CSCE 625
Programing Assignment #5
due: Tues, Nov 17 (by start of class)

Propositional Satisfiability Solver (DPLL)

The goal of this assignment is to implement DPLL and to apply it to solving a multi-agent coordination problem. The objective of this assignment is to see how Boolean satisfiability can be used as an algorithm to make intelligent decisions, and how this can be accomplished by finding a model (satisfying propositional truth assignment) using DPLL.

Assembling Teams of Builder Agents

Imagine you work as a dispatcher for a company that sends out teams of robots to work sites to build or repair things. You have a total of 8 robots available. Each has a different set of capabilities.

agent	capabilities
a	painter stapler recharger welder
b	cutter sander welder stapler
c	cutter painter
d	sander welder recharger
e	painter stapler welder
f	stapler welder joiner recharger
g	stapler gluer painter recharger
h	cutter gluer

The transport vehicle only has room for 3 agents at a time. Thus, when a request comes in, you must identify a team of 3 agents or less whose combination of skills covers the task requirements.

Your job (as programmer) is to encode this domain as a Boolean Satisfiability problem and solve it using your implementation of DPLL. This will require translating the constraints of the problem (and the capabilities of the 8 robots) into sentences in Propositional Logic. You will also have to convert them to clauses (CNF) for DPLL. Then, for a given set of task requirements, you will have to add these as facts and use DPLL to determine whether the whole set of clauses is satisfiable. If a solution can be found, then the truth values in the model should tell you which team of robots can be dispatched to handle the task.

Here are some examples. Note that I have two command line arguments: an input file with propositional clauses in the same format as Project 4 (resolution), and a list of job requirements (to assert as propositional facts).

```
> python dpll.py agent_Jobs.kb "painter sander gluer joiner"
... lots of tracing information showing each step ...
... solution!: print out truth assignment for model ...
agent team: b f g

> python dpll.py agent_Jobs.kb "cutter welder painter jointer recharger"
... lots of tracing information showing each step ...
... solution!: print out truth assignment for model ...
agent team: a b f
```

Implementation

You should implement DPLL pretty much as described in Figure 7.17 in the textbook. You can use any programming language you like. Your program will start by reading a set of input clauses in the same file format as in Programming Assignment #4 (each clause on a separate line, given as a list of literals (without the implicit disjunction symbols), and using a minus prefix to indicate negation). For example, the clauses { A v B v ¬C, ¬B v D} would be written as follows:

```
A B -C
-B D
```

DPLL is basically a recursive backtracking procedure that searches the space of truth assignments (over the propositional symbols mentioned in the clauses). You start with an empty assignment. Then, with each recursive call, the routine chooses the next unassigned variable (proposition), tries binding it to True, makes a recursive call to DPLL to extend it, and if that fails (does not yield a solution), then tries False. DPLL is made more efficient through the use of two heuristics: Unit Clause and Pure Symbol. You will have to write functions for these that are called by DPLL and, given a set of clauses and a partial assignment, determines whether there is a variable/truth-value combination that has the Unit Clause or Pure Symbol property (remember to ignore clauses that are already satisfied by the partial model).

When you develop your DPLL code, you should test it on a simple problem like the one we did in class, or the one shown in the transcript below. You should print out the partial assignment with each call, along with other useful tracking information like which variable/truth-value is tried at each choice point, when backtracking occurs and why (i.e. which clause was violated), and which variable/value combination is selected whenever the Unit Clause or Pure Symbol heuristic is invoked. I recommend you start by developing the DPLL routine *without* the heuristics (just comment-out these steps) to get the initial backtracking mechanism working. Then you can add-in the heuristics. You should see a reduction in the number of nodes of the state space searched, due to increased search efficiency. For the example KB shown in the transcript, I initially solved it using only backtracking, and it took 56 step. When I added the Unit clause heuristic, the number of nodes searched went down to 34. And with both heuristics, it went down to 16 (and backtracking steps were completely eliminated).

DPLL mode	nodes are searched before a solution is found*
backtracking alone	56
backtracking with Unit Clause heuristic	34
backtracking with Unit and Pure heuristics	16

* Nodes are partial truth assignments checked. You could track this by number of calls to DPLL().

