University of California, Los Angeles
CS 281 Computability and Complexity

Instructor: Alexander Sherstov
Scribe: Aditya Gomatam
Date: January 29, 2024

LECTURE
6

Comparing Problems, and the Universal Turing Machine

In the previous lecture, we proved the Cook-Levin theorem, that SAT and 3-SAT are **NP**-Complete. This entails the humbling result that the computational power of Turing Machines equals the computational power of Boolean CNF formulas, modulo a quantifier. In this lecture, we focus on solving previous challenge problems, promoting a discussion of the differences between optimization, search, and decision versions of problems, as well as the limitations of CNFs compared to Turing Machines. We show the equivalence of decision and search versions of problems in **NP** by investigating the INDSET language. We show that the PARITY and MAJORITY languages require at least exponential-size CNFs and DNFs. Finally, we specify the Universal Turing Machine in preparation for our discussion of Diagonalization, a technique that will yield many results on the power of Turing Machines.

6.1 NP-Completeness of INDSET

Recall the INDSET language [3]:

Definition 6.1. INDSET $= \{(G, k) : G$ is a graph that has an independent set of size $k\}$. An independent set S is a subset of the vertices of G such that between any two elements of S, there is no edge in G.

We investigate INDSET to illustrate the interplay between optimization, search and decision versions of problems. As we have defined deciders, a decider D for INDSET would solve the *decision* version of the independent set problem — D would output 1 iff a given $(G, k) \in$ INDSET, else 0 [2]. That is, given (G, k), D would output 1 if there *exists* an independent set of size k in G; else D would output 0. D would not be concerned with actually finding an independent set of size k in G, should one exist. This latter problem is the *search* version of INDSET. Note that the languages of the decision and search versions are exactly the same — it is possible to compute an independent set of size k in a graph G only if $(G, k) \in$ INDSET. Intuitively then, it seems that the search and decision versions of INDSET are "equivalent" in some sense. Somewhat less clear is the relationship of decision and search versions to the *optimization* version of

INDSET — find an independent set of maximal size in G. The language MAX-INDSET $= \{(G,k) : G$ is a graph whose independent set of maximal size is of size $k\}$ clearly corresponds to a strict subset of the language in Definition 6.1. We will explore the relationship between these problem versions more thoroughly in the next section.

First, we show that the decision version of INDSET is **NP**-Complete. It was shown in Lecture 4 that the decision version of INDSET is in **NP** [3]. It remains to show that INDSET is **NP**-Hard. Since we now know that 3-SAT is **NP**-Complete, if we are able to reduce 3-SAT to INDSET in polynomial time, we are done.

THEOREM 6.2. *INDSET is **NP**-Hard.*

Proof. We reduce 3-SAT to INDSET. Let ϕ be a 3-CNF with m clauses. ϕ can be written as $C_1 \wedge C_2 \wedge ... \wedge C_m$ for some clauses $C_1, C_2, ..., C_m$. We wish to construct a graph G such that G has an independent set of some size iff there exists a satisfying assignment to the variables of ϕ. Naturally then, a lack of edges between two vertices (their independence) should correspond to a possible satisfying variable assignment for ϕ.

This is precisely how we approach the reduction. We construct the graph $G = (V, E)$ as follows. For every clause C_i in ϕ, we create a node in V representing a possible satisfying assignment for the variables in C_i. Since each clause is a disjunction of at most 3 literals, there are at most $2^3 - 1 = 7$ ways to satisfy it. G therefore contains at most $7m$ nodes.

Now, if there are more than 3 variables in ϕ, each generated node represents only a *partial* satisfying assignment to the variables of ϕ, and guarantees the satisfaction of exactly 1 clause. Since ϕ is a CNF, we need to show there exists a variable assignment that satisfies every clause in ϕ. In order for such an assignment to correspond to an independent set in G, we must have no edges between nodes of compatible variable assignments. Thus, we create an edge (u, v) in E if and only if u and v correspond to partial variable assignments that conflict (i.e. contradict one another).

This process is made more concrete by the following example:

EXAMPLE 6.3. Constructing the graph G for $\phi(x_1x_2x_3x_4) = (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_2)$ to reduce 3-SAT for ϕ to INDSET for G. Consider each clause in turn, and create a node for each satisfying assignment:

- $x_1 \vee x_2 \vee x_3$ has 7 satisfying assignments: $[x_1x_2x_3 = 001], [x_1x_2x_3 = 010], [x_1x_2x_3 = 011], [x_1x_2x_3 = 100], [x_1x_2x_3 = 101], [x_1x_2x_3 = 110], [x_1x_2x_3 = 111]$.
- $x_2 \vee \overline{x_4}$ has 3 satisfying assignments: $[x_2x_4 = 00], [x_2x_4 = 10], [x_2x_4 = 11]$.
- $\overline{x_1} \vee x_2$ has 3 satisfying assignments: $[x_1x_2 = 00], [x_1x_2 = 01], [x_1x_2 = 11]$.

