CS 170 Algorithms HW 11
Fall 2014 David Wagner

Due Nov. 26, 6:00pm

Instructions. This homework is due **Wednesday**, November 26th, at 6:00pm electronically. Same rules as for prior homeworks. See `http://www-inst.cs.berkeley.edu/~cs170/fa14/hws/instruct.pdf` for the required format for writing up algorithms.

1. (15 pts.) Beyond suspicion

Most researchers suspect that $\mathbf{P} \neq \mathbf{NP}$. We also know that if $\mathbf{P} \neq \mathbf{NP}$, then 3SAT $\notin \mathbf{P}$. Therefore it seems reasonable to say that 3SAT $\notin \mathbf{P}$ is suspected to be true.

Mark each of the following statements as either True, False, Suspected True, or Suspected False. True means that the statement is provably true. Suspected True means that the statement has not been proven to be true, but it can be proven to follow from a conjecture that is widely suspected to hold (such as that $\mathbf{P} \neq \mathbf{NP}$). Similarly for False and Suspected False. You don't need to justify your answer.

(a) HAMILTONIAN PATH $\in \mathbf{P}$.

(b) HAMILTONIAN PATH $\in \mathbf{NP}$.

(c) HAMILTONIAN PATH is NP-complete.

(d) There is a polynomial-time algorithm for the 3-COLORING problem.

(e) There is no polynomial-time algorithm for the TRAVELLING SALESMAN problem.

(f) HAMILTONIAN PATH $\in \mathbf{P}$ if and only if 3-COLORING $\in \mathbf{P}$.

(g) 2-COLORING $\in \mathbf{P}$.

(h) 2-COLORING is NP-complete.

2. (20 pts.) Approximation algorithms

The Steiner tree problem is the following:

Input: An undirected graph $G = (V, E)$ with non-negative edge weights $\mathrm{wt} : E \to \mathbb{N}$, a set $S \subseteq V$ of special nodes.

Output: A Steiner tree for S whose total weight is minimal

A Steiner tree for S is a tree composed of edges from G that spans (connects) all of the special nodes S. In other words, a Steiner tree is a subset $E' \subseteq E$ of edges, such that for every $s, t \in S$, there is a path from s to t using only edges from E'. The total weight of the Steiner tree is the sum of the weights included in the tree, i.e., $\sum_{e \in E'} \mathrm{wt}(e)$.

The Steiner tree problem has many applications in different areas, including creating genealogy trees to represent the evolutionary tree of life, designing efficient networks, to even planning water pipes or heating ducts in buildings. Unfortunately, it is NP-hard.

Here is an approximation algorithm for this problem:

1. Compute the shortest distance between all pairs of special nodes. Use these distances to create a modified and complete graph $G' = (S, E_S)$, which uses the special nodes as vertices, and the weight of the edge between two vertices is the shortest distance between them.
2. Find the minimal spanning tree of G'. Call it M.
3. Reinterpret M to give us a Steiner tree for G: for edge in M, say an edge (r,s), select the edges in G that correspond to the shortest path from r to s. Put together all the selected edges to get a Steiner tree.

Prove that this algorithm achieves an approximation ratio of 2.

Hint #1: You might want to review the approximation algorithm for the TSP in Section 9.2.3 of the book.

Hint #2: Given the minimal-weight Steiner tree for G, construct a tour in G that visits each special vertex of G at least once. See if you can figure out how the following four quantities relate to each other: (a) the total weight of the minimal Steiner tree, (b) the total weight of this tour, (c) the weight of M, (d) the total weight of the Steiner tree output by the above approximation algorithm.

3. (20 pts.) Algorithm design practice

Here is a problem that arises in computational biology. Suppose we have a DNA sequence from a newly sequenced genome, and we are trying to detect whether a contains a particular gene. Unfortunately, the gene might not appear as a contiguous substring of the DNA sequence: it might be broken into several pieces, with other trash fragments interspersed in between these pieces. Therefore, we want to check whether there exists some way to break up the gene into pieces, so that the pieces appear in the correct order in the new DNA sequence.

