Key Implementation Ideas

Kevin Wu 1600012832

1 Clarification

After finishing DPLL, I found it quite slow in some unsat cases. So I implemented CDCL. They all have advantages. The classification in *main.py* was derived from test results of several different kinds of dataset.

2 DPLL

1. **Data Structure:** D: *a dictionary of variable mapping to the set of clause it appears in*; E: *a list of clauses which are variable sets*; L: *a list of current clause index.* (These are abbreviations of the real name in code.)
2. **Preparation:** Eliminate those clauses with both X and $\neg X$ inside in advance.
3. **Basic Property:** Any time in the running, it is guaranteed that the 3 data structures only represent current situation, which means eliminated variables and clauses will not appear in any of them.
4. **Set Variable:** To maintain *Property*, when using D to go through clauses E_k where variable v appears, these need to be checked during assignment:
   (a) **case 1:** E_k is unitary. If E_k becomes $false$, return a conflict; otherwise, eliminate it in L.
   (b) **case 2:** E_k is not unitary and it becomes *true*. Remove all $v' \in E_k$ and get $D[v']$ rid of k.
5. **BCP:** *BCP* needs to go through E to single out unitary clauses. Use a *set* to store literals. When one is added, check whether its negation is inside. Also, consider this special case—$\{X\}, \{\neg X, \neg Y\}, \{Y\}$. X and Y can not be set simultaneously without checking $\{\neg X, \neg Y\}$. I solve this by the checker in *Set Variable*.
6. **Set Pure:** *Set Pure* needs to go through D to pick the literals, the negation of which does not appear in E. After this, set them to *true*, which will maximumly preserve the satisfiability.
7. **Decision:** I tried several strategies like *Random Pick*, *Frequent Pick*[1], *Balanced Pick*[2]. *Frequent Pick* is the fastest.
8. **DPLL:** For efficiency, I have 2 *list*; one records changes from *BCP* and *Set Pure*, one records *Decision* change. When program first backtracks, undo the *Decision* change; after its second recursion, fix all the change and return.

3 CDCL

1. **Main Process:**
   (a) Select a variable and assign *true* or *False*.
   (b) Apply Boolean constraint propagation (BCP).
   (c) Build implication graph.
   (d) If conflict occurs, analyze it and back jump. Otherwise continue from step 1 until all variables are assigned.
2. **Set Pure:** Based on the instructions from Wikipedia, there is no *Set Pure* procedure. But this process does no harm to other parts; so I added this feature in my code like DPLL. Since it preserves the satisfiability, those literals eliminated by *Set Pure* do not need to be drawn into *Implication Graph.*
3. **Implication Graph:** This graph (DAG) shows how the program determines every literals step by step. Basically, when you do *BCP*, mark the variable as *deduction* and add edges from other variables in the clause to it; when you do *Decision*, mark the variable as *decision.*
4. **Conflict Analysis:** This tells the program why conflict occurs. From conflict point, back search all the *decision* points that contribute to this *deduction*; these variables form a new clause. And then add it to the original clause group. Since it is a DAG, I implicitly build the graph during *BCP*, and store the clause with the variable. So when conflict happens, I can obtain the clause immediately rather than waste time on back searching.
5. **Clause Learning:** The learned clause is from *Conflict Analysis.* But how to deal with those eliminated literals (in previous *Decision* and *BCP*) in the new clause needs further consideration. Since my implementation needs to guarantee that any time, data structures only represent current situation, I have to maintain the stack manually and mark those literals as eliminated in former stack level so as to recover them in recursion.
6. **Back Jump:** From the learned clause, the program can jump back to the second latest *decision*, for the last one can be determined by the *BCP* there.
7. **Decision:** Generally, I find *Frequent Balanced Pick* is better, which is assume X appears l_1 times and $\neg X$ appears l_2 times, choose the literal with maximum $l_1 + l_2 - |l_1 - l_2|$. But the old strategy works better in *flat* dataset.

[1] *Frequent Pick* means to choose the variable (and its negation) that mostly appears.
[2] *Balanced* means the smallest difference between variable and its negation. It indicates $true$ and $false$ assignment are more balanced in *DFS.*