,$$
$$\text{with non integral } 1 > x_e > 0, \forall e \in E.$$

An optimal solution $\tilde{x}$ of this Subtour LP (or also Held-Karp relaxation) is a lower bound on an optimal solution $x* \in \{0,1\}^E$ to the underlying integer program, i.e.

$$c^T \tilde{x} \leq c^T x* = OPT_{tour}.$$

with OPT_{tour} the cost of the optimal tour in G with costs c. A graph $G = (V, E)$ satisfies the *triangle inequality* if $E = \binom{V}{2}$ and the direct distance between every two vertices u and w is smaller or equal than the distance from u to any v in addition to v to w. With this inequality being satisfied, it can be shown that

$$c^T \tilde{x} \leq c^T x* = OPT_{tour} \leq \frac{3}{2} c^T \tilde{x}.$$

Note that a *non-hamiltonian* graph (such as the Petersen-graph) can have a Subtour solution that is not integral (and the integral LP has no solution). There is a worry looming though: the subtour LP has an exponential number of constraints $(n + (2^n - 2) + 2m)$. So can we solve it efficiently? We know, that we can find a violated cut constraint, if it exists, via a min-cut algorithm in polynomial time. This provides exactly the type of polynomial separation oracle we need for an efficient employment of the ellipsoid method.

4.8 Minimum Spanning Tree

Given a graph $G = (V, E)$ we search for a connected subgraph $T = (V, E')$ that is connected and has no cycle with minimum cost of edges. It is well known that

there are several equivalent characterizations of a tree, e.g. *(i) it has exactly n - 1 edges, and (ii) it is connected*. This suggests to express the characteristic vectors of spanning trees by the following constraints:

$$x \in \{0,1\}^E, \sum_{e \in E} x_e = n-1, \text{ and } \sum_{e \in \delta(S)} x_e \geq 1, \forall S \subseteq V, \emptyset \neq S \neq V.$$

We now consider the LP relaxation. The *Loose ST LP for G with* $c \in \mathbb{R}$ is:

$$\begin{gathered} \min c^T x \text{ subject to} \\ \sum_{e \in E} x_e = n-1 \text{ (edge count)} \\ \sum_{e \in \delta(S)} x_e > 1, \forall S \subset V, \emptyset \neq S \neq V \text{ (connected)} \\ 1 > x_e > 0, \forall e \in E. \end{gathered}$$

Let the graph $G_{k,l}$ consist of k + l - 1 vertices and be a k-clique with a path of length l attached to two vertices of the clique, that has costs $c_e = 0$ for all edges of the k-clique and, for some real positive number $\gamma, c_e := \gamma$ for all edges on the path). Then the following lemma gives us an approximation of the loose LP.

Lemma 4.9. *(i) For the graph $G_{k,l}$ with costs as described above a minimum spanning tree has cost $(l-1)\gamma$. (ii) The corresponding Loose ST LP has a value of at most $\frac{l}{2}\gamma$ if $l = k(k-3)+4$.*

Therefore we see that the Loose ST LP allows a fractional solution that is roughly a factor $\frac{1}{2}$ smaller than the cost of a MST. An even better LP is possible. For that we make use of the tree property of having no cycle. This leads us to the *Tight ST LP*:

$$\begin{gathered} \min c^T x \text{ subject to} \\ \sum_{e \in E} x_e = n-1 \text{ (edge count)} \\ \sum_{e \in E \cap \binom{S}{2}} x_e \leq |S| - 1, \forall S \subset V, \emptyset \neq S \neq V \\ 1 \geq x_e \geq 0, \forall e \in E. \end{gathered}$$

Theorem 4.10. *(Edmonds, 1970). Every basic feasible solution of the Tight Spanning Tree LP is integral. Therefore, the value of the Tight Spanning Tree LP equals the cost of the MST for every cost vector c.*

4.9 Back to the Subtour LP

We have seen a "perfect" LP for the MST problem and would like to know what the situation with the subtour LP is. For $\tilde{x}$ an optimal solution and x^* for the integral counterpart, we are interested in how big the ratio $\frac{c^T x^*}{c^T \tilde{x}}$ may get. This is called the *integrality ratio* (which is always 1 for the Tight ST LP).
We can always obtain a so-called *graph metric* cost function on the complete graph by using the costs of paths between two vertices that do not have an direct edge between each other.

Lemma 4.11. *Let the graph 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 one.*

Now we consider a specific graph G_k with $n := 3k$ vertices. It consists of three disjoint paths of length $k-1$ (i.e. k vertices each), plus pairwise edges between the three starting points and also between the three end points of these paths. Let G'_k be the graph with the graphic metric induced by G_k.

Lemma 4.12. *Let $k \in \mathbb{N}$ and let $n := 3k$ (the number of vertices of G_k 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 $\frac{4}{3} - O(1)$ and therefore every tour in G'_k has cost at least $\frac{4}{3}n - O(1)$.*

We conclude that the Subtour LP is not perfect and that we have a family of graphs G'_k with costs c which exhibit an integrality ration of roughly $\frac{4}{3}$ as the size n grows.

4.10 Subtour LP versus Tight Spanning Tree LP

We now relate the two LP to get an upper bound on the integrality ration of the Subtour LP.

Lemma 4.13. *For a given graph $G = (V, E)$, if $x \in \mathbb{R}^E$ is a feasible solution of the Subtour LP, then $\frac{n-1}{n}x$ is a feasible solution of the Tight Spanning Tree LP.*

Corollary 4.14. *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 $c^T\tilde{x} \geq \frac{n}{n-1}OPT_{mst}$.*

Recall that, under the assumption of the triangle inequality, we have

$$OPT_{tour} \leq 2OPT_{mst}$$

Corollary 4.15. *For a graph G whose costs c satisfy the triangle inequality the integrality ratio of the Subtour LP is at most 2.*

We now describe the *Christofides approximation*, which generatres a tour of at most $\frac{3}{2}$ times the cost of the optimal tour as follows:

1. Choose a MST of optimal cost
2. For U the set of vertices of odd degree in this tree, compute the minimum weight matching $OPT_{match(U)}$ covering all vertices in U (this set is of even size).
3. The union of the MST and the matching is Eulerian (i.e. connected and all vertices have even degree, if edges both in tree and matching are counted twice).
4. Such a walk can be turned into a tour of at most this cost (exploiting the triangle inequality).

