Chapter 2

Preliminaries

2.1 Graphs, digraphs, and trees

Let G be an undirected graph. The vertex set and edge set of G are denoted by $V(G)$ and $E(G)$, respectively. The *neighborhood* of a vertex v in G, denoted $N_G(v)$, is the set of vertices in G adjacent to v; $d_G(v) = |N_G(v)|$ is the *degree* of v in G. Vertices of degree 0 are called *isolated vertices*. For $U \subseteq V$, let $N_G(U) = \cup_{u \in U} N_G(u)$, and $G[U]$ be the graph (U, F), where $F = \{\{u, v\} \mid u, v \in U \text{ and } \{u, v\} \in E\}$. A subset M of E is a *matching* in G if no two edges in M have a common endpoint. A matching is *maximal* if it is not properly included in any other matching. We use $V(M)$ to denote the set of all vertices v such that v is an endpoint of some edge in a matching M. By a *path*, we always mean a simple path. Note that a single vertex is considered as a path (of length 0) with one endpoint. The *girth* of a graph G is the length of a shortest cycle in G. A set F of edges in G is called a *path set* if $(V(G), F)$ is a forest in which each connected component is a path. Intuitively speaking, if F is a path set, then $(V(G), F)$ is a collection of vertex-disjoint paths. A *maximal path set* (MPS) in G is a path set that is not properly contained in another path set. The MPS problem is to find, given G, an MPS in G. A subset U of V is called a *linear forest set* (LFS) if $G[U]$ is a forest in which each connected component is a path. Intuitively speaking, if U is an LFS, then $G[U]$ is a collection of vertex-disjoint paths. A *maximal linear forest set* (MLFS) in G is an LFS that is not properly contained in another LFS. The *maximal linear forest* (MLF) problem is to find, given a graph G, an MLFS in G. Throughout this dissertation, unless stated otherwise, G always denotes the input (undirected) graph, Δ denotes the maximum degree of G, and n and m denote the numbers of vertices and edges in G, respectively.

Now we turn to the definitions of *digraphs*. Let D be a digraph. The vertex set and arc set of D are denoted by $V(D)$ and $A(D)$, respectively. The *tail* and *head* of an arc (u, v) are u and v, respectively. For a subset M of $A(D)$, we use $V(M)$ to denote the set of all vertices v such that v is the tail or head of some arc in M. The *underlying graph* of D is the undirected graph $(V(D), E)$, where E consists of those edges $\{u, v\}$ with $(u, v) \in A(D)$ or $(v, u) \in A(D)$. The *indegree* (or *outdegree*) of a vertex u in D is the number of arcs with head (respectively, tail) u in D and is denoted by $d_D^-(u)$ (respectively, $d_D^+(u)$). The *total degree* of a vertex u is $d_D^-(u) + d_D^+(u)$ and is denoted by $d_D(u)$. Hereafter, a path in D always means a simple directed path. A set B of arcs in D is called a *directed path set* (DPS) if $(V(D), B)$ is an acyclic digraph in which the indegree and outdegree of each vertex are both at most 1. A *maximal directed path set* (MDPS) in D is a DPS that is not properly contained in another DPS. The MDPS problem is to find, given D, an MDPS in G. Throughout this subsection, D always denotes the input digraph, and n and m denote the numbers of vertices and arcs in D, respectively.

A *tree* is an acyclic connected graph. A *rooted tree* is a tree in which we identify a distinguished vertex r which we call *root*. We define the *depth* of a vertex; vertices at distance i from the root have depth i; the root itself has depth 0. A *leaf* of a tree is a vertex of the tree of degree 1. An *internal node* of a tree is a vertex being not a leaf. We call a digraph a *rooted directed tree* if its underlying graph is a tree, and there are directed paths from its root to every other vertex. Let (u, v) is an arc in a directed tree such that the depth of u is less than the depth of v. Then, u is called the *parent* of v, and v is called a *son* of v. If there is a directed path from a vertex u to a vertex v in a directed tree, then u is called an *ancestor* of v, and then v is called a *descendant* of u. The subgraph of a directed tree induced by a vertex v and all its descendants is called the *subtree rooted at* v. An *ordered tree* is a directed tree in which the set of sons of each vertex is ordered. A *binary tree* is an ordered tree in which no vertex has more than two sons. One of the sons is called the *left son*, while the other is called the *right son*. The subtree rooted at the left and right sons of v is called the *left* and *right subtrees*, respectively.

