CS221 Fall 2015 Homework [scheduling]

Problem 0: Warmup

0a.

Let's create variables $X_1, X_2 \;...\; X_m$ where X_j represents whether button j is pressed.

X_j is a boolean variable that is the button state, where $X_j=1$ represents the "pressed" state and $X_j=0$ represents the "not pressed" state.

Let S_i be the set of buttons that control light bulb i , therefore $S_i = \{j: \; i \in T_j\}$

For any light bulb to be turned on, an odd number of buttons that control the light must be in the "pressed" state, since if an even number of buttons are "pressed" the effect will be to toggle the light bulb into an off state.

We will therefore precisely and concisely need one constraint in the CSP per light bulb, for a total of *n* constraints. The constraint for bulb i would be $\bigoplus_{j \in S_i} X_j = 1$

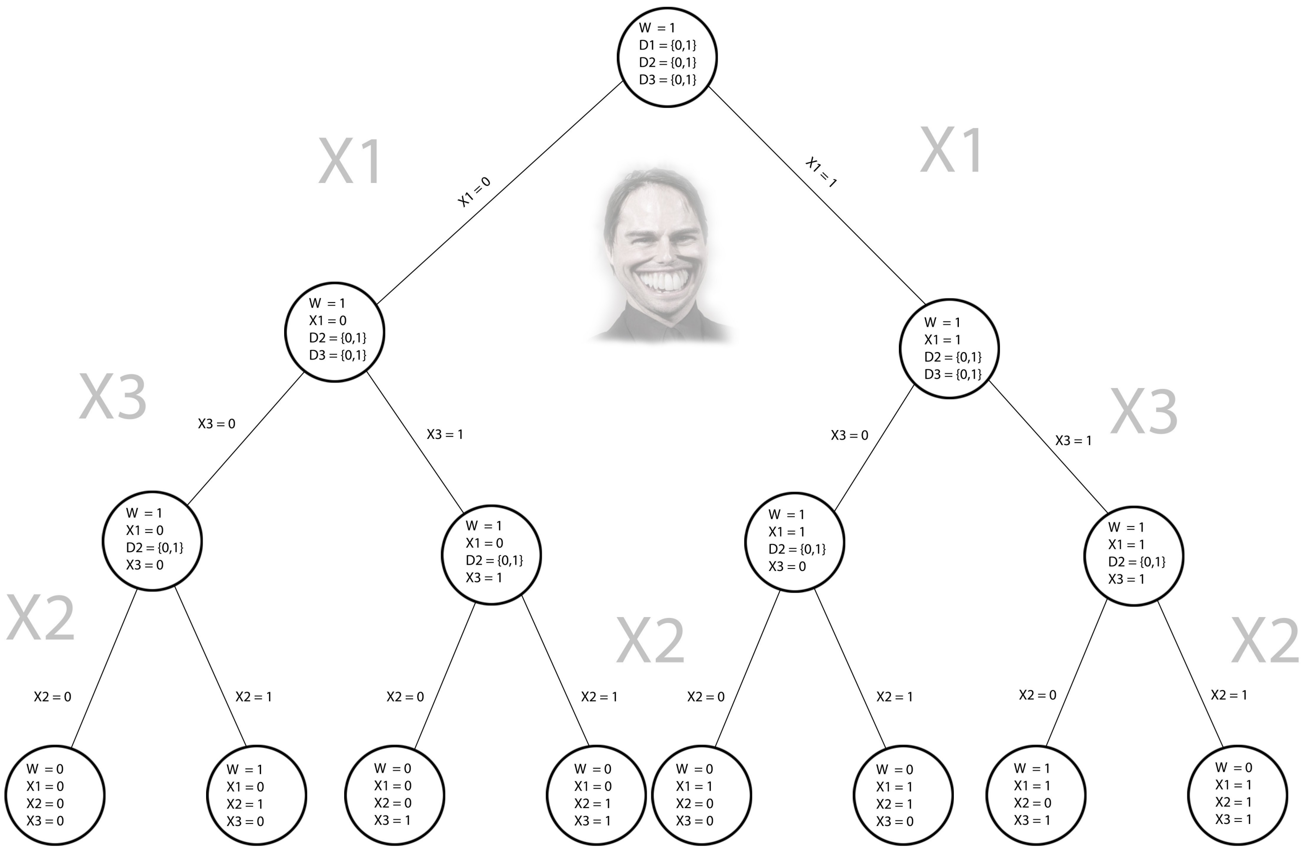
0b.

i. There are two consistent assignments $(X_1=1, \; X_2=0, \; X_3=1)$ and $(X_1=0, \; X_2=1, \; X_3=0)$

ii ...(continued on next page)

0b.ii.

