CIS 511 Homework 5

Stephen Phillips, Dagaen Golomb

April 8, 2015

Problem 1

The NP-Completeness proof for SUBSET-SUM works because the input is encoded in $\log(n)$ bits, so solving the problem takes time exponential in the number of bits. Since complexity is measured against the size of the input, this places SUBSET-SUM $\in$ NP.

However, for UNARY-SSUM, each number occupies length equal to its value. Therefore, we can convert each number into binary which can be done in polynomial time via successive division by two and checking any remainder (which becomes the next significant digit). This runs in polynomial time. We do this for each number. Now, we have the classic SUBSET-SUM problem. We can solve this in a brute-force manner by simulating the non-deterministic TM.

This simulation takes time exponential in the length of the [created] input, which is $\log(n)$ of the original input. Therefore, this runs in time polynomial ($O(e^{log(n)}) = O(n)$) of the original input. Thus, UNARY-SSUM $\in$ P.

Problem 2

Part a

We know that $CNF_2 = \{\langle\phi\rangle \mid \phi$ is a satisfiable CNF formula with each variable appears at most 2 times$\}$ is in P since we can make an algorithm for it.

> **function** $M(\phi)$
>> Check that all variables x_i appear at most twice as
>> Create ψ, a modifiable copy of ϕ
>> Create bit vector b for the final assignment
>> **for all** $i \in \{1, \ldots, n\}$ **do**
>>> **if** There is one clause left **then**
>>>> Accept
>>> **else if** There is one variable left **then**
>>>> **if** Clauses x_i and $\bar{x}_i$ both are in ϕ **then**
>>>>> Reject
>>>> **else**
>>>>> Accept
>>>> **end if**
>>> **else if** the literal x_i appears once **then**
>>>> $\psi \leftarrow \psi(x_i = 1)$ (and simplify ψ)
>>> **else if** the literal $\bar{x}_i$ appears once **then**
>>>> $\psi \leftarrow \psi(x_i = 0)$
>>> **else if** the literal x_i appears twice **then**
>>>> $\psi \leftarrow \psi(x_i = 1)$
>>> **else if** the literal $barx_i$ appears twice **then**
>>>> $\psi \leftarrow \psi(x_i = 0)$
>>> **else**
>>>> Combine the clauses with x_i and $\bar{x}_i$ in ψ by creating a new clause with all the other literals in
> the two clauses together

end if
  end for
 end function

We need to show that this works, so we analyze each case in the if else chain in the loop. We show that each branch either makes the problem smaller, or solves the problem.

- If there is only one clause, we can set any literal in the clause to true and we have satisfied the problem, so we are done.

- If there is one variable x left, since we know each variable only appears twice, it is unsatisfiable if and only if the clauses left are x and $\bar{x}$, so we accept accordingly.

- If a literal x only appears once in the formula, we can just it to true and get rid of the clause that the the literal is in. Similarly, if the literal $\bar{x}$ is in only once, we can just set it to false to get rid of the clause

- If the literal x appears twice in the formula (or equivalently $\bar{x}$), then we can like before just set the literal to the appropriate value and get rid of the clauses that it occupies.

- The tricky case - if x and $\bar{x}$ both appear in different clauses. Note however that:

$$(x + a)(\bar{x} + b) \iff \bar{x}a + xb + ab \iff a + b$$

Since we can choose x accordingly. So it is satisifiable if and only if $a + b$ is true, so we combine the clauses and continue.

Since we reduce the problem by at least one variable each time, with simple boolean formula manipulations in each step, we can run this in polynomial time in the number of clauses and variables. Therefore, this is in P.

Part b

We show that $CNF_3 = \{\langle \phi \rangle \mid \phi$ is a satisfiable CNF formula with each variable appears at most 3 times$\}$ is NP-Complete. We show how to change an instance of $3SAT$ to an instance in CNF_3.

Given a $3SAT$ we can increase the number of variables to make sure that no one variable shows more than 3 times. Let us say a variable x shows up in k clauses. Then we create new variables $y_1, \ldots, y_k$. We create a 'chain' of implications:

$$y_1 \implies y_2 \implies y_3 \implies \ldots \implies y_n \implies y_1$$

This means all these variables must have the same value. In CNF form, using $x \implies y \iff (\bar{x} + y)$, this would be

$$(\bar{y_1} + y_2)(\bar{y_2} + y_3) \ldots (\bar{y_n} + y_1)$$

Then for each clause that x is in, we use one of the y_i variables. Now each of the y_i variables appears twice in the 'chain' of implications, and once in the actual clauses, meaning it appears a total of 3 times.

