CS 361 Meeting 35 — 12/7/16

Announcements

1. Homework due today.

2. Final exam: Open text/notes. 24-hour exam available from registrar’s office. Limit of 9 consecutive hours from start to finish of work on the examination.

3 Dimensional Matching

1. Last time we were discussing a reduction that show that the 3-dimensional matching problem is NP-complete:

   Given three sets $X$, $Y$ and $Z$ and a set $A \subseteq X \times Y \times Z$, a three dimensional matching $M$ is a subset of $A$ such that if $(x_1, y_1, z_1), (x_2, y_2, z_2) \in M$ then $x_1 \neq x_2$, $y_1 \neq y_2$ and $z_1 \neq z_2$. In the case that $|X| = |Y| = |Z| = |M|$ we say that $M$ is a maximal or complete matching.

2. This is just a formal version of what I had earlier presented as the “Awkward Diners” problem.

3. To provide a better way to visualize how one can reduce 3SAT to 3DM, I had introduced another way to diagram the relation between the sets $X, Y$ and $Z$ and the allowable triples using triangles to enclose the elements of each triple.

We will use this notation to help capture the structure of the sets and triples we construct to map an instance fo 3SAT into an instance of 3DM.

4. We discussed the first step in the construction of a 3D-matching problem that encodes a given formula $\phi$ viewed as an instance of 3SAT:

   - If $\phi$ has $t$ clauses using $v$ variable, we will include $t \times v$ items in $X$ corresponding to true variables and another $t \times v$ items corresponding to false versions of the same variables. Formally:
     
     $$\{x_{i}^{c_j}, \overline{x_{i}^{c_j}} \mid 1 \leq i \leq v, 1 \leq j \leq t\} \subseteq X$$

   - We then include triples that will force anyone trying to construct a maximal matching to pick triples that include either all of the $x_{i}^{c_j}$ or all of the $\overline{x_{i}^{c_j}}$ for a given value of $j$.
   - In the set $Z$ we will include $t \times v$ items named $b_{i}^{c_j}$ each of which can only be included in matching triples with the $x_{i}^{c_j}$ or $\overline{x_{i}^{c_j}}$ items in $X$. 

5. The visualization of triples as triangles can help clarify what is going on here. The diagram below shows the triples and elements we have been describing for $x_1$ in the case that we have a formula with 4 clauses.

- In the set $Y$ we will include $t \times v$ items named $a_i^{c_j}$ which will be included in triples in such a way that $a_i^{c_j}$ is only included in triples with $b_i^{c_j}$ and either $\overline{x_i^{c_j}}$ or $x_i^{c_j-1}$ if $j > 1$ or $x_i^{c_1}$ if $j = 1$.
- Essentially, there are two triples that could be used to include $\overline{x_i^{c_j}}$ or $x_i^{c_j-1}$ in a matching using a triple involving the $a$ and $b$ elements of $Y$ and $Z$. If we pick any of the triples including $x_i^{c_j}$ then the only way to include triples that cover all of the $a_i^{c_j}$ and $b_i^{c_j}$ triples for that value of $i$ in the matching is to include all of the $x_i^{c_j}$ triples for the given value of $i$. Similarly, if we include any $\overline{x_i^{c_j}}$ then to use all of the $a_i^{c_j}$ and $b_i^{c_j}$ triples for a given $i$ we must include all of the $\overline{x_i^{c_j}}$ items for that $i$.
- Ultimately, we will interpret this choice of matching the plain or negated copies of $x_i^{c_j}$ as choosing to set $x_i$ true if the plain copies are unmatched and vice versa.

Because each of the $a$ and $b$ items are shared between an $x_i^{c_j}$ triple and a $\overline{x_i^{c_j}}$ triple, no matching can exist in which both $x_i^{c_j}$ and $\overline{x_i^{c_j}}$ are matched. On the other hand, all of the triples for $x_i^{c_j}$ or $\overline{x_i^{c_j}}$ can be included in a matching without conflicts.

6. At this point, it is worth doing a bit of arithmetic. We have placed $2t \times v$ $x_i^{c_j}$ and $\overline{x_i^{c_j}}$ items in $X$ but only $t \times v$ $a_i^{c_j}$ and $b_i^{c_j}$ items in $Y$ and $Z$. We need to make the three sets of equal size so we need another $t \times v$ items.

