Fall Semester, 2014 **B.Y. Choueiry**

CSCE 421/821: Foundations of Constraint Processing

Homework 5

Assigned: Wednesday, October 29th, 2014

Due: Wednesday, Nov 12th, 2014 (Wednesday, Nov 5th, 2014 progress report due).

Total value: 100 points (+ 25 bonus points). Penalty of 20 points for lack of clarity in the code.

Notes: You are stongly encouraged to initiate discussions with colleagues and check your results with them. However, you must do your own homework. *If you receive help from anyone, you must clearly acknowledge it.* Always acknowledge sources of information (URL, book, class notes, etc.). Please inform instructor quickly about typos or other errors.

Contents

---

1 N-Queen Problem as a CSP (23 points + 5 bonus)

Part 1: Consider the 4-queens problem where each queen is associated with a row and can be assigned to any column in the row.

1. Define this problem as a CSP. Specify the variables, their domains, and each binary constraint by 'extension.' (1 point)

2. Define a binary constraint C_{Q_i,Q_j} between two variables Q_i and Q_j by 'intension.' (4 points)
3. What is the size of this CSP (which is the size of the search tree it may yield)? (1 point)
4. Draw the constraint graph. (2 points)
5. Arc-consistency of a binary constraint C_{Q_i,Q_j} between two variables Q_i and Q_j ensures that every value for the variable Q_i has a support (at least one consistent value) in the domain of Q_j and vice-versa. Run manually arc-consistency on the 4-Queens problem. Can you remove any value? At the end of the operation the CSP is said to be arc-consistent. (2 points)
6. Arc-consistency is also called 2-consistency because it considers all combinations of two variables at the same time. Let's consider all combinations of 3 variables at the same time and let's check whether or not every value in the domain of a given variable has a support in the domain of the two other variables (simultaneously). If it does not, the value can be removed. Can you remove any value? This consistency property is called (1,2)-consistency. (4 points)

Part 2: Alternative modeling of the N-queens problem.

Adapted from Bernard Nadel'1990.

1. Since there can be no more than one queen per right (downward-sloping) diagonal we could associate a variable with each diagonal. According to this model, how many variables are necessary to model the N-queens problem as a CSP? Give your answer in terms of N. (3 points)
2. Each variable taking its value to indicate in which square of the corresponding diagonal the queen is, there are less queens than diagonals and some diagonals must contain no queen. Find a way to define the domains of these variables to allow the modeling of the statement: "a diagonal may contain no queen." (4 points)
3. What is the size of the CSP resulting from this model (state in terms of $N = 4$)? (2 points)
4. **Bonus:** Express the size of the CSP as a general expression in N. (5 points)

2 Latin square (8 points)

Adapted from of Daphne Koller, Stanford University.

A Latin Square is a $N \times N$ array filled with colors, in which no color appears more than once in any row or column. Finding a solution to a 4×4 Latin Square can be formulated as a CSP, with a variable for each cell in the array, each having a domain of $\{r, g, b, y\}$, and a set of constraints asserting that any pair of cells appearing in the same row must have different colors, and that any pair of cells appearing in the same column must have different colors.

1 **r**	2 **g**	3 **b**	4 **y**
5 **g**	6 **y**	*r g b y*	r g b y
7 **b**	r g b y	r g b y	r g b y
r g b y	r g b y	r g b y	r g b y

Figure 1: *Current state of search.*

At this point in the search, seven of the cells have been instantiated (displayed in **boldface** in Figure 1), and the initial domains of the remaining cells are shown. The next cell to instantiate has the domain values in *italics*.

Re-execute (from scratch) the *same seven* first assignments in the specified order and, at *each* assignment, draw the Latin Square while filtering the domains of the relative future variables. Do this process for the two following look-ahead strategies:

- the partial look-ahead strategy, forward checking (FC) (4 points)
- the full look-ahead strategy, maintaining arc consistency. (4 points)

Indicate eliminated values by crossing them out or just erasing them.

3 Simple scheduling problem (10 points)

Courtesy of Rina Dechter

Consider the problem of scheduling five tasks: T_1, T_2, T_3, T_4, and T_5, each of which takes one hour to complete. The tasks may start at 1:00, 2:00, 3:00. Any number of tasks can be executed simultaneously provided the following restrictions are satisfied.

- T_1 must start after T_3.
- T_3 must start before T_4 and after T_5.
- T_2 cannot execute at the same time as T_1.

- T_2 cannot execute at the same time as T_4.
- T_4 cannot start at 2:00.

