Introduction to Descriptive Complexity

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 600

Subsection 1

Preliminary Definitions

Turing Machines and Conventions

- We assume familiarity with the Turing machine.
- We survey computational complexity theory.
- We write $M(w)\downarrow$ to mean that Turing machine M accepts input w.
- We write $L(M)$ to denote the language accepted by M,

$$L(M) = \{w \in \{0,1\}^* : M(w)\downarrow\}.$$

- Instead of just accepting or rejecting, Turing machines may compute functions from binary strings to binary strings.
- We use $T(w)$ to denote the binary string that Turing machine T leaves on its write-only output tape when it is started with the binary string w on its input tape.
- If T does not halt on input w, then $T(w)$ is undefined.

Encoding of Structures as Boolean Strings

- Everything that a Turing machine does may be thought of as a query from binary strings to binary strings.
- In order to make Descriptive Complexity rich and flexible it is useful to consider queries that use other vocabularies.
- To relate such queries to Turing machine complexity, we fix a scheme that encodes the structures of vocabulary τ as boolean strings.
- To do this, for each τ, we define an encoding query,

$$\text{bin}_\tau : \text{STRUC}[\tau] \to \text{STRUC}[\tau_s].$$

  where $\tau_s = \langle S^1 \rangle$ is the vocabulary of boolean strings.
- The details of the encoding are not important.
- It is important, however, to know that, for each τ, bin_τ and its inverse are first-order queries.

The Binary Encoding

- Consider the vocabulary

$$\tau = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle.$$

- Let

$$\mathcal{A} = \langle \{0, 1, \ldots, n-1\}, R_1^{\mathcal{A}}, \ldots, R_r^{\mathcal{A}}, c_1^{\mathcal{A}}, \ldots, c_s^{\mathcal{A}} \rangle$$

  be an ordered structure of vocabulary τ.
- The relation $R_i^{\mathcal{A}}$ is a subset of $|\mathcal{A}|^{a_i}$.
- We encode this relation as a binary string

$$\text{bin}^{\mathcal{A}}(R_i)$$

  of length n^{a_i}, where "1" in a given position indicates that the corresponding tuple is in $R_i^{\mathcal{A}}$.
- For each constant $c_j^{\mathcal{A}}$, its number is encoded as a binary string

$$\text{bin}^{\mathcal{A}}(c_j)$$

  of length $\lceil \log n \rceil$.

The Binary Encoding (Cont'd)

- The binary encoding of the structure $\mathcal{A}$ is then just the concatenation of the bit strings encoding its relations and constants,

$$\text{bin}_\tau(\mathcal{A}) = \text{bin}^{\mathcal{A}}(R_1)\text{bin}^{\mathcal{A}}(R_2)\cdots\text{bin}^{\mathcal{A}}(R_r)\text{bin}^{\mathcal{A}}(c_1)\cdots\text{bin}^{\mathcal{A}}(c_s).$$

- We do not need any separators between the various relations and constants because the vocabulary τ and the length of $\text{bin}_\tau(\mathcal{A})$ determines where each section belongs.
- Observe that the length of $\text{bin}_\tau(\mathcal{A})$ is given by

$$\begin{aligned}\widehat{n}_\tau &= \|\text{bin}_\tau(\mathcal{A})\| \\ &= n^{a_1} + \cdots + n^{a_r} + s\lceil \log n \rceil.\end{aligned}$$

Conventions Concerning the Encoding

- We do not bother to include any numeric predicates or constants in $\text{bin}_\tau(\mathcal{A})$ since they can be easily recomputed.
- However, the coding $\text{bin}_\tau(\mathcal{A})$ does presuppose an ordering on the universe.
- There is no way to code a structure as a string without an ordering.
- Since a structure determines its vocabulary, in the sequel we usually write $\text{bin}(\mathcal{A}) := \text{bin}_\tau(\mathcal{A})$ for the binary encoding of $\mathcal{A} \in \text{STRUC}[\tau]$.
- We view bin as the union of bin_τ over all vocabularies τ.
- In the special case where τ includes no input relations symbols, we pretend that there is a unary relation symbol that is always false.

  Example: If $\tau = \varnothing$, then $\text{bin}(\mathcal{A}) = 0^{\|\mathcal{A}\|}$.
- We do this to insure that the size of $\text{bin}(\mathcal{A})$ is at least as large as $\|\mathcal{A}\|$.

Length of Input and Coding

- When $\tau = \tau_s$, the map bin_τ maps strings to strings.
- In this case, bin_τ is the identity map and, thus, $\widehat{n}_{\tau_s} = n$.
- In complexity theory, n is usually reserved for the length of the input.
- Here, we use n to denote the size of the input structure, $n = \|\mathcal{A}\|$.
- When the inputs are structures of vocabulary τ, the length of the input is $\widehat{n}_\tau$.
- For the case of binary strings, these two sizes coincide ($\widehat{n}_{\tau_s} = n$).
- When τ is understood, we write $\widehat{n}$ for $\widehat{n}_\tau$.
- Observe that n and $\widehat{n}$ are always polynomially related.
- There are two requirements of a coding function such as "bin".
  - First, it must be computationally very easy to encode and decode.
  - Secondly, the coding must be fairly space efficient.
    E.g., coding in unary would not be acceptable.

Computing a Query

Definition

Let $I : \mathrm{STRUC}[\sigma] \to \mathrm{STRUC}[\tau]$ be a query. Let T be a Turing machine. Suppose that, for all $\mathcal{A} \in \mathrm{STRUC}[\sigma]$,

$$T(\mathrm{bin}(\mathcal{A})) = \mathrm{bin}(I(\mathcal{A})).$$

$$\begin{array}{ccc}
\mathrm{STRUC}[\sigma] & \xrightarrow{I} & \mathrm{STRUC}[\tau] \\
\downarrow{\scriptstyle \mathrm{bin}_\sigma} & & \downarrow{\scriptstyle \mathrm{bin}_\tau} \\
\mathrm{STRUC}[\tau_s] & \xrightarrow[T]{} & \mathrm{STRUC}[\tau_s]
\end{array}$$

Then we say that T **computes** I.

