Answer the questions below.

1. The Hamiltonian path problem asks if there is a path in a graph that visits every vertex exactly once. Give a polynomial time reduction of the Hamiltonian path problem (in an undirected graph) to the CNF SAT problem.

https://www.udacity.com/course/viewer#!/c-ud557/l-1209378918/m-2463548775

Completed

2. Show that if P = NP then there is a polynomial-time computable function f such that if ϕ is a satisfiable boolean formula then $f(\phi)$ is a satisfying assignment. Hint: Use a polynomial-time algorithm for satisfiability repeatedly to find an assignment variable by variable.

   The document at this location helped: http://cseweb.ucsd.edu/~mihir/cse200/decision-search.pdf

   As Prof. Brubaker noted in the forum, P=NP implies that there is a polynomial-time algorithm for satisfiability. Let us call this function (or *oracle*) g. Here is how we can use g to find a satisfying assignment for f($\phi(x_1, x_2,\ldots,x_n)$):

   - Call g on ϕ. If g returns FALSE, then return FALSE
   - If g returns true, we know that a satisfying assignment exists
     - Reduce the complexity if the problem by one variable by assigning a truth value to the selected variable and reducing the problem. Note: This will actually create two Boolean formulas, one for setting the selected variable to TRUE and one for the selected variable being FALSE
     - One of these reduced formulas has to be true (because the unreduced formula is satisfiable and we have covered all the possible assignments of the variable we eliminated). Use oracle g to determine if either is true. In the event that both reduced formulas happen to be true, assign the variable to FALSE.
     - We already have a method to solve the newly reduced formula, namely the one we just described above. Call the method again, but with the newly reduced formula as long as the number of variables remaining is greater than 1

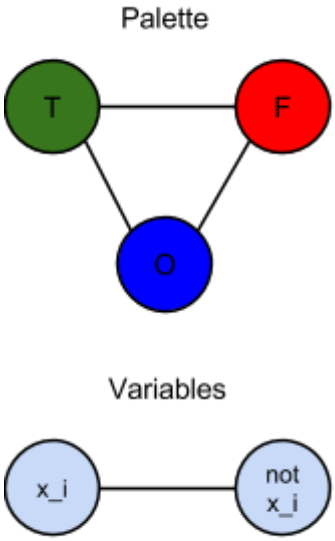
   In pseudo-code:

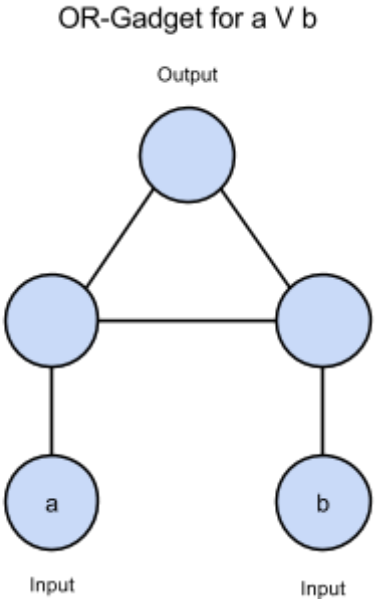
   Note: Since this function is defined recursively, the value for n will change

```
f(φ(x1,…,xn)):
      if g(φ(x1,…,xn)) = FALSE
            return 0
      else
            assignment = []
            if g(φ(0,…,xn)) = TRUE
                  assignment.add(0)
                  if(n > 1) f(φ(0,…,xn) )
            else
                  assignment.add(1)
                  if(n > 1) f(φ(1,…,xn) )
                  return assignment
```

   Function f still runs in polynomial time. We call polynomial-time function g n^2 times. Taking the product of these two running times results in a running time that is still polynomial.

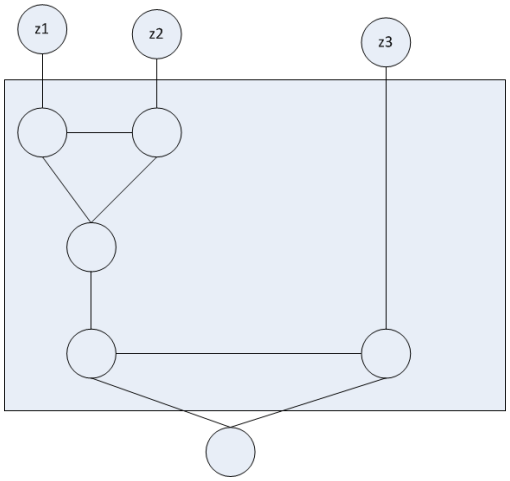
3. A k-coloring of a graph is an assignment of one of k colors to each vertex of the graph such that no edge is incident on vertices of the same color. Show that 3-coloring a graph is NP-complete. (Hint: Reduce 3-CNFSAT. You may find the subgraphs below useful).



