CS 361 Meeting 31 — 5/11/20

Space: The Final Frontier

(Click for video)

1. We spent the last week talking about the relationship between the amount of time an algorithm uses and the tasks it can complete.

2. A classic principle of computer science is that the design of algorithms involves so called space-time tradeoffs. Situations in which it is often possible to make an algorithm require less space (i.e. memory) by taking more time or less time by using more space.

3. Accordingly, it seems natural that we should also consider the relationships between limitations placed on the amount of space an algorithm uses and the tasks it can complete. To do this, we stick with big-O notation and define classes of languages associated with memory limitations using:

   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.

4. Again, we will focus on languages associated with polynomial time bounds. Therefore, we say:

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

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

5. There are a number of useful things we can say about the relationships between time and space complexity classes.

   $TIME(f(n)) \subseteq SPACE(f(n))$

   This follows from the fact that an algorithm that executes for only n steps can at most look at n tape cells.

   $P \subseteq PSPACE$

   This is a special case of the preceding observation.

   $SPACE(f(n)) \subseteq TIME(2^{kf(n)})$

   If a Turing machine ever repeats a configuration, it will loop rather than halt. Since languages in our space and time complexity classes require Turing machines that halt, we can assume that the number of possible distinct configurations is an upper bound on the running time of a space limited Turing Machine. The number of configurations that are possible with at most $f(n)$ tape cells is roughly the size of the tape alphabet raised to the power of the number of cells used or $|\Gamma|^{kf(n)}$.

6. Continuing to mimic our approach to time complexity classes, an obvious next step is to consider the space complexity classes associated with nondeterministic machines:

   Definition: Let $s : \mathcal{N} \to \mathcal{R}^+$. Define the **space complexity class,** $NSPACE(s(n))$ to be the collection of all languages that are decidable by nondeterministic Turing machines using $O(s(n))$ distinct tape cells on any computation path on inputs of size n.