What to Turn In (via CSNet):

- your source code
- a short document describing how to compile and run your program
- the knowledge base for the builder-agent team assignment problem
- a transcript of running your program on the abstract Boolean problem (see below) and the multi-agent task-assignment problem (for the two queries shown above)
- a table of number of nodes searched for each of these problems using a) backtracking alone, b) backtracking with the Unit clause heuristic, and c) backtracking and both heuristics
- we will also test your code by running it on another set of propositional clauses

Example Transcript

Consider the following example KB:

$\overline{a} \vee \overline{f} \vee g$
$\overline{a} \vee \overline{b} \vee \overline{h}$
$a \vee c$
$a \vee \overline{i} \vee \overline{l}$
$a \vee \overline{k} \vee \overline{j}$
$b \vee d$
$b \vee g \vee \overline{n}$
$b \vee \overline{f} \vee n \vee k$
$\overline{c} \vee k$
$\overline{c} \vee \overline{k} \vee \overline{i} \vee l$
$c \vee h \vee n \vee \overline{m}$
$c \vee l$
$d \vee \overline{k} \vee l$
$d \vee \overline{g} \vee l$
$\overline{g} \vee n \vee o$
$h \vee \overline{o} \vee \overline{j} \vee n$
$\overline{i} \vee j$
$\overline{d} \vee \overline{l} \vee \overline{m}$
$\overline{e} \vee m \vee \overline{n}$
$\overline{f} \vee h \vee i$

Suppose you want to show that this set of clauses is satisfiable, and to produce a model (i.e. truth assignment) that satisfies it. First, we represent the clauses in the input file format:

example.kb:

```
-a -f g
-a -b -h
a c
a -i -l
a -k -j
b d
b g -n
b -f n k
-c k
-c -k -i l
c h n -m
c l
d -k l
d -g l
-g n o
h -o -j n
-i j
-d -l -m
-e m -n
-f h i
```

Then we run DPLL on this, which finds and prints out a model at the bottom.

```
201 sun> python dpll.py example.kb
props:
a b c d e f g h i j k l m n o
initial clauses:
0: (-a v -f v g)
1: (-a v -b v -h)
2: (a v c)
3: (a v -i v -l)
4: (a v -j v -k)
5: (b v d)
6: (b v g v -n)
```

```
7: (b v -f v k v n)
8: (-c v k)
9: (-c v -i v -k v l)
10: (c v h v -m v n)
11: (c v l)
12: (d v -k v l)
13: (d v -g v l)
14: (-g v n v o)
15: (h v -j v n v -o)
16: (-i v j)
17: (-d v -l v -m)
18: (-e v m v -n)
19: (-f v h v i)
-----------
model= {}
pure_symbol on e=False
model= {'e': False}
pure_symbol on f=False
model= {'e': False, 'f': False}
pure_symbol on i=False
model= {'i': False, 'e': False, 'f': False}
pure_symbol on j=False
model= {'i': False, 'j': False, 'e': False, 'f': False}
pure_symbol on m=False
model= {'i': False, 'm': False, 'j': False, 'e': False, 'f': False}
pure_symbol on d=True
model= {'e': False, 'd': True, 'f': False, 'i': False, 'j': False, 'm': False}
pure_symbol on h=False
model= {'e': False, 'd': True, 'f': False, 'i': False, 'h': False, 'j': False, 'm':
False}
pure_symbol on a=True
model= {'a': True, 'e': False, 'd': True, 'f': False, 'i': False, 'h': False, 'j':
False, 'm': False}
pure_symbol on b=True
model= {'a': True, 'b': True, 'e': False, 'd': True, 'f': False, 'i': False, 'h':
False, 'j': False, 'm': False}
pure_symbol on g=False
model= {'a': True, 'b': True, 'e': False, 'd': True, 'g': False, 'f': False, 'i':
False, 'h': False, 'j': False, 'm': False}
pure_symbol on k=True
model= {'a': True, 'b': True, 'e': False, 'd': True, 'g': False, 'f': False, 'i':
False, 'h': False, 'k': True, 'j': False, 'm': False}
pure_symbol on c=True
model= {'a': True, 'c': True, 'b': True, 'e': False, 'd': True, 'g': False, 'f':
False, 'i': False, 'h': False, 'k': True, 'j': False, 'm': False}
trying l=T
model= {'a': True, 'c': True, 'b': True, 'e': False, 'd': True, 'g': False, 'f':
False, 'i': False, 'h': False, 'k': True, 'j': False, 'm': False, 'l': True}
trying n=T
model= {'a': True, 'c': True, 'b': True, 'e': False, 'd': True, 'g': False, 'f':
False, 'i': False, 'h': False, 'k': True, 'j': False, 'm': False, 'l': True, 'n':
True}
trying o=T
model= {'a': True, 'c': True, 'b': True, 'e': False, 'd': True, 'g': False, 'f':
False, 'i': False, 'h': False, 'k': True, 'j': False, 'm': False, 'l': True, 'o':
True, 'n': True}
-----------
nodes searched=16
solution:
a=True
b=True
c=True
d=True
```

```
e=False
f=False
g=False
h=False
i=False
j=False
k=True
l=True
m=False
n=True
o=True
-----------
true props:
a
b
c
d
k
l
n
o
```

