PA-5 [21F] [Benjamin Cape] [CNF]

Description

> How do your implemented algorithms work? What design decisions did you make? How you laid out the problems?

My algorithms work almost identical to the psuedocode that was provided. One key component that I tried to make clear in my code is the representation of the problem at hand, and of the CNF. As we know from class, a CNF is a conjunction of disjunction of variables. To represent a CNF I created three data types which make thinking about a CNF clear and concise.

1. `CNF`. A `CNF` contains a list of `Disjunction`s. The list is the world that is true under the CNF.
2. `Disjunction`. A `Disjunction` is a list of Variables, where at least one of the variables must be true in order to the `Disjunction` to be satisfied. This is due to the `OR` restriction on the variables.
3. `Variable`. A `Variable` contains two pieces of data:

```
{
    var: int,
    true: bool
}
```

A `Variable` represents whether an integer (which we use to represent distinct variables in an assignment (or model)), is meant to be true or false in some `Disjunction`. This is helpful in solving on the problems mentioned in the implementation notes of differentiated `0` and `-0` as two different values for a variable.

This also doesn't take any extra storage, since we are essentially created our own way of storing a signed integer with 33 bits of data, rather than using 2s complement to use 32 bits to store number up to 31 bits as signed.

These three data structures combine to help us manipulate CNFs, and ask coherent questions about them. For example, each of these three DS's has a function `satisfied(model: Model)` which checks whether a CNF, a Disjunction, or a variable, is satisfied under the specified model. There are three outputs for this function: `Satisfied, UnSatisfied, Unknown`. `Unknown` will occur if we cannot determine whether the CNF is true under that model, and some variable doesn't have a value.

The reason for making this more robust is an attempt to also develop for the dpll algorithm, rather than only for the fully realized models of the `*SAT` algorithms. The GSAT and the WALKSAT algorithms can do only with the `Satisfied` and `UnSatisfied` solutions since all models will contain every variable assigned.

GSAT

This algorithm is implemented IDENTICALLY to the psuedocode provided. It uses random sampling to pick a variable to flip, and uses a simply for loop to loop over all variables to find one to flip if we don't exceed the threshold. In order to quickly query the score with the max score, we save all variables with a given score in an associative map, and keep track of the max score along the way. Then we sample from that list. In the enhanced walksat we manage the size of this dictionary so that it doesn't get too big.

WALKSAT

This algorithm is nearly identically, but, as noted in the assignment uses a slightly different approach that is only checking the variables of some randomly selected unsatisfied clause to as candidates for swapping. To check this we use the helper functions specified in the Description above to find sentences that are not satisfied, and then use `random` to sample one of them.

Enhanced WALKSAT To speed up the WALKSAT algorithm I have implemented some useful functions that implement a sort of cache layer over the WALKSAT unsatisfied checks, as well as a visited dictionary to make sure that we don't re-check visited nodes.

The first enhancement, again picks a random unsatisfied clause, and determines the number of satisfied clauses under that model. Then, using a map of variables to the clauses that they exist within, for each variable that is a candidate to switch, only loops over the clauses that it exists within to check if they have gone from `sat -> unsat` or `unsat -> sat`. Using this value we can more quickly and easily find the number of satisfied clauses when swapping a variable.

Another addition, if we've already tried to flip all the variables with the highest score, then we default to flipping a random one. This way we never check a model twice.

Evaluation

> Do your implemented algorithms actually work? How well? If it doesn't work, can you tell why not? What partial successes did you have that deserve partial credit?

Yes, my algorithms work correctly. You can see this by testing on small sets which return quickly. You can also see that CNFs that are inherently false will never find a solution, and will search all 2^n possible assignments.

