CSCE 420 - Fall 2024
Programming Assignment PA3

due: Thurs, Nov 12, 2024, 11:59 pm, pushed to your class Github account

Overview

The goal of this project is to write a C++ or Python program to do logical inference using **DPLL** (Davis-Putman procedure, i.e. Boolean satisfiability solver). You will use DPLL to answer some questions about the **Australia map-coloring** problem, to generate solutions to the **N-queens problem**, and for **Sammy's Sport Shop**. You will also implement the **unit-clause heuristic**, and evaluate its effect on the efficiency of finding a model.

Implementing DPLL

Command-line interface for the program:

```
usage: DPLL <filename> <literal>*
or:    python DPLL.py <filename> <literal>*
```

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 that maps propositions (strings) to truth values (integers: 1=true, -1=false, 0=unknown). You may also include extra literals (as strings) on the command line, which should be appended to the KB (asserted facts, as unit clauses). See an example of this with the map-color problem below.

DPLL() should be implemented as a recursive function, as shown in Fig 7.17 in the textbook. It will determine whether a KB is satisfiable. If a model that is complete and consistent can be found, then 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. 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.) *You do not have to implement the Pure Symbol heuristic (just skip that line in the algorithm).*

When the program first starts, echo out the command line args (to save in the output file). 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 (the model, or 'unsatisfiable'), 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.

Propositional KB File Format

You will write your KB files in the following ASCII format for Propositional Logic, for readability. You will use a tool provided to you (*convCNF.py*) to automatically transform your KBs into CNF file format, for input to your DPLL program.

ASCII format for Propositional Logic: All sentences are represented in prefix notation, using nested parentheses, where the operator comes first, followed by the arguments. The Boolean operators recognized are: 'and', 'or', 'not', and '->'. . A positive literal can be given by itself on a line without parens or operators. 'and' and 'or' can have 1, 2, or more args. Parentheses may also be used to group sub-expressions. Each propositional sentence must be on a single line. Blank lines are allowed. Comments prefixed by '#' maybe added at the end of any line. Atomic sentences/atoms are *propositions* (tokens; strings of arbitrary characters separated by spaces).

The ASCII syntax for Propositional KB files is defined as follows:

```
<sentence> ::= Prop | <complexsentence>

<complexsentence> ::= (and <sentence>*) | (or <sentence>*) |
                      (xor <sentence>*) | (not <sentence>) |
                      (implies <sentence> <sentence>) | (<-> <sentence> <sentence>)
```

Note that implications ('implies') have exactly 2 arguments, a precedent and an antecedent (in that order). Therefore '(implies A B)' means 'A→B' in infix ordering.

Here are some examples of propositional sentences in this ASCII prefix syntax:

```
(and cloudy windy)
(or banana5_is_green banana5_is_yellow banana5_black)
(implies banana5_ripe (and banana5_yellow (not banana5_green))
(implies (and at_door126 (not locked_door126)) (can_access room126))
(implies (or mammal fish bird reptile) animal)
```

However, the DPLL algorithm requires an input KB to be represented as *clauses* (disjunctions, with only a top-level 'or' operator and list of literals). The *convCNF.py* script (provided for this project) can be used to convert a file in propositional (ASCII) format into CNF format. Remember that the algorithm for converting propositional sentences to CNF can generate multiple clauses from a single sentence. A single literal by itself is also a (degenerate) clause of length one (representing a fact), e.g. "dogFido". Note that the input lines are also copied to the output as comments. Here is an example of using *convCNF.py* to convert a propositional KB into CNF format:

```
animals.kb
dogFido
(not barkFido)
(or wagFido barkFido (not awakeFido))
(-> dogFido mammalFido)
(-> (or on_A_B (and on_A_C on_C_B)) above_A_B)
(<-> rainy (not sunny))
(xor dogFido catFido)

> python convCNF.py animals.kb
# dogFido
(or dogFido)
```

```
# (not barkFido)
(or (not barkFido))

# (or wagFido barkFido (not awakeFido))
(or wagFido barkFido (not awakeFido))

# (-> dogFido mammalFido)
(or (not dogFido) mammalFido)

# (-> (or on_A_B (and on_A_C on_C_B)) above_A_B)
(or (not on_A_B) above_A_B)
(or (not on_A_C) (not on_C_B) above_A_B)

# (<-> rainy (not sunny))
(or (not rainy) (not sunny))
(or sunny rainy)

# (xor dogFido catFido)
(or dogFido catFido)
(or (not catFido) (not dogFido))
```

To make it easier for the DPLL program to parse the input CNF file, you can give the '-DIMACS' flag to the *convCNF.py* script to convert your propositional KB files into an alternative version of CNF format inspired by the DIMACS standard. This intermediate file format is easier to parse, because every line contains only a single clause (a disjunction) and we drop the parentheses and operators, e.g. **'P Q -R'** (which means **'P ∨ Q ∨ ¬R'**).