Time and Space Complexity

- $\text{DTIME}[t(n)]$ denotes the set of boolean queries that are computable by a deterministic multi-tape Turing machine in $O(t(n))$ steps for inputs of universe size n.
- $\text{NTIME}[t(n)]$ denotes the set of boolean queries that are computable by a nondeterministic multi-tape Turing machine in $O(t(n))$ steps for inputs of universe size n.
- $\text{DSPACE}[s(n)]$ denotes the set of boolean queries that are computable by a deterministic multi-tape Turing machine using $O(s(n))$ work tape cells for inputs of universe size n.
- $\text{NSPACE}[s(n)]$ denotes the set of boolean queries that are computable by a nondeterministic multi-tape Turing machine using $O(s(n))$ work tape cells for inputs of universe size n.

Complexity Classes

- We assume that the reader is familiar with the following classical complexity classes.
  - $\mathrm{L} = \mathrm{DSPACE}[\log n]$;
  - $\mathrm{NL} = \mathrm{NSPACE}[\log n]$;
  - $\mathrm{P} =$ polynomial time $= \bigcup_{k=1}^{\infty} \mathrm{DTIME}[n^k]$;
  - $\mathrm{NP} =$ nondeterministic polynomial time $= \bigcup_{k=1}^{\infty} \mathrm{NTIME}[n^k]$;
  - $\mathrm{PSPACE} =$ polynomial space $= \bigcup_{k=1}^{\infty} \mathrm{DSPACE}[n^k] = \bigcup_{k=1}^{\infty} \mathrm{NSPACE}[n^k]$;
  - $\mathrm{EXPTIME} =$ exponential time $= \bigcup_{k=1}^{\infty} \mathrm{DTIME}[2^{n^k}]$.
- To talk about space $s(n)$, for $s(n) < \widehat{n}$, the Turing machine is assumed to have:
  - A read-only input tape of length $\widehat{n}$;
  - Some number of work tapes of total length $O(s(n))$.

Queries Computable in a Class $\mathcal{C}$

- To consider only boolean queries, as ordinarily done in the definitions of complexity classes as sets of decision problems, we adopt

Definition ($Q(\mathcal{C})$, the Queries Computable in $\mathcal{C}$)

Let $I : \text{STRUC}[\sigma] \to \text{STRUC}[\tau]$ be a query. We say that I is **computable in** $\mathcal{C}$ iff the boolean query I_b is an element of $\mathcal{C}$, where

$$I_b = \{(\mathcal{A}, i, a) : \text{The } i\text{th bit of bin}(I(\mathcal{A})) \text{ is "}a\text{"}\}.$$

Let $Q(\mathcal{C})$ be the set of all queries computable in $\mathcal{C}$,

$$Q(\mathcal{C}) = \mathcal{C} \cup \{I : I_b \in \mathcal{C}\}.$$

- For each of the above resources (deterministic and nondeterministic time and space) there is a hierarchy theorem saying that more of the given resource enables us to compute more boolean queries.

Space and Time Constructibility

- We say that a function $s : \mathbb{N} \to \mathbb{N}$ is **space constructible** (respectively, **time constructible**) iff there is a deterministic Turing machine running in space $O(s(n))$ (respectively, time $O(s(n))$) that on input 0^n, i.e., n in unary, computes $s(n)$ in binary.
- This is the same thing as saying that $s' \in Q(\text{DSPACE}[s(n)])$, respectively $s' \in Q(\text{DTIME}[s(n)])$, where s' is the function that on input 0^n computes $s(n)$ in binary.
- Every reasonable function is constructible, as is every function one finds in our discussion.
- Many theorems need the assumption that the time and space bounds in question are constructible.

The Space Hierarchy Theorem

The Space Hierarchy Theorem

For all space constructible $s(n) \geq \log n$, if

$$\lim_{n \to \infty} \frac{t(n)}{s(n)} = 0,$$

then DSPACE$[t(n)]$ is strictly contained in DSPACE$[s(n)]$.

- This is a diagonalization argument, but one has to be careful.

  On input M, the diagonalization program:

  - Marks off $s(|M|)$ tape cells;
  - Simulates machine M on input M.

  If $M(M)$ exceeds the given space or loops, then it should accept.

  Otherwise, do the opposite of what M would do.

Class Relations

- When comparing different resources, we are able to prove much less.
- For example, by Savitch's Theorem, for $s(n) \geq \log n$,

$$\mathrm{DSPACE}[s(n)] \subseteq \mathrm{NSPACE}[s(n)] \subseteq \mathrm{DSPACE}[(s(n))^2];$$

- However, we know only the trivial relationships between nondeterministic and deterministic time,

$$\mathrm{DTIME}[t(n)] \subseteq \mathrm{NTIME}[t(n)] \subseteq \mathrm{DTIME}[2^{O(t(n))}].$$

- Consider the following series of containments:

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}.$$

  - It follows from Savitch's Theorem and the Space Hierarchy Theorem that NL is not equal to PSPACE;
  - No other inequalities, including that L is not equal to NP, are known.

Subsection 2

Reductions and Complete Problems

Introducing Oracle Turing Machines and Reducibility

- Let A and B be boolean queries that may be difficult to compute.
- An **oracle** for B is a mythical device that, when given a structure $\mathcal{B}$, will answer in unit time whether or not $\mathcal{B}$ satisfies query B.
- We say that A is **Turing reducible** to B if it is easy to compute query A given an oracle for B.

Oracle Turing Machines

Definition (Oracle Turing Machine)

An **oracle Turing machine** is a Turing machine equipped with an extra tape called the **query tape**.

Let M be an oracle Turing machine and B be any boolean query.

M^B denotes the oracle Turing machine M equipped with an oracle for B.

M^B may write on its query tape like any other tape.

At any time, M^B may enter the "query state".

Assume that the string

$$w = \text{bin}(\mathcal{A})$$

is written on the query tape when M^B enters the query state.

At the next step, on the query tape, there will appear a:

$$\begin{cases} \text{"1"}, & \text{if } \mathcal{A} \in B, \\ \text{"0"}, & \text{otherwise.} \end{cases}$$

Turing Reducibility

- Obviously, M^B may answer any membership question

$$\text{"Does } \mathcal{A} \text{ satisfy } B\text{?"}$$

in linear time, the time to copy the string $\text{bin}(\mathcal{A})$ to its query tape.

