CS3105 Practical 1: WalkSAT

Ian Gent

February 23, 2016

This practical carries 50% of the coursework assessment for the module and is due at 21.00 on Tuesday 8 March, 2016. This practical concerns the local search algorithm WalkSAT.

1 WalkSAT

WalkSAT is a lovely search algorithm for the SAT problem, discussed in Lecture 7 (Search Lecture 4.)[1]

Here is the pseudocode for the algorithm presented in the lecture (with some minor changes for clarity in the later discussion). Remember that the input is a set of clauses, and we have preselected a probability p. I have adapted the code slightly to explicitly have the functions `pickvar` in line 6 and `flip` in line 9.

```
 1. Choose a random complete truth assignment T
 2. while (T leaves at least one unsatisfied clause) {
 3.     choose an unsatisfied clause C at random
 4.     generate a random number r between 0 and 1
 5.     if (r > p) {
 6.         set v = pickvar(C) // select variable v in C to flip which
            }                  // maximises number of satisfied clauses
 7.     else {
 8.         set v = randomly chosen variable in C
            }     // end if
 9.     set T = flip(v)        // T with v set to the opposite value
10. }        // end while
11. Return T
```

2 SAT and CNF Reminder

SAT is the problem of Propositional Satisfiability. This was covered fairly extensively in CS2002, if you took that. However, if you didnt do CS2002 you dont need to go over it in detail, as this practical should be self-contained.

You only have to consider SAT formulas in CNF: Conjunctive Normal Form. In CNF a SAT formula is ONLY allowed to be: A conjunction (AND) of disjunctions (OR) of literals (variables or negated variables). We often call a formula in CNF a set of 'clauses'. The clauses in the set are implicitly ANDed together. Each clause is a disjunction. For this practical you do NOT need to convert SAT formulas to CNF, but just deal with clause sets in CNF already.

There is a standard file format for CNF instances, called the Dimacs CNF format. This is an extremely simple format so writing a parser for it in your chosen language should present little obstacle. Here is an example:

[1] As you may remember, one of the original paper authors, Bram Cohen, went on to become very famous for writing BitTorrent.

```
c Example CNF in Dimacs format
c A cnf encoding of the letter SAT problem used in Search lectures
c Comment lines before declaration start with c
c Declaration starts with p, gives format (cnf) and num vars/clauses
c Variables are 1 ... num vars,  Negated variables are negative integers.
c Clauses are lists of positive or negated variables terminated by a 0
c
p cnf 3 7
1 2 3 0
1 2 -3 0
1 -2 3 0
-2 -3 1 0
-1 2 3 0
3 -2 -1 0
-1 -2 -3 0
```

A file describing Dimacs format is available at the following url. Note that you only need to be able to parse input formulas in the CNF format in Section 2.1 of this document.

`http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf`

3 Implementing WalkSAT

There are two closely linked issues in efficient implementation of WalkSAT. Those are to implement `pickvar` and `flip`. We want each to be as efficient as possible, while not making the other one too inefficient. As a bad example of what to do, we could store the truth assignment `T` as a bitarray, and then `flip` would be extremely simple: simply change one bit in the bitarray. But this would make `pickvar` very inefficient. To find out which variable to choose we would have to go through the entire clause set, counting occurrences of each variable. But this would be very bad, because many clauses in the clause set would be unaffected by the change, so much time would be wasted. Just as bad in the other direction would be something which made `pickvar` really easy, but which required data structures that made `flip` very expensive to implement.

So there has been a lot of research on efficient implementation techniques for WalkSAT. The following is one which gives a reasonable compromise and has been used by efficient WalkSAT implementations. To optimise the implementation of WalkSAT we need some additional data structures.

- `clausesContainingLiteral`: for each literal, gives the list[2] of clauses that contains that literal. Note this is per *literal*, not per variable. So for each variable `x` store a separate list for `x` and ¬`x`.
- `numSatisfiedLitsPerClause`: for each clause, records the integer of how many literals are currently true in the clause. If this is 0 then the clause is not satisfied, for example. If this is 2 then the clause is satisfied by 2 different literals with the current assignment of `T`.
- `falseClauses`: a list of clauses in the problem which are not satisfied by the current truth assignment.

You have to initialise everything before you start. Notice that `clausesContainingLiteral` is static after initialisation. With these data structures in place, all of randomly selecting an unsatisfied clause, `pickvar` and `flip` become reasonably efficient.

- Because we have `falseClauses` we can simply select a random element of this list at line 3.
- Consider `flip` next, at line 9. The easy bit is flipping `v` from true to false or false to true, but then we have to update the data structures appropriately. If `v` has just become true then all clauses containing ¬`v` have one fewer satisfying literal and clauses containing `v` have one more. Similarly if `v` has just become false then all clauses containing ¬`v` have one more satisfying literal and clauses containing `v` have one fewer.

  We now have to update `numSatisfiedLitsPerClause` and `falseClauses`. Fortunately we have `clausesContainingLiteral`, so we can easily update all of the clauses containing `v` and ¬`v`. A very

[2]I've said list here but it doesn't necessarily have to be a list, you may have a more efficient way of storing it.

important point is to change `falseClauses` appropriately. Whenever a clause goes from having 1 satisfied literal to 0, we have to add it to `falseClauses`. If a clause goes from having 0 satisfied literals to 1, then we have to remove it from `falseClauses`.

