Homework 6

Artificial Intelligence
Fall 2021 CS47100-AI

Student name: ——————————————— Student PUID: ——————————
Note: You are free to use your intuition to find the steps in the proof. But, make sure you do not use your intuition to justify steps in your proofs. [1]

Problem 1. 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: constraints that can be functions of up to n variables. 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.

Solution:

Problem 2. Now, let's consider a simple CSP with 3 variables and 2 binary factors:

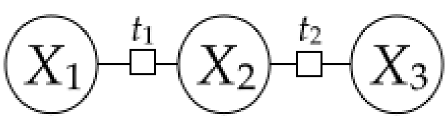

Figure 1: CSP

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 $t_2(X) = x_2 \oplus x_3$).
Now answer the following subquestions:

i What are the consistent assignments for this CSP?

ii Let's use backtracking search to solve the CSP without using any heuristics (MCV, LCV, AC-3).

In this problem, we will ask you to produce the call stack for a specific call to Backtrack(). A call stack is just a diagram tracing out every recursive call. For our purposes, for each call to Backtrack() you should specify which variable is being assigned, the current domains, and which parent call to Backtrack() it's called within. For example, if the order in which we assign variables is X_1, X_2, X_3, the call stack would be as follows:

[1] *The contents of this problem set is based on the AI course CS221 taught at Stanford University.*

$$\{[0,1],[0,1],[0,1]\} \xrightarrow{X_1=0} \{\mathbf{0},[0,1],[0,1]\} \xrightarrow{X_2=1} \{\mathbf{0},\mathbf{1},[0,1]\} \xrightarrow{X_3=0} \{\mathbf{0},\mathbf{1},\mathbf{0}\}$$
$$\xrightarrow{X_1=1} \{\mathbf{1},[0,1],[0,1]\} \xrightarrow{X_2=0} \{\mathbf{1},\mathbf{0},[0,1]\} \xrightarrow{X_3=1} \{\mathbf{1},\mathbf{0},\mathbf{1}\}$$

The notation $\mathbf{1},[0,1],[0,1]$ means that X_1 has been assigned value 1, while X_2 and X_3 are currently unassigned and each have domain $[0,1]$. We avoid the weight variable for simplicity; the only possible weights for this problem are 0 and 1. In this case, backtrack is called 7 times. Notice that Backtrack() is not called when there's an inconsistent partial assignment ($\delta = 0$); for example, we don't call Backtrack() on $X_2 = 1$ when X1 is already set to 1.

Draw out the call-stack if we instead assign variables in the order X_1, X_3, X_2. How many calls do we make to Backtrack()? Why can this number change depending on the ordering?

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()

Solution:

We'll now pivot towards creating more complicated CSPs, and solving them faster using heuristics. Notice we are already able to solve the CSPs because in submission.py, a basic backtracking search is already implemented. For this problem, we will work with unweighted CSPs that can only have True/False factors; a factor outputs 1 if a constraint is satisfied and 0 otherwise. The backtracking search operates over partial assignments, and specifies whether or not the current assignment satisfies all relevant constraints. When we assign a value to a new variable X_i, we check that all constraints that depend only on X_i and the previously assigned variables are satisfied. The function satisfies_constraints() returns whether or not these new factors are satisfied based on the unaryFactors and binaryFactors. When satisfies_constraints() returns False, any full assignment that extends the new partial assignment cannot satisfy all of the constraints, so there is no need to search further with that new partial assignment.

Take a look at BacktrackingSearch.reset_results() to see the other fields which are set as a result of solving the weighted CSP. You should read BacktrackingSearch class carefully to make sure that you understand how the backtracking search is working on the CSP.

Now, answer the following 3 questions

Problem 3. Let's create an unweighted CSP to solve the `n-queens` problem: Given an $n \times n$ board, we'd like to place n queens on this board such that no two queens are on the same row, column, or diagonal. Implement create_nqueens_csp() by adding n variables and some number of binary factors. Note that the solver collects some basic statistics on the performance of the algorithm. You should take advantage of these statistics for debugging and analysis. You should get 92 (optimal) assignments for $n = 8$ with exactly 2057 operations (number of calls to backtrack()).

Hint: If you get a larger number of operations, make sure your CSP is minimal. Try to define the variables such that the size of domain is $O(n)$.

Note: Please implement the domain of variables as 'list' type in Python (you can refer to create_map_coloring_csp() and create_weighted_csp() in util.py as examples of CSP problem implementations), so you can compare the number of operations with our suggestions as a way of debugging.

Solution:

Problem 4. You might notice that our search algorithm explores quite a large number of states even for the 8×8 board. Let's see if we can do better. One heuristic we is using most constrained variable (MCV): To choose an unassigned variable, pick the X_j that has the fewest number of values a which are consistent with the current partial assignment (a for which satisfies_constraints() on $X_j = a$ returns True). Implement this heuristic in get_unassigned_variable() under the condition self.mcv = True. It should take you exactly 1361 operations to find all optimal assignments for 8 queens CSP — that's 30% fewer!

Some useful fields:

In BacktrackingSearch, if var has been assigned a value, you can retrieve it using assignment[var]. Otherwise var is not in assignment.

Solution:

Problem 5. The previous heuristics looked only at the local effects of a variable or value. Let's now implement arc consistency (AC-3). After we set variable X_j to value a, we remove the values b of all neighboring variables X_k that could cause arc-inconsistencies. If X_k's domain has changed, we use X_k's domain to remove values from the domains of its neighboring variables. This is repeated until no domain can be updated. Note that this may significantly reduce your branching factor, although at some cost. In backtrack() we've implemented code which copies and restores domains for you. Your job is to fill in arc_consistency_check().

With AC-3 enabled, it should take you 769 operations only to find all optimal assignments to 8 queens CSP — That is almost 45% fewer even compared with MCV!

Solution: