DPLL Based SAT-Solver

Kevin Wu
1600012832

July 16, 2017

1 Key Implementation Ideas

1. **Data Structure:** The way I chose to express clauses is to have D *a dictionary of variable to the set of equation index it appears*, E a *list of equations which are sets of variables*, and L *a list of current equation index.*
2. **Preparation:** In *DPLL*, my program will not check whether there exists both X and $\neg X$ in the same clause. So I checked this in my preparation part, which can save some search time.
3. **Basic Property I:** Any time in the program running, it is guaranteed that the 3 data structures above only represent current situation, which means eliminated variables and equations will not appear in any of them.
4. **Set Variable:** *Set Variable* is a fundamental function for *BCP*, *Set Pure*, and *Decision.* To keep *Property I*, when using D to go through the equation E_k where variable v appears, I need to check two thing:
   (a) If the equation becomes empty, I need to eliminate it in L.
   (b) If the equation becomes true, I need to eliminate it in L, and remove all the other variables v' in this equation which clears E_k, and get $D[v']$ rid of k.
5. **Basic Property II:** Whenever program goes into *Set Variable*, it is assumed that the assignment will not generate conflicts. Thus *Set Variable* does not check satisfiability. Moves described below will guarantee this.
6. **BCP:** *BCP* needs to go through E to single out those one literal equation. Use a *set* to store them. When one is added, check whether its negation is inside. There is a potential trap in it depending how you implement *BCP*, I will describe it below.
7. **Set Pure:** *Set Pure* needs to go through D to pick those literals the negation of which does not appear in equations. After this, set them to *true*, which will maximumly preserve the satisfiability.
8. **Decision:** *Decision* is what program will do after *BCP* and *Set Pure.* The way to pick what variable to try the assignment is pretty tricky. I tried several strategies, such as *Random Pick*, *Frequent Pick*, *Balanced Pick*[1]. It turns out *Pick the most frequently appeared* is about 30% faster than *Random Pick* and way better than *Balanced Pick. Frequent Pick* and *Balanced Pick* combined together can not provide substantial improvement.
9. **DPLL:** *DPLL* is basically the combination of the algorithms above. To maintain *Property I*, I need to fix change when program recalls. Considering efficiency, I have 2 set of *lists*; one records changes from *BCP* and *Set Pure*, one records changes from *Decision.* When program backtracks from the first assignment trial, undo the *Decision* change; when it returns from the second trial, undo all the change and return to parent function.

2 Potential Traps

1. In *BCP*, there is a case needs further consider. Assume after you go through all the clauses containing one literal, these literals are set *true* at the same time without further consideration, there will be a potential bug. Considering this case $\{X\}, \{\neg X, \neg Y\}, \{Y\}$. If you set $X = true$ and $Y = true$ without checking the clause $\{\neg X, \neg Y\}$, it will be satisfiable, which is incorrect. So in my version, I set them one by one.
2. Pay attention to Python Import. Name file and function carefully. I was bothered by the strange error log when I have file like *string.py* or *parser.py* with function *parse* in it.
3. I used test data from UBC website[2]. In these data, there is a mysterious % at the end of every file input. So, I had to add a special check for this in my *CNFparser.py.*

Note: Aside from DPLL based SAT-Solver, I also implemented a CDCL based SAT-Solver, the code and report of which can be found in `thiswebsite`

[1] *Balanced Pick* means the difference between it and its negation is as small as possible. It indicates either *true* or *false* assignment is balanced in *DFS.*

[2] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`