CS 40: Computational Complexity

Sair Shaikh

September 29, 2025

Problem 3. Consider the language

$$\text{SATISFIES} = \{\langle \varphi, \alpha \rangle : \varphi \text{ is a CNF formula and the assignment } \alpha \text{ satisfies } \varphi\}.$$

For concreteness, assume the following encoding for $\langle \varphi, \alpha \rangle$.

- There is an explicit separator '#' between the φ and the α portions.
- The string $\alpha \in \{0,1\}^m$, where m is the number of variables appearing in φ.
- The formula φ is encoded over the alphabet $\{0, 1, \vee, \wedge, \neg, (,)\}$ by enclosing each clause in parentheses and indicating a variable x_i by writing the subscript i in binary. Thus, for instance, the formula

  $$(x_2 \vee x_5) \wedge x_3 \wedge (x_3 \vee x_1)$$

  is represented as the string

  $$(10 \vee \neg 101) \wedge (11) \wedge (\neg 11 \vee \neg 1)$$

Our proof that SAT $\in$ NP boiled down to showing that SATISFIES $\in$ P. Prove the stronger result that SATISFIES $\in$ L.
This problem is all about careful implementation, so take care to provide enough detail, i.e., precisely what information you will store on the work tape of your TM.

Solution. At a high-level, the algorithm is as follows:

Scan clauses left to right. In the state, store the answers to the questions "Is the current clause being scanned correct", "have finished checking clause", "Is the machine in scan-clause or read-literal mode" (we will also store other things). If you read an $\wedge$ symbol with the current clause incorrect and have finished checking clause, REJECT. If you read the # symbol, are in scan-clause mode, have finished checking clause, REJECT if current clause incorrect, ACCEPT if current clause correct.

Before describing the algorithm in detail, lets assume that a turning machine can maintain a counter (in binary) on its work-tape. That is, it can increment it, decrement it, and zero it out. We will store the counter least-significant-bit first, so that we have room to count up. This is weaker than assuming binary addition and subtraction are in L.

The algorithm is the following:

1. Start in some special initialization state. Move forwards until you read #. Then, continiue moving forwards. If the symbol read is 0 or 1, increment a counter on the work-tape; if the symbol is the special symbol corresponding to end-of-input, decrement the counter and write the special character b on the work-tape in the first cell not occupied by the counter. The b corresponds to a boundary between sections of the tape. Thus, now we have the number $m-1$ stored in binary on the work-tape.
   This requires a finite number of states, as there are a finite number of things to keep track off, i.e. "is in initialization mode", "haven't found # yet", "have decremented counter", "is the b written", etc. I will assume this level of detail is not necessary for future steps.

2. Clear the counter by replacing all bits before the b with 0. Now, we have allocated space to count up to m. We have used $1 + \log(m)$ space.

3. Write a 0 after b on the work-tape to initialize a new counter which will help track of the head-position within φ (for when we want to pause scanning and go check a literal). This counter will count the number of $\vee$ and $\wedge$s read so far, thus this counter will not be bigger than log of the number of $\vee$ and $\wedge$ in φ, thus less than $\log(\langle\varphi\rangle)$. We also move work-tape head back to the start.

4. Go back to the start of the input. Switch to state 11, where the first bit encodes whether we are in scan-clause mode, and the second bit encodes the bit value we want the next character checked to contain to make the current clause true.

5. While in scan-clause mode, do the following:

   - If you read a $\neg$, switch to 10 as you now want the next literal checked to be 0 to make the current clause true. Move forward.
   - If you read a 0 or 1, copy it to the work-tape (work-tape head is at the start). Move both input head and work-tape forward by 1.
   - If you read a (or), move forward. This will copy the index that we want to read in the first counter.
   - If you read a $\vee$ or $\wedge$, increment the second counter (after b) by one and switch to "read-literal" mode. We have finished reading and index and are going to go check it.

6. While in read-literal mode, do the following:

   - Move forwards to #.
   - Move forwards and decrement the first counter, counting to the index you want to read.
   - If the first counter is 0 (i.e. we've reached the literal we want to read), check if the bit at the current head position matches the third bit of the state (the bit that will make the clause correct).
   - If the check fails, return both heads to the start of their respective tapes and go into a "reset with fail" state.
   - If the check passes, return both heads to the start of their respective tapes and go into a special "reset with pass" state.