We can formulate this as a question about intervals. If ℓ, r are two integers, we use $[\ell, r]$ to denote the interval of integers from ℓ to r, inclusive, i.e., $\{\ell, \ell+1, \ell+2, \ldots, r\}$. We are given k intervals, and a weight for each one; our goal is to find a subset of these that are non-overlapping and whose total weight is as large as possible. More precisely, the task is:

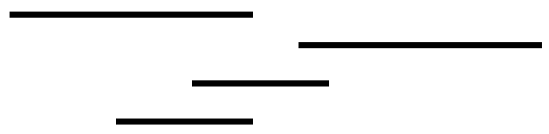
Input: intervals $[\ell_1, r_1], \ldots, [\ell_k, r_k]$ (where $1 \le \ell_i \le r_i \le n$); weights $w_1, \ldots, w_k$

Output: A subset of the intervals that don't overlap each other and whose total weight is maximal.

Design an efficient algorithm to solve the intervals problem above. It's enough to find the best total weight attainable—you don't have to find the corresponding set of intervals.

You can assume the intervals are sorted by increasing value of their right endpoint, so that $r_1 \le r_2 \le \cdots \le r_k$.

For instance, suppose we have the intervals $[1,10], [5,10], [7,13], [12,20]$, with weights $3,2,4,2$. You can visualize the intervals as shown below:



Then the optimum solution takes the intervals $[1,10]$ and $[12,20]$, for a total weight of 5.

Comment: If you're curious about what the interval problem has to do with recognizing genes, here is the relationship. We have one interval $[\ell_i, r_i]$ for each partial match between part of the gene and the DNA sequence: if $x[1..n]$ is the DNA sequence, $x[\ell_i..r_i]$ is one partial match for part of the gene. w_i is a score representing how good a match it is. We're looking for the best collection of partial matches, whose total match score is as high as possible, but where the partial matches don't overlap each other. However you don't have to understand any of the computational biology to solve this problem: you can just treat this as a question about intervals.

4. (15 pts.) Learn to use a SAT solver

This question will introduce you to a SAT solver and teach you how to use it. Most SAT solvers are not very user-friendly: they expect to be provided a boolean formula in CNF (Conjunctive Normal Form), in a particular input format. Therefore, I'm going to introduce you to STP, a solver that has a more user-friendly interface. STP works by pre-processing your input, to generate a boolean formula in CNF, and then invoking a SAT solver.

To use STP, you declare a number of boolean variables, write down some constraints on the boolean variables that must be true (some boolean formulas that must be true), and then STP tries to see whether it can find any assignment to the boolean variables that makes all of the constraints true. If it can find an assignment, it reports "Satisfiable." and outputs a satisfying assignment; otherwise, it reports "Unsatisfiable." You can declare boolean variables `x`, `y` like this:

```
x, y : BOOLEAN;
```

Let's say we want to test whether the formula $(\texttt{x} \vee \neg\texttt{y}) \wedge (\neg\texttt{x} \vee \texttt{y})$ is satisfiable. We're going to introduce a constraint that encodes this formula.

```
x, y : BOOLEAN;
ASSERT((x OR NOT(y)) AND (NOT(x) OR y));
```

Log into a Linux machine (`hive1.cs.berkeley.edu` - `hive24.cs.berkeley.edu`), store the constraints above into an input file, say `test1.in`, and run the following command:

```
/home/ff/cs170/bin/easystp test1.in
```

You should get the following output:

```
Satisfiable.
ASSERT( x  = FALSE  );
ASSERT( y  = FALSE  );
```

The first line tells you that the formula is satisfiable. The second line gives you an example of a satisfying assignment: namely, $\texttt{x} = \texttt{y} =$ false.

Note that STP can only be run from instructional machines that are running Linux. So if you get an error message, double-check that you are logged into `hive1.cs.berkeley.edu`-`hive24.cs.berkeley.edu`.

It is OK to introduce multiple `ASSERT` statements. The STP solver will look for a satisfying assignment that makes every `ASSERT` statement true. For instance, an equivalent way to do the above example would have been:

```
x, y : BOOLEAN;
ASSERT(x OR NOT(y));
ASSERT(NOT(x) OR y);
```

Try it; you'll see you get the same answer. STP supports other boolean operators as well: => is logical implication ($\texttt{x} \implies \texttt{y}$ can be encoded in STP as `x => y`); <=> is bidirectional implication, i.e., equality of booleans ($\texttt{x} = \texttt{y}$ could be encoded in STP as `x <=> y`); `XOR` is exclusive or; and `NAND` is nand (`x NAND y` is equivalent to `NOT(x AND y)`).

