CS 361 Meeting 32 — 5/13/20

Is it Hard to Decide if a Finite Automaton Accepts All Strings?

(Click for video)

1. In the last segment, we considered the classes of problems associated with Turing machines that operate using some limited amount of space/tape cells.

   Definition: Let $s : \mathcal{N} \to \mathcal{R}^+$ be a function (that increases at least linearly with its input). Define the **space complexity class,** $SPACE(s(n))$ to be the collection of all languages that are decidable by a Turing machine using $O(s(n))$ distinct tape cells on inputs of size n.

2. In particular, we looked at the set of language decidable by a Turing machine using space limited by some polynomial bound.

   Definition: PSPACE is the class of language that are decidable in polynomial space on a deterministic single-tape Turing machine. In other words

   $$PSPACE = \bigcup_k SPACE(n^k)$$

3. We also defined the class of languages decidable by nondeterministic Turing machines in polynomial space:

   Definition: NPSPACE is the class of language that are decidable in polynomial space on a nondeterministic single-tape Turing machine. In other words

   $$NPSPACE = \bigcup_k SPACE(n^k)$$

Click here to view the slides for this class

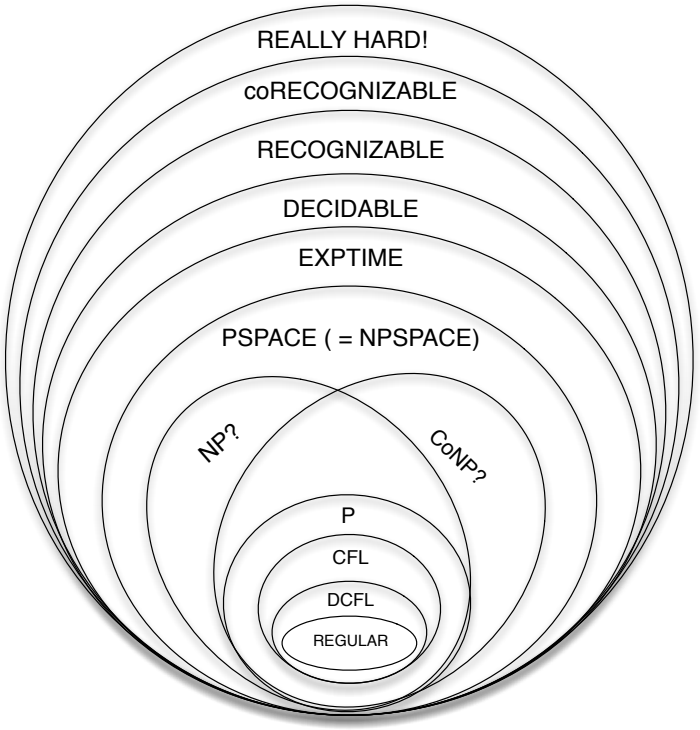
4. We then explained Savitch's Theorem which shows that we don't need to consider NPSPACE since in fact NPSPACE = PSPACE.

   The key to this result was to explore for ways in which a Turing machine could move from a given initial state to some final state recursively.

```
for every accepting configuration f {
    if ( canReach( initial configuration, f, 2^s(n) ) ) {
        accept
    } }
reject
```

```
canReach( start, end, steps) {
if steps == 1 {
    return start == end
} else if steps == 2 {
    return ( start yields end )
} else {
    for every mid ∈ configurations {
        if ( canReach(start, mid, steps/2) &
                canReach(mid, end, steps/2) ) {
            return true
        }
    }
return false
}
```

5. This brought us a step closer to an accurate map of the classes of languages we have discussed this semester:



I say closer because this map is still uncertain. It is known that

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

and that

$$P \neq EXPTIME$$

so we know that one of the $\subseteq$s on the first line must be a $\subsetneq$, but we don't know which one.

6. Just as we suspect P $\neq$ NP, it seems likely that NP is a proper subset of PSPACE (even if P = NP). If we want to show that this is true, our best hope is to look at the hardest problems in PSPACE and show that one of these problems requires more than polynomial time.

7. With this in mind, let's revisit a language we considered weeks ago.

$$ALL_{DFA} = \{\langle A \rangle \mid A \text{ is a DFA and } L(A) = \Sigma^*\}$$

8. Earlier, we argued that this language is decidable (remember?).

9. Of course, now we want to think of this question about DFAs in the context of complexity of Turing machine algorithms by asking whether ALL_{DFA} belongs to any (or all) of the time and space complexity classes we have just discussed:

   - PSPACE, NP, CoNP, P, CFL, etc.

10. This is an easy one to warm you up! A DFA will reject some string iff there is a path in the graph of states derived from the transition function from the machine's start state to some non-final state. So, we just make sure there are non-final states and then run something like Dijkstra's algorithm to see if there are any paths from the start state to any of the non-final states. If so (or if the input is not a valid encoding), reject $\langle A \rangle$. Otherwise, accept. All this can be done in polynomial time (for a small polynomial).

