CS4244 Project 1

Collaborators: Seah Wei Quan (A0173596N), Melvin Vito (A0173284B)

Program Description

How to run

We are using python3.7 for this project.

To run the program: `python main.py`

Program File Structure

Root

- **/CDCL**
  - **/formula_helper**
    - parser.py (parse the input dimacs formula)
    - solution_checker.py (if satisfiable, check if the solution is valid)
    - unsat_prover.py (for stage 2, write out a resolution proof)
  - cdcl_solver.py (contains the entire program logic)
  - main.py (to be run – enter user specifications here)
- **/CNF**
  - (contains various cnf files for testing)
- **/OUTPUT**
  - (contains output files including: solution, statistics, proof)

User specifications

In the main.py the following user specifications can be toggled:

- input_file: change to specify the input file
- heuristic: (default: vsids)
- generate_proof: (default: false)
- eliminate_pure_literal: (default: false)
- random_restart: (default false)

Remarks:
The default settings are the best performance.
Using random_restart and generate_proof simultaneously is not supported as of now.

Heuristics Implementation

Random-choice heuristic (random)

The basic selection heuristic. Select randomly an unassigned literal during each iteration.

Ordered Selection (order)

The literals are selected based on the in order, from the first unassigned literal to the last based on the order the variables are in the input file.

2-clause heuristic (twoclause)

2-clause are clauses with only two literals. These literals are stored in two_clause_dict → format: { lit : count } where count is the number of 2-clauses that contains the lit.

The two_clause_dict will be checked and updated whenever a new_clause is learnt.

During selection, the two_clause_dict is sorted and the unassigned lit with the largest count is selected.

Idea: The idea is that the literals in a 2-clause can be checked if they are assigned correctly quicker through unit propagation and it can also reduce the size of the new clause learnt during conflict analysis.

Vsids heuristic (vsids)

The vsids is based on [M. Moskewicz, 2001]. We implemented the vsids with a twist: instead of using multiplicative decay, we used a multiplicative 'boost'. Since we use multiplicative boost, we do not have to iterate through all the literals to decay their activity score O(num_literals), and only need to boost that one literal O(1). The boost factor is chosen randomly.

Each (signed) literal has an activity score stored in vsids_activity. The activity of a score is boosted under 2 conditions: (1) The literal is being selected in during the selection phase. (2) The literal is part of the new_clause that is learnt from conflict analysis.

During selection, the vsids_activity is max heapify O(num_literals) and the unassigned lit with the highest activity score is selected.

Idea: An advantage VSIDS has over other heuristics is its low computational overhead. The key idea is to collect statistics over learnt clauses to guide the direction of the search, where recent learnt clauses are favoured.

Conflict Analysis

One-uip

We decided to implement the one-uip as cutting at the first encountered Unique Implication Point (UIP) is considered to be the best learning scheme based on experimental testing[1]. A unique UIP is any node in the implication graph, other than the conflict, that is on all paths from the current decision literal to the conflict clause. We made a cut to partition the literals to conflict side and resolution side. This would make the cut to be as relevant to the conflict as possible.

Implementation Details

We have been taught the idea of UIP and how it can be used to reduce the size of the new_clause learnt. From the conflict_clause, we find the last_assigned_literal in the clause that is at the same decision level as the conflict decision level. We retrieve the antecedent clause of this lit to be used as the resolve_clause.

From there on, we resolve the resolve_clause with the conflict_clause and recurse backwards until the one-uip is reached. The final new_clause is the clause after all the resolutions from the conflict_clause to the one-uip.

Remarks: We were amazed when we were introduced to the one-uip scheme in class. We learnt that it could reduce the size of the new_clause, while keeping the new_clause literals as relevant to the conflict as possible. We refer to the following image (and its slides) and it helped us trace and debug to finally produce a correct implementation of the one-uip clause learning.

