CS221 Fall 2018 Homework 6

SUNet ID: 05794739
Name: Luis Perez
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Problem 0

(a) Constructing the CSP from this problem statement is somewhat straight-forward. We present the construct CSP below. We assume that each lighbulb is initially off.

- Variables: For each button $j = 1, \cdots, m$, we have a variable $B_i \in \{0, 1\}$. We explicitly mention now that the Domain of each variables is $\{0, 1\}$. Intuitively, each variable simply represents whether or not button j is pressed.
- Constraints: Lightbulbs are on only if toggled an odd number of times by the buttons selected. We set-up n constraints (1-per lightbulb).
  - Constraint Scope: For each lightbulb i, we define the set $M_i = \{j \mid i \in T_j\}$ (this is the set of buttons that toggle lighbulb i) and the scope $\text{Scope}(i) = \{B_k \mid k \in M_i\}$.
  - Constraint Expression: We then have the constraint expression given by $f_i(\text{Scope}(i)) = (\sum_k B_k) \mod 2$.

  The above enforce the constraint that for each lighbulb, an odd number of toggles must have been triggered given the configuration (since the lighbulbs start off, and odd number of toggles will leave them in the on position).

The above Variables and Constraints are sufficient to solve the problem. A solution exists if and only if the weight is 1. Given our definition of constraints above, this implies that $\forall i,\ f_i(b) = 1$, which means that our selection of touched buttons must lead to all lighbulbs being turned on.

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

  ii 'backtrack' will be called 9 times (including initial call) if we use the fixed ordering specified in the problem statement. The resulting call-stack (in tree form) is presented in Figure 1.

  iii 'backtrack' will be called 7 times (including initial call) if we use the fixed ordering specified in the problem statement with AC-3. The resulting call-stack (in tree form) is presented in Figure 2.

(c) In "submission.py".

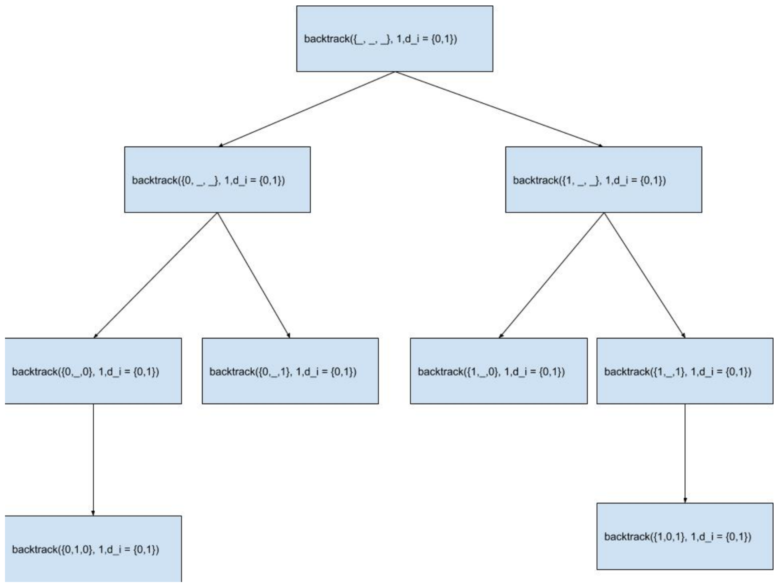
backtrack({_, _, _}, 1,d_i = {0,1})
backtrack({0, _, _}, 1,d_i = {0,1})
backtrack({1, _, _}, 1,d_i = {0,1})
backtrack({0,_,0}, 1,d_i = {0,1})
backtrack({0,_,1}, 1,d_i = {0,1})
backtrack({1,_,0}, 1,d_i = {0,1})
backtrack({1,_,1}, 1,d_i = {0,1})
backtrack({0,1,0}, 1,d_i = {0,1})
backtrack({1,0,1}, 1,d_i = {0,1})

Figure 1: Backtrack using domain ordering $0, 1$ and variable ordering X_1, X_3, X_2

