

SAT

Due: 11:59 PM 4/07/08

Introduction

There are many NP-Complete problems floating around in the world. Though they take exponential time in the input to solve in the worst case, there are many pragmatic circumstances in which one must solve NP-Complete problems. It is therefore desirable to develop an algorithm that solves these problems effectively in the general cases. Even more agreeable would be an algorithm that solves many NP-Complete problems effectively in the general cases. For this assignment you will be implementing a number of solvers for the SAT decision problem, a problem that many NP-Complete problems can be reduced to.

SAT

The boolean satisfiability problem (SAT) is the problem of deciding if there is a truth assignment for the symbols that appear in a sentence such that it assigns the value *True* to the sentence in question. The sentence is presented in conjunctive normal form (CNF), i.e., it is a conjunction of disjunctions of sentence symbols or negated sentence symbols. CNF is described by the following BNF.

$$\begin{aligned} \text{Sentence} &\rightarrow (\text{Clause}) \mid \text{Sentence} \wedge (\text{Clause}) \\ \text{Clause} &\rightarrow \text{Literal} \mid \text{Literal} \vee \text{Clause} \\ \text{Literal} &\rightarrow \neg \text{Symbol} \mid \text{Symbol} \\ \text{Symbol} &\rightarrow A \mid B \mid \dots \end{aligned}$$

The sentence

$$(A \vee B \vee C) \wedge (A \vee \neg B \vee \neg C \vee D) \wedge (\neg C) \wedge (\neg A \vee \neg D)$$

is an example of a sentence in CNF. This sentence has a satisfying assignment; if the truth assignment maps *A* to *True*, *B* to *True*, *C* to *False*, and *D* to *False*, then the sentence will be assigned the value *True* since each clause contains at least one literal that is assigned the value *True* by this truth assignment. Note that a sentence may have more than one satisfying assignment or none at all. For the above example, the truth assignment that maps *A* to *True*, *B* to *False*, *C* to *False*, and *D* to *False* is also a satisfying assignment. An example of a sentence that has no satisfying assignment is the sentence

$$(A) \wedge (\neg A).$$

Not all instances of SAT problems are equally difficult. With a clever approach, an under-constrained problem (with very many variables and very few clauses) can be solved trivially, as can an over-constrained problem (many clauses and few variables).

For a review of propositional logic terminology you can also consult section 7.4 in Russell and Norvig (p. 204).

What to Do

To complete this assignment you must implement three SAT solvers. The first must use WalkSAT, a random walk algorithm. The other two must use the DPLL algorithm, with them differing in the heuristic that they use to select the variable to branch on. The `StandardDpllSatSolver` class should branch on the unassigned variable that appears in the greatest number of clauses. The `SpecialDpllSatSolver` should use a heuristic of your own design to choose the branching variable. Feel free consult web sites, the book, Amy's notes, astrology, or whatever you want for inspiration.¹ Make sure that your heuristic outperforms the standard heuristic in at least some cases. Particularly good heuristics will be rewarded with extra credit points.

Writeup

For this project we want you to turn in a writeup that is more like a writeup you might do for a small research paper. **We strongly recommend that you budget your time so that you can run your tests two days before the due date so that you leave yourself time to work on your report.** We want you to do some investigation of how the different solvers you've written perform.

For each of the solvers, we would like you to report on how the average time spent solving a problem varies with respect to number of clauses and number of variables per clause. That means that for each solver, you should test them on, at the very least, four conditions: a high (about 650) clause, high variable (about 150) condition; a high clause, low variable (about 70) condition; a low (about 350) clause, low variable condition; and a low clause, high variable condition. The empirically determined most difficult clause/variable ratio is 4.3. Try to confirm or disconfirm that result in your own writeup. On each condition, try to get at least 30 trials so that you average out some of the random noise in your results.

You should investigate how both the time spent and the number of branches taken vary with respect to problem size. Also, think about the best ways to gather and present your data to demonstrate interesting results. Should you count both solvable and unsolvable instances together, or separately? You should vary your instances on the DPLL conditions. But what about WalkSAT conditions? How will you get meaningful results about an algorithm that is, itself, random?

