**Spring Semester, 2008** **B.Y. Choueiry**

**CSCE 421/821: Foundations of Constraint Processing**

## Homework 3

# Implementation of Backtrack Search (BT)

**Assigned:** Monday, February 18, 2008

**Due:** Wednesday, February 27, 2008

**Total value:** 85 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



---

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 1. 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 5 points
- Implementing the vanilla-flavor solver: backtracking search (BT) 30 points
- Implementing functions for manipulating data and ordering variables 30 points
- Reporting the results obtained for finding *one solution* for each of the 12 (binary) examples loaded in Homework 1 (available from the site of the instructor). 20 points

- **Bonus:** Reporting the results obtained for finding *all solutions* for each of the 12 (binary) examples loaded in Homework 1 (available from the site of the instructor). 10 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.
- You can use the programming language of your choice. We are not able to provide any programming/debugging help. You are check *your results* with colleagues.
- Please inform instructor quickly about typos or other errors that may appear in the specifications or the files made available online.
- If you have a better idea for implementing your code and data structures, then you should experiment with it, and consider the data structures below *as mere recommendations*.

# 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* (Optional). 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. Lispers can use `defclass`. (Generally speaking, a local-search solver can be declared as an subclass of this general CSP solver.) This data structure should have the following attributes:
   - `id`: that can be given or automatically generated (e.g., using a generator of strings).
   - `problem`: A pointer to the CSP instance being solved.
   - `time-setup`: (optional) The value of CPU time that the solver has spent on the set-up, such as the creation and initialization of the data structures necessary for the solver.
   - `cpu-time`: The value of CPU time that the solver has spent working on solving the instance.
   - `cc`: The number of calls to `check` (which is the number of constraint checks).

3. *SystematicSearch-solver*. Create a class object for storing any backtrack search hybrid. 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 1-dimensional array of length $n+1$ (where $n$ is the number of variables in the CSP instance loaded) that stores in each entry the structure of a variable instantiated at the level of the entry.

     When using static variable ordering, the array is initialized before search is started. Under dynamic ordering, these entries are filled as search proceeds. Note that unlike the way it is described in Prosser's paper, we do not store the assigned values in this array.
   - `assignments`: A hash-table (or a vector) of maximum length $n+1$. It stores for each variable struture as a key, the value assigned to the variable as a value.
   - `current-domain`: A hash-table (or a vector) of maximum length $n+1$. It stores for each variable struture as a key, the current domain of the variable as a value.
   - `nv`: The number of times a value is instantiated to a variable, which is the number of nodes visited. Be careful not to confuse the number of nodes *expanded* (i.e., those that have children) with that of the number of nodes *visited*. That is a common source of error.
   - `bt`: The number of times backtracking occurs. That number is typically equal to the number of time x-unlabel is called.
   - `variable-ordering-heuristic`: should store the name of the variable ordering heuristic used.
   - `var-static-dynamic`: is equal to 'static if static variable ordering is used, otherwise equal to 'dynamic.
   - `value-ordering-heuristic`: should store the name of the value ordering heuristic used. (We will ignore this aspect of search in the homework, but you may need it in your projects.)
   - `val-static-dynamic`: is equal to 'static if static variable ordering is used, otherwise equal to 'dynamic. (We will ignore this aspect of search in the homework, but you may need it in your projects.)

4. *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. Because BT does not require any special data structures, this class does not need any special attributes.

# 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 an instance of a CSP-solver and returns the list of instantiated variables in the problem instance being solved.
- `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 that applies to a *constraint* and returns the list of variables in the scope of the constraint that have been instantiated.

Create the following general functions:

- Two functions that implement the *least-domain* variable-ordering heuristic:
  1. `ld-var`: takes a set of variables and returns the variable with the smallest domain.
  2. `ld-vars`: takes a set of variables and returns a setf of the same variables sorted in increasing domain size.

  item Two functions that implement the *degree* variable-ordering heuristic:
  1. `deg-var`: takes a set of variables and returns the variable with the largest degree. (Implement as 'largest future degree,' as in largest number of *unassigned* neighbors.)
  2. `deg-vars`: takes a set of variables and returns a set of the same variables sorted in decreasing degree value.
- Two functions that implement the *domain-degree ratio* variable-ordering heuristic:
  1. `ddr-var`: takes a set of variables and returns the variable whose ratio of domain size to degree, `ddr`$= \frac{|domain\ size|}{degree}$, is the smallest (where degree is the number of *un-instantiated* adjacent variables).
  2. `ddr-vars`: takes a set of variables and returns a set of the same variables sorted in increasing `ddr` value.

Alert to Lisp users: `sort` is powerful but destructive.

The functions `ld-vars`, `deg-vars`, `ddr-vars` are useful for static variable ordering (i.e., ordering the variables before search). The functions `ld-var`, `deg-var`, `ddr-var` are to be used for dynamic variable ordering.

## 3 Backtrack search (BT)

Implement a simple backtrack search with static variable ordering using the above-defined data structures, functions, and methods. However, you should prepare your code to take a dynamic variable ordering, which will be used in the following homework[1].

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.*

To test your implementation, run it on very simple, toy problems of increasing difficulty such as the ones provided by the instructor: http://cse.unl.edu/~choueiry/CSPTestInstances/. Compare the results with what you find by hand and also with those of your classmates.

## 4 Performance comparison

Finally, run your code on the 12 CSP instances you loaded in Homework 1, measuring the number of constaint checks #CC, number of nodes visited #NV, number of backtrack points #BT, and CPU time. Note that:

- #NV is incremented every time a value is assigned to a variable (which happens in X-LABEL, e.g., at line 4 of the function bt-label).
- #BT is incremented every time a *consistent* assignment is undone (which happens in X-UNLABEL). Although #NV and #BT are strongly related, measuring and recording #BT may help you debugging your code by comparing with results obtained by hand or by colleagues.
- #CC is incremented every time you check whether two variable-value pairs are consistent (i.e., a call to CHECK(i,j) in X-LABEL).
- Finding all solutions allows you to debug your code because all techniques must find the same number of solutions.

Handin your documented code and results as in the table shown below. Conclude with your observations.

[1]Dynamic variable ordering without some form of lookahead is very likely to increase the CPU time but it will not reduce #CC or #NV. Indeed, the benefit of dynamic variable ordering will not become visible until you implement FC in the following homework.

| Problem | Ordering | One solution | | | | All solutions | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | #CC | #NV | #BT | CPU time | #CC | #NV | #BT | CPU time | # Solutions |
| 4-queens | ld | | | | | | | | | |
| 4-queens | deg | | | | | | | | | |
| 4-queens | ddr | | | | | | | | | |
| 6-queens | ld | | | | | | | | | |
| 6-queens | deg | | | | | | | | | |
| 6-queens | ddr | | | | | | | | | |
| Zebra | ld | | | | | | | | | |
| Zebra | deg | | | | | | | | | |
| Zebra | ddr | | | | | | | | | |
| etc. | | | | | | | | | | |