Algorithms, Probability and Computing HS2010 [1]

Stefan Heule

February 8, 2011

Contents

1 Random(ized) Search Trees

1.1 Definition

Definition 1.1 (Search tree)**.** *A* search tree B_S *on a set* S *of* n, $n \in \mathbb{N}_0$ *keys, is an ordered binary tree with a distinguished node, called the* root*. The set* S *must be a totally ordered set.*

Every node in the tree has at most two children, and it is called a leaf *if it has no children at all. All nodes in the left subtree of* B_S *with root* x *are from the set*

$$S^{<x} := \{a \in S \mid a < x\},$$

and similarly, the keys in the right subtree are from

$$S^{>x} := \{a \in S \mid a > x\}.$$

In such a binary search tree, we define the *depth* of a node v as

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

The *height* of a tree is then the maximum depth among its nodes.

Definition 1.2 (Random search tree)**.** *A random search tree on the set* S *as above, is either the empty tree* λ *if* $S = \emptyset$*, or has root* $x \in_{u.a.r.} S$*, and random search trees on sets* $S^{<x}$ *and* $S^{>x}$ *as children. That is, every node* $x \in S$ *is equally likely to appear as the root.*

Lemma 1.1. *Let* $S \subset \mathbb{R}$ *be finite. Given a tree in* $\mathcal{B}_S$*, the set of all binary search trees on* S*, we let* $w(v)$*,* v *a node, denote the number of nodes in the subtree rooted at* v*.*

The probability of the tree according to the above definition is the product of one over $w(v)$ *over all nodes:*

$$\prod_v \frac{1}{w(v)}$$

One interesting and useful property of this probability distribution is that it is the same as the one obtained by inserting the keys in S into an initially empty tree in random order drawn u.a.r. from all permutations.

1.2 Overall and Average Depth of Keys

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

$$\mathrm{rk}\,(x) = \mathrm{rk}_S(x) := 1 + |\{y \in S \mid y < x\}|$$

In particular, if $x \in S$, then x is the $\mathrm{rk}\,(x)$-smallest element in S. Furthermore, we write $\mathrm{D}_n^{(i)}$ for the random variable for the depth of they key of rank i in a random search tree of n keys.

Lemma 1.2 (Depth of the Smallest Key)**.** *The expected depth of the smallest key in a random search tree is given by*

$$d_n = \mathbb{E}\,[D_n] = \mathbb{E}\left[D_n^{(1)}\right] = H_n$$

Similarly, we can look at the overall depth in a random search tree, which is given as $\sum_{i=1}^{n} \mathrm{D}_n^{(i)}$.

Theorem 1.1 (Overall Depth of Keys)**.** *Let* $n \in \mathbb{N}_0$*. The expected overall depth of a random search tree for* n *keys is* $2(n+1)H_n - 4n = 2n \ln n + \mathcal{O}(n)$*.*

1.3 Expected Height

Theorem 1.2 (Expected Height of Random Search Trees)**.** *The expected height of a random search tree for* n *keys is upper bounded by* $c \ln n$*, where* $c = 4.311...$ *is the unique value greater than 2 which satisfies* $(\frac{2e}{c})^c = e$*.*

1.4 Expected Depth of Individual Keys

We want to compute $d_{i,n} = \mathbb{E}\left[\mathrm{D}_n^{(i)}\right]$, the expected depth of the node holding the key of rank i in a random search tree for n keys. Instead of the usual approach of computing a recurrence, we use indicator variables and linearity of expectation. In particular, for $i, j \in \{1 \dots n\}$, we introduce the indicator variable

$$A_i^j := [\text{node } j \text{ is ancestor of node } i]$$

(in a random search tree for n keys), i.e.

$$A_i^j = \begin{cases} 1, & \text{if node } j \text{ is ancestor of node } i, \text{ and} \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, we have $\mathrm{D}_n^{(i)} = \sum_{j=1, j=i}^{n} A_i^j$, and by linearity of expectation,

Lemma 1.3. *Let $i, j \in \mathbb{N}$. In a random search tree for $n \geq \max\{i, j\}$ keys*

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

Theorem 1.3. *$i, n \in \mathbb{N}$, $i \leq n$. Then $\mathbb{E}\left[D_n^{(i)}\right] = H_i + H_{n-i+1} - 2 \leq 2 \ln n$.*

1.5 Quicksort

The choice of the pivot is crucial for the performance, since bad choices can lead to as much as $\Omega(n^2)$ comparisons. Here, we consider randomized quicksort, where the pivot is chosen uniformly at random.

Algorithm 1 quicksort(S)

Require: $S \subseteq \mathbb{R}$, finite.
Ensure: returns S as a sequence sorted in increasing order.

```
1: if S = ∅ then
2:    return ⟨⟩
3: else
4:    x ←_{u.a.r.} S
5:    split S into S^{<x}, {x}, S^{>x}
6:    return quicksort(S^{<x}) ∘ ⟨x⟩ ∘ quicksort(S^{>x})
7: end if
```

Theorem 1.4. *$n \in \mathbb{N}_0$. The expected number of comparisons performed by the procedure quicksort() for sorting a set of n numbers is $2(n+1)H_n - 4n$.*

There is an intuitive analogy between random search trees and the structure of the procedure quicksort(). In fact, during an execution of quicksort() we can build a binary search tree on the side, always making the pivot the root of a new subtree. In this way, every computation of quicksort(S) maps to a search tree B in $\mathcal{B}_S$, and it is to show that the resulting distribution is the same as for random search trees.

1.6 Quickselect

Given a set S of keys, we want to know the k-smallest element in this set. Obviously, we could sort the set, and return the kth entry in the sorted sequence, but we can do better than that.

Theorem 1.5. *$k, n \in \mathbb{N}_0, 1 \leq k \leq n$. 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 $4n$.*

1.7 Randomized Search Trees

Definition 1.3 (Treap)**.** *"Treap" is a coined word that stands for a symbiosis of a binary search* tree *and a* heap*. It is defined for sets $Q \subseteq \mathbb{R} \times \mathbb{R}$. The first component of an item x is its* key*, $\mathrm{key}(x)$, and the second component is its* priority*, $\mathrm{prio}(x)$. Suppose 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 (i.e. if x is parent of y, then $\mathrm{prio}(x) \leq \mathrm{prio}(y)$).*

Algorithm 2 quickselect(k,S)

Require: $S \subseteq \mathbb{R}$, finite, $k \in \mathbb{Z}$, $1 \leq k \leq |S|$.
Ensure: returns the element of rank k in S.

```
x ←_{u.a.r.} S
split S into S^{<x}, {x}, S^{>x}
l ← |S^{<x}| + 1    (i.e. l = rk(x))
if k < l then
    return quickselect(k, S^{<x})
else if k = l then
    return x
else
    return quickselect(k − 1, S^{>x})
end if
```

Note that if the priorities are chosen uniformly at random from the interval $[0, 1)$, 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 hence assume all the wonderful properties like expected and high probability logarithmic height of the tree.

1.7.1 Insertions and Rotations

To insert a new item x in a treap, we first proceed just as with a normal search tree, and insert x as a leaf. Then, we have to re-establish the heap property, and we achieve this by successive rotations on the parent y of x, if it has a higher priority.

The runtime for the insertion is proportional to the depth of the new item before any rotations, plus the number of rotations. As both are bounded by the height of the tree, this is expected $\mathcal{O}(\log n)$.

In order to analyze the number of necessary rotations in more detail, we define the *left (right) spine* of a subtree rooted at node v as the sequence of nodes on the path from v to the smallest (largest, respectively) key in the subtree. Now we 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 by exactly 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 1.4. *$n \in \mathbb{N}, j \in \{1 \dots n\}$. 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 $1 - \frac{1}{n-j+1}$.*

Theorem 1.6. *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 $\mathcal{O}(\log n)$, n the number of keys currently stored. The expected number of rotations necessary for an insertion or deletion is always less than 2.*

1.7.2 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.

For a given pivot value s, a *split* of a treap for items Q generates two treaps for items $Q^{<s}$ and items $Q^{>s}$ (assuming $s \notin Q$). An easy implementation inserts an item with key s and priority $-\infty$. This item ends up at the root, with all elements from $Q^{<s}$ in its left subtree, and the elements from $Q^{>s}$ in the right subtree. The time is again bounded by the height, the expected number of rotations is not constant, though.

The *join* operation takes two treaps with one holding keys all of which are smaller than that of the other one. Thus, this operation can be seen as an inverse split, and its implementation is rather similar. We introduce a node with an in-between key s and priority $-\infty$, that we use as a root for the two treaps. Then, this item is deleted from the tree.

1.7.3 Random Real Numbers?

While real numbers as priorities is very handy for the analysis, in practice we would like to avoid them. However, since we only need to compare priorities, we can simulate the necessary functionality by storing the priorities as a sequence of 0's and 1's. One can think of this as the priorities binary representation, and we only store

as many bits as we currently need. If we encounter two identical priorities, and thus cannot decide which is smaller, we generate for both an additional random bit and try to compare again.

2 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.

2.1 Point/Line Relative to a Convex Polygon

Definition 2.1 (Convex Hull)**.** *The convex hull* $\mathrm{conv}(P)$ *of a finite set* P *of points in the plane is a bounded convex set. Unless* $\mathrm{conv}(P)$ *of the empty set, a single point or a line segment, it is bounded by a* convex polygon*, a closed simple piecewise linear curve that separates the interior of* $\mathrm{conv}(P)$ *from the exterior. This polygon can be finitely described by a sequence of its vertices, in counter-clockwise order, say.*

2.1.1 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 to decide about the answer to the query. Summing up, the structure needs $\mathcal{O}(n)$ space and $\mathcal{O}(\log n)$ query time. If the vertices of C are provided in sorted order along C, building the structure takes linear time as well.

2.1.2 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 $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ 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 to vertices v and v'. Now l hits the polygon iff v and v' lie on different sides of l.

Again, this results in an optimal structure with linear storage and preprocessing time, and with logarithmic query time.

2.2 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, and are back in known territory.

2.2.1 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 $\mathcal{O}(k + \log n)$ for reporting the k points below a 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 point in p 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.

Definition 2.2 (Onion)**.** *The* onion *of a general set* P *is the sequence*

$$(R_0, V_0), (R_1, V_1), \ldots, (R_T, V_t)$$

If P is empty, the sequence is empty. Otherwise, let $R_0 := \mathrm{conv}(P)$ and let V_0 be the vertex set of R_0. $(R_1, V_1), \ldots, (R_T, V_t)$ 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 completly above or on a line l, then all $R_j, j \in \{i \ldots t\}$ are above or on the line l, and so are all $V_j, j \in \{i \ldots t\}$.*
- *$V_0, V_1, \ldots, V_t$ is a partition of P into non-empty sets.*

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 $\mathcal{O}(k_i + \log n_i)$. Having established the structures for all $i \in \{0, \ldots t\}$, we start a query with a line l in the structure for V_0, report all points in V_0 below l, then proceed to the structure for V_1, etc. until either (1) we have reached V_j that is completely above or on l, or (2) we get to the situation that we have exhausted all sets, i.e. reached V_t. In case (1), this takes time

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

We use $j \leq k$ here, since each k_i is at least 1. In the second case (2) we take time

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

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

2.2.2 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 the previous iterations. For a better solution we first enhance the sets S_i with extra information, resulting in sets $\overline{S}_i$. The "last" set S_t is left untouched, so $\overline{S}_t = S_t$. Otherwise for $i = t-1, t-2, \ldots, 0$, to obtain $\overline{S}_i$ we add to S_i every other number from $\overline{S}_{i+1}$ in its sorted order starting with the second number. Hence, $|\overline{S}_i| \leq |S_i| + \frac{|\overline{S}_{i+1}|}{2}$.

Each of this $\overline{S}_i$'s are stored in a linear array. For $0 \leq i \leq t-1$, each interval I of $\overline{S}_i$ is either completely contained in an interval I' of $\overline{S}_{i+1}$, or it contains exactly one number $a \in \overline{S}_{i+1}$ in its interior. In the former case, we add a pointer from I to I', in the latter a pointer from I to a; a comparison of the query number β with a determines the interval of $\overline{S}_{i+1}$ containing β. Hence, in this enhanced structure, we can locate β in constant time in $\overline{S}_{i+1}$, given we have it already located in $\overline{S}_i$. This brings the query time down to optimal $\mathcal{O}(k + \log n)$.

Lemma 2.1. *The size of this structure is at most $2n$, i.e. $\sum_{i=0}^{t} \overline{n}_i \leq 2n$.*

Theorem 2.1. *A set P of n points in the plane can be preprocessed in time $\mathcal{O}\left(n^2 \log n\right)$ and linear storage so that the set of points below a query line l can be reported in $\mathcal{O}(k + \log n)$, k the number of points below the query line.*

Note that the method of cascading some of the elements of one list to another list is called *fractional cascading*, which has numerous applications.

2.2.3 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". Duality exploits this fact by mapping points to lines and lines to points.

Let * be the mapping that maps points to lines and lines to points by

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

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

Observation 2.1. *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^*.*

2.2.4 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, $i \in \{0 \dots n\}$, and what is left is to locate q in these regions.

For $k \in \{1 \dots n\}$, the *k-level*, Λ_k, of a set L of n 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. It is easy to see that every vertical line intersects the k-level in exactly one point.