We can visualize the graph so far as in Figure 6.1.

The separation into clusters in Figure 6.1 for each clause is deliberate. There are exactly as many clusters as clauses in the original 3-CNF. Now, within each cluster, every node corresponds to a different assignment to the same variables (as these nodes all originate from the same clause). Thus, every node in each cluster is incompatible with any other node in the cluster, so we create an edge between every two nodes in each cluster. Moreover, wherever the nodes between any two clusters conflict, we add another edge. The final graph is shown in Figure 6.2.

We noted earlier that an arbitrary 3-CNF ϕ with m clauses induces a graph G with at most $7m$ nodes. Thus, since the edge set E is a subset of $V \times V$, there are at most $49m^2$

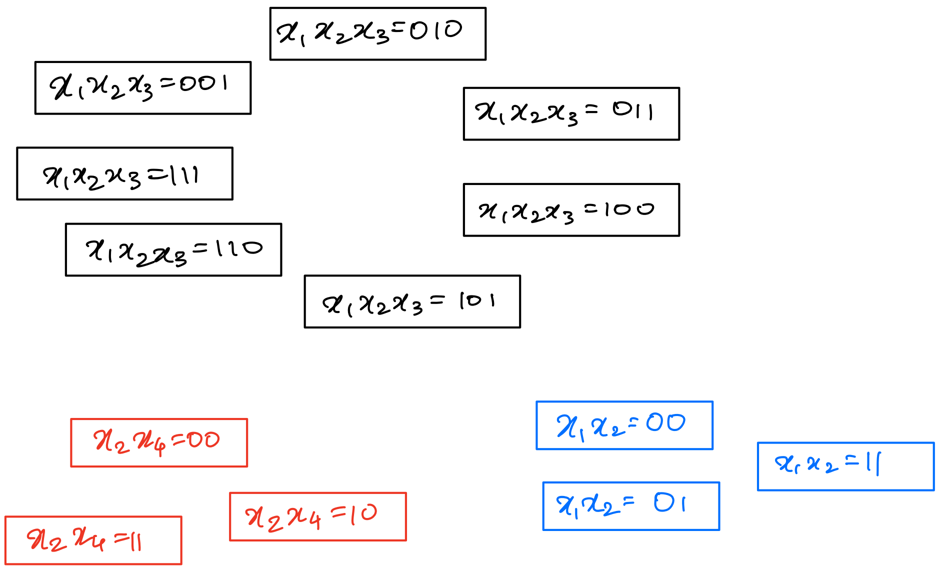

FIGURE 6.1: The nodes created in the process of reducing the example 3-CNF ϕ to INDSET.

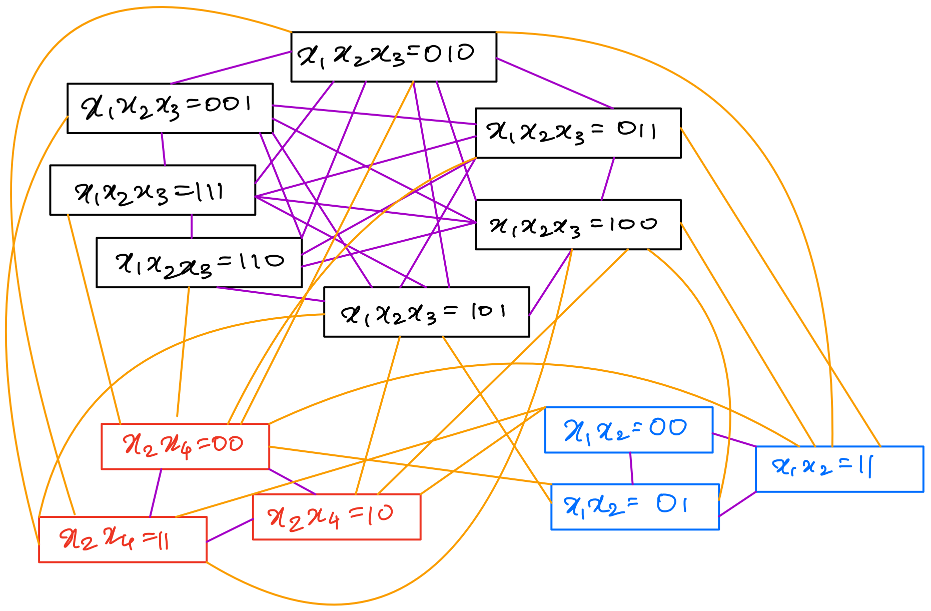

FIGURE 6.2: Some edges filled in between nodes with conflicting variable assignments. Purple edges highlight the full-connectedness within a cluster, yellow edges connect nodes across clusters.