Here is the DIMACS version of the animals KB above:

```
> python convCNF.py animals.kb -DIMACS
# dogFido
dogFido
# (not barkFido)
-barkFido
# (or wagFido barkFido (not awakeFido))
wagFido barkFido -awakeFido
# (-> dogFido mammalFido)
-dogFido mammalFido
# (-> (or on_A_B (and on_A_C on_C_B)) above_A_B)
-on_A_B above_A_B
-on_A_C -on_C_B above_A_B
# (<-> rainy (not sunny))
-rainy -sunny
sunny rainy
# (xor dogFido catFido)
dogFido catFido
-catFido -dogFido
```

DIMACS CNF File Format

The input files for DPLL will be KBs consisting of propositional clauses transformed into the following DIMACS CNF format. Since a clause is a disjunction of literals (propositions or negated props), we can write each clause as a list of symbols, ignoring the 'or' symbols, and prefixing negated propositions with a minus sign '-'. For example, *apple v cherry v ¬lime* would be written as "apple cherry -lime". Propositional symbols can also contain digits and other characters like '_'. Each clause in the CNF file is written on a separate line; additional spaces don't matter. Blank lines can be skipped. Comments on any line can start with a '#' and everything after the '#' is ignored. Here is an example of a KB written in CNF format:

```
# room2 is accessible only if doors 1 and 2 are unlocked
-door1_unlocked -door2_unlocked room2_accessible
weekend door1__unlocked      # (not weekend) -> door1_unlocked
-broken_door2 door2_unlocked # broken_door2 -> door2_unlocked
-wednesday -weekend          # wednesday -> (not weekend)
# facts
wednesday
broken_door2
```

Since clauses only contain top-level 'or' operators, we drop them and just list the remaining positive or negative literals, which are propositions or their negations (which are prefixed with minus sign): e.g. **'P Q ¬R'**. Thus, each line in a DIMACS CNF file is simply a list of literals, which are implicitly assumed to be disjoined by 'or' operators (e.g. **'P ∨ Q ∨ ¬R'**). Using the '-DIMACS' flag will enable you to transform your KBs info a format for input to your DPLL program that it easier to parse.

Your DPLL program only has to support the DIMACS CNF input file format.

Unit Clause Heuristic

While back-tracking makes a big improvement in the efficiency of DPLL (by avoiding large parts of the search space containing models that won't satisfy the clauses), the Unit Clause Heuristic (UCH) can improve the efficiency of finding a satisfying model even further (necessary for scaling DPLL up to larger applications with thousands of variables and clauses).

In the simple version of the DPLL algorithm, with each call, the "next" unassigned proposition (in order) is chosen and tested by assigning it first to T and then F (a choice-point). Instead, UCH scans the list of clauses for a clause that is not satisfied by the current model, and for which all of its literals are made false by the model except one, which is unknown (unassigned) in the model. If such a clause can be found, then DPLL can just bind the truth value of that proposition to the truth value based on its sign (as a literal) in the clause. This circumvents the choice-point; there is no need to try binding the prop to T (and recursively calling DPLL to see

the model can be completed) or F (i.e. “guessing”); the sign of a unit clause determines which truth value can be used.

For example, suppose we have a set of clauses:

1. a -b c -d // (a v -b v c v -d)
2. d -c -e
3. c -f

Next, consider the model {a=F, b=F, c=?, d=F, e=T, f=?}. Clause 1 is irrelevant (UCH skips it) because it is already satisfied by -b. However, clause 2 is effectively considered to be a unit clause, given this model, because d and -e are falsified by the model, and -c is the only remaining literals which value is unknown in the model. Clause 3 is not unit because it has 2 literals with unknown truth values. Therefore, in this situation, UCH would directly extend the model by adding c=F (since it is a negative literal in the clause), without having to go through the choice-point. Think of writing down the satisfaction of the literals *given the model* as “satisfied” (S) “unsatisfied” (U) or “unknown” (?):

1. a -b c -d U , S , ? , S
2. d -c -e U , ? , U
3. c -f ?, ?

