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

CSCE 421/821: Foundations of Constraint Processing

Homework 3

Implementation of Backtrack Search (BT)

Assigned: Wednesday, Oct 1, 2014

Due: Wednesday, Oct 15, 2014 (Wednesday Oct 8, 2014 progess report due)

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

Contents

The goal of this homework and future couple of homework 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 ones and perhaps even your project. The various components of the homework will address the following issues:

- *Solver:* Implementing the data structures of a generic backtrack-search solver. 5 points
- *Basic functions and ordering heuristics:* Implementing functions for manipulating data and for variable-ordering heuristics. 30 points

- *BT:* Implementing the vanilla-flavor backtrack search (BT) to find a single solution and all solutions. Make sure to check the correctness of your solution using using the cheker provided in "Tool SolutionChecker" in the page: `http://www.cril.univ-artois.fr/~lecoutre/software.html#`. 30 points

- *Performance reporting:* Reporting the results obtained for finding *one* and *all* solutions for each of the binary instances loaded in Homework 1. 20 points

- *Progess report* (**Due Wednesday, Oct 8, 2014**): Submit a progress report documenting how far along you are and submit some version of your code. 15 points

1 General indications

- 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.

- A README file must be submitted. Otherwise, the entire homework is declared invalid. The README file should describe the content and purpose of the submitted files, and have *all* the necessary steps to compile and run the program on `cse.unl.edu`.

- You are strongly encouraged to discuss it with colleagues, but please do your own implementation. Always acknowledge sources of information (URL, book, class notes, etc.).

- You are strongly encouraged to put *your results* (in terms of nodes visited, constraint checks and CPU time) *as soon as possible* on the wiki page provided for this purpose and share them with colleagues for quick feedback and debugging.

- Please inform instructor quickly about typos or other errors that may appear in the specifications or the files made available online. Also, do email us any suggestions for improvement.

- 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.

2 Basic data structures for backtrack search

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*.

Below we specify (as best we can) the different fields required for the CSP solver class, which is the BT in this homework.

- `problem`: A pointer to the CSP instance being solved, which is the data structure you implemented in Homework 1.
- `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 times a constraint is accessed to check whether two variable-value pairs are consistent. This value is not the number of calls to the `check` function discussed in class. The `check` should first check the existence of a constraint between two variables before checking the consistency of the two variable-value pairs. When no constraint exists between two variables, #CC should not be incremented.
- `nv`: The number of nodes visited 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`: The number of times backtracking occurs. That number is equal to the number of times a variable is uninstantiated 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.
- `variable-ordering-heuristic`: should store the name of the variable ordering heuristic used. By default, we always use increasing lexicographical ordering. Also, we always break ties using increasing lexicographical ordering.
- `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. By default, we are going to use increasing lexicographical ordering. In the homework, we do not plan on implementing value heuristics at all, unless individual students choose to do so in their project, for their own benefit, or for bonus points (discuss with instructor).
- `val-static-dynamic`: is equal to 'static' if static variable ordering is used, otherwise equal to 'dynamic'.

The information contained in the fields listed above should be printed at the end of every program execution.

During the search process, the following data structures are required to hold the state of the algorithm:

- `current-path`: A 1-dimensional array or list that stores in each entry a pointer to 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.

  Alternatively, you may use a 2-dimensional array to store the variable and its assigned value.

- `assignments`: A data structure to hold the values assigned to each variable. Note that the information that this data structure is holding could be included in `current-path`.

  Having the `current-path` and `assignments` data structures is a major convenience for debugging and printing the results.

- `current-domain`: The current domain of each variable as search proceeds. As search proceeds, values from the domain of the variables are removed or added back. This data structure should be carefully designed so that the removal or addition of values can be performed (ideally) in constant time.

  You may choose to access the domain of the variables as stored in the problem instance itself (ref. Homework 1). Be careful not to ever modify/write over this data structure during problem solving.