Given a point q and a bi-finite x-monotone curve C, we write $C \succ q$ ($q \succ C$) if q lies below (above, respectively) the curve C.

Observation 2.2. *For $k \in \{1 \dots n\}$, the k-level Λ_k is an x-monotone, piecewise linear, 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 liens in L above a point q is $\min\{k \mid q \succeq \Lambda_k\} - 1$, 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 Λ_k's, which can be done by binary search with $\mathcal{O}(\log n)$ comparisons with these curves. Remains the issue how we efficiently compare q to such curves.

2.2.5 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* (dimension 0), *edges* (dimension 1) and *cells* or *faces* (dimension 2). The vertices, edges and the faces, with their incident structure they are called the *line arrangement* of L. Two faces are incident if one is contained in the relative closure of the other.

Lemma 2.2. *$n \in \mathbb{N}_0$. 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, 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 $\mathcal{O}(m_k)$ space, and we can decide for a given point q whether it lies below the level in $\mathcal{O}(\log m_k) = \mathcal{O}(\log n)$ 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 and an edge e where q's x-coordinate is between the 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 $\mathcal{O}(n^2)$ space, so that the number of lines above a query point can be determined in time $\mathcal{O}((\log n)^2)$.

To improve the runtime, we again call for fractional cascading. Let us store (at least conceptually) the levels of an arrangement in a balanced tree for locating q among them, with leaves holding the number of lines above a point q whose search ends there. In every inner node v of the tree, we have to find x_q the interval I among the set S_v of x-coordinates of the respective level with $x_q \in I$, and compare q with the line holding the edge in the span of the interval. Depending on the outcome of the comparison, we proceed to the left or to the right child, and - unless this is a leaf - have to locate x_q again, etc. At present, we do the location anew from scratch again and again.

Instead, we now enhance the sets S_v by extra values inherited from its descendants in the tree, resulting again in sets $\overline{S}_v$. If a node v has no child which is an inner node, we leave the set unchanged, i.e. $\overline{S}_v = S_v$. Otherwise we add to S_v every other value from each of the sets $\overline{S}_u$, u non-leaf child of v.

After this bottom-up generation of the sets in the nodes of the tree, we proceed top-down and add for each node v pointers from each interval I determined by $\overline{S}_v$, one pointer for each non-leaf child u of v. This pointer either directly points at an interval determined by $\overline{S}_u$ contained in I, or to the unique number of $\overline{S}_u$ that is contained in the interior of I. In this way, we can again proceed in constant time from an interval in a node to the interval in its child. Only in the root of the tree we will have to spend logarithmic time for an initial location of x_q.

Theorem 2.2. *A set P of n points in the plane can be preprocessed with $\mathcal{O}(n^2)$ storage so that the number of points below a non-vertical line can be computed in $\mathcal{O}(\log n)$.*

3 Maximum Flow

3.1 Definitions and Statement of the Main Theorem

Definition 3.1 (Network)**.** *A* network *is a quadruple (G, s, t, c), where $G = (V, E)$ is a directed graph, s and t are two distinct vertices of G (we call them the* source *and the* sink *or* target*), and the* capacity *$c : E \to \mathbb{R}_0^+$ is a function assigning non-negative real values to edges.*

We introduce another useful notion: Let $f : E \to \mathbb{R}$ be a real function on the edges of G, so far arbitrary. The *net outflow* of a vertex $v \in V$ is defined by

$$\mathrm{NetOut}_f(v) = \sum_{(v,x)\in E} f(v,x) - \sum_{(x,v)\in E} f(x,v)$$

Definition 3.2 (Flow)**.** *A* flow *in a network (G, s, t, c) is any function $f : E \to \mathbb{R}_0^+$ that satisfies*

1. *$0 \le f(e) \le c(e)$ for every edge $e \in E$, and*
2. *$\mathrm{NetOut}_f(v) = 0$ for every vertex $v \in V$ different from s and t*

The value *of the flow f is*

$$\mathrm{val}(f) = \mathrm{NetOut}_f(s)$$

Proposition 3.1 (Existence of Maximum Flow)**.** *Every network has a* maximum flow*, i.e. a flow f with $\mathrm{val}(f) \ge \mathrm{val}(f')$ for every flow f' in the network.*

Again, we introduce some more notation: For disjoint sets $A, B \subseteq V$ we write $E(A, B) = \{(u, v) \in E \mid u \in A, v \in B\}$ for the set of edges going from A to B. Similarly, we write $c(A, B) = \sum_{e\in E(A,B)} c(e)$ and $f(A, B) = \sum_{e\in E(A,B)} f(e)$.

Definition 3.3 (s-t-cut)**.** *An s-t-*cut *in a network (G, s, t, c) is any subset $S \subset V$ such that $s \in S$ and $t \notin S$. The* capacity *of an s-t-cut is the number*

$$\mathrm{cap}(S) = c(S, V \setminus S)$$

Since there is only a finite number of s-t-cuts, there is clearly a minimum s-t-cut.

Theorem 3.1 (Maxflow-Mincut Theorem)**.** *For every network the maximum value of a flow equals the minimum capacity of an s-t-cut:*

$$\max_{f \text{ a flow}} \mathrm{val}(f) = \min_{S \text{ an } s\text{-}t\text{-cut}} \mathrm{cap}(S)$$

3.2 Proof of Main Theorem

Lemma 3.1. *For every flow f and every s-t-cut S we have*

$$\mathrm{val}(f) = f(S, V \setminus S) - f(V \setminus S, S)$$

Thus the net flow over the boundary of S, written on the right-hand side, is the same for every S.

In particular, setting $S = V \setminus \{t\}$, we see that the value of a flow which was defined as the net outflow from the source, also equals the net inflow in the sink, $\mathrm{val}(f) = -\mathrm{NetOut}_f(t)$.

Lemma 3.2. *The value of every flow is bounded above by the capacity of any s-t-cut:* $\mathrm{val}(f) \le \mathrm{cap}(S)$.

Definition 3.4 (Residual Network)**.** *Let (G, s, t, c) be a network that contains no pair of opposite edges and let f be a flow in it. The* residual network *(G_f, s, t, r) is defined as follows. The vertex set of G_f is the same as the vertex set of G, and the edge set consists of directed edges specified by the following rules:*

- *If e is an edge of G with $f(e) < c(e)$, then e is an edge of G_f.*
- *If $e = (u, v)$ is an edge of G with $f(e) > 0$, then the edge $e^{opp} = (v, u)$ is an edge of G_f.*

The residual capacity *of an edge e of G_f is*

$$r(e) = \begin{cases} c(e) - f(e), & \text{if } e \in E(G) \text{ (and so } f(e) < c(e)) \\ f(e^{opp}), & \text{if } e^{opp} \in E(G) \text{ (and so } e^{opp} > 0). \end{cases}$$

Note that $r(e) + r(e^{\text{opp}}) = c(e)$ for all edges e.

Proposition 3.2. *A flow f in a network (G, s, t, c) is maximum if and only if there is no directed path from s to t in the residual network G_f. For every maximum flow f there is a cut S such that* $\text{val}(f) = \text{cap}(S)$.

3.3 The Ford-Fulkerson Algorithm

Let us call a directed path in G_f from s to t an *augmenting path*. Such a path can be used to increase the flow in a network, which is exploited by the following algorithm.

Algorithm 3 Ford-Fulkerson Algorithm(G,s,t,c)

```
f ← the zero flow
while an augmenting path exists do
    choose an augmenting path P
    modify f along P by the minimum residual capacity on P
end while
return the current f as a maximum flow.
```

Observation 3.1. *Here are two theoretical consequences of the Ford-Fulkerson algorithm.*

1. *If the capacities of all edges are rational, the Ford-Fulkerson algorithm is finite. For networks with rational capacities this proves the existence of a maximum flow.*
2. *Every network with integral capacities has an integral maximum flow (i.e. a maximum flow f where $f(e) \in \mathbb{N}_0$ for all edges e).*

Unfortunately there are networks with irrational capacities where the Ford-Fulkerson algorithm need not end, and it even need not converge to a correct solution. Even for integral capacities the algorithm may be very slow, if the augmenting paths are chosen badly. The algorithm runs in $\mathcal{O}(mnU)$ time, where U is the largest edge capacity.

3.4 Capacity Scaling

The first idea for improving the worst-case efficiency of the Ford-Fulkerson algorithm is to use augmenting paths that increase the flow value as much as possible. For given parameter Δ, let $G_f(\Delta)$ denote a subgraph of the residual network G_f consisting only of edges with residual capacity at least Δ.

Algorithm 4 CapacityScaling(G,s,t,c)

```
Δ ← the smallest power of 2 exceeding U
f ← the zero flow
repeat
    Δ ← Δ/2
    while there is a directed s-t-path P in G_f(Δ) do
        augment f along P
        update G_f(Δ) accordingly
    end while
until an augmenting path exists
return the current f as a maximum flow.
```

Note that we do not insist on the really largest possible increase, which would complicate the computation and would not gain us much, but we are satisfied with a "reasonably large" one.

The algorithm correctly finds a maximum flow, since the flow remains integral at all times, and so $G_f(1) = G_f$. Obviously, the algorithm goes through $\mathcal{O}(\log U)$ values for Δ.

Lemma 3.3. *If there is no directed s-t-path in $G_f(\Delta)$, then* $\mathrm{val}(f)$ *differs from the value of a maximum flow by at most $m\Delta$.*

For each value of Δ, the algorithm makes no more than $2m$ augmentations, thus the total running time is $\mathcal{O}\left(m^2 \log U\right)$.

3.5 Shortest augmenting paths

Another simple idea for improving the efficiency of Ford-Fulkerson is this: Always choose an augmenting path with the smallest possible number of edges. Let us introduce some convenient notation:

- Let $l(G_f)$ denote the length of the shortest directed s-t-path in G_f.
- Let $\mathrm{SPE}(G_f)$ denote the set of all edges of G_f that lie on at least one shortest directed s-t-path in G_f.

Lemma 3.4. *Let H be a directed graph with two distinghished vertices s and t, and let H^+ be the directed graph with $V(H^+) = V(H)$ and $E(H^+) = E(H) \cup \mathrm{SPE}(H)^{opp}$. Then*

- $l(H^+) = l(H)$,
- $\mathrm{SPE}(H^+) = \mathrm{SPE}(H)$.

Corollary 3.1. *Let us suppose that a flow g was obtained from a flow f by a single augmentation along a shortest path in the residual network G_f. Then*

- *either $l(G_g) > l(G_f)$,*
- *or $l(G_g) = l(G_f)$, $\mathrm{SPE}(G_g) \supseteq \mathrm{SPE}(G_f)$, and the inclusion is proper.*

The corollary shows that the algorithm works in phases. First it augments with paths of some length k_1, then with paths of length $k_2 > k_1$, and so on.

Altogether there are at most n phases, and during each phase, the length of the augmenting paths remains fixed. Each phase has at most $2m$ augmentations, since initially $\mathrm{SPE}(G_f)$ has no more than $2m$ edges, and each augmentation decreases the number of edges by at least one. Thus:

Theorem 3.2. *The variant of the Ford-Fulkerson algorithm with shortest augmenting paths has at most n phases with $\mathcal{O}(m)$ augmentations per phase. Thus, the total running time is at most $\mathcal{O}\left(m^2 n\right)$, independent of capacities.*

3.6 Bipartite Matchings and Hall's Theorem

Definition 3.5 (Matching in a Graph)**.** *A* matching *in a graph G is a set of edges $F \subseteq E(G)$ such that every vertex $v \in V(G)$ belongs to at most one edge of F.*

A maximum matching in a bipartite graph can be computed by any maximum flow algorithm, using a simple reduction. We add two new vertices s and t to the graph, and connect s to all vertices in A, and similarly, all vertices from B with t. Furthermore, we direct the edged between A and B to point to vertices in B. All edge capacities are set to 1.

For instance, the Ford-Fulkerson algorithm can be used to find a maximum matching in $\mathcal{O}(mn)$ time, since all capacities are 1.

4 Minimum Cut

4.1 Definition and Overview of Algorithms

In this chapter we deal with undirected graphs: no arrows, no loops, edges are unordered pairs of vertices. We permit multiple edges between the same pair of nodes, though.

Definition 4.1 (Minimum Cut)**.** *A* cut *in a graph G is a subset $C \subseteq E(G)$ of edges such that the graph $(V(G), E(G) \setminus C)$ is disconnected. The* minimum cut *is a cut with the minimum possible number of edges, and we let $\mu(G)$ denote the size of a minimum cut in G.*

4.1.1 Flow-Based Algorithms

We replace every undirected edge $\{u, v\}$ with multiplicity w by two edges (u, v) and (v, u), both with capacity w. Now for any two nodes s and t, we can compute a maximum flow (using any algorithm described previously), and thereby getting a minimum s-t-cut (take the vertices reachable from s in G_f).

In order to find a general minimum cut, we can keep s fixed, and choose all $n - 1$ possibilities for t, thus computing a minimum cut.

4.2 Preliminaries

Let G be a multigraph and let $e = \{u, v\}$ be an edge of G. The *contraction* of e means that we "glue" u and v together into a single vertex, and then we remove loops that may have arisen in this way (while multiple edges are retained). The resulting graph is denoted by G/e. Note that this decreases the number of vertices by exactly one.