We can conclude that

$$OPT_{tour} \leq OPTmst + OPTmatch(U).$$

Besides $OPT_{mst} \leq c^T \tilde{x}$ one can also show that $OPT_{match(U)} < \frac{1}{2} c^T \tilde{x}$. Therefore we can conclude:

Theorem 4.16.

If G is a complete graph with costs satisfying the triangle inequality and if $\tilde{x}$ is an optimal solution to the Subtour LP for G, then

$$c^T \tilde{x} \leq OPT_{tour} \leq \frac{3}{2} c^T \tilde{x}.$$

(for OPT_{tour} the cost of an optimal tour in G). For all $\epsilon > 0$ there exists a graph with costs satisfying the triangle inequality such that $OPT_{tour} > (\frac{4}{3} - \epsilon) c^T \tilde{x}$.

5 Randomized Algebraic Algorithms

In this chapter we consider algorithms based on the idea of probabilistic checking, which proved extremely fruitful in computer science. It led to the celebrated PCP theorem, which essentially says that correctness of every solution of a problem in NP can be checked extremely fast.

5.1 Checking Matrix Multiplication

Given two matrices A and B and a black box program that solves the matrix multiplication extremely fast, we want to check whether the output C = AB is valid. In the following we describe a probabilistic approach to checking the validity.

Using a random number generator, we pick a random n-component vector x of zeros and ones. Therefore each vector $x \in \{0,1\}^n$ has the same probability 2^{-n}. The algorithm computes the products Cx and ABx with both $O(n^2)$ operations and outputs YES if the results are equal, and NO if not.
If C = AB then the algorithm will always answer YES, which is correct. But if $C \neq AB$, it can answer both YES and NO. We claim that we detect a wrong matrix multiplication with probability at least $\frac{1}{2}$. With $y := Dx$ and $d_{ij} \neq 0$, we have each entry $y_i = 0$ with probability at most $\frac{1}{2}$. Imagine we choose the values of the entries of x according to coin tosses, and we only look at the last coin toss. Then the sum up to the last toss is either unchanged ($x_j = 0$) or we add the nonzero number d_{ij} to it. For this case we have the desired probability of at most $\frac{1}{2}$ as claimed. With *probability amplification* we achieve an error probability of 2^{-r}, with r repetitions.

5.2 Is a Polynomial Identically Zero?

We often consider a polynomial given as a black box, that evaluates the polynomial at any given point. We can never be sure if the polynomial is zero everywhere unless we evaluate it at all points, but polynomials of low degree

have the property that they are either zero everywhere, or nonzero almost everywhere. Therefore two low-degree polynomials either coincide or they differ almost everywhere.
For *univariate polynomials* (with single variable) we know from algebra that if it has coefficients from a field and degree at most d, then it has at most d zeros. Therefore it suffices to evaluate it at $d+1$ points.
For polynomials in more than one variable it is more complicated since the zero set of *multivariate polynomials* can be rather complicated. For the following theorem, let $\mathbb{F}[x_1, ..., x_n]$ denote the ring of all polynomials in the variables $x_1, ..., x_n$ with coefficients in $\mathbb{F}$.

Theorem 5.1. ***(Schwartz-Zippel theorem).*** *Let $p(x_1, ..., x_n) \in \mathbb{F}[x_1, ..., x_n]$ be a (nonzero) polynomial of degree $d > 0$, and let $S \subset \mathbb{F}$ 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 $\frac{d}{|S|}$.*

5.3 Testing for Perfect Bipartite Matchings

We want to know whether a graph G has a *perfect matching*, i.e. a matching covering all vertices. A maximum matching can be computes in time $O(m\sqrt{n})$, but known algorithms are rather complicated. Here we explain very simple algorithms testing for the existence using the *Schwartz-Zippel theorem* by testing whether a suitable associated with the considered graph is zero.
Since we consider bipartite graphs, we have color classes U and V with vertices. Then a perfect matching in G, if one exists, corresponds to a permutation π of numbers $\{1, ..., n\}$ and has the form $\{\{u_i, v_{\pi(i)}\}\}, i = 1, ..., n$. Let us define the matrix A as follows:

$$a_{ij} := \begin{cases} x_{ij}, \text{ if } \{u_i, v_j\} \in E(G) \\ 0, \text{ otherwise.} \end{cases}$$

So A is a function of the x_{ij}, and $det(A)$ is a polynomial in these $|E(G)|$ variables.

Lemma 5.2. *The graph G has a perfect matching if and only if the polynomial $det(A)$ is not identically zero.*

The determinant of A can thus be used for testing whether G has a perfect matching, but we cannot afford to compute it, since it may have exponentially many terms, but we can use the Schwartz-Zippel theorem.
Since $deg(det(A)) \leq n$, we need a set S of size larger than n, say $2n$, for choosing the random values of the x_{ij}. To make computation less complex, it is better to work with a *finite field*. For that we choose a prime number p, $2n \leq p \leq 4n$, and regard $det(A)$ as a polynomial with coefficients from *GF(p)*, the field of integers modulo p. Then the above lemma still holds and we can compute the determinant of the matrix over GF(p) in $O(n^3)$ arithmetic operations using gaussian elimination. The arithmetic operations in GF(p) are fast if we prepare a table of inverse elements in advance.

Summarizing, we have an algorithm that can test whether a given bipartite graph has a perfect matching, has running time $O(M(n))$ (where $M(n)$ is the time for matrix multiplication) and has a failure probability of at most $\frac{1}{2}$. As usual, the failure probability can be made smaller than any given $\delta > 0$ by $O(\log \frac{1}{d})$ repetitions.