3 Main functions/methods

It is essential that you design the different modules of your program appropriately because some of the modules that you implement here will be used in combination with others in the next homeworks. For this purpose it is essential that you carefully study Prosser's paper, and make sure you read it completely to get a sense of what modules are going to be replaced by others.

3.1 Basic functionalities

Consider implementing the following methods that will be used in this or next programming tasks. These methods also enhance the modularity of your code.

- `unassigned-variables`: a method/function that returns the list of unassigned variables in the problem instance being solved.

- `instantiated-vars`: a method/function that returns the list of instantiated variables in the problem instance being solved.

- `unassigned-vars`: a method/function 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/function that applies to a *constraint* and returns the list of variables in the scope of the constraint that have been instantiated.

3.2 Variable/value ordering

In the current homework, we will restrict ourselves to static variable and static value ordering. Dynamic variable ordering is significant only if we are using some kind of lookahead, such as forward checking. (Guess why.) We will implement dynamic variable ordering in a future homework.

Anyway, please prepare your code to be modular in the following way. Provide a function that 'asks' for the next variable to be instantiated (respectively, the next value to be assigned).

- In case the ordering is static, the ordering will be determined before search starts and stored in some data structure (e.g., $\widehat{c}$urrent-path).
- In case the ordering is dynamic, the function will execute the heuristic to determine the variable to be instantiated (respectively, value to be assigned).
- In all cases, *ties in the heuristics should be broken lexicographically* by the names of the variable (respectively, value). This feature is essential to avoid randomness in the results and so that you can compare your results with your colleagues.

The following four orderings are computed before beginning the search, using the initial state of the variables. The order does not change during search, hence they are referred to as static ordering heuristics.

- `id-var-st`: takes a set of variables and returns the sequence of the variables sorted lexicographically by the name of the variable.
- `ld-var-st` implements static *least-domain* variable-ordering heuristic. It takes a set of variables and returns the variables sorted with the smallest domain first.
- `deg-var-st` implements the static *degree* variable-ordering heuristic. It takes a set of variables and returns the ordered sequence of variable with the largest degree first.
- `ddr-var-st`: A function that implement the static *domain-degree ratio* variable-ordering heuristic. `ddr-var` takes a set of variables and returns the variables with the smallest ratio of domain size to degree, `ddr`= $\frac{|domain\ size|}{degree}$, first.

We will implement dynamic variable ordering when we implement FC.

4 Backtrack search (BT)

Implement a simple backtrack search with *static variable ordering* using the above-defined data structures, functions, and methods.

4.1 Guidelines

- Although in this homework you are implementing only static variable ordering, you should prepare your code to take a dynamic variable ordering, which will be used in a following homework.
- *Naturally, the search mechanism described by Prosser should be modified to take into account the above-listed choices.*

- Make sure to start back-checking from the root (level 0) down through the current path, so that #CC is comparable and BJ/CBJ/BM are sound and complete.
- Do not check two vvp's for consistency unless there exists a constraint between the two variables. If there is a universal constraint between the two variables, then you should *not* check the consistency of the two vvp's and you should *not* increment the #CC counter.
- Make sure to run node consistency before you run search. Do not run any arc-consistency algorithm at this stage (which would be a pre-processing step).
- 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 using the wiki. The quicker results are reported in the wiki, the better the debugging by everyone.

4.2 Input

You should pass parameters to the main function that specify

- 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.

Specify the above three parameters by using the following flags:

- **-a BT** for backtrack search
- **-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
- **-f <filename>** for the file of the CSP problem

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

4.3 Output

As output, your solver should return the following information either on the screen or in a file or both (your choice):

- `problem`: The name of the problem.
- `time-setup` (optional)
- `cpu-time`
- `cc`
- `nv`
- `bt`
- `variable-ordering-heuristic`
- `var-static-dynamic`
- `value-ordering-heuristic`
- `val-static-dynamic`
- The first solution it finds as a sequence of values for the variables in the order specified in the xml file, so it can be checked using the