The project is due in hardcopy (in Prof. Khardon’s mailbox in the main office) and via provide by Monday April 29, 12:00 noon.

Note: If you want to do another project of your own choice, please come to talk to me. I will accept any project that exercises and tests some of the material we covered, as long as it is not too easy or too hard. BUT you must negotiate such a project in advance.

1 Introduction

In this project we will study algorithms for Satisfiability, the canonical NP-Complete problem. The input is a formula in conjunctive normal form over a set of $n$ Boolean variables. The question is whether the formula can be satisfied by some assignment of values to the variables. For example, the input formula

$$(x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor \overline{x_3}) \land (x_1 \lor \overline{x_3})$$

where $x_1, x_2, x_3$ are the Boolean variables, $\overline{x_i}$ stands for the negation of $x_i$, “$\lor$” stands for logical Or, and “$\land$” stands for logical And, is satisfiable by the assignment $x_1x_2x_3 = 000$.

Although the problem is NP-Hard it is known that “random” instances are easy to solve and various heuristics exist for the problem. A lot of research has gone into understanding and characterizing when and why instances are easy or hard; we will not explore these aspects in the project. Instead we will implement two simple algorithms for solving it and evaluate their performance on some instances.

Problems to test on: Input problems for your programs are available at /comp/160/files/project/.\footnote{Note that this is a path on the CS system and not a web accessible address.}

The formulas are encoded in a simple text annotation where the above example is represented as:

```
x1 nx2 x3
nx1 x2 nx3
x1 nx3
```

Each formula is given in a separate file.

2 Algorithm 1: Davis Putnam

This is a deterministic algorithm that always returns a correct answer (but may run for a long time in some cases).

The algorithm heuristically picks a variable and assigns a value for it (0 or 1). In our example we might assign $x_2 = 0$. This value is “plugged in” to the formula to produce a smaller one. Three effects are possible:
• Some clauses are already satisfied by this assignment; they can be removed from the formula when exploring it further. For example \((x_1 \lor \overline{x_2} \lor x_3)\) is satisfied when \(x_2 = 0\) regardless of the values of \(x_1, x_3\).

• Some clauses simply get smaller. For example \((\overline{x_1} \lor x_2 \lor \overline{x_3})\) becomes \((\overline{x_1} \lor \overline{x_3})\).

• Some clauses, that do not include the chosen variable, are not affected. This is the case for \((x_1 \lor \overline{x_3})\).

In our example, after assigning \(x_2 = 0\) we get the smaller formula \((\overline{x_1} \lor \overline{x_3}) \land (x_1 \lor x_3)\). After making this assignment the algorithm recursively tries to satisfy the smaller formula. If this succeeds the solution can be returned. If this fails, the algorithm uses the other possible value \((x_2 = 1\) in our example) and tries to satisfy the resulting formula. Again if this succeeds the solution can be returned. If the second attempt fails then the formula is not satisfiable.

Notice that there is an important difference between removing a clause because it is already satisfied (the first case), and the case where a clause has no more literals when it gets smaller (in the second case). An empty clause (all its variables have already been removed and none was satisfied) cannot be satisfied. Therefore in this case the algorithm can return failure immediately. In other words, in this case there is no need for the recursive call. Taking this observation a step further, one should avoid using an assignment leading to such a situation because it simply wastes computation time.

To summarize, the algorithm makes a heuristic choice of variable to assign \((x_2\) in our example) and the value to assign to it first \((0\) in our example). The algorithm makes one or two recursive calls in order to find a satisfying assignment. If one call succeeds the algorithm returns the solution. If both fail the algorithm reports that the formula is not satisfiable.

3 Algorithm 2: WalkSat

This is a randomized algorithm that searches for a satisfying assignment of the input expression. If it finds one, it can safely output Yes and the assignment. This is obviously correct. If it does not find a satisfying assignment within a bound on its run time it stops and says No. This may be incorrect.

The algorithm belong to a family of algorithms using randomized local search. It works as follows. The algorithm starts by drawing a random assignment to all the variables; let’s call this assignment \(A\). It then works iteratively, each time “improving” \(A\) or stopping when \(A\) satisfies the expression or when its upper bound on the number of iterations is exhausted.