5.4 Perfect Matchings in General Graphs

First let's recapitulate some basic notations about permutations. Let S_n be the set of all bijective mappings $\{1..n\} \to \{1..n\}$. The parity of the number of transpositions in a representation of a permutation π is an invariant of π and it defines the sign of π: $sign(\pi) = +1$ if the number of transpositions is even, otherwise it's -1.
Given a permutation $\pi \in S_n$, the set $\{(i, \pi(i)|i \in \{1..n\}\}$ constitutes a set of directed edges on the vertex set $\{1..n\}$. Every vertex has out- and in-degree 1, and therefore it partitions into cycles (some may be loops of length 1). The sign of a permutation is given as $(-1)^{\text{number of even cycles}}$.

The algorithm from the previous section can be extended to arbitrary, non-bipartite graphs, but we need to work with another matrix. For a graph G with vertex set V, we introduce variables x_{ij} and define the *Tutte matrix* A of G by

$$a_{ij} := \begin{cases} +x_{ij}, \text{ if } i < j \text{ and } \{v_i, v_j\} \in E(G), \\ -x_{ij}, \text{ if } i > j \text{ and } \{v_i, v_j\} \in E(G) \\ 0, \text{ otherwise.} \end{cases}$$

Theorem 5.3. *(Tutte). The polynomial $det(A)$ is nonzero if and only if G has a perfect matching.*

Proof Sketch. One direction is easy. If G has a perfect matching, set $x_{ij} = 1$ if the edge is in the matching, and 0 otherwise. Then the resulting matrix has exactly one nonzero entry $+1$ or -1 in every row and the determinant will be ± 1. The opposite direction is harder than for the bipartite case, because it is no longer true that nonzero terms in the expansion of the $det(A)$ are contributed only by perfect matchings. To prove this direction, we must show that all terms that do not come from a perfect matching always cancel out. Since a permutation of the graph will only consist of cycles, we check what terms contribute to the determinant of the *Tutte* matrix. The determinant is defined as

$$det(A) = \sum_{\pi \in S_n} sign(\pi) a_{1,\pi(1)} ... a_{n,\pi(n)}.$$

We call π *important* if $a(\pi) \neq 0$. We define $E_\pi := \{\{v_i, v_{\pi(i)}\} : i = 1, 2, ..., n\}$, so E_π is an undirected version of the graph of π and π is important iff $E_\pi \subseteq E(G)$. In particular, we note that the graph of an important π has no loops. We want

to show that *if det(A) $\neq 0$, then there is at least one important π with all cycles of even length.* This is sufficient since if all cycles have even length, then E_π contains a perfect matching, and thus G has a perfect matching as well. This is done in detail on p. 134 of the script.
Using Tutte's theorem and tools from the previous sections we obtain a randomized algorithm for testing whether a given graph has a perfect matching, with $O(M(n))$ running time and probability of failure at most $\frac{1}{2}$.

5.5 Comparison with Other Matching Algorithms

The randomized testing algorithm can be extended to also find a maximum matching (not necessarily perfect). The randomized approach via determinants can also efficiently be parallelized. No fast versions are know for deterministic maximum matching algorithms.
The randomized approach can also solve other matching-type problems for which no polynomial-time deterministic algorithms are known. For example, suppose that each edge of a given bipartite graph is colored red or blue. Does there exist a perfect matching using exactly k red edges? The algorithm from section 5.3 can be extended to solve this problem (left as an exercise).

5.6 Counting Perfect Matchings in Planar Graphs

Sometimes it is even possible to compute the number of perfect matchigs in a graph G, denoted by **pm(G)**. For that matter we define for $G = (V_G, E_G), V_G = \{1, ..n\}$ a so called *orientation* of G, i.e. a $\vec{G} = (V_G, E_{\vec{G}})$ where every edge $\{i, j\}$ in G is represented by exactly one of (i, j) or (j, i) in $\vec{G}$. For such an orientation we define the following matrix:

$$A_S(\vec{G}) = (a_{ij} \in i, j = 1^n \in 0, +1, -1^{n\times n}, \text{where}$$
$$a_{ij} := \begin{cases} +1, \text{ if } (i,j) \in E_{\vec{G}}, \\ -1, \text{ if } (j,i) \in E_{\vec{G}}, \\ 0, \text{otherwise.} \end{cases}$$

This is a *skew-symmetric* matrix and nothing else but the previously seen *Tutte* matrix where we have plugged in $+1$ or -1 for the variables $x_{ij}, i < j, \{i, j\} \in E_G$.

Lemma 5.4. $\det(A_S(\vec{G}) \leq pm(G)^2$

Proof. We will sketch the proof from script p. 138 here. It makes use of the fact that for two perfect matchings in G, their union is an edge set consisting of even cycles and independent edges. Then if **U** denotes the set of all unions of two perfect matchings in G and $\|U\|, U \in \mathbf{U}$ is the number of even cycles in U, then $pm(G)^2 = \sum_{U\in\mathbb{U}} 2^{\|U\|}$. This is exactly the number of important permutations with all cycles even, which we obtain via the determinant. □

We still have to find an orientation, such that the determinant will achieve the desired value. This is indeed possible for planar graphs. Let C be an unidirected cycle in G of even length. We call C *oddly oriented* in $\vec{G}$ if going through C is some direction we encounter an odd number of edges in $\vec{G}$ oriented in our traversal direction - and, therefore, also an odd number of edges oriented in the opposite direction (hence, since C is even, the definition does not depend on the direction in which we chose to traverse C).
For a cycle in G with vertex set V_G, we call C *nice* if $G[V_G \setminus V_C]$ has a perfect matching. Observe that if G has any perfect matching, then n has to be even and every nice cycle has to be even. So "nice" and "oddly oriented" let us nicely express a sufficient condition for what we want:

Lemma 5.5. *If every nice cycle in G is oddly oriented in $\vec{G}$, then*

$$\det(A_S(\vec{G}) = pm(G)^2.$$

Such orientations are called Pfaffian.