7. While in either of the reset states, do the following:

   - First, copy the contents of the second counter to a third counter. This will require storing a fourth counter to keep track of how much of the second counter has been copied over. The third counter matches the size of the second, and the fourth is logarithm in the size of the second counter, thus we have still only used at most space logarithmic in the input.
   - Zero out the fourth counter once done and change state to indicate end of the copying phase.
   - Scan forwards on the input tape, reading as you go. Decrement the third counter if $\wedge$ or $\vee$ is read.
   - If the third counter is all 0s, and you were in "reset with pass":
     - If the current character is $\vee$, keep reading till the next $\wedge$ is detected, incrementing the 2nd counter appropriately. If there are no more $\wedge$s, and end of string is reached, we've verified all clauses, thus, ACCEPT.
     - If the current character is $\wedge$, do not do this part.

     Now that you're at the start of the next clause, having verfied the previous one, switch to 11 (i.e. scan-clause mode, looking at a not-yet negated literal).
   - If the third counter is all 0, and you were in "reset with fail":
     - If the current character is $\vee$, move forward and switch to 11 (i.e. scan-clause mode, looking at a not-yet negated literal) as we have not yet proved this clause.
     - If the current character is $\wedge$, that means we've finished checking a clause and its not true, thus, REJECT.

Thus, this TM implements SATISFIES using only 4 counters, which are all at most logarithmic in the input size, we have shown that SATISFIES $\in$ L.

Problem 4. Consider the language

$$\text{ALLNFA} = \{\langle \Lambda, M \rangle : M \text{ is a NFA over } \Lambda \text{ such that } L(M) = \Lambda^*\}$$

Note that the alphabet Λ is specified as part of the encoding of the NFA M.
Prove that ALLNFA is PSPACE-complete.
Hint: This problem is challenging. While reducing from TQBF may be tempting as an approach, it may be a better idea to carefully study the proof of Sipser 5.13 and try to adapt that.

Solution. To show that ALLNFA is PSPACE-complete, we need to show that ALLNFA is in PSPACE and is PSPACE-hard.

First, let's show that ALLNFA is in PSPACE. Let N be an NFA over Λ with states Q, accept states F, start state q_0, and transition function δ. From this information, we define a graph as follows:

- The vertices are subsets of Q, i.e. correspond to elements of the power-set.
- There is an edge between v and u, if there exists states $A \in v$ and symbol $\lambda \in \Lambda$ such that $\delta((A, \lambda)) = u$. That is, if there is a way to transition from any of the states in v to any of the states in u by reading one character.

Now, $L(N) \neq \Lambda$ (it accepts all strings) if and only if there is a path from the vertex corresponding to $\{q_0\}$ to any vertex v where $v \cap F = \emptyset$ (i.e. v contains only reject states). Given a path, we can construct a string using the sequence of characters that generated those vertices. Given a rejected string, we can come up with such a path by construction of our graph.

Thus, our problem has been translated into a finite set of reachibility problems, i.e. if any of the vertices containing only reject states are reachable, $\langle \Lambda, N \rangle$ is not in ALLNFA.

We know the reachibility problem is NP in the number of vertices, thus exponential in the the size of the input, if we can generate edges in PSPACE. However, we can do this by constructing a turning machine that goes over the encoding of the transition function of δ, stops at a random mapping, and outputs an edge based off of that mapping. Thus, ALLNFA is in $PSPACE$.

Next, let's show that ALLNFA is PSPACE-hard. Let L be a language in PSPACE. Then, by the definition of PSPACE, there exists a deterministic turning machine T and positive integer k, positive real a such that $\text{SPACECOST}_M(n) \leq a \cdot n^k$. For any input x to T, it suffices to construct an NFA N_x over some Λ_x such that:

$$\langle \Lambda_x, N_x \rangle \in \text{ALLNFA} \iff L(N_x) = \Lambda_x^* \iff x \notin L$$

(i.e. deciding $x \in L$ can be reduced to checking if $\langle \Lambda_x, N_x \rangle \in$ ALLNFA and flipping the answer.)

We will construct N_x to accept every string that is not a valid accepting computation history starting with x. We pick Λ to contain all the characters needed to encode a finite computation path for T in the format:

$$C_0 \# C_1 \# \cdots \# C_n$$

(i.e. all T states, symbols of the alphabet, #, head symbol, etc). We can pick this independent of x, even though we do not need to.

