Assignment 6. Constraint Satisfaction Problems

Hwanjo Yu
CSED342 - Artificial Intelligence

Contact: TA Seungha Hong (shhong97@postech.ac.kr)
Deadline: May 23rd 2023 at 2:00 pm. (50% penalty for every 1 day)

General Instructions

⌨This icon means you should write code. You can add other helper functions outside the answer block if you want. Do not make changes to files other than `submission.py`.

Please use **Python 3.9** to develop your code. You have to write answers in the same format as provided in `submission.py`.

You should modify the code in `submission.py` between

```
# BEGIN_YOUR_ANSWER
```

and

```
# END_YOUR_ANSWER
```

Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in `grader.py`, but the correct outputs are not. To run all the tests, type

```
python grader.py
```

This will tell you only whether you passed the basic tests. On the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. You can also run a single test (e.g., `3a-0-basic`) by typing

```
python grader.py 3a-0-basic
```

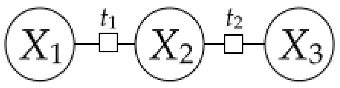
We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run `grader.py`.

In this assignment, you will formulate some problems as constraint satisfaction problems (CSPs) which consist of variables and unary or binary factors between the variables. Once you defined CSPs, you can find the solutions by backtracking search. However, its run-time increases exponentially with the number of variables and factors. Therefore, you will also implement some heuristics which reduce the required time for backtracking.

Problem 0. Warm-up

Problem 0a [3 points] ⌨

Let's consider a CSP with n variables $X_1, ..., X_n$ and $n-1$ binary factors $t_1, ..., t_{n-1}$ where $X_i \in \{0,1\}$ and $t_i(X) = x_i \bigoplus x_{i+1}$. Note that the CSP has a chain structure. The figure below illustrates an example of the factor graph with 3 variables.



Implement `create_chain_csp()` by creating a generic chain CSP with XOR as factors.

Note: We've provided you with a CSP implementation in `util.py` which supports unary and binary factors. For now, you don't need to understand the implementation, but please read the comments and get yourself familiar with the CSP interface. For this problem, you'll need to use `CSP.add_variable()` and `CSP.add_binary_factor()`.

Problem 1: CSP solving

So far, we've only worked with unweighted CSPs, where $f_j(x) \in \{0,1\}$. In this problem, we will work with weighted CSPs, which associates a weight for each assignment x based on the product of m factor functions $f_1, \ldots, f_m$:

$$\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)$$

where each factor $f_j(x) \geq 0$. Our goal is to find the assignment(s) x with the highest weight. As in problem 0, we will assume that each factor is either a unary factor (depends on exactly one variable) or a binary factor (depends on exactly two variables).

For weighted CSP construction, you can refer to the CSP examples we provided in `util.py` for guidance (`create_map_coloring_csp()`, `create_weighted_csp()` and `create_or_csp()`). You can try these examples out by running

```
python run_p1.py
```

Notice we are already able to solve the CSPs, because in `submission.py`, a basic backtracking search is already implemented. Recall that backtracking search operates over partial assignments and associates each partial assignment with a weight, which is the product of all the factors that depend only on the assigned variables. When we assign a value to a new variable X_i, we multiply in all the factors that depend only on X_i and the previously assigned variables. The function `get_delta_weight()` returns the contribution of these new factors based on the unaryFactors and binaryFactors. An important case is when `get_delta_weight()` returns 0. In this case, any full assignment that extends the new partial assignment will also be zero, 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 `submission.BacktrackingSearch` carefully to make sure that you understand how the backtracking search is working on the CSP.

Problem 1a [4 points]

Let's create a 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 $O(n)$.

Problem 1b [4 points]

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 discussed in class 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 `get_delta_weight()` on $X_j = a$ returns a non-zero value). 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:

- `csp.unaryFactors[var][val]` gives the unary factor value.
- `csp.binaryFactors[var1][var2][val1][val2]` gives the binary factor value. Here, `var1` and `var2` are variables and val1 and val2 are their corresponding values.

