CSCE 625
Programing Assignment #7
due: Friday, April 3 (by start of class)

Propositional Satisfiability Solver (DPLL)

The goal of this assignment is to implement DPLL and to apply it to solving the Farmer-Fox-Chicken-Grain problem. Implementing DPLL is easy; the real objective of this assignment to see how Boolean satisfiability can be used to solve problems and produce intelligent behavior. The Farmer-Fox-Chicken-Grain problem is a simple puzzle in which a farmer must use a canoe to ferry some items across a stream without letting the fox eat the chicken or the chicken eat the grain. There is a sequence of actions that solves the puzzle. It can be solved by many methods, including state-space search. However, you can also encode this problem as Boolean expression such that it is satisfiable iff there is a solution. Furthermore, the satisfying truth-assignment tells you the sequence of actions that solves the puzzle (through Boolean variables representing actions). This illustrates the broad generality and utility of propositional satisfiability algorithms for solving many different kinds of reasoning problems.

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 #6 (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 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

The Farmer-Fox-Chicken-Grain Problem

The Farmer-Fox-Chicken-Grain problem is a simple puzzle in which a farmer must use a canoe to ferry some items across a stream without letting the fox each the chicken or the chicken eat the grain. The farmer is returning from his field with a fox, a chicken, and some grain. He arrives at the left back of a stream. His goal is to get all the items across to the right bank. There is a canoe on the left bank, and the farmer can only take one item at a time in the canoe (or none). The constraint is that, if the farmer ever leaves the fox alone with the chicken, the fox will eat the chicken. Similarly, if he leave the chicken alone with the grain, the chicken will eat the grain. Is there a sequence of actions (canoe crossings) by which the farmer can safely transport all the items to the right bank of the stream? (This is a variant of many other well-known puzzles, such as Missionaries-and-Cannibals.)

How can this puzzle be encoded as a propositional satisfiability problem? We need some Boolean variables (propositional symbols) to represent states and actions. First, consider the location of an item like the chicken. We could use two variables, ChL and ChR, representing whether it is on the left or right bank. (Actually, since there are only 2 states, we could use a single variable, ChL, with 2 truth-values, T and F; but it is easier to write down the rest of the rules if we have ChL and ChR available, so we will simply add some additional clauses to constrain that these to have opposite truth values, e.g. { ChL $\vee$ ChR, $\neg$ChL $\vee\neg$ChR}).

Now, the location of the chicken (and other objects) is time-dependent. In the initial state, it is on the left, but the goal is to get it to the right. So we can expand it to a series of variables *prefixed by a time index*. For example, if we assume that a solution can be found in up to 7 times steps, then the propositional variables for the location of the chicken at different times becomes:

T0_ChL, T1_ChL, T2_ChL, T3_ChL, T4_ChL, T5_ChL, T6_ChL
T0_ChR, T1_ChR, T2_ChR, T3_ChR, T4_ChR, T5_ChR, T6_ChR

Similarly, we can have time-dependent variables for the location of the farmer, fox, and grain.

Next, we need variables representing actions. One approach is to model this domain as having 8 basic actions: moving the fox from one side to the other (L-to-R or R-to-L) in the canoe, moving the chicken, moving the grain, or going back across the river with no cargo. These 8 variables could be represented as something like this: mv_Fx_LR,_mv_Fx_RL, mv_Ch_LR, mv_Ch_RL, mv_Gr_LR, mv_Gr_RL, mv_No_LR, and mv_No_RL (the last 2 are for moving Nothing across). In each case, it is implicit that the farmer must go across with the canoe. Each of these actions can occur at different time steps. Therefore these variables must also be expanded to include all these possibilities by prefixing time indices like this:

T0_mv_Fx_LR, T1_mv_Fx_LR, T2_mv_Fx_LR,... T6_mv_Fx_LR
T0_mv_Fx_RL, T1_mv_Fx_RL, T2_mv_Fx_RL,... T6_mv_Fx_RL
T0_mv_Ch_LR, T1_mv_Ch_LR, T2_mv_Fx_LR,... T6_mv_Ch_LR
...

The action variable T0_mv_Fx_LR means that the fox is moved from left to right at time 0.

While there are a lot of action variables, only one action can be true at each time step. Thus you will have to write clause that enforce that at least one action variable is true in each step, and at most one action variable is true in each time step.

Since there will be a lot of these clauses and it is painful (and error-prone) to type them out by hand, **you should write a simple script to enumerate all the clauses** (by looping over time steps, objects, directions, etc.)