2.2 Models of computation

We assume a standard Turing machine model (see, e.g., [HU79] for further details). A *Turing machine* (TM) is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

where

Q is the finite set of *states*,
Γ is the finite set of allowable *tape symbols*,
B ($B \in \Sigma$) is the *blank*,
Σ ($\Sigma \subseteq \Gamma - \{B\}$) is the set of *input symbols*,
δ is the *transition function*, a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$,
q_0 in Q is the *start state*,
F ($F \subseteq Q$) is the set of *final states*.

We denote an *instantaneous description* (ID) of the Turing machine M by $\alpha_1 q \alpha_2$, where q is the current state of M (in Q), and $\alpha_1\alpha_2$ is the string in Γ^* that is the contents of the tape. We define a *move* of M as follows. Let $X_1X_2 \cdots X_{i-1}qX_i \cdots X_n$ be an ID. Suppose $\delta(q, X_i) = (p, Y, L)$. Then we write

$$X_1X_2 \cdots X_{i-1}qX_i \cdots X_n \vdash_M X_1X_2 \cdots X_{i-2}pX_{i-1}YX_{i+1} \cdots X_n.$$

Alternatively, suppose $\delta(q, X_i) = (p, Y, R)$. Then we write

$$X_1X_2 \cdots X_{i-1}qX_i \cdots X_n \vdash_M X_1X_2 \cdots X_{i-2}X_{i-1}YpX_{i+1} \cdots X_n.$$

If two ID's are related by $\vdash_M$, we say that the second results from the first by one move. If one ID results from another by some finite number of moves, they are related by the symbol $\vdash^*_M$. We drop the subscript M when no confusion results.

We sometimes call a TM defined above a *deterministic Turing machine* (DTM) in comparison with an nondeterministic Turing machine defined later.

Definition 2.1 A DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *accepts* an input x in Σ^* if $q_0x \vdash^* \alpha_1 p \alpha_2$ for some $p \in F$, and $\alpha_1, \alpha_2 \in \Gamma^*$. The language accepted by M is the set of strings in Σ^* accepted by M, and denoted by $L(M)$.

A *nondeterministic Turing machine* (NTM) is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

where each element is as in the deterministic case, with the exception that the transition function is defined by

$$\delta : Q \times \Gamma \to \{Q \times \Gamma \times \{L, R\}\}^2.$$

Remark that we standardize, without loss of generality, the NTM that has exactly two choices (called *0-choice* and *1-choice*, respectively) per step.

The *instantaneous description* (ID) of the NTM M is defined as the same as in the deterministic case. The *move* of the NTM M is defined as follows. Let $X_1X_2 \cdots X_{i-1}qX_i \cdots X_n$ be an ID. When $(p, Y, L) \in \delta(q, X_i)$, we write

$$X_1X_2 \cdots X_{i-1}qX_i \cdots X_n \vdash_M X_1X_2 \cdots X_{i-2}pX_{i-1}YX_{i+1} \cdots X_n.$$

Alternatively, when $(p, Y, R) \in \delta(q, X_i)$, we write

$$X_1X_2 \cdots X_{i-1}qX_i \cdots X_n \vdash_M X_1X_2 \cdots X_{i-2}X_{i-1}YpX_{i+1} \cdots X_n.$$

The notation $\vdash_M^*$ is defined as the same way as in the deterministic case.

Definition 2.2 An NTM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ *accepts* an input x in Σ^* if $q_0x \vdash^* \alpha_1p\alpha_2$ for some $p \in F$, and $\alpha_1, \alpha_2 \in \Gamma^*$. The language accepted by M is the set of strings in Σ^* accepted by M, and denoted by $L(M)$.

We sometimes call states of an NTM *nondeterministic states*.

