DPLL & Walksat Implemented, Compared

Daniel Jones

December 11, 2008

Implementation

For this assignment, the Walksat and DPLL algorithms were implemented in the OCaml language, using a mostly purely functional style (with the exception of pseudo-random number generation needed by Walksat, which must store its state).

The representation of CNF formulas was a very natural one for a functional, list based language. Each literal is represented as a string, boolean tuple. The string provides a name, the boolean denotes the polarity (x as opposed to $\neg x$). Very simply then, a clause is just a list of literals, and a formula is a list of clauses. The OCaml type signature for this is:

(bool * string) list list

A model is represented very similarly. The same way as a clause in fact.

(bool * string) list

The important (implicit) difference, is that the boolean here represents an assignment to the variable rather than its polarity in the clause.

There are no other especially notable choices made. I followed typical functional style, abstracting to down to small functions, using list operations and recursion in place of loops. So, I will spare the reader the fine details of each algorithms implementation. However, I should point out that after proper abstraction, they could be written to be not much longer than the pseudocode in the Russel and Norvig book. The DPLL and Walksat algorithms are both only a handful of lines. The WalkSat algorithm looked just a bit different as it was done recursively. With both algorithms and all the apparatus to test them, there are well under 300 lines of code.

Unfortunately, representing formulas and models as lists and implementing each algorithm in a purely functional manner presented some downsides. A notable example is finding pure clauses in the DPLL algorithm. This heuristic is meant assigns values to variables that always have the same polarity and can not ever force any clause to be false. The expectation is that making these obvious choices will speed up the algorithm. Unfortunately, because of the

Variables	DPPL Time	DPPL Satisfied	WalkSat Time	WalkSat Satisfied
5	0.180	0	2.360	0
6	0.238	2	2.486	2
7	0.450	1	2.699	1
8	0.546	3	2.740	3
9	0.890	4	3.184	1
10	0.967	10	2.938	6
11	1.577	9	3.136	5
12	1.604	15	3.757	5
13	2.711	13	3.452	6
14	2.383	18	3.662	6
15	2.734	19	3.656	8
16	3.359	20	3.768	8
17	4.942	20	4.595	4
18	1.902	20	4.512	7
19	5.835	20	5.086	6
20	6.046	20	5.248	4

Table 1: Each trial used 20 randomly generated formulas of 50 clauses, each with 3 terms.

representation used, finding pure clauses takes so much computation that its costs outweigh its benefits in most of my tests. This could undoubtedly be improved by building a more complicated solution.

1 Results*

In the end it was a worthwhile endeavor to implement these algorithms in a functional language, but they are done so in such an unoptimized manner, that I hesitate to draw any strong conclusions about the algorithms themselves. Though, these results are interesting in a way. The implementations of DPLL and Walksat ended up being of very comparable complexity (in fact, neither implementation is very complex at all). The benchmarks do inform us about which algorithm is superior if one were to choose one to implement in as simple a manner as possible. It is a test of what performance can be achieved, while optimizing for conciseness and simplicity.

Tests were performed using rather naive generation of random CNF formulas. The program is given a number for clauses, clause length, and variables. Then variables and polarities are chosen at random for each clause being generated. Because of the fundamental inefficiency of this implementation, tests were not at as large a scale as would be desirable, but we can still observe some interesting results.

The immediate difference to note is the run time DPLL tends to be very dependent on the number of variables. Increasing the number of variables ex-

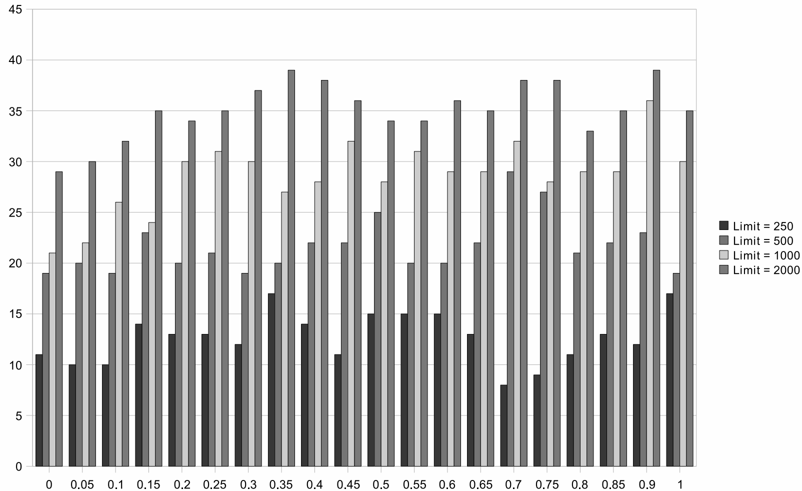

Figure 1: The horizontal axis is the value for p, the vertical axis is the number of clauses that were discovered to be satisfiable. Each bar represents repeating the test with a different flip limit.

ponentially increases the size of the search space, and since DPLL must exhaustively check an unsatisfiable formula, this result is not surprising. Walksat on the other hand, simply gives up after the upper limit is reached, providing a convenient upper bound. Though Walksat's performance is very much dependent on choosing an upper bound that is large enough but not too large. This can only be found through experimentation it seems.

The Walksat has two extra parameters and it would be useful to explore how these affect the outcome the the tests. The p value gives the probability that a to flip symbol is chosen randomly from a clause, rather than choosing the symbol that will satisfy the most clauses.

A large number of trials were performed with formulas consisting of 50 clauses of 3 terms each, and 10 variables. Also tried were several values for the maximum flip limit. The outcome is mostly inconclusive. It would have to be run at a finer granularity with many more trials to determine and actual optimal. It is at least interesting to see that no very clear optimal value for p emerges.

There results here are intriguing, though by no means comprehensive. It is difficult to determine how these algorithms behave in general. Small changes seem to vastly change the results in some cases. If one were to choose an algorithm to for a specific problem, the only reasonable way to do so would be to

take consider the specific problem. Is Walksat's lack of soundness a problem? Is the time spend by DPLL completely exploring the search space for and unsatisfiable formula a liability? Each algorithm has its strengths and weaknesses and are both quite useful.