Scheduling - Stanford CS 221 Fall 2017-2018

Rohit Apte

2017-11-03

0 Problem 0: Warmup

a Let's create a CSP. Suppose you have n light bulbs, where each light bulb $i = 1, \ldots, n$ is initially off. You also have m buttons which control the lights. For each button $j = 1, \ldots, m$, we know the subset $T_j \subseteq 1, \ldots, n$ of light bulbs that it controls. When button j is pressed, it toggles the state of each light bulb in T_j (For example, if $3 \in T_j$ and light bulb 3 is off, then after the button is pressed, light bulb 3 will be on, and vice versa). Your goal is to turn on all the light bulbs by pressing a subset of the buttons. Construct a CSP to solve this problem. Your CSP should have m variables and n constraints. For this problem only, you can use n-ary constraints. Describe your CSP precisely and concisely. You need to specify the variables with their domain, and the constraints with their scope and expression. Make sure to include T_j in your answer.

A light bulb can be either off (0) or on (1). So if we have 2 bulbs the possible states are $(0, 0), (0, 1), (1, 0), (1, 1)$.

Variables: We define the variable X_i to be one of m switches. The values of X_j ($j = 1, \cdots, m$) are a tuple of size n where each column is a 1 if that switch toggles the bulb and 0 otherwise. Formally
$X_j = (x_1, x_2, \ldots, x_n)$ where $x_i =$
$$\begin{cases} 1 \text{ if } x_i \subseteq T_j \\ 0 \text{ otherwise} \end{cases}$$

Constraints: We have n constraint where constraint(i) implies that the sum of column i for all our variables X_j should be odd (i.e. the sum of columns mod $2 = 1$). We will have n constraints since our tuple is size n.

b Let's consider a simple CSP with 3 variables and 2 binary factors: where $X_1, X_2, X_3 \in \{0, 1\}$ and t_1, t_2 are XOR functions (that is $t_1(X) = x_1 \oplus x_2$ and $t_2(X) = x_2 \oplus x_3$).

b.i How many consistent assignments are there for this CSP?

There are 2 consistent assignments. See table below.

X_1	X_2	X_3	t_1	t_2	$Consistent$
0	0	0	0	0	No
0	0	1	0	1	No
0	1	0	1	1	Yes
0	1	1	1	0	No
1	0	0	1	0	No
1	0	1	1	1	Yes
1	1	0	0	1	No
1	1	1	0	0	No

b.ii To see why variable ordering is important, let's use backtracking search to solve the CSP without using any heuristics (MCV, LCV, AC-3) or lookahead. How many times will backtrack() be called to get all consistent assignments if we use the fixed ordering X_1, X_3, X_2? Draw the call stack for backtrack(). (You should use the Backtrack algorithm from the slides. The initial arguments are x=$\emptyset$, $w = 1$, and the original Domain.)

In the code, this will be BacktrackingSearch.numOperations.

Backtrack will be called 9 times (including the initial call). We will initially call backtrack on $\emptyset$ with $w = 1$. The call stack is as follows

```
Backtrack({∅}, 1, {0, 1})
    X_1 = 0
    δ = 1
    Backtrack({X_1 = 0}, 1, {0, 1})
        X_3 = 0
        δ = 1
        Backtrack({X_1 = 0, X_3 = 0}, 1, {0, 1})
            X_2 = 0
            δ = 0
            continue
            X_2 = 1
            δ = 1
            Backtrack({X_1 = 0, X_3 = 0, X_2 = 1}, 1, {0, 1})
                COMPLETE ASSIGNMENT. Update best and return
        X_3 = 1
        δ = 1
        Backtrack({X_1 = 0, X_3 = 1}, 1, {0, 1})
            X_2 = 0
            δ = 0
            continue
            X_2 = 1
            δ = 0
            continue
    X_1 = 1
```

```
δ = 1
Backtrack({X1 = 1}, 1, {0, 1})
    X3 = 0
    δ = 1
    Backtrack({X1 = 1, X3 = 0}, 1, {0, 1})
        X2 = 0
        δ = 0
        continue
        X2 = 1
        δ = 0
        continue
    X3 = 1
    δ = 1
    Backtrack({X1 = 1, X3 = 1}, 1, {0, 1})
        X2 = 0
        δ = 0
        Backtrack({X1 = 1, X3 = 1, X2 = 0}, 1, {0, 1})
            COMPLETE ASSIGNMENT. Update best and return
        X2 = 1
        δ = 0
        continue
```

b.iii To see why lookahead can be useful, let's do it again with the ordering X_1, X_3, X_2 and AC-3. How many times will Backtrack be called to get all consistent assignments? Draw the call stack for backtrack()

Backtrack will be called 7 times including the initial call. The call stack is as follows

```
Backtrack({∅}, 1, {0, 1})
    X1 = 0
    δ = 1
    Backtrack({X1 = 0}, 1, {0, 1})
    Enforce arc consistency on neighbors
        Add X1 = 0 to set
        While Set is nonempty
            Remove X1 = 0 from set
            Enforce arc consistency on X2.  Domain of X2 = {1}
            Domain changed so add X2 = 1 to set
            Remove X2 = 1 from set.
            Enforce arc consistency on X3.  Domain of X3 = {0}
            Domain changed so add X3 = 0 to set
            Remove X3 = 0 from set.
            Set is empty so return
```

