SAT Solvers

David Tooley

CS 463

Fall 2014

The three algorithms that I used to solve the boolean formulas were DPLL, Hill Climbing, and Genetic, programmed using Python 3.4.1

To make reading different files easy I have made it so that the program takes all .cnf files that are in the same directory as the python scripts, opens, reads, and then solves them with the name of the file followed by the results.

**Main**

The main function starts by reading all file names that have a .cnf extension that are saved in the same folder as the python files. It loops through each line in the files and ignores every line that begins with c and will not look at any numbers until the line that begins with p is found and number of variables and clauses have been established. After those numbers are found the 2d cnf array is built. The index of the current clause is tracked by a variable as the file reads in and checks each item to be an integer before accepting it into the array. The 2d cnf array has a row for every clause and a column for every variable. The integer read in is saved into the index of the absolute value of the red in value. This assumes that the cnf file is correct and does not have a higher number of variables than stated in the original p line.

After the file has been read in it prints the name of the current file, calls Hill Climbing, Genetic, and DPLL and then restarts with the next file.

**GeneralSatTools**

This is a small file containing functions that are useful to more than one of the algorithms.

testAssignments takes a single assignment and the 2d cnf array and then checks the number of satisfied clauses. An assignment is an array that is the length of the number of variables with each index containing a 1 (true) or -1 (false).

GenerateRandomAssignment takes the number of variables in the cnf and creates an array of that length with either -1 for false or 1 for true. All assignments are in this form.

DeepCopy stems from the need to deep copy the array. My original implementation did not work correctly and I found that there was an outside library that could already do what I needed but I did not want to search through and change all items in all files.

**DPLL**

To begin the DPLL algorithm Solve must be called with the cnf in the expected format. The Solve function calls DpllSatisfiable where it creates the available symbols array which an array containing a list of all symbols (index values) that are not in the model. Then the DPLL function is called, given the 2d cnf array, symbols array, an empty model array, an empty satisfied clauses array and the Tracker. First it creates a deep copy of all given arrays so that they may be manipulated without effecting the previous calls to the function. Next Unit Propagate is called. This function is given the cnf, available symbols array, the array representing the model, and the list of satisfied clauses to be ignored. First the satisfied clauses list is updated, then it loops through every clause in the cnf that is not in the satisfied clauses array. It loops through every available symbol and counts how many there are. If there is only one then it must be set to make that clause true and then recursively call UnitPropagate to update the clause search and the solved clauses array. After that loop finishes it returns to DPLL where it calls FindPureSymbol next. This begins by updating the satisfied clauses array and then loops through every available symbol. It then checks to find the last looked location, which is the first clause not inside of the satisfied clauses array. Then it loops through every clause and checks the next clause against the previous to see if they are the same. If one is found to not be the same it stops and continues with the next symbol. It the loop completes, then the symbol was the same in all unsatisfied clauses and can then be set to either true or false to make as many clauses true as possible. When one is found FindPureSymbol must be called recursively to restart the search with an updated satisfied clauses array and a reduced number of available symbols. Once this returns to DPLL Check is called. This function is given the cnf, the model, and Tracker. This function creates an assignment based on the model, with all items not listed in the model assumed to be false, and then compares the number of satisfied clauses inside the Tracker and returns the number of satisfied clauses. If the number of satisfied clauses returned matches the number of clauses we stop because a solution has been found. Otherwise we check to see if there are any symbols remaining, if there are none we back track. Last we check for timeout, which had been set to 70 seconds. If none of those conditions are met it makes a random choice for the next symbol and sets it to true. It the calls DPLL to see if that made a viable solution, if not it guesses false and returns that result.

**Genetic**

To start solving the cnf formula with the genetic algorithm the solve function is passed the cnf to solve, P the number of initial assignments in the population, R the number of assignments to reproduce, m an integer from 0 to 100 which determines the percentage to mutate in every iteration, and s the number of iterations to loop through. For my purposes I chose to have a smaller initial population (150), and reproduce a lot of assignments each time (800), mutate 10% of the population every iteration, and iterate through 10 times. The function starts be creating the initial population number of assignments and finding their fitness values. The highest fitness value is remembered and checked against the number of clauses to see if it satisfies the cnf. Then NaturalSelection is called, given the population of assignments, the corresponding fitness values, and the number of clauses in the cnf. A random number of the population between 10 and 30% are destroyed weighted by fitness. The new population of assignments is returned. Next the remaining assignments reproduce with R offspring. Two randomly chosen assignments are taken and randomly mixed together in a new assignment. If there is only one assignment remaining it is allow to reproduce by itself but otherwise there is a check to ensure the assignments are not the same. The new population of assignments is returned. Lastly, m% of the population is mutated. First the index of the assignments to be mutated are collected in an array, then between 10-30% of the genes are given the chance to mutate in the selected assignments. This function returns the new population of assignments. After all iterations have completed or an acceptable solution is found the number of clauses satisfied is returned.

**Hill Climbing**

The solve function is the only item called outside of the Hill Climbing file and is passed the clauses in cnf and the number of random restarts, s. GetRandomAssignment gives the number of variables in the clause and returns an array of the same length with either 1 for true or -1 for false. This along with the cnf is passed to the Hill Climbing function. This function first tests the assignment to get the number of clauses satisfied and then look at the best possible next variable to change. If the best change does not improve the assignment then the iteration of Hill Climbing has completed and it returns the highest number of clauses satisfied, if the assignment is better Hill Climbing is called recursively with the better assignment. The GetBestAssignment function takes the cnf and the current assignment. It loops through every value in the assignment, changes it and looks at the new number of clauses satisfied. If that number is better than the current best the assignment is remembered. Then the assignment is changed back to how it was originally passed and continues through the loop. After the loop completes the best next assignment is passed back and has a maximum of one value changed.

**Data**

10 Variable Times is comparing the time in seconds taken by each algorithm to solve the cnf equations with 10 variables. Hill Climbing and DPLL were both very quick to find an answer while the Genetic algorithm lagged behind on multiple occasions. This was most likely caused by having to go through more iterations to find a correct solution while Hill Climbing found an answer earlier.

10 Variable Solutions shows that for all of the 10 variable cnf equations the Hill Climbing, Genetic and DPLL algorithms found the best possible solution.

100 Variable Solutions shows that the Genetic algorithm consistently underperformed given the same iterations as Hill Climbing. The average number of clauses satisfied by Genetic was 405.36, for Hill Climbing was 427.68 and DPLL had the best average with 428.24.

100 Variable Times shows the comparison of times for each cnf equation. The Genetic and Hill Climbing algorithms were allowed ten iterations to find the highest possible clause while DPLL had a timeout set at 70 seconds. All but one time DPLL timed out but on the time it found a solution (in 100.440.223352132.cnf) it found it faster than the other two algorithms took to run and not find a solution. Hill Climbing took an average of 55.98 seconds, Genetic took an average of 48.68 seconds and DPLL took on average 70.18 seconds to complete.

During this assignment I was able to experiment with the different ways to solve CNF equations and see for myself how fast solution tree can grow and the importance of reducing repeated calculations as much as possible. I believe that DPLL delivers the best results but was also by far the most difficult to implement. In roughly the same amount of time it on average found a better solution while also solving one 100 variable equation in less than a minute.

**Sources and Collaborators**

Class notes

Discussed with Mark Thurman on how to best implement DPLL and reading in the file inputs.