CSCE 420 - Spring 2022

Assignment A4

due: Tues, April 19, 2022, 3:55pm (start of class) pushed to your Github.tamu.edu account

Objective

The objective of this assignment is to implement the DPLL algorithm in C++ and use it to solve some problems using propositional Satisfiability. In your program, should implement the pseudocode for DPLL as shown in the book (Fig. 7.17), including the **Unit-Clause Heuristic (UCH)** only (you may skip the lines for the Pure-Symbol Heuristic).

You will use your DPLL program to solve 3 problems. First, you will input the KB (in CNF form) for **Sammy's Sport Shop** (from Assignment A3) and show the C2W is satisfied in the model. Second, you will use DPLL to solve the **Australia map-coloring** problem in the text book (for which there are multiple solutions). Finally, you will use your program to solve the **N-queens problem**. If you write the CNF KB correctly, you should be able to find a solution for the N-queens problem. While the 4-queens problem is relatively easy (can be solved without the UCH), the 8-queens problem has a larger KB and is harder to solve and will probably require the UCH.

Thus, after writing your program, you will test it by writing KBs (in CNF) for these problems and run DPLL on them to generate a model. In addition, you will be evaluating the impact of the Unit Clause Heuristic by monitoring the run-time (number of DPLL calls) with and without the UCH.

CNF File Format

Each line contains a clause, which is a space-separated list of literal. Positive literals are just propositional symbols (which can be any length of alphanumeric chars, plus other like '-', '_', '?', etc). Negative literals are prefixed with a '-' (meaning 'not').

Here is an example called testKB.cnf:

```
-a -b c
-raining GroundWet
a
b
-GroundWet
```

This of course represents the following KB: {¬a v ¬b v c , ¬raining v GroundWet, a , b , ¬GroundWet }, where the first 2 sentences are the CNF forms of the rules: a ∧ b → c, raining → GroundWet.

You will have to write a function to parse these files when you read them in. Also allow for blank lines and comments (starting with a "#').

You will probably want to implement a class for Clauses, which are a list of Literals. A Literal is a propositional symbol (string) with a sign (bool or int), indicating whether it is a positive or

negative literal. You can also add member functions, such as for testing whether a clause is a unit clause, or whether a clause is satisfied or falsified by a model.

DPLL Algorithm

Here is the pseudo-code from the textbook (Fig. 7.17):

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(*clauses*, *symbols*, { })

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true* ←return the model
if some clause in *clauses* is false in *model* **then return** *false* ←return a "null model" or nullptr
P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*}) ignore these 2 lines
P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*}) turn these off with "-UCH"
P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
return DPLL(*clauses*, *rest*, *model* ∪ {P=*true*}) **or**
  DPLL(*clauses*, *rest*, *model* ∪ {P=*false*}))

(Remember, you can skip the 2 lines about FIND-PURE-SYMBOL.)

Note that the last step in the algorithm is:

return `DPLL(...,model∪{P=T}) or DPLL(...,model∪{P=F})`

What this means is that this a choice-point, where you are guessing a truth value for P; you first try setting it to True and call DPLL recursively, and if that doesn't work out, you try changing it to False. Thus if the first call returns a complete model, you have succeeded and just return that model to the caller. Otherwise you make the second call to DPLL and return whatever it returns.

Model

A model in propositional logic is an assignment of truth values for each propositional symbol. In DPLL, we will also be working with partial truth assignments, where only some of the props might have truth values (and the rest are 'unbound'). Thus you can think of a model as a mapping of prop symbols to 3 values (chars 'T', 'F', '?' ; or ints 1, -1, 0 ; or however you want to store it). The model could be implemented as a data structure in several different ways, but I recommend using a hash table (*unordered_map* in STL), since you will probably need to support a 'lookup' operation, e.g. to efficiently answer 'what is the truth value for P in this model?'.

Importantly, in the first two lines of the DPLL function, where it tests the base cases (either all clauses are satisfied by the model, or at least 1 clause is falsified by the model), instead of just returning *true* or *false*, you should return either the Model, or a value indicating false (i.e. failure, which initiates backtracking). If you are allocating your models on the program heap (e.g. using new Model(…)) and then returning a Model pointer (Model*) from DPLL, then you could return a ***nullptr*** in cases like this. On the other hand, if you implement DPLL to return Model objects directly, then you might want to include something like a Boolean data member variable like 'OK' (class Model {…bool OK;…}, which is set to true for most actual models (with truth binding for props), but is set to false when you want to return a 'null object' indicating failure; the caller can then look at the value of model.OK.

A really useful function you will want to write is **Satisfies(model,clause)**, which determines whether a model satisfies a clause or not. Remember, there is a third possibility, i.e. that the model does not determine the truth value for the clause, which could happen for partial models where not every variable has a truth assignment. So one idea is that it could return an int (+1 for satisfies, -1 for falsifies, or 0 for undetermined). A model satisfies a clause if at least one of the literals is made true by the model (i.e. P=T if it is a positive literal, P, or P=F if it is a negative literal, ¬P. A model falsifies a clause if all the literals are made false. A model does not determine the truth value of a clause if none of the literals is satisfied, 0 or more are falsified, and there is at least 1 literal whose proposition does not have a truth value assignment in the current model. Satisfies() will get called in the first two lines of the DPLL all (for the base cases). You will probably need to write other similar functions, such as to find the first literal without a truth assignment, or to determine whether a clause is a **Unit Clause** given the current model. Remember that the simplest form of unit clauses is a clause of length 1 (with exactly 1 literal). However, you also need to consider clauses where there are multiple (n>1) literals, and where n-1 literals are falsified by the model and the only remaining literal is undetermined.

