CS 221 – Autumn 2018
Anand Natu
anatu - 06264867

Homework #6: Course Scheduling

Problem 0

Problem 0a

Variables and Constraints: This search problem has m variables and n constraints, corresponding to the m buttons and n lightbulbs. Therefore, the buttons are the variables and the lightbulbs are the constraints:

$$X = \{X_1, X_2, \dots X_m\}$$

$$f = \{f_1, f_2, \dots f_n\}$$

Domain of Variables: The domain of each variable is simply whether or not the button is pressed. Therefore, the domain is simply whether the button has been pressed or not. Note that this assumes that a solution can be reached for this CSP by pressing each button a maximum of once (which can be proven):

$$D(X) = \{0,1\}$$

Expression of Constraints: Because each button is only pressed a maximum of one time, constraint satisfaction requires that at least two of the buttons that control a particular bulb differ in their assignment, and in the case where only one button controls it, that it be pressed at all. This yields

$$for\ f_i, let\ \alpha_i = \{X_j \mid i \in T_j\}$$

$$f_i = XOR[\alpha_i]$$

Scope of Constraints: Because we have n bulbs and we assume each bulb is controlled by at least one button, this yields n constraints, one for each button, where the scope of the constraint consists of each of the lightbulbs that are controlled by that button:

$$Scope(f_j) = T_j$$

Problem 0b

i) Since the XOR function returns True only when the inputs differ, consistent assignments will happen when $X_1 \neq X_2$ and $X_2 \neq X_3$ for $X \in \{0,1\}$. This happens for two possible assignments, which are $X = \{X_1, X_2, X_3\} = \{1,0,1\}\ or\ \{0,1,0\}$.

ii) The call stack for the CSP using the given variable ordering is depicted in the figure below. Note that the domains of the variables remain constant because we use no lookahead. Based on the below, it takes 5 calls of the backtrack() function before both of the consistent assignments are found.

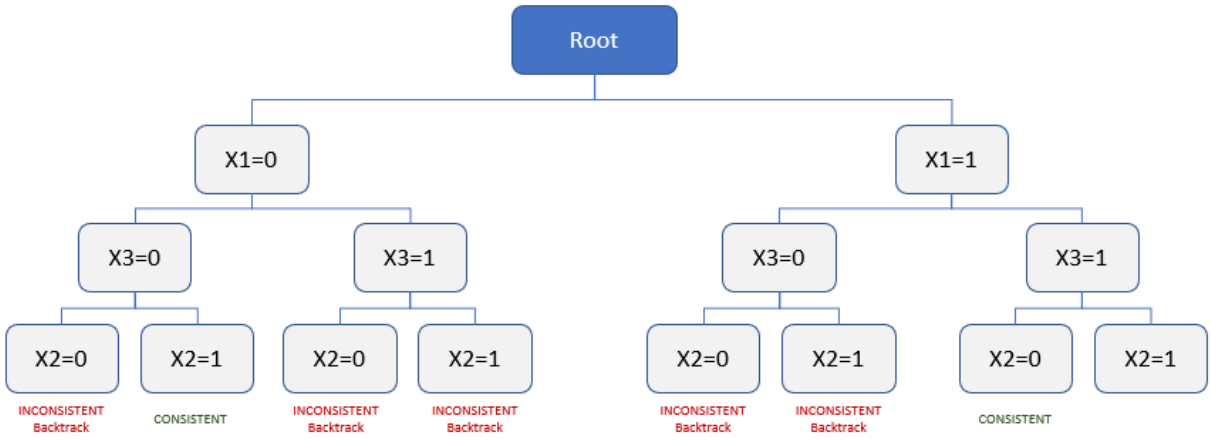

Figure 1: Call stack for backtracking search without lookahead

iii) The call stack for this CSP using AC-3 lookahead is shown in the figure below. As can be shown in the figure below, only 3 calls of the backtrack() function are required here. Two of the inconsistent cases are pruned in advance, since the AC-3 lookahead on variable X_3 enforces arc-consistency on its neighbor X_2 and prunes the assignments which violate the constraint $t_2(X) = X_1 \oplus X_2$.