Definition (Cont'd)

Let A, B be two boolean queries. Let $\mathcal{C}$ be a complexity class.
We say that A is $\mathcal{C}$-**Turing reducible** to B if there exists an oracle Turing machine M, such that:

- M^B runs in complexity class $\mathcal{C}$;
- $\mathcal{L}(M^B) = A$.

We denote this by $A \leq_{\mathcal{C}}^{T} B$, where "$T$" stands for Turing reduction.
An important example is polynomial-time Turing reduction, $\leq_{\mathrm{P}}^{T}$.

Example

- Define the boolean query CLIQUE to be the set of pairs $\langle G, k\rangle$, such that G is a graph, having a complete subgraph of size k.
- The vocabulary for CLIQUE is $\tau_{gk} = \langle E^2, k\rangle$.
- We can identify the universe of a structure $\mathcal{A} \in \text{STRUC}[\tau_{gk}]$ with the set $\{0, 1, \ldots, n-1\}$, where $n = \|\mathcal{A}\|$ is the number of vertices of $\mathcal{A}$.
- The constant k also represents a number between 0 and $n-1$.
- We will see later that CLIQUE is an NP-complete problem.
- Define the query MAX-CLIQUE(G) to be the size of a largest clique in graph G.
- We show that the boolean version of MAX-CLIQUE is polynomial time Turing reducible to CLIQUE.

Example (Cont'd)

- In symbols, we show that $\text{MAX-CLIQUE}_b \leq_{\mathrm{P}}^{T} \text{CLIQUE}$, where

$$\text{MAX-CLIQUE}_b = \{(G, i, a) : \text{bit } i \text{ of } \text{bin}(I(G)) \text{ is "}a\text{"}\}.$$

- The reduction is as follows.
- Consider a given input (G, i, a) for MAX-CLIQUE_b.
- Perform a binary search using an oracle for CLIQUE to determine the size s of the maximum clique for G.
- That is, ask if $(G, \frac{n}{2}) \in \text{CLIQUE}$.
  - If yes, ask if $(G, \frac{3n}{4}) \in \text{CLIQUE}$;
  - If no, ask if $(G, \frac{n}{4}) \in \text{CLIQUE}$.
- After $\log n$ queries to the oracle, s has been computed.
- Now accept iff bit i of s is "a".

Many-One Reductions

- A simpler and more popular kind of reduction in complexity theory is the *many-one reduction*.
- In descriptive complexity, we use *first-order reductions*, first-order queries that are at the same time many-one reductions.

Definition (Many-One Reduction)

Let $\mathcal{C}$ be a complexity class.
Let $A \subseteq \text{STRUC}[\sigma]$ and $B \subseteq \text{STRUC}[\tau]$ be boolean queries.
A $\mathcal{C}$-**many-one reduction from** A **to** B, in symbols $A \leq_{\mathcal{C}} B$, is a query

$$I : \text{STRUC}[\sigma] \to \text{STRUC}[\tau],$$

such that:

- I is an element of $Q(\mathcal{C})$;
- For all $\mathcal{A} \in \text{STRUC}[\sigma]$, $\mathcal{A} \in A$ iff $I(\mathcal{A}) \in B$.

Turing vs. Many-One Reductions

- For example:
  - When I is a first-order query, it is a first-order reduction ($\leq_{fo}$);
  - When $I \in Q(\mathrm{L})$, it is a logspace reduction ($\leq_{\log}$);
  - When $I \in Q(\mathrm{P})$, it is a polynomial-time reduction ($\leq_{\mathrm{P}}$).
- Many-one reduction is a particularly simple kind of Turing reduction.
- To decide whether $\mathcal{A}$ is an element of A:
  - Compute $I(\mathcal{A})$;
  - Ask the oracle whether $I(\mathcal{A})$ is an element of B.
- Many-one reductions are simpler than Turing reductions.
- Moreover, they seem to be sufficient in most situations.

Example

- We give a first-order reduction from PARITY to MULT_b.
- PARITY is the boolean query on binary strings that is true iff the string has an odd number of ones.
- We will see later that PARITY is not first-order.
- MULT, the multiplication query, maps a pair of boolean strings of length n to their product, a boolean string of length $2n$.
- Let $\tau_{ab} = (A^1, B^1)$ be the vocabulary of structures consisting of a pair A, B of boolean strings.
- Then

$$\text{MULT} : \text{STRUC}[\tau_{ab}] \to \text{STRUC}[\tau_s].$$

- Reductions map boolean queries to boolean queries.
- So we actually deal with the boolean version of MULT.
- MULT_b is a boolean query on structures of vocabulary $\tau_{abcd} = \langle A^1, B^1, c, d\rangle$ that is true iff bit c of the product of A and B is d.

Example (Cont'd)

- Recall that $\tau_s = \langle S^1 \rangle$ is the vocabulary of boolean strings.
- The first-order reduction

$$I_{PM} : \mathrm{STRUC}[\tau_s] \to \mathrm{STRUC}[\tau_{abcd}]$$

is given by the following formulas:

$$\begin{array}{rcl}\varphi_A(x,y) & \equiv & y = \max \wedge S(x) \\ \varphi_B(x,y) & \equiv & y = \max \\ I_{PM} & \equiv & \lambda_{xy}\langle \mathbf{true}, \varphi_A, \varphi_B, \langle 0, \max\rangle, \langle 0, 1\rangle\rangle.\end{array}$$

Example (Cont'd)

- Observe that the effect of this reduction is to line up all the bits of string A into column $n-1$ of the generated product.

		n				n					n			
A		0	…	0	s_0	0	…	0	s_1	…	0	…	0	s_{n-1}
B	$\times$	0	…	0	1	0	…	0	1	…	0	…	0	1
		0	…	0	s_0	0	…	0		…				
		0	…	0	s_1	0	…	0		…				
		0	…	0	$\vdots$	0	…	0		…				
		0	…	0	s_{n-1}	0	…	0		…				
			…		P					…				

Example (Cont'd)

- We have that

$$\mathcal{A} \in \text{PARITY} \quad \text{iff} \quad I_{PM}(\mathcal{A}) \in \text{MULT}_b.$$

- Thus, PARITY $\leq_{\text{fo}}$ MULT_b.
- So, if MULT were first-order, then PARITY would be as well.
- We will see later that PARITY is not first-order.
- This allows us to conclude that MULT is not first order.