edges in G. But moreover, determining the compatibility of the partial variable assignments of any two nodes is a constant time process — there are at most 6 variables to check. Thus, we can create the edge set E in $\mathcal{O}(6 \cdot 49m^2)$ time. Thus, the creation of G as a whole takes $\mathcal{O}(7m + 6 \cdot 49m^2) = \mathcal{O}(m^2)$ time.

Finally, we show that ϕ can be satisfied iff G has an independent set of size m.

Suppose ϕ can be satisfied. Then, there exists a complete variable assignment that satisfies every clause in ϕ. This complete variable assignment must induce for each clause a partial variable assignment that satisfies the clause. Moreover, across all m clauses, these partial variable assignments must be compatible with each other, as they come from a complete variable assignment. Since we created a node for each satisfying assignment for each clause, we must therefore be able to find m distinct nodes that each correspond to and satisfy a distinct clause. Since these nodes all correspond to compatible variable assignments, there will be no edges between any two of them. Thus, these nodes form an independent set of size m in G.

Now, suppose G has an independent set of size m. No two nodes in this independent set can be in the same cluster, as the clusters are fully connected. Thus, each node in the independent set corresponds a distinct clause in ϕ. As each node is generated from a satisfying assignment to the associated clause, we also get m partial variable assignments that each satisfy a distinct clause. Since these nodes form an independent set, there are no edges between them, and thus their partial variable assignments are compatible with each other. Thus, taking the complete variable assignment induced by the partial variable assignments, we satisfy each of the m clauses in ϕ. Thus there exists a satisfying assignment to the variables of ϕ.

Thus, as 3-SAT is **NP**-hard, and 3-SAT is polytime-reducible to INDSET, INDSET is **NP**-Hard. $\square$

It is interesting to note that G, as constructed in the above proof, cannot have an independent set of size $k > m$. There are m clusters in G, each corresponding to a different clause in ϕ. If we take a subset of the vertices of G of size $k > m$, then by the pigeonhole principle, at least two nodes would be in the same cluster. But if two nodes are in the same cluster, there must be an edge between them. Thus, this subset cannot be an independent set. Thus, ϕ is satisfiable iff the *maximal* independent set in G is of size m.

This note leads to the next section.

6.2 INDSET: Decision, Search, and Optimization

Consider the following previously-posed challenge problem.

Problem 6.4. Suppose there is a polynomial-time algorithm A that takes as input a graph G and an integer k and determines whether G has an independent set of size k. Use A to construct a polynomial-time algorithm that takes as input a graph and outputs an independent set of the maximum size in it.

This problem asks us to solve the *optimization* version of INDSET using a solver for the decision version. We will do this by recursively “disabling” nodes from being independent in the given graph G, letting us infer which nodes are necessarily present in some independent set. But importantly, this will show that the optimization version of INDSET is polytime-