Definition 2.3 The class of the languages recognized by a DTM in a polynomial time of the input length is denoted by P. On the other hand, the class of languages recognized by an NTM in a polynomial time of the input length is denoted by NP. We denote by PSPACE the class of the languages recognized by a DTM in a polynomial space of the input length.

Remark that the class of the languages recognized by an NTM in a polynomial space is equal to PSPACE. However, it is very famous open problem to determine whether $\mathsf{P} = \mathsf{NP}$ or not (see, e.g., [HU79, BDG88, Joh90] for further details).

A *probabilistic Turing machine* (PTM) is a Turing machine with distinguished states called probabilistic states defined by Gill [Gil77]. It is defined as follows; a PTM is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

where each element is as in the nondeterministic case. The *instantaneous description* (ID) and the *move* of the PTM M are defined as the same as in the nondeterministic case. Without loss of generality, we assume that every move ends to the states in F, and each state in F is either *accepting state* or *rejecting state*.

Definition 2.4 Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a PTM. The *probability function* p_M, a partial function from ID to $[0, 1]$, is defined as follows: $p_M(X) = 1$ if X contains an accepting state; $p_M(X) = 0$ if X contains a rejecting state; and $p_M(X) = \frac{1}{2}(p_M(Y) + p_M(Z))$ where X, Y, and Z are IDs of M such that $X \vdash_M Y$ and $X \vdash_M Z$. We say that M *accepts* x *with probability* $p_M(q_0x)$.

We sometimes call states of a PTM *probabilistic states*.

Definition 2.5 The classes of the languages recognized by a PTM in a polynomial time of the input length are defined as follows:

A language L is in RP if there exists a PTM M such that (1) for $x \in L$, M accepts x with probability greater than $\frac{1}{2}$, and (2) for $x \notin L$, M always rejects.

A language L is in BPP if there exists a PTM M such that (1) for $x \in L$, M accepts x with probability greater than $\frac{3}{4}$, and (2) for $x \notin L$, M accepts x with probability less than $\frac{1}{4}$.

A language L is in PP if there exists a PTM M such that (1) for $x \in L$, M accepts x with probability greater than $\frac{1}{2}$, and (2) for $x \notin L$, M accepts x with probability less than $\frac{1}{2}$.

A *stochastic Turing machine* (STM) M is a Turing machine that has nondeterministic states and probabilistic states. It is denoted as follows; an STM is denoted

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F),$$

where each element is as in the nondeterministic case. The *instantaneous description* (ID) and the *move* of the STM M are defined as the same as in the nondeterministic case. We assume that every state in Q is either *probabilistic state* or *nondeterministic state*. Without loss of generality, we assume that every move ends to the states in F, and each state in F is either *accepting state* or *rejecting state*. Moreover, without loss of generality, we assume that an STM starts from a nondeterministic state, alternates between nondeterministic states and probabilistic states, and ends in a probabilistic state.

Definition 2.6 Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be an STM. The *probability function* p_M, a partial function from ID to $[0, 1]$, is defined as follows: $p_M(X) = 1$ if X contains an accepting state; $p_M(X) = 0$ if X contains a rejecting state; $p_M(X) = \max(p_M(Y), p_M(Z))$ where X, Y, and Z are IDs of M such that $X \vdash_M Y$, $X \vdash_M Z$, and X contains a nondeterministic state; and $p_M(X) = \frac{1}{2}(p_M(Y) + p_M(Z))$ where X, Y, and Z are IDs of M such that $X \vdash_M Y$, $X \vdash_M Z$, and X contains a probabilistic state. We say that M *accepts* x *with probability* $p_M(q_0x)$.

The classes ABPP and APP are defined by Papadimitriou [Pap94]:

Definition 2.7 The classes of the languages recognized by an STM in a polynomial time of the input length are defined as follows:

A language L is in ABPP if there exists an STM M such that (1) for $x \in L$, M accepts x with probability greater than $\frac{3}{4}$, and (2) for $x \notin L$, M accepts x with probability less than $\frac{1}{4}$.

A language L is in APP if there exists an STM M such that (1) for $x \in L$, M accepts x with probability greater than $\frac{1}{2}$, and (2) for $x \notin L$, M accepts x with probability less than $\frac{1}{2}$.