Completeness for a Complexity Class

- Suppose $\mathcal{C}$ is a weak complexity class such as FO or L.
- The intuitive meaning of $A \leq_{\mathcal{C}} B$ is that the complexity of problem A is less than or equal to the complexity of problem B.
- A being *complete for* $\mathcal{C}$ means that A is a hardest query in $\mathcal{C}$.
- That is, every query in $\mathcal{C}$ can be rephrased as an instance of A.

Definition (Completeness for a Complexity Class)

Let A be a boolean query.
Let $\mathcal{C}$ be a complexity class.
Let $\leq_r$ be a reducibility relation.
We say that A is **complete for** $\mathcal{C}$ **via** $\leq_r$ if:

1. $A \in \mathcal{C}$;
2. For all $B \in \mathcal{C}$, $B \leq_r A$.

Completeness and Reductions

- When we say that a problem is *complete for a complexity class* without mentioning under what reduction, then we implicitly mean via first-order reductions $\leq_{\text{fo}}$.
- We will show that, if a problem is complete via first-order reductions, then it is also complete via:
    - Logspace reductions;
    - Polynomial-time reductions.

Short List of Complete Problems

- Complete for L:
  - CYCLE: Given an undirected graph, does it contain a cycle?
  - REACH_d: Given a directed graph, is there a deterministic path from vertex s to vertex t? (A **deterministic path** is such that, for every edge (u, v) on the path, there is only one edge in the graph from u.)
- Complete for NL:
  - REACH: Given a directed graph, is there a path from vertex s to vertex t?
  - 2-SAT: Given a boolean formula in conjunctive normal form, with only two literals per clause, is it satisfiable?
- Complete for P:
  - CIRCUIT-VALUE-PROBLEM (CVP): Given an acyclic boolean circuit, with inputs specified, does its output gate have value one?
  - NETWORK-FLOW: Given a directed graph, with capacities on its edges, and a value V, is it possible to achieve a steady-state flow of value V through the graph?

Short List of Complete Problems (Cont'd)

- Complete for NP:
  - SAT: Given a boolean formula, is it satisfiable?
  - 3-SAT: Given a boolean formula in conjunctive normal form with only three literals per clause, is it satisfiable?
  - CLIQUE: Given an undirected graph and a value k, does the graph have a complete sub graph with k vertices?
- Complete for PSPACE:
  - QSAT: Given a quantified boolean formula, is it satisfiable?
  - HEX, GEOGRAPHY, GO: Given a position in the generalized versions of the games hex, geography or go, is there a forced win for the player whose move it is?

Example

- SAT is the set of boolean formulas in conjunctive normal form (CNF) that admit a satisfying assignment, i.e., a way to set each boolean variable to **true** or **false** so that the whole formula evaluates to **true**.

  Example: Consider the following boolean formulas ($\overline{v}$ means $\neg v$):

$$\begin{aligned}\varphi_0 &= (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_4) \wedge (x_2 \vee \overline{x}_3 \vee x_5);\\ \varphi_1 &= (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3)\\ &\quad (\overline{x}_1 \vee \overline{x}_4 \vee x_5) \wedge (\overline{x}_1 \vee x_4 \vee \overline{x}_5) \wedge (\overline{x}_1 \vee x_4 \vee x_5) \wedge (\overline{x}_1 \vee \overline{x}_4 \vee \overline{x}_5).\end{aligned}$$

  It can be verified that $\varphi_0 \in \text{SAT}$ and $\varphi_1 \notin \text{SAT}$.
- It is easy to see that SAT is in NP.

  In linear time, a nondeterministic algorithm can write down a "0" or a "1" for each boolean variable.

  Then it can deterministically check that each clause has been assigned at least one "1", and if so, accept.

Example (Cont'd)

- A boolean formula φ that is in CNF may be thought of as a set of clauses, each of which is a disjunction of literals.
- Recall that a literal is an atomic formula - in this case a boolean variable - or its negation.
- Thus, a natural way to encode φ is via the structure

$$\mathcal{A}_\varphi = \langle A, P, N \rangle.$$

  - The universe A is a set of clauses and variables.
  - The relation $P(c, v)$ means that variable v occurs positively in clause c;
  - $N(c, v)$ means that variable v occurs negatively in clause c.
- We can think of every element of the universe as a variable and a clause.
- Accordingly, $n = \|\mathcal{A}_\varphi\|$ is equal to the maximum of the number of variables and the number of clauses occurring in φ.

Example (Cont'd)

- If v is really a variable but not a clause, we can harmlessly make it the clause $(v \vee \overline{v})$ by adding the pair (v, v) to the relations P and N.

  Example: We revisit

$$\varphi_0 = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_4) \wedge (x_2 \vee \overline{x}_3 \vee x_5).$$

  A structure coding φ_0 in this way is:

$$\begin{array}{rcl} \mathcal{A}_{\varphi_0} & = & \langle \{1,2,3,4,5\}, P, N \rangle; \\ P & = & \{(1,1),(1,3),(2,4),(3,2),(3,5),(4,4),(5,5)\}; \\ N & = & \{(1,2),(2,1),(2,2),(3,3),(4,4),(5,5)\}. \end{array}$$

Reduction from SAT to CLIQUE

- We show that SAT is first-order reducible to CLIQUE.
- Let $\mathcal{A}$ be a boolean formula in CNF with:
  - Clauses $C = \{c_1, \ldots, c_n\}$;
  - Variables $V = \{v_1, \ldots, v_n\}$.
- Let $L = \{v_1, \ldots, v_n, \overline{v}_1, \ldots, \overline{v}_n\}$.
- Define the instance of the clique problem

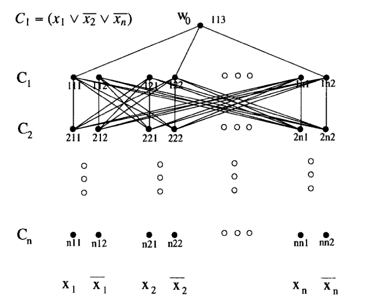
$$g(\mathcal{A}) = (V^{g(\mathcal{A})}, E^{g(\mathcal{A})}, k^{g(\mathcal{A})})$$

as follows:

$$\begin{array}{rcl} V^{g(\mathcal{A})} & = & (C \times L) \cup \{w_0\}; \\ E^{g(\mathcal{A})} & = & \{(\langle c_1, \ell_1 \rangle, \langle c_2, \ell_2 \rangle) : c_1 \neq c_2 \text{ and } \overline{\ell}_1 \neq \ell_2\} \cup \\ & & \{(w_0, \langle c, \ell \rangle), (\langle c, \ell \rangle, w_0) : \ell \text{ occurs in } c\}; \\ k^{g(\mathcal{A})} & = & n + 1 = \|\mathcal{A}\| + 1. \end{array}$$

