CSCE 420 - Spring 2021
Programing Assignment #4
due: Fri, Apr 9, 2021, 12:00pm (noon)

Overview

The goal of this project is to write a C++ program to do logical inference using **DPLL** (Boolean satisfiability). You will use DPLL to answer some questions about the **Australia map-coloring** problem, and to generate solutions to the **N-queens problem**. Note that, while you will use the same file format as Proj3 for writing your knowledge bases, in this case, your KBs will be restricted to clauses (i.e. disjunctions, or's) only.

Implementing DPLL

Command-line interface for the program:

```
usage: DPLL <filename> [-unit]
```

After reading in your KB (list of clauses), you will call a function like: `DPLL(clauses,model)`, with model initially set to the empty truth assignment over all propositional symbols appearing in the KB. A model can be represented by a hash table (e.g. *unordered_map* in the STL) that map strings to truth values (bool).

```
typedef unordered_map<string,bool> MODEL;
MODEL* DPLL(vector<Expr*> clauses,MODEL* model);
```

DPLL() will be implemented as a recursive function, as shown in Fig 7.17 in the textbook. It will return whether a complete and consistent model could be found. If the KB is satisfiable, print out the final model (truth assignment that satisfies all the clauses); otherwise, print 'unsatisfiable'.

An important part of making DPLL efficient is implementing the **unit-clause heuristic** (UCH), where you give preference to resolving pairs of clauses where one is a unit clause. (You do NOT have to implement the pure-symbol heuristic.) Remember that you have to re-check clauses as the search progresses, because clauses with more than one literal can become unit-clauses, if all but one of the literals are made false by the model.) Whenever you assign a variable in the model, **you should print out tracing information** that indicates whether it is a **choice-point** (e.g. last 2 lines of the pseudocode), or whether you **forcing** the assignment of a truth value due to the unit-clause heuristic. You should also print out a message whenever **back-tracking** occurs (second line of the pseudocode).

At the end, when you print out the result ('unsatisfiable', or the model), you should also **print out the number of times DPLL() was called** (you can use a simple global variable as a counter). This is effectively equivalent to the number of nodes in the Search Tree (state space) that are searched. See the example transcript at the end of this handout.

Finally, in addition to passing in the KB filename on the command line, you should handle an optional flag called **'-unit'**. By default, your DPLL() function should use the unit-clause heuristic. But this flag allows the user to turn off the unit-clauses heuristic. This will allow you to evaluate

the effect of the heuristic on computational complexity. *How many DPLL calls are made on each problem with versus without the unit-clause heuristic?*

Australia Map Coloring (example in textbook)

- write the KB for map-color in CNF
- use DPLL to generate a model (*please give the answers to all these questions in your write-up, RESULTS.txt, and along with the transcripts of outputs*)
- generate a second model, where you force Queensland to be a different color (e.g. add QG to the KB, if QR is true in the first model)
- add another fact the will force an inconsistency. For example, if QG is true in the second model, then it turns out the Victoria will have all have to be green (and thus can't be blue). To make the KB unsatisfiable, add both QG and VB as facts, and use DPLL to show this is *unsatisfiable*.
- In each case, report h*ow* many DPLL calls are made on each problem **with versus without** the unit-clause heuristic.

N-queens

Use your DPLL program to solve the **6-queens problem**. As described in the textbook, the N-queens problem refers to how to place N queens on an NxN chess board such that no queens can attack each other (either vertically, horizontally, or diagonally). Your goal is to encode this as a Boolean satisfiability problem and then use DPLL to generate a model.

To represent the problem, we can use propositional symbols like "Qcr" where c is the column and r is the row. For example, for 4-queens, the layout looks like this:

Q11 Q21 Q31 Q41
Q12 Q22 Q32 Q42
Q13 Q23 Q33 Q43
Q14 Q24 Q34 Q44

You will need to write a script to generate all the clauses necessary for the N-queens problem (where N is an input argument), including clauses precluding two queens being in the same row, column, and diagonal. For example, to say the queens in column 1 and 2 cannot both be in row 4, you could say "(or (not Q14) (not Q24))". Your generated KB will have lots of clauses. There are many ways to do this. Don't forget to include sentences saying there has to be at least 1 queen in each column and/or row. (Hint: 2 queens Qia and Qjb are in the same diagonal if abs(i-j)=abs(a-b)).

A transcript of my solution to the 4-queens problem is shown at the end of this handout.

(note: there is no solution for 3-queens, so if you generate the KB and DPLL, it should fail to find a model and say 'unsatisfiable')

Your goal is to find a solution to the 6-queens problem (using the unit-clause heuristic).

Format for KB Files

To represent propositional files, we will use an ASCII-based syntax called 'symbolic expressions'. In this syntax, there are only symbols and parentheses (for representing hierarchical/nesting structure).

- Symbols include any sequence of characters or digits (or some special chars, like '_'). For example, 'A12B', 'leftTurn', '3.1415', 'not', and 'implies' are symbols. These can be atomic sentences (propositions) by themselves; operator names are also symbols.
- Logical operators are represented as lists using prefix notation, that is, a sequence of symbols or sublists surrounded by parentheses, with the operator listed first. For example, **'(and P Q)'**, **'(not X)**', and '**(or (not X) (and P Q))**'. While the unary 'not' operator always has only 1 argument, 'and' and 'or' operators may have an arbitrary number of arguments (not restricted to binary).
- Implications are written similarly as lists using the 'implies' operator, such as "**(implies (and P Q) R)**", with 2 arguments: antecedent, followed by consequent.
- Extra white space doesn't matter; also, we will assume the parser is *case-insensitive*.
- KB files can also have blank lines, and lines beginning with '#' are assumed to be comments.