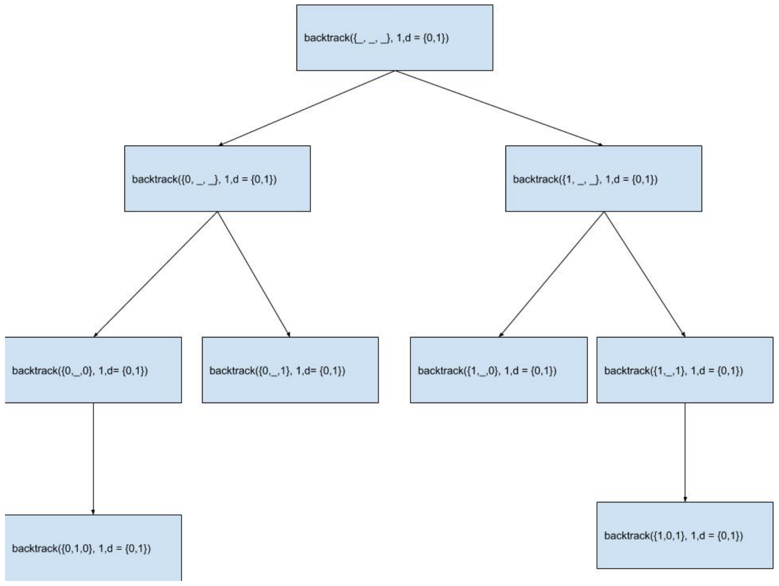
backtrack({_, _, _}, 1,d = {0,1})
backtrack({0, _, _}, 1,d = {0,1})
backtrack({1, _, _}, 1,d = {0,1})
backtrack({0,_,0}, 1,d= {0,1})
backtrack({0,_,1}, 1,d= {0,1})
backtrack({1,_,0}, 1,d = {0,1})
backtrack({1,_,1}, 1,d = {0,1})
backtrack({0,1,0}, 1,d = {0,1})
backtrack({1,0,1}, 1,d = {0,1})

Figure 2: Backtrack using domain ordering $0, 1$ and variable ordering X_1, X_3, X_2 with AC-3

Problem 1

(a) In “submission.py”.

(b) In “submission.py”.

(c) In “submission.py”.

Problem 2

(a) As the hint suggest, we draw inspiration from the lecture slides. We therefore introduce the auxilirary variables S_i for $1 \leq i \leq 3$ consisting of a tuple. The first element of this tuple will be equal the sum of the elements $X_1, \cdots, X_{i-1}$, and the second element of this tuple will be equal to the sum of variables $X_1, \cdots X_i$. For simplicity, we can make all these variables have the same domain which is simply given by: $\{0,1,2,3,4,5,6\} \times \{0,1,2,3,4,5,6\}$.

More generally, the straight-forward domain is given by the below, where we assume that $\forall i, x \in \text{Domain}(X_i), x \geq 0$:

$$\text{Domain}(S_i) = \{0, 1, \cdots, m\} \times \{0, 1, \cdots m\}$$

where $m = \sum_{i=1}^{n} \max\{\text{Domain}(X_i)\}$. A simple modification to restrict our domain a bit further is to make m depend on i such that $m_i = \sum_{j=1}^{i} \max\{\text{Domain}(X_j)\}$ and $m_1 = 0$. In that case, or domain is given by:

$$\text{Domain}(S_i) = \{0, 1, \cdots, m_{i-1}\} \times \{0, 1, \cdots m_i\}$$

If we care to, we can restrict the domain even further, to the below:

$$\begin{aligned} \text{Domain}(S_1) &= \{0\} \times \{\text{Domain}(X_1)\} \\ \text{Domain}(S_i) &= \{(s_1, s_2) \mid (\_, s_1) \in \text{Domain}(S_{i-1}), s_2 \in \{s_1 + v \mid v \in \text{Domain}(X_i)\}\} \end{aligned}$$

Regardless of which domain we pick, we now describe the factors that we must introduce.

- Initializations: $\mathbb{1}[S_1[1] = 0]$. This enforces that we start our sum at 0.
- Processing: $\forall i, \mathbb{1}[S_i[2] = S_i[1] + X_i]$. This enforces that we are computing the sum.
- Final outputs: $\mathbb{1}[S_n[2] \leq k]$. This is enforcing our initial constraint from the problem that the sum must be less than our equal to k.
- Consistency: $\forall i, \mathbb{1}[B_i[1] = B_{i-1}[2]]$. This enforces that our computed sums are transferred correctly between tuples.

The above constraints, along with a concrete example, can the nicely summarized in Figure 3. It is clear that the above scheme will work, assigning only valid values to X_i.

(b) In "submission.py".

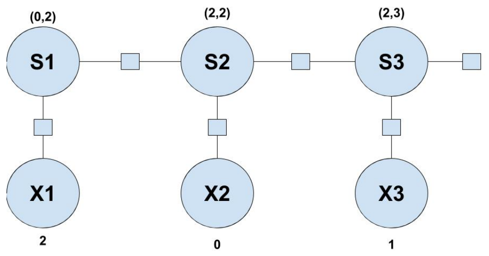

Figure 3: Reducing $n - ary$ constraint to binary and unary constraints.