Observation 4.1. *Let G be a multigraph and e an edge of G. Then $\mu(G/e) \geq \mu(G)$. Moreover, if there exists a minimum cut C of G such that $e \notin C$, then $\mu(G/e) = \mu(G)$.*

4.3 Random Contractions

Let us consider a first very basic probabilistic algorithm, called *basic min-cut*.

Algorithm 5 BasicMinCut(G)

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

Lemma 4.1. *Let G be a multigraph with n vertices. Then the probability of $\mu(G) = \mu(G/e)$ for a randomly chosen $e \in E(G)$ is at least $1 - \frac{2}{n}$.*

For a multigraph G, let $P_0(G)$ denote the probability that the algorithm basic min-cut succeeds. Let $P_0(n)$ denote the infimum of $P_0(G)$ for all multigraphs on n vertices.

We have $P_0(2) = 1$, and by recursion, we get

$$P_0(n) \geq \left(1 - \frac{2}{n}\right) \cdot P_0(n-1) \geq \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}$$

Even though this looks hopeless, we can repeat the procedure often to get a reasonably small error probability.

4.4 Bootstrapping

The key observation is that toward the end, the success probability shrinks considerably. Therefore, we randomly contract until we only have t vertices left, and then use some slower, but more reliable algorithm. As a first idea, we can use N_1 times a call to basic min-cut, resulting in the following improved algorithm.

Algorithm 6 BetterMinCut(G)

```
set parameters t and N_1 suitably
H ← RandomContract(G,t)
return minimum of N_1 runs of basic min-cut(H).
```

This idea can be used to over and over, resulting in the following algorithm, called min-cut.

Lemma 4.2. *The running time $T(n)$ of MinCut on an n-vertex graph is bounded by $\mathcal{O}\left(n^2 \log n\right)$.*

Theorem 4.1. *Let $a \geq 1$ be a parameter. The randomized algorithm consisting of $N = Ca \log^2 n$ repetition of MinCut, where C is a suitable constant, has running time $\mathcal{O}\left(n^2 \log^3 n\right)$, and for every n-vertex input graph it computes the minimum cut size correctly with probability at least $1 - n^{-a}$.*

Algorithm 7 RandomContract(G,t)

while $|V(G)| > t$ **do**
  for random $e \in E(G)$: $G \leftarrow G/e$
end while
return G

Algorithm 8 MinCut(G)

if $n \leq 16$ **then**
  compute $\mu(G)$ by some deterministic method
else
  $t \leftarrow \lceil n/\sqrt{2} \rceil + 1$
  $H_1 \leftarrow$ RandomContract(G,t)
  $H_2 \leftarrow$ RandomContract(G,t)
  return min(MinCut(H_1),MinCut(H_1))
end if

5 Lovász Local Lemma

5.1 Constraint Satisfaction Problems

We are interested in a special class of combinatorial problems: *Constraint satisfaction problems*, or *CSP* for short. Their building blocks are variables and constraints. We consider the set of *variables* $V = \{x_1, \ldots, x_n\}$, each x_i with its own *domain* L_i. For simpler notation we also consider $L = \bigcup_i L_i$. An *assignment* if a function $\alpha : V \to L$ with $\alpha(x_i) \in L_i$ for all $x_i \in V$.

A *constraint* is a finite set of literals, where a *literal* is an expression of the form $x_i \neq v$ for some variable $x_i \in V_i$ and a value $v \in L_i$. Suppose

$$C = \{x_{i_1} \neq v_1, \ldots, x_{i_k} \neq v_k\}$$

The number of literals $k = |C|$ is the *size* of the constraint, and we write $\mathrm{vbl}\,(C) = \{x_{i_j} \mid 1 \leq j \leq k\}$ for the set of variables that appear in C.

The semantics of a constraint is as follows: it is not allowed to simultaneously assign x_{i_1} the value v_1 and x_{i_2} the value v_2 and ... and x_{i_k} the value v_k. Thus, you might want to read the constraint C as

$$[x_{i_1} \neq v_1] \vee [x_{i_2} \neq v_2] \vee \ldots \vee [x_{i_k} \neq v_k]$$

Correspondingly, C is said to be *satisfied* by a given assignment α, if for some $1 \leq j \leq |C|$ we have $\alpha(x_{i_j}) \neq v_j$. Note that an empty constraint thus can never be satisfied.

A *CSP* is now a finite set F of constraints, and we call F a *formula*. F is satisfied by α, if α satisfies all clauses of F.

5.2 Examples

We first consider hypergraphs $H = (V, E)$, that is a finite set V of vertices, and a set E of vertices, where $E \subseteq \mathcal{P}(V)$. Note that edges in a hypergraph can contain more than two edges, and regular graphs are a special case where $E \subseteq \binom{V}{2}$. A *c-coloring* of H is a mapping $V \to \{1, \ldots, c\}$, and it is called *proper* if no edge of H is monochromatic (i.e. only consists of same-colored vertices).

Historically, a hypergraph with a proper 2-coloring is said to have *property B*.

Lemma 5.1. *If all edges in a hypergraph $H = (V, E)$ have size k and $|E| < c^{k-1}$, then there exists a proper c-coloring of H.*

Proof. We choose a random coloring where every variable draws its value uniformly at random from its domain. Thus, the probability of a constraint not being satisfied is

$$\prod_{x \in \mathrm{vbl}\,(C)} \frac{1}{|L_x|}$$

In our setting all domains have size c, and all constraints have size k, that is the probability above becomes c^{-k}. There are $c \cdot |E|$ constraints, therefore the expected number of unsatisfied constraints is exactly $|E| \cdot c^{1-k}$.

With the hypotheses this number is less than 1, which shows that there is a positive probability that there is no unsatisfied constraint. This shows that there is an assignment that satisfies all constraints. □

We can get a very similar result for CNF-formulas, as the following lemma shows. Note that every CNF formula F translates naturally to a CSP formula F' as follows. V can be taken as the set of variables of F', each one with the domain $\{0,1\}$ of size 2. A literal x in F translates to the literal $x \neq 0$ in F' and $\overline{x}$ translates to $x \neq 1$.

Lemma 5.2. *If all clauses in a CNF-formula F have size k and the number of clauses is less than 2^k, then F is satisfiable.*

5.3 The Statement

For a constraint C we define its *weight* $w(C)$ to be the probability that an assignment to its variables chosen uniformly at random violates the constraint. That is,

$$w(c) = \prod_{x_i \in \mathrm{vbl}\,(C)} \frac{1}{|L_i|}$$

We define the *neighborhood* of C in F as

$$\Gamma_F(C) = \{D \in F \mid D \neq C, \mathrm{vbl}\,(C) \cap \mathrm{vbl}\,(D) \neq \emptyset\}$$

Using the neighborhoods, we can draw the so-called *dependency graph* G_F of a formula. Every constraint is a node in G_F, and there is an edge between $C, D \in F$ if $\mathrm{vbl}\,(C) \cap \mathrm{vbl}\,(D) \neq \emptyset$.

Theorem 5.1 (Lovász Local Lemma)**.** *Let F be a CSP. If there exist numbers $\mu_C \in (0,1)$ for $C \in F$ such that*

$$\forall C \in F: \; w(C) \leq \mu_C \prod_{D \in \Gamma_F(C)} (1 - \mu_D)$$

then F is satisfiable.

We can also consider a simpler case of this very general theorem. We say that the formula F is a (d,k)-CSP if each constraint has size k and the domains of the variable are $L_1 = \cdots = L_n = \{1, \ldots, d\}$. In such a formula, each constraint C has the same weight $w(C) = d^{-k}$.

Corollary 5.1. *Let F be a (d,k)-CSP for $d, k \geq 2$. If*

$$\forall C \in F: \; |\Gamma_F(C)| \leq \frac{d^k}{e} - 1$$

then F is satisfiable.

5.4 Finding a Satisfying Assignment

Algorithm 9 Fast algorithm to find a satisfying assignment.

```
for all x_i ∈ V do
    α(x_i) ←u.a.r. L_i
end for
while a violated constraint exists do
    choose any violated constraint C
    for all x_i ∈ vbl(C) do
        α(x_i) ←u.a.r. L_i
    end for
end while
return α
```

The algorithm 9 looks very simple and arbitrary, yet we claim that it solves the problem efficiently. In particular we are interested in the number N of *correction steps*, i.e. the number of times the loop is repeated and a constraint is selected and re-sampled.

Theorem 5.2. *If F is a CSP and $\mu_C \in (0,1)$ for $C \in F$ numbers such that the formula from theorem 5.1 is satisfied, then*

$$\mathbb{E}[N] \leq \sum_{C \in F} \frac{\mu_C}{1-\mu_C}$$

Note that theorem 5.2 implies theorem 5.1, as the algorithm can only terminate if a satisfying assignment is found.

Corollary 5.2. *Let F be a (d,k)-CSP for $d,k \geq 2$. If F satisfies the conditions of corollary 5.1, then $\mathbb{E}[N] = \mathcal{O}(|F|)$.*

5.5 Alternative Formulation

In this section we present an alternative formulation of the Local Lemma, taken from chapter 5 of *The Probabilistic Method* by Noga Alon and Joel H. Spencer.

Lemma 5.3. *Let $A_1, A_2, \ldots, A_n$ be events in an arbitrary probabilistic space. A directed graph $D = (V,E)$ on the set of vertices $V = \{1,2,\ldots,n\}$ is called dependency digraph for the events $A_1,\ldots,A_n$ if for each i, $1 \leq i \leq n$, the event A_i is mutually independent of all events $\{A_j \mid (i,j) \notin E\}$. Suppose that $D = (V,E)$ is a dependency digraph for the above events and suppose that there are real numbers $x_1, x_2, \ldots, x_n$ such that $0 \leq x_i < 1$ and*

$$\forall i:\ 1 \leq i \leq n:\ x_i \prod_{(i,j)\in E} (1-x_j)$$

Then

$$\Pr\left[\bigwedge_{i=1}^{n} \overline{A_i}\right] \geq \prod_{i=1}^{n} (1-x_i)$$

In particular, with positive probability no event A_i holds.

6 Randomized Algebraic Algorithms

6.1 Checking Matrix Multiplication

Consider the following procedure for checking whether C is the product of two matrices A and B. We pick a random n-component vector x of zeros and ones. More precisely, each vetro in $\{0,1\}^n$ appears with the same probability of 2^{-n}. The algorithm then computes Cx and ABx, both with $\mathcal{O}\left(n^2\right)$ operations. If the results agree, then the answer is YES, and NO otherwise.

But what is the success probability of such an algorithm? Let us set $D := C - AB$. We show that if D is any non-zero matrix and $x \in \{0,1\}^n$ is random, then the vector $y := Dx$ is zero with probability at most $\frac{1}{2}$.

Suppose that $d_{ij} \neq 0$. We want to derive that then the probability of $y_i = 0$ is at most $\frac{1}{2}$. We have

$$y_i = d_{i1}x_1 + d_{i2}x_2 + \cdots + d_{n1}x_n = d_{ij}x_j + S$$

If $d_{ij} = S$, then we detect the error if $x_j = 0$, and otherwise, we detect the error for certain if $x_j \neq 0$, possibly even if $x_j = 0$. Since x_j is chosen uniformly at random, in either case we have success probability of at least $\frac{1}{2}$.

By repeating the procedure a small number of times, we can bring the success probability down to a negligible value.

6.2 Is a Polynomial Identically Zero?

Definition 6.1 (Polynomial)**.** *A* general polynomial *in n variables is a finite sum of terms of the form $a_{i_1,i_2,\ldots,i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$ (called* monomials*), where the exponents $i_1, i_2, \ldots, i_n$ are non-negative integers and $a_{i_1,i_2,\ldots,i_n}$ is a coefficient. The* degree *of this term is $i_1 + i_2 + \cdots + i_n$, and the* degree of a polynomial *is the maximum of the degrees of its terms (with non-zero coefficients).*

Theorem 6.1 (Schwartz-Zippel Theorem)**.** *Let $p(x_1,\ldots,x_n) \in \mathbb{F}[x_1,\ldots,x_n]$ be a (non-zero) polynomial of degree $d \geq 0$, and let $S \subseteq \mathbb{F}$ be a finite set. Then the number of n-tuples $(r_1, r_2, \ldots, r_n) \in S^n$ with $p(r_1, r_2, \ldots, r_n) = 0$ is at most $d \cdot |S|^{n-1}$. In other words, if $r_1, \ldots, r_n \in S$ are chosen independently and uniformly at random, then the probability of $p(r_1, r_2, \ldots, r_n) = 0$ is at most $\frac{d}{|S|}$.*

6.3 Testing for Perfect Bipartite Matchings

We begin with the bipartite case. Let us consider a bipartite graph G with color classes $\{u_1, u_2, \ldots, u_n\}$ and $\{v_1, v_2, \ldots, v_n\}$. Let $\mathcal{S}_n$ denote the set of all permutations of $\{1, 2, \ldots, n\}$. We note that a perfect matching in G, if one exists, corresponds to a permutation $\pi \in \mathcal{S}_n$: It has the form

$$\{\{u_1, v_{\pi(1)}\}, \{u_2, v_{\pi(2)}\}, \ldots, \{u_n, v_{\pi(n)}\}\}$$

We introduce an $n \times n$ matrix B by letting

$$b_{ij} := \begin{cases} 1 & \text{if } \{u_i, v_j\} \in E(G) \\ 0 & \text{otherwise.} \end{cases}$$

A given permutation π determines a perfect matching in G if an only if the product $b_{1,\pi(1)} \cdot b_{2,\pi(2)} \cdots b_{n,\pi(n)}$ equals 1, and so the sum

$$\operatorname{perm}(B) := \sum_{\pi \in \mathcal{S}_n} b_{1,\pi(1)} \cdot b2, \pi(2) \cdots b_{n,\pi(n)}$$

called the *permanent* of B, counts all perfect matchings of G.