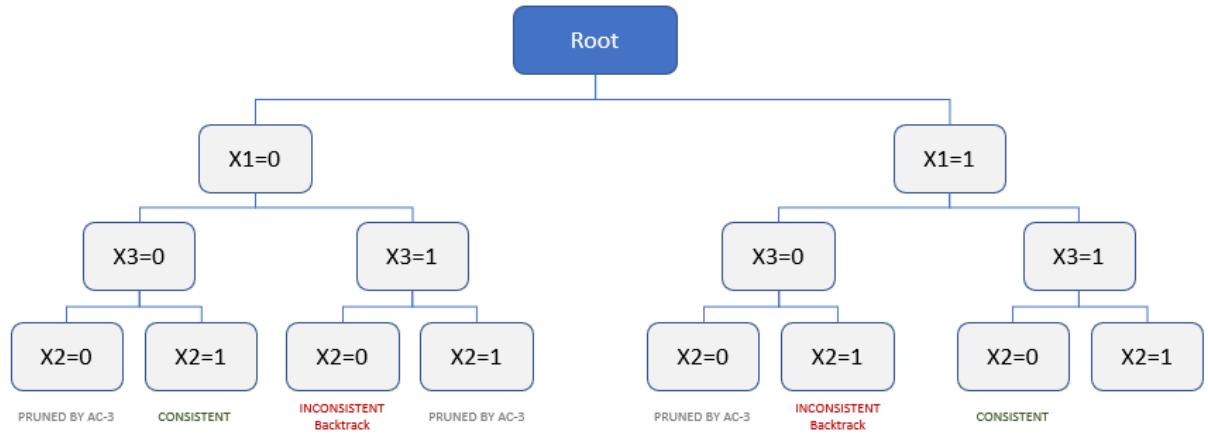

Figure 2: Call stack using AC-3 lookahead

Problem 2

Problem 2a

In order to reduce the CSP to only unary / binary constraints, we will introduce an auxiliary variable A which will store the intermediate result of the computation of the sum $X_1 + X_2 + X_3$.

CS 221 – Autumn 2018
Anand Natu
anatu - 06264867

The set of auxiliary variables A is defined for all $X_i, i \in \{1,2,3 \dots T\}$ where each variable A_i will store two values, the cumulative sum before and after incremental addition of that given X_i. Generally, this gives the following representation of the auxiliary variable and its factors:

- **Initialization:** $A_0[0] = 0$
- **Processing:** $A_i[1] = A_i[0] + X_i$
- **Intermediate Consistency:** $A_{i-1}[1] = A_i[0]$
- **Output Consistency:** $A_T = \sum_{i=1}^{T} X_i$
- **Output Constraint:** $A_T \leq K$
- **Domain:** $Domain(A) = (x_i, y_i) \ \ for\ x, y \in \{1,2 \dots maxSum\}$.

The diagram below shows a graphical representation of the modified CSP making use of this new auxiliary variable. This is as compared to the original state, where a single N-ary constraint depends on all variables X in the CSP.

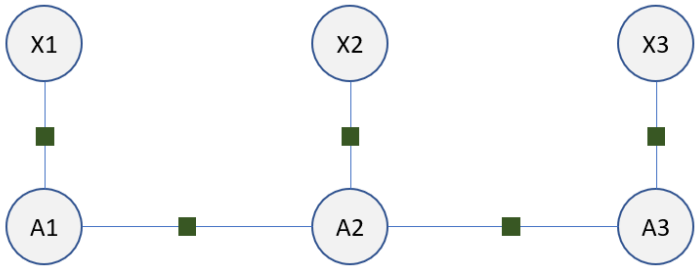

Figure 3: Graph representation of CSP with auxiliary variables and unary/binary constraints

Problem 3

Problem 3c

The profile that I specified for this problem was as follows:

```
# Unit limit per quarter
minUnits 1
maxUnits 5

# Quarters
register Aut2015
register Win2016
register Spr2016

# Courses taken
taken MATH51

# Courses requested
```

CS 221 – Autumn 2018
Anand Natu
anatu - 06264867

```
request CS106B
request CS107
request CS109
request CS140
request CS229
```

Which produced the following schedule:

Quarter	Units	Course
Aut2015	4	CS229
Win2016	4	CS140
Spr2016	5	CS106B

Given the constraints outlined in the profile, this appears to be a reasonable schedule assignment.