1. Formulate the problem as a CSP by stating: the variables, their domain, and the applicable constraints. (4 points)
   Hints: focus on the start time of a task.
2. Draw the constraint graph. (2 points)
3. Apply arc-consistency to each constraint in the CSP until no values can be ruled out (i.e., the CSP becomes arc-consistent). (4 points)

4 FC with static and dynamic variable ordering (59 points + 20 bonus points)

The goal of this exercise is to implement the forward checking algorithm (FC). (As something to think about, in the next homework, you will have to extend this code to the hybrid FC-CBJ by merging FC of this homework with CBJ of homework 4.)

- Implement FC with static variable ordering. 15 points
- Report the results of FC with static ordering on the examples of Homework 2 for both one solution and all solutions. 5 points
- Implement FC with the four dynamic variable ordering heuristics studied so far. 15 points
- Report the results of FC with dynamic ordering on the examples of Homework 2 for both one solution and all solutions. 5 points
- Implement the domino effect 5 points
- Choose any two variable orderings. For example, one heuristic with static and dynamic ordering. Alternatively, dynamic ordering on two different ordering heuristics (e.g., future ddr). Report the results, *for finding one solution*, obtained on the random CSP instances Case 17d of the benchmarks. Draw the six graphs Bonus 20 points
  {ordering1,ordering2} $\times$ {#CC,#NV,CPU}.
- Your impressions on comparing the performance of BT, CBJ, and FC. 4 points
- *Progress report* (**Due Wednesday, Oct 22, 2014**): Submit a progress report documenting how far along you are and submit some version of your code. 10 points

4.1 General Indications

Please follow the general indications below:

- *Please make sure that you keep your code and protect your files.* Your name, date, and course number must appear in each file of code that you submit.
- All programs must be compiled, run, and tested on `cse.unl.edu`. Programs that do not run correctly in this environment will not be accepted unless prior approval is obtained. You must include a Makefile with your program so that your code can be compiled by issuing 'make' while on `cse.unl.edu`. You also must include a script called 'runProgram.sh' that contains the command to run your program.
- You must submit a README file with precise steps on how to compile, run and test your code. Failure to do so may result in no points for the homework.
- To facilitate debugging and the expectations of the homework assignment, web grader is set up to quickly evaluate the correctness of your program: `https://cse.unl.edu/~cse421/grade/`. After you have files submitted through webhandin, you will be able to run the web grader.

4.2 Domino Effect

Your code for dynamic variable ordering *must* implement the *domino effect*: Whenever a variable's domain is (or becomes) a singleton, the variable must be chosen for instantiation. Thus, whenever you apply the variable ordering heuristic, you should choose first the variable whose domains is a singleton, *breaking ties lexicographically.*

4.3 Running the Code

Your procedure(s) for FC should take the parameters specifying the ordering heuristic: LD, degree, or ddr. You are responsible for the dynamic ordering of the variables.

Specify the search algorithm BT, CBJ, or FC by passing parameters to the program. Your program should support BT and CBJ from the previous homework in addition to FC. You are required to implement the following flags to specify the algorithm and the ordering heuristic:

- **-s BT** for backtrack search
- **-s CBJ** for conflict directed backtrack search
- **-s FC** for forward checking
- **-u LX** for lexicographical ordering heuristic
- **-u LD** for least domain ordering heuristic
- **-u DEG** for degree domain ordering heuristic

- **-u DD** for domain degree domain ordering heuristic
- **-u dLD** for dynamic least domain ordering heuristic
- **-u dDEG** for dynamic degree domain ordering heuristic
- **-u dDD** for dynamic domain degree domain ordering heuristic
- **-f <filename>** for the file of the CSP problem

Notice that exactly one **-s**, one **-u** and one **-f** flags are passed to the program. Failure to follow the specification of the flags above may results in deduction of substantial amount of points.

Your output should be the same as in Homework 3/4. The output format that web grader will check is in the following:

```
Instance name: XXX
Search: BT|CBJ|FC
variable-order-heuristic: LX|LD|DEG|DD
var-static-dynamic: static|dynamic
value-ordering-heuristic: LX
val-static-dynamic: static
cc: XXX
nv: XXX
bt: XXX
cpu: XXX
First solution: <sequence of values for the variables in order to pass to the SolutionChecker>
all-sol cc: XXX
all-sol nv: XXX
all-sol bt: XXX
all-sol cpu: XXX
Number of solutions: XXX
```

where the XXX should be replaced with the corresponding values.

4.4 Data structures

In this homework you are required to implement additional data structures to store, retrieve, and maintain the past and future variables relative to assigned variables. Because these data structures will be used extensively during search, you should consider an efficient implementation of these functionalities. Irrespective of the programming language or the libraries of data structures you are using, for acceptable performance implement operations on the data structures that take constant time *whenever possible*. That is, every time you add or remove an element, the cost should be constant whenever possible. Avoid traversing the list for addition or removal of items unless the cost is negligible.