Planar graphs are graphs that can be drawn in the plane so that no pair of edges crosses. *Euler's relation* tells us that $v - e + f = 2$ (don't forget to count the unbounded face).
A maximal planar graph (where no further edge can be added without violating planarity) is called a *triangulation.*

Lemma 5.6. *Let $\vec{T}$ be a plane oriented triangulation with at least 3 vertices where every finite face (a triangle) has an odd number of edges oriented clockwise.*

- *Let C be an undirected cycle in T with k of its edges oriented clockwise in $\vec{T}$ and with v vertices of T inside C (i.e. in the region surrounded by C, not including C). Then $k = v + 1(mod 2)$*
- *$\vec{T}$ is a Pfaffian orientation.*

Theorem 5.7. *(Kasteleyn) Every planar graph has a Pfaffian orientation which can be computed in linear time.*

Observe that if a graph G has a Pfaffian orientation $\vec{G}$, then all subgraphs of $\vec{G}$ (with some edges removed) are Pfaffian orientations. This holds, since removing edges just means setting some entries in the matrix to 0.

Corollary 5.8. *The number of perfect matchings in a planar graph can be determined in polynomial time. The number of perfect matchings in a planar graph with n vertices is at most $n^{\frac{n}{4}}$*

6 Parallel Algorithms

6.1 Warm Up: Adding Two Numbers

Suppose we are given two n-bit binary numbers a and b, where a_i, b_i denote the i^{th} least significant bits of a and b, respectively. A basic algorithm for computing the summation $a + b$ is *carry-ripple*. We compute the output bits by adding the bits from a and b, producing a carry bit c_i, which is added to the next higher bits a_{i+1}, b_{i+1}. This process can be built as a Boolean circuit with $O(n)$ AND, OR and NOT gates. This cannot be parallelized since each bit of the output needs to wait for the computation of the previous carry bit.

To make it parallelizable, we introduce the *carry-look ahead* algorithm. Note that once we can compute all the carry bits, we can complete the computation in O(1) time with n processors available. Also note that for the carry bit if we have $a_i = b_i = 1$, then $c_i = 1$; if $a_i = b_i = 0$, then $c_i = 0$; and if $a_i \neq b_i$, then $c_i = c_{i-1}$. For obvious reasons we can determine an $x_i \in \{g, k, p\}$ which indicates if a carry bit is *generated(g), killed(k),* or *propagated(p).*
Now let $y_0 = k$ and define $y_i \in \{k, p, g\}$ as $y_i = y_{i-1} \times x$. y_0 indiates that there is no carry before the first bit. Once we compute y_i we know the carry bit c_i, in particular $y_i = k$ implies $c_i = 0$, $y_i = g$ means $c_i = 1$. Note that we can never have $y_i = p$, since we then would propagate until the end. Computing y_i is a simple task for parallelism, and is known as *Parallel Prefix.*
A classic method for it is using a binary tree on top of the indices $\{1, ..., n\}$, given that the x_i are known. We pass up the product of all descendants, toward to root in $O(\log n)$ parallel steps. Then, using $O(\log n)$ extra steps we pass down the indices necessary to compute all the y_i. The computation is made of $O(\log n)$ steps, where in each step (layer in the tree) all tasks can be performed in parallel. Moreover, each step involves at most $O(n)$ computations. In fact, we have $O(n)$ computations in total.
The whole process can be built with a circuit of $O(\log n)$ depth and $O(n)$ gates.

6.2 Models and Basic Concepts

6.2.1 Circuits

We consider *Boolean circuits* made of AND, OR and NOT gates, connected by wires. Two key measures are the number of gates in the circuit and the depth (time needed for producing output). We assume that each gate operation takes one time-unit (i.e. one clock cycle in synchronous circuits).
There exist two distinctions on the number of inputs. The *Nick's class* NC(i) denotes the set of all decision problems that can be decided with 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. The *Alternative class* AC(i) covers the case where we have gates with unbounded fan-in, *poly(n)* gates and depth at most $O(\log^i n)$.

Lemma 6.1. *For any k, we have* $NC(k) \subseteq AC(k) \subseteq NC(k+1)$.
We define $NC = \cup_i NC(i) = \cup_i AC(i)$.

6.2.2 Parallel Random Access Machines (PRAM)

In this model we consider p number of RAM processors, each with its own local registers, which all have access to a global memory. In each synchronous time step, each processor can do a RAM operations or it can read/write to one global memory location. We have four variations with regard to concurrent reads and writes are resolved: Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), ERCW, and CRCW. When concurrent writes to one location are allowed, there are different rules of how to determine output, e.g. we can choose an arbitrary write that takes effect. Similar to NC, we use e.g. CRCW(k) to denote decision problems that can be computed by the correspondig version of the PRAM model with *poly(n)* processors and $O(\log^k n)$ time steps.

Lemma 6.2. *For any k, we have* $CRCW(k) \subseteq EREW(k+1)$.

It can be seen that PRAM can simulate circuits and vice versa. As a consequence, we see that $NC = \cup_k EREW(k)$

6.2.3 Some Basic Problems

Parallel Prefix
In this problem we have an input array A of length n and want to compute $B[j] = \sum_{i=1}^{j} A[i]$, i.e. the prefix sums. An $O(\log n)$ time EREW PRAM algorithm for this can be constructed from the binary tree idea discussed in chapter 6.1, that uses $O(n)$ processors and also performs $O(n)$ computations in total.

List Ranking (Suffix Sum) via Pointer Jumping
We are given a linked list represented by a content (value) array $c[1..n]$ and a successor pointer array $s[1..n]$. The desired output is to know all the suffix sums, from any starting point in the linked list to the end of it. For an $O(\log n)$-time EREW PRAM algorithm we make use of *pointer jumping*, which gets frequently used in parallel algorithms.
We have $\log n$ iterations, each of which has two steps:

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 $s(i) = s(s(i))$, which is called pointer jumping.