   and

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

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

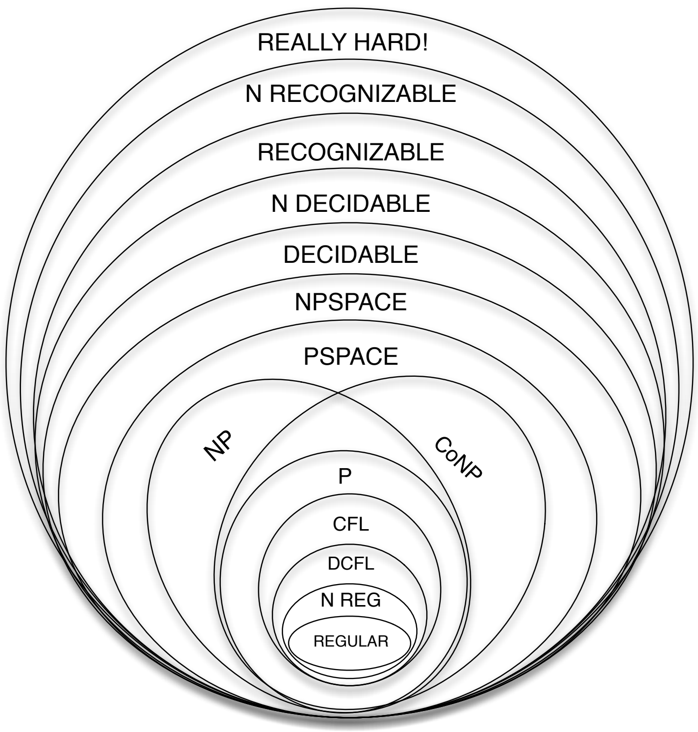
7. It should be clear that

$$P \subseteq NP \subseteq PSPACE \subseteq NPSPACE$$

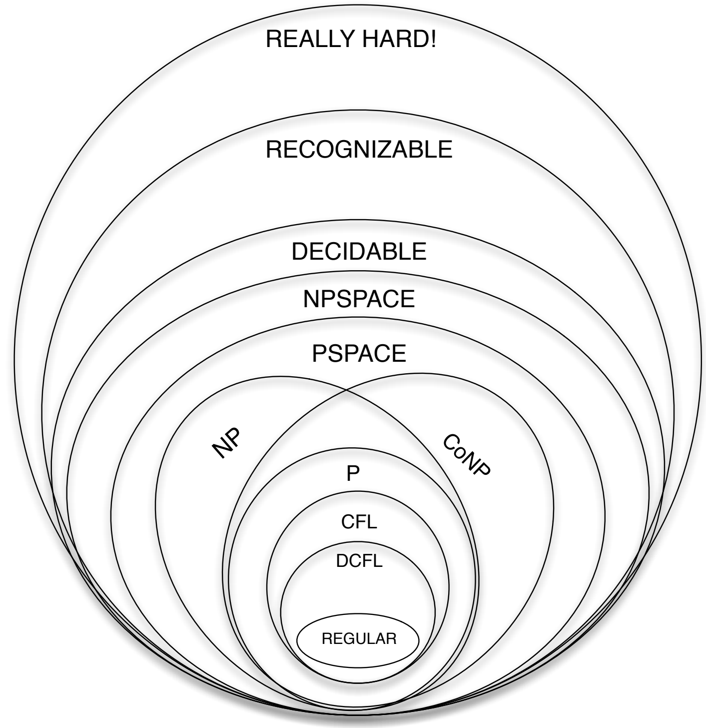
   but it turns out not to be as clear whether any of the $\subseteq$s can be replaced by $\subset$.

Click here to view the slides for this class

8. Throughout the semester, we have seen that sometimes nondeterminism gives us additional power to express algorithms, but sometimes it doesn't. Consider the following "map" of the inclusion relations that might have existed between various classes of languages we have considered:



It shows what this universe would look like if adding nondeterminism always added expressive power. In fact, we know that the real "map" of these complexity classes looks more like:



because decidability, recognizability and regularity are properties of classes of languages that are unchanged whether we include nondeterminism or not.

9. An interesting question to consider, therefore, is whether nondeterminism matters when considering SPACE vs. NSPACE.

Why NPSPACE Might Properly Contain PSPACE

(Click for video)

1. Just as it seems intuitively reasonable that NP would be a proper superset of P, It seems likely that NPSPACE might properly contain PSPACE.

2. One should, however, recall that we were able to simulate a nondeterministic Turing machine with a deterministic machine. Given this, we should at least think about how much space that simulation might require.

   - Recall, that our technique for simulating all of the possible paths a non-deterministic TM on a single deterministic TM involved a form of dovetailing. One way of thinking about the simulation is as a doubly nested loop of the form:

     ```
     Repeat
         For each ongoing path being explored
             add all possible next configurations to our tape
     until we find an accept state or run out of configurations.
     ```

   - As we implement it, we require enough space on our tape to store every possible configuration in the tree of possible execution paths on our tape.
   - The height of the tree is bounded by the length of the longest execution path. We argued earlier that if only k tape cells are used, there must be at most $O(2^k)$ steps taken because otherwise a configuration would be repeated. At each step, the number of leaves in the tree can multiply by a constant determined by the degree of nondeterminism in the machine's δ function. So, the number of configurations we must store is doubly-exponential!
   - We can "economize" by only storing the configurations that have not yet been expanded (unlike time, we can reclaim space storing information we no longer need), but this is still doubly-exponential.

3. Our original simulation used breadth-first search because we could not otherwise ensure that every thread eventually made progress. If each thread guaranteed to halt in a finite number of steps (which is true since we are assuming these machines decide their languages) it is safe to instead use depth-first search. This, however still requires a stack of exponential size so that we can back out and explore other threads.

4. Basically, it seems hard to imagine any way to explore the entire space of reachable configurations of the nondeterministic machine, until you learn about...