It happens that in this case, the Pure Symbol heuristic gets used multiple times, and backtracking is totally avoided. A model is found after searching 16 partial assignments (states in the search space). However, if I turn off the Pure Symbol heuristic and use only the Unit Clause heuristic, more nodes (34) are searched and some backtracking occurs:

```
202 sun> python dpll.py example.kb
props:
a b c d e f g h i j k l m n o
initial clauses:
0: (-a v -f v g)
1: (-a v -b v -h)
2: (a v c)
3: (a v -i v -l)
4: (a v -j v -k)
5: (b v d)
6: (b v g v -n)
7: (b v -f v k v n)
8: (-c v k)
9: (-c v -i v -k v l)
10: (c v h v -m v n)
11: (c v l)
12: (d v -k v l)
13: (d v -g v l)
14: (-g v n v o)
15: (h v -j v n v -o)
16: (-i v j)
17: (-d v -l v -m)
18: (-e v m v -n)
19: (-f v h v i)
-----------
model= {}
trying a=T
model= {'a': True}
trying b=T
model= {'a': True, 'b': True}
unit_clause on (-a v -b v -h) implies h=False
model= {'a': True, 'h': False, 'b': True}
trying c=T
```

```
model= {'a': True, 'h': False, 'c': True, 'b': True}
unit_clause on (-c v k) implies k=True
model= {'a': True, 'h': False, 'c': True, 'b': True, 'k': True}
trying d=T
model= {'a': True, 'c': True, 'b': True, 'd': True, 'h': False, 'k': True}
trying e=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'h': False, 'k': True}
trying f=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'f': True, 'h': False,
'k': True}
unit_clause on (-a v -f v g) implies g=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': True,
'h': False, 'k': True}
unit_clause on (-f v h v i) implies i=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': True,
'i': True, 'h': False, 'k': True}
unit_clause on (-c v -i v -k v l) implies l=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': True,
'i': True, 'h': False, 'k': True, 'l': True}
unit_clause on (-i v j) implies j=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': True,
'i': True, 'h': False, 'k': True, 'j': True, 'l': True}
unit_clause on (-d v -l v -m) implies m=False
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': True,
'i': True, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True}
unit_clause on (-e v m v -n) implies n=False
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': True,
'i': True, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True, 'n': False}
unit_clause on (-g v n v o) implies o=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': True,
'i': True, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True, 'o': True, 'n':
False}
backtracking
trying f=F
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'f': False, 'h': False,
'k': True}
trying g=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'h': False, 'k': True}
trying i=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': True, 'h': False, 'k': True}
unit_clause on (-c v -i v -k v l) implies l=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': True, 'h': False, 'k': True, 'l': True}
unit_clause on (-i v j) implies j=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': True, 'h': False, 'k': True, 'j': True, 'l': True}
unit_clause on (-d v -l v -m) implies m=False
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': True, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True}
unit_clause on (-e v m v -n) implies n=False
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': True, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True, 'n': False}
unit_clause on (-g v n v o) implies o=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': True, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True, 'o': True, 'n':
False}
backtracking
trying i=F
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True}
trying j=T
```

```
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True}
trying l=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'l': True}
unit_clause on (-d v -l v -m) implies m=False
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True}
unit_clause on (-e v m v -n) implies n=False
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True, 'n': False}
unit_clause on (-g v n v o) implies o=True
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'm': False, 'l': True, 'o': True, 'n':
False}
backtracking
trying l=F
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'l': False}
trying m=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'm': True, 'l': False}
trying n=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'm': True, 'l': False, 'n': True}
trying o=T
model= {'a': True, 'c': True, 'b': True, 'e': True, 'd': True, 'g': True, 'f': False,
'i': False, 'h': False, 'k': True, 'j': True, 'm': True, 'l': False, 'o': True, 'n':
True}
-----------
nodes searched=34
solution:
a=True
b=True
c=True
d=True
e=True
f=False
g=True
h=False
i=False
j=True
k=True
l=False
m=True
n=True
o=True
true props:
a
b
c
d
e
g
j
k
m
n
o
```