CS 517
Homework 2

Ameyassh Nagarajan

April 2024

1 Problem 1

Theorem 1 *It is undecidable to determine whether a given context-free grammar (CFG) is ambiguous.*

Determining whether a given context-free grammar (CFG) is ambiguous is undecidable.

The proof of this theorem is based on reducing the Post Correspondence Problem (PCP), a well-known undecidable problem, to the problem of determining CFG ambiguity.

Understanding PCP: The PCP involves finding whether there exists a sequence of indices for two given lists of strings, $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$, such that the concatenation of the strings from U using these indices equals the concatenation of the strings from V using the same sequence of indices. Formally, the challenge is to find if there exists a sequence $(i_1, \ldots, i_k)$ such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$.

CFG Construction: We construct a CFG G designed to mimic the structure needed to represent possible solutions to PCP. The grammar G includes:

- Non-terminal symbols $S, A_1, \ldots, A_m, B_1, \ldots, B_m$.
- Terminal symbols corresponding to each unique character found in the strings u_i and v_i.
- Production rules:

$$\begin{aligned} &S \to A_i S B_i \text{ for each } 1 \leq i \leq m, \\ &A_i \to u_i \text{ for each } 1 \leq i \leq m, \\ &B_i \to v_i \text{ for each } 1 \leq i \leq m, \\ &S \to \epsilon. \end{aligned}$$

These rules allow the grammar to generate sequences where any chosen string from U can be followed by its counterpart from V in reverse order, potentially interspersed with other such pairs, creating a palindromic structure if matched perfectly.

Link to PCP Solution: If a solution to PCP exists (i.e., a sequence $(i_1, \ldots, i_k)$ such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$), the CFG G constructed above will be able to generate the string $u_{i_1} \cdots u_{i_k} v_{i_k} \cdots v_{i_1}$ in at least two ways:

1. Directly through the use of $S \to A_{i_1} S B_{i_1}, \ldots, A_{i_k} S B_{i_k}$ followed by $S \to \epsilon$.
2. Potentially through another path if segments of U and V produce the same concatenation via a different sequence of productions.

Implications: The existence of multiple parse trees for any string in the language of G signifies ambiguity in G. Therefore, if we could determine whether G is ambiguous, we would effectively solve the PCP by constructing G and checking for its ambiguity. Since PCP is undecidable, the problem of determining whether G is ambiguous is also undecidable.

This reduction shows that the undecidability of PCP directly implies the undecidability of determining CFG ambiguity.

2 Problem 2

Theorem 2 *The language $L = \{\langle M \rangle \mid M$ is a Turing machine that has polynomial worst-case running time$\}$ is undecidable.*

Assume for the purpose of contradiction that L is decidable. This implies the existence of a Turing machine D which decides L. That is, given a description of a Turing machine $\langle M \rangle$, D determines whether M operates within polynomial worst-case running time for all inputs.

We will show that this assumption leads to a contradiction by demonstrating that if such a D existed, we could solve the Halting Problem, which is known to be undecidable. The Halting Problem asks whether a given Turing machine M halts on a given input w.

Let's construct a new Turing machine M' based on any Turing machine M and input w:

1. On input x, M' simulates M on w.
2. If M halts on w, then M' enters a loop that will run for $|x|^k$ steps for some fixed k, before halting.
3. If M does not halt on w, then M' also does not halt.

We now use D to decide if $\langle M' \rangle \in L$. If D accepts, then by the construction of M', M must halt on w. If D rejects, then M does not halt on w. This decision procedure effectively solves the Halting Problem using D, which is a contradiction because the Halting Problem is undecidable.
Therefore, our initial assumption that L is decidable is false, and the language L is undecidable.

3 Problem 3

Theorem 3 *If $P = NP$, then for any language L in NP characterized by a witness-checking algorithm R, there exists a polynomial-time algorithm M such that, on input x in L, $M(x)$ outputs a valid witness for x, and on input $x \notin L$, $M(x)$ outputs the string "no witness".*

Let L be an arbitrary NP language with a witness-checking algorithm R, such that $L = \{x \mid \exists w : R(x, w) = 1\}$. Assume $P = NP$.

Define a language B as follows:

$$B = \{\langle x, y\rangle : \exists z \in \Sigma^* \text{ such that } |yz| \leq |x|^k + c \text{ and } R(\langle x, yz\rangle) \text{ accepts}\}.$$

Here, $k, c \in \mathbb{N}$ are constants ensuring that any string $x \in L$ has a witness w of size at most $|x|^k + c$.

Since $P = NP$, and we have constructed B such that it is in NP, B must also be in P. Therefore, there exists a polynomial-time algorithm M_B that decides B.

