Project - 2: N-Queens

Adapted from the Course Scheduling of Stanford CS221

Introduction

In this project, you are going to construct a CSP for N-Queens problem. You will be given several python files.

Files you'll edit:

submission.py	Where you need to fill in codes in three places.

Files you need to look at:

grader.py	You can use this file to grade your program.
util.py	Some important supporting functions are provided including the class CSP.

Files to Edit and Submit: You will fill in portions of submission.py during the assignment. You should submit this file with your code and comments. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

Evaluation: Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's judgements -- will be the final judge of your score. We will review and grade assignments individually to ensure that you receive correct credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

CSP for N-queens

Notice we are already able to solve the CSPs, because in submission.py, a basic backtracking search is already implemented. The function will help you solve the CSP. You should read BacktrackingSearch carefully to make sure that you understand how the backtracking search is working.

Take a look at BacktrackingSearch.reset_results() to see the other fields which are set as a result of solving the CSP.

a. [5 points] Let's create a CSP to solve the n-queens problem: Given an n×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).

b. [5 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 Xj that has the fewest number of values a which are consistent with the current partial assignment (a for which get_delta_weight() on Xj=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.

c. [10 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 Xj to value a, we remove the values b of all neighboring variables Xk that could cause arc-inconsistencies. If Xk's domain has changed, we use Xk'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().

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!

Take a deep breath! This part requires time and effort to implement — be patient.

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.

Submission (IMPORTANT THINGS! READ THREE TIMES ☺)

- Submit your code, **only one: submission.py**, through E-learning in a zip file with the name "pj2-search".
- **Project 2 is due on May 7th, Sunday, 11:59pm, 2018.**

Something more you might want to know:

You will notice the term "weight" in the program. Each assignment might have various weights in some real problems (mentioned as "preference" in our class). **In this homework, you only need to care about unweighted CSP, in which the weight is either 0 or 1.**