Graduate Artificial Intelligence 15-780

Homework 2: *Tree Search and SAT solving*

Out on February 1
Due on February 15

Problem 1: Proving with InstGen

Consider the following formula:

$$\mathscr{F} = (\forall x, P(x) \vee Q(x)) \wedge (\neg P(a) \vee \neg P(b)) \tag{1}$$

Prove that $\mathscr{F} \Rightarrow \exists x, Q(x)$ using the InstGen algorithm given in class (pp. 38-47 in the slides for January 18th). For each iteration $\{0, \ldots, n\}$, please provide:

- The first-order formula $\mathscr{G}_i$;
- The propositionalized formula $\mathscr{G}_i^{(Gr)}$;
- A propositional model $M^{(Gr)}$ for $\mathscr{G}_i^{(Gr)}$;
- The lifted model M for $\mathscr{G}_i$;
- The set of discordant pairs;
- An application of InstGen on one of those discordant pairs that leads to $\mathscr{G}_{i+1}$

Discuss how you formed $\mathscr{G}_0$, and why you are using it to prove $\mathscr{F} \Rightarrow \exists x, Q(x)$. For the final iteration n report which of the algorithm's termination conditions you hit.

We found a proof that took 4 applications of InstGen, but your proof may be longer. The propositional formulas $\mathscr{G}_i^{(Gr)}$ should be simple enough to find models by hand.

Problem 2: Build-A-SAT-Solver Workshop

In this problem you need to build a boolean satisfiability solver that takes a set of variables and constraints in conjunctive normal form (CNF) and returns either a satisfying assignment or determines that no satisfying assignment is possible.

In particular, we are asking you to implement the DPLL algorithm—it is the basis for some of the world's fastest SAT solvers. Even though SAT is an NP-complete problem DPLL typically has good *empirical* performance.

The input will be provided in a simplified DIMACS CNF format (see, for example, `http://www.satcompetition.org/2009/format-benchmarks2009.html`):

- Each line that starts with a '`c`' is a comment;
- The first non-comment line must be of the form: '`p cnf` ⟨`numOfVariables`⟩ ⟨`numOfClauses`⟩;
- All other lines are space delimited lists of literals with a positive value indicating the variable and a negative value indicating its negation. For example, the number '`-4`' represents $\neg x_4$. Each line is terminated by a '0'. (Note: because of this don't call any variable x_0. Not only is this symbol reserved, but also $0 = -0$.)
  For example, the formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$ can be represented as:

```
c Some made-up problem
p cnf 3 2
1 -2 0
-1 2 -3 0
```

The solution should be a space delimited two column file with a row per variable. Each row should have the format va, where v is the variable number and a is either 0 or 1 (false or true). For example, a solution file for the above problem might look like:

```
1 0
2 0
3 0
```

In this solution, each variable was set to false.

Some test SAT problems can be found on the homework page. You can use the provided Python model verifier to check your solutions. Feel free to compare against minisat (`http://minisat.se/`) or any other existing solver. Just make sure that you hand in your own code.

In your implementation of DPLL, consider including the following features:

- Unit propagation;
- Variable ordering heuristics;
- Value ordering heuristics;
- Handling special cases (*e.g.* Horn clauses)

Please do not include clause learning or or conflict-directed backjumping at this stage of your code. The solver that you build here will be extended in the next problem set. Design your SAT solver with this in mind—make sure that your implementation is robust and easily extensible.

We will not specify what language you should use, but you **must** provide two scripts to ease automated grading:

- **build.sh**, a script that builds your project. For C++ this may be a bunch of g++ calls, or a single make call. For a scripting language like Python this might be an empty file or a pleasant `echo` message for the grader.
- **run.sh**, a script that takes in a single DIMACS CNF formatted file as input and prints either a solution or "UNSAT".

In particular, your scripts should hook into a grading script that looks like:

```
${DIR}/build.sh
${DIR}/run.sh file1.cnf > ${DIR}/run_1.out
${DIR}/run.sh file2.cnf > ${DIR}/run_2.out
...
```

How you will be graded:

You must provide the following files:

- The commented code for your solver;
- Both **build.sh** and **run.sh**.

Do not include any binaries.

If your scripts do not conform to our naming standards, your code does not compile, or your code is a wrapper to an external SAT solver, **the grader will be unhappy and mark accordingly**. If the grader is in a good mood, they might provide limited 'run-time' debugging, but **do not count on it**. Well commented code is always appreciated and is known to put graders in a good mood.

Your code will be first run against the provided test examples, then run against a reserved set of CNF problems. You will get full marks if your solver provides either a satisfying assignment or correctly identifies a formula as UNSAT within **two minutes**. (This is a generous cut-off and is per run.) Bonus marks will be awarded to the top three fastest solvers for each problem instance in the reserved set.

Code hand-in instructions will be sent out soon via email.

Problem 3: Getting Empirical On Your SAT

Write a random 3SAT generator. This generator should pass in two numbers: c and n. Given these numbers the generator should generate c clauses and write the result in the DIMACS format described in Problem 2. Each clause should have three literals selected uniformly from all possible literals of n variables. (recall that there are $2n$ literals if there are n variables).

Generate 25 different random 3SAT instances for $n = 25$ and $c = \{75, 85, \ldots, 175\}$. Generate a further 25 different random 3SAT instances for $n = 50$ and $c = \{150, 170, \ldots, 350\}$. This is a total of 550 problem instances—start these experiments early. Run your SAT solver from Problem 2 on each instance, and record the time that it takes to solve it and whether it was satisfiable or not.

a) Plot the probability of SAT vs. the ratio $\frac{c}{n}$ for both $n = 25$ and $n = 50$ (separately; do not aggregate them). What do you notice? Can you explain this behavior?

b) Plot a box-and-whiskers graph (see, for example, Matlab's `boxplot(X)` command) for the runtime of your solver vs. the ratio $\frac{c}{n}$ for both $n = 25$ and $n = 50$ (separately; do not aggregate them). What do you notice? Can you explain this behavior?

Problem 4: Randomized Solvers

Modify your solver from Problem 2 to branch randomly—branch by selecting a variable uniformly from the set of remaining variables (not branched on; not unit-propagated). Please copy/backup/branch your old solver first. Do *not* hand this version in as your answer for Problem 2 *unless* this is the fastest solver that you have found. You can also use a flag in the command line to change your solver's branching heuristic, but make sure that your `run.sh` file in Problem 2 has the correct flag set.

We will call your solver from Problem 2 S_G and this randomized solver S_R.

a) For `problem10.cnf` in the provided files, run S_R your random solver at least 25 times and plot the cumulative distribution function (CDF) of the runtimes. (More runs yield a smoother plot.) You can, for example, use the `cdfplot(X)` command in Matlab.

b) Suppose that you have run your S_R on a particular problem instance many times. After plotting the CDF (as above) you notice that the CDF function of your solver's runtime looks almost exactly like

$$F(x) = 1 - \frac{1}{x+1}, \tag{2}$$

where $F(x)$ is the probability that the run took less than x seconds to complete. (Your CDF plot from part (a) will not necessarily look like this plot.)

How can you use this empirical observation to speed up your solver? Propose a modification to S_R, and simulate this modified solver's runtime using the idealized CDF given in Eq 2. *I.e.*, write a program to sample from the distribution of runtimes for this modified solver, and plot the resulting CDF.

c) Is S_G better or worse than S_R on `problem10.cnf` in the provided files? Think of at least three ways to compare the performance of two potentially random algorithms. What are the benefits and drawbacks of each method of comparison?