We now construct a polynomial-time Turing machine M which, given an input x, produces a witness w for x if $x \in L$, and outputs "no witness" otherwise.

Machine M operates as follows:

1. On input x, run M_B on $\langle x, \epsilon\rangle$ to determine if $x \in L$.
2. If M_B rejects, output "no witness".
3. If M_B accepts, incrementally construct the witness w by testing each bit extension using M_B.
4. For each bit extension, run $M_B(\langle x, w'\rangle)$ where w' is the current prefix of w, extended by one bit (0 or 1).
5. When a prefix w' is found for which M_B accepts, adopt this prefix and extend the next bit.
6. Continue this process until the full witness w is constructed, such that $R(\langle x, w\rangle)$ accepts.

This construction ensures that M runs in polynomial time, as it only makes a polynomial number of calls to M_B and the input size for each call to M_B and R is at most $O(|x|^k)$. Hence, we can conclude that if $P = NP$, not only can we decide membership in L in polynomial time, but we can also find a valid witness in polynomial time.

4 Problem 4

4.1 Part a

Theorem 4 $NP = \{L \mid L \leq_p SAT\}$

Proof for $L \in NP \implies L \leq_p SAT$

Assume L is a language in NP. By definition, there exists a nondeterministic Turing machine (NTM) that decides L in polynomial time. We need to construct a polynomial-time reduction from L to SAT:

1. Given an input x for L, construct a Boolean formula Φ that represents the computation of the NTM on x.
2. Variables in Φ represent the states, the tape content, and the head positions at each step.
3. Clauses in Φ enforce the legality of the transitions according to the machine's transition function.
4. Φ is satisfiable if and only if there is a sequence of transitions leading to an accepting state of the NTM.
5. The reduction from x to Φ can be done in polynomial time since the NTM operates in polynomial time, and the size of Φ is polynomially related to the size of the computation.

Therefore, $x \in L$ if and only if Φ is satisfiable, implying that $L \leq_p SAT$.

Proof for $L \leq_p SAT \implies L \in NP$

Now assume $L \leq_p SAT$. This implies there exists a polynomial-time computable function f such that for any string x, $x \in L$ if and only if $f(x)$ is satisfiable.

Given that SAT is in NP, there exists a nondeterministic polynomial-time Turing machine M_{SAT} that decides SAT. Since f is computable in polynomial time, use M_{SAT} to decide L as follows:

1. On input x, compute $f(x)$.
2. Simulate M_{SAT} on $f(x)$.
3. Accept x if M_{SAT} accepts $f(x)$; otherwise, reject x.

This procedure shows that L can be decided by a nondeterministic Turing machine in polynomial time, thus $L \in NP$.

Conclusion

Both directions have been proven:

- $L \in NP$ implies $L \leq_p SAT$, showing every language in NP can be reduced to SAT in polynomial time.
- $L \leq_p SAT$ implies $L \in NP$, showing any language that can be reduced to SAT in polynomial time is in NP.

Therefore, we conclude $NP = \{L \mid L \leq_p SAT\}$, providing an alternative definition of NP.

4.2 Part b

Theorem 5 *A language L is NP-complete if and only if its complement $\overline{L}$ is coNP-complete.*

We will prove the theorem in both directions.

($\Rightarrow$) Assume that L is NP-complete. By definition, L is in NP, and every language M in NP has a polynomial-time reduction to L. Because L is NP-complete, it is also NP-hard, which means every language N in NP is polynomial-time reducible to L. For the complements of these languages in NP, which are in coNP, this implies that if $N \in NP$, then $\overline{N} \in coNP$.

Given a language $\overline{N}$ in coNP, it polynomial-time reduces to $\overline{L}$. If N reduces to L, then a 'yes' instance of N translates to a 'yes' instance of L, and a 'no' instance of N translates to a 'no' instance of L. For the complements, a 'yes' instance of $\overline{N}$ corresponds to a 'no' instance of N, and thus to a 'yes' instance of $\overline{L}$. Therefore, $\overline{L}$ is coNP-hard. Since L is in NP, $\overline{L}$ is in coNP, and hence $\overline{L}$ is coNP-complete.

($\Leftarrow$) Now assume $\overline{L}$ is coNP-complete. Similarly, every language $\overline{M}$ in coNP can be reduced in polynomial time to $\overline{L}$, and therefore every language M in NP is reducible to L because the complements of languages in coNP are in NP. Thus, L is NP-hard. Given that $\overline{L}$ is in coNP, L must be in NP. Hence, L is NP-complete.

This proves that L is NP-complete if and only if $\overline{L}$ is coNP-complete.