Problem 4

Problem 4a

In the worst case, we know that the longest possible notable pattern has a length of n. Because the pattern must in this case be checked over all of the variables, it means that the corresponding factor also has an arity of n. **Therefore, in the worst case (where we perform no elimination), the tree-width is n corresponding to the n-ary constraint imposed by a notable pattern of length n.**

Problem 4b

Generally, the complexity of the naïve method of string searching for a pattern of length m within a string of length n is $O_{naive} = f(nm)$. In order to reduce the complexity, we will perform preprocessing by creating a "prefix" function which encodes the pattern information into a form which makes the pattern matching against the actual set of variables faster.

Prefix Function: The prefix function works by creating a tree of the variables which collectively form all of the patterns in the set. This provides the advantage of letting us search for patterns in parallel, instead of sequentially as in the naïve approach. For example, for the pattern set {ab, aac, b, ba, bb}, the general tree representation is depicted below.

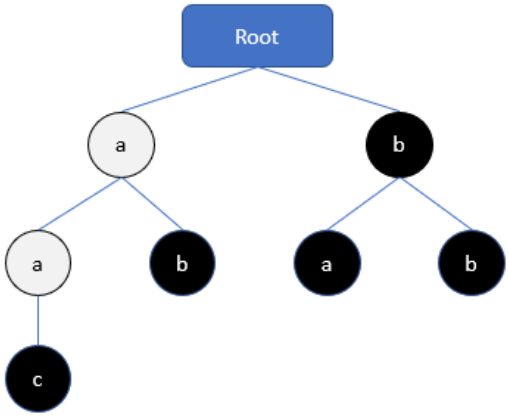

Figure 4: Tree representation of Prefix function. Black nodes denote matched patterns (i.e. current node + ancestors = pattern)

Failure Pointers: Next, we must augment our tree with "failure pointers". These are pointers which allow the algorithm to most quickly find the next feasible candidate pattern in the event that the current partial pattern does not complete. For example, in the above, if we have the string "aaba", upon reaching the "b" we want to most quickly get to a node where a potential pattern may still exist. To do this, every node points to the closest ancestor of the same value relative to the root node, as is shown in the diagram below for the same example {ab, aac, b, ba, bb}.

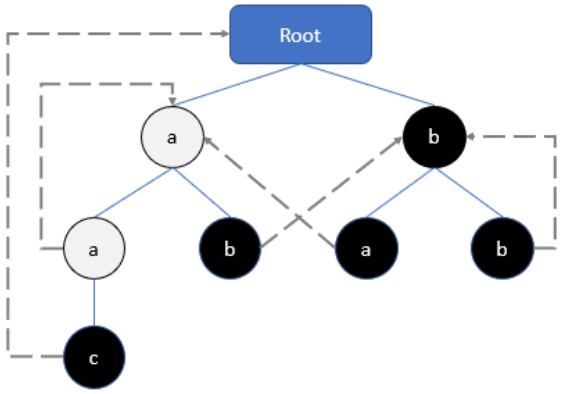

Figure 5: Prefix tree with failure pointers for every node

Complexity: If we denote n to be the length of the text, we see that the worst-case construction of this tree involves iterating through a pattern that is ostensibly the same length as the text, which means the complexity of prefix tree generation is $O_{prefix} = f(n)$.

For the actual matching step, if we denote the total length of the pattern vocabulary as m, and the number of matches found in the text as z, we find that the complexity of the matching step is $O_{match} = f(m + z)$, since in the worst-case the matching will have to go through the entire prefix tree of m nodes, and report z matches resulting from the search. Since the prefix tree construction and matching operations take place in series, it follows that $\boldsymbol{O_{tot} = O_{prefix} + O_{match} = f(n + m + z)}$.