CS4244 Project1

SAT Solver Based on CDCL Algorithm

Progress overview

I have implemented a SAT solver based on CDCL algorithm.
My idea is first to build a naive solver using the basic algorithm on branching and conflict analysis, and then attempt to improve upon them.
So far, I have not come to improve the branching algorithm. But I have made some progress on conflict analysis.

Implementation details

Previously, I primarily used C/C++ and Python, but I decided to try a new language for this project and chose Java.
There are 3 important parts in my implementation:

1. CNF
   The CNF part describes the data structure of CNF and some methods to manipulate the CNF. There is also a CNFConstructor class used to convert a string to CNF. (Now I use string as input for convenience in debug, later will change to DIMACS format.)
2. Trace
   Trace is used in conflict analysis. It will record the information in unit propagation.
3. Solver
   Solver includes the algorithms to solve the SAT.
   (Additionally, I wrote a tool called “CloneUnit” to perform deep cloning, since there is no existing Java API to do so.)

CDCL algorithm

Below is the algorithm used to solve SAT in my implementation:

```
// make a copy

CNF cnfCopy = CloneUnit.deepClone(this.cnf);

while(true) {

 // do unit propagation first

 unitPropagation();

 // check whether there is conflict

 if(findConflict()) {

  if(this.currentLevel == 0) {
```

```
   // UNSAT
   return null;
  }
  else {
   // do conflict analysis
   backTrack(learnFromConflict());
   continue;
  }
 }
 // if run to here means there is no conflict
 // choose a branching variable
 if(doOneDecision() == true) {
  // have unassigned literal
  continue;
 }
 else {
  // have no unassigned literal --> SAT solved
  cnfCopy = reAssignFromTrace(cnfCopy , this.trace);
  return cnfCopy;
 }
}
```

The most critical aspects of this algorithm are methods “doOneDecision()” and “learnFromConflict()”.

1. doOneDecision()
   Method “doOneDecision()” is used to choose the branching variable, now using the naive algorithm: choose the first meet unassigned proposition. It should have the same effectiveness as “random choose” in the statistical sense.
2. learnFromConflict()

Method “learnFromConflict()” will learn from the information recorded in trace when a conflict is detected. It will learn a new clause and add it to the cnf. Then return the decision level should backtrack to.

Conflict analysis

I found a better heuristic to learn the new clause and determine the decision level to backtrack to.
As in the lecture, we can build a directed acyclic graph to describe the process of doing assignments and unit propagation.

Definition:

1. UIP(unique implication point):
   A 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.

2. UIP cut
   A cut W=(A, B) is UIP cut if and only if B is the set of all nodes that is the successor of a UIP and has a path to the conflict node, and A is the set of rest all nodes.

3. Reason set
   W=(A, B) is a UIP cut. The reason set of W is:
   $R=\{l \in A \mid \exists l' \in B, (l, l') \in E\}$.

4. Decision node
   A decision node is any node assigned by branching variable picking instead of unit propagation.

We can observe that the decision node at the current decision level must be a UIP.
When there is a conflict, we first find the current decision level node *p*. Then find its UIP cut W=(A, B) and the reason set R.
The new clause learned is: $C = V \neg l \ (l \in R)$.
Next, we use C to determine the decision level to backtrack to:
If all literals in C have the same decision level, we backtrack to level 0.
Otherwise, we backtrack to the second-largest decision level in C.

Future work

1. Improve the branching variable picking algorithm.
2. Find the first UIP in conflict analysis instead of simply choosing the decision node at the current decision level.
3. Switch the input from my self-defined string format to the DIMACS format.