We illustrate with an example, using the simple example with only the variable x:

$$(x + a_1 + b_1)(x + a_2 + b_2)(\bar{x} + a_2 + b_2)(\bar{x} + a_4 + b_4)$$

This becomes with these new rules:

$$(\bar{y_1} + y_2)(\bar{y_2} + y_3)(\bar{y_3} + y_4)(\bar{y_4} + y_1)$$
$$(y_1 + a_1 + b_1)(y_2 + a_2 + b_2)(\bar{y_3} + a_2 + b_2)(\bar{y_4} + a_4 + b_4)$$

Now we need to make sure this is still a polynomial time transformation. Say there are n variables and m clauses. In the worse case, a variable appears in every clause. This means we add m variables to the equation, and m clauses for the 'chain' of implications. If this happens for each variable we get mn new clauses, and mn new variables, given $O(mn)$ sized problem, which is just a quadratic blow up in this size of the formula, and therefore still a polynomial time transformation. Therefore if we can solve CNF_3 we can solve $3SAT$ which we know to be NP-complete. Since checking a boolean formula is in NP since we can use the assignment as a certificate, this is NP-complete

Problem 3

Let the Mine Consistency Problem (MCP) be as defined in the problem:
$\{G : G$ has a satisfiable assignment of mines$\}$.

This is clearly in P: given a graph with mines and numbers, it is easy to check if this it is a consistent and accurate labeling. Now we will show MCP $\in$ NP by reducing MCP to SAT. This will show MCP is at least as hard as SAT and hence in NP.

Let us define a variable x_{ijk} that, for each $0 \leq k \leq 8$, represents if the cell at row i and column j has value k. Note in this notation we are using $k = 0$ to represent the presence of a mine.

The reduction works as follows. For all i, j, k, we create a conjunction $(x_{ijk} \cdot x_{ijk'}^-)$ for each $k' \neq k$. This enforces that each cell cannot be assigned two different values. We also produce the disjunction $(x_{ij0} + x_{ij1} + ... + x_{ij8})$ for each i, j pair. This means each cell must be assigned some value. These expressions in total enforce each cell to have exactly one value.

Next we will enforce the rules of Minesweeper. For each x_{ijk} we enumerate all positional combinations of k mines around the number and build expressions to match them. Note for x_{ij0} we will not place any restrictions since the expressions for the numbered cells will enforce the game rules. For example, for $k = 1$ for some i, j, we create the conjunction $(x_{ij1} \cdot x_{(i+1)j1} \cdot x_{(i+1)j1}^- \cdot ...)$ with the remaining adjacent cells being not literals. This is an example of a valid configuration for a cell marked as 1 with only a mine in the position below it. Note that these expressions are similar for edge nodes with the omission of cells that do not exist (making the construction even easier). Note that although combinations in general grow combinatorially, we are only dealing with specific combinations which are a constant number of expressions each. Therefore, this operation still only takes constant time for each cell. Finally, we "or" each combination together to allow any combination of k mines around the cell.

We now assert that the above Boolean formula is satisfiable $\iff$ there is a consistent mine placement.

- $\implies$ Given a satisfiable assignment to the above formula, pick the true x_{ij0}'s for each i, j. By construction, no other x_{ijk} value is true, and each adjacent numbered cell must account for this mine. Therefore, this mine is placed in a valid position and consistent with the game rules. Since this holds for each mine, this is a consistent mine placement.

- $\impliedby$ Given a consistent mine placement, set each mine position's x_{ij0} to true and set the literals for other values of k to false. For each numbered cell set the corresponding x_{ijk} to true (if it is marked with k) and set the other literals over i, j to false. Now each cell has a distinct assignment so the first part of the construction is true. Also, since this is a consistent marking, the second expression is true by construction as well. Therefore, the conjunction of these two are true and the formula is satisfiable.

Problem 4

If $P = NP$, then there exists a polynomial time decider D for SAT runs in polynomial time. From this we can create a new machine M that finds the assignment of the variables. The idea is that we assign true or false to each of the variables in order, and use D to determine if it is a valid assignment. If there are multiple assignments we just pick the one that works in order of the variables.

> **function** $M(\phi)$
>   Create ψ, a modifiable copy of ϕ
>   Create bit vector b for the final assignment
>   **for all** $i \in \{1, \ldots, n\}$ **do**
>   Set $\psi' \leftarrow \psi(x_i = 1)$

```
    if D(ψ′) accepts then
        ψ ← ψ′
        b_i ← 1
    else
        ψ ← ψ(x_i = 0)
        b_i ← 0
    end if
  end for
  Output b
end function
```

In words, we greedily go through the variables assigning them. We initially assign each variable x_i to 1, and use D to check if that assignment is feasbile. If it is, then there must be a valid assignment where x_i is 1, and if not x_i must be 0. So we assign x_i appropriately. Then we update the formula to reflect this choice, and continue appropriately.

Now we show this runs in polynomial time. Let n be the number of variables, and m be the number of clauses in the original formula. On each iteration of the outer loop, we update the formula, which takes $O(m)$ time, and we run the decider D, which runs polynomial time for some polynomial $p(n + m)$. There are n iterations of the loop, so in total this takes $O(nmp(n + m))$ time, which is still a polynomial.

Problem 5

Let P = NP. We have seen that 3COL $\in$ NP, so by our assumption that P = NP there is a polynomial time algorithm that decides if a graph is 3-colorable. Call this L.

Note that by a similar construction as the original decidability proof, we can show that deciding if a graph is colorable given an already partially-colored graph (i.e., with some nodes given colors already) is NP-complete (and by our assumption, in P). The idea that for already assigned colors we can force their presence in the end solution by only allowing them to be assigned the color they have been given. The proof is otherwise the same.

