CS 270 Lab Extra Credit 1 (Satisfiability and SAT Solvers)

Week 8 - Nov. 13 – Nov. 17, 2017.

Name 1:_____

Drexel Username 1:_____

Instructions: For this extra credit lab you must submit your own solution. There are two problems both relating to satisfiability and SAT solvers. You are to use the MiniSat SAT solver as demonstrated in class. You can access MiniSAT through either the browser interface (see the link from the MiniSAT lecture outline on the course webpage) or through /home/jjohnson/bin/MiniSat on tux.

This lab introduces material from complexity theory – in particular NP-completeness. NP stands for nondeterministic polynomial time, and is the class of decision problems (determine if a given problem instance has a solution or not) that that can be verified in polynomial time. By polynomial time we mean there is an algorithm whose computing time is some polynomial function of the input size. The class of problems, P, for which there are polynomial time solutions, as compared to exponential time solutions, are considered those which have fast solutions. Think of the class NP as those problems which are easy to check but potentially hard to solve. For example satisfiability is such a problem. In this case you want to determine if a given Boolean formula has an assignment of truth values to variables such that the formula is true. Given a proposed assignment it is easy to check, simply evaluate the expression, if the formula is true for the given assignment.

Clearly those problems that have polynomial algorithms to construct the solution are in NP. However, to be in the class it is only necessary to be able to check the solution quickly; we are allowed to magically guess (this is the nondeterministic part) the solution and simply have to check if the proposed solution is a solution. The question of whether all problems in NP have polynomial time algorithms to construct solutions (the P = NP problem) is the most famous outstanding problem in Computer Science.

The hardest class of problems in NP are those that are said to be NP-complete. A problem is NP-complete if a polynomial solution to it can be used to solve every other problem in NP in polynomial time. Thus finding a polynomial time solution to an NP-complete problem would prove that P=NP. The first problem shown to be NP-complete was the satisfiability problem. It was shown by Cook and Levin that any problem in NP can be reduced to satisfiability in polynomial time. In this lab you will show how to reduce two problems to satisfiability and moreover, will construct the reduction so that the problems can be solved using a SAT solver. The first problem shows how arithmetic, a problem in P, can translated into a satisfiability problem where a SAT solver can find the solution to a+x=b through an assignment of Boolean variables. The second problem, which is NP-complete, is to determine if a graph (a collection of vertices and edges that connect some pairs of vertices) has a Hamiltonian path (a permutation of vertices such that there is an edge between consecutive vertices in the permutation). In this case solving an assignment corresponding to a solution of the corresponding satisfiability problem will provide a permutation of the vertices that is a Hamiltonian path. To show that the Hamiltonian path problem is NP-complete we must also show that satisfiability can be encoded as a Hamiltonian path problem, which we will not do here.

1. [Reducing arithmetic to satisfiability.] In this problem you will generate an instance of the satisfiability problem that corresponds to binary addition, and then you will use MiniSat to find an assignment that provides a specified sum. Recall from Lab 2 that a full adder with inputs a, b and cin is defined by a+b+cin = 2*cout + sum and that we found Boolean expressions for sum and cout

   FullAdder(a,b,cin;sum,cout) ≡
   sum = a ⊕ b ⊕ cin, where ⊕ is xor.
   cout = (a ∧ b) ∨ (a ∧ cin) ∨ (b ∧ cin)

   These equations can be represented by the Boolean expressions

   sum ↔ a ⊕ b ⊕ cin, where ⊕ is xor.
   cout ↔ (a ∧ b) ∨ (a ∧ cin) ∨ (b ∧ cin)

   and an assignment to a, b, cin, sum and cout that makes these formulas true must satisfy the defining equation of a full adder. Moreover, if we specify values found a, cin, sum and cout, an assignment will provide the value of b that satisfies the full adder equation.

   Before doing this, try a simpler example to make sure you understand how to convert a logic circuit to a satisfiability problem and how to run and interpret the output using MiniSAT. Try for example c ↔ (a ∧ b).

   N-bit binary addition can be performed by chaining together N full adders. For example, 3-bit addition s = a+b with the binary representation of a = $a_2a_1a_0$, b = $b_2b_1b_0$ and s = $s_2s_1s_0$.