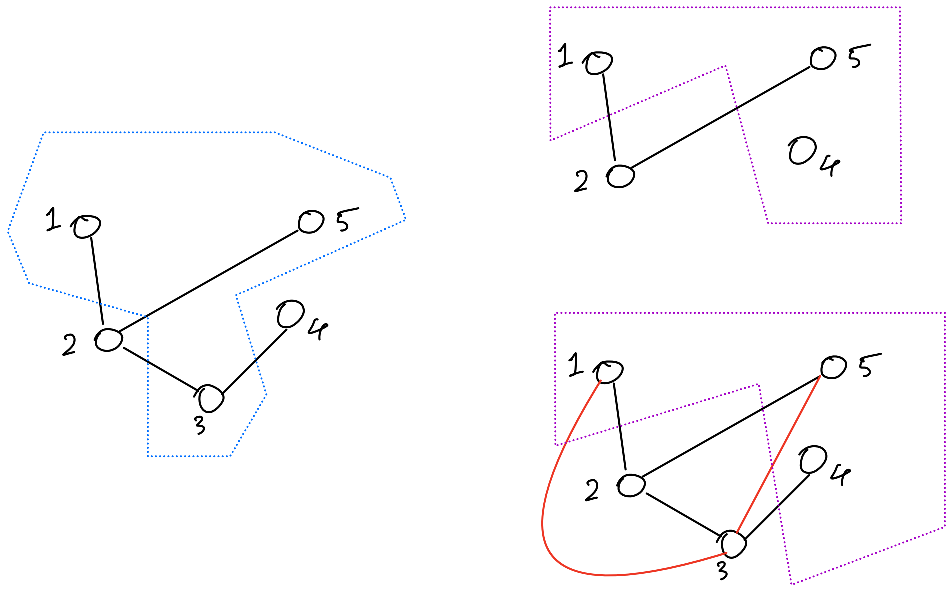

Figure 6.3: The two ways to "disable" a node. In the original graph on the left, we outline a maximal independent set. On disabling node 3 from the graph, either by removing it as in the upper figure, or connecting it to the nodes to which it was previously disconnected as in the lower figure, we can still find a maximal independent set of size 3 in this example.

reducible to the decision version, which will affect our later discussion on the relationship of the optimization version of problems to their decision and search versions.

Theorem 6.5. *The optimization version of INDSET is reducible in polynomial time to the decision version of INDSET.*

Proof. (due to Siddharth and Oliver) Let G be a given graph with n nodes. Let A be an algorithm used to solve the decision version of INDSET, i.e., for any integer $k, A(G,k) = 1 \iff$ there is an independent set of size k in G.

The key idea in the proof is that of "disabling" a node — making sure it cannot be in any independent set in G. Suppose we want to disable $u \in V$, where V is the vertex set of G. One way to do this is to connect u to every other node in V (in $\mathcal{O}(n)$ time). Then, any subset of V containing u and any other node cannot be an independent set, as there is necessarily an edge from u to that other node. Another way we could disable u is by removing from G all edges to or from u, and then removing u from V (also in $\mathcal{O}(n)$ time). Then, u is no longer in G and vacuously cannot be in any independent set in G. These two methods of disabling a node are illustrated in Figure 6.3.

If we know the maximum size m of independent sets in G, then by recursively disabling nodes in G, we learn which nodes *must* be in maximal independent sets. Specifically, if disabling u causes the maximum independent set size to decrease to $m-1$, u must be in all independent sets of size m in G. Otherwise, if the maximum independent set size did not decrease, then we can safely recurse on the new graph, knowing we will still be able to construct from it a maximal independent set of size m that does not contain u that was in G.

We now formally specify the reduction. Let $G_0 = G$. Then, to construct a maximal independent set in G we recursively apply the following procedure on the current graph G_i, where i starts at 0 (note that n_i is the number of vertices in G_i, which may change if we disable by removing nodes):

1. First, find the maximum size m of an independent set in G_i: for each k from 1 to n_i, run $A(G_i, k)$ until $A(G_i, k) = 0$; set m to the largest k such that $A(G_i, k) = 1$. This process incurs at most n applications of A.
2. If $m = 1$, return a singleton set containing any node in G_i (this is trivially a maximal independent set as no two nodes in the set contain an edge between them), and stop.
3. If $m = n_i$, return the entire vertex set, and stop.
4. Recurse on G_{i+1} to find a maximal independent set S in G_{i+1}.
5. If $|S| = m - 1$, then u must be in independent sets of size m in G_i. So, return $\{u\} \cup S$, a maximal independent set of G_i.
6. Else, $|S| = m$. Then, G_{i+1} has an independent set of size m. This independent set will also be an independent set of size m for G_i, as G_{i+1} is the same as G_i, except with one node disabled. So, return S.

This procedure will produce a maximal independent set for G. Note that this procedure recurses at most n times, each time applying A $\mathcal{O}(n)$ times and performing a disabling operation, which also takes $\mathcal{O}(n)$ time. Thus, the time complexity of this algorithm is $\mathcal{O}(n(nT(n) + n) = \mathcal{O}(n^2T(n))$, where $T(n)$ is the time complexity of running A once. Assuming $T(n)$ is polynomial in n, this procedure gives us a polytime algorithm for the optimization version of INDSET. $\square$

We conclude this section with the promised discussion of the relationship between decision, search and optimization versions of problems. First, note that the above procedure for solving the optimization version of INDSET also solves the search version of INDSET (given a graph G and an integer k, find an independent set of size k in G), using the decision version. First find a maximum size independent set S in G. If $|S| < k$, then output 0, indicating no independent set of size k exists in G. Else, take any k-length subset of S — this is necessarily an independent set of size k in G. Thus, both the optimization and search versions of INDSET are polytime-reducible to the decision version. Additionally, we have just reduced the search version to the optimization version.