Savitch's Theorem, Pt. 1

(Click for video)

1. So that you don't miss the point of this next topic, remember that we all believe that $P \neq NP$ because we can't imagine any way to capture all the paths a nondeterministic machine might take in polynomial bounded time. Similar logic might lead to the assumption that it must be true that $PSPACE \neq NPSPACE$.

2. Consider the following alternate approach to "exploring" the space of configurations that a polynomial space limited nondeterministic Turing machine might reach. In particular to explore whether it can reach any terminal configuration.

   - Define a recursive boolean function canReach (details shortly) that takes two states and a step limit and returns true only if the nondeterministic Turing machine being considered could reach the second state from the first in at most the step limit moves.
   - Execute the algorithm

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

   - First, observe that if we ignore the space required by the call to canReach, this loop only requires polynomial space. To see why:
     - Imagine an odometer that counts in base n where n is the size of the alphabet used to encode configurations using the letters of that alphabet as digits.
     - Make the odometer contain $p(|w|) + 1$ digits and initialize it to 1. Whenever the odometer hits a value whose first non-zero high-order digit is 1, treat all the digits that follow the first non-zero digits as the next string over the alphabet used to encode configurations.

– Scan each such string to make sure it follows the rules for encoding a configuration. If it does, then apply canReach to the string.

Savitch's Theorem, Pt. 2

(Click for video)

1. The tricky part is how we define canReach:

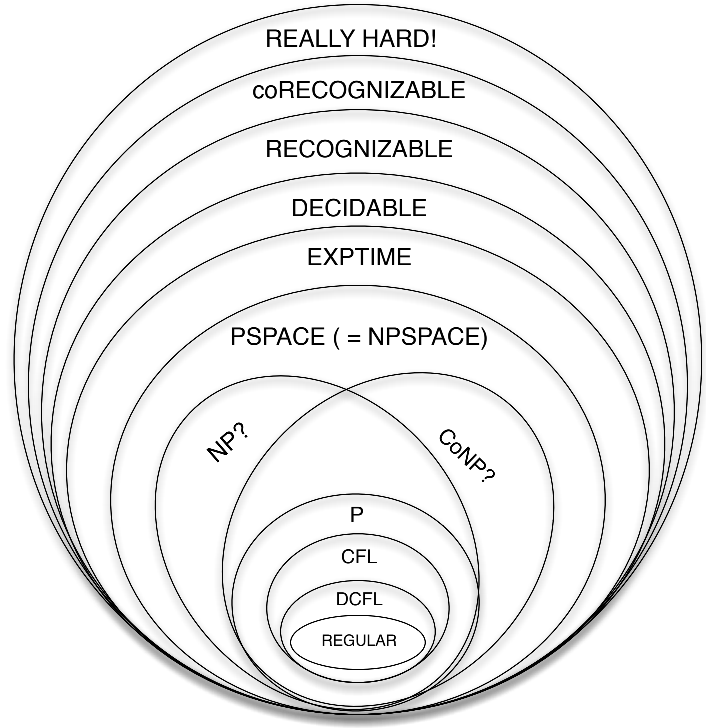
```
canReach( start, end, steps) {
if steps == 1 {
    return start == end
} else if steps == 2 {
    return ( start yields end )
} else {
    for every mid ∈ configurations {
        if ( canReach(start, mid, steps/2) &
                canReach(mid, end, steps/2) ) {
            return true
       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

2. Note that each recursive invocation of this function requires space to store the four parameters/local variables start, end, mid and steps. Each of these requires $O(p(n))$ tape cells.

3. The depth of the recursion will be at most $O(p(n))$ as well since "steps" starts out at $O(2^{kp(n)})$ and is halved with each nested call effectively reducing the power of 2 used by 1. After $O(p(n))$ of these repeated divisions by 2 will yield 1.

4. As a result, the complete computation will require $O(p^2(n))$ space. We can therefore conclude that in general $NSPACE(p(n)) \subseteq SPACE(p^2(n))$ and in particular that $PSPACE = NPSPACE$!

5. This brings us a step closer to an accurate map of the classes of q 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 $\subset$, but we don't know which one or how many.