For this project, all the sentences in your KB will be converted to CNF (manually, by you). In this syntax (i.e. file format), CNF sentences are just disjunctions (lists or literals with the 'or' operator). Remember that facts are just clauses (disjunctions) of length 1. For example:

- (or (not P) (not Q) R) // ¬P v ¬Q v R
- (or P) // a fact, represented as a clause of length 1

You can use the same parser (Expr class) provided for Project 3.

What to Turn In

Create a subdirectory in your (private) Github project for the class (on the TAMU Github server) called 'Proj3'. In Proj3/, you should have at least the following files:

- **README** – should explain how your program works (e.g. command-line arguments), and any important parameters or constraints
- **makefile** – we must be able to compile your C++ code on compute.cs.tamu.edu by simply calling 'make'
- **DPLL.cpp**: contains your implementation of DPLL, conforming to the command-line usage above
- KB files:
  - .cnf files, written in same file format as .kb files in P3, but with clauses only
  - **mapcolor.cnf, mapcolor2.cnf, etc** // alternative versions of the mapcolor KB, with different facts added
  - **6queens.cnf**
- transcripts: – show the output
  - **DPLL_mapcolor.txt, DPLL_mapcolor_without_UCH.txt, DPLL_mapcolor2.txt…etc**
  - **DPLL_6queens.txt**
- **RESULTS.txt** – summarize the runs you did, the inputs (which KB), the output transcript filenames, and what the results were (was it satisfiable? which model did you get? how many DPLL calls were made with and without the unit-clause heuristic? did back-tracking occur?)

The date you commit your files and push them to the TAMU Github server will be used to determine when it is turned in and whether any late penalties will be applied.

Grading

The materials you turn in will be graded according to the following criteria:

- 20% - does it compile and run without problems on compute.cs.tamu.edu?
- 20% - does the implementation look correct? (DPLL)
- 20% - do the knowledge base files look correct? (clauses)
- 20% - does it run correctly on test cases?
- 20% - do the RESULTS look correct? (transcripts, tracing info, #DPLL calls with and without unit-clause heuristic)

Example

```
> DPLL mapcolor.cnf
… print out all the clauses…
model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=? WAG=? WAR=?
num clauses satisfied: 0 out of 76
trying WAR=0
```

```
model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=? WAG=? WAR=F
num clauses satisfied: 6 out of 76
trying WAG=0

model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=? WAG=F WAR=F
num clauses satisfied: 10 out of 76
forcing WAB=1

model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=T WAG=F WAR=F
num clauses satisfied: 11 out of 76
forcing NTB=0

...

model: NSWB=F NSWG=? NSWR=? NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 72 out of 76
forcing NSWG=0

model: NSWB=F NSWG=F NSWR=? NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 75 out of 76
forcing NSWR=1

model: NSWB=F NSWG=F NSWR=T NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 76 out of 76

success!
number of DPLL calls=22 (WITH unit-clause heuristic)
here is a model:
NSWB = F
NSWG = F
NSWR = T
NTB = F
NTG = F
NTR = T
QB = T
QG = F
QR = F
SAB = F
SAG = T
SAR = F
TB = T
TG = F
TR = F
VB = T
VG = F
VR = F
```

```
WAB = T
WAG = F
WAR = F
```

> DPLL mapcolor.cnf -unit

```
… print out all the clauses…
model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=? WAG=? WAR=?
num clauses satisfied: 0 out of 76
trying WAR=0

model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=? WAG=? WAR=F
num clauses satisfied: 6 out of 76
trying WAG=0

model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=? WAG=F WAR=F
num clauses satisfied: 10 out of 76
trying WAB=0

model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=F WAG=F WAR=F
num clauses satisfied: 12 out of 76
back-tracking...
trying WAB=1

model: NSWB=? NSWG=? NSWR=? NTB=? NTG=? NTR=? QB=? QG=? QR=? SAB=? SAG=?
SAR=? TB=? TG=? TR=? VB=? VG=? VR=? WAB=T WAG=F WAR=F
num clauses satisfied: 11 out of 76
trying VR=0

...

model: NSWB=T NSWG=F NSWR=F NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 74 out of 76
back-tracking...
trying NSWG=1

model: NSWB=? NSWG=T NSWR=F NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 71 out of 76
back-tracking...
trying NSWR=1

model: NSWB=? NSWG=? NSWR=T NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 67 out of 76
trying NSWG=0
```

```
model: NSWB=? NSWG=F NSWR=T NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 72 out of 76
trying NSWB=0

model: NSWB=F NSWG=F NSWR=T NTB=F NTG=F NTR=T QB=T QG=F QR=F SAB=F SAG=T
SAR=F TB=T TG=F TR=F VB=T VG=F VR=F WAB=T WAG=F WAR=F
num clauses satisfied: 76 out of 76

success!
number of DPLL calls=39 (WITHOUT unit-clause heuristic)
here is a model:
NSWB = F
NSWG = F
NSWR = T
NTB = F
NTG = F
NTR = T
QB = T
QG = F
QR = F
SAB = F
SAG = T
SAR = F
TB = T
TG = F
TR = F
VB = T
VG = F
VR = F
WAB = T
WAG = F
WAR = F
```

> python queens.py 4 > 4queens.cnf
> DPLL 4queens.cnf

```
…
success!
number of DPLL calls=36 (WITH unit-clause heuristic)
here is a model:
Q11 = F
Q12 = F
Q13 = T
Q14 = F
Q21 = T
Q22 = F
Q23 = F
Q24 = F
Q31 = F
Q32 = F
```

```
Q33 = F
Q34 = T
Q41 = F
Q42 = T
Q43 = F
Q44 = F
```

If we were to visualize this solution, it would like this:

```
. Q . .
. . . Q
Q . . .
. . Q .
```

Note, there are also other solutions.