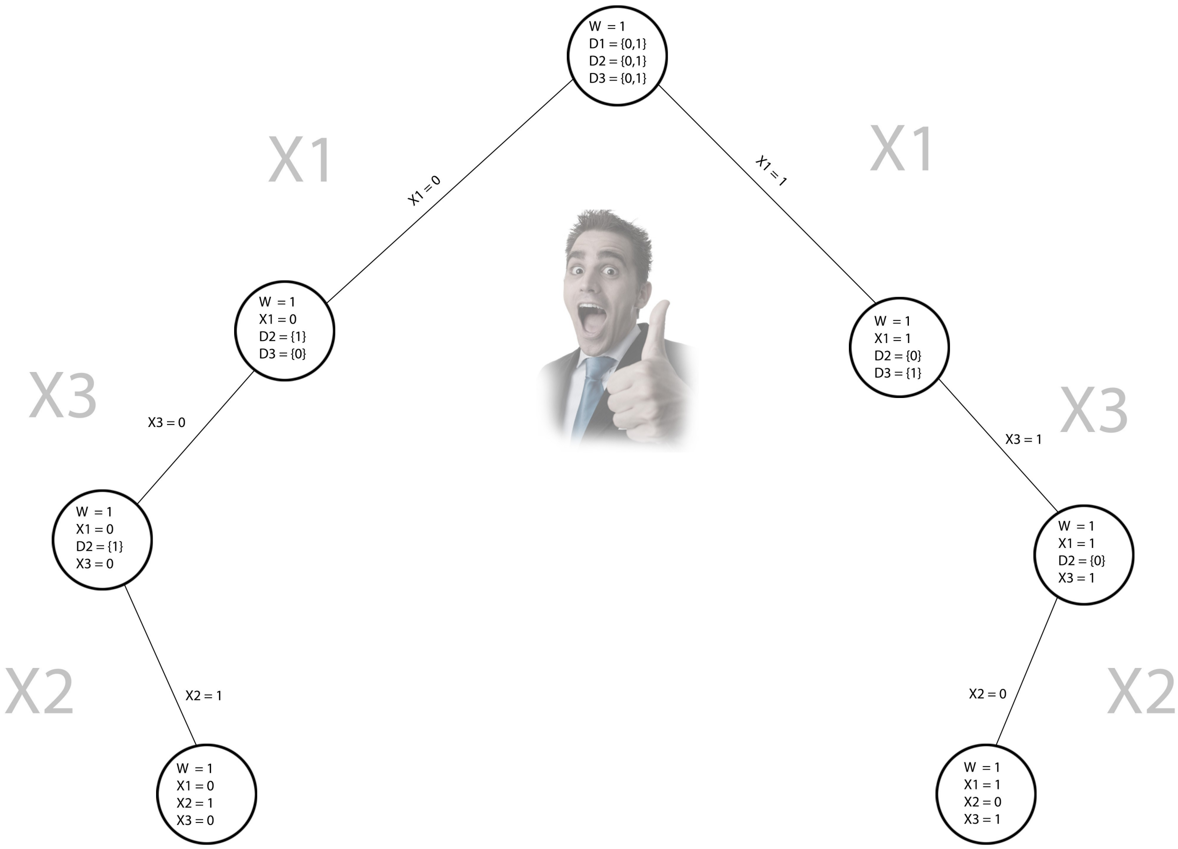
Let D1, D2, and D3 be the running domains of the variables X1, X2, and X3 respectively. In the diagram below, the backtrack() function is called 15 times (once per node).



iii. (continued on next page)

0b.iii.

In the diagram below, the resultant graph using AC-3 is shown; the backtrack() function is called 7 times, again, once for each node, but here, each time we're pruning inconsistent domain values of the unassigned variables using arc consistency.



Problem 2: Handling *n*-ary factors

2a.

We start by introducing the auxiliary variables B_1, B_2, B_3

Each auxiliary variable B_i has two components: $B_i[1]$ and $B_i[2]$ where $B_i[1]$ represents $\sum_{j=1}^{i-1} X_j$ and $B_i[2]$ represents $\sum_{j=1}^{i} X_j$

The domains of X_1, X_2, X_3 are {0, 1, 2}

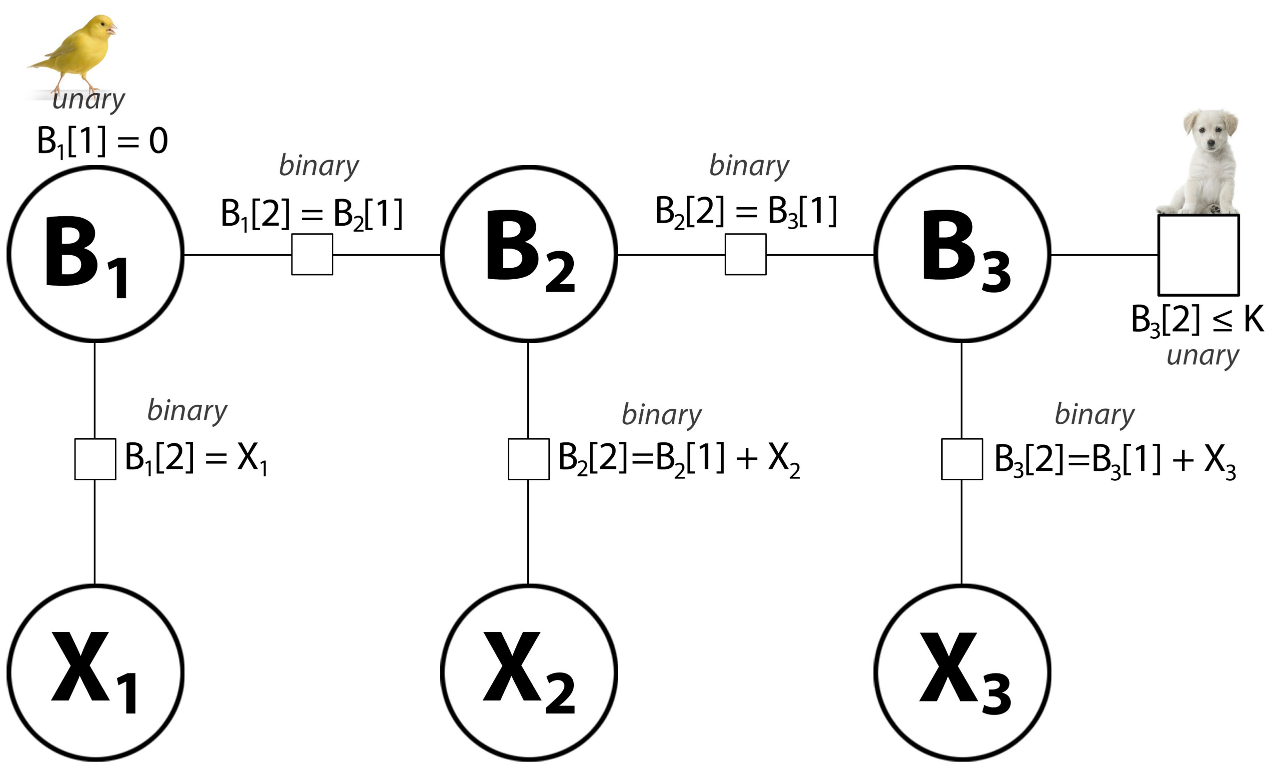
The domain of $B_1 : \{0\} \times \{0,1,2\}$

The domain of $B_2 : \{0,1,2\} \times \{0,1,2,3,4\}$

The domain of $B_3 : \{0,1,2,3,4\} \times \{0,1,2,3,4,5,6\}$

The scheme works because the unary and binary factors as shown in the figure below adhere to the meaning of $B_i[1]$ and $B_i[2]$ Also, $B_3[2]$ represents the sum $X_1 + X_2 + X_3$ which is set to be less than or equal to *K*, as required in the problem definition.

See illustrated reduced CSP in the figure below.



Problem 3: Course Scheduling

3c.

For my simple profile example, I modeled what my possible current and possible schedule for next quarter might be within the CS department had I used this system prior to taking this class, and also if I take another Ai class here at Stanford next quarter, as follows:

```
minUnits 3
maxUnits 12
# These are the quarters that I need to fill. It is assumed that
# the quarters are sorted in chronological order.
register Aut2015
register Win2016
# Courses I've already taken
taken CS124
taken CS107
taken CS103
taken CS106X
taken CS106B
# Courses that I'm requesting
request CS221 in Aut2015
request CS228 in Win2016
```

Further, I ran the CSP solver as instructed and am and printing the data structures for an optimal solution, as follows:

('CS221', 'Aut2015'): 4,
('CS221', 'Win2016'): 0,
('CS228', 'Aut2015'): 0,
('CS228', 'Win2016'): 4,

Using: **import pprint** *(and)* **pprint.pprint(alg.optimalAssignment)** *I printed the above and below internal data of the alg object to explore data generated from the solver. Here is a small sample of some internal data using the Request object being hashed as a key in the data structure after the solver has run and determined optimal assignments:*

(Request{['CS221'] ['Aut2015'] [] 1}, 'Aut2015'): 'CS221',
(Request{['CS221'] ['Aut2015'] [] 1}, 'Win2016'): None,
(Request{['CS228'] ['Win2016'] [] 1}, 'Aut2015'): None,
(Request{['CS228'] ['Win2016'] [] 1}, 'Win2016'): 'CS228',

Cool. This generated schedule might work for me, and now that I have thought about it, and now that I have also run this code, this may turn out to be my actual schedule for next quarter, for one reason in the least: there is certainly, without a doubt much more about Probabilistic Graphical Models to learn in the field of Artificial Intelligence!

Extra Credit: weighted CSPs with notable patterns

The answer is:

1

Since the function ***g*** relates consecutive variables in the order $X_1, X_2, X_3, \ldots, X_n$ it induces constraints that form a chain CSP with a tree-width of one. Recognizing patterns on the same order of the variables can be implemented as chain constraints so that each constraint is from x_i to x_{i+1} also resulting in a tree-width of one... kind of, sort of like a deterministic finite state machine **:D**

...the problem / question doesn't really ask for a super detailed explanation, so hope that works! Thank you in advance for all the points! **:D**

Further as a matter of editorial, I would like to mention my personal opinion that CSPs seem particularly well suited for a variety of elaborate Rube Goldberg machine designs (meant both ironically and literally) and solving CSPs begins to conjure thoughts in my head such as: