CS4244 Stage 1 Report

Name: Zhang Zhongwei

Student number: A0130503B

Program

Use Python 3 to run the solver.

```
python3 -m pkg.main <file_name> <solver_to_use>
```

where `<file_name>` is the DIMACS CNF file, and `<solver_to_use>` is the branching heuristics. There are 4 heuristics used in this project, which are explained in the next section. To use any of the heuristics, run one of the following:

```
python3 -m pkg.main <file_name> OrderedChoiceSolver
python3 -m pkg.main <file_name> RandomChoiceSolver
python3 -m pkg.main <file_name> FrequentVarsFirstSolver
python3 -m pkg.main <file_name> DynamicLargestIndividualSumSolver
```

If heuristics not specified, `FrequentVarsFirstSolver` is the default.

CDCL Findings

Pick Branching Variable

Heuristics

There are four heuristics implemented currently with my CDCL program, namely:

- *Ordered choice* (OrderedChoiceSolver)

  Choose propositions with ascending order, *i.e.*, 1, 2, 3, ...

- *Random choice* (RandomChoiceSolver)

  Choose propositions randomly

- *3-clause heuristic* (FrequentVarsFirstSolver)

  Modified from *2-clause heuristic*, which we choose propositions with maximum occurrences in 3-clauses, and break ties randomly

- *Dynamic largest individual sum* (DynamicLargestIndividualSumSolver)

  (Explained later)

Dynamic Largest Individual Sum

The heuristics that I chose to perform better than either *random-choice heuristic* (*RAND*) or *3-clause heuristic* (*3CH*), is the *dynamic largest individual sum heuristic* (*DLIS*). This is a popular choice because it falls somewhere in the middle of a spectrum of two extremes [Moskewicz, 2001]. One of the extreme is the *random-choice heuristic*, as it does not consider the distribution of the literals; the other extreme is using complex function of the current variable state and clause database (*e.g. variable state independent decaying sum* or *maximum occurrence of clauses of minimum size*).

DLIS was chosen for several reasons. Firstly, it is simple and intuitive. It actually is a dynamic variation of the *3CH*, where not only is the number of occurrences are considered for choosing the branching variable, also does it consider the sign of the literals. It can be possible that if a literal appears more often as a negative, it is more likely to be negative. Secondly, according to [Marques-Silva, 1999], *DLIS* performs more or less consistently, having less extreme values, in areas relevant to this project (CPU time and number of decisions), than other heuristics. *DLIS* is also simple to implement, so it is easier to extend the base code to other benchmarking heuristics in this project.

DLIS is described as following. For a given variable x, $C_{x,p}$ is the number of unresolved clauses in which x appears positively, and $C_{x,n}$ is the number of unresolved clauses in which x appears negatively. Then we find 2 literals a, and b, such that $C_{a,p}$ is maximum, and $C_{b,n}$ is maximum. If $C_{a,p} > C_{b,n}$, then we choose a to be *True*, otherwise y to be *False*.

Benchmark

The test sets are taken from this page of the University of British Columbia. I used sets from the "Uniform Random-3-SAT" section. The ones bolded are used for all 4 heuristics.

1. **20 variables, 91 clauses, all satisfiable**
2. **50 variables, 218 clauses, all satisfiable**
3. 50 variables, 218 clauses, all unsatisfiable
4. **75 variables, 325 clauses, all satisfiable**
5. **100 variables, 430 clauses, all satisfiable**
6. 125 variables, 538 clauses, all satisfiable
7. 150 variables, 645 clauses, all satisfiable

Average time used

Heuristics\Test sets	1	2	4	5
Ordered	0.070s	1.744s	25.331s	761.093s
Random	0.071s	1.988s	42.995s	2118.475s
3-clause	0.049s	0.569s	3.813s	49.263
DLIS	0.039s	0.433s	2.655s	14.268

Average branches taken

Heuristics\Test sets	1	2	4	5
Ordered	11.5	76.0	354.5	1747.4
Random	11.8	90.5	519.5	3512.7
3-clause	8.4	32.1	94.8	306.7
DLIS	8.7	29.5	80.0	185.6

It is clear from the above observations, *DLIS* performs much better than *ordered-choice heuristic* or *random-choice heuristic*, and slightly better than *3-clause heuristic*, at around 30% reduction in time, and 20% reduction in branches taken.

