Review Session Handout [Decidability & NP]

1. Decidability Review. Suppose you have the following language L:

L = { R | R is a regular expression, with at least one w L(R), s.t. 101 is substring of w }

Is this language decidable / undecidable?

Answer: This problem is decidable; you cannot apply Rice's Theorem to demonstrate this problem's undecidability. Rice's Theorem applies to properties of the recursively enumerable languages (a property is some subset of the RE languages, with one such property being the regular languages). To say a property is undecidable means we cannot say, given a machine <M> if L(M) is in the property (the subset we picked). So, given a machine M, you cannot always tell if the language described by the machine is regular. But, given e.g. two DFAs (whose languages are already known to be regular) and asked to decide their intersection, it is possible to decide this problem as we are not dealing with whether the language of a Turing machine belongs to a particular set. This is why it is possible to decide anything about regular languages, even though one cannot decide whether the language of a particular Turing machine is regular.
Having said this, one can easily design an algorithm to decide the above language L as follows:

- Convert the regular expression to a DFA
- Mark all states reachable from the start state as belonging to group "A"
- Mark all states that reach the finish state as belonging to group "B"
- For each edge between states x and y in the DFA marked "0":
    - See if a state in A transitions to x on a "1"
    - See if y transitions to a state in B on "1"
    - If both conditions are met for some x,y, then "101" is part of some w L(R), so *accept*
- Else, *reject*

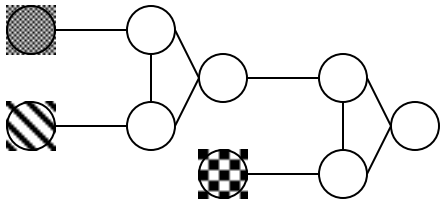
2. 3-Coloring. The *3-Coloring problem* requires one decided, when given a graph, whether it is possible to color a graph with 3 colors (making sure that adjacent nodes are of different color). This is a problem found in the homework problems section [7.34] of Chapter 7 in Sipser (please refer to it for diagrams elaborating on the terms mentioned here). Show that this problem is NP-complete.

Answer: First we must demonstrate that the problem is in NP. This is a simple task since, given a coloring with 3-colors in a graph, we can simply verify that the adjacent vertices (up to n-1, if the graph has n-nodes) have different colors. This can be verified in polynomial time, and hence the problem is in NP.
A reduction from 3SAT will demonstrate that all problems in NP reduce to an instance of the 3-coloring problem. The reduction has several parts:

1) Construct a palette graph which is a 3-clique; this will require 3 colors to color in correctly. We can designate the three colors as "True", "False", and "Dummy", as we will use this palette to enforce certain colorings later.

2) For each variable "x" in the original 3SAT expression construct two nodes, one representing "x" and the other its complement. Since these nodes are connected they cannot be the same color. Further, we must restrict them to be one of two colors – "True" or "False." Thus , we connect each node to the "Dummy" node in the palette graph above, preventing either from assuming the "Dummy" color.

3) For each clause of the form (x_1 v x_2 v x_3), use the associative rule to group it as ((x_1 v x_2) v x_3) and connect each group of literal nodes referenced together using the OR-gadget. In the example given here, this would yield



where the colored nodes represent x_1, x_2, and x_3 (top to bottom, respectively). Note the OR-gadget ensures that if both bottom nodes are colored "False," the top node must be colored "False" as well (a similar argument applies if both bottom nodes are colored "True" – the top node must then be colored "True" as well). However, if one of the two is "True," then it is possible to color the gadget such that the output is not "True" (it could be "Dummy" or "False"). So, when SAT is satisfied the output of the clause must be true, so we can enforce the output of the clause graph to be colored "True" by connecting the output node to both the "False" and "Dummy" nodes in the palette (if the graph can be successfully colored at all).

This graph can be colored using 3-colors iff the corresponding 3SAT boolean expression is satisfiable (the graph expresses the same constraints as the original 3SAT expression). Thus, we have completed the proof that 3-coloring is NP-complete.

3. Star-Free-Inequivalence. This problem is concerned with regular expressions that have no star occurring within them and only make use of the union and concatenation operators. The language we are considering is:

$$L = \{ (R1, R2) \mid L(R1) \neq L(R2) \text{ and } R1, R2 \text{ are both *-free regular expressions } \}$$

Demonstrate that this language is NP-complete.

Answer: Without the *-operator, we may note that any string in L(R) for some regular expression R has length $\leq |R|$ (to see this consider, a single symbol regular expression, then the set of strings produced by the union of two regular expressions and then the set produced by the concatenation of two regular expressions – the argument proceeds from this basis inductively). So, to establish membership in the language, we need only exhibit one string that is L(R1) and not in L(R2) or vice versa. A NTM may guess a string to try in each direction, and then determine the result in polynomial time (the guesses may be generated in O(|R1) + |R2|) time). Thus, this problem is in NP.

To demonstrate NP-completeness, we will reduce from SAT (note we are assuming the boolean expression is in CNF). Suppose there are variables x_1 ... x_n and clauses C_1 ... C_m in the original boolean expression. We will convert these into two *-free regular expressions over the alphabet{0,1}. The first regular expression (R1) will be the union of all failing assignments to the variables (i.e., if x_i is assigned true then the ith character of the reg-expression is "1," else it is "0.") R2, then, is simply the regular expression (0+1)(0+1)....(0+1) (where (0+1) is repeated n times), which represents all possible assignments to the n variables. Note if L(R1) = L(R2), then all assignments are failing, and thus the corresponding SAT formula cannot be satisfied. If they are not equal, then some assignment exists that is not a failing assignment (i.e., is not in L(R1)). Thus, the original boolean formula in this case is satisfiable, and the reduction is complete.
The only remaining detail is how one may construct R1. An assignment can fail if any one clause of the boolean expression yields a false value. For a given clause, we can generate a regular expression (a_1 ... a_n) that corresponds to a failing assignment by making sure all literals are false as follows:

- Set a_i to be "0" if x_i appears in the clause under question
- Set a_i to be "1" if the complement of x_i appears in the clause under question
- Otherwise, set a_i to be (0+1), since the corresponding x_i does not appear in the clause under question.

Then, R1 is simply formed by taking the union of these regular expressions for each clause C_1 ... C_m.

4. Satisfying assignment for SAT instance. Assuming that P=NP, give a polynomial time algorithm which finds a satisfying assignment for a boolean expression φ, if it can be satisfied.

Answer: Since P=NP, we know SAT is now in P; we will assume we have a DTM that can decide SAT in polynomial time (call it A). We can build a new machine B which finds a satisfying assignment in polynomial time using the following strategy:

- Run A on φ. If it rejects, then reject the input.
- Else, φ can be satisfied. Begin with x_1, and guess that the value assigned to it is "True." Modify φ to obtain φ' in which x_i has been replaced with the value "True"; this can be done in time linear in the length of φ.
- Run A on φ'. If it accepts, then proceed. Otherwise, x_1 must be "False", so modify φ accordingly to obtain the correct φ' and then proceed.
- Set φ to be φ', and repeat the above process for each of the remaining variables in the formula. When this completes, the satisfying assignment will be known.

A runs in polynomial time, and we run this once for each variable. Further, for each variable, we make an O(n) pass over the boolean formula as we make an assignment to a variable. Thus, this yields a net polynomial running time for the algorithm B above.