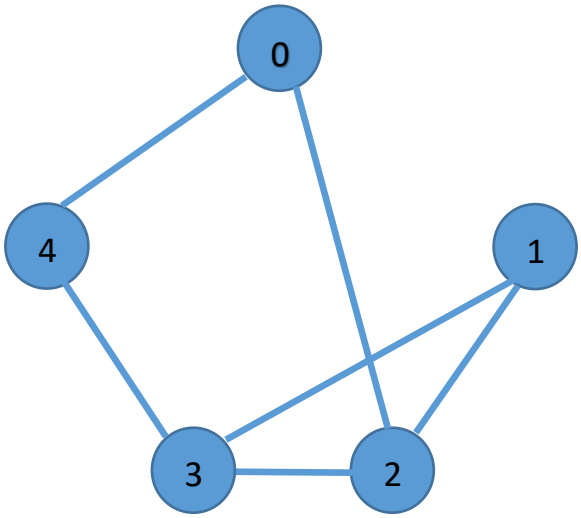
   FullAdder(a_0,b_0,0;s_0,$cout_0$) ∧ FullAdder(a_1,b_1, $cout_0$;s_1,$cout_1$) ∧ FullAdder(a_2,b_2, $cout_1$;s_2,$cout_2$)

   First convert the expressions for sum and cout to CNF. Then write a function FullAdder that generates input for MiniSAT in DIMACS format. Use this function to generate the satisfiability problem corresponding to 3-bit addition. Add clauses that specify a given binary sum and input a, and run MiniSAT to find the b, that satisfies a+b=sum.

   Note that converting the Boolean expressions to CNF using a truth table or the algorithm discussed in class can cause the formula to expand exponentially. An alternative approach for converting a logic circuit to CNF involves assigning variable for the output of each gate (Tseytin transformation – see the Wikipedia page). Variables are introduced using ↔ as was done for sum and cout above.

2. [Reducing Hamiltonian path to satisfiability] A graph G = (V,E) is a set of nodes V = {v_1,…,v_n} and a set of edges E = {e_1,…,e_t} where an edge connects a pair of vertices v_i and v_j. We will only deal with undirected graphs so that the order of the vertices does not matter. An undirected graph

can be drawn with dots for vertices and lines connecting the pair of vertices for an edge. A graph can be represented in several ways. For our purposes we will simply give the number of vertices, n, and represent the set of vertices by the numbers 0,…,n-1. We will use an adjacency matrix to record the edges: The (i,j) and (j,i) entries of the matrix will be set to 1 if there is an edge connecting vertices i and j and 0 otherwise. The following is an example of a graph with 5 vertices and 6 edges. It's adjacency matrix has 1's in locations (0,2), (0,4), (1,2), (1,3), (2,0), (2,1), (2,3),(3,1),(3,2),(3,4), (4,0) and (4,3). It has a Hamiltonian path through vertices 0,2,1,3,4.



Determining whether a graph has a Hamiltonian path can be translated into a satisfiability problem. Let the variable X_{ij}, for $0 \le i,j < n$, be true if the ith vertex in the Hamiltonian path is vertex j. In the Hamiltonian path above the variables X_{00}, X_{12}, X_{21}, X_{33}, and X_{44} would be true and all other variables would be false. The constraints on the variables X_{ij} for a Hamiltonian path are as follows:

a. [Each node j must appear in the path] $X_{1j} \vee \ldots \vee X_{nj}$ for j=1,…,n
b. [No node can appear twice in the path] $X_{ij} \rightarrow \neg X_{kj}$ for all i,j,k with $i \neq j$
c. [Every position i on the path must be occupied] $X_{i1} \vee \ldots \vee X_{in}$ for i=1,…,n
d. [Only one node in each position i] $X_{ij} \rightarrow \neg X_{ik}$ for all i,j,k with $j \neq k$
e. [Only nodes connected by an edge can be adjacent in the path] $X_{i,j} \rightarrow \neg X_{i+1,k}$ for i=1,…,n-1 and all $(j,k) \notin E$.

Note that the number of variables is n^2 and the number of constraints is $O(n^3)$, so that the reduction to satisfiability only increase the problem size by a polynomial.

Create an instance of the satisfiability problem for MiniSAT using DIMACS format for the input for the Hamiltonian path problem for the graph above. Run MiniSat and see if it finds a Hamiltonian path. The input for this problem is too large to do by hand, so you should write a program to generate the input clauses. Such a program would need to read the graph, which

can be done by reading the number of vertices n and the edges. You should construct the adjacency matrix so that it is easy to determine the missing edges as you construct constraint e).

Before writing the program and trying to solve this particular problem, you should try an easier example. E.G. a graph with 3 nodes {0,1,2} and with edges (0,1), (1,0), (1,2) and (2,1).