The definition of the permanent of a matrix looks very similar to the definition of its determinant, but the different is crucial. We can have $\det(B) = 0$ even if $\operatorname{perm}(B) \neq 0$, as nonzero terms may cancel out in the determinant.

To avoid such cancellations, we introduce one variable (indeterminate) x_{ij} for each edge $\{u_i, v_j\} \in E(G)$, and we define another matrix A:

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

So A is a function of the variables x_{ij}, and $\det(A)$ is a polynomial in these $|E(G)|$ variables. Since the determinant is the sum of products of n variables each, the degree of $\det(A)$ is n (unless the polynomial is constant 0).

Lemma 6.1. *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 to test whether G has a perfect matching. We cannot afford to compute the polynomial, as it can 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 random values for the x_{ij}. Recall here that Schwartz-Zippel tells us that the error probability is at most $\deg(\det(A))/|S|$, so this better be small. If we consider $\det(A)$ as polynomial with rational coefficients, we need to compute with numbers of up to n bits. Therefore, it is better to work with a finite field. The simplest way is to choose a prime number $2n \leq p < 4n$ and regard $\det(A)$ as polynomial with coefficients in $\mathrm{GF}(p)$. Then,

- It is still true that $\det(A)$ is nonzero if and only if G has a perfect matching.
- We choose the random values from at least $2n$ elements, so the probability of detecting that $\det(A)$ is nonzero is at least $\frac{1}{2}$.
- We can compute the determinant in $\mathcal{O}\left(n^3\right)$ operations, e.g. using Gaussian elimination.
- Such a prime p always exists.

6.4 Perfect Matchings in General Graphs

Again, $\mathcal{S}_n$ is the set of all bijective mappings $\{1, \ldots, n\} \to \{1, \ldots, n\}$. This set equipped with composition forms a group generated by transpositions (i, j), defined as

$$i \to j, j \to i \quad \text{and, for } k \notin \{i, j\}, \quad k \to k$$

Although a permutation can be written in several ways as composition of transpositions, 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: $\operatorname{sign}(\pi) = +1$ if the number of transposition is even, and $\operatorname{sign}(\pi) = -1$ otherwise.

Given a permutation $\pi \in \mathcal{S}_n$, the set $\{(i, \pi(i) \mid i \in \{1, \ldots, n\}\}$ constitutes a set of directed edges on the vertex set $\{1, \ldots, n\}$. Clearly, every vertex has out- and in-degree 1, and therefore it partitions $\{1, \ldots, n\}$ into cycles (some may be of length 1, i.e. loops). It turns out that the sign of a permutation is

$$(-1)^{\text{number of even cycles}}$$

that is the sign is positive if and only if the number of even cycles is even.

The algorithm from the previous section can work with arbitrary graphs, but we need to work with another matrix. For a graph G with a vertex set $V = \{v_1, \ldots, v_n\}$ we introduce a variable x_{ij} for every edge $\{v_i, v_j\} \in E(G)$, and we 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 6.2 (Tutte)**.** *The polynomial* $\det(A)$ *is nonzero if and only if* G *has a perfect matching.*

6.6 Counting Perfect Matchings in Planar Graphs

Let $G = (V_G, E_G)$, $V_G = \{1, \ldots, n\}$, be a graph with n even and let $\vec{G}$ be an orientation of G, i.e. every edge $\{i, j\}$ in G is represented by either (i, j) or (j, i) in $\vec{G}$. For such an orientation we define the following matrix

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

This is what is called a skew-symmetric matrix (meaning that $a_{ij} = -a_{ji}$, and in particular $a_{ii} = 0$). Note that the matrix is nothing else but the previously mentioned Tutte matrix A of G where we have plugged in +1 or -1 for the variables x_{ij}.

Lemma 6.2. *For the number of perfect matchings* $\mathrm{pm}(G)$ *it holds*

$$\det(A_S(\vec{G})) \leq \mathrm{pm}\,(G)^2$$

Proof. For perfect matchings M_1 and M_2 in G consider their union $U = M_1 \cup M_2$. This is an edge set $U \subseteq E_G$ consisting of even cycles and independent edges only. Given such a U, for how many pairs (M', M'') of perfect matchings can we write $U = M' \cup M''$? This is exactly 2^k, where k is the number of cycles in U (not counting the independent edges), because the edges of an even cycle partition into two perfect matchings of its vertex set and we can choose for M' one of the two for each even cycle. So if $\mathcal{U}$ denotes the set of all unions of two perfect matchings in G and $\|U\|$ is the number of even cycles in U, then

$$\mathrm{pm}\,(G)^2 = \sum_{U \in \mathcal{U}} 2^{\|U\|}$$

Note that for a permutation $\pi \in \mathcal{S}_n$ with all of its cycles even, the previously defined set $E_\pi = \{\{i, \pi(i)\} \mid i \in \{1, \ldots, n\}\}$ is of the form that it decomposes into even cycles and independent edges (for cycles of length 2 in π). It easily follows that

$$E_\pi \subseteq E_G \quad \iff \quad E_\pi \in \mathcal{U}$$

Given $U \in \mathcal{U}$, for how many permutations π do we have $E_\pi = U$? Interestingly, this is again $2^{\|U\|}$ since we have for each cycle of length at least 4 exactly two possibilities to orient the cycle - for the independent edges in U there is no choice for the cycle of length two associated with it. So we can extend the equivalence above to

$$\mathrm{pm}\,(G)^2 = \sum_{U \in \mathcal{U}} 2^{\|U\|} = |\,\mathrm{IE}\,|$$

where $\mathrm{IE} := \{\pi \in \mathcal{S}_n \mid E_\pi \subseteq E_G, \text{ all cycles in } \pi \text{ even}\}$, the set of important permutations with all cycles even. Now we inspect the determinant. We have argued before that

$$a(\pi) := a_{1\pi(1)} a_{2\pi(2)} \cdots a_{n\pi(n)} \neq 0 \quad \text{i.e. } +1 \text{ or } -1$$

if and only if $E_\pi \subseteq G$ and that the terms of such permutations with odd cycles cancel each other out. Therefore

$$\begin{aligned} \det(A_S(\vec{G})) &= \sum_{\pi \in \mathcal{S}_n} \mathrm{sign}\,(\pi) \underbrace{a(\pi)}_{=\pm 1} \\ &\leq |\,\mathrm{IE}\,| = \mathrm{pm}\,(G)^2 \end{aligned}$$

□

Whether or not a term $\operatorname{sign}(\pi)\, a(\pi)$ for $\pi \in \mathrm{IE}$ is +1 or -1 depends on how we orient the graph $\vec{G}$. Ideally, we wish to choose an orientation in such a way that this is always +1. We will see that this is indeed possible for planar graphs.

To this end, recall that $\operatorname{sign}(\pi) = (-1)^{\text{number of even cycles}}$. For a cycle c: $i_1 \to i_2 \to \cdots \to i_k \to i_1$ in a permutation π we write

$$a[c] := a_{i_1 i_2} a_{i_2 i_3} \cdots a_{i_{k-1} i_k} a_{i_k i_1}$$

Then, for $\pi \in \mathrm{IE}$

$$\operatorname{sign}(\pi)\, a(\pi) = \prod_{c \text{ cycle in } \pi} (-a[c])$$

since in $\pi \in \mathrm{IE}$ all cycles are even, so the minus in "$(-a[c])$" takes nicely care of the $\operatorname{sign}(\pi)$'s contribution. Note right away that for a cycle c: $i \to j \to i$ of length 2, $a[c] = a_{ij} a j i = -1$ (skew-symmetry) and therefore the term $(-a[c])$ is +1.

Now we would like to orient the graph in such a way that for any cycle c we have $a[c] = -1$.

6.6.1 Nice Cycles in G and Oddly Oriented Cycles in $\vec{G}$

Definition 6.2 (Oddly oriented)**.** *Let C be an undirected cycle in G of even length. We call C* oddly oriented *in $\vec{G}$ if going through C in 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.*

Definition 6.3 (Nice cycle)**.** *For C a cycle in G with vertex set V_C, we call C* nice*, if $G[V_G \setminus V_C]$ (the subgraph of G induced by $V_G \setminus V_C$) 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. Note also, that if permutation $\pi \in \mathrm{IE}$ has a cycle c of length at least 4, then its undirected counterpart C is a nice cycle in G. And $a[c] = -1$ if and only if C is oddly oriented in G.

Lemma 6.3. *If every nice cycle in G is oddly oriented in $\vec{G}$, then*

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

Orientations in $\vec{G}$ with $\det(A_S(\vec{G})) = \operatorname{pm}(G)^2$ are called *Pfaffian*.

6.6.2 Pfaffian Orientations of Planar Graphs

Planar graphs are graphs that can be drawn in the plane so that no pair of edges crosses. A concrete crossing-free embedding is called *plane graph*. If v is the number of vertices of a connected plane graph, e the number of edges, and f the number of faces, then Euler's relation says that $v - e + f = 2$.

A maximal planar graph (i.e. no further edge can be added without violating planarity) is called a *triangulation*. In any crossing-free embedding of a triangulation, all faces (including the unbounded one) are triangles, at least as long as the number of vertices is at least 3.

Lemma 6.4. *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. the region surrounded by C, not including C). Then $k \equiv v + 1 \mod 2$.*
- *$\vec{T}$ is a Pfaffian orientation.*

Theorem 6.3 (Kasteleyn)**.** *Every planar graph has a Pfaffian orientation which can be computed in linear time.*

Proof. 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 are well. This holds, since removing edges just means setting some entries in $A_S(\vec{G})$ to 0. In this way we cannot generate negative terms in the sum $\sum_{\pi \in \mathrm{IE}} \operatorname{sign}(\pi)\, a(\pi)$.

So it suffices to show the claim for triangulations. Let us fix a crossing-free embedding of a trinangulation T. T has a spanning tree B, so far so good. Now consider the dual graph B^* of T where we use only edges not in B. That is, every face represents a vertex (including the infinite face), and two such vertices are connected

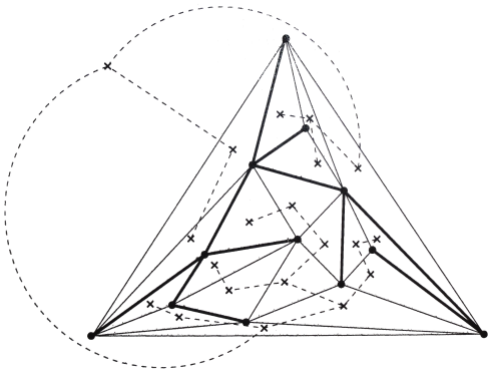

Figure 1: The trees B and B^* for a triangulation.

by an edge if their faces share a common edge which is not in B. Since B has no cycles, B^* is connected, and since B is connected, B^* has no cycles. So B^* is a spanning tree of the dual graph of T. Figure 1 shows the trees B and B^* for a triangulation.

Towards the Pfaffian orientation $\vec{T}$, start by orienting the edges of B arbitraily. Now consider a leaf in B^*. This represents a face t, a triangle, where two edges are already oriented and one is still floating. Choose an orientation for this floating edge so that things to well for t, i.e. so that t has an odd number of clockwise oriented edges. Remove the vertex representing t from B^*, and continue the same, but never choosing the infinite face, even if it turns into a leaf while B^* get truncated. In this way all faces have indeed an odd number of clockwise edges, except for the infinite face, which we are perfectly happy to accept.

All steps can be done in linear time. □

7 Cryptographic Reductions

7.1 Introduction

7.1.1 Computational Problems

An important type of computational problem are *decision problems*. A decision problem is characterized by a predicate $Q : \mathcal{X} \times \mathcal{W} \to \{0, 1\}$, where $\mathcal{X}$ is an instance space and $\mathcal{W}$ is a witness space. The problem consists of deciding, for a given $x \in \mathcal{X}$, whether there exists a witness $w \in \mathcal{W}$ such that $Q(x, w) = 1$.

Another type of problem which is of interest is called *search problem* and is the problem of *finding* a witness (which can be called a solution). Typically one considers problems where it is known (or guaranteed) that a witness exists.

Definition 7.1 (One-way function)**.** *A function f is a* on-way function *if no efficient algorithm can solve the inversion problem with non-negligible success probability.*

7.2 The Diffie-Hellman Key-Agreement Protocol

The Diffie-Hellman key-agreement (or key-distribution) protocol is based on the *discrete logarithm problem*. This problem is conjectured to be generally computationally infeasible, i.e. exponentiation modulo a large prime is a (conjectured) one-way function. The protocol uses only an authenticated channel to generate a shared, secret key.

