Fall Semester, 2005 **B.Y. Choueiry**

CSCE 421/821: Foundations of Constraint Processing

Homework 3

Assigned: Thursday, September 22, 2005

Due: Tuesday, October 11, 2005

Total value: 100 points. Penalty of 20 points for lack clarity and documentation in code.

Notes: This homework must be done individually. *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

Implementation of backtrack search

The goal of this exercise is to implement generic CSP solvers based on backtrack search and test their performance on the problem instances of Homework 2. Again, you are advised to do this homework carefully as it will provide the building-blocks to the following one. The various components of the homework will address the following issues:

• Implementing the data structures of a generic solver	20 points
• Implementing the vanilla-flavor solver: backtracking search (BT)	40 points
• Implementing functions for manipulating data and ordering variables	20 points
• Finally, reporting the results obtained from solving the four examples of homework 2	20 points

General indications:

- *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.
- A README file must be submitted. Otherwise, the entire homework is declared invalid.

1 Basic data structures

Below we specify (as best we can) the data structures (class objects) that need to be defined for storing the information necessary for a CSP solver. Every time we launch a solver on a particular CSP instance, we will generate an instance of one of these classes.

1. *All-Solvers.* Create a global variable (e.g., a linked list) for storing all instances of solvers generated. Every time a solver is launched, it should be pushed (preferably automatically) into this list. This is the data structure that we will go to in order to access the information regarding the various executions of solvers and the results of the executions.
2. *CSP-solver.* Create a class object for storing an instance of a CSP solver. (For lispers, use `defclass`.) This data structure should have the following attributes:
    - `id`: that can be given or automatically generated (e.g., using a generator of strings).
    - `csp-instance`: A pointer to the CSP instance being solved.
    - `cpu-time`: The value of CPU time that the solver has spent working on the instance (including creation and initialization of the data structures necessary for the solver).
    - `cc`: The number of calls to `consistent-p` (which is the number of constraint checks).
3. *BT-solver.* Create a class object for storing an instance of a BT solver. (For lispers, use `defclass`.) This class object should be a sub-class of the previous one and have the following attributes:
    - (Naturally, this class should inherit all the attributes of the solver class.)
    - `current-path`: A 2-dimensional array of length $n+1$ (where n is the number of variables in the CSP instance loaded) that stores in one entry of a row the name of a variable and in the second entry of the same row the value assigned to the variable.

      When using static variable ordering, the first entries in each row are initialized before search is started. Under dynamic ordering, these entries are filled as search proceeds. This is why we need two dimensional array instead of the vector used by Prosser in his paper.

- `current-domain`: A 2-dimensional array of length $n+1$. It stores in one entry of a row the name of a variable and in the second entry of the same row the current domain of the variable.
- `nv`: The number of times a value is instantiated to a variable (which is the number of nodes visited).
- `var-ordering-static`: is a Boolean equal to 1 if static variable ordering is used and equal to 0 otherwise.
- `variable-ordering`: A pointer to a function for variable ordering.
- `variable-ordering-heuristic`: should store the name of the variable ordering heuristic used.
- `val-ordering-static`: is a Boolean equal to 1 if static value ordering is used and equal to 0 otherwise.
- `value-ordering`: A pointer to a function for value ordering.
- `value-ordering-heuristic`: should store the name of the value ordering heuristic used. (We will ignore this aspect of search in the homework).

4. X-SOLVER. Create a class objects for storing an instance of an X-solver, where X is BM, FC, BJ, and CBJ. (For lispers, use `defclass`.) Each class should be a sub-class of BT-solver and should store in its attributes the data structures necessary for particular search mechanism. Create all four classes. For FC-SOLVER, implement *reductions* as a 2-dimensional array, storing in one entry of a row the name of a variable and in the second entry of the same row the list of list of values removed from the domain of the variable (treated as a stack).

2 Main functions/methods

Create the following general methods:

- `unassigned-variables`: a method that applies to an instance of a CSP-solver and returns the list of unassigned variables in the problem instance being solved.
- `instantiated-vars`: a method that applies to a *constraint* and returns the list of variables in the scope of the constraint that have been instantiated.
- `unassigned-vars`: a method that applies to a *constraint* and returns the list of variables in the scope of the constraint that have not been instantiated.
- `instantiated-vars`: a method an *instance of a BT-solver* and returns the list of instantiated variables in the problem instance being solved.
- `unassigned-vars`: a method an *instance of a BT-solver* and returns the list of unassigned variables in the problem instance being solved.

- `consistent-p`: a method that applies to a *constraint* and checks whether the instantiations of the instantiated variables in the scope of the constraint (i.e., the set of variable-value pairs) listed in sorted order of the instantiated variables are consistent with the constraint definition. The implementation of this method should take into account whether the constraint is declared as an intensive constraint or an extensive one.

  (Advanced note for non-binary constraints: careful for the arity of the tuple given as input as it may be different from that of the constraint. This requires some thinking...)

Create the following general functions:

- `least-domain`: a function that implements the *least-domain* variable-ordering heuristic. Given a set of variables, it returns the variable with the smallest domain.

- `degree`: a function that implements the *degree* variable-ordering heuristic. Given a set of variables, it returns the un-instantiated variable whose degree (where degree is the number of un-instantiated neighbors) is the smallest.

- `ddr`: a function that implements the *domain-degree ratio* variable-ordering heuristic. Given a set of variables, it returns the variable whose ratio of domain size to degree (where degree is the number of *un-instantiated* adjacent variables) $\frac{|domain\ size|}{degree}$ is the smallest.

3 Backtrack search (BT)

Implement a simple backtrack search using the above-defined data structures, functions, and methods. You should have a main function that takes as input the type of search to apply (in this case BT-solver), the name of the ordering heuristic, and whether the heuristic should be applied statistically or dynamically. *Naturally, the search mechanism described by Prosser should be modified to take these choices into account.*

4 Performance comparison

Finally, run your code on the 4 CSP instances you loaded in homework 3. Along with your documented code, report the results in tables shown below. Conclude with your observations.

Note that the number of constraint checks is incremented every time you check whether two variable-value pairs are consistent (i.e., a call to CHECKI,J IN X-LABEL), AND THE NUMBER OF NODES VISITED INCREMENTED EVERY TIME A VALUE IS ASSIGNED TO A VARIABLE (WHICH ALSO HAPPENS IN X-LABEL).

Chronological Backtrack Search						
	Least-domain					
	static			dynamic		
	#CC	#NV	CPU time	#CC	#NV	CPU time
4-Queens						
6-Queens						
Zebra						
Picture puzzle						

Chronological Backtrack Search						
	degree					
	static			dynamic		
	#CC	#NV	CPU time	#CC	#NV	CPU time
4-Queens						
6-Queens						
Zebra						
Picture puzzle						

Chronological Backtrack Search						
	ddr					
	static			dynamic		
	#CC	#NV	CPU time	#CC	#NV	CPU time
4-Queens						
6-Queens						
Zebra						
Picture puzzle						