Furthermore, if we have a solution to the search version of INDSET, we can easily solve the decision and optimization versions of INDSET. To solve the decision problem to check whether there exists an independent set of size k in G, just try to construct one. To solve the optimization problem, iteratively search for independent sets of different sizes until you cannot find one with greater size. Finally, the decision version is trivially reducible to the optimization version — output 1 if the maximum size independent set is not empty, else 0.

So, we can reduce any one of the search, decision, and optimization versions of INDSET to each other in polynomial time. In particular, since the decision version of INDSET is **NP**-Hard, both the search and optimization versions of INDSET are also **NP**-Hard. It is generally true that if the decision version of a problem is **NP**-Hard, then its optimization and search versions also are [1].

In the previous section, we noted there is a kind of equivalence between the search and decision versions of INDSET, in terms of the INDSET language. In particular, as explained in Lecture 4, INDSET is in **NP** because we can use a *candidate* independent set of size k as the certificate to a verifier for INDSET[3]. Notice that such a certificate is a solution to the search version of INDSET! This is precisely the equivalence between search and decision versions in **NP** — search versions are not in **NP** as they are not decision problems, but they are closely tied to the process of determining whether their decision version is in **NP**. In fact, for any language L in **NP**, the task of producing a certificate u for a given $x \in L$ is polytime-reducible to a decider for L, which naturally translates to the polytime-reducibility of search versions of problems to their decision versions [1].

However, the optimization version is generally more difficult. This is intuitive. In fact, the language MAX-INDSET introduced earlier is not in **NP**[1], therefore is not **NP**-Complete. However, it is important to note that if $P = NP$, i.e., if there were a polytime solver for the decision version of INDSET, then we could find a polytime solver for the optimization version of INDSET, as shown in Theorem 6.5.

The key takeaway of this discussion is that for the class **NP**, decision and search problems are "equivalent" in the sense that in order to prove a particular problem is in **NP**, you may need to solve the search problem; but as shown through INDSET, the search version is in general polytime-reducible to the decision version. Thus, as noted in earlier lectures, when studying problems in **NP**, it is sufficient to study only the decision version.

6.3 Majority and Parity: Limitations of CNFs and DNFs

Recall the parity and majority languages.

Definition 6.6. PARITY $= \{x \in \{0,1\}^* : x \text{ has odd parity}\}$. That is, x has an odd number of 1s.

Definition 6.7. MAJORITY $= \{x \in \{0,1\}^* : \text{greater than } \frac{|x|}{2} \text{ bits of } x \text{ are } 1\}$.

We bring up these languages to qualify the previous Cook-Levin theorem. This theorem only states that the computational power of Turing Machines equals that of Boolean CNFs — it says nothing of the time and space complexity required by these computational models.

In particular, deciders for PARITY and MAJORITY are trivial on Turing Machines. For PARITY, a Turing Machine can compute the XOR of every bit of an input x with the next, and return the result. For MAJORITY, a Turing Machine can count the number of 1s in an input x as well as the length of x, and return 1 if the number of ones is greater than half the length of x. These two solutions both take $\mathcal{O}(n)$ time and space complexity.

However, as we will show by solving the following challenge problems, any CNF or DNF that computes the parity function (i.e. decides PARITY) or the majority function (i.e. decides MAJORITY) must be of size exponential in the length of the input string. The solutions to both problems follow nearly the same procedure — reason about what happens when the terms of DNF do not specify enough variables.

Problem 6.8. Prove that the parity function on n bits requires DNF and CNF formulas of size exponential in n.

Proof. (due to Siddharth) Let $\phi = T_1 \vee T_2 \vee ... \vee T_m$ be a DNF that computes the parity function on n bits. Then each term must specify all n literals. To see this, suppose $n = 4$. Then, if one of the terms were $x_1 \wedge x_4$, then both 1001 and 1011 would satisfy this term and cause ϕ to output 1. But, 1001 does not have odd parity. More generally, if any term T_i in ϕ specifies $k < n$ literals, and if x is a string with odd parity, then flipping any one of the $n - k$ unspecified bits would create a string x' with even parity that still satisfies T_i, causing $\phi(x')$ to output 1. Since ϕ computes parity exactly, this is a contradiction, and so each term T_i must specify at least n literals. Now, if a term specifies more than n literals, by the pigeonhole principle, there is at least one repeated variable. Since each term is a conjunction, the repeated literals can be resolved into one if they are the same sign ($x_i \wedge x_i = x_i, \overline{x_i} \wedge \overline{x_i} = \overline{x_i}$), or the entire term is unsatisfiable, meaning it can be removed from the disjunction. Thus, without loss of generality, we can assume that every term in ϕ specifies exactly n variables.