Lemma 6.3. *At the end of the above process, for each* $i \in \{1, ...n\}$*, c(i) is equal to the summation of the values from the location of that element to the end of the linked list.*

The above algorithm uses $O(\log n)$ time and $O(n \log n)$ computations, where we need at least n processors to start gaining (which is bad).

6.2.4 Work-Efficient Parallel Algorithms

We refer to the total number of rounds as the *depth* of the computation and we refer to the summation of the number of computations over all the rounds as *total work*. The primary goal is to have small depth, a secondary goal is to have small total work. We can relate this depth and work to an actual time measure for our computation on parallel processing systems, depending on how many processors we have.

Theorem 6.4. *If an algorithm does x computations in total and has depth t (i.e. t rounds or perhaps more formally a critical path of length t), then using p processors, this algorithm can be run in $x/p + t$ time.*

The above principle assumes that we are able to schedule the computational task of each round on the processors in an ideal way and in particular, it is possible for each processor to determine the steps that it needs to simulate in an online fashion. This can be quite non-trivial.

It is desirable to have a total amount of work that is proportional to the amount of work in the sequential case. We call such parallel algorithms *work-efficient*. Once we have such an algorithm, there is no significant loss due to the parallelism. In the next sections we want to obtain work-efficient algorithms.

6.3 Lists and Trees

6.3.1 List Ranking

Note that the list ranking problem is closely related to the parallel prefix problem, where we already achieved a work-efficient algorithm. The difficulty for list ranking stems from the issue that an element of the linked list does not know whether it is at an odd or even position. This makes it hard so structure the elements in the lower level of the recursion tree.
To overcome this problem, we notice that we do not need exactly even positions, it suffices to construct a set S of cn elements in the list, for some $c < 1$, such that the distance between any two consecutive members of S is small. Then we can solve the linked list problem by recursion, as follows:

(A) Build a contracted list for S, where each element in S has the first next linked-list element that is in S as its successor. The value of each element in the new linked list would be the sum of the values of the elements starting from (and including) itself and ending right before the first next element in S (i.e. its new successor).

(B) Recursively solve the problem for this contracted list of size $|S|$.

(C) Extend the solution to all elements, in time proportional to the maximum distance between two consecutive elements of S in the original list, and with work proportional to the length of the original list.

Repeating the above idea recursively and using the known linked list algorithm as soon as the list size falls below $n/\log n$, we obtain a solution with $O(\log n)$ time and $O(n/\log n \cdot log(n/\log n) = O(n))$ total work. Note that after $O(\log\log n)$ repetitions, we reach the case were the size is sufficiently small.
We now have to discuss how to choose S and how to "compact" is, as desired in part (A).

Selecting S
We use a simple randomized idea to mark an *independent* set I of elements, i.e. a set such that no two elements of it are consecutive in the linked list. We will then remove the elements of I from the list and thus the set S will simply be the resulting list.
First use a fair coin to mark each element of the linked list as head or tail. An element is in I, if and only if it holds a head and its successor holds a tail coin. Note that we have $\mu = E[I] \geq n/8$ and to be in I is independent for different pairs of elements. Using the Chernoff bound we find out that with large probability we have $|S| \leq 15n/16$.

Compacting the Linked List
To prepare for recursion we now have to build linked lists for content and successor pointer of S. We can solve the compaction problem using the parallel prefix algorithm. The numbering of items in S using distinct numbers from $\{1, 2, ..., |S|\}$ (ignoring their order in the linked list) can be done using parallel prefix by starting with 1 for each item in S and 0 for each item in I, and then computing the parallel prefix on the array that keeps the items. Once we have these numbers it takes a few simple operations to effectively remove the elements of I and set the appropriate content and successor arrays for those in S. Since I is an independent set, for each element $s \in S$, the successor of S in the new linked list is either its previous successor (in case that one is in S) or the successor of its previous successor, which can be found in $O(1)$ depth, for all elements in S in parallel, using $O(|S|)$ total computation. Similarly, we can prepare the content for the new linked list. The new content is either just the previous content, or the summation of previous content of s and its successor.

Overall Complexity
The above approach via parallel prefix requires $O(n)$ work and $O(\log n)$ depth, for just one level of computation. Over all the $O(\log\log n)$ compaction levels until we reach size $n/\log n$ we use $O(n)$ computational work and $O(\log n \cdot \log\log n)$ depth. While this algorithm is work-efficient, this depth bound is an $O(\log\log n)$ factor away from the ideal bound. There are known methods for reducing the depth complexity to $O(\log n)$.
We will often make use of the list ranking problem as subroutine and then assume that we have such an algorithm with ideal bounds, i.e. $O(\log n)$ depth and $O(n)$ work.

6.3.2 The Euler Tour Technique

In this section we introduce a technique that leads to work-efficient algorithms with $O(\log n)$ depth for a number of problems related to tree structures. Suppose we have a tree $T = (V, E)$ given as adjacency lists of its vertices. Now consider a directed variant $T' = (V, E')$, where each edge is replaced with two directed arcs. Then T' has an Eulerian tour, which can be defined easily by identifying for each arc $\langle u, v\rangle \in E'$ the successor arc $\langle v, w\rangle$ that the cycle should take after arriving at node v through arc $\langle u, v\rangle$. One can similarly define predecessor pointers.

Problem 1: Rooting the Tree and Determining Parents
Break the Eulerian cycle into a path by removing the last incoming arc of the root node r. We then search for a numbering of the arcs in the cycle, such that it is monotonically increasing along the successor pointers of the arcs of the Eulerian cycle. This can be computed by an instance of the *list ranking* problem, where we put the content of each arc to be 1 and then we compute all the partial prefix sums of these contents, according to the linked list of successor pointers. We already know that this can be done with $O(\log n)$ depth and $O(n)$ time. The parents are now easy to identify, since each edge defines such a relation and it can be read from the numbering that we obtained from the parallel prefix. We can check this for all edges in parallel.

