ECE208 Extra Credit Assignment

General Instructions: This programming assignment is worth a total of 10 extra credit points. There are 4 programming questions in this assignment. In order to get full points, you have write programs that adhere to the specifications provided below. Additionally, you must implement error-handling code, i.e., your code must be able to handle erroneous inputs appropriately. I will review your code and I may subject your code to my own tests. The input and output formats for all questions are well specified below, and hence you should be able to write your own tests. You are also required to submit a report describing your code and results for each question (at least 1 page per question in 11 point font). You may have up to 2 members in your respective teams.

Team Composition: You have to declare the names, UW IDs, and email IDs of your team members to the TAs via email. This has to be done asap. If you want to change your team subsequently, please consult with us.

Marking Scheme: The marks per question are provided below. The marks you get in this assignment will be added to your final marks you receive for your regular course work. If this sum happens to be greater than 100, your total final marks will be 100.

Choice of Programming Languages: You may choose one of C, C++, Java, or Python. While we don't disallow other languages, we prefer the ones specified here. Chances are that it is more productive for you to write efficient code for the specified programs (such as solvers and transformers) in these languages.

Due Date: This assignment is due at the end of the S2019 term.

1 Definitions

The NNF Format

First, we define the textual format for Boolean formulas in negation normal form (NNF). It is assumed that you are familiar with the inductive definition of NNF formulas over connectives $\neg, \vee, \wedge$ and left/right parentheses.

- Every Boolean variable is represented as a 32-bit number, written in decimal (just as the int datatype in C++ or Java). For example, the number "1" corresponds to the Boolean variable x_1. The number "0" is reserved for end of line.
- Negated variables are represented by negative numbers, e.g., $\neg x_1$ is represented as -1, and x_{199} is represented as 199.
- The Boolean connective $\vee$ (OR) is represented by $+$ symbol. The Boolean connective $\wedge$ (AND) is represented by the . symbol. You may use the left and right parentheses to demarcate sub-formulas.

You may assume that the constants $\bot$ and $\top$ have been simplified away from all input formulas. If you really need to represent these constants, you can use suitable formulas over variables (e.g., $\bot$ is equivalent to $(x) \wedge (\neg x)$).

The CNF Format

Here we define the textual format for Boolean formulas in conjunctive normal form (CNF). It is assumed that you are familiar with the inductive definition of CNF. (This text format is the official DIMACS format used by all SAT solvers).

- Every Boolean variable is represented by a 32-bit number, exactly as in NNF. For example, the number "1" corresponds to the Boolean variable x_1. The number "0" is reserved for end of line.
- Negated variables are represented by negative numbers, e.g., $\neg x_1$ is represented as -1.
- Every clause is represented by a space separated list of numbers, ending in a 0 to denote end of line. That is, the Boolean connective OR in disjunctive clauses is represented by a white space.

  E.g., the clause $(x_{42} \vee \neg x_1 \vee x_{100} \vee x_{22} \vee \neg x_{90})$ would be written as 42 -1 100 22 -90 0.
- A formula in CNF is a list of clauses, one clause per line. (The AND is implicit.)

You may assume that the constants $\bot$ and $\top$ have been simplified away from all input formulas. If you really need to represent these constants, you can use suitable formulas over variables (e.g., $\bot$ is equivalent to $(x) \wedge (\neg x)$).

Question 1: Tseitin Transformer (maximum 2 points)

Write a Tseitin transformer that takes as input a Boolean formula F(x) in NNF and outputs an equisatisfiable formula G(y,x) in CNF. Both input and output must be text files.

Question 2: DPLL SAT Solver (maximum 4 points)

Write a DPLL SAT solver as specified in lecture 2 slides. Your solver must take as input CNF formulas in DIMACS format, and determine their satisfiability. Output has to be the string "SAT" (respectively, "UNSAT") printed to screen, if the input is satisfiable (respectively, unsatisfiable). Perhaps the hardest task here is to write the BCP routine, given there are many different kinds of BCP implementations. Please use the greedy BCP specified in class.

Question 3: Solvers and Symbolic Execution-based Testing (maximum 2 points)

Dynamic systematic testing (aka dynamic symbolic testing or concolic testing) is aimed at both 1) automatic systematic test coverage, and 2) finding deep security vulnerabilities. For convenience, I will interchangeably use the term concolic testing and symbolic-execution based testing (or techniques). Symbolic execution is one of the biggest industrial application of SAT solvers.

We learnt in class the power of the combination of symbolic execution, concretization and constraint solvers (e.g., SAT solvers) in automatically finding security vulnerabilities and systematically testing software. In this question, we will analyze the role of combining symbolic execution with concrete execution (aka concolic testing), and see how it is more powerful than either symbolic or concrete execution by themselves. (The term concolic is a portmanteau of concrete and symbolic.)

Definitions

Concrete Execution: The term *concrete execution* refers to the normal execution or *run* of a program P on a computer on *concrete* inputs, i.e., the inputs to the program P are values from P's input domain.