Note that each configuration C_i is polynomial length in $|x|$ as the work-tape contents are bounded in length by $a \cdot |x|^k$, the input contents are equal to $|x|$, the number of states is finite (hence finitely encodabel), and we can track the heads by adding special characters at the correct places in the encodings of the tape contents. Keeping this in mind, we design our NFA as follows:

There are three ways the string can fail to be a valid computation path. First, we guess amongst the these three options:

1. Checking if C_0 is not the valid starting configuration.
2. Checking if C_n is a not a valid accepting configuration
3. Checking if some pair C_i, C_{i+1}, is not a valid transition.

If we guess (1), the NFA proceeds to check if C_0 is a valid configuration. Note that the starting configuration is known, i.e. it has x on the input tape, empty work-tape both heads at position 0, and is in the starting state. Thus, we can non-deterministically guess an index by making a binary choice while consuming the configuration. Then, check whether that index agrees with the start configuration, and accept if it doesn't. Since $|C_0|$ is polynomial in $|x|$, the number of states needed to count up to the picked index (distance from #) will be constant (log of a polynomial).

If we guess (2), we can non-deterministically guess some # (make a binary choice at that point) to be the one preceeding the final configuration. Then if the state symbol in this configuration is not one of the accept states of T, we scan the rest of the input to verify the # we guessed in fact corresponded to the final configuration in the computation path (i.e. there should no more # after it). This needs a constant number of states.

If we guess (3), we can non-deterministically guess some # (make a binary choice at that point) to pick a configuration i. Since Turing machine configuration changes are local, using ideas similar to in the proof for Cook-Levin, it suffices to check a small window (size 3) of

C_i against C_{i+1}. Thus, non-deterministically guess an start index (binary guess as you parse through or whether to pick or not pick), and compare the window after that start index to the corresponding window in the next state. If the transition is not valid, accept. Here, valid means identical for all positions away from the work-tape head and consistent with the transition function around it. Similar to (1), this can also be done in a number of states polynomial in $|x|$ (i.e. on the order of the configuration length).

Then, if $x \in L$, there exists a string $s \in \Lambda^*$ that represents a valid computation path and will not be accepted by N_x. Thus, $L(N_x) \neq \Lambda^*$ and thus $\langle \Lambda, N_x \rangle \notin$ ALLNFA.
Conversely, if $\langle \Lambda, N_x \rangle \in$ ALLNFA, then $L(N_x) = \Lambda^*$. Thus, N_x accepts all strings. Thus, no strings are valid computation paths leading to an accept result from T on x. Thus, $x \notin L$.
Together, these show:

$$\langle \Lambda, N_x \rangle \in \text{ALLNFA} \iff x \notin L$$

Finally, we need to show that this can be done in polynomial space. Note that:

- Λ has a constant size and can be written down by going over all states, alphabet symbols of T, etc, which requires constant work-space, as all of these have sets have fixed size.
- The number of states of N_x is polynomial in $|x|$ as its the union of the states needed by each of the 3 paths.
- Each non-deterministic guess we make is between 2 or 3 choices.
- For most of the transitions in the transition function, there is no dependence on x. That is, fixing a T and having its instruction set, we can produce most of the transition function for N_x without reading x. For instance, C_i and C_{i+1} can be compared without knowing x as its either equality testing or verification using the T instruction set. Thus, this can be done with constant amount of work-space.
- For checking the input configuration, however, we need to reference the contents of x. However, the transition function of N_x corresponding to this bit depends on x one symbol at a time (i.e. checking the ith symbol of the encoding of the input tape in C_0 depends only on the ith bit of x). Thus, this can be done in log-space, to maintain a counter for what bit of x we are reading.

Using all of these, we know that the transition function has a polynomial number of distinct relations. More concretely, let Q be the set of states and the transition function be $\delta : Q_x \times \Lambda \to P(Q_x)$ where P is the power-set. Then, there are a polynomial number of inputs, as $|Q|$ is polynomial in $|x|$ and $|\Lambda|$ is fixed finite. Thus, if we output the NFA as delimiter seperated alphabet, states, accept states, and transition function, the counters we need to maintain are constant sized for the first 3 and $\log(|Q_x \times \Lambda|)$ which are all in PSPACE.

Overall, this means the reduction can happen in PSPACE. Thus, ALLNFA is PSPACE-hard. Thus, it is PSPACE-complete.