Problem 2: Computing a Pre-Order Numbering of Vertices
Consider a Depth First Search traversal of the vertices (which happens to coincide with how we defined the Eulerian path). We want a pre-order numbering of vertices, i.e. for each node v, first v appears, then a pre-order numbering of its first child, then second child, and so on. Using the Eulerian tour technique we can solve the problem easily, by setting the weight of forward edges (from parent) to 1 and backwards edges (to parent) to 0. Then, we compute all prefix sums of these weights along the path. Each arc knows how many forward arcs were before it. We are left with setting the ordering of the root $pre(r) = 0$, and for all other nodes $v \neq r$ we set $pre(v)$ to be the prefix sum of the arc $\langle \text{parent}(v), v\rangle$.

Problem 3: Computing the depth of vertices
This is actually similar to computing the pre-order numbering and we obtain a result by setting the weight of forward edges to $w(\langle \text{parent}(v), v\rangle) = 1$ and the corresponding backward edges to $w(\langle v, \text{parent}(v)\rangle) = -1$.

6.4 Merging and Sorting

6.4.1 Merge Sort

In the *merge sort* algorithm we break the input of n items into two parts, sort recursively and then merge the resulting two sorted parts. The sorting of independent parts can of course be done in parallel, therefore we have to investigate the merge step.

Basic Merging
For an item $x \in A \cup B$ it suffices to know the number k of items in $A \cup B$ that are smaller than this item x. Then we would set $C[k+1] = x$ in the resulting array. This can be done by $n/2$ binary searches in parallel, having $O(\log n)$ depth and $O(n \log n)$ work in total. Something similar can be done for the elements in B (where we have to add the offset in the already sorted B to obtain the final position in C. Now it is easy to see that parallel merge sort with the basic merging has a depth of $O(\log^2 n)$ and total work of $O(n \log^2 n)$, which is not work efficient.

Improved Merging
Consider two sorted arrays $A[1..n], B[1..m]$. We know create a recursive algorithm where we choose evenly spaced-out "fenceposts" $A[\alpha_1], ..., A[\alpha_{\sqrt{n}}]$ where $\alpha_i = (i-1)\sqrt{n}$ and similarly for B and $\beta_i = (i-1)\sqrt{m}$. We perform all the $\sqrt{nm}$ comparisons in parallel on $\sqrt{nm} = O(n+m)$ processors. Then we use the same amount of processors such that we can determine the elements $B[\beta_j] \leq A[\alpha_i] \leq B[\beta_{j+1}]$ in $O(1)$ time in the CREW model. Note that we use $B[0] = -\infty$ and $B[m+1] = +\infty$, such that β_j is well-defined. Then, in parallel, we compare each $A[\alpha_i]$ to the $\sqrt{m}$ elements in its corresponding interval in B. We can therefore use the rank of each α_i to break the problem into $\sqrt{n}$ many subproblems of merging $\sqrt{n}$ elements of A with some number of elements of B. Note that we have m elements in B but it is possible that a subproblem is of size m. Regardless, we get the recursion $T(n) = O(1) + T(\sqrt{n})$ and thus after $O(\log \log n)$ recursions the problem boils down to finding one items placement in another array of length m, which can be done using m processors in O(1) time in the CREW model.
We can therefore merge two sorted array of length $n/2$ in $O(\log \log n)$ time using $O(n \log \log n)$ total computation. This leaves us with a merge sort algorithm with $O(\log n \log \log n)$ depth and $O(n \log n \log \log n)$ computation.
Note that we can get rid of the $\log \log n$ term by "pipelining" the merges, which was not covered in the lecture, and then gives us a work efficient algorithm.

6.4.2 Quick Sort

With quicksort, we pick a random index as pivot and break the array into two parts B and B' by comparing all elements with the index and then recurse on the subarrays.
In a *basic parallel approach*, we use the parallel prefix sum algorithm to determine the position of an element in the subarrays, split by pivot element $A[i]$. Suppose for each k, the bit $b(k) \in \{0, 1\}$ indicates whether $A[k] \leq A[i]$ or not (1 in the former case). The parallel prefix sum in A on content $b(k)$ can determine for each k such that $A[k] \leq A[i]$ the number $x(k)$ of indices $k' \leq k$ such that $A[k'] \leq A[i]$. Then, we should write $B[x(k)] = A[k]$. A similar operation can be used for B'. As we already know, these parallel prefix subproblems can be solved with $O \log n)$ depth and $O(n)$ work. We can show that the number of

recursion levels is also in $O(\log n)$ with high probability. For that matter, let us call a level of quick sort on an array of length n *success*, if the random index $i \in [n/4, 3n/4]$, and *fail* otherwise. Then the probability of success is at least $1/2$. Note that after $\log_{4/3} n$ successes, we end up with an array size of 1. Using the Chernoff bound on the independent levels (of all previous levels), we get that the probability of having at least $\log_{4/3} n$ successes in a branch of size at most $50 \log n$ is $1 - 1/n^4$. Therefore with a union bound over all branches, we end up with a probability of $1 - 1/n^3$. This gives us now $O(\log n)$ recursion levels, each implemented in $O(\log n)$ depth and $O(n)$ work, therefore an overall complexity of $O(\log^2 n)$ depth and $O(n \log n)$ work, which is work-efficient, but the depth is not optimal yet.

We know outline an *improved quick sort algorithm using multiple pivots*, that sends each element to the appropriate branch of the problem with a more significant reduction of the problem (i.e. more than the constant factor above):

(A) Pick $\sqrt{n}$ pivot indices in $\{1, ..., n\}$ at random (with replacement).