Here is another example. Let's say we want to test whether the formula $((\texttt{x} \implies \texttt{y}) \implies (\texttt{y} \implies \texttt{x})) \wedge (\texttt{x} \vee \texttt{y}) \wedge \neg\texttt{y}$ is satisfiable. One way to do so would be as follows:

```
x, y : BOOLEAN;
ASSERT(((x => y) => (y => x)) AND (x OR y) AND NOT(y));
```

Another approach would be to introduce temporary variables for some of the subexpressions, and break this down into multiple constraints:

```
x, y : BOOLEAN;
t, u : BOOLEAN;
ASSERT(t <=> (x => y));
ASSERT(u <=> (y => x));
ASSERT(t => u);
ASSERT(x OR y);
ASSERT(NOT(y));
```

Either way, the result is satisfiable, and STP can find a satisfying assignment for you.

SAT solvers make solving many logic puzzles so easy that it takes all the fun out of it. Here's one from Smullyan. You are visiting an obscure island, where every inhabitant is either a knight or a knave. Knights always tell the truth. Knaves always lie (they only tell falsehoods). Some of the inhabitans are werewolves and have the annoying habit of sometimes turning into wolves at night and devouring unlucky tourists. A werewolf can be either a knight or a knave. You run into three inhabitants, Alice, Bob, and Carol. Alice declares, "I am a werewolf." Bob states "I am a werewolf." Carol says "At most one of us is a knight." Is it possible that Alice is a werewolf?

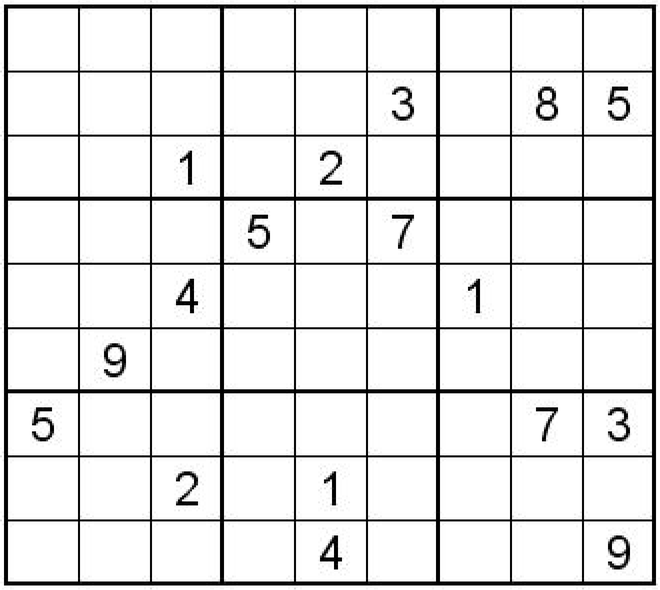
Solve this by expressing it as a satisfiability problem and then using STP to solve the satisfiability problem. You might want to introduce a boolean variable or two for each person indicating that person's status, and then encode their statements as assertions. Your homework solution should include three parts: (1) The final answer (could Alice be a werewolf?); (2) A list of the boolean variables you used and explanation of the meaning of each; and (3) A print-out or listing of the STP input file you use and of the output from STP.

5. (30 pts.) Sudoku

You may have heard of the game of Sudoku. Now you get a chance to write a computer program to solve it.

Suduku is a game with a 9×9 grid, broken down into a 3×3 arrangement of blocks, where each block contains 3×3 cells. Each block contains a permutation of the numbers 1,2,...,9 in some order (no number is repeated). Also, each row and each column contains a permutation of those numbers. Some of the cells are filled in for you, and your task is to fill in the rest. You can find more information on Sudoku on the web.

Your job is to solve the following Sudoku puzzle, using a SAT solver. (You wouldn't want to solve it by hand.)



					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

In particular, I want you to generate an input to STP, so that the output of STP tells you how to solve the Sudoku puzzle. I suggest that you think about how to come up with a bunch of boolean variables for each cell, where the values of the boolean variables associated with that cell tells you what number should go in that cell. Then, express the constraints that a valid solution to the Sudoku puzzle must satisfy as boolean formulas in terms of these boolean variables, and feed them into STP. You might want to write a program to generate the input to STP.

Your homework answer should include: (1) the main idea of your approach, (2) a description of the boolean variables you used, (3) the solution to the above Sudoku puzzle (generated using your program), and (4) a print-out or listing of the program you wrote to generate the input to STP.