Feel free to discuss these issues with a TA.

If you are interested in creating graphs, you might look at gnuplot, which is a free plotter that

¹Make sure to cite any sources that you use.

is installed on the CS department computers.² Excel and other programs can also be used to create pretty graphs.

The writeup will constitute a significant portion of your grade for the project. It should have easily readable data and well thought out analysis. Extra credit will be given to particularly outstanding writeups. It is very important to clearly label all your data. If you are taking an average over many runs, you should include how many runs you are averaging, if you are leaving out any outliers when you take the average, etc.

What to Handin

Please hand in the following:

1. All of your code with the three SAT solvers.
2. Your write-up, including any graphs or tables in electronic form.

All written work must be submitted *electronically* as a pdf document, a ps document, or Star Office document. All of your work can be handed in by typing `cs141handin sat` in the shell from the directory with your work. Make sure all of your code and written work are stored in the same directory.

Support Code

Copy the support code from `/course/cs141/src/sat/` to your course directory. The support code comprises a number of classes for modeling a SAT problem instance, an Ant build file, and four stencil class in which you will implement your solvers:

- `DpllSatSolver`:
This is the class in which you will implement the DPLL algorithm. It is an abstract class and contains an abstract method `Variable heuristic(SatInstance si)`. Your implementation of the DPLL algorithm should call this method to decide which Variable in the current `SatInstance` to branch on. You fill fill in the heuristic method in the subclasses of this class.
- `WalkSatSolver`:
This is the class in which you will implement the WalkSAT algorithm.
- `StandardDpllSatSolver`:
This is the class in which you will implement the standard heuristic. Recall that the

²For more information, see <http://t16web.lanl.gov/Kawano/gnuplot/index-e.html>.

standard heuristic should return the unassigned variable that occurs the greatest number of clauses in the SAT instance.

- **SpecialDpllSatSolver:**

This is the class in which you will implement your own heuristic. Your own heuristic should outperform the standard heuristic in at least some cases and should be reasonably efficient.

We've also provided you with some classes to help you create, save, and reuse SAT instances. **SatGen** is a class that will produce random SAT instances that you can then pass to your solver. The **Parser** class allows you to read in SAT instances which you have saved to a file. We've included some files containing SAT instances in the **samples** directory of the support code. The **SatInstance** class which will be described shortly contains a method **printToFile(String filename)** that will print a representation of the the SAT instance to the specified file. This representation can then be read by the **Parser** class.

There are classes in the support code concerned with modeling SAT instances. These classes are complete and should not require any modification, though you are welcome to make small changes if you want to, but do not make any large structural changes. What follows is a description of these classes. For more details, see the javadocs.

- **Variable:**

This class models a variable in a SAT instance. It contains a unique identifier for the variable as well as its truth assignment.

- **Clause:**

This class models a clause in a SAT instance. It maintains sets of positive and negated **Variables** that you can quickly query. It also supports a number of useful operations, such as adding and removing variables, and updating the truth assignment of a variable that belong to the clause. This class also provides a method that determines the clause's truth assignment based on the truth assignment of the variables that belong to it.

- **SatInstance:**

This class models a SAT problem instance. It contains a set of clauses and maintains a mapping from **Variables** to the set of **Clauses** in which the variable occurs.

All of these classes are immutable, i.e., once an instance has been created, it cannot be modified. Whenever a method of the above classes returns a data structure that is mutable (such as a **Set**), mutating that data structure will have no effect on the object which returned it. For example, you can query a **SatInstance** for its clauses using the **clauses()** method. This method returns a **Set** of **Clauses**. Removing a clause from this set will not remove it from the **SatInstance** to which it belongs. Any method that returns a **Set** first creates a shallow copy of it and returns that shallow copy. Also, update operations defined on these classes return a new instance of the object reflecting the desired changes rather than mutating the objects. In short, this support code is written in a functional style.

As usual, you can run `ant projecthelp` to see a brief description of the targets included in the build file. You can also run `ant help` to learn how to select a specific solver and how to pass a SAT instance to your solver when using Ant. For this project we've provided a target called `repeat`, which should help you automate your testing. See `ant help` for the details.