11. When we discussed ALL_{DFA} a few weeks ago, we actually considered an alternative to the graph search algorithm that was a bit less clever, but quite simple.

    - The idea was to just "check all strings" over the alphabet to make sure they were all accepted.
    - Actually, checking all strings would clearly not be practical, but we argued that the pumping lemma gave us an easy way to limit the search.
    - Since we can easily take a DFA and switch its final and non-final states to obtain a DFA for its complement, we can in some sense apply the pumping lemma to strings not in the language.
    - That is, if w is not in the language of the DFA and is of length greater than the number of states, we know that there is some substring of w that we can pump (up or) down and always obtain another string not in the language.

- Based on this, we know if there is any string not in a DFA's language there must be one of length shorter than the size of the DFA's state set.
- With this in mind, we only have to search through the set of all strings of length $|Q|$.
- There are, of course $|\Sigma|^{|Q|}$ such strings. So, this approach takes exponential time. (Note, that the size of the DFA's encoded description will grow at least linearly with the size of the state set.)
- So, if this was the only algorithm we knew of for the problem, we might suspect it belongs in $PSPACE$ but not in P.
- This is also a good chance to talk about a class of problems I haven't mentioned previously, CoNP.
- CoNP is the set of problems that are complements of problems in NP. If we want to decide if a DFA's language is "not ALL", we can just guess a $w \notin L$ and check that we guessed right. So, ALL_{DFA} is in CoNP since the complement of ALL_{DFA} is clearly in NP (which is not surprise since both languages are actually in P).

What About Non-deterministic Finite Automata?

(Click for video)

1. We now know that ALL_{DFA} belongs to P.
2. Now, consider the very similar looking problem

$$ALL_{NFA} = \{\langle A \rangle \mid A \text{ is a NFA and } L(A) = \Sigma^*\}$$

3. We also know that this problem is decidable.
   - We can use the subset construction to convert the NFA to a DFA and then use the polynomial time decision procedure for ALL_{DFA}.
   - Unfortunately, in the worst case, the subset algorithm produces a deterministic machine with 2^s states given a nondeterministic machine with s states.
   - Therefore, the procedure we have outlined appears to take exponential time.
4. The naïve algorithm would also use exponential space just to write/store the description of the deterministic machine.

Squeezing ALL_{NFA} into PSPACE

(Click for video)

1. We can do better.
2. Rather than actually building the deterministic machine, we can simulate the execution of the nondeterministic machine on various (and very many) possible inputs by keeping track of the states it could be in after processing each symbol of the input. It only takes linear space to encode the current subset of states for a given prefix of the input.
3. Thus for a given input w we would run the algorithm

NFA-accepts(Q, Σ, δ, q_0, F, w) =

currentStates = { q_0 }
while more symbols in w and currentStates non-empty {
  x = next symbol of w

  nextStates = $\bigcup_{s_i \in \text{ currentStates}} \delta(s_i, x)$

  currertStates = nextStates
}
if no more input & currentStates contains a final state {
  accept
} else {
  reject
}

4. The "various" inputs we have to consider will be all strings of length less than or equal to 2^s (where s is still the number of states in the NFA described by $\langle A \rangle$). This is because of the pumping lemma. We

know that the pumping length of a regular language is bounded above by the number of states of any DFA that recognizes the language. If $w \notin L(A)$, then $w \in \overline{L(A)}$ and we know that the pumping length of $\overline{L(A)}$ must be 2^s or less since we get a DFA that recognizes $\overline{L(A)}$ by just flipping the accept and reject states in the DFA for $L(A)$. As a result if $|w| > 2^s$, we can pump down to find a smaller $w' \notin L(A)$ which means that there must be some such w' for which $|w'| < 2^s$.

5. Unfortunately, storing just one of these "short" inputs would take exponential space, so at first glance, this simulation algorithm seems to have done nothing more than move the exponential space requirement from the representation of the machine to keeping track of the input possibilities.

6. It is time for nondeterminism to come to our rescue. Suppose we use a nondeterministic algorithm to search for $w \notin L(A)$. That is, we modify our loop to just guess each symbol in an input of length less than or equal to 2^s.

   NFA-rejects-some-w(Q, Σ, δ, q_0, F) =

   currentStates = { q_0 }
   length = 0
   while length $\leq 2^{|Q|}$ & currentStates $\cap F \neq \emptyset$ {
   guess the next letter of w
   length = length + 1
   nextStates = apply δ to all elements of currentStates
   currertStates = nextStates
   }
   if currentStates $\cap F \neq \emptyset$ {
   reject
   } else {
   accept
   }

7. The only data the nondeterministic algorithm needs to store is the set currentStates which we already argued requires only $O(s)$ space and the value of length. The value of length can get exponentially large, but using place notation this also only requires $O(s)$ space. So, our algorithms space requirements are linear in the size of its input!

8. Then, we can invoke Savitch's Theorem to conclude that we could convert this to an $O(s^2)$ space deterministic algorithm!