Similarly, we define the new class ARP as follows:

Definition 2.8 A language L in ARP is accepted by an STM M in a polynomial time of the input length such that (1) for $x \in L$, M accepts x with probability greater than $\frac{1}{2}$, and (2) for $x \notin L$, M always rejects.

For APP and ABPP, the following theorem is well known:

Theorem 2.9 ABPP = APP = PSPACE.

Outline of proof. Babai introduced "Arthur-Merlin games", and defined the class AM(*poly*) using the games [Bab85]. An Arthur-Merlin game directly corresponds to the computation of an STM; an Arthur's (resp. Merlin's) turn corresponds to a probabilistic (resp. nondeterministic) state. Thus it is easy to see that ABPP = AM(*poly*). On the other hand, Goldwasser, Micali, and Rackoff introduced interactive proof systems and defined the class IP using the systems [GMR85], and Goldwasser and Sipser showed that IP = AM(*poly*)[GS86]. Moreover, Shamir showed that PSPACE = IP [Sha90]. Thus ABPP = AM(*poly*) = IP = PSPACE. It is easy to see that ABPP $\subseteq$ APP $\subseteq$ PSPACE. ∎

Now we turn to the parallel computation model. In this dissertation, we use the *parallel random-access machine* (PRAM) model (see e.g. [KR90]). A PRAM consists of several independent sequential processors, each with its own private memory, communicating with one another through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM operation, and write into one global or local memory location. An *Exclusive-Read Exclusive-Write* (EREW) PRAM is the parallel model where the processors operate synchronously and share a common memory, but no two of them are allowed simultaneous access to a memory cell (whether the access is for reading or for writing in that cell). A *Concurrent-Read Concurrent-Write* (CRCW) PRAM differs from the EREW PRAM in that both simultaneous reading and simultaneous writing to the same cell are allowed; in case of simultaneous writing, the processor with lowest index succeeds.

Definition 2.10 The class of the languages recognized by a PRAM that uses at most $\log^{O(1)} n$ time and $n^{O(1)}$ processors on inputs of size n is denoted by NC. A language L is in RNC if and only if there is a polynomial p and a language L' in NC such that for all input strings x the following properties hold; (1) if $x \in L$, then for over half of the possible strings y with $|y| = p(|x|)$, the string xy is in L'; and (2) if $x \notin L$, then for all of the possible strings y with $|y| = p(|x|)$, the string xy is not in L'.

We remark that RNC could be defined in terms of probabilistic PRAMs with certain probabilities of obtaining the correct answer (see e.g. [Joh90]).

As the input representation of a graph G, we assume that $V(G) = \{0, 1, \cdots, n-1\}$ and that each vertex has a list of the edges incident to it. Thus, each edge $\{i, j\}$ has two copies — one in the edge list for vertex i and the other in the edge list for vertex j. We assume that each of the two copies has a pointer to the other. The input representation of a digraph D is the same as that of an undirected graph except that each vertex i now has *two* lists; one of the lists consists of all arcs with tail i and the other consists of all arcs with head i.

2.3 Model of an efficient computation

We define an *efficient algorithm* $\mathcal{A}_p$, representing an efficient simulation by a PTM with $\mathcal{C}_p$, by a finite rooted binary directed tree such that;

(i) each internal node, representing a coin-tossing, has two arcs to its sons,
(ii) one of the two arcs above is labeled p, and the other is labeled $1-p$, and
(iii) each leaf is labeled "$\mathcal{H}$" or "$\mathcal{T}$".

The *depth of a leaf* (or the *depth of an internal node*) is the number of the arcs of the directed path from the root to the leaf (or the internal node, respectively). The *depth of an efficient algorithm* is the maximal depth of its leaves. The *probability of a leaf* is defined by the product of all labels on the path from the leaf to the root. An efficient algorithm $\mathcal{A}_p$ *simulates* $\mathcal{C}_q$, if the sum of all probabilities of the leaves of $\mathcal{A}_p$ labeled "$\mathcal{H}$" is equal to q. We write $\mathcal{C}_p \to \mathcal{C}_q$ if there

is an efficient algorithm $\mathcal{A}_p$ simulating $\mathcal{C}_q$, and by $\mathcal{C}_p \not\to \mathcal{C}_q$ otherwise. An efficient algorithm is called a *complete efficient algorithm* if the depths of all leaves are equal to each other. Note that for each efficient algorithm of depth k, there exists a complete efficient algorithm of depth k, simulating the same coin. Let $\mathcal{S}_p$ be the set of biases efficiently simulated by $\mathcal{A}_p$.