Program Path or Trace: A program path (simply path or trace) is the sequence of instructions executed by a program on a concrete or symbolic execution.

Symbolic Execution: The term *symbolic execution* refers to an execution or *run* of a program P on *symbolic* inputs (i.e., inputs are not concrete) but instead range over all values from the input domain of the program P. Symbolic execution of a program P can be acheived on a computer by executing the program P symbolically using an interpreter or a symbolic virtual machine, e.g., the KLEE symbolic virtual machine (http://klee.llvm.org).

A symbolic virtual machine constructs a map from program variables to logic expressions over input variables. The symbolic virtual machine executes a program symbolically by stepping through the program one instruction at a time, and updating the map from program variables to logic expressions. As the program is executed by the symbolic virtual machine, this map is updated to reflect the most current symbolic expression for each program variable. The result of symbolic execution is a path constraint in a suitable logic, defined below.

Path Constraint: Given a program path $p(\bar{x})$ in a program whose inputs are denoted by $\bar{x}$, the *path constraint* corresponding to $p(\bar{x})$ refers to the logic formula $\phi(\bar{x})$ that is constructed as follows: Step 1: Symbolically execute a chosen path $p(\bar{x})$ in the given program; Step 2: Construct a conjunction of equalities between program variables and their final symbolic expression values. This formula, over the inputs $\bar{x}$ to the program, is called the path constraint and compactly represents all input values that will exercise the path $p(\bar{x})$.

Systematic testing using Symbolic Execution, Concretization and Solvers

Now that we know how to symbolically execute paths of a program and what a constraint solver is, it is easy to see how the two can be combined to systematically test every path in the program: First, we symbolically execute a program path to obtain a path constraint. Second, the path constraint is solved by a constraint solver to obtain an assignment of values to the input variables of the program-under-test, such that when the program is executed on these values it takes the path that was symbolically executed in the first step. By doing this systematically for every path in the program, we obtain a set of test inputs to concretely execute all paths in the program.

There are several problems that need to be overcome in order for such a systematic testing technique to become scalable.

- The path constraints may be too difficult for constraint solvers to solve
- The number of paths in a program is typically exponential in the number of if-conditionals (or branches) in the program (it can even be infinite if the program has an infinite loop)
- Symbolic execution is much slower than normal concrete execution

In this question, we will focus on the first problem, namely, constraints may be too difficult for solvers to solve. One way around this is to concretize certain inputs such that hard parts of constraints become easy to solve. This approach to executing the program symbolically, where parts of the input are concretized, is referred to as *concolic execution*, and has partially led to the amazing success of symbolic execution based techniques to scale to testing of very large programs (other reasons for the success of symbolic techniques include better program analysis and very efficient constraint solving). The question of which inputs should be concretized and which should be left symbolic is a heuristic, determined through appropriate automatic program analysis and the intuition of the user.

Symbolic Execution Questions

Question 3.1 (warm up exercise)

Consider the following piece of C code:

```
int obscure(int x, int y)
{
  if (x == hash(y)) {
      //call to function with error
  }
  else {
     //call to some function
  }
}
```

The function hash is some standard-issue cryptographic hash function like SHA1. In class, we learnt when a constraint solvers solves a constraint it is essentially inverting the function/relation encoded by the constraint. We also learnt that cryptographic hash functions are difficult for constraint solvers to invert. How would you systematically construct a test suite for the above code using the idea of concolic testing?

Question 3.2 (warm up exercise)

Consider the following piece of C code, where "constant" is a 128 bit long constant that is difficult for humans to guess:

```
int obscure(int x, int y, string * message)
{
  if (constant == hash(y,x,message)) {
     // call to function with error
  }
  else {
     //call to some function
  }
}
```

The function hash is some standard-issue cryptographic hash function like SHA1. The above piece of code is similar to the one in Question 1.1 with some important differences. Will your testing technique that you proposed as an answer to Question 1.1 work in this case? If yes, why? If no, why not? (A testing technique 'works' when you can use it to automatically test all the paths of a given program.)

Question 3.3: Programming Assignment with the KLEE

Please follow the tutorials at the following website about the KLEE symbolic virtual machine and tester:

https://klee.github.io/tutorials/

Use KLEE to find bugs in all of the following Coreutils. Describe the bugs found and suggest patches. The programs-under-test are:

- paste
- mkdir
- md5sum

All are version 6.10.

Question 4: SAT solvers and the Graph Vertex Cover Problem (2 point)

The vertex cover problem is a classic problem in graph theory and has many industrial applications. A vertex cover of a graph G is a set S of vertices such that each edge of the graph G is incident to at least one vertex of the set S. The decision version of the vertex cover problem is known to be NP-complete and there are well-known reductions from the vertex cover problem to SAT. The statement of the vertex cover problem is the following:

Given a graph G and a positive number k, does G have a vertex cover of size at most k?

How will you encode this problem as a CNF formula? Please write an encoder that takes as input the adjacency matrix of a graph and a number k, and generates an appropriate CNF formula F. Then use your solver to decide F, whose satisfiability answers the input graph cover problem.