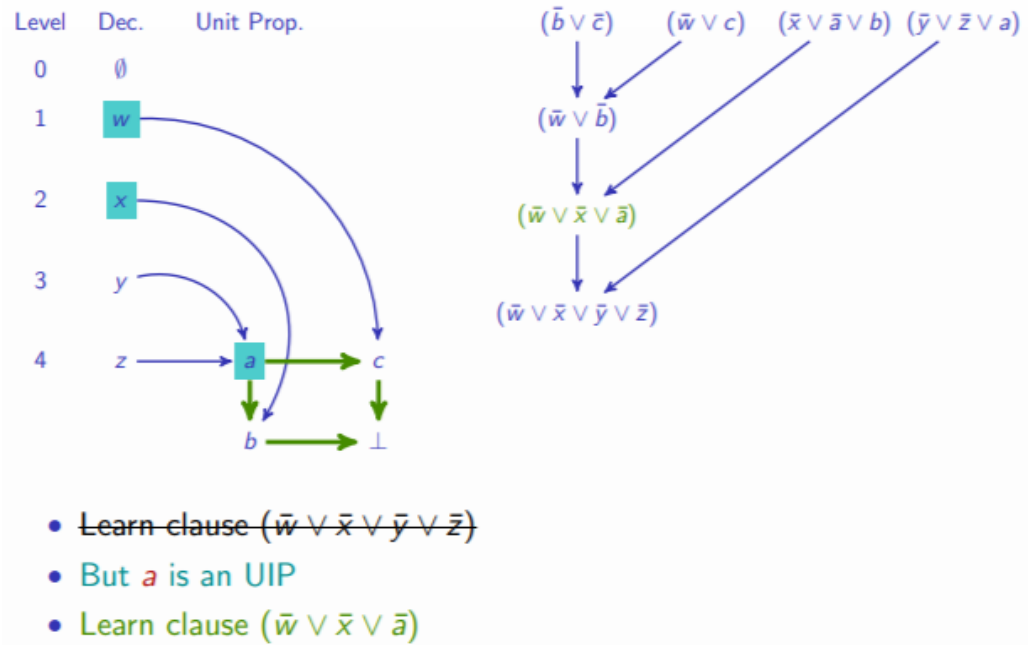

Image taken from: http://satsmt2013.ics.aalto.fi/slides/Marques-Silva.pdf

[1] http://fmv.jku.at/papers/SoerenssonBiere-SAT09.pdf

Performance

The test cases we used are provided here: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

More specifically, we are using these following satisfiable cnf benchmarks:

- uf75-01 (75 variables, satisfiable)
- uf100-01 (100 variables, satisfiable)
- uf150-01 (150 variables, satisfiable)

We ran each input files 10 times and average out the performance for each heuristic.

Average Time Taken (seconds) in 2 d.p

Testcase	**random**	**order**	**twoclause**	**vsids**
uf75	3.57	2.16	0.77	0.20
uf100	38.32	29.99	15.69	3.81
uf150	4213.92	2706.12	175.43	4.12

Average Branching count rounded up to nearest whole number

Testcase	**random**	**order**	**twoclause**	**vsids**
uf75	415	302	126	61
uf100	2266	1463	3093	159
uf150	11738	7342	5414	232

Number of new clauses

Testcase	**random**	**order**	**twoclause**	**vsids**
uf75	22	26	27	18
uf100	256	262	213	207
uf150	356	289	241	148

Performance Summary
As we can see, the VSIDS is the best performing heuristic with the lowest average time taken as well as the lowest branch taken to find the SAT outcome

For SAT cnf:
We can solve 200 clauses in an average of 380 seconds.

For UNSAT cnf:
We can solve 100 clauses in an average of 16 seconds.

CNF encoding of Einstein Puzzle

The encoding of the puzzle can be found at einstein.cnf. There is a total of 5 categories, each having 5 values of their own. Below shows the table for each category and their values:

House Colour	Red (0)	Green (1)	White (2)	Blue (3)	Yellow (4)
Nationality	British (5)	Swedish (6)	Danish (7)	Norwegian (8)	German (9)
Drink	Tea (10)	Coffee (11)	Water (12)	Beer (13)	Milk (14)
Smokes	Prince (15)	Blends (16)	Pall Mall (17)	Bluemasters (18)	Dunhill (19)
Pet	Dog (20)	Cat (21)	Bird (22)	Horse (23)	Fish (24)

In total there are 125 unique variables. We have encoded each variable to be used in the CDCL program to be in the format where:

$$Category\ (x, y) = y * 5 + x$$

x: House number, [1,5], from left to right

y: Corresponding integer value based on the table above.

E.g.: Nation (3, Danish) = 38. This means that if variable 38 is true, the 3rd house owners' nationality would be Danish. In other words, the proposition that the owner of the 3rd house is Danish.