What makes clause 2 a unit clause is that it has no satisfied literals (S) and exactly 1 unknown literal (U). There is only one possible truth value remaining for that proposition (c) that would make the model satisfy the clause: in this case, c clearly has to be assigned F.

Problems to Solve with DPLL

Australia Map Coloring (example in textbook, Ch. 6)

- write the KB file for map-color in propositional logic: mapcolor.kb
- convert it to CNF format:
    - > python convCNF.py mapcolor.kb -DIMACS > mapcolor.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*)
- create a modified KB file to 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)
- create a modified KB file by adding another fact that 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.

Sammy's Sport Shop

Save the knowledge base you created in Homework #2 and converted to CNF as a .cnf file (i.e. clauses in the ASCII format as described above). The KB should only have the general rules in it, but nothing specific to any particular scenario (i.e. do not include any facts).

Use your DPLL program to generate the solution to the scenario described in Homework #2. Note that you can add the corresponding facts (labels and observations) to the command-line.

Scenario A (same as in HW#2):

- facts: O1Y, O2W, O3Y, L1W, L2Y, L3B
- commands:
  - `python convCNF.py sammy.kb -DIMACS > sammy.cnf`
  - `DPLL sammy.cnf O1Y O2W O3Y L1W L2Y L3B`
- expected result: check that C2W is in the final model

box #:	1	2	3
observed ball:	yellow	white	yellow
labels (incorrect):	white	yellow	both
inferred contents:	?	?	?

Now try solving a different version of the problem, where a yellow ball is drawn from box 2 and white from boxes 1 and 3, and box 1 is labeled 'white' (incorrectly) and boxes 2 and 3 are labeled 'both'.

Scenario B (new):

- facts: O1W O2Y O3W L1W L2B L3B
- command: `DPLL sammy.cnf O1W O2Y O3W L1W L2B L3B`
- result: are the true contents of the boxes different than in scenario A?

box #:	1	2	3
observed ball:	white	yellow	white
labels (incorrect):	white	both	both
inferred contents:	?	?	?

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 propositional KB with all the logic sentences necessary for the N-queens problem (where N is an input argument), including constraints *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 “-Q14 -Q24”. Don’t forget to include sentences saying there has to be *at least 1* queen in each column and/or row. Then convert it to DIMACS CNF format for input to DPLL. There will be a separate KB for 3-queens, 4-queens, 5-queens, and 6-queens. Your KBs will have lots of similar sentences (because you cannot use variables in propositional logic). However, many of the sentences are repetitive and follow a pattern, so you might want to write a *script* to generate them, for each version of the problem (N=3..6).

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 run DPLL on it, it should fail to find a model and say ‘unsatisfiable’)

Your goal is to find a solution to the 4-queens, 5-queens, and 6-queens problem (using the unit-clause heuristic), and show that **3-queens** is *unsatisfiable*.

What to Turn In

Create a subdirectory in your (private) Github project for the class (on github.com) called ‘P3’. In P3/, 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’, or be able to run it using python3
- **DPLL.cpp or DPLL.py**: contains your implementation of DPLL, conforming to the command-line usage above
- **KB files**: files with propositional sentences written in ASCII format described above
  - **mapcolor.kb**
  - **sammy.kb**
  - **3queens.kb, 4queens.kb, 5queens.kb, and 6queens.kb**
- **CNF files**: .cnf files with clauses in DIMACS format, generated using convCNF.py
  - **mapcolor.cnf**
  - **sammy.cnf**

    - **3queens.cnf, 4queens.cnf, 5queens.cnf, and 6queens.cnf**
- **transcripts**: – show the outputs
    - **results_mapcolor.txt, results_mapcolor_QG.txt, results_mapcolor_QG_VB.txt (adding facts via the command line)**
    - **results_sammy_scenarioA.txt, results_sammy_scenarioB.txt**
    - **results_3queens.txt, results_4queens.txt, results_5queens.txt, results_6queens.txt**
- **RESULTS.txt** –
    - a text file with a table summarizing the results of all your runs:
        - command line inputs
        - model output (just the positive literals) (or 'unsatisfiable')
        - number of DPLL calls

The date you commit your files and push them to the 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, UCH)
- 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)