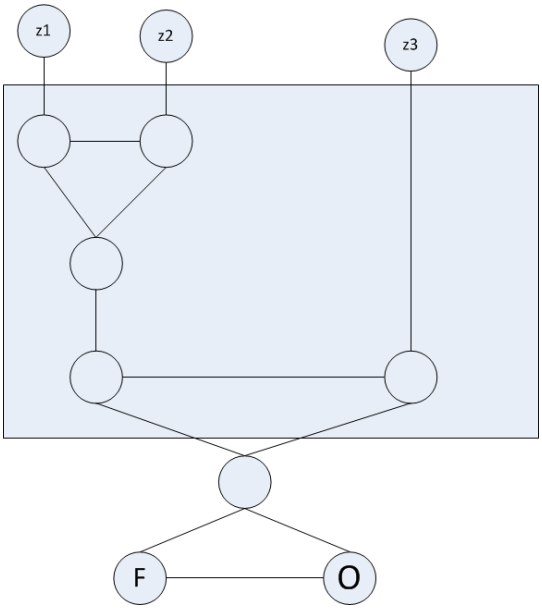


In the course of looking up information on the use of gadgets for reductions, I found this Wikipedia entry that I found useful: http://en.wikipedia.org/wiki/Gadget_(computer_science), particularly this image: http://en.wikipedia.org/wiki/Gadget_(computer_science)#mediaviewer/File:3SAT-3COL_reduction.svg

The first step in reducing 3-CNFSAT to 3-coloring a graph is to reduce each of the individual clauses to a graph. The provided OR-gadget needs to be extended to allow for three inputs. This is accomplished by taking the output from the OR-gadget using the first two literals of the clause and feeding it as input into a second OR-gadget with the remaining input coming from the last literal. In other words, we have a diagram like this:



However, the output of the 3-input OR is constrained to be true since in order for the CNF formula to be true, each of the clauses must evaluate to true, so we need modify it to be as below:



This is our basic 3-input OR building block.

The final constraint is on how we define the input. In order to satisfy the 3-coloring rule, we need pairs of variables. In other words, we use the "variables" graph given in the hint. Then, we connect these variables to then inputs of the OR gates according to the CNF formula to construct the final graph.

I would illustrate this, but the Wikipedia image referenced above works nicely. However, I would revise it in a couple ways. First, the shared "ground" does not need to be connected to all the literals. By construction those 3-colorings will always apply, so those edges provide no more information or constraints. Second, the "False" and "Ground" nodes are shared between all the OR gadgets, but it works just as well for them to be a part of the complete OR-gadget as in my construction. This required more nodes, but putting the OR-gadgets together is simpler if they are part of the OR-gadget itself.

Now that we have constructed the reduction, we need to show that it is valid. We need to show that if the 3-CNF formula is satisfiable, then a 3-coloring exists **and** if a 3-coloring exists, then the 3-CNF formula is satisfiable.

To prove the first one, note that if a 3-CNF formula is satisfiable, then each of the clauses evaluates to true, so one or more literals in each clause must be true. A 3-coloring using the OR-gadgets constructed above is possible for every combination of True and False (order doesn't matter) except all False. Thus this direction is satisfied.

To prove the second one, note that if a 3-coloring exists that satisfies the OR-gadget constructed above, then the inputs into the gadget cannot be all False. In other words, at least one of the inputs must be True. This satisfies the second requirement.

So we have shown that the reduction is valid, now we need to show that the reduction takes place in polynomial time. Since the OR-gadgets are pre-built, the only connections that need to be made are between the inputs to the OR-gadgets and the literals themselves. This takes place in polynomial time.

4. Show that the following problem can be decided in polynomial time.
Input: A graph $G = (V, E)$.
Decision: Is there a 2-coloring of the graph?

This problem can be decided (and solved) in polynomial time through a Breadth First Search approach. Select one node as the start node. Color this node one color, say red. Next, color all of this vertex's neighbors the other color, say blue. Continue this process alternating colors. If the search comes across connected nodes that are the same color, then a 2-coloring of the graph is not possible and FALSE is returned. If the search completes no connected nodes are found that have the same color, then a 2-coloring of the graph is possible and TRUE is returned. The running time of this algorithm is the same as BFS, $O(v^2)$ where v is the number of vertices.

Note: an alternative solution would be to find the lengths of the various cycles in the graph G. A 2-coloring of a graph is possible if and only if there are no cycles of odd length in the graph (see http://mathworld.wolfram.com/BipartiteGraph.html). However, I was unable to come up with a polynomial time algorithm for finding the lengths of the cycles of the graph.