(B) Sort these $\sqrt{n}$ pivots in $O(\log n)$ depth and using $O(n)$ total work (performing all the pairwise comparisons simultaneously and then use parallel prefix on the results to know the number of pivots smaller than each pivot).

(C) Use these sorted pivot elements as *splitters*, and insert each element in the appropriate one of the $\sqrt{n} + 1$ branches in $O(\log n)$ depth and $O(n \log n)$ total work. Notice that two things need to be done: (1) Identifying for each element the subproblem in which this element should proceed (using n separate binary searches, one for each element, searching in between the sorted splitters). (2) Creating an array for the subproblem between each two splitters (can be done in $O(\log n)$ depth and $O(n \log n)$ work).

(D) Recurse on each subproblem between two consecutive splitters. In this recursion, once a subproblem size reaches $O(\log n)$ - where n is the size of the original sort problem - solve it in $O(\log n)$ time and $O(\log^2 n)$ work by brute force deterministic algorithm (compare element i with all the other elements j and compute the number of them that are smaller than element i).

We expect the branches to have size roughly $\sqrt{n}$. Thus intuitively we have the following recursion: $T(n) = O(\log n) + T(\sqrt{n}) = O(\log n + \log n/2 + ...) = O(\log n)$. Let us be more formally and classify the problem based on the size: we say the problem size is in *class* i if its size is in $[n^{(2/3)^{i+1}}, n^{(2/3)^i}]$. When we start with a subproblem in class i, then the expected size for the subproblem is the square root of it, i.e. at most $n^{(2/3)^i \cdot 1/2}$. We say the branch of recursion *fails* if the size of the subproblem is $n^{(2/3)^i \cdot 2/3} >> n^{(2/3)^i \cdot 1/2}$, and therefore the problem stays in class i.

Lemma 6.5. *The probability of failing is at most* $\exp(-\Theta(n^{(2/3)^i \cdot 1/6}))$

This is the probability of failing in class i.

Lemma 6.6. *Define $d_i = (1.49)^i$. So long as $n^{(2/3)^i} = \Omega(\log n)$, the probability of having at least d_i consecutive failures at class i before moving to class $i+1$ is at most $1/n^3$.*

Lemma 6.7. *With probability $1 - 1/n^2$, the depth of each branch of recursion is $O(\log n)$.*

Lemma 6.8. *With probability $1 - 1/n^2$, the total work over all branches if $O(n \log n)$.*

6.5 Connected Components

We now want to discuss an algorithm that for a given graph $G = (V, E)$ identifies its connected components. We assume that the graph is given as a set of adjacency lists and the output will be an identifier $D(v)$ for each vertex v. Note that isolated vertices can be identified in $O(1)$ depth and $O(n)$ work. Moreover, we can remove them from the set V by renumbering V and E in $O(\log n)$ depth and $O(m)$ work, by parallel prefix. Hence we will focus on graphs without isolated vertices for the rest of the section and assume the ARBITRARY CRCW model.

6.5.1 Basic Deterministic Connected Components Algorithm

In this algorithm we have $\log n$ iterations, where at the beginning of each iteration i, all nodes are in (vertex-disjoint) fragments $F_1, ..., F_{n_i}$, e.g. in the first iteration we have $n_1 = n$ fragments. In each iteration we merge more and more fragments while maintaining the property that all nodes in a fragment belong to the same connected component.
We maintain each connected component F_j as a star with one root and a number of leaf nodes, which makes it easy to check wheter two nodes are in the same fragment. Each node v has a pointer $D(v)$ equal to the root node of its star fragment, where the root node points to itself. In the beginning of the algorithm each node is the root of its fragment.
For each fragment F_j rooted at node r_j we want to identify the minimum root node r_k such that there is an edge $e = (u, v)$ from some node v with $D(v) = r_j$ and $D(u) = r_k$. In such a case, we say fragment F_j *proposes* to merge with F_k and remember this as $p(r_j) = r_k$. If for a fragment there is no such edge, we set $p(r_j) = r_j$. We can compute all of these minimums simultaneously, one per fragment in $O(\log n)$ depth and $O(m)$ work (left as an exercise).
The set of proposed merge edges, declared by the pointers $p(r_j)$ for each root r_j determine a *pseudo-forest* P among the nodes in the graph. In a pseudo-forest each connected component is a *pseudo-tree*, which is simply a tree plus one extra edge that creates one cycle. In our case we have even more structure, i.e. the arcs are oriented and each node has an out-degree of exactly one. This implies that each connected component is a (directed) tree plus one cycle creating arc. Since we propose to the minimum neighboring fragment, we obtain the following lemma:

Lemma 6.9. *Each cycle in P is either a self-loop or it contains exactly two arcs.*

We now have to transform each pseudo-tree in the forest into a star shape. For each root node r, remove its self-loop in its fragment and instead set $D(r) = p(r)$, i.e. the root of the proposed fragment with which the fragment of r wants to merge. Then do $\log n$ repetitions of pointer jumping, such that after these the pointer $D(v)$ of each node is to one of the at most two nodes in the cycle of P containing v. In a final step, set $D(v)$ to the minimum of the two root candidates, such that all nodes of the same component P now point to the same root node. This transformation to star shapes is basically $O(\log n)$ iterations of pointer jumping plus one final step of updating and thus uses $O(\log n)$ depth and $O(n \log n)$ total work.
We need $\log n$ iterations to reach a setting where each fragment is exactly one of the connected components of the graph, because the number of fragments shrinks by a factor of 2 as long as nodes are in two or more fragments. Each iteration uses $O(\log n)$ depth to find the proposal edges and also to shrink pseudo-trees to stars and $O(m \log n)$ work, therefore for all iterations we have $O(\log^2 n)$ depth and $O(m \log^2 n)$ work in total.