Examples

```
> python DPLL.py mapcolor.cnf
model: {'NSWR': 0, 'QR': 0, 'TR': 0, 'VG': 0, 'WAG': 0, 'SAB': 0, 'NTR': 0,
'SAG': 0, 'VB': 0, 'WAB': 0, 'VR': 0, 'TG': 0, 'NSWB': 0, 'QB': 0, 'QG': 0,
'NSWG': 0, 'SAR': 0, 'NTB': 0, 'TB': 0, 'WAR': 0, 'NTG': 0}
trying NSWB=T
model: {'VB': 0, 'VG': 0, 'WAG': 0, 'SAG': 0, 'WAB': 0, 'VR': 0, 'SAR': 0,
'WAR': 0, 'NSWR': 0, 'QR': 0, 'SAB': 0, 'NTR': 0, 'TR': 0, 'NSWB': 1, 'QB':
0, 'QG': 0, 'NSWG': 0, 'TG': 0, 'NTB': 0, 'TB': 0, 'NTG': 0}
trying NSWG=T
model: {'VB': 0, 'VG': 0, 'WAG': 0, 'TR': 0, 'WAB': 0, 'VR': 0, 'TG': 0,
'WAR': 0, 'NSWR': 0, 'QR': 0, 'SAB': 0, 'NTR': 0, 'SAG': 0, 'NSWB': 1, 'QB':
0, 'QG': 0, 'NSWG': 1, 'SAR': 0, 'NTB': 0, 'TB': 0, 'NTG': 0}
backtracking
trying NSWG=F
model: {'VB': 0, 'VG': 0, 'WAG': 0, 'TR': 0, 'WAB': 0, 'VR': 0, 'TG': 0,
'WAR': 0, 'NSWR': 0, 'QR': 0, 'SAB': 0, 'NTR': 0, 'SAG': 0, 'NSWB': 1, 'QB':
0, 'QG': 0, 'NSWG': -1, 'SAR': 0, 'NTB': 0, 'TB': 0, 'NTG': 0}
...
```

```
solution:
NSWB: 1
NSWG: -1
NSWR: -1
NTB: 1
NTG: -1
NTR: -1
QB: -1
QG: 1
QR: -1
SAB: -1
SAG: -1
SAR: 1
TB: 1
TG: -1
TR: -1
VB: -1
VG: 1
VR: -1
WAB: -1
WAG: 1
WAR: -1
just the Satisfied (true) propositions:
VG WAG SAR NSWB QG NTB TB
total DPLL calls: 36
```

> DPLL mapcolor.cnf

```
...
just the Satisfied (true) propositions:
VG WAG SAR NSWB QG NTB TB
total DPLL calls: 22
```

Since Victoria is green in the model above, re-run with -VG to get a different solution:

> DPLL mapcolor.cnf -VG

```
...
just the Satisfied (true) propositions:
SAG VR WAR QR NSWB NTB TB
total DPLL calls: 74
```

> DPLL 4queens.cnf

```
model: {'Q33': 0, 'Q44': 0, 'Q42': 0, 'Q43': 0, 'Q21': 0, 'Q34': 0, 'Q31': 0,
'Q32': 0, 'Q14': 0, 'Q22': 0, 'Q23': 0, 'Q11': 0, 'Q13': 0, 'Q12': 0, 'Q24':
0, 'Q41': 0}
trying Q11=T
model: {'Q12': 0, 'Q44': 0, 'Q42': 0, 'Q43': 0, 'Q32': 0, 'Q41': 0, 'Q22': 0,
'Q33': 0, 'Q21': 0, 'Q31': 0, 'Q23': 0, 'Q11': 1, 'Q13': 0, 'Q34': 0, 'Q24':
0, 'Q14': 0}
forcing Q12=-1 by UCH
model: {'Q34': 0, 'Q44': 0, 'Q42': 0, 'Q43': 0, 'Q41': 0, 'Q21': 0, 'Q31': 0,
'Q33': 0, 'Q32': 0, 'Q22': 0, 'Q23': 0, 'Q11': 1, 'Q13': 0, 'Q12': -1, 'Q24':
0, 'Q14': 0}
...
solution (model):
Q11: -1
Q12: 1
Q13: -1
Q14: -1
Q21: -1
Q22: -1
Q23: -1
Q24: 1
Q31: 1
Q32: -1
Q33: -1
Q34: -1
Q41: -1
Q42: -1
Q43: 1
Q44: -1
just the Satisfied (true) propositions:
Q43 Q31 Q12 Q24
total DPLL calls: 34
```

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

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

Note, there are also other solutions. If you add “-Q43” on the command-line, it will show you a different solution with no queen in this position:

```
> DPLL 4queens.cnf -Q43
...
just the Satisfied (true) propositions:
Q34 Q42 Q21 Q13
total DPLL calls: 44
```

If we were to visualize this alternative solution, it would like this, which is symmetric to the solution above:

. . Q .
Q . . .
. . . Q
. Q . .