Reduction from SAT to CLIQUE (Construction)

- The graph $g(\mathcal{A})$ is an $n \times n$ array of vertices containing:
  - A row for every clause in $\mathcal{A}$;
  - A column for every literal in L, plus a top vertex w_0.
- There are edges between vertices $\langle c_1, \ell_1 \rangle$ and $\langle c_2, \ell_2 \rangle$ iff:
  - $c_1 \neq c_2$, i.e., the points come from different clauses;
  - $\overline{\ell_1} \neq \ell_2$, i.e., literals ℓ_1 and ℓ_2 are not the negations of each other.



- The other edges in the graph are between w_0 and those $\langle c, \ell \rangle$ such that literal ℓ occurs in clause c.

Reduction from SAT to CLIQUE (Proof)

- Observe that a clique of size $n+1$ must involve w_0 and one vertex from each clause.
- This corresponds to a satisfying assignment, because no literal and its negation can be in a clique.
- Conversely, consider a satisfying assignment to $\mathcal{A}$.
- It determines an $(n+1)$-clique consisting of w_0 together with one literal per clause that is assigned "true".
- It follows that mapping g is indeed a many-one reduction,

$$(\mathcal{A} \in \text{SAT}) \quad \Leftrightarrow \quad (g(\mathcal{A}) \in \text{CLIQUE}).$$

Reduction from SAT to CLIQUE (Query)

- We now give the rather technical details of writing g as a first-order query
$$g = \lambda_{x^1x^2x^3y^1y^2y^3}\langle\varphi_0,\varphi_1,\psi_1\rangle.$$
- We encode the vertices as triples $\langle x^1, x^2, x^3\rangle$, where:
  - x^1 corresponds to the clause;
  - x^2 corresponds to the variable;
  - $x^3 = 1$ means the variable is positive and $x^3 = 2$ means the variable is negative.
- Vertex w_0 is $\langle 1,1,3\rangle$, the only triple with $x^3 > 2$.
- The numeric formula φ_0, which describes the universe of $g(\mathcal{A})$, is
$$\varphi_0 \equiv (x^3 \le 2) \vee (x^1x^2x^3 = 113).$$

Reduction from SAT to CLIQUE (Query Cont'd)

- Next, we define the edge relation.
- First, let

$$\begin{array}{rcl} \varphi_1'(\overline{x},\overline{y}) & \equiv & \alpha_1 \vee (\alpha_2 \wedge P(y^1,y^2)) \vee (\alpha_3 \wedge N(y^1,y^2)), \\ \alpha_1 & \equiv & x^1 \neq y^1 \wedge x^3 < 3 \wedge y^3 < 3 \wedge (x^2 = y^2 \to x^3 = y^3); \\ \alpha_2 & \equiv & x^3 = 3 \wedge y^3 = 1; \\ \alpha_3 & \equiv & x^3 = 3 \wedge y^3 = 2. \end{array}$$

- Next, let φ_1 be the symmetric closure of φ_1',

$$\begin{array}{l} \varphi_1(x^1,x^2,x^3,y^1,y^2,y^3) \\ \equiv \varphi_1'(x^1,x^2,x^3,y^1,y^2,y^3) \vee \varphi_1'(y^1,y^2,y^3,x^1,x^2,x^3). \end{array}$$

- Notice that φ_1 is a direct translation of the equation defining $E^{g(\mathcal{A})}$.

Reduction from SAT to CLIQUE (Query Cont'd)

- We are thinking of the elements of the ordered universe as $1, 2, \ldots, n$ instead of the usual $0, 1, \ldots, n-1$.
- For this reason, the number $n+1$ which would usually be represented by 011 in lexicographic order, is instead 122.
- Formula ψ_1 identifies k as $n+1$:

$$\psi_1(x^1, x^2, x^3) \equiv x^1 x^2 x^3 = 122.$$

- We have correctly encoded the desired first-order reduction g.
- Moreover, equivalence $\mathcal{A} \in \text{SAT}$ iff $g(\mathcal{A}) \in \text{CLIQUE}$ holds.

Subsection 3

Alternation

Complements and Complementary Classes

- Let $A \subseteq \text{STRUC}[\tau]$ be a boolean query.
- Define its **complement** $\overline{A} = \text{STRUC}[\tau] - A$.
- Let $\mathcal{C}$ be a complexity class.
- Define the **complementary class** co–$\mathcal{C}$ by

$$\text{co-}\mathcal{C} = \{\overline{A} : A \in \mathcal{C}\}.$$

Example: We know that SAT is in NP.

Its complementary problem $\overline{\text{SAT}} = \text{UNSAT}$ is in co-NP.

NP and co-NP

- The question whether NP is closed under complementation, i.e., whether NP is equal to co-NP, is open.
- Most people believe that these classes are different.
- Notice that if one could really build an NP machine, then one could also build a co-NP machine.
- All that is needed is a single gate to invert the former machine's answer.
- Thus from a very practical point of view, the complexity of a problem A and its complement, $\overline{A}$, are identical.

Parallel Machine View of NP

- One way to imagine a realization of an NP machine is via a parallel or biological machine with many processors.
- At each step, each processor p_i:
  - Creates two copies of itself;
  - Sets them to work on two slightly different problems;
  - If either of these offspring ever accepts, i.e., says "yes" to p_i, then p_i in turn says "yes" to its parent.
- These "yes" answers travel up a binary tree to the root and the whole nondeterministic process accepts.

Parallel Machine View of NP and Alternation

- In such a view of nondeterminism, in time $t(n)$ we can build about $2^{t(n)}$ processors.
- However, these processors are not taken full advantage of.
  - Their pattern of communication is very weak;
  - Each processor can compute only the "or" of its children.
- Thus, the whole computation is one big "or" of its leaves.
- *Alternation* generalizes nondeterminism so that:
  - It is closed under complementation;
  - Makes better use of its processors.