Testing against 150 variables and 645 clauses tests, *DLIS* needs on average `1394.541s` and `1579.3 branches` to complete. The performance could be much increased if the solver is implemented in a faster language like C, or Java.

Conflict Analysis

A unique implication point (UIP) is any node at the current decision level such that any path from the decision variable to the conflict node must pass through it.

Learnt is generated by finding the UIP of a conflict first. The source code of conflict analysis is given below.

```
def conflict_analyze(self, conf_cls):
    def next_recent_assigned(clause):
        """
        According to the assign history, separate the latest assigned
variable
        with the rest in `clause`
        :param clause: {set of int} the clause to separate
        :return: ({int} variable, [int] other variables in clause)
        """
        for v in reversed(assign_history):
            if v in clause or -v in clause:
                return v, [x for x in clause if abs(x) != abs(v)]

    assign_history = [self.branching_history[self.level]] +
list(self.propagate_history[self.level])

    pool_lits = conf_cls
    done_lits = set()
    curr_level_lits = set()
    prev_level_lits = set()

    while True:
```

```
        for lit in pool_lits:
            if self.nodes[abs(lit)].level == self.level:
                curr_level_lits.add(lit)
            else:
                prev_level_lits.add(lit)

        if len(curr_level_lits) == 1:
            break

        last_assigned, others = next_recent_assigned(curr_level_lits)
        done_lits.add(abs(last_assigned))
        curr_level_lits = set(others)
        pool_clause = self.nodes[abs(last_assigned)].clause
        pool_lits = [
            l for l in pool_clause if abs(l) not in done_lits
        ] if pool_clause is not None else []

    learnt = frozenset([l for l in curr_level_lits.union(prev_level_lits)])
    if prev_level_lits:
        level = max([self.nodes[abs(x)].level for x in prev_level_lits])
    else:
        level = self.level - 1
    return level, learnt
```

From the pool of literals (first from the conflict clause), we separate the literals that are at the current level or at a lower level. If the number of literals that are at the current level is 1, then we can break, as then we have found the UIP. Else, we keep finding the previously assigned literals. The learnt is then computed using the literals at the current level and at lower levels. An illustration of the cut is the following.

Einstein's Puzzle

Variables encoding

Color	Red: 0	Green: 1	White: 2	Blue: 3	Yellow: 4
Nationality	Brit: 5	Swede: 6	Dane: 7	Norwegian: 8	German: 9
Drink	Tea: 10	Coffee: 11	Water: 12	Beer: 13	Milk: 14
Cigarette	Prince: 15	Blends: 16	Pall Mall: 17	Bluemasters: 18	Dunhill: 19
Pet	Dog: 20	Cat: 21	Bird: 22	Horse: 23	Fish: 24

Formula

We use the formula below to generate a series of possibilities of combination between the house number and the properties. Since there are 5 of each categories, a multiplier of 5 is applied. So, a value 98, represents the owner of house 3 smokes Dunhill, since $98 = 3 + 5 * 19$. The mapping of relationships is generated in `reference.txt`.

```
nation(x, y) -> x + 5 * y
color(x, y) -> x + 5 * y
drinks(x, y) -> x + 5 * y
cigars(x, y) -> x + 5 * y
pets(x, y) -> x + 5 * y
```

A pair relationship can be represented using the CNF form by resolution. A $x \wedge y$ relationship is the same as $(\neg x \vee y) \wedge (x \vee \neg y)$. So, the hint "The Brit lives in the red house" can be encoded in CNF as

```
for a in range(1, 6):
    print('-{} {} 0'.format(nation(a, brit), color(a, red)))
    print('{} -{} 0'.format(nation(a, brit), color(a, red)))
```

The loop is because Brit can be in any of the house.

The generation of the CNF formulae is implemented in `einstein.py`, and the actual formulae are in `einstein.cnf`.

Solution

Upon feeding the formulae to the SAT solver, the results are saved in `solutions.txt`. We can use the generated `reference.txt` to check which relationship the positive literals represent.

```
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)
```

We can see that house 4 has a pet fish and in it resides a German.

References

Marques-Silva, J. (1999, September). The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence* (pp. 62-74). Springer, Berlin, Heidelberg.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001, June). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference* (pp. 530-535). ACM.