CSE 101 Homework 6

Winter 2015

This homework is due Friday March 13th at the start of class. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommended though not required.

Question 1 (Search vs. Decision, 50 points). Consider two slightly different versions of the problem SAT.

1. The decision version, in which you are supposed to determine whether or not there is a satisfying assignment.

2. The search version, in which not only must you determine whether a satisfying assignment exists, but if one does, you must also find one.

It turns out that these versions are essentially equivalent. Show in particular, that if you have some algorithm, \( A \) that solves the decision version of the problem in \( f(n) \) time, that there exists an algorithm that solves the search version of the problem in \( \text{poly}(n) + O(n f(n)) \) time.

Hint: Build up a solution to the problem one bit at a time, using the decision algorithm to figure out if there is a solution using that bit.

Solution 1.

Algorithm:

```latex
procedure propagate(E, v, val)
// E = a boolean expression
// v = a variable in E.V
// val = the value of v to propagate in E

E' = E // E' will be the new expression

for literal l in E:
    if l == v:
        E'.replace(l, val) // replaces the literal with the value
    if l == v': // the complement of v
        E'.replace(l, val')

return E'
```

1
procedure search(E)
    // E = a boolean expression
    if E.V = empty set:
        return ""
    if A(E) == False:
        return NULL
    x = first(E.V) // first variable in E.V
    E' = propagate(E, v, True) // E'.V doesn't contain v
    if A(E') == True:
        return "x <- True ".concat(search(E'))
    else:
        E' = propagate(E, v, False)
        return "x <- False ".concat(search(E'))

Justification: Let V be the set of boolean variables which appear in the expression E to be satisfied. First, we determine whether E can be satisfied at all using A(E). If E cannot be satisfied, we should not search for a solution.

Otherwise, since we know that a solution exists, we can begin satisfying E by experimentally setting one variable, v1, to "True"; propagating this value through E to create E'; and asking A if E' can be satisfied. Since E can be satisfied, v1 can be set to "True" or "False" to satisfy E. So if A(E') returns true, we know that we can set v1 = "True", and move on to other variables that remain in E' with variable set V' = V - {v1}. If, on the other hand, A(E') returns false, then it must be the case that v1 is set to "False" in any variable assignment that satisfies E. So we set v1 = "False", and continue with E' and variables V = V - {v1}, as before.

Complexity:
Note that the propagation step takes time proportional to the total length of the SAT expression, which should be polynomial.

So, for each variable that appears in E, we perform at most two value propagations, and two calls to A (we can reduce this to one call, if the algorithm recursively remembers that E can be solved after the first call to A). Therefore, our algorithm has time complexity nO(propagation(E)) + O(nf(n)). Since propagation has polynomial complexity, this reduces to O(poly) + O(nf(n)).

Question 2 (NP Completeness Reduction, 50 points). Problem 8.23 of the book.

Solution 2.
Membership in NP:

NODE-DISJOINT PATHS (NDP) is in NP if an NDP solution can be verified in polynomial time. The following simple verifier runs in polynomial time:

procedure VerifyNDP(G,S,T,P)
    // G = a graph
    // S = set of source nodes in G
    // T = set of destination/sink nodes in G
    // P = set of paths linking sources to sinks
    for n in G.V // G.V = vertices of G
        n.visited = False
// Check that all paths distinct
for p in P:
    first_node = p.nextNode // returns next node in path
    if first_node not in S:
        return False // Path should start with source node
    last_node = first_node
    if last_node.visited:
        return False // no node should be visited twice if paths disjoint
    last_node.visited = True
    while p.hasNext:
        next = p.nextNode
        if next.visited:
            return False
        next.visited = True
    // Assume G is in adjacency list form
    if next not in last_node.neighbors:
        return False // need edge between nodes for path to exist
    // update last_node
    last_node = next
    if last_node != S.sink // assume sources know their corresponding sink
        return False // Sources should be linked with appropriate sink

// Check that all sources/sinks have been visited
for s in S:
    if s.visited == False:
        return False
for t in T:
    if t.visited == False:
        return False

// all checks passed
return True

This verifier runs in $O(|V|^2)$ time. Initializing the “visited” field on each vertex takes $O(|V|)$ time. Checking that each path uses distinct nodes runs in $O(|V|)$ time, since each node can only be used once, and we return early if we find a conflict. Checking that each path begins with a source node requires $O(|V|)$ time per path, and there can be up to $\frac{|V|}{2} \in O(|V|)$ paths, so this requires $O(|V|^2)$. We could improve this to $O(|V|)$ if we know that the paths in $\bar{P}$ are arranged in order of the starting sources $S$. Finally, checking that all sources and sinks have been visited while checking the paths takes $O(|V|)$ squared. Therefore, this verifier runs in polynomial time.

Reduction from 3-SAT:
We reduce 3-SAT to NDP as follows. Let $E$ be a 3-SAT expression consisting of $m$ clauses and $n$ variables. For each clause $c = (x_i \lor x_j \lor x_k)$, create a source and sink node pair, $(s_c, t_c)$. Then, for each variable $x_i$ in the clause, create a node for the literal and a node for its complement, $x_i$ and $\neg x_i$, respectively. Finally, add edges from $s_c$ to each literal in the clause, and additionally from each literal in the clause to $t_c$. For example, in the above clause we add edges to create $(s_c, t_c)$ paths through nodes $x_i, x_j$, and $x_k$, but not $\neg x_i, \neg x_j$ or $\neg x_k$.

Next, for each variable $x$ in $E$, create a source and sink node pair $(s_x, t_x)$. For each such pair, create two node-disjoint paths – one using all of the $x$ nodes and one using all of the $\neg x$ nodes already created in the previous step. This yields a total of $k = m + n$ source/sink pairs.

If NDP finds $k$ node-disjoint paths, the satisfying assignment for $E$ can be constructed as follows: For each variable $x$, assign the literal in the $(s_x, t_x)$ path to "False." That is, if the variable $x_i$ uses the $\neg x_i$ path, assign $x_i = "True"$ (since $\neg x_i$ is assigned "False").

Now, assume there is a satisfying assignment for $E$. Then we show there must be $k$ node-disjoint paths in the resulting graph. If there is a satisfying assignment, at least one literal $x_i$ in every clause must evaluate to "True." Suppose the $(s_c, t_c)$ pair uses such a literal node on its $(s_c, t_c)$ path. Then the only path available now for the $(s_x, t_x)$ path is the path using the $\neg x_i$ nodes. Since there is a satisfying assignment, variables will be assigned consistently, and there are $k$ node-disjoint paths.

On the other hand, assume there is not satisfying assignment for $E$. This means that there is no way to consistently assign values to variables, and so it is impossible to have some clause path be disjoint from a variable path.

To illustrate the above, assume there are $k$ node-disjoint paths in the graph. Then this implies a satisfying assignment as described in the transformation above. For any $(s_x, t_x)$ path, whichever literal is used is now available for assignment for any clause. That is, if $(s_x, t_x)$ used the $\neg x$ nodes in its path, only the $x$ nodes are available to be "True" literals for any clause since all paths are node disjoint. In this way, variable assignments are consistent.

Reduction Complexity:

We have a source and a sink for each variable, and a source and a sink for each clause, so we have to construct $O(n+m)$ nodes. Furthermore, we have two nodes for each literal in each clause, requiring $3m \in O(m)$ more nodes. For each literal node $l$, we create a path $s_c - l - t_c$, where $c$ is the clause in which $l$ appears. This requires $6m \in O(m)$ edges to be added. For each literal node $l$ for variable $v$, we connect it to the appropriate path between $s_v$ and $t_v$, requiring $O(m)$ work. Finally, we complete the paths between $s_v$ and $t_v$ by connecting the last node in the true path and the last node in the false path to $t_v$, adding $2n \in O(n)$ edges. So overall, this reduction has complexity $O(n+m)$.

**Question 3** (Extra credit, 1 point). Free point!