7. We also need to make sure that all of the clauses in $\phi$ are related to our sets and triples in such a way that it is possible to complete a matching if and only if it is possible to satisfy all the clauses.

8. With this in mind, we want to add some triples so that they will be part of a matching if each clause is satisfiable. A clause is satisfied as long as one of its literals is true. So, we will include triples such that one triple will be included for each clause corresponding to the literal in that clause that is satisfied.

- For each clause $c_j$ we will add an item $c_j$ to $Y$ and $d_j$ to $Z$.
- If $x_i$ appears in $c_j$, we will add $(x_i^{c_j}, c_j, d_j)$ to $A$.
- If $\overline{x_i}$ appears in $c_j$, we will add $(\overline{x_i^{c_j}}, c_j, d_j)$ to $A$.

9. If a subset of $A$ can be found that makes it possible to build a matching that includes all of the $a_i$, $b_i$, $c_j$ and $d_j$ elements we have added to $Y$ and $Z$, then $\phi$ is satisfiable, and if $\phi$ is satisfiable, we can build such a matching. The only problem is that a lot of elements of $X$ will not be included in the matching.

10. We need to add $2t \times v - (t \times v + t) = (v - 1) \times t$ more items to $X$ and $Y$ to have any hope of finding a total of $2t \times v$ triples to include in our matching.

11. With this in mind, we add a set of items $\{e_k \mid 1 \leq k \leq (v - 1) \times t\}$ to $Y$ and a set of items $\{f_k \mid 1 \leq k \leq (v - 1) \times t\}$ to $Z$. 


12. We want these items to be able to “gobble up” any of the \((v - 1) \times t\) unmatches items in \(X\), so we include a triple for every item in \(X\) for every pair of \(e\) and \(f\) items:

\[
\{(x^e_i, e_k, f_k) \mid 1 \leq i \leq v, 1 \leq j \leq t, 1 \leq k \leq (v - 1) \times t\} \subset A
\]

\[
\{(x^e_j, e_k, f_k) \mid 1 \leq i \leq v, 1 \leq j \leq t, 1 \leq k \leq (v - 1) \times t\} \subset A
\]

**Space: The Final Frontier**

1. We have spent the last week or two 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 \(s : \mathbb{N} \rightarrow \mathbb{R}^+\) 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 \(s : \mathbb{N} \rightarrow \mathbb{R}^+\). 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 $\subset$.

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.

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

11. 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:

  \begin{verbatim}
  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.
  \end{verbatim}

- 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 \( \delta \) function. So, the number of configurations we must store is doubly-exponential!

- We can “economize” by only storing the configurations that have not yet be expanded (unlike time, we can reclaim space storing information we no longer need), but this is still doubly-exponential.

12. Our original simulation used breadth-first search because we could not otherwise ensure that every thread eventually made progress. If each thread is limited in the amount of space it can use, 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.

13. 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**

1. So that you don’t miss the point of this next topic, remember that we all believe that \( P = NP \) because we can’t imagine any way to capture all the paths a nondeterministic machine might take in polynomial bounded time.

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 \( \text{canReach} \) (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

  \begin{verbatim}
  for every accepting configuration \( f \) {
    if ( \text{canReach}( \text{initial configuration}, f, 2^{\ell(n)} ) ) {
      accept
    } else {
      reject
    }
  }
  \end{verbatim}

- The tricky part is how we define \( \text{canReach} \):

  \begin{verbatim}
  \text{canReach}( \text{start}, \text{end}, \text{steps} ) {
    if \text{steps} == 1 {
      return start == end
    } else if \text{steps} == 2 {
      return ( start yields end )
    } else {
      for every mid \in \text{configurations} {
        if ( \text{canReach}(\text{start}, \text{mid}, \text{steps}/2) \&
             \text{canReach}(\text{mid}, \text{end}, \text{steps}/2) ) {
          return true
        }
      }
      return false
  }
  \end{verbatim}
• 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.

• 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. After \(O(p(n))\) these repeated divisions by 2 will yield 1.

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

3. This brings 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 \(\subset\), but we don’t know which one.