A final way of solving this would be by reducing this problem to a problem known to be in P. If this can be done, we know that determining the possibility of 2-coloring a graph is in P by transitivity of reduction. Two-coloring a graph can be reduced to 2-CNF, which from lecture we know to be in P. We can do this again by using a Breadth First Search over the input graph to construct 2-literal clauses. These clauses take the following form:

$$(v_1 \vee v_2) \wedge (\overline{v_1} \vee \overline{v_2})$$

This formula ensures that a pair of vertices v_1 and v_2 cannot have the same coloring (or truth assignment). Using the BFS algorithm, we construct such clauses for every pair of vertices that have an edge between them and AND them all together to form the 2-CNF formula. Again, the running time of this reduction is the same as the BFS algorithm, which is $O(n^2)$.

5. Given a complete graph G = (V, E) with non-negative integer costs on the edges $c: E \to Z^+$ and an integer bound k, the Traveling Salesperson problem (TSP) is to decide if there is a Hamiltonian cycle in G whose total cost is at most k. Recall that a Hamiltonian cycle visits every vertex once.

   Metric TSP is the constrained case where the edge cost function c satisfies the triangle inequality. That is, for all vertices $u, v, w \in V$, we have $c(u, w) \leq c(u, v) + c(v, w)$.

   Give a polynomial time reduction from TSP problem to Metric TSP. Hint: You only need to change the capacity function *c.*

   In the definition for the TSP, the capacity (I will call it cost) function can be defined arbitrarily. In other words, we can have two edges that are the same length, but they can have different costs. Similarly, one edge could be longer than another edge, but have less cost than the shorter edge. Thus, in the plain TSP, the triangle inequality is not satisfied by default.

   Since all we have to do is satisfy the triangle inequality, all we need to do is ensure that the sum of the lengths of any two sides of the triangle (this is a fully connected graph so there is a triangle between any 3 vertices) is greater than or equal to the remaining side. In other words, we don't have to ensure that the law of cosines, Pythagorean's Theorem, etc., hold. We can ensure this by searching the graph for the greatest cost and adding that cost to all the old costs. In other words,

$$M = max_{i,j} c(i, j)$$
$$c'(i, j) = c(i, j) + M$$

   Then, for any u,v,w

$$c'(u, v) + c'(v, w) \geq 2M \geq c'(u, w)$$
$$c'(u, w) = c(u, w) + M \leq 2M$$

   So we are guaranteed to satisfy the triangle inequality.

   This reduction takes place in polynomial time. The number of costs that need to be searched in order to find the maximum grows at most quadratically as the number of vertices is increased and the time to rescale the costs grows linearly. Summed together this is still polynomial.

6. The Knapsack problem may be stated as follows. Given $n$items with weights $w_1 \ldots w_n$ and corresponding values $v_1 \ldots v_n$, is there a subset of items $S \subseteq \{1 \ldots n\}$such that the sum of the weights is at most some capacity W (i.e.$\sum_{i \in S} \quad w_i \leq W$) and the sum of the values is at least V(i.e. $\sum_{i \in S} \quad v_i \geq V$)? The traditional interpretation is that a robber wants to rob an art gallery and make off with most valuable collection of pieces that he can carry in his knapsack. Show that Knapsack is NP-complete (Hint: reduce from Subset-Sum).

First, we need to show that Knapsack is in NP. Second, we need to show that every problem in NP can be reduced to the Knapsack problem. This will be done by transitivity by reducing the Subset Sum problem to Knapsack in polynomial time.

1. Show that Knapsack is in NP.
   It is very easy to verify if a collection of weights sums to no more than a given value and that a collection of values sums to at least some number. In fact, this computation is linear in the number of weights and values to sum.
2. Show that every problem in NP can be reduced to Knapsack.
   As previously mentioned, this will be done by reducing Subset Sum to Knapsack.

   Every Subset Sum problem can be converted into an equivalent Knapsack problem by setting the appropriate Knapsack values to the corresponding Subset Sum values. Using the notation defined in lecture for Subset Sum, we define the following

$$w_i = v_i = a_i$$
$$W = V = k$$

   In other words, we define a Knapsack problem where the weights and values are identical to the subset in the Subset Sum problem and where the maximum sum of the weights and the minimum sum of the values is the same as the desired sum of the subset sum problem. This reduction takes place in polynomial time because it is a simple assignment of variables.

   This reduction works because if there is a subset that sums to *k*, then the weights will be equal to W and the values will be equal to V. Since the relationships for the Knapsack problem are "or equal to," this satisfies both conditions. If there is not a subset that sums to *k*, then one of the conditions will not be met. Either the sum will be greater than the maximum weight or less than the desired value. Thus we have reduced Subset Sum to Knapsack. By transitivity of reduction, since Subset Sum is NP-complete, Knapsack is also NP-complete.