This works as follows: A large prime p and a basis g are chosen as public parameters. Then, both Alice and Bob choose a number at random from $0, \ldots, p-2$, say x_A and x_B respectively. They exponentiate the base by there secret parameter modulo p and send the result, y_A and y_B, to the other person. Now, both parties exponentiate what they got by their secret parameter (again, modulo p), and thereby get the secret key:

$$k_{AB} \equiv_p y_B^{x_A} \equiv_p (g^{x_B})^{x_A} \equiv g^{x_A x_B} \equiv k_{BA}$$

This protocol can be generalized from computation modulo p to any cyclic group $G = \langle g \rangle$ generated by a generator g in which the discrete logarithm problem is computationally hard. The only modifications are x_A and x_B must be selected from $\{0, \ldots, |G| - 1\}$ and multiplication modulo p is replaced by the group operation of G. In practice, one often uses so-called elliptic curves for which the discrete logarithm is believed to be substantially more difficult than for the group $\mathbb{Z}_p^*$.

7.2.1 Security of the Diffie-Hellman Protocol

The described protocol is believed to be a computationally secure key agreement protocol. To investigate the security, we state three assumptions for a cyclic group $G = \langle g \rangle$.

Definition 7.2 (DL-assumption)**.** *The* discrete logarithm (DL) assumption *holds for G if it is computationally infeasible to compute a from g^a for uniformly chosen $a \in \{0, \ldots, |G| - 1\}$.*

Definition 7.3 (CDH-assumption)**.** *The* computational Diffie-Hellman (CDH) assumption *holds for G if it is computationally infeasible to compute g^{ab} from g^a and g^b for uniformly chosen $a, b \in \{0, \ldots, |G| - 1\}$.*

Definition 7.4 (DDH-assumption)**.** *The* decisional Diffie-Hellman (CDH) assumption *holds for G if, if for uniformly chosen $a, b, c \in \{0, \ldots, |G| - 1\}$, the triples (g^a, g^b, g^{ab}) and (g^a, g^b, g^c) are computationally indistinguishable.*

It is easy to see that the DDH-assumption implies the CDH-assumption which implies the DL-assumption, as can be proven by trivial reductions.

7.3 Computational Methods for Computational Problems

7.3.1 Definitions

Definition 7.5. *We consider an abstract setting with a set $\mathcal{M}$ of* computational methods (CM)*, a set $\mathcal{P}$ of* computational problems (CP) *and a* performance function

$$\pi : \mathcal{M} \times \mathcal{P} \to \mathbb{R}$$

where $\pi(A, C)$ is the performance of CM A on CP C. An α-solver *for CP C is a CM with performance at least α.*

Definition 7.6. *A* computational method constructor (CMC) *R is a mapping $\mathcal{M} \to \mathcal{M}$ which for any (invoked) CM A defines the CM denoted by $R(A)$ or simply RA.*

One can associate some form of cost or complexity with every computational method.

Definition 7.7. *A* complexity function *is a function*

$$\gamma : \mathcal{M} \to R^+$$

where $\gamma(A)$ is the complexity of A. A non-decreasing function $\gamma_R : \mathbb{R}^+ \to \mathbb{R}^+$ is a complexity function for a CMC R if

$$\forall A \; \gamma(RA) \leq \gamma_R(\gamma(A))$$

Definition 7.8. *The* performance-complexity function *of a CP C is the function $\overline{\pi}^C : \mathbb{R}^+ \to \mathbb{R}$ defined by*

$$\overline{\pi}^C(c) = \sup_{A:\gamma(A)\leq c} \pi(A, C)$$

that is $\overline{\pi}^C(c)$ is the performance of the best solver C with complexity at most c.

7.4 Reductions

7.4.1 The Reduction Concept

Definition 7.9. *A* ρ-reduction *of a CP C to a CP C' (for a function $\rho : \mathbb{R} \to \mathbb{R}$) is a CMC R which is a $\rho(\alpha)$-solver for C when it invokes an arbitrary α-solver for C'. This is denoted as $C \preceq_\rho^R C'$. More formally:*

$$C \preceq_\rho^R C' \quad \Longleftrightarrow \quad \forall A \; \forall \alpha \left(\pi(A, C') \geq \alpha \; \rightarrow \; \pi(RA, C) \geq \rho(\alpha) \right)$$

This does not yet involve complexities, which are required when we give interpretations of the reduction concept.

A reduction can be interpreted in two different ways. A direct formulation of the above definition of a reduction reads as follows: If A is an α-solver for C', then RA is a $\rho(\alpha)$-solver for C. This formulation is useful if one knows how to construct a solver for C' and wants to obtain one for C.

The definition can also be read in the reverse direction: If there exists no $\rho(\alpha)$-solver with complexity $\gamma_R(c)$ for C, then there exists no α-solver with complexity c for C'. This formulation is useful if one wants to prove a lower bound on the complexity of problem C', assuming one knows a lower bound on the complexity of problem C. In other words we have

Lemma 7.1. *If $C \preceq_\rho^R C'$ and $\overline{\pi}^C(\gamma_R(c)) < \rho(\alpha)$, then $\overline{\pi}^{C'}(c) \leq \alpha$. Formally:*

$$C \preceq_\rho^R C' \quad \Longrightarrow \quad \forall c \; \forall \alpha \left(\overline{\pi}^C(\gamma_R(c)) < \rho(\alpha) \; \rightarrow \; \overline{\pi}^{C'}(c) \leq \alpha \right)$$

Proof. To arrive at a contradiction, suppose that $C \preceq_\rho^R C'$ and $\overline{\pi}^C(\gamma_R(c)) < \rho(\alpha)$, but that $\overline{\pi}^{C'}(c) > \alpha$. Then, $\overline{\pi}^{C'}(c) > \alpha$ implies that there exists a A with $\gamma(A) \leq c$ and $\pi(A, C') > \alpha$. Hence, due to $C \preceq_\rho^R C'$, there exists a A' (namely $A' = RA$) with $\gamma(A') \leq \gamma_R(c)$ and $\pi(A', C) \geq \rho(\alpha)$. Thus $\overline{\pi}^C(\gamma_R(c)) \geq \rho(\alpha)$, which is a contradiction. □

Reductions can also be composed, as shown in the following lemma.

Lemma 7.2. *If R is a ρ-reduction of C to C' and if R' is a ρ'-reduction of C' to C'', then RR' is a $\rho\rho'$-reduction of C to C''. More formally,*

$$C \preceq_\rho^R C' \; \wedge \; C' \preceq_{\rho'}^{R'} C'' \quad \Longrightarrow \quad C \preceq_{\rho\rho'}^{RR'} C''$$

7.4.2 Generalized Reductions

Definition 7.10. *A* $(\rho_1, \ldots, \rho_n)$-reduction *of a list* $(C_1, \ldots, C_n)$ *of CPs to a CP* C *(for functions* $\rho_i : \mathbb{R} \to \mathbb{R}$*) is a list* $(R_1, \ldots, R_n)$ *of CMC such that for every* α*-solver* A *of* C*, for some* $1 \leq i \leq n$*,* $R_i A$ *is a* $\rho_i(\alpha)$*-solver for* C_i*. More formally:*

$$(C_1, \ldots, C_n) \preceq^{(R_1,\ldots,R_n)}_{(\rho_1,\ldots,\rho_n)} C \quad \iff \quad \forall A \, \forall \alpha \left(\pi(A, C) \geq \alpha \; \to \; \bigvee_{i=1}^{n} \pi(R_i A, C_i) \geq \rho_i(\alpha) \right)$$

Such generalized reductions can of course also be composed.

Lemma 7.3. *If* $(C_1, \ldots, C_n) \preceq^{(R_1,\ldots,R_n)}_{(\rho_1,\ldots,\rho_n)} C$ *and* $\overline{\pi}^{C_i}(\gamma_{R_i}(c)) < \rho_i(\alpha)$ *for all* i*, then* $\overline{\pi}^{C}(c) \leq \alpha$*.*

7.5 Search Problems

7.5.1 Definition

Definition 7.11. *A* search problem $C = (\mathcal{X}, \mathcal{Y}, Q, \mathrm{P}^C_X)$ *is defined by an* instance space $\mathcal{X} \supseteq \{0,1\}^*$*, a* solution *(or* witness*) space* $\mathcal{Y} \subseteq \{0,1\}^*$*, a predicate* $Q : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$*, as well as a distribution* P^C_X *over the instance space. The search problem consists of finding, for a given instance* $x \in \mathcal{X}$*, a* $y \in \mathcal{Y}$ *such that* $Q(x, y) = 1$*. Such a* y *is called a solution or witness. Typically one assumes that every instance has at least one solution, but other cases can also be considered by allowing solvers that can can output "no solution".*

Definition 7.12. *A* solver *for a search problem* $C = (\mathcal{X}, \mathcal{Y}, Q, \mathrm{P}^C_X)$ *is a (possibly randomized) CM (e.g. an algorithm)* A *that takes as input an instance* $x \in \mathcal{X}$ *and outputs* $Y \in \mathcal{Y} \cup \{\bot\}$ *(where* $\bot$ *means that no output is produced). The* success probability of A on input x *is*

$$suc^A(x) = \mathrm{P}^{CA}(Q(X, Y) = 1 \mid X = x)$$

Note that A defines a conditional probability distribution $p^A_{Y|X}$ and that

$$suc^A(x) = \sum_{y \in \mathcal{Y}:\; Q(x,y)=1} p^A_{Y|X}(y, x)$$

We can define two performance measures for search problems, the average-case and the worst-case success probability. Note that a search problem is a CP only when the performance measure is defined.

Definition 7.13. *The (average)* success probability *of* A *for* C *is*

$$\pi(A, C_{av}) = \mathbb{E}\left[suc^A(X)\right] = \mathbb{E}\left[\mathrm{P}^{CA}(Q(X, Y) = 1)\right]$$

where X *is chosen according to* P^C_X*. The* worst-case success probability *of* A *for* C *is*

$$\pi(A, C_{wc}) = \inf_{x \in \mathcal{X}} suc^A(x)$$

7.5.2 Performance Amplifications for Search Problems

If the predicate Q can be evaluated efficiently, one might be tempted to repeat the α solver A many times to achieve a success probability very close to one. However, this is in general not possible, as it is very well possible that A is successful with probability 1 on a α fraction of the inputs, and fails always in the other cases.

We can investigate reductions between C_{av} and C_{wc}. For this reasoning, let id be the trivial reduction that just calls the solver without modification, and let $\mathrm{rep}\,(q)$ be the simple reduction that calls the solver q times.

Lemma 7.4. $C_{\mathrm{av}} \preceq^{\mathrm{id}} C_{\mathrm{wc}}$

Lemma 7.5. $C_{\mathrm{wc}} \preceq_{\rho}^{\mathrm{rep}\,(q)} C_{\mathrm{wc}}$ *for any* q*, where* $\rho(\alpha) = 1 - (1 - \alpha)^q$*.*

If we want to reduce a worst-case solver to an average-case solver, we need a special property of the computational problem:

Definition 7.14. *A search problem* $C = (\mathcal{X}, \mathcal{Y}, Q, \mathrm{P}^C_X)$ *is called* random self-reducible *if there exists an algorithm* R *that can perform two tasks:*

1. *It converts any instance $x \in \mathcal{X}$ into a random instance x' (distributed according to P_X^C), and*
2. *it converts any solution for x' into a solution for x.*

For instance, the discrete logarithm problem is self-reducible.

Lemma 7.6. *If C is random self-reducible, then $C_{\mathrm{wc}} \preceq^R C_{\mathrm{av}}$.*

We can use lemmas 7.2, 7.4 and 7.6 to obtain the following result:

Theorem 7.1. *If C is random self-reducible, then*

$$C_{\mathrm{av}} \preceq_\rho^{\overline{R}} C_{\mathrm{av}}$$

for any q, where $\rho(\alpha) = 1 - (1-\alpha)^q$ and where $\overline{R}$ calls the solver q times with randomized instances.

This theorem can be interpreted in two different ways, which we present for the DL problem. One conclusion from the theorem is that if the DL problem cannot efficiently be solved with high success probability, then it cannot even be solved with small probability. A dual conclusion is that in order to solve the DL problem with high success probability, it suffices to find an algorithm with only a very small success probability.

7.6 Hardness Amplifications for Search Problems

In this section, search problems are always considered with the average success probability as the performance measure.

7.6.1 Combining Computational Problems

Definition 7.15. *For $C_1 = (\mathcal{X}_1, \mathcal{Y}_1, Q_1, \mathrm{P}_{X_1}^{C_1})$ and $C_2 = (\mathcal{X}_2, \mathcal{Y}_2, Q_2, \mathrm{P}_{X_2}^{C_2})$ we define the combined search problem as*

$$C_1 \wedge C_2 = (\mathcal{X}_1 \times \mathcal{X}_2, \mathcal{Y}_1 \times \mathcal{Y}_2, Q_1 \wedge Q_2, \mathrm{P}_{X_1}^{C_1} \cdot \mathrm{P}_{X_2}^{C_2})$$

7.6.2 A Lemma on Measures

Definition 7.16. *A* measure *on a set S (or simply a S-measure) is a function $S \to [0,1]$.*

Definition 7.17. *For a given probability distribution P_S on S and a measure μ on S, the* average *of μ is*

$$\overline{\mu} = \sum_{s \in S} \mathrm{P}_S(s) \cdot \mu(s)$$

Consider now a measure μ on the product set $\mathcal{S} \times \mathcal{T}$, as well as a probability distribution P_S on $\mathcal{S}$ and P_T on $\mathcal{T}$, defining (independent) random variables S and T, respectively. We define $\overline{\mu}_1(s) = \mathbb{E}[\mu(s,T)]$ as the average of μ when $S = s$, and similarly for t, i.e.

$$\overline{\mu}_1 = \sum_{t \in \mathcal{T}} \mathrm{P}_T(t) \cdot \mu(s,t) \qquad \text{and} \qquad \overline{\mu}_2 = \sum_{s \in \mathcal{S}} \mathrm{P}_S(s) \cdot \mu(s,t)$$

We have $\overline{\mu} = \mathbb{E}[\overline{\mu}_1(S)] = \mathbb{E}[\overline{\mu}_2(T)]$.