Alternating Turing Machines: States and Configurations

Definition

An **alternating Turing machine** is a Turing machine whose states are divided into two groups:

- The **existential states**;
- The **universal states**.

Recall that a **configuration**, or an **instantaneous description** (**ID**), of any Turing machine consists of:

- The machine's state;
- The contents of the work-tape;
- The head positions.

Alternating Turing Machines: Acceptance

Definition (Cont'd)

The notion of when such a machine accepts an input is defined by induction.

The alternating Turing machine **accepts in a given configuration** c if one of the following hold:

1. c is in a final accepting state;
2. c is in an existential state and there exists a next configuration c' that accepts;
3. c is in a universal state, there is at least one next configuration, and all next configurations accept.

Finally, the alternating Turing machine **accepts** if its accepts in its initial configuration.

Adding Random Access Read-Only Input

- Turing machines access their tapes sequentially.
- This makes it difficult for them to do anything in sublinear time.
- Alternating Turing machines can sensibly use sublinear time.
- So it is more reasonable to use machines that have a more random access nature.
- As a compromise, from now on we assume that our Turing machines have a **random access read-only input**:
    - There is an **index tape**, which can be written and read like other tapes;
    - Whenever the value v, written in binary, appears on the index tape, the read head automatically scans bit v of the input.

Alternating Classes

- Define the complexity class $\text{ASPACE}[s(n)]$ to be the set of boolean queries accepted by alternating Turing machines using a maximum of $O(s(n))$ tape cells in any computation path on an input of length n.
- Define the complexity class $\text{ATIME}[t(n)]$ to be the set of boolean queries accepted by alternating Turing machines using a maximum of $O(t(n))$ time steps in any computation path on an input of length n.

Alternating and Deterministic Classes

- The main relationships between alternating and deterministic complexity classes are given by the following theorem.

Theorem

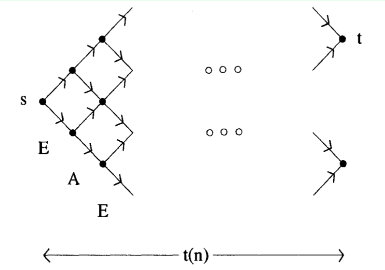
For $s(n) \geq \log n$, and for $t(n) \geq n$,

$$\begin{aligned} \bigcup_{k=1}^{\infty} \mathrm{ATIME}[(t(n))^k] &= \bigcup_{k=1}^{\infty} \mathrm{DSPACE}[(t(n))^k]; \\ \mathrm{ASPACE}[s(n)] &= \bigcup_{k=1}^{\infty} \mathrm{DTIME}[k^{s(n)}]. \end{aligned}$$

In particular, $\mathrm{ASPACE}[\log n] = \mathrm{P}$ and alternating polynomial time is equal to PSPACE.

Simplifying Conventions

- The figure shows the computation graph of an alternating machine.
- We assume for convenience that such machines:
  - Have a unique accepting configuration "s";
  - Have a unique rejecing configuration "t";
  - Each configuration has at most two possible next moves.



- We also assume that these machines have clocks that uniformly cause the machines to shut off at a fixed time that is a function of the length of the input.
- "Shutting off" means entering the reject configuration unless the machine is already in the accept configuration.

Comments on the Model

- The letters "E" and "A" below the vertices indicate whether the corresponding configurations are existential or universal.
- If they were all existential, then this would be a nondeterministic computation.
- The time $t(n)$ measures the depth of the computation graph.
- We may think that, at each branching move, an extra processor is created.
- In time $t(n)$ potentially $2^{O(t(n))}$ processors are created.

Comments on the Model (Cont'd)

- The two processors created take the two branches.
- Eventually, they complete their tasks and report their answers to their parent.
  - If the parent was existential, then it reports "accept" iff at least one of its children accepts;
  - If the parent is universal, then it reports "accept" iff both of its children accept.
- The space used by an alternating machine is the maximum amount of space used in any path through its computation graph.

Boolean Circuits

Definition

A **boolean circuit** is a directed acyclic graph (DAG),

$$C = (V, E, G_\wedge, G_\vee, G_\neg, I, r) \in \mathrm{STRUC}[\tau_c],$$

where

$$\tau_c = \langle E^2, G_\wedge^1, G_\vee^1, G_\neg^1, I^1, r\rangle.$$

Internal node w is:

- An **and-gate** if $G_\wedge(w)$ holds;
- An **or-gate** if $G_\vee(w)$ holds;
- A **not-gate** if $G_\neg(w)$ holds.

The nodes v with no edges entering them are called **leaves**.
The **input relation** $I(v)$ represents the fact that the leaf v is on.
Often we will be given a circuit C and, separately, its input relation I.

The Circuit Value Problem

Definition

We define two problems.

- The **Circuit Value Problem** (**CVP**) consists of those circuits whose root gate r evaluates to one.
- The **Monotone Circuit Value Problem** (**MCVP**) is the subset of CVP in which no negation gates occur.

Complexity of MCVP

Proposition

MCVP is recognizable in ASPACE$[\log n]$.

- Let G be a monotone boolean circuit.
  Define the procedure "EVAL(a)", where a is a vertex of G.
  1. if $I(a)$ then accept
  2. else if a has no outgoing edges then reject
  3. if $G_\wedge(a)$ then in a universal state choose a child b of a
  4. else in an existential state choose a child b of a
  5. Return (EVAL(b))

  The machine M simply calls EVAL(r).

  Observe that EVAL(a) returns "accept" iff gate a evaluates to one.

  The space used by EVAL is just the space to name two vertices a, b.

  Thus, M is an ASPACE$[\log n]$ machine accepting MCVP.

  An appropriate time limit for the machine would be $n = \|G\|$, which is an upper bound on the length of the longest path.

The Quantified Satisfiability Problem

Definition

The **Quantified Satisfiability Problem** (**QSAT**) is the set of true formulas of the following form:

$$\Psi = (Q_1 x_1)(Q_2 x_2) \cdots (Q_r x_r)\varphi,$$

where φ is a boolean formula, each Q_i is either $\forall$ or $\exists$ and $x_1, \ldots, x_r$ are the boolean variables occurring in φ.