To deal with a rational number as a bias, we introduce the notation on the a-ary system. The rational numbers consist of the finite decimals and the recurring decimals. For a finite decimal p on the a-ary system, we denote p by $(0.p_1p_2\cdots p_k)_a$, where $k > 0$, $0 < i \leq k$, $0 \leq p_i < a$ and $p_k \neq 0$, and $p = p_1a^{-1} + p_2a^{-2} + \cdots + p_ia^{-i} + \cdots + p_ka^{-k}$. The number of figures of p on the a-ary system is denoted by $|p|_{(a)}$ (equal to k, in the case above). For a recurring decimal q on the a-ary system, $|q|_{(a)}$ is defined as an infinity. Note that for a rational number p, it depends on a whether $|p|_{(a)}$ is finite or not (e.g. $\frac{1}{2} = (0.1)_2 = (0.11\cdots)_3$).

2.4 Complexity classes with a semi-random source

A *semi-random source* with parameter δ, or simply a *δ-random source*, is defined as follows.

Definition 2.11 [SV84] Let δ be a number such that $0 \leq \delta \leq \frac{1}{2}$. A *semi-random source* with parameter δ outputs bits $X_1X_2\cdots$, such that for all i and for all $x_1, x_2, \cdots$,

$$\delta \leq \Pr[X_i = x_i \mid X_1 = x_1, \cdots, X_{i-1} = x_{i-1}] \leq 1 - \delta.$$

Definition 2.12 [VV85] A language L is in δ-RP if there exists a PTM M such that (1) for $x \in L$, M accepts with probability greater than $\frac{1}{2}$ for every δ-random source, and (2) for $x \notin L$, M always rejects.

Definition 2.13 [Vaz86] A language L is in δ-BPP if there exists a PTM M such that (1) for $x \in L$, M accepts with probability greater than $\frac{3}{4}$ and (2) for $x \notin L$, M accepts with probability less than $\frac{1}{4}$ for every δ-random source.

We note that the class δ-RP is denoted by SR_p in [VV85], and the class δ-BPP is denoted by $SBPP$ in [Vaz86]. To unify the notations, we use δ-RP and δ-BPP following from [Pap94]. Here, we introduce new classes δ-PP, $^{\exists}\delta$-RP, $^{\exists}\delta$-BPP, and $^{\exists}\delta$-PP:

Definition 2.14 A language L is in δ-PP if there exists a PTM M such that (1) for $x \in L$, M accepts with probability greater than $\frac{1}{2}$ and (2) for $x \notin L$, M accepts with probability less than $\frac{1}{2}$ for every δ-random source.

Definition 2.15 A language L is in $^{\exists}\delta$-RP if there exists a PTM M such that (1) for $x \in L$, M accepts with probability greater than $\frac{1}{2}$ for *at least one* δ-random source and (2) for $x \notin L$, M always rejects.

Definition 2.16 A language L is in $^{\exists}\delta$-BPP if there exists a PTM M such that (1) for $x \in L$, M accepts with probability greater than $\frac{3}{4}$ for *at least one* δ-random source and (2) for $x \notin L$, M accepts with probability less than $\frac{1}{4}$ for *every* δ-random source.

Definition 2.17 A language L is in ${}^{\exists}\delta$-PP if there exists a PTM M such that (1) for $x \in L$, M accepts with probability greater than $\frac{1}{2}$ for *at least one* δ-random source and (2) for $x \notin L$, M accepts with probability less than $\frac{1}{2}$ for *every* δ-random source.