Solution

Using our program and einstein.cnf, our answer to riddle is show below:

```
SATISFIABLE -1 -2 3 -4 -5 -6 -7 -8 9 -10 -11 -12 -13 -14 15 -16 17 -18 -19 -20 21 -22 -23 -24 -25 -26 -27 28 -29 -30 -31 -32 -33 -34 35 -36 37 -38 -39 -40 41 -42 -43 -44 -45 -46 -47 -48 49 -50 -51 52 -53 -54 -55 -56 -57 -58 59 -60 61 -62 -63 -64 -65 -66 -67 -68 -69 70 -71 -72 73 -74 -75 -76 -77 -78 79 -80 -81 82 -83 -84 -85 -86 -87 88 -89 -90 -91 -92 -93 -94 95 96 -97 -98 -99 -100 -101 -102 -103 -104 105 106 -107 -108 -109 -110 -111 -112 113 -114 -115 -116 117 -118 -119 -120 -121 -122 -123 124 -125 0
```

Extracting and inferring with only the positive values from above:

```
3: color(3, red)
9: color(4, green)
15: color(5, white)
```

```
17: color(2, blue)
21: color(1, yellow)
28: nation(3, british)
35: nation(5, swedish)
37: nation(2, danish)
41: nation(1, norwegian)
49: nation(4, german)
52: drink(2, tea)
59: drink(4, coffee)
61: drink(1, water)
70: drink(5, beer)
73: drink(3, milk)
79: cigar(4, prince)
82: cigar(2, blends)
88: cigar(3, pallmall)
95: cigar(5, bluemasters)
96: cigar(1, dunhill)
105: pet(5, dog)
106: pet(1, cat)
113: pet(3, bird)
117: pet(2, horse)
124: pet(4, fish)
```

From clause 124: the pet fish is in the 4^{th} house. The owner of the 4^{th} house is German. Hence, the German owns the fish.

Time taken: 0.0717776 seconds

Branches taken: 5

Improvements from Feedback Submission

Vsids implementation

As mentioned, based on the TA's suggestion, we modified the vsids from multiplicative decay to multiplicative boost. Testing on 150 clauses, this has trimmed down our time taken from 30 seconds to <10 seconds. Upon further research, we understood that vsids activity scores are relative where we only pick the higher score, thus using a boost is sufficient and much more efficient.

Restart and Forget

We also tried to implement the restart and forget technique. The intuition is that for very heavy long-tailed operations, random restart can possibly find a better path by picking a better literal to check on. This technique was found to be successful in modern cdcl but our implementation does not yield a significant improvement. Used in most modern SAT solvers [Biere, 2007].

We implemented this by keeping an activity_score for the new clauses. The activity_score of a clause is computed by finding the average of the vsids score of each literal in the clause. This restart and forget mechanism will be activated when the number of new clauses exceed a threshold. Once activated, the clauses with the lower activity_scores will be deleted. The threshold will be periodically increased by constant factor (1.5) when the solver continues learning new clauses to gradually allow more clauses to be learnt. The idea is to only keep the k most relevant clauses, where k is the threshold that will be increased as the solver goes on. Finally, the solver will backtrack to level 0 with the reduced clause size.

Remarks: Unfortunately, we are unable to achieve much desired improvements with this technique, and it is better to disable it in the user specifications to achieve better results.

Things to explore

Further reduction of new clause size

According to this paper[2], we can further reduce the size of the new clause in one-uip implementation. We can apply sub-suming/local minimisation of the clause literals. Most modern SAT solvers (including miniSAT) are using this technique. We realised through our project that even with one-uip, the new clause can have large size (up to >20 literals) in a 150 literals cnf problem. Hence, we believe this technique will help make the new clause learning be more efficient.

Two-watched Literal

Currently, our unit propagation uses of naive propagation where the whole formula has to be iterated once to check for unit clauses and satisfiability. As unit propagation takes up the bulk of the running time, implementing a new data structure like a two-watched literal scheme will be helpful in improving our runtime.

Expected grade

We would assume to receive a A- grade for this project as we have accomplished all deliverables expected by the project. Our performance analysis is complete and we have gone further to explore other improvement techniques.

Afterthoughts and Lessons learnt

Our discovery and research of the VSIDS showcase our resourcefulness to find a new and better heuristic that can be applied to the CDCL algorithm which reduces overhead and processing cost. We have learned how to implement the CDCL algorithm with the VSIDS heuristic. Using the right heuristic can drastically improve the CDCL algorithm, and how great the performance increase has been compared to the basic DPL algorithm. Also, the one-uip heuristic use for the conflict analysis also helps in the performance of our implementation. Additionally, through reading other research papers, we find many different techniques that researchers have been using to try to improve existing CDCL solvers like miniSAT. We gained a better appreciation of research work in computer science, especially in the field of satisfactory problems.

Our time spent on this CDCL code has inspired us to further improve our code, which is why we have chosen to do Task 3 for Stage 2 of the project, so that we can build on our solver.

Reference:

Marques-Silva, J., Lynce, I, Malik, S. (2009). Conflict-Driven Clause Learning SAT Solvers, Chapter 4.

Hui Liang, J., Ganesh, V., Zulkoski, E., Zaman, A., Czarnecki, K. (2015, September). Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers.

[2] http://fmv.jku.at/papers/SoerenssonBiere-SAT09.pdf