Lemma 7.8. *Consider arbitrary random variables S and T, as above. For $\beta, \beta' \in [0,1]$ any ε and ε', and every measure μ on $\mathcal{S} \times \mathcal{T}$ with average*

$$\overline{\mu} \geq \beta\beta' + \varepsilon \cdot (1-\beta) + \varepsilon' \cdot (1-\beta')$$

we have either

$$\mathrm{P}(\overline{\mu}_1(S) \geq \varepsilon) \geq \beta \qquad \text{or} \qquad \mathrm{P}(\overline{\mu}_2(T) \geq \varepsilon') \geq \beta'$$

7.6.3 Hardness Amplification for Two Instances

A solver A for $C_1 \wedge C_2$ can be used as a solver $R_1 A$ for C_1 and as a solver $R_2 A$ for C_2, for the following reductions R_1 and R_2. The theorem below proves lower bounds on the performance of these solvers.

R_1 works as follows. Given $x_1 \in \mathcal{X}_1$, for at most q_1 times (where q_1 is a parameter) choose a random value $x_2 \in \mathcal{X}_2$, call $A(x_1, x_2)$, check whether A has returned a correct solution $(y_1, y_2) \in \mathcal{Y}_1 \times \mathcal{Y}_2$, and if so R_1 returns y_1 and stops. R_2 works analogously.

Theorem 7.2. *For any $\delta_1, \delta_2 \in [0,1]$ and any function $\rho_1(\alpha)$ and $\rho_2(\alpha)$ satisfying*

$$(\rho_1(\alpha) + \delta_1) \cdot (\rho_2(\alpha) + \delta_2) \leq \alpha$$

we have

$$(C_1, C_2) \preceq^{(R_1,R_2)}_{(\rho_1,\rho_2)} C_1 \wedge C_2$$

for

$$q_1 = \frac{3 \ln\left(\frac{2(\rho_1(\alpha)+\delta_1)}{\delta_1}\right)}{(\delta_1 \cdot (\rho_2(\alpha) + \delta_2))} \qquad \text{and} \qquad q_2 = \frac{3 \ln\left(\frac{2(\rho_2(\alpha)+\delta_2)}{\delta_2}\right)}{(\delta_2 \cdot (\rho_1(\alpha) + \delta_1))}$$

7.6.4 Hardness Amplification for Many Instances

The statement on hardness amplifications can be generalized to n search problem, i.e. we consider the search problem $C_1 \wedge \cdots \wedge C_n$ where $C_i = (\mathcal{X}_i, \mathcal{Y}_i, Q_i, \mathrm{P}^{C_i}_{X_i})$. We define

$$\overline{\mu}_i(x_i) = \mathbb{E}\left[\mu(X_1, \ldots, X_{i-1}, x_i, X_{i+1}, \ldots, X_n)\right]$$

Lemma 7.8 the becomes:

Lemma 7.10. *For any $\beta, \varepsilon \in [0,1]$ and every measure μ on $\mathcal{X}_1 \times \cdots \times \mathcal{X}_n$ with average $\overline{\mu} \geq \beta^n + n\varepsilon(1-\beta)$, there exists an $i \in \{1, \ldots, n\}$ such that*

$$\mathrm{P}\left(\overline{\mu}_i(X_i) \geq \varepsilon\right) \geq \beta$$

Theorem 7.1 can also be generalized to state that the hardness of a search problem C can be computed by taking as a new search problem C^n n copies of C in parallel: $C^n = C \wedge \cdots \wedge C$. The theorem roughly states that if there exists an efficient α-solver for C^n, then there exists an efficient $\sqrt[n]{\alpha}$-solver for C (note that in general $\sqrt[n]{\alpha} \gg \alpha$). Or, turned around: If there exists no efficient α-solver for C, then there exists no efficient $\sqrt[n]{\alpha}$-solver for C^n.

Let R be the reduction that works as follows. Given $x \in \mathcal{X}$, for at most q times (where q is a parameter) choose a random $i \in \{1, \ldots, n\}$ and $n-1$ random values $x_1, \ldots, x_{i-1}, x_{1+1}, \ldots, x_n \in \mathcal{X}$, let $x_i = x$, and call $A(x_1, \ldots, x_n)$. If A is successful and returns a solution $(y_1, \ldots, y_n) \in \mathcal{Y}^n$, then R returns y_i and stops.

Theorem 7.3. *For any $\delta \in [0,1]$, and any $\rho(\alpha)$ satisfying*

$$(\rho(\alpha) + \delta)^n \leq \alpha$$

we have

$$C \preceq^R_\rho C^n$$

for $q = \mathcal{O}\left(\ln(\frac{\tau}{\delta}/(\delta(\alpha) + \delta)\right)$.

7.7 Bit-Guessing Problems

7.7.1 Definitions

Definition 7.18. *A* bit-guessing problem $\mathcal{B} = (\mathcal{X}, \mathrm{P}^{\mathcal{B}}_{X\,B})$ *is defined by an instance space $\mathcal{X}$ and a probability distribution $\mathrm{P}^{\mathcal{B}}_{X\,B}$ over $\mathcal{X} \times \{0,1\}$.*

One can think of either B or X as being first generated, and then the other random variable as being generated according to the appropriate conditional distribution:

$$\mathrm{P}^{\mathcal{B}}_{X\,B} = \mathrm{P}^{\mathcal{B}}_B \, \mathrm{P}^{\mathcal{B}}_{X|B} = \mathrm{P}^{\mathcal{B}}_X \, \mathrm{P}^{\mathcal{B}}_{B|X}$$

7.7.2 Guessers for Bit-Guessing Problems

Definition 7.19. *A* guesser *for bit guessing problem $\mathcal{B} = (\mathcal{X}, \mathrm{P}^{\mathcal{B}}_{X\,B})$ is a (possibly randomized) CM A that takes as input an $x \in \mathcal{X}$ and outputs a bit Z. (A is characterized by a conditional probability $\mathrm{p}^{A}_{Z|X}$.) The* advantage *of A for input $x \in \mathcal{X}$ is*

$$adv^{A}(x) = 2 \cdot \left(\mathrm{P}^{\mathcal{B}A}(Z = B \mid X = x) - \frac{1}{2} \right)$$

The factor 2 is used to normalize the advantage to the range $[-1, 1]$, where -1 means that always $Z \neq B$, 0 is simply achievable by an unbiased random bit Z, and 1 means that always $Z = B$. We have

$$\mathrm{P}^{\mathcal{B}A}(Z = B \mid X = x) = \mathrm{P}^{\mathcal{B}A}_{B \oplus Z|X}(0, x) = \tfrac{1}{2} + \tfrac{1}{2} adv^{A}(x)$$

The natural performance measure for bit-guessing problems is the average advantage, which we consider here as the standard performance measure:

$$\pi(A, \mathcal{B}) = \mathbb{E}\left[adv^{A}(X)\right] = 2 \cdot \left(\mathrm{P}^{\mathcal{B}A}(Z = B) - \tfrac{1}{2}\right)$$

7.7.3 Computing the Advantage

The advantage is best expressed by introducing the quantities $\mu(x)$ and $\nu(v)$ as the (normalized) deviation from $\frac{1}{2}$ of $\mathrm{p}^{A}_{Z|X}(1, x)$ and $\mathrm{P}^{\mathcal{B}}_{B|X}(1, x)$,respectively:

$$\mu(x) = 2\left(\mathrm{p}^{A}_{Z|X}(1, x) - \tfrac{1}{2}\right) \qquad \Longleftrightarrow \qquad \mathrm{p}^{A}_{Z|X}(1, x) = \tfrac{1}{2} + \tfrac{1}{2}\mu(x)$$

and

$$\nu(x) = 2\left(\mathrm{P}^{\mathcal{B}}_{B|X}(1, x) - \tfrac{1}{2}\right) \qquad \Longleftrightarrow \qquad \mathrm{P}^{\mathcal{B}}_{B|X}(1, x) = \tfrac{1}{2} + \tfrac{1}{2}\nu(x)$$

Then

$$\mathrm{P}^{\mathcal{B}A}(Z = B \mid X = x) = \tfrac{1}{2} + \tfrac{1}{2}\mu(x)\nu(x)$$

and hence

$$adv^{A}(x) = \mu(x)\nu(x) \qquad \text{and} \qquad \pi(A, \mathcal{B}) = \mathbb{E}\left[\mu(X)\nu(X)\right]$$

7.8 Performance Amplification for Guessers

Suppose that for a bit-guessing problem $\mathcal{B} = (\mathcal{X}, \mathrm{P}^{\mathcal{B}}_{XB})$ we have a guesser A with performance $\pi(A, \mathcal{B})$ and that we want to amplify its performance. In general this is not possible. However, if it is known that for every x the bit output by A is not far from being unbiased, namely

$$\forall x \; |\mu(x)| \leq t$$

for some t, then the performance can be amplified by a scaling factor $\sigma \approx 1/t$ by scaling up $\mu(x)$ by σ as follows. Let the guesser A^{σ} with parameter $\sigma \leq 1/t$ be defined as follows: For input $x \in \mathcal{X}$, A^{σ} flips the bit Z (probabilistically) according to

$$\mathrm{p}^{A^{\sigma}}_{Z|X}(1, x) = \tfrac{1}{2} + \tfrac{1}{2}\sigma\mu(x)$$

Note that for $\sigma = 1$, A^{σ} is identical to A, and for $\sigma > 1$ the only difference between A and A^{σ} is that $\mu(x)$ is replaced by $\sigma\mu(x)$. We have

$$adv^{A^{\sigma}}(x) = \sigma\mu(x)\nu(x)$$

Hence, A^{σ} is better than A by a factor of σ:

$$\pi(A^{\sigma}, \mathcal{B}) = \mathbb{E}\left[adv^{A^{\sigma}}(X)\right] = \sigma\mathbb{E}\left[\mu(X)\nu(X)\right] = \sigma\pi(A, \mathcal{B})$$

Note that $\mu(x)$ is not known in general (for given x), but can be estimated by a statistical procedure.

7.9 Hardness Amplification for Bit-Guessing Problems - the "XOR-Lemma"

7.9.1 Combining Bit Guessing Problems

Definition 7.20. *For two BGPs $\mathcal{B}_1 = (\mathcal{X}_1, \mathrm{P}^{\mathcal{B}_1}_{X_1 B_1})$ and $\mathcal{B}_2 = (\mathcal{X}_2, \mathrm{P}^{\mathcal{B}_2}_{X_2 B_2})$, the BGP $\mathcal{B}_1 \oplus \mathcal{B}_2$ is defined as guessing $B_1 \oplus B_2$ when given a pair (X_1, X_2), i.e.*

$$\mathcal{B}_1 \oplus \mathcal{B}_2 = (\mathcal{X}_1 \times \mathcal{X}_2, \mathrm{P}^{\mathcal{B}_1\mathcal{B}_2}_{(X_1,X_2),B_1 \oplus B_2})$$

7.9.2 Guessers for $\mathcal{B}_1$ and $\mathcal{B}_2$

Consider a guesser for A for $\mathcal{B}_1 \oplus \mathcal{B}_2$ with advantage $\pi(A, \mathcal{B}_1 \oplus \mathcal{B}_2) = \alpha$. We consider the following two guessers derived from A, one for $\mathcal{B}_1$ and one for $\mathcal{B}_2$.

For a fixed x_2, let $A_1^{x_2}$ be the guesser for $\mathcal{B}_1$ which, for a given instance x_1, simply calls $A(x_1, x_2)$ and outputs this bit. We have

$$adv^{A_1^{x_2}}(x_1) = adv^A(x_1, x_2)$$

and hence

$$\pi(A_1^{x_2}, \mathcal{B}_1) = \mathbb{E}^{\mathcal{B}_1 A}\left[adv^A(X_1, x_2)\right] =: \varphi(x_2)$$

where $\varphi(x_2)$ is defined as the "row-average" of $adv^A(x_1, x_2)$ for the row labeled x_2.

We now define a guesser A_2 for $\mathcal{B}_2$ as follows: For a given instance x_2, A_2 generates a sample of (X_1, B_1) according to $\mathrm{P}_{X_1 B_1}^{\mathcal{B}_1}$, calls $A(X_1, x_2)$ to obtain a bit Z, and output $Z \oplus B_1$ as the guess for bit B_2. A_2 is correct if and only if A is correct, i.e.

$$adv^{A_2}(x_2) = \mathbb{E}^{\mathcal{B}_1 A}\left[adv^A(X_1, x_2)\right] = \varphi(x_2)$$

and hence

$$\pi(A_2, \mathcal{B}_2) = \alpha$$

7.9.3 Hardness Amplification for Two Instances

Theorem 7.4. *For any $\delta \in [0, 1]$ and any functions $\rho_1(\alpha)$ and $\rho_2(\alpha)$ satisfying*

$$(\rho_1(\alpha) + \delta)\, \rho_2(\alpha) \leq \alpha$$

we have

$$(\mathcal{B}_1, \mathcal{B}_2) \;\preceq^{(R_1, R_2)}_{(\rho_1, \rho_2)}\; \mathcal{B}_1 \oplus \mathcal{B}_2$$

where R_1 makes a single call to A and R_2 makes $q = \mathcal{O}\left(-\log \rho_1(\alpha)/\delta^2\right)$ calls to A.