The following proposition follows from the definitions immediately:

Proposition 2.18 For every δ with $0 < \delta < \frac{1}{2}$, the following hold.

$$\begin{aligned}
\mathsf{P} &= 0\text{-}\mathsf{RP} \subseteq \delta\text{-}\mathsf{RP} \subseteq \frac{1}{2}\text{-}\mathsf{RP} = \mathsf{RP} = {}^{\exists}\frac{1}{2}\text{-}\mathsf{RP} \subseteq {}^{\exists}\delta\text{-}\mathsf{RP} \subseteq {}^{\exists}0\text{-}\mathsf{RP} = \mathsf{NP}, \\
\mathsf{P} &= 0\text{-}\mathsf{BPP} \subseteq \delta\text{-}\mathsf{BPP} \subseteq \frac{1}{2}\text{-}\mathsf{BPP} = \mathsf{BPP} = {}^{\exists}\frac{1}{2}\text{-}\mathsf{BPP},\ {}^{\exists}0\text{-}\mathsf{BPP} = \mathsf{NP}, \\
\mathsf{P} &= 0\text{-}\mathsf{PP} \subseteq \delta\text{-}\mathsf{PP} \subseteq \frac{1}{2}\text{-}\mathsf{PP} = \mathsf{PP} = {}^{\exists}\frac{1}{2}\text{-}\mathsf{PP}, \text{ and } {}^{\exists}0\text{-}\mathsf{PP} = \mathsf{NP}.
\end{aligned}$$

Proof. Since a $\frac{1}{2}$-random source is a perfect random source, it is easy to see that $\mathsf{RP} = \frac{1}{2}\text{-}\mathsf{RP} = {}^{\exists}\frac{1}{2}\text{-}\mathsf{RP}$, $\mathsf{BPP} = \frac{1}{2}\text{-}\mathsf{BPP} = {}^{\exists}\frac{1}{2}\text{-}\mathsf{BPP}$, and $\mathsf{PP} = \frac{1}{2}\text{-}\mathsf{PP} = {}^{\exists}\frac{1}{2}\text{-}\mathsf{PP}$. For δ and δ' with $0 \leq \delta \leq \delta' \leq \frac{1}{2}$, a δ'-random source is also a δ-random source. A 0-random source generates 1 with probability between 0 and 1. Thus for 0-RP, 0-BPP and 0-PP, since every computation for each input must produce same answer, the algorithm must be deterministic. This implies $0\text{-}\mathsf{RP} = 0\text{-}\mathsf{BPP} = 0\text{-}\mathsf{PP} = \mathsf{P}$. Conversely, for ${}^{\exists}0\text{-}\mathsf{RP}$, ${}^{\exists}0\text{-}\mathsf{BPP}$ and ${}^{\exists}0\text{-}\mathsf{PP}$, since it is sufficient to accept that at least one computation reaches accepting state, the algorithm is nondeterministic. This implies ${}^{\exists}0\text{-}\mathsf{RP} = {}^{\exists}0\text{-}\mathsf{BPP} = {}^{\exists}0\text{-}\mathsf{PP} = \mathsf{NP}$. ∎

Note that for $\delta < \delta'$, ${}^{\exists}\delta'\text{-}\mathsf{BPP} \subseteq {}^{\exists}\delta\text{-}\mathsf{BPP}$ and ${}^{\exists}\delta'\text{-}\mathsf{PP} \subseteq {}^{\exists}\delta\text{-}\mathsf{PP}$ do *not* hold from its definition immediately, whereas ${}^{\exists}\delta'\text{-}\mathsf{RP} \subseteq {}^{\exists}\delta\text{-}\mathsf{RP}$ holds. To see this, let M be a PTM such that $L(M) \in {}^{\exists}\delta'\text{-}\mathsf{BPP}$ and δ and δ' be two numbers with $0 \leq \delta < \delta' \leq \frac{1}{2}$. It likely holds that $L(M) \in {}^{\exists}\delta\text{-}\mathsf{BPP}$ since the input x with $x \in L(M)$ satisfies the condition of the definition of ${}^{\exists}\delta\text{-}\mathsf{BPP}$ since a δ'-random source is also a δ-random source. However, for $x \notin L$, M must accepts with probability less than $\frac{1}{4}$ for *every* δ-random source. Since the set of δ-random sources properly contains the set of δ'-random sources when $\delta < \delta'$, M may not satisfy the condition.