To improve \(A\) we pick a random clause among those not satisfied by \(A\). We then pick a variable in the clause and flip its assignment. In our example, consider starting with \(A = 001\). Evaluating the expression on \(A\) we see that only the third clause \((x_1 \lor \overline{x_3})\) is not satisfied by \(A\). In general there will be more than one such clause. We then pick a random clause in this set and pick a variable in that clause and flip its value. In the example there is only one clause to choose and it has two variables \(x_1\) and \(x_3\) that we may flip. At this stage there are two possibilities for choosing the variable. The greedy choice picks the variable that maximizes the number of clauses satisfied after the change. In our example if we flip \(x_1\) to get \(A = 101\) the second clause is not satisfied. If we flip \(x_3\) to get \(A = 000\) the expression is satisfied. Therefore the greedy step will choose to flip \(x_3\). Unfortunately the greedy step on its own is not sufficient to yield an effective algorithm.
Therefore, the algorithm includes a random element; the algorithm chooses a random variable with probability \( p = 0.5 \) and the greedy step with probability \( 1 - p = 0.5 \).

To summarize, the algorithm initializes \( A \) and repeats changing \( A \) until a satisfying assignment is found or \( I \) iterations (use \( I = 10000 \) by default) have passed. In each iteration, it picks a random unsatisfied clause from the expression and then with \( p = 0.5 \) flips a random variable in that clause and with probability \( 1 - p = 0.5 \) flips the variable selected by the greedy choice. If you wish you may change the heuristics proposed here for the greedy choice and probabilistic choice but should keep the general structure of the algorithm intact.

4 Your Program

You program should include code to read a formula from file, and the choice of algorithm (1 or 2). The code should then run the selected algorithm and return a Yes/No answer plus the satisfying assignment in case one is found.

Your design choices include the appropriate representation for formulas in memory, and any supporting data structures. In addition you need to pick the heuristic for choosing the variable and value to assign for the Davis Putnam procedure, and any details you may want to add to the WalkSat algorithm.

It is advisable to first spend some time choosing a heuristic, then analyze what data representation will be best for calculating it, before embarking on an implementation. In fact, as part of the project submission you need to analyze the complexity of one iteration of each algorithm and explain why your choices are useful to minimize this cost and/or the total cost of the algorithm.

You should implement your program in C++ or Java and avoid using any libraries other than the basic types. As part of the project you have to implement all data structures and algorithms that are required. In particular, libraries for vectors and their sorting algorithms, hashes, etc should be avoided and if you need to use them then you should implement your own versions of these.

In addition, please make sure that your program runs without problems on the department’s servers.

5 Experimenting

After developing and debugging your program you should run some timing experiments to see how the algorithm scales to larger problems. You should run your program on all the test files provided in the directory listed above and report run times and results. As a first step you should compare the performance of Davis-Putnam with time bound of 5 minutes to WalkSat using \( I = 10000 \). You should then follow with a more thorough comparison and report run time, number of iterations (for WalkSat), and correctness/completion results for the two algorithms.

[extra work] Additional exploration of the algorithms, and of hard instances for satisfiability (is not required and) will received extra credit.
6 Report

Write a short report that (1) explains the heuristics you use and why you chose them, considers the complexity of implementing one iteration using the heuristics, and the choice of data structures for this purpose, (2) presents the results from the experiments, and (3) draws any conclusions you can from these.

7 What to Submit

You should submit

- A printout of: all your code for this project, and the report. Please make sure that your code is well documented, and that as part of the documentation you explain how to run the code on the CS system.

- Submit the same (code and report) through the provide system. Please put all the files from the previous item into a zip or tar archive, for example call it project.zip. Then submit using provide compl60 proj project.zip.

8 Criteria for grading projects

The projects will be graded based on the (1) amount of work done, (2) quality of the code, (3) documentation of the code, (4) explanation of choices of algorithms and data structures used, and (5) quality of the report.