

SAT

Due: 6:00 PM 11/10/09

Introduction

The "Word" offensive proved to be a spectacular success against the Cylons, rendering the toasters inoperative for hours while their processors attempted to solve the complex puzzles you generated. During this time, the Alpha squadron was able to infiltrate and destroy one of the enemy's major production factories. Unfortunately, this operation revealed that the automatons are developing an even deadlier machine that would be impervious to all our attacks. Fortunately, the team did manage to retrieve a prototype of this new drone for study.

Working tirelessly on the problem, your TAs created a virus to infect the new robots. But, due to sleep deprivation they are not sure if their malware will be effective. Before collapsing from exhaustion, they managed to convert the problem to a series of SAT instances. As part of the Earth's best defenders, it falls to you to develop a solver to evaluate these SAT instances and determine whether the virus is ready for deployment.

SAT

In this assignment, you will be creating three satisfiability problem solvers and writing a report on their performance on various SAT problems.

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 consult section 7.4 in Russell and Norvig (p. 204).

What to Do

To complete this assignment you must implement three SAT solvers.

1. WalkSAT: a random walk algorithm. There are many versions of this algorithm and you can choose which to implement, but be sure to discuss your choices in the write up. As a starting point check out the version described in the textbook: section 7.6 in Russell and Norvig (p. 223), or the one mentioned on wikipedia: <http://en.wikipedia.org/wiki/WalkSAT>.
2. Standard DPLL: a DPLL algorithm that branches on the unassigned variable that appears in the most number of clauses.

3. Special DPLL: a DPLL algorithm with a branching heuristic of your choosing. Be sure to explain your choice in the write-up. Feel free consult web sites, the book, Meinolf's notes, astrology, or whatever you want for inspiration.¹ Make sure that your heuristic outperforms the standard heuristic in at least some cases, and include these cases in your handin. Particularly good heuristics will be rewarded with extra credit points.

1 Write-up

For this project we want you to turn in a writeup that is similar to what 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 to 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 clause, high variable condition; a high clause, low variable condition; a low clause, low variable condition; and a low clause, high variable condition. The number of clauses should range between about 650 on the high end and 150 on the low end. The number of variables should range between about 350 on the high end and 70 on the low end.

The empirically determined most difficult clause/variable ratio is 4.3. Try to confirm or disconfirm that result in your own writeup. For 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 is installed on the CS department computers.² Excel, Matlab or 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

¹Make sure to cite any sources that you use.

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

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.

2 What to Hand In

Please hand in the following:

1. All of your code for 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. All of your work can be handed in by typing `cs141-handin 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.

3 Support Code

You can get the source code you need for this project by running `cs141-install sat`, which will copy all of the files in `/course/cs141/src/sat` to your home directory in `~/course/cs141/sat`.

The support code is configured as an Eclipse workspace. Simply import the `sat` directory as an existing Eclipse project to get started.

Javadocs for all of the support code are available at <http://cs.brown.edu/courses/cs141/sat/index.html>.

The support code comprises a number of classes for modeling a SAT problem instance, and four stencil classes 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 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 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 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. 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.