For $0 < \delta \leq \frac{1}{2}$, the following results are known.

Theorem 2.19 [VV85] $\delta\text{-}\mathsf{RP} = \mathsf{RP}$.

Theorem 2.20 [Vaz86] $\delta\text{-}\mathsf{BPP} = \mathsf{BPP}$.

2.5 Computation trees with a semi-random source

The probability that a PTM with a δ-random source accepts, depends not only on the number of the accepting leaves, but also on the distribution of the accepting leaves in the computation tree of the PTM. To consider the distribution, we need several definitions and propositions.

A *δ-tree* is a rooted full binary directed tree whose every leaf is labeled "accept" or "reject". The leaves labeled "accept" and "reject" are respectively called *accepting* and *rejecting leaves*. For a δ-tree T, we denote by $|T|$ the number of accepting leaves of T.

Definition 2.21 A *δ-assignment* F to a δ-tree is a mapping from the set of arcs of the δ-tree to the interval $[\delta, 1-\delta]$ such that the two arcs leaving each internal node are assigned values whose sum is 1.

Definition 2.22 Let T be a δ-tree, and F be a δ-assignment to T. The *probability of a node* of T under F is the product of the values assigned to the arcs on the path from the root to the node by F. The *probability of* T under F, denoted by $\Pr[T|F]$, is the sum of the probabilities of the accepting leaves.

Definition 2.23 For a given δ-tree T and an assignment F to T, F is *a maximum assignment* if $\Pr[T|F] \geq \Pr[T|F']$ for every δ-assignment F' to T.

Proposition 2.24 For every δ-tree T, there exists a maximum assignment F to T such that F only assigns δ or $1-\delta$ to each arc of T.

Proof. For a given δ-tree T, we recursively define an assignment F as follows: First, let v be a node whose sons are leaves. If at least one of its sons is accepting leaf, assign $(1-\delta)$ to the arc between v and the son, and assign δ to the other arc. Otherwise, assign $(1-\delta)$ and δ to the two arcs between v and its two sons in any way. Second, let u be a node whose sons are the roots of two subtrees, say T_0 and T_1, whose all arcs are already assigned. If $\Pr[T_0|F] > \Pr[T_1|F]$, assign $(1-\delta)$ to the arc between u and the root of T_0 and assign δ to the arc between u and the root of T_1. Otherwise, assign $(1-\delta)$ to the arc between u and the root of T_1 and assign δ to the arc between u and the root of T_0. Using a simple induction on the depth of the tree, we find that $\Pr[T|F] \geq \Pr[T|F']$ for every δ-assignment F' to T. ∎

By the proposition, it is sufficient to consider δ-assignments that only assign the values δ and $1-\delta$ to maximize the probability. For a given δ-tree T, we denote by $F_{\max}(T)$ the maximum assignment defined in the proof of the proposition.

Definition 2.25 A δ-tree T_w is *worst* if $\Pr[T_w|F_{\max}(T_w)] \leq \Pr[T|F_{\max}(T)]$ for every δ-tree T with $|T_w| = |T|$. A δ-tree T_b is *best* if $\Pr[T_b|F_{\max}(T_b)] \geq \Pr[T|F_{\max}(T)]$ for every δ-tree T with $|T_b| = |T|$.

Proposition 2.26 Let T_w be a worst tree and T_b be a best tree such that $\Pr[T_w|F_{\max}(T_w)] = \Pr[T_b|F_{\max}(T_b)]$. Then, for every δ-tree T such that $\Pr[T_w|F_{\max}(T_w)] = \Pr[T|F_{\max}(T)]$, $|T_w| \geq |T| \geq |T_b|$.