Program Interface:

```
usage: dpll <filename.cnf> [-UCH]
```

The program should take a CNF file on the command line, and should run until it 1) finds a model (complete truth assignment that satisfies all the clauses) or determines that the KB is unsatisfiable (has systematically explored all possible models and concluded that a satisfying one does not exist. When the program first loads the CNF input file, it should **print out a numbered list of all the clauses**. While the program is running, you might find it useful to print out tracing information, such as the models being considered in each call to DPLL or the choices (truth assignments being made), or when the Unit Clause Heuristic is used. In the case in which model is found, your program should **print out the model** (print out the truth value of every proposition, and it will also be convenient to print out just the propositions that are True); if no model is found, then it should print out a message saying 'failure' or 'unsatisfiable'. At the end, also **print out the total number of calls to DPLL**, as a measure of the run time. We will use this to see if models can be found faster with the Unit Clause Heuristic than without it. Here is an example:

```
> dpll testKB.cnf
*** DPLL ***
0. -a -b c
1. -raining GroundWet
2. a
3. b
4. -GroundWet
...tracing info...
success! found a model
model: GroundWet=F a=T b=T c=T raining=F
true props: a b c
DPLL_calls: 6
```

Importantly, you should include an optional command line flag called **"-UCH" which turns OFF the Unit Clause Heuristic**, which should be assumed to be **on by default**.

Test Cases

1. Sammy's Sport Shop

Run your DPLL program on the CNF form of the KB for Sammy's Sport Shop from A3. Show that the model includes the answer, C2W, that the contents of the middle box must be white tennis balls.

Hint: For Sammy's Sport Shop, there should only be 9 true propositions in the final model: the 6 original facts, plus 3 more for the contents of each box. If you get more, you might need to add some additional facts for the things NOT observed and the labels NOT on the boxes. For example, if box 1 was labeled W, then besides including L1W as a fact, you can also add -L1Y and -L1B, since it was not labeled these ways.

2. Map Coloring

Write a CNF KB for the Australia map-coloring problem as described in the textbook (Ch. 5). Use it to solve the problem, i.e. find a set of color assignments such that no 2 adjacent states (that share a border) have the same color.

Importantly, check whether your first solution (model) is the SAME as the coloring shown in the textbook (on in the slides on CSP). Remember that there are multiple ways to color this map. Depending on which model is returned first, it might or might not be the same as what you were expecting. Try finding a different solution by forcing one of the states to be a particular color. For example, if you initial KB (mapcolor.cnf) led to a solution in which Queensland was red (QR=T), the make a modified KB (mapcolor2.cnf) where you add an extra fact forcing Queensland to be green (QG). (You could even try adding -QR as a fact, meaning that you want any other solution where Q is not red).

to turn in:

- **2 CNF KBs** (mapcolor.cnf, mapcolor2.cnf)
- **2 transcripts**, each showing a different model
- show the two models you found in **RESULTS.txt** (and which fact you added to force the 2nd model)

3. N-queens

Use your DPLL program to solve the N-queens problem, as described in the textbook. If you write the CNF KB correctly, you should be able to find a solution for the N-queens problem for N=4, N=6, and N=8. In class, we observed that there are 2 solutions for the 4-queens problem (i.e. placing 4 queens on a 4x4 chess board such the none can attack each other). There are multiple solution for N=6 and N=8, but they are harder to find. While I could find solutions for N=4 and N=6 without the UCH, and had to use the UCH to find a solution for the 8-queens problem.

to do:

- Write yourself a script to generate all the clauses (CNF KB) for each version of the N-queens problem, where you give the value for N on the command line. Mine looked like this: 'python Nqueens.py 4 > 4queens.cnf'. Check-in your **script and each of the KBs** into your github project (in A4/)
- Show the **solution** and the **number of DPLL calls with and without the UCH** for: 4-queens, 6-queens, and 8-queens (you might not be able to solve 8-queens without UCH)
- show that there is **no solution for the 3-queens problem** (on a 3x3 chess board), by showing that the corresponding KB is **unsatisfiable** (does not have a model)
- check-in the transcripts of all these runs
- add a **summary table in RESULTS.txt** with the number of DPLL calls with and without UCH for N=4, N=6, and N=8

What to Turn In:

Put these files in subdirectory called 'A4' of your github project for this course.

- **dpll.cpp** (and any other source code files)
- **makefile** (with any flags necessary to compile on compute.cs.tamu.edu)
- **README.txt** – documentation of how to run your program (command-line arguments and flags; interactive commands, etc)
- **KBs** - knowledge bases in CNF format for each of the 3 Test Cases. If you write a script to automatically generate the KBs (since many of the clauses are repetitive and follow simple patterns), check that in too. You will *have* to do this the N-queens problem. The script can be written in any language you want.
- **Transcripts** – text files showing outputs for the Test Cases defined above.
- **RESULTS.txt** – include a summary of results for the 3 Test Cases. This includes the models you generate for the Map-color problem and Sammy's Sport Shop. For all

problems, show the number of DPLL class (using UCH by default). For the N-queens problem, show the number of DPLL calls WITH and WITHOUT for N=4, N=6, and N=8. Don't forget to show what happens for 3-queens.

Grading:

- 20% - Does your DPLL program compile on compute.cs.tamu.edu (by running 'make')?
- 20% - Does it run on simple test cases and generate the expected output?
- 20% - Does the implementation of DPLL look correct?
- 20% - Does it produce the right models for the 3 Test Cases? (and sub-cases, since some of the have multiple questions requiring several runs)
- 20% - Does the Unit Clause Heuristic lead to a more efficient search by reducing the number of DPLL calls?