Next, we need to add clauses that encode the preconditions and effects of actions. Here is a general method for capturing knowledge about actions. Suppose there are n possible actions in an environment, call them A1,...,An. What we are looking for is a sequence of actions that solves the problem. To represent this in a propositional way, create an expanded set of Boolean variables where each time step is prefixed to each action. For example, if you assume the problem can be solve in t time steps, then let there be nXt variables like this:

T0_A1, T0_A2, ... T0_An
T1_A1, T1_A2, ... T1_An
...

T(t-1)_A1, T(t-1)_A2, ... T(t-1)_An

A variable like T1_A2 means action A2 was done at time step 1.

Only one of these actions can be true at each time step. You will have to add clauses that enforce these constraints (i.e. that *at least* one action variable is satisfied at each time step, and

at most one action variable is satisfied at each time step. Thus most of these action variables will be false in any solution, but there will be one action satisfied for each time step, from which you can read off the "solution" in the form of the sequence of actions that achieves the goal (in the final step). For example, if the satisfied variables are T0_A3=True, T1_A9=True, T2_A4=True, then the plan that solves the problem is the sequence of actions: A3-A9-A4.

For each action, you will then need to write clauses that express the knowledge about the preconditions and effects of that actions. For example, suppose action A requires propositions P and Q to be true, and it results in proposition R being true (e.g. starting a car require that you have the key and are at the car, and results in the car running). You can encode such knowledge by writing clauses saying that the action executed at any time i implies the preconditions hold in time i and the effects hold in the successor time i+1. For example:

$Ti\_A \rightarrow [Ti\_P \wedge Ti\_Q \wedge T(i+1)\_R]$

What these sentences mean is that, if action A occurs at time i, then this implies P and Q must have been true at time i and R must be true at time i+1. More generally:

$Ti\_A \rightarrow [Ti\_precondition\_1 \wedge ... Ti\_precondition\_J \wedge T(i+1)\_effect\_1 \wedge ... T(i+1)\_effect\_K]$

You will have to think about how to use this method to encode all the preconditions and effects of all the action variables in the Farmer-Fox-Chicken-Grain problem. For example, consider a variable like T3_mv_Gr_RL. This has the meaning that in time step 3, the grain was moved from right to left. If this were true (in a model), this would imply that the farmer and the grain must be on the right side in time 3, and the farmer and the grain must be on the left side on in time 4. So you need to generate a bunch of additional clauses that capture these "consistency" constraints (between each time i and i+1) for all the actions, which will ultimately connect the set of satisfied variables between each adjacent time step as a function of the action taken between them.

Finally, you need to add a few more clauses for indirect (or "contingent") effects. For reasons we will discuss later in the semester (in the section on Planning), you also have to include clauses describing all the things that do *not* change with each action. For example, suppose the state of variable S is unaffected by the action A. Thus, if S were True in time i and A was executed at time i, then S would be True in the successor time i+1. And if S were False at i and A was executed, then it would still be False at i+1. Here are the propositional sentences corresponding to this example, which can easily be converted into clauses:

$Ti\_A \wedge Ti\_S \rightarrow T(i+1)\_S$
$Ti\_A \wedge \neg Ti\_S \rightarrow \neg T(i+1)\_S$

An example in this domain would be that the location of the fox does not change if the farmer transports the chicken across the river. You need to capture all such indirect effects at all time-steps by writing more clauses (or generating them with your clause-enumeration script).

In the end you will probably have on the order of a hundred variables and a thousand clauses. Then you add a few facts representing the initial state and the goal. Note that you need to 'guess' the number of time steps required to achieve the goal ahead of time. In this problem, it takes 5 steps just to shuttle the 3 items across the river one-by-one without worrying about the fox eating the chicken or the chicken eating the grain (I suggest you try to solve this simplified problem first, without the additional constraints), and it takes 7 steps to solve the full problem (with all the clauses). So you would add facts like these to all your clauses, and then run DPLL.

```
# init
T0_FaL
T0_FxL
T0_ChL
T0_GrL

# goal
T7_FaR
T7_FxR
T7_ChR
T7_GrR
```

This propositional satisfiability problem is complex enough that it will definitely require the heuristics (Unit Clause and Pure Symbol) to find a model. However, it is surprisingly do-able. My implementation finds a solution after searching 526 nodes of the search space (partial assignments).

What to Turn In (via CSNet):

- your source code
- a short document describing how to compile and run your program
- a transcript of running your program on the abstract Boolean problem (see below) and the Farmer-Fox-Chicken-Grain problem
- 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
```