Now since each term is a conjunction that specifies all n variables, it necessarily corresponds to a unique bitstring of length n with odd parity. That is, each T_i covers exactly 1 bitstring in PARITY. Therefore, there are at least as many terms in ϕ as there are bitstrings with odd parity. But there are 2^{n-1} bitstrings of length n with odd parity. Thus, there are 2^{n-1} terms in ϕ and the size of ϕ is $\Omega(2^n)$, exponential in n.

Now, let ψ be a CNF that computes the parity function. Then, look at the $\neg PARITY$ language, which is computed by $\neg\psi$. In particular, following De Morgan's laws, $\neg\psi$ is a DNF with the same number of terms as clauses in ψ. Now, $\neg PARITY$ is the set of all strings in $\{0,1\}^*$ with an odd number of 0s. This language has all the same properties as PARITY — any DNF for $\neg PARITY$ must have terms that specify exactly n variables, there are exactly 2^{n-1} bitstrings with an odd number of 0s, and so $\neg\psi$ has at least 2^{n-1} terms by the above reasoning. But this means that ψ has $\Omega(2^n)$ clauses. Thus the size of ψ is exponential in n. ∎

Problem 6.9. Prove that the majority function on n bits requires DNF and CNF formulas of size exponential in n.

Proof. (due to Siddharth) Let $\phi = T_1 \vee T_2 \vee ... \vee T_m$ be a DNF that computes the majority function on n bits. Then, each term must specify at least $\frac{n}{2}$ literals. Suppose that a term T_i specified $k < \frac{n}{2}$ literals. This term would require at most k bits in the string to be 1. Thus, there would exist an x that would have fewer than $\frac{n}{2}$ 1s but would satisfy T_i. Since ϕ is a disjunction, this would cause ϕ to output 1 on a bitstring that does not have the majority of its bits set to 1, a contradiction. Thus, every term in ϕ must specify at least $\frac{n}{2}$ literals.

Then, for each term, there remain at most $\frac{n}{2}$ unspecified literals. Each term thus covers at most $2^{n/2}$ satisfying assignments for ϕ. There are 2^{n-1} bitstrings of length n that satisfy the majority function. Thus, the number of terms in ϕ will be:

$$\frac{\text{total satisfying assignments}}{\text{assignments covered per term}} \geq \frac{2^{n-1}}{2^{n/2}} \text{ terms} = 2^{n/2-1} \text{ terms}$$

Thus, ϕ must have at least $2^{n/2-1}$ terms, and so has size $\Omega(2^{n/2})$, exponential in n.

Now suppose that ψ is a CNF for the majority function. Consider $\neg MAJORITY$, which is solved by $\neg\psi$. Following De Morgan's Laws, if ψ has m clauses, then $\neg\psi$ has m terms.

However, $\neg MAJORITY$ is the language of bitstrings $x \in \{0,1\}^*$ that have greater than $\frac{|x|}{2}$ bits set to 0. This language has all the same properties as MAJORITY — any DNF for $\neg MAJORITY$ must have terms that specify at least $2^{n/2}$ literals, $\neg MAJORITY$ has 2^{n-1} satisfying assignments, and so $\neg\psi$ has $\Omega(2^{n/2})$ terms, by the same above reasoning. But this means that ψ has $\Omega(2^{n/2})$ clauses. Thus, the size of ψ is exponential in n. $\square$

These two problems provide an important caveat for the Cook-Levin Theorem. While CNFs and DNFs can solve the same problems as Turing Machines, while they solve problems in constant time as constant-depth Boolean circuits, they cannot solve certain problems with an efficient amount of space, even if Turing Machines can.

6.4 The Universal Turing Machine

We now turn towards what Turing Machines *cannot* do, given certain constraints. As the reader likely knows, there are problems that are undecidable by Turing Machines. A more interesting result is the Time Hierarchy Theorem, which (informally) postulates that given more time, Turing Machines can decide more problems. The primary technique used to show these two results is known as *diagonalization*, which Georg Cantor developed to show that there is no bijection from any set to its power set. An equally essential tool in these proofs is the Universal Turing Machine, which can simulate any other Turing Machine exactly. This tool allows us to discuss sets of Turing Machines as languages in an alphabet, and thus to examine their properties via computation. The rest of this lecture focuses on the Universal Turing Machine, leaving the discussion of Undecidability and the Time Hierarchy Theorem for the next lecture.