I think the enhanced algorithm works relatively well. We can see based on some time trial testing I have done. WE also see at higher values, it visits WAY fewer nodes.

```
SAT Stats:
-------
Name: one_cell.cnf
Threshold: 0.7
Sentences: 37
Constants: 0
Variables: 9
-------

Enhanced Walksat 0.0016100406646728516
Enhanced walksat:  (6, True)
8 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0

walksat:  0.0011420249938964844
walksat:  (5, True)
gsat:  0.003648042678833008
gsat:  (4, True)
```

Here the difference is negligible, But when the number of sentences goes up we see a dramatic difference:

```
SAT Stats:
-------
Name: all_cells.cnf
Threshold: 0.7
Sentences: 2997
Constants: 0
Variables: 729
-------

Enhanced Walksat 2.2174251079559326
Enhanced walksat:  (403, True)
6 4 9 | 9 1 5 | 6 1 7
5 1 5 | 8 8 2 | 1 5 5
7 8 2 | 9 5 9 | 2 5 5
---------------------
4 7 7 | 7 8 2 | 6 7 7
```

```
6 6 9 | 4 9 9 | 5 5 9
1 4 2 | 1 3 6 | 6 6 9
---------------------
6 6 3 | 1 1 1 | 4 2 5
5 5 7 | 9 1 4 | 4 1 6
8 3 7 | 2 6 1 | 4 9 7

walksat:  3.853825092315674
walksat:  (483, True)
```

Or an even bigger difference

```
SAT Stats:
-------
Name: rules.cnf
Threshold: 0.7
Sentences: 3240
Constants: 0
Variables: 729
-------

Enhanced Walksat 32.51748299598694
Enhanced walksat:  (4518, True)
3 7 9 | 5 1 4 | 6 2 8
8 1 5 | 7 6 2 | 9 4 3
2 6 4 | 9 3 8 | 7 5 1
---------------------
4 5 2 | 1 7 3 | 8 9 6
1 9 3 | 8 4 6 | 5 7 2
6 8 7 | 2 5 9 | 3 1 4
---------------------
5 4 8 | 3 9 1 | 2 6 7
7 2 1 | 6 8 5 | 4 3 9
9 3 6 | 4 2 7 | 1 8 5
```

```
SAT Stats:
-------
Name: puzzle1.cnf
Threshold: 0.7
Sentences: 3250
Constants: 90
Variables: 729
-------

Enhanced Walksat 286.7374448776245
Enhanced walksat:  (37497, True)
5 1 8 | 7 4 6 | 9 2 3
```

```
2 7 6 | 3 9 5 | 8 4 1
9 4 3 | 2 8 1 | 5 6 7
---------------------
8 2 5 | 1 6 4 | 7 3 9
1 6 9 | 8 7 3 | 4 5 2
7 3 4 | 5 2 9 | 6 1 8
---------------------
6 5 1 | 9 3 7 | 2 8 4
4 9 2 | 6 1 8 | 3 7 5
3 8 7 | 4 5 2 | 1 9 6

SAT Stats:
-------
Name: puzzle2.cnf
Threshold: 0.7
Sentences: 3251
Constants: 99
Variables: 729
-------

Enhanced Walksat 73.5916051864624
Enhanced walksat:  (9963, True)
6 3 1 | 9 8 7 | 5 4 2
5 2 7 | 1 4 6 | 8 3 9
4 9 8 | 2 5 3 | 6 1 7
---------------------
8 1 9 | 7 6 4 | 2 5 3
3 5 6 | 8 2 1 | 9 7 4
7 4 2 | 3 9 5 | 1 8 6
---------------------
9 7 5 | 4 1 2 | 3 6 8
1 8 3 | 6 7 9 | 4 2 5
2 6 4 | 5 3 8 | 7 9 1
```

Extra Credit

The Enhanced WALKSAT is my extra credit. It saves visited nodes, and terminates when we have visited all the possible nodes. It makes sure to keep track of constants and not to swap constants. It also automatically flips a random variable if we cannot/have already tried flipping all the ones that are in the available list.

I have also implemented a threaded walksat, that starts walksat with golden numbers ranging from 0.1 to 0.9, and terminates when it finds the first solution, then returns the best golden number for the given problem, and the time it took to compute it! My computer doesn't like to run lots of computationally heavy

stuff all together, so for me this slows my computer down pretty substantially. But it was quite effective when I did try it.

Solving the bonus problem:

```
SAT Stats:
-------
Name: puzzle_bonus.cnf
Sentences: 3267
Constants: 243
Variables: 729
-------

Enhanced Walksat 465.41109800338745
Enhanced walksat:  (39186, True)
5 3 4 | 6 7 8 | 1 9 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 3 4 2 | 5 6 7
---------------------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 9 7 1
7 1 3 | 9 2 4 | 8 5 6
---------------------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 7 1 9
```