9. Note that while this reduction from exponential space to n-squared space is pretty amazing, the algorithm described still requires exponential time even when viewed as a non-deterministic algorithm so we are far from NP.

A PSPACE Complete Problem

(Click for video)

1. If we dream of showing that PSPACE is actually a bigger set than NP, we should be looking for problems within PSPACE that are as hard as possible. That is why the NP-complete problems are so interesting relative to the P = NP question. If any problems in NP are not in P, then the NP-complete problems have to be among them. So, we should look for examples of PSPACE complete problems.

2. One might expect to define a new notion of reducibility to replace $\leq_p$ in the definition of PSPACE-compete. Maybe something like $\leq_{NP}$. The problem is that it may be the case that $P = NP$. So, PSPACE completeness is defined using polynomial time reductions:

   Definition: We say that a language B is *PSPACE-Complete* if $B \in$ PSPACE and for all $A \in$ PSPACE, $A \leq_p B$.

3. Recall how we explored NP-Completeness:

   - The Cook-Levin theorem used encodings of computation histories as boolean formulas to show that any language recognized by a TM that ran in nondeterministic polynomial time could be reduced to an instance of 3SAT.
   - Then, we could show that another language is NP-complete by showing how to reduce 3SAT to that language (or to any other language previously shown to be NP-complete in this indirect way).

4. So, we would like to find some way to encode computation histories of Turing machines that run in polynomial space as members of some other language, thereby showing that any PSPACE computation can be reduced to this other language.

5. Consider the language of computation histories of a Turing machine:

   Definition: Given a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ and a string $w \in \Sigma^*$, we define the language of computation histories for M on w as:

   $L_{Accepting-Computation-History}(M, w) =$

   $\{\#w_0\#w_1\#\ldots\#w_n\# \mid$

   - each w_i is a configuration for M,
   - w_0 is the initial configuration for w,
   - w_n is an accept configuration, and each w_i yields w_{i+1} according to δ

   $\}$

6. We used this language previously when discussing whether the language ALL_{CFG} was recognizable. In that case, we reversed every other configuration to make it possible to encode all invalid histories as a CFL.

Reducing PSPACE Problems to ALL_{NFA}

(Click for video)

1. Here, we will limit the language to histories of a polynomial space Turing machine and discovers this makes its complement relatively easy to recognize. So, consider:

   Definition: Given a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ which uses at most $p_M(|w|)$ tape cells on input w and a string $w \in \Sigma^*$, we define the language of polynomial space computation histories for M on w as:

   $L_{Accepting-Polynomial-Computation-History}(M, w, p_M) =$

   $\{\#w_0\#w_1\#\ldots\#w_n\# \mid$

   - each w_i is a configuration for M,
   - w_0 is the initial configuration for w,
   - w_n is an accept configuration, and each w_i yields w_{i+1} according to δ, and
   - for all $i, |w_i| < p_M(|w|)$

   $\}$

2. The somewhat shocking surprise is that the complement of this language is regular!

   - We can build a NFA that guesses how and where within a string the requirements of the language are violated and then checks/verifies this guess.
     - The first guess is that the format of the input just violates the formatting expectations in some way (for example, there must be exactly one symbol representing a state between every pair of #s). Each such rule is easily checked by a DFA that our NFA can branched to if it guesses that format is the problem.
     - Next, we could verify that w_0 is not an encoding of the initial configuration. This requires a large sub-DFA with $p_M(|w|)$ states, but is easy to do.
     - The NFA might guess that the problem is that q_{accept} never appears in a configuration. Again this is easy to do with a sub-DFA whose size is independent of $|w|$.
     - The final component is a bit tricky:
       * Remember how in the proof of the Cook-Levin theorem we used small boolean formulas to describe each of the "bad" 2 by 3 sub-blocks of cells in the table of configurations underlying the proof.
       * Our NFA can guess the rows and columns where such a bad configuration would occur in the potential computation history provided as its input.
         · Guessing the row is easy. Just pick your favorite #.
         · Guessing the column is a bit harder because you have to make sure you take the same number of steps in both w_i and w_{i+1} when verifying the badness of the 2x3 blocks.

- For each starting column, it takes at most $p_M(|w|)$ states to take the machine to column c in row i and then the same number of states to accurately get to the same column in the next configuration.
- We need such a set of states for every possible column we might guess. So, in total, we need $p_m^2(|w|)$ states for such positioning. This is big, but it is polynomial! An NFA can have this many states.

3. The conclusion is a bit surprising. Given any A in PSPACE, there must be a Turing machine M_A that uses space bounded by some polynomial p_A. As a result, given a potential input w to M_A, we can create a description of an NFA that recognizes the complement of the set of polynomial computation histories of M_A on w in polynomial time in such a way that this machine's description belongs to ALL_{NFA} iff $w \in A$.

4. In other words, ALL_{NFA} is PSPACE complete!

A Final Word

(Click for video)

1. No words can capture this material. You just have to watch the video.