- Observe that for any boolean formula φ on variables $\overline{x}$,

$$\begin{array}{lcl} \varphi \in \text{SAT} & \Leftrightarrow & (\exists \overline{x})\varphi \in \text{QSAT}; \\ \varphi \notin \text{SAT} & \Leftrightarrow & (\forall \overline{x})\neg\varphi \in \text{QSAT}. \end{array}$$

- Thus QSAT logically contains both SAT and $\overline{\text{SAT}}$.

Complexity of QSAT

Proposition

QSAT is recognizable in ATIME$[n]$.

- Construct an alternating machine A that works as follows.
  Suppose given input $\Phi \equiv (\exists x_1)(\forall x_2)\cdots(Q_r x_r)\alpha(\overline{x})$.
  In an existential state, A writes down a boolean value for x_1.
  In a universal state it writes a bit for x_2, and so on.
  Next A must evaluate the quantifier-free formula α on these values.
  This is easy for an alternating machine.
  For each "$\wedge$" in α, A universally chooses which side to evaluate.
  For each "$\vee$", A existentially chooses.
  Thus, A only has to read one of the chosen bits x_i and accept iff it is true and occurs positively, or false and occurs negatively.
  A runs in linear time and accepts the sentence Φ iff Φ is true.

Alternating Time and (Non)deterministic Space

Theorem

Let $s(n) \geq \log n$ be space constructible. Then,

$$\mathrm{NSPACE}[s(n)] \subseteq \mathrm{ATIME}[s(n)^2] \subseteq \mathrm{DSPACE}[s(n)^2].$$

- We start with the first inclusion.

  Let N be an $\mathrm{NSPACE}[s(n)]$ Turing machine.

  Let w be an input to N.

  Let G_w denote the computation graph of N on input w.

  N accepts w iff there is a path from s to t in G_w.

  We construct an $\mathrm{ATIME}[s(n)^2]$ machine A that accepts the same language as N.

Alternating Time and (Non)deterministic Space (Cont'd)

- A does this by calling the subroutine, $P(d, x, y)$.

  P accepts iff there is a path in G_w of length at most 2^d from x to y.

  For $d > 0$, P is defined by

  $$P(d, x, y) \equiv (\exists z)(P(d-1, x, z) \wedge P(d-1, z, y)).$$

  P works by:
  - Existentially choosing a middle configuration z;
  - Universally choosing the first half or the second half;
  - Checking that the appropriate path of length 2^{d-1} exists.

  Thus, the time $T(d)$ taken to compute $P(d, x, y)$ is the sum of:
  - The time to write down a new, middle configuration;
  - The time to compute $P(d-1, x', y')$.

Alternating Time and (Non)deterministic Space (Cont'd)

- The number of bits in a configuration of G_w is $O(s(n))$, where $n = |w|$.

  Thus,
  $$T(d) = O(s(n)) + T(d-1) = O(d \cdot s(n)).$$

  The length of the maximum useful path in G_w is bounded by the number of configurations of N on input w.

  That is, it is bounded by $2^{cs(n)}$, for an appropriate value of c.

  Thus, on input w, A:
  - Calls $P(cs(n), s, t)$;
  - Receives its answer in time
  $$O(cs(n)s(n)) = O(s(n)^2).$$

Alternating Time and (Non)deterministic Space (Cont'd)

- We turn now to the second inclusion.

  Let A be an ATIME$[t(n)]$ machine.

  On input w, A's computation graph has:
  - Depth $O(t(n))$;
  - Size $2^{O(t(n))}$.

  A deterministic Turing machine can systematically search this entire and-or graph using space $O(t(n))$.

  It keeps a string

  $$c_1 c_2 \dots c_r * \dots *$$

  of length $O(t(n))$, denoting that:
  - We are currently simulating step r of A's computation;
  - We have made choices $c_1 \dots c_r$ on all of the existential and universal branches up until this point.

Alternating Time and (Non)deterministic Space (Cont'd)

- The rest of the simulation will report an **answer** as to whether choices $c_1 \ldots c_r$ will lead to acceptance, as follows.

  Suppose one of the following holds:

  1. $c_r = 1$;
  2. **answer** = "yes" and step r was existential;
  3. **answer** = "no" and step r was universal.

  Then, let $c_r = *$ and report **answer** back to step $r - 1$.

  Otherwise, set $c_r = 1$ and continue.

  Note that $c_1 c_2 \ldots c_r * \ldots *$ uniquely determines which configuration of A to go to next.

  So we do not have to store intermediate configurations of the simulation.

Consequences

- An immediate corollary is Savitch's Theorem.

Corollary (Savitch's Theorem)

Let $s(n) \geq \log n$ be space constructible. Then,

$$\mathrm{NSPACE}[s(n)] \subseteq \mathrm{DSPACE}[s(n)^2].$$

- It is the best known simulation of nondeterministic space by deterministic space.
- It is unknown whether equality holds in either or both of the inclusions

$$\mathrm{NSPACE}[s(n)] \subseteq \mathrm{ATIME}[s(n)^2] \subseteq \mathrm{DSPACE}[s(n)^2].$$

Alternating Space and Deterministic Time

- Another corollary of the theorem is the first part of the following.

Theorem

For $s(n) \geq \log n$, and for $t(n) \geq n$,

$$\bigcup_{k=1}^{\infty} \mathrm{ATIME}[(t(n))^k] = \bigcup_{k=1}^{\infty} \mathrm{DSPACE}[(t(n))^k];$$

$$\mathrm{ASPACE}[s(n)] = \bigcup_{k=1}^{\infty} \mathrm{DTIME}[k^{s(n)}].$$

In particular, $\mathrm{ASPACE}[\log n] = \mathrm{P}$ and alternating polynomial time is equal to PSPACE.

- We show the second part next.

Proof of the Theorem

- We show that ASPACE$[s(n)]$ is DTIME$[O(1)^{s(n)}]$.

  One direction is obvious.

  An ASPACE$[s(n)]$ machine has $O(1)^{s(n)}$ possible configurations.

  Thus, its entire computation graph is of size $O(1)^{s(n)}$.

  Thus, it may be traversed in DTIME$[O(1)^{s(n)}]$.

  The same traversal algorithm as in the second half of the proof of the preceding theorem works in this case.

Proof of the Theorem (Cont'd)

- We now work for the reverse inclusion.