8 Introduction to PCP

8.1 Approximation: Algorithms and Hardness

A common way of handling NP-hard optimization problems is by approximation algorithms. One tries to design an efficient algorithm that finds a solution whose value is not too far from the value of an optimal solution.

8.1.1 The Approximation Ratio

Let us consider an optimizations problem, where the goal is to maximize some quantity $Q \in (0, \infty)$, and let A be an approximation algorithm for it. Let $Q_{\mathrm{OPT}}(I)$ denote the maximum A for a given input I, while $Q_A(I)$ denotes the value achieved by A for the same input I. Since we deal with a maximization problem, we always have $Q_{\mathrm{OPT}}(I) \geq Q_A(I)$.

We say that the algorithm A has *approximation ratio* ρ (or that is a *ρ-approximation algorithm*), where $\rho \in (0, 1]$, if

$$\frac{Q_A(I)}{Q_{\mathrm{OPT}}(I)} \geq \rho \quad \text{for all inputs } I$$

Sometimes the approximation ratio deteriorates as a function of the size of the input, and it is estimated by a function of the size. That is, the algorithm has approximation ratio $\rho(n)$ if

$$\frac{Q_A(I)}{Q_{\mathrm{OPT}}(I)} \geq \rho(n) \quad \text{for all inputs } I \text{ of size at most } n$$

Similar definitions are made for minimization problems, where we have

$$\frac{Q_A(I)}{Q_{\mathrm{OPT}}(I)} \leq \rho \quad \text{for all inputs } I$$

8.1.2 A Maximization Problem: max-3-SAT

We first recall the well-known NP-complete problem 3-SAT. The input is a boolean formula in conjunctive normal form, a *3-CNF formula* for short. In general there are m *clauses*, each consisting of 3 *literals*, where a literal is a variable x_i or its negation $\overline{x}_i$. There are n variables $x_1, \ldots, x_n$, and we always have $n \leq 3m$.

A 0/1-assignment to the variables satisfies a clause if at least one literal is assigned 1. An assignment is *satisfying* if it satisfies all m clauses, and a formula is satisfiable if it has a satisfying assignment. The problem 3-SAT is to determine whether a given 3-CNF formula is satisfiable. Since it is NP-complete, there is no polynomial time algorithm for it unless P=NP.

The problem max-3-SAT is an optimization problem arising naturally from 3-SAT: Given a 3-CNF formula φ, we want to compute an assignment such that the maximum possible number of clauses are satisfied.

Let $m_{\mathrm{OPT}} = m_{\mathrm{OPT}}(\varphi)$ denote the number of clauses of φ that are satisfied by an optimal assignment. Since we have $m_{\mathrm{OPT}} = m$ exactly if φ is satisfiable, we can see that max-3-SAT is at least as hard as 3-SAT, and thus is NP-hard.

We recall that being NP-hard means that for any $L \in \mathrm{NP}$ there exists a polynomial time reduction Φ which maps instances of L to instances of 3-SAT such that if $x \in L$ then $\Phi(x)$ is a satisfiable 3-SAT instance, and if $x \notin L$ then $\Phi(x)$ is unsatisfiable. However, as max-3-SAT is not a yes/no problem, it does not belong to NP, and is thus not NP-complete.

First we note that every clause is satisfied by the assignment of all 1's or by the assignment of all 0's. Hence, the following trivial algorithm has approximation ratio $\frac{1}{2}$: Return the better of the two assignments $x_1 = \cdots = x_n = 1$ and $x_1 = \cdots = x_n = 0$.

Next, we restrict ourselves to a subclass of all 3-CNF formulas: We call a 3-CNF formula *proper* if each variable occurs at most once in every clause. We claim that every proper 3-CNF formula with m clauses has an assignment that satisfies at least $\frac{7}{8}m$ of the clauses.

To prove this, we consider a random assignment, where every x_i is set randomly and independently to 0 or 1, each value having probability $\frac{1}{2}$. We check that the expected number of clauses satisfied by a random assignment is $\frac{7}{8}m$, and hence there always exists an assignment that satisfies at least $\frac{7}{8}m$ of the clauses.

8.1.3 Inapproximability

Theorem 8.1 (PCP Theorem; approximation version)**.** *There exists a constant $\rho > 0$ such that if a polynomial time algorithm approximates 3-SAT with ration $1 - \rho$ then P=NP.*

We now generalize the notion of NP-hardness: we want to be able to say that the reduction only outputs *some* strings (the members of Y and N below).

Definition 8.1. *Let $Y, N \subseteq \{0,1\}^*$ be two disjoint sets. We say that it is "NP-hard to distinguish Y from N" if, for any $L \in$ NP there exists a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L$ implies $f(x) \in Y$ and $x \notin L$ implies $f(x) \in N$.*

Note that the roles of Y and N in this definition are not symmetric. Also, we remark that this definition generalizes the usual definition of NP-hardness, where N has set to be $\{0,1\}^* \setminus Y$ and Y is the language which is NP-hard.

Theorem 8.3. *There exists $\rho > 0$ such that is is NP-hard to distinguish satisfiable 3-SAT formulas from 3-SAT formulas where each assignment satisfies at most fraction $1 - \rho$ of the clauses.*

In other words, the theorem states that for any language L in NP there exists a reduction $f(x)$ which the following properties:

1. If $x \in L$ then $f(x)$ is satisfiable 3-SAT formula.
2. If $x \notin L$ then $f(x)$ is a 3-SAT formula for which each assignment satisfies at most $1 - \rho$ fraction of the clauses.

Theorem 8.4. *For any $\varepsilon > 0$ it is NP-hard to distinguish satisfiable 3-SAT formulas from 3-SAT formulas where each assignment satisfies at most fraction $\frac{7}{8} + \varepsilon$ of the clauses.*

Proof of theorem 8.1, given theorem 8.3. Let ρ be the parameter in theorem 8.3. We will prove theorem 8.1 for $\rho' = \rho/2$. Thus, assume A is a polynomial time algorithm which computes a $1 - \rho'$-approximation for 3-SAT, and let $L \in$ NP. We show how to decide L, i.e. we show that $L \in$ P and thus P=NP, as necessary to show theorem 8.1. On input x, run the reduction given by theorem 8.3, and let the resulting 3-SAT formula be Φ. Run the approximation algorithm A on Φ. If the produced assignment satisfies at least $1 - \rho'$ fraction of the clauses of Φ, output $x \in L$, otherwise output $x \notin L$. This is a polynomial time algorithm.

If $x \in L$, then this algorithm always outputs $x \in L$, because the 3-SAT formula produced by the reduction is satisfiable, and so the approximation algorithm must satisfy at least fraction $1 - \rho'$ of the clauses.

In case $x \notin L$, this algorithm always outputs $x \notin L$, because the 3-Sat formula produced by the reduction has the property that no assignment satisfies $1 - \rho = 1 - 2\rho'$ of the clauses. □

8.2 Probabilistically Checkable Proofs

8.2.1 NP as Problems with Checkable Proofs

Recall that NP is the class of problems possessing deterministically checkable proofs for yes-answers. More formally, we can define NP as a set of languages, such that: A language $L \subseteq \{0,1\}^*$ is in NP if there is a polynomial time computable verification procedure $V(x, w)$, i.e. a polynomial time algorithm that obtains inputs x and w and either accepts or rejects, such that

- For every $x \in L$ there is a "witness" w such that $V(x, w)$ accepts.
- For every $x \notin L$ and every w, $V(x, w)$ rejects.

The running time of V should be polynomial in $|x|$ and thus w should also be polynomial in $|x|$, as otherwise the algorithm is not able to read all of w.

8.2.2 Probabilistically Checkable Proofs

Theorem 8.5 (PCP theorem; proof-checking version)**.** *There exists $q_0 \in \mathbb{N}$ such that for every $L \in$ NP there exists a polynomial time verifier $V(x, w)$ with the following properties. The verifier expects a proof w of size polynomial in $|x|$ for the statement $x \in L$. It first reads x, tosses $\mathcal{O}(\log |x|)$ random coins, reads q_0 bits of w, then accepts or rejects.*

- *If $x \in L$, then there exists a w such that the verifier $V(x, w)$ accepts with probability 1.*
- *If $x \notin L$, then for any w the probability that $V(x, w)$ rejects is at least $\frac{1}{2}$.*

We can think of the theorem as follows. We have two characters, a *prover* and a *verifier*. The prover has unlimited computing power, and constructs a proof w which is transmitted to the verifier. The verifier does not read the whole proof, but rather looks at constantly many positions of the proof, which he determines using polynomial time and some random bits. Then he makes up his mind whether to accept or reject the proof.

The prover knows exactly what the verifier is going to do with the proof, except for the random bits of course. He can use this knowledge and his unlimited computing power, to try to convince the verifier that φ is satisfiable even when it is not.

Amazingly the theorem states, that this tricking the verifier into accepting a wrong proof is possible with probability at most $\frac{1}{2}$. Note that the constant $\frac{1}{2}$ is not important. Indeed, if the verifier repeats his checking of the proof k times using k times more random bits and looking at k times more positions of w, the probability that he is cheated is at most $\frac{1}{2^k}$.

Also note that the verifier can toss $\mathcal{O}(\log |x|)$ random coins, i.e. for any L there exists some constant c_L such that the verifier tosses $c_L \cdot \log |x|$ random coins. This means that it is possible to enumerate all choices of the random coins in $2^{c_L \log |x|} = |x|^{c_L}$ time, which is polynomial.

8.3 A Baby PCP Theorem

8.3.1 Circuit-SAT

We use the fundamental NP-complete problem called circuit-SAT where the circuit is restricted to use NAND-gates. A circuit C is a directed acyclic graph, with input nodes and internal nodes. The internal nodes are also called gates, and each gate has two predecessors (which can be the same). The circuit C with k input nodes and a single gate which has no successor (a sink) then computes a function $C : \{0,1\}^k \to \{0,1\}$ as follows: label each input node with the corresponding bit from the input. Then, label each internal node whose predecessors have been labeled with the NAND of the labels of the predecessors. The label of the sink will be the result of the function.

Circuit-SAT is the following problem: given a circuit with a single bit output, is there an input on which the circuit computes 1?

8.3.2 The Baby PCP Theorem

We first introduce the concept of *oracle verifiers* and *oracle Truing machines*. An oracle Turing machine has an additional, special tape on which it can write as well as a special state σ. In order to describe the behavior of an oracle Turing machine, a function $w : \{0,1\}^* \to \{0,1\}$ needs to be given additionally, which is called the oracle. Given such an oracle, the machine behaves as a usual Turing machine, except in case it enters state σ. If the machine enters this state, the contents q of the special tape are read and erased. Then, q is interpreted as input to a function w, and $w(q)$ is written onto the special tape.

We write $V^{(w)}(x)$ to denote a run of the oracle machine, when the function w is given as an oracle. Theorem 8.5 can then be reformulated as:

Theorem 8.6 (PCP theorem; proof-checking version with oracle machines)**.** *There exists $q_0 \in \mathbb{N}$, $\rho > 0$, and a polynomial time verifier $V^w(C)$ with the following properties. The verifier first reads the description of its input C, a NAND-circuit with $|C|$ gates. It then tosses $\mathcal{O}(\log |C|)$ random coins, queries w at most q_0 times, then accepts or rejects.*

- *If $C(y) = 1$ for some y, then there exists a $w : \{0,1\}^{\mathcal{O}(\log |C|)} \to \{0,1\}$ such that the verifier $V^w(C)$ accepts with probability 1.*
- *If $C(y) = 0$ for all y, then for any w the probability that $V^w(C)$ rejects is at least ρ.*

Because w takes only $\mathcal{O}(\log |C|)$ bits as input, the truth table of it has length polynomial in $|C|$. Also note that the fact that we only use circuit-SAT as language is no problem, since circuit-SAT is NP-complete.

We can replace $\mathcal{O}(\log |C|)$ by $|C|^2$ to obtain the baby PCP theorem; a much weaker, but still surprising result.

Theorem 8.7 (Baby PCP theorem)**.** *There exists $q_0 \in \mathbb{N}$, $\rho > 0$, and a polynomial time verifier $V^w(C)$ with the following properties. The verifier first reads the description of its input C, a NAND-circuit with $n = |C|$ gates. It then tosses some random coins, queries w at most q_0 times, then accepts or rejects.*

- *If $C(y) = 1$ for some y, then there exists a $w : \{0,1\}^{n^2} \to \{0,1\}$ such that the verifier $V^w(C)$ accepts with probability 1.*
- *If $C(y) = 0$ for all y, then for any w the probability that $V^w(C)$ rejects is at least ρ.*

8.3.3 Useful Tools

We now develop some useful tools needed for the proof of theorem 8.7.