In order for a Turing Machine to simulate another Turing Machine, it must be able to take that Turing Machine as input. But Turing Machines only take strings as input. Thus, we must show that Turing Machines are encodable as binary strings. Ultimately this is possible because there are only finitely many components of a Turing Machine.

Proposition 6.10. *Any Turing Machine is representable as a finite binary string.*

Proof. A Turing Machine M is a tuple of the form (k, Γ, Q, δ), where k is its number of tapes, Γ is its (finite) alphabet, Q is its (finite) set of states, and δ is its transition function. Note that $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, R, S\}^k$. k is fixed, Q and Γ are finite, so δ is a function from one finite set to another. We can write k in binary, as it is an integer. Any finite set S can be enumerated in binary using $\log |S|$ bits, so Q and Γ can be encoded in binary. We can write out δ as a long sequence of (input, output) pairs, each input being assigned a unique encoding in $\log |Q \times \Gamma^k|$ bits, each output being assigned a unique encoding in $\log(3^k|Q \times \Gamma^{k-1}|)$ bits (there are 3^k elements in $\{L, R, S\}^k$). Finally, we can choose some scheme for delimiting different elements of the tuple (such as a blank ␣), and for delimiting different elements of a set (such as two blanks ␣␣), and then write out k, then every element of Γ, then every element of Q, then every (input, output) pair of δ in the binary encodings we have assigned. This is clearly a finite string, as each component of M is encoded with a finite number of bits, and there are only finitely many things to write out. Thus any Turing Machine M is representable as a finite binary string. $\square$

Following this result, we introduce 2 conventions:
Convention: Every Turing Machine is representable by infinitely many strings. We ignore

leading 0s. More formally, a string representation α of a Turing Machine M must start with 1, and all strings of the form $0^n\alpha$ represent the same Turing Machine M.

Convention: Every binary string α encodes some Turing Machine. Such strings can be thought of as programs that encode the Turing Machine. If such a program is not syntactically valid, then it corresponds to the Turing Machine that halts immediately and outputs 0. This machine does not look at its input — the moment it starts, it terminates instantly. We denote by M_α the Turing machine represented by the string α.

The following corollary will be quite useful for diagonalization. In fact, without countability, the results of diagonalization would be much weaker. We would not be able to show with complete certainty that a particular problem is not solvable by a specific class of Turing Machine.

Corollary 6.11. *The set of Turing Machines is countably infinite.*

Proof. For every Turing Machine M there is a binary string α that encodes M. By our second convention, for every string α, there is a Turing Machine M that α encodes. Thus, the set of Turing Machines can be mapped bijectively to the set of binary strings, $\{0,1\}^*$. However, $\{0,1\}^*$ is countably infinite:

$$\{0,1\}^* = \bigcup_{n\in\mathbb{Z}_{\geq 0}}^{\infty} \{0,1\}^n$$

where $\{0,1\}^n$ is the set of bitstrings of length n. Each $\{0,1\}^n$ is a finite set, containing 2^n strings. There are countably many n that we union over, so $\{0,1\}^*$ is a countable union of finite sets, which is countably infinite.

Thus, the set of Turing Machines is countably infinite. ∎

Finally, we are ready to specify the Universal Turing Machine.

Theorem 6.12. *There exists a Universal Turing Machine U which:*

- *takes as input a string $\alpha \in \{0,1\}^*$, and a string $x \in \{0,1\}^*$ (α is a program that encodes another Turing Machine, and x is data for that program)*
- *computes $M_\alpha(x)$ and writes the output of $M_\alpha(x)$ onto its output tape*
- *terminates within $c_\alpha(|x| + t\log t)$ where t is the worst-case running time of $M_\alpha(x)$ and c_α is a constant that depends only on α, not on x*

The specification we develop does not introduce any fundamentally new material. Even in arguing that the overhead in time is at most logarithmic, we rely on our previous result that any k-tape Turing Machine can be simulated with 2 tapes with at most logarithmic overhead [3]. This underscores how natural the idea of a Universal Turing Machine is.

Proof. We construct the Turing Machine U as follows. It has an "input" tape to store the data string x. It has an "encoding of TM" tape to store the encoding α of the Turing Machine to simulate. It has a "current state of M_α" tape to record the current state of the simulated Turing Machine during execution. Finally, it has a set of work tapes, and an output tape. A schematic of U's tapes is shown in Figure 6.4.