- Finally consider `pickvar`. For this one, we have to look at each variable in the chosen clause `C`. For a literal `x` in `C`, the change it will make to the total number of satisfied clauses can be expressed as `make(x) - break(x)`, where `make(x)` is the number of clauses that will become satisfied and `break(x` that will become unsatisfied by flipping `x`. To calculate `make(x)` we count the number of clauses in `clausesContainingLiteral(¬ x)` which have the value 0 for `numSatisfiedLitsPerClause`. To calculate `break(x)` we count the number of clauses in `clausesContainingLiteral(x)` which have `numSatisfiedLitsPerClause`=1.

4 Input Format

Your program should accept command line input of a file containing a SAT instance in Dimacs CNF format.

You should also supply command line arguments in an appropriate way to change the maximum cpu time allowed and probability value p, and (if appropriate for your program) any other parameters.

Your program should output a line containing either the word SATISFIABLE or the word UNKNOWN (since it can't prove unsatisfiability). It should also print out the solution which solves the problem if Satisfiable. Finally it should print out statistics such as run time, number of flips made, and other statistics you think might be interesting.

A set of sample CNF instances guaranteed to be satisfiable will be supplied in studres.

5 Evaluation

As well as description of your program plus report on testing etc, your report should contain an evaluation of its performance. This should be against the sample CNFs provided and if you wish others ones either downloaded from the web or that you create yourself. In reporting how you answered questions like these, feel free to include graphs or other means of presentation that make your points most effectively. Your report should answer questions such as the following, and other questions you might find interesting:

- How many flips per second can your program do on problems on different sizes?
- What are the largest problems your program can solve in a reasonable time?
- What are the smallest satisfiable problems that your program can't solve in a reasonable time?

6 Assessment and Extensions

There is no fixed weighting between the different parts of the practical. Your submission will be marked as a whole according to the standard mark descriptors published in the Student handbook at

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html`

The three main areas in which extension work is possible are

- Find alternatives to this version of WalkSAT, implement one or more, and evaluate them against the version in the practical.
- Optimise your code, either using your own methods or methods from the literature, and evaluate against the core implementation.
- Extend the evaluation significantly beyond the questions asked above. For example: what is a good value of p to use? Does the value of p make an important difference? Does the best value change between instances?

Of course what you do is up to you, but I would rather see one of these extensions done well than weak attempts at more than one of them. As well as your own ideas, you can find pointers in all these areas in the literature references provided below. Also I should mention that if you get an idea from a reference below, but find e.g. it is not as good after all, report that! Negative results are important and if your implementation and evaluation is good you will get full credit.

7 Selected References

The following are references about WalkSAT from the academic literature. It should not be necessary to look at any of these to obtain a good mark, but they may provide additional material to help you overcome any difficulties, as well as providing things to look at for extension work. Also of course you can find many other papers yourself, as well as other tutorial/lecture materials on the topic.

- Local Search Strategies for Satisfiability Testing. Selman, Kautz, and Cohen, 1993. This is the original paper introducing WalkSAT.
  `http://www.cs.cornell.edu/selman/papers/pdf/dimacs.pdf`
- On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT. Hoos, 1999. This discusses some variants of WalkSAT if you want some ideas for alternative forms of WalkSAT.
  `https://www.aaai.org/Papers/AAAI/1999/AAAI99-094.pdf`
- An Adaptive Noise Mechanism for WalkSAT. Hoos, 2000. This discusses an "Adaptive Novelty" mechanism.
  `https://www.aaai.org/Papers/AAAI/2002/AAAI02-098.pdf`
- Efficient Implementations of SAT Local Search. Fukunaga, 2004. Some optimisations to the implementation you can try.
  `http://www.satisfiability.org/SAT04/programme/106.pdf`
- Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses. Balint , Biere, Fröhlich , and Schöning, 2014. More on implementation optimisations, including a cute XOR trick.
  `http://fmv.jku.at/papers/BalintBiereFrohlichSchoning-SAT14.pdf`

8 Language and Libraries

You may use any language to program in of your choice, provided it is supported on lab machines (without further installation). Provide a README file with appropriate build instructions which will work on lab machines , preferably on an ssh terminal (remembering I might not be familiar with the language you use). Provide a Makefile/similar if appropriate.

Again you may use appropriate libraries as you wish, *provided* that they do not implement core functionality required by this practical. So e.g. if your language doesn't have lists as a builtin data structure, you could certainly use a list processing library, but not a library which provides implementations of search algorithms.

If in doubt on either issue as to whether your choices are acceptable, please consult the lecturer before doing serious work on your practical.

9 Submission and Notes

Submit a zip file containing both your report as a pdf and your code. Your code directory should include a makefile which will compile your program to produce the executable.

You should submit your work on MMS by the deadline. The standard lateness penalties apply to coursework submitted late, as indicated at

`https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#lateness-penalties`

I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at:

`https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#Good_Academic_Practice`