We will use L to 3-color a graph when given a 3-colorable graph. Given a graph, choose a node and give it a color. Ask L if the graph is 3-colorable. If if is, move on to another uncolored node. If it is not, try the another color. If any node fails for all colors, the graph cannot be 3-colored. Otherwise, all nodes will eventually be colored and the graph will have a valid 3-coloring.

Now, for each node we must try at most 3 colors, and once a node is visited it is not recolored (note we do not need to worry about recoloring since color assignments can be permuted between the node groups for each color given one valid assignment). So each node takes a constant amount of time to process. Since 3COL in in P by assumption, L runs in polynomial time. Finally, we visit each node once, so this is linear in the number of nodes. Therefore, this whole process runs in polynomial time.

Thus, if P = NP, we can 3-color a graph in polynomial time.

Problem 6

We want to, given a CNF formula ϕ, construct an NFA that accepts all nonaccepting configurations of ϕ. The idea is we create a sub-NFA for each clause that accepts if and only if the clause is false. To illustrate, we use a simple formula over 4 variables $\phi(x_1, x_2, x_3, x_4) = (x_2 + x_3 + \bar{x}_4)$
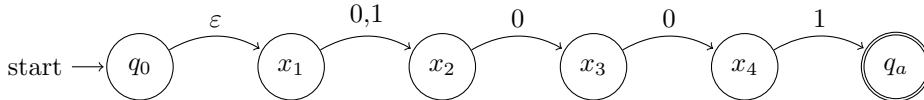

This takes in as input the bit string representing the assignment of all the variables. Each of the successive nodes represents the variables, and they transition to the next node on values that would allow the clause to reject. Notice that since x_1 does not appear in the clause, it can take whatever value it wants. So this accepts the non-satisfying strings $0, 1001$.

We can generalize this to more clauses, by making the initial state have ε-transitions to a chain for each clause, constructed in the exact same way as the above example. That would give, with m variables and c

clauses, cm nodes. The transition function would take then $O(cm)$ space to create. This means it can be created in $O(cm)$, or polynomial, time. This new NFA would accept on any rejecting input since it only takes one failing clause to make the formula reject.

If this formula has no accepting input, this NFA would reduce to one that accepts $\{0,1\}^*$, or:

$$0,1$$

$$\text{start} \longrightarrow q_0$$
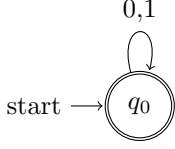
Therefore if we could reduce this NFA in polynomial, we would be able to solve SAT in polynomial time. In other words we just reduced SAT to minimizing an NFA. Therefore if $P \neq NP$ then there is no polynomial time solution for this.

Problem 7

We want to show that the problem of designating directions to edges of an undirected graph G so that a given subset C will have all nodes in it have either indegree 0 or outdegree 0, and all other nodes have indegree at least one is NP-complete. We reduce SAT to this. And example of the mapping is show for the formula $\phi(x_1, x_2, x_3, x_4) = (x_1 + \bar{x}_2 + x_3)(\bar{x}_1 + x_2 + x_4)(\bar{x}_2 + \bar{x}_3 + \bar{x}_4)$
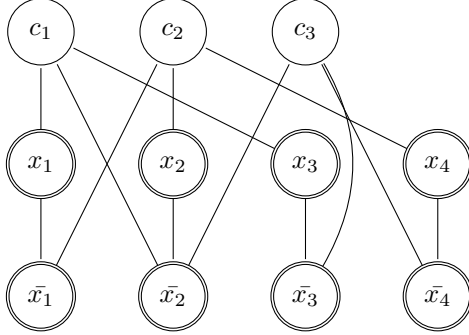
The highlighted nodes are the ones in C.

We want to create such a graph for any SAT formula ϕ. So in words the procedure is for every variable x_i, create nodes x_i and $\bar{x}_i$, and for every clause c_j create a node c_j. Connect the node x_i (or $\bar{x}_i$) to node c_j if the variable x_i (or $\bar{x}_i$) is in the clause c_j. Also create an edge between nodes x_i and $\bar{x}_i$. Place all the nodes x_i and $\bar{x}_i$ into C.

This graph has a solution the the direction problem if and only if the formula ϕ has a solution. If the formula ϕ has a solution, select the true literals $x_i/\bar{x}_i$ as the 'all outdegree' or 0 indegree nodes in C, and the complement pair as the 'all indegree' or 0 outdegree ones. Since only the pairs of literal nodes $x_i/\bar{x}_i$ are connected in C, this does not lead to an inconsistency. If this is the satisfying assignment then we should have at least one edge into each clause node c_j. Similarly, if we have such an assignment of edge directions, we can reconstruct the satisfying assignment, since only one in each of the pairs of literal nodes $x_i/\bar{x}_i$ has 'all outdegree' we can set those literals to true, and we know by the direction assignment that every clause will have one corresponding true literal inside it. Therefore, this graph has a solution to the problem if and only if the the formula has a satisfying assignment.