Proof. There exists a worst tree T' such that $|T'| = |T|$ and $\Pr[T'|F_{\max}(T')] \leq \Pr[T|F_{\max}(T)] = \Pr[T_w|F_{\max}(T_w)]$. Thus, $|T_w| \geq |T'| = |T|$. A symmetric argument works for the best tree. ∎

Proposition 2.27 Let T_b be a best tree of depth d with a δ-random source. Then, the probability of the best tree under a maximum assignment is: $\Pr[T_b|F_{\max}(T_b)] = \sum_{i=0}^{t_0} \binom{d}{i} \delta^i (1-\delta)^{d-i} + t_1 \delta^{t_0+1}(1-\delta)^{d-t_0-1}$, where $|T_b| = \sum_{i=0}^{t_0} \binom{d}{i} + t_1$ with $t_1 \leq \binom{d}{t_0+1}$.

Proof. For each i with $0 \leq i \leq d$, a δ-tree of depth d has $\binom{d}{i}$ leaves corresponding to the probability $\delta^i(1-\delta)^{d-i}$, and $\delta^i(1-\delta)^{d-i} < \delta^{i-1}(1-\delta)^{d-i+1}$ since $\delta < (1-\delta)$. Thus, taking $|T_b|$ leaves which correspond to the largest $|T_b|$ probabilities, and labeling them "accept", we have a best tree. This implies the proposition. ∎

The structure of a best tree is characterized completely by Proposition 2.27. Hereafter, we will only consider the structure of a worst tree. From now on, we explicitly distinguish the two sons of each internal node in a δ-tree as the left and right sons. We denote by T_l and T_r the left and right subtrees of T, respectively. We similarly denote by T_{lr} the right subtree of T_l, and so on. We call the root of T the *parent* of T_l and T_r.

Definition 2.28 Let T be a δ-tree of depth $d \geq 0$. T is a *clustered tree* if
(1) $d = 0$,
(2) $|T_l| = 0$, and T_r is a clustered tree, or
(3) T_l is a clustered tree, and $|T_r| = 2^{d-1}$.

Intuitively, a clustered tree is a tree whose rightmost leaves are all accepting leaves. We fix the assignment defined as follows; for each internal node, assign $(1 - \delta)$ to every arc incident to a right son, and δ to every arc incident to a left son. Simple induction shows that the fixed assignment is the maximum assignment to the clustered trees.

2.6 Complexity classes based on one Boolean function

A *query* is an input $a \in \{0,1\}^n$. In the following we sometimes identify a query a with the query set $A = \{i \mid a_i = 1\}$. A query set leading to the answer 1 is called a *winner*. For each i with $1 \leq i \leq n$, we denote the query set $\{i\}$ of cardinality 1 by E_i.

The classes OR and $\mathsf{OR}(k)$ are based on the disjunction of n variables. The nonessential variables are replaced by the constant 0. For a set $S \subseteq \{1,\ldots,n\}$, we denote the disjunction of all x_i with $i \in S$ by OR_S. Then, the class $\mathsf{OR}(k)$ consists of all OR_S where $S \subseteq \{1,\ldots,n\}$ and $|S| = k$, and the class OR is $\bigcup_{k=0}^{n} \mathsf{OR}(k)$. In other papers the class $\mathsf{OR}(k)$ is called 1-DNF of size k (over n variables). We use the shorter notion $\mathsf{OR}(k)$. In the same way we obtain the classes PAR and $\mathsf{PAR}(k)$ for the parity function which decides whether the number of ones in the input is odd or even. The threshold function on n variables and threshold t decides whether the input contains at least t ones. Again we replace the nonessential variables by 0. Then we consider the classes THR, $\mathsf{THR}(k)$, and $\mathsf{THR}_t(k)$, i. e., we distinguish whether we consider all $S \subseteq \{1,\ldots,n\}$ or all $S \subseteq \{1,\ldots,n\}$ of cardinality k and we distinguish whether the threshold t is unknown or known. In the theory of neural nets and also in learning theory more general threshold functions are considered. For given weights $w_1,\ldots,w_n$ and a threshold t the output of the corresponding threshold function is 1 iff $w_1x_1 + \cdots + w_nx_n \geq t$. We restrict ourselves to the case $w_1 = \cdots = w_n = 1$.

By de Morgan's rules it easily follows that all results for disjunctions also hold for conjunctions, if the nonessential variables for conjunctions are replaced by ones.