  We are given a DTIME$[k^{s(n)}]$ machine M.

  Let w be an input to M, with $n = |w|$.

  We can view M's computation as a $k^{s(n)} \times k^{s(n)}$ table.

	Space						
	1	2	p	n			$T(n)$
Time 0	$\langle q_0, w_1 \rangle$	w_2	$\cdots$	w_n	b	$\cdots$	b
1	w_1	$\langle q_1, w_2 \rangle$	$\cdots$	w_n	b	$\cdots$	b
	$\vdots$	$\vdots$	$\vdots$			$\vdots$	
t			$a_{-1}a_0a_1$				
$t+1$			a				
	$\vdots$	$\vdots$	$\vdots$			$\vdots$	
$T(n)$	$\langle q_f, 0 \rangle$	$\cdots$	$\cdots$			$\cdots$	

  Cell (t,p) of this table contains the symbol that is in position p of M's tape at time t of the computation.

  Furthermore, if M's head was at position p at time t, then this cell should also include M's state at time t.

Proof of the Theorem (Cont'd)

- We define an alternating procedure $C(t,p,a)$.

  C accepts iff the contents of cell p at time t in M's computation on input w consist of symbol a.

  $C(0,p,a)$ holds iff a is the correct symbol in position p of M's initial configuration on input w.

  This means that position 1 contains $\langle q_0, w_1 \rangle$, where:
  - q_0 is M's start state;
  - w_1 is the first symbol of w.

  Similarly, for $2 \le p \le n$, $C(0,p,a)$ holds iff $a = w_p$.

  Inductively, $C(t+1,p,a)$ holds iff

  > the three symbols $a_{-1}a_0a_1$ in tape positions $p-1, p, p+1$ lead to an "a" in position p in one step of M's computation.

  We denote this symbolically as $(a_{-1}, a_0, a_1) \stackrel{M}{\to} a$.

Proof of the Theorem (Cont'd)

- This condition can be read directly from M's transition table,

$$\begin{array}{rcl} C(t+1,p,a) & \equiv & (\exists a_{-1},a_0,a_1)((a_{-1},a_0,a_1) \overset{M}{\to} a \wedge \\ & & (\forall i \in \{-1,0,1\})(C(t,p+i,a_i))). \end{array}$$

  The formula can be evaluated by an alternating machine using the space to hold the values of t and p.

  This space requirement is $O(\log k^{s(n)}) = O(s(n))$.

  Note that M accepts w iff $C(k^{s(n)},1,a_f)$ holds, where a_f is the contents of the first cell of M's accept configuration.

  For example, we can use $a_f = \langle q_f, 0\rangle$, where q_f is M's accept state.

Subsection 4

Simultaneous Resource Classes

The Polynomial Hierarchy

- Let ASPACE-TIME$[s(n), t(n)]$ be the set of boolean queries accepted by alternating machines simultaneously using space $s(n)$ and time $t(n)$.
- Let ATIME-ALT$[t(n), a(n)]$ be the set of boolean queries accepted by alternating machines simultaneously using time $t(n)$ and making at most $a(n)$ alternations between existential and universal states, starting with existential.
- By, definition, ATIME-ALT$[n^{O(1)}, 1]$ = NP.
- Define the **polynomial time hierarchy** (**PH**) to be the set of boolean queries accepted in polynomial time by alternating Turing machines making a bounded number of alternations between existential and universal states,

$$\text{PH} = \bigcup_{k=1}^{\infty} \text{ATIME-ALT}[n^k, k].$$

Nick's Class NC

- Define NC (**Nick's Class**) to be the set of boolean queries accepted by alternating Turing machines in $\log n$ space and poly log time:

$$\mathrm{NC} = \bigcup_{k=1}^{\infty} \text{ASPACE-TIME}[\log n, \log^k n].$$

- A more usual definition of NC (to be encountered later) is as the class of boolean queries accepted by a parallel random access machine using:
  - Polynomially much hardware;
  - Poly log parallel time.

Subsection 5

Summary

Complexity Classes

- A list of the complexity classes defined so far:

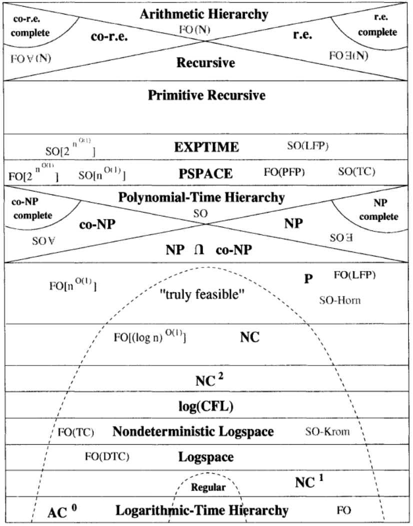
$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{NC} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PH} \subseteq \mathrm{PSPACE}.$$

- These containments are easy to prove.
- On the other hand there is very little known about the strictness of the above inclusions.

  It has not yet been proved that L is not equal to PH, or that P is not equal to PSPACE.
- The fact that we cannot prove these inequalities reveals just the tip of the iceberg of what we do not know concerning the computational complexity of important computational problems.

  E.g., for all known NP complete problems, the best known algorithms to get an exact solution are all exponential time in the worst case. However, no proof exists that they are not computable in linear time.

Computability and Complexity World



Feasibility

- The set of boolean queries called "truly feasible" are the queries that can be computed exactly with an "affordable" amount of time and hardware, on all "reasonably sized" instances.
- The truly feasible queries are a proper subset of P.
- Many important problems that we need to solve are not truly feasible.
- The theory of algorithms and complexity helps us determine whether the problem we need to solve is feasible.
- If it is not, it suggests ways to choose a limited set of instances of the problem - or easier versions of them - that are feasible.

Descriptive Complexity

- Complexity via Turing machines is isomorphic to descriptive complexity, i.e., the theory of complexity via logic formulas.
- We will give descriptive characterizations of some of the classes in the figure.
- We mention here some examples.
  - The logarithmic time hierarchy is equal to the set of first order boolean queries (LH = FO);
  - The polynomial time hierarchy is the set of second order boolean queries (PH = SO);
  - The arithmetic hierarchy is defined to be the set of boolean queries that are describable in the first order theory of the natural numbers.