U operates as follows:

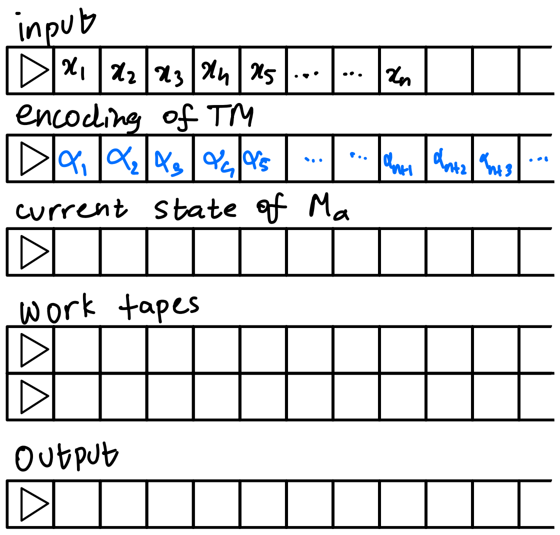
input
x_1 x_2 x_3 x_4 x_5 ... ... x_n
encoding of TM
α_1 α_2 α_3 α_4 α_5 ... ... α_{n+1} α_{n+2} α_{n+3} ...
current state of M_α
work tapes
output

FIGURE 6.4: The Universal Turing Machine U, with inputs α and x written on their tapes.

1. Check that α is a syntactically valid encoding of some Turing Machine M_α. If α is not valid, write 0 on the output tape, and halt. Note that α may be some incredibly large string, so checking it for validity may take an exorbitant amount of time. Still, this induces only constant overhead dependent only on α.

2. Initialize the "current state of M_α" tape to hold the starting state of M_α.

3. Re-encode x using M_α's alphabet. This must be done to ensure the correct simulation of M_α's transition function. Note that the time to re-encode each character of x is a constant dependent only on M_α's alphabet, which is specified by α. Thus, this step takes $c_\alpha|x|$ time for some c_α dependent only on α.

4. To simulate one step of M_α
   - Scan M_α's transition table (M_α's transition function encoded as a table of (input, output) pairs, as in the proof to Proposition 6.10) for the next action.
   - Update the "current state of M_α" as well as work tapes based on the action, encoding characters using M_α's alphabet; write on the output tape what M_α would have written on its output tape.
   - Repeat until M_α's state is HALTED, then halt.

The final step of U involves a constant overhead to simulating every step of M_α's computation — specifically in scanning M_α's transition table, and in writing characters on the tape using an encoding of M_α's alphabet. However, once again, this slowdown is constant, dependent only on α. Moreover, if M_α has k tapes and runs in t steps, then it can be simulated with $t\log t$ steps using 2 tapes. Following this result, U needs only 2 work tapes, and the final step takes $c_\alpha t\log t$ time.

Thus, the overall time complexity of U in simulating M_α is $c_\alpha(|x| + t \log t)$.

The key reason why U can simulate any M_α is precisely because M_α can be encoded as a string α. Keeping track of M_α's would-be current state as a string, and looking up in a table written on a tape what M_α is going to do next, U can simply do what M_α would do. This makes U quite simple. $\square$

To conclude our discussion of the Universal Turing Machine, we posit the existence of an efficient Nondeterministic Turing Machine. First, note that Nondeterministic Turing Machines can be encoded in binary, just like Deterministic Turing Machines — you simply have one more transition function to encode.

Theorem 6.13. *There exists a Universal Nondeterministic Turing Machine U which:*

- *takes as input $\alpha \in \{0,1\}^*$, and $x \in \{0,1\}^*$*
- *nondeterministically computes $M_\alpha(x)$, where M_α is now the Nondeterministic Turing Machines specified by α*
- *halts in $c_\alpha(|x| + t)$ timesteps, where t is the worst-case running time of $M_\alpha(x)$, and c_α is a constant dependent only on α, not x.*

In particular, due to the absence of logarithmic or quadratic overhead in time complexity, this theorem suggests that such an efficient Universal Nondeterministic Turing Machine is able to simulate any other Nondeterministic Turing Machine with just 1 tape. The proof of Theorem 6.13 is left as a challenge problem.

6.5 Challenge Problems

Problem 6.14. Prove that the halting problem is **NP**-Hard.

Problem 6.15. Prove Theorem 6.13, that there is an efficient universal nondeterministic Turing machine U. Specifically, U takes as input the encoding $\alpha \in \{0,1\}^*$ of nondeterministic Turing machine and a string $x \in \{0,1\}^*$ and nondeterministically computes $M_\alpha(x)$ in time at most $c_\alpha(|x| + t)$ where t is the worst-case running time of $M_\alpha(x)$.

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[2] Y. Kou. Deterministic computation, 2024. Lecture 3 Scribe Notes.

[3] I. Shkirko. Universal computation and intro to nondeterminism, 2024. Lecture 4 Scribe Notes.