6.5.2 Randomized Connected Components Algorithm

In this approach we want to overcome the time-consuming fragment shrinkage steps. Note that the $O(\log n)$ repetitions of pointer jumping before were necessary because of the possibility that the edges we choose for a merge in one iteration can form long chains. We now want to overcome this with a randomized approach.
The first change is that we do not care about the minimum neighbor anymore, but do it with any neighboring fragment that is connected via an edge. This can be done in $O(1)$ depth and $O(m)$ work in the ARBITRARY CRCW model. Additionally, we toss a HEAD/TAIL coin and accept a proposed merge edge only if it points from a TAIL fragment to a HEAD fragment. The result of this choice is that now only HEAD fragments can have others merge it, and the depth of the resulting merge is $O(1)$, which means that $O(1)$ iterations of pointer jumping suffice to turn each new fragment into a star shape. The downside is that we did not use all of the proposed merge edges. But for the number of iterations of merging we still have:

Lemma 6.10. *In each iteration i where $n_i \geq 2$, the expected number n_{i+1} of new fragments is at most $7/8$ of the number n_i of the old fragments.*

Lemma 6.11. *After $L = 20 \log n$ iterations, the number of fragments $n_L = 1$, with probability at least $1 - 1/n^2$.*

6.6 (Bipartite) Perfect Matching

We will now discuss parallel algorithms for the problem of computing a perfect matching on bipartite graphs. First we generalize the already seen class $NC(k)$

to randomized algorithms. We say a decision problem is in $RNC(k)$ if there is a randomized circuit with depth $O(log^k n)$ and size $poly(n)$ which outputs the correct solution with probability at least 2/3. Again, we define $RNC = \cup_k RNC(k)$, and clearly we have $NC \subset RNC$. More generally, we can also consider search problems. Again, our goal is to find parallel algorithms with depth $poly(\log n)$ depth and $poly(n)$ work. This is what we will consider as an *efficient algorithm.*

6.6.1 Randomized Parallel Algorithm

We first state a remark for the best known bounds for matrix operations, which we will use as a black box for this chapter.

There is a randomized parallel algorithm that can compute the determinant, inverse, and adjoint of any $n \times n$ matrix with k-bit integer entries using $O(log^2 n)$ depth and $\tilde{O}(n^{3.5}k)$ computation. Recall that the adjoint of a matrix A is an $n \times n$ matrix whose entry (i, j) is $(-1)^{i+j} det(A_{ij})$, where A_{ij} denotes the submatrix of A resulted from removing its j^{th} row and i^{th} column.

Now let $G = (V, U, E)$ be the graph with vertex partitions U and V. A natural idea for computing a perfect matching would be to remove each edge e, and see if the resulting graph still has a perfect matching. However, this does not work in parallel in the general case, since we could remove two edges in parallel, but the resulting graph without both edges may not contain a perfect matching anymore. The only scenario in which this idea would work is if the graph has a unique perfect matching. We therefore want to mimic this scenario by using weights on the edges and searching for a minimum-weight perfect matching. The following lemma helps us obtain such an algorithm and has an interesting, yet simple proof.

Lemma 6.12. *(Isolation Lemma) Let (E, F) consist of a finite set E of elements, with $m = |E|$, and a family F of subsets of E, i.e. $F = \{S_1, S_2, ..., S_N\}$, where $S_i \subseteq E, \forall 1 \leq i \leq N$. Assign to each element $e \in E$ a random weight $w(e)$ uniformly and independently chosen from $\{1, 2, ...2m\}$. Define the weight of any set $S_i \in F$ to be $\sum_{e \in S_i} w(e)$. Then, regardless of how large N is, with probability at least $1/2$, there is a unique minimum weight set in F.*

Given the isolation lemma and the intuition discussed above the algorithm is clear: set random weights, and we will have a unique minimum perfect matching in the graph with probability at least 1/2. We can find it as follows:

- Let A be the $n \times n$ matrix where $a_{ij} = 2^{w(e_{ij})}$, and $a_{ij} = 0$ if there is no such edge.
- Compute $det(A)$ using the parallel algorithm with the best known complexity (mentioned before) and let w be the highest power of 2 that divides $det(A)$.
- Compute $adj(A)$ using the same parallel algorithm where the (i, j) entry

will be equal to $(-1)^{i+j} det(A_{ij})$. Here, A_{ij} denotes the submatrix of A resulted from removing the j-th row and the i-th column of A.

- For each edge $e_{ij} = (u_i, v_j)$, do in parallel:
  Compute $\frac{det(A_{ij})2^{w(e_{ij})}}{2^w}$ for all edges. If it is odd, include e_{ij} in the output matching.

To amplify the success probability of the isolation lemma, we can run it several times in parallel and output a perfect matching if any of the runs found one. Running it $10 \log n$ times ensures that we find a perfect matching with probability at least $1 - (1/2)^{10 \log n} = 1 - 1/n^{10}$ The following two lemmata explain that if the random weights are chosen such that the minimum-weight perfect matching is unique, then the above procedure will output it.

Lemma 6.13. *Suppose that $G = (V, U, E)$ has a unique minimum weight perfect matching M and its weight is w. Then, the highest power of 2 that divides $det(A)$ is 2^w.*

The proof of above lemma makes use of the fact that each perfect matching in G corresponds to one permutation $\sigma \in S_n$, and how this particular permutation is correlated to the determinant of A. Define $value(\sigma) = \prod_{i=1}^{n} a_{i\sigma(i)}$. Then the determinant can be defined as the signed sum over all values of all permutations. For any other permutation than the minimum weight perfect matching permutation the value is either 0 or a strictly higher power of 2 (this we saw in section 5.3), therefore $det(A)$ being a summation of them is divisible by 2^w.

Lemma 6.14. *Suppose that $G = (V, U, E)$ has a unique minimum weight perfect matching M and its weight is w. Then, edge $e_{ij} = (u_i, v_j)$ is in M if and only if $\frac{det(A_{ij})2^{w(e_{ij})}}{2^w}$ is odd.*

6.6.2 Deterministic Parallel Algorithm

Omitted.