8.3.3.1 Linear Functions

Definition 8.2. *A function $f : \{0,1\}^n \to \{0,1\}$ is* linear *if*

$$f(x) \oplus f(y) = f(x \oplus y)$$

where $x \oplus y$ is the component-wise XOR, i.e. $z = x \oplus y$ if and only if $z_i = x_i \oplus y_i$ for all coordinates i.

8.3.3.2 Linearity Test

Consider the following linearity test: pick x and y uniformly at random and check whether $f(x) \oplus f(y) = f(x \oplus y)$. If so, output *accept*, otherwise output *reject*. More formally, this test is shown as algorithm 10.

Algorithm 10 Linearity test LT^f

```
1: if f(x) ⊕ f(y) = f(x ⊕ y) then
2:     return accept
3: else
4:     return reject
5: end if
```

Per definition, whenever f is linear, the test accepts. However, for some non-linear functions, the test rejects only with very small probability. But as the next lemma states, if the test only catches non-linearity with very small probability, then indeed the function must be *almost* linear.

Lemma 8.9. *Let f be a function for which the above tests accepts with probability $1 - \delta$. Then, there is a linear function g with $\Pr_{x \in \{0,1\}^n}[f(x) = g(x)] \geq 1 - \delta$.*

8.3.3.3 Local Decoding

Suppose there is an almost linear function f and the linear function g from lemma 8.9, and you are given $x \in \{0,1\}^n$. If you want to guess $g(x)$, but can only query f two times, you might try to just output $f(x)$, as f and g coincide on most x. However, x might be chosen such that $f(x) \neq g(x)$, in which case this guess is not clever at all. The following lemma shows how to improve your success probability, and algorithm 11 implements the idea.

Lemma 8.10. *Let f be a function, g be linear, and $\Pr_{x \in \{0,1\}^n}[f(x) = g(x)] \geq 1 - \delta$. Then, for any fixed $x \in \{0,1\}^n$:*

$$\Pr_{y \in \{0,1\}^n}[g(x) = f(y) \oplus f(x \oplus y)] \geq 1 - 2\delta$$

Algorithm 11 Local decoding $\mathrm{LD}^w(x)$ for input $x \in \{0,1\}^m$ and oracle function $w : \{0,1\}^m \to \{0,1\}$

```
1: y ←_u.a.r. {0,1}^m
2: return w(y) ⊕ w(x ⊕ y)
```

8.3.3.4 Checking Whether a Matrix is Zero

Suppose we want to check whether a given vector is zero, and we are allowed to query any XOR of the positions. The following lemma says: if we pick a random XOR, for any non-zero vector we get a 1 with probability $\frac{1}{2}$, and similar for a matrix with probability $\frac{1}{4}$.

Lemma 8.11. *Let $s \in \{0,1\}^n$ be a non-zero vector. Pick a random subset $I \subseteq \{1,\dots,n\}$ such that every i is included in I with probability $\frac{1}{2}$, independent of all other coordinates. Then,*

$$\Pr_I\left[\bigoplus_{i\in I} s_i = 1\right] = \frac{1}{2}$$

Let $M \in \{0,1\}^{n\times n}$ be a non-zero matrix. Pick subsets $I \subseteq \{1,\dots,n\}$ and $J \subseteq \{1,\dots,n\}$ independently of each other as before. Then,

$$\Pr_{I,J}\left[\bigoplus_{i\in I}\bigoplus_{j\in J} M_{i,j} = 1\right] \geq \frac{1}{4}$$

Proof. Let s be a non-zero vector, and j be a position with $s_j = 1$. We first decide whether $i \in I$ for all $i \neq j$, and compute $b = \bigoplus_{i\in I, i\neq j} s_i$. If $b = 0$ then $\bigoplus_{i\in I} s_i = 1$ exactly if $j \in I$, which happens with probability $\frac{1}{2}$. If $b = 1$, the opposite happens.

For the second part suppose that a non-zero matrix M is given. Pick first I, and consider the vector with components $y_j = \bigoplus_{i\in I} M_{i,j}$. With probability at least $\frac{1}{2}$ this vector is non-zero, because of the first part. Then, applying the first part again, $\bigoplus_{j\in J} y_j$ is 1 with probability at least $\frac{1}{4}$. □

8.3.4 Proof of the Baby PCP Theorem

8.3.4.1 Description of the Oracle

We start by describing how w looks like in case there is an input to the circuit for which it outputs 1. We label all values which occur in a circuit from y_1 to y_n, such that the first k are the input wires and y_n is the output of the circuit. Now pick an input $(y_1,\dots,y_k)$ for which $C(y) = 1$, and extend y such that $y_i \in \{0,1\}$ for $i = 1,\dots,n$ is the value on wire i for this input.

Now, consider the following $n \times n$-matrix W: the entry $W_{i,j}$ is $y_i y_j$, i.e. the binary AND of y_i and y_j. The function w takes as input a binary $n \times n$-matrix M and outputs the XOR of the values $W_{i,j}$ for which $M_{i,j} = 1$:

$$w(M) = \bigoplus_{i=1}^{n}\bigoplus_{j=1}^{n} W_{i,j}M_{i,j} = \bigoplus_{i=1}^{n}\bigoplus_{j=1}^{n} y_i y_j M_{i,j}$$

The function w is linear.

8.3.4.2 Description of the Verifier

On an informal level, the verifier will do the following steps:

1. Check whether w is a linear function.
2. Assuming that $w(M) = \bigoplus_{i=1}^{n}\bigoplus_{j=1}^{n} W_{i,j}M_{i,j}$, check whether $W_{i,j} = y_i y_j$ for some vector y and all $i,j \in \{1,\dots,n\}$.
3. Assuming that $w(M) = \bigoplus_{i=1}^{n}\bigoplus_{j=1}^{n} W_{i,j}M_{i,j}$, check whether y is a vector which is consistent with all gates of the circuit.
4. Assuming that $w(M) = \bigoplus_{i=1}^{n}\bigoplus_{j=1}^{n} W_{i,j}M_{i,j}$, check whether $y_n = 1$.

For each of those tests the verifier uses a constant number of queries (3,6,2, and 2, respectively). The exact procedure is shown as algorithm 12 with input gates $1,\dots,k$, NAND gates $k+1,\dots,n$ (given by $y_i = y_{l(i)} \overline{\wedge} y_{r(i)}$) and output gate n.

The individual steps can be described as follows:

The linearity test (3 queries) In order to test if w is a linear function, the verifier uses the linearity test in lemma 8.9. Thus, it picks random matrices R_1 and R_2 and checks whether $w(R_1) \oplus w(R_2) \neq w(R_1 \oplus R_2)$.

Checking the form (6 queries) Next, the verifier would like to check if there is a vector y such that $W_{i,j} = y_i y_j$. If there is such a vector y then $y_i = W_{i,i}$, i.e. the diagonal contains the vector y. Therefore, the verifier can also check whether $W_{i,j} = W_{i,i}W_{j,j}$ for all i,j.

Algorithm 12 Circuit-SAT verifier $V^w_{\text{Circuit-SAT}}(C)$

// Linearity test
$R_1 \leftarrow_{u.a.r.} \{0,1\}^{n^2}$
$R_2 \leftarrow_{u.a.r.} \{0,1\}^{n^2}$
if $w(R_1) \oplus w(R_2) \neq w(R_1 \oplus R_2)$ **then**
  return reject
end if
// Check form
$a \leftarrow_{u.a.r.} \{0,1\}^n$
$b \leftarrow_{u.a.r.} \{0,1\}^n$
if $\left(\text{LD}^w(\text{diag}(a)) \wedge \text{LD}^w(\text{diag}(b))\right) \neq \text{LD}^w(a \cdot b^T)$ **then** // $\text{diag}(v)$ is the $n \times n$-matrix with v on its diagonal
  return reject
end if
// Check gates
$G \subseteq_R \{k+1, \ldots, n\}$
$M := \bigoplus_{i \in G}(E_{i,i} \oplus E_{l(i),r(i)})$ // $E_{i,j}$ is a $n \times n$-matrix with zeros everywhere except at position (i,j)
if $\text{LD}^w(M) \neq |G|$ is odd **then**
  return reject
end if
// Check output
if $\text{LD}^w(E_{n,n}) \neq 1$ **then**
  return reject
end if

For the check, the verifier picks two random vectors a and b from $\{0,1\}^n$. He then computes the matrix $\text{diag}(a)$ which contains the vector a on the diagonal and 0 in all other entries. Also, he computes $\text{diag}(b)$ and the matrix $a \cdot b^T$ which contains a 1 in position (i,j) if $a_i = b_j = 1$. If w would be linear, he would then check whether $w(\text{diag}(a)) \wedge w(\text{diag}(b)) = w(\text{diag}(a \cdot b^T)$. However, at this point we will only know that w is close to linear, and so we use lemma 8.10 for this check.

Checking the gates (2 queries) The verifier wants to check whether $W_{i,i} = \neg W_{l(i),r(i)}$ for each NAND-gate $y_i = y_{l(i)} \overline{\wedge} y_{r(i)}$. For this, he picks a random subset G of the gates and takes the XOR of all equations $W_{i,i} \oplus W_{l(i),r(i)} = 1$ for $i \in G$, and checks it again using 8.10.

Checking the ouput (2 queries) Next, the verifier wants to check that $W_{n,n} = 1$. He thus queries w to see whether this holds.

It is now rather easy to show that correct proofs are never rejected, and unprovable statements are rejected.

8.4 Fourier Transforms and the Linearity Test

It remains to prove lemma 8.9, for which we use the concept of Fourier transforms. In the theory of Fourier transforms it is convenient to replace the domain $\{0,1\}$ by $\{-1,1\}$ where -1 denotes true, and so the multiplication denotes the XOR.

8.4.1 Linear Functions

With the new notation, a function $f : \{-1,1\}^n \to \{-1,1\}$ is linear if $f(x)f(y) = f(x \cdot y)$ for all $x, y \in \{-1,1\}^n$, where $x \cdot y$ denotes the point-wise product of two vectors: $(x \cdot y)_i = x_i y_i$.

Lemma 8.12. *A function $f : \{-1,1\}^n \to \{-1,1\}$ is linear if and only if there is a set $S \subseteq \{1, \ldots, n\}$ for which $f(x) = \prod_{i \in S} x_i$.*

8.4.2 The Characters

The linear functions are called *characters*, and are denoted by $\mathcal{X}_S$ for subsets $S \subseteq \{1, \ldots, n\}$. For $n = 2$, all the characters are given in the following table:

	$\mathcal{X}_\emptyset$	$\mathcal{X}_{\{1\}}$	$\mathcal{X}_{\{2\}}$	$\mathcal{X}_{\{1,2\}}$
$(1,1)$	1	1	1	1
$(1,-1)$	1	1	-1	-1
$(-1,1)$	1	-1	1	-1
$(-1,-1)$	1	-1	-1	1

We can interpret the rows and columns as vectors in $\mathbb{R}^4$, in which case they are orthogonal. The next lemma states that his holds for all n. The symbol $\delta_{x,y}$ is used as Kronecker-delta, which is 1 if $x = y$ and 0 otherwise.

Lemma 8.13. *The characters are orthogonal:*

$$\sum_{x\in\{-1,1\}^n} \mathcal{X}_U(x)\mathcal{X}_S(x) = 2^n\delta_{U,S} = \begin{cases} 2^n & \text{if } U = S \\ 0 & \text{otherwise} \end{cases}$$

Analogously,

$$\sum_{S\subseteq\{1,\dots,n\}} \mathcal{X}_S(x)\mathcal{X}_S(y) = 2^n\delta_{x,y} = \begin{cases} 2^n & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

8.4.3 The Fourier Transform

We can now define the Fourier coefficients: for a function $f : \{-1,1\}^n \to \{-1,1\}$ and a set $S \subseteq \{1,\dots,n\}$ we define the S-Fourier coefficient of f as:

$$\hat{f}(S) = \frac{1}{2^n} \sum_{x\in\{-1,1\}^n} f(x)\mathcal{X}_S(x)$$

Up to some shift, the S-Fourier coefficient describe $\Pr[f(x) = \mathcal{X}_S(x)]$ as we will see next. The expression $\frac{1}{2}(1 + \mathcal{X}_S(x)f(x))$ is 1 if $\mathcal{X}_S(x) = f(x)$ and 0 otherwise. Thus

$$\Pr_{x\in\{-1,1\}^n}[f(x) = \mathcal{X}_S(x)] = \frac{1}{2^n} \sum_{x\in\{-1,1\}^n} \frac{1}{2}(1 + \mathcal{X}_S(x)f(x)) = \frac{1 + \hat{f}(S)}{2}$$

Knowing the Fourier coefficients of a function gives complete knowledge of the function. In fact, the next lemma gives a formula to extract the function from its Fourier coefficients, and it also contains Parseval's theorem: the sum of squares of the Fourier coefficients is 1.

Lemma 8.14. *Any function $f : \{-1,1\}^n \to \{-1,1\}$ is described by its Fourier coefficients in the following way*

$$f(x) = \sum_{S\subseteq\{1,\dots,n\}} \hat{f}(S)\mathcal{X}_S(x)$$

Also, we have

$$\sum_{S\subseteq\{1,\dots,n\}} \hat{f}^2(S) = 1$$

These facts now allow the proof of lemma 8.9 using rather simple calculations.