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

Hint: you can simply use `get_delta_weight()` rather than `csp.unaryFactors` and `csp.binaryFactors`.

Problem 1c [5 points] ⌨

The previous heuristics looked only at the local effects of a variable or value. Let's now implement arc consistency (AC-3) that we discussed in lecture. 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 domains have changed. 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()`.

You should make sure that your existing MCV implementation is compatible with your AC-3 algorithm as we will be using all three heuristics together during grading.

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!

Hint 1: documentation for `CSP.add_unary_factor()` and `CSP.add_binary_factor()` can be helpful.
Hint 2: although AC-3 works recursively, you may implement it iteratively. Using a queue might be a good idea. `li.pop(0)` removes and returns the first element for a python list `li`.
Hint 3: you should remove inconsistent values from `self.domains[var]` for some `var`, but the order of domain values should be kept. For example, when a value 2 is popped from `[1, 2, 3, 4]`, the next domain value list is `[1, 3, 4]` rather than `[3, 1, 4]`.

Problem 2: Handling n-ary factors

So far, our CSP solver only handles unary and binary factors, but for 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, and 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 variable should be true. 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 $(X_1, X_2, X_3, \text{True})$, which would return a new (auxiliary) variable X_4, and then add another constraint $[X_4 = \text{True}]$.

The second type of n-ary factors is constraints on the sum over n variables. You are going to implement reduction of this type.

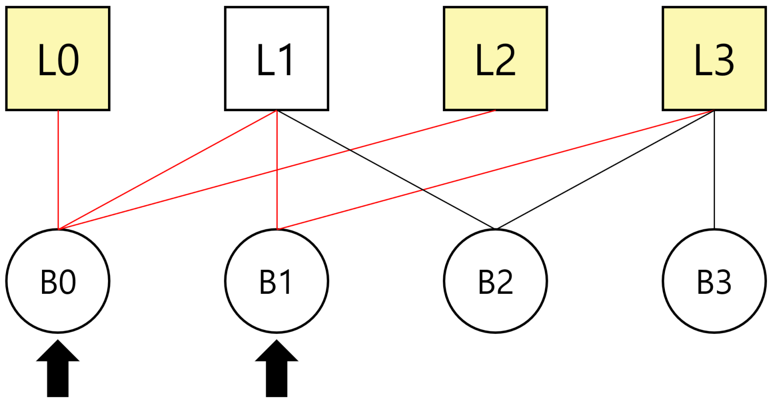
Problem 2a [5 points] ⌨

Let's implement `get_sum_variable()`, which takes in a sequence of non-negative integer-valued variables and returns a variable whose value is constrained to equal the sum of the variables. You will need to access the domains of the variables passed in, which you can assume contain only non-negative integers. The parameter `maxSum` is the maximum sum possible of all the variables. You can use this information to decide the proper domains for your auxiliary variables.

How can this function be useful? Suppose we wanted to enforce the constraint $[X_1+X_2+X_3 \leq K]$. We would call `get_sum_variable()` on (X_1, X_2, X_3) to get some auxiliary variable Y, and then add the constraint $[Y \leq K]$. Note: You don't have to implement the $\leq$ constraint for this part.

Problem 2b [4 points] ⌨

Let's create a CSP. Suppose you have n light bulbs, where each light bulb $i = 0, \ldots, n-1$ is initially off. You also have m buttons which control the lights. For each light bulb $i = 0, \ldots, n-1$, we know the subset $L_i \subseteq \{0, \ldots, m-1\}$ of buttons that control it. When button j is pressed, it toggles the state of light bulbs whose corresponding L includes j (If buttons B0 and B1 are pressed in the example shown in the figure, light bulbs L0, L2, and L3 will turn on, while L1 will remain off.). In code, L_i corresponds to `buttonSets[i]`. Your goal is to turn on all the light bulbs by pressing a subset of the buttons. Implement `create_lightbulb_csp` to solve this problem.


L0
L1
L2
L3
B0
B1
B2
B3