CS1510 Reduction Problems 6, 10, & 15

Rebecca Negley, Sean Myers

October 26, 2011

6. Show that if one of the following three problems has a polynomial time algorithm then they all do:

   - The problem is to determine whether a Boolean Circuit (with gates NOT, binary AND, and binary OR) has some input that causes all of the output lines to be 1. Assume that the fan-out (the number of gates that the output of a single gate can be fed into) of the gates in a circuit may be arbitrary.
   - The problem is to determine whether a Boolean Circuit (with gates NOT, binary AND, and binary OR), and fan-out at most 2, has some input that causes all of the ouput lines to be 1.
   - The problem is to determine whether a planar Boolean Circuit (with gates NOT, binary AND, and binary OR) has some input that causes all of the output lines to be 1. A circuit is planar if it can be laid out on the 2D plane so that no pair of lines cross (if you like, you can assume that the layout is part of the input).

   Solution: First, let's get the obvious reductions out of the way. We know that (according to the group discussion), the input for planar Boolean Circuit doesn't have to be changed to reduce it to Boolean Circuit, and it still be a biconditional output relationship. So planar Boolean Circuit $\leq$ Boolean Circuit. The same goes for Boolean Circuit–Fan, the input doesn't need to be changed to reduce it to Boolean Circuit (again, this is according to the assumption discussed in the group). Our current reductions are: A = Boolean Circuit
   B = Boolean Circuit – Fan
   C = Boolean Circuit – Planar
   B $\leq$ A
   C $\leq$ A

   Now let's do A $\leq$ B. We must take the input of A, which could be anything and make sure the fan out to any of the circuit's is less than 2. Let's assume that the input to any gate is a variable, and that variable is determined by the previous gates or the inputs to the problem. Let's say we have a variable x that repeats k times in the circuit. We must take the 2nd to kth input and make them new variables: $x_2, ..., x_k$. We can't just "call" them new variables, we have to set it up in such a way such that x_i feeds into x_{i+1}, so that the fan out of the variables is always 2; 1 for the equation that is using it, and one for the input to the next variable. Next, to feed the variables into eachother, all we need to do is use k BUFFERs(two NOTs in a row, the signal never changes), and take the two-fan of the BUFFER and feed one into the equation that we replaced that variable with, and the other into the next BUFFER in the line. In very, very pseudo-ish language, it would look like this:

```
boolCircuit(input):
    Assign variable names to all outputs (and the initial inputs) of the gates, if
the input is not like that already.
    for(i= 0 to n-variables):
        for(curr = 0 to the end of the equation):
```

```
        count current $ith$ variable.
        if(count ¿ 3):
          input.change(BUFFERIZE(variable))
  return boolCircuitFan(input);
```

”Bufferize” does what we described above, swapping variables and adding new ones, etc.

Now that we have A ≤ B, all we need is A ≤ C This is very trivial to do, depending on how the input is structured to boolCircuit. If we are given an actual circuit (or diagram of one), then we have wires in that circuit, which we can test if they cross, by stretching them so they are individual line segments going to two vertices, and then testing intersection between those line segments. So basically, if the input is already a pseudo-directed-graph, then we are good, we just need to search the edges for intersection. If not, we must convert it to that so we can understand the layout of the circuits, but this should take polynomial time to do.

Let's say we have two arbitrary inputs, one coming from the left and one from the right. If they intersect, we need to make it so that there is a gate sequence that will make it so that the two do not intersect, but the desired output of the left will now be on the right side of the intersection, and the desired output of the right is now on the left side of the prior intersection.

So once an intersection is detected, all you need to do is XOR the two inputs, then re-XOR that output with the given side to get the other input. So if we have two inputs, a on the left and b on the right, the XOR would be in the middle of those two. Then you take the input of a and XOR the output of the previous XOR, and that will give you b. Do the same with b and it will give you a. Since we aren't allowed to do XORs, I attached a nice picture for you, using only ANDs and NOTs. As you can see in the picture, no wire is crossing, and as I tested in Logisim, the desired outputs are now on the sides they need to go to. Do this for every intersection, and there will be no intersection.

The pseudocode would look something like this:

```
boolCircuit(input):
  Create a graph to detect intersections, and put all intersections, and the inputs that intersect into a collection
  for(i= 0 to set.length):
     input.change(UNCROSS(set.get(i)
  return boolCircuitPlanar(input);
```

The UNCROSS function is how we described it above with the XORs/picture.

Since: B ≤ A
C ≤ A
A ≤ B
A ≤ C

If there is a polynomial time algorithm for one of the three algorithms, all the algorithms will be polynomial time.

10. Show that the clique problem is self-reducible. The decision problem is to take a graph G and an integer k and decide if G has a clique of size k or not. The optimization problem takes a graph G and returns a largest clique in G. So you must show that if the decision problem has a polynomial time algorithm then the optimization problem also has a polynomial time algorithm. Recall that a clique is a collection of mutually adjacent vertices.

Solution: Observe that a clique can be no larger than size n, where n is the total number of vertices in G. Then, we originally choose our largest clique size (call it j) to be 0. We then check to see if there exists a clique of size $j+1$ in G. If so, increment j and check again. If not, return j. (Note that clique of size m contains cliques of every size $i < m$, so if no cliques exist of size i, none can exist of size m.) Since we call the decision problem at most n times, the clique problem must have a polynomial time algorithm if the clique decision problem has a polynomial time algorithm. Hence, the clique problem is self-reducible.

15. Consider the following variant of the MST problem. The input consists of an undirected graph G and an integer k. The problem is to find a spanning tree T of G such that the degree of each node in T is at most k, or report that no such tree exists. Show by reduction that if this problem has a polynomial time algorithm then the Hamiltonian path problem has a polynomial time algorithm. The Hamiltonian path problem asks you to determine whether a graph has a simple path that spans the vertices.

    Solution: We will reduce the Hamiltonian path problem to the variant MST problem. For any graph G, we can call the variant MST problem with inputs G and $k = 2$. Then, the variant MST problem determines if there is a spanning tree T such that each node of the tree has degree at most 2. We argue that a spanning tree where each node has at most degree 2 is equivalent to a simple path that spans the vertices. The tree must have $n-2$ nodes of degree 2 and 2 nodes of degree 1. (Since there must be at least $n-1$ edges for all vertices to be connected, the total of the degrees of all vertices must be at least $2*(n-1)$. We must have at most 2 nodes of degree 1 to satisfty this requirement. We also must have at least 2 nodes of degree 1 as the leaves of our tree. Thus, we must have exasctly 2 nodes of degree 1.) The tree is then a simple path from one leaf node to the other that automatically spans all of the vertices (by the maximum degree being 2). This shows that a spanning tree with each node having at most degree 2 must be a simple path that spans the vertices. Suppose there exists some simple path that spans the vertices that is not a spanning tree with every node having at most degree 2. A simple path (with no repeated vertices) is a tree by default, and it must be a spanning tree since it spans the vertices. Since a simple path cannot repeat any vertices, no vertex can have degree more than 2. Hence, a spanning tree of G where every node has degree at most 2 exists if and only if a simple path in G exists that spans all vertices. Hence, our output is correct. Since we did no real work in solving the Hamiltonian path problem other than calling the variant MST problem, we have that if a polynomial time algorithm exists for the variant MST problem, then a polynomial time algorithm exists for the Hamiltonian path problem.