$X_3 = 0$
$\delta = 1$
Backtrack($\{X_1 = 0, X_3 = 0\}, 1, \{0, 1\}$)
    $X_2 = 1$
    $\delta = 1$
    Backtrack($\{X_1 = 0, X_3 = 0, X_2 = 1\}, 1, \{0, 1\}$)
        COMPLETE ASSIGNMENT. Update best and return

$X_1 = 1$
$\delta = 1$
Backtrack($\{X_1 = 1\}, 1, \{0, 1\}$)
Enforce arc consistency on neighbors
    Add $X_1 = 1$ to set
    While Set is nonempty
        Remove $X_1 = 1$ from set
        Enforce arc consistency on X_2. Domain of $X_2 = \{0\}$
        Domain changed so add $X_2 = 0$ to set
        Remove $X2 = 0$ from set.
        Enforce arc consistency on X_3. Domain of $X_3 = \{1\}$
        Domain changed so add $X_3 = 1$ to set
        Remove $X_3 = 1$ from set.
        Set is empty so return
    $X_3 = 1$
    $\delta = 1$
    Backtrack($\{X_1 = 1, X_3 = 1\}, 1, \{0, 1\}$)
        $X_2 = 0$
        $\delta = 0$
        Backtrack($\{X_1 = 1, X_3 = 1, X_2 = 0\}, 1, \{0, 1\}$)
            COMPLETE ASSIGNMENT. Update best and return

1 Problem 1: CSP Solving

2 Problem 2: Handling n-ary factors

So far, our CSP solver only handles unary and binary factors, but for course scheduling (and really any non-trivial application), we would like to define factors that involve more than two variables. It would be nice if we could have a general way of reducing n-ary constraint to unary and binary constraints. In this problem, we will do exactly that for two types of n-ary constraints.

Suppose we have boolean variables X_1, X_2, X_3, where X_i represents whether the i-th course is taken. Suppose we want to enforce the constraint that $Y = X_1 \vee X_2 \vee X_3$, that is, Y is a boolean representing whether at least one course has been taken. For reference, in util.py, the function get_or_variable() does such a reduction. It takes in a list of variables and a target value, and returns a boolean variable with domain [True, False] whose value is constrained to the condition of having at least one of the variables assigned to the target value. For example, we would call get_or_variable() with arguments $(X_1, X_2, X_3, True)$, which would return a new (auxiliary) variable X_4, and then add another constraint $[X_4 = True]$. The second type of n-ary factors is constraints on the sum over n variables. You are going to implement reduction of this type but let's first look at a simpler problem to get started:

a Suppose we have a CSP with three variables X_1, X_2, X_3 with the same domain $\{0, 1, 2\}$ and a ternary constraint $[X_1 + X_2 + X_3 \leq K]$. How can we reduce this CSP to one with only unary and/or binary constraints? Explain what auxiliary variables we need to introduce, what their domains are, what unary/binary factors you'll add, and why your scheme works. Add a graph if you think that'll better explain your scheme.
Hint: draw inspiration from the example of enforcing $[X_i = 1$ for exactly one $i]$ which is in the first CSP lecture.

We know that X_i has domain $\{0, 1, 2\}$. Let us define a new variable A_i as follows
Factors
Initialization: $A_0 = 0$
Processing: $A_i = A_{i-1} + X_i$
Final output: $A_4 \leq K$
We now need to pack A_{i-1} and A_i into one variable B_i where B_i represents the tuple (A_{i-1}, A_i).
Factors
Initialization: $B_1[1] = 0$
Processing: $B_i[2] = B_i[1] + X_i$
Final output: $B_4[2] \leq K$
Consistency: $B_{i-1}[2] = B_i[1]$

3 Problem 3: Course Scheduling

a see submission.py

b see submission.py

c Now try to use the course scheduler for the winter and spring (and next year if applicable). Create your own profile.txt and then run the course scheduler: python run_p3.py profile.txt You might want to turn on the appropriate heuristic flags to speed up the computation. Does it produce a reasonable course schedule? Please include your profile.txt and the best schedule in your writeup; we're curious how it worked out for you!

I ran mine on a simple premise - I am an SCPD student and havent done a college course since I graduated in 2001. I wasnted to see if I could take a 6 month sabbatical, how many courses I could take in as a part time student at Stanford. This is my profile file

```
# Unit limit per quarter.
minUnits 6
maxUnits 12

# These are the quarters that I need to fill.  It is assumed that
# the quarters are sorted in chronological order.
register Win2017
register Spr2018

# Courses I've already taken
taken CS221
taken MATH51
taken CS106B
taken CS124
taken CS107
taken CS103

# Courses that I'm requesting
request CS223A
request CS347
request CS341
request CS110
request CS802
request CS801
request CS227B
request CS1U
```

```
request CS1C
request CS248 in Win2017
```

This is the schedule I get. Its reasonable.

```
WARNING: missing prerequisite of CS248 -- CS148:  Introduction to Computer
Graphics and Imaging; you should add it as 'taken' or 'request'
Found 1696 optimal assignments with weight 1.000000 in 124295 operations
First assignment took 4719 operations
Here's the best schedule:
```

Quarter Units Course
Win2017 5 CS110
Win2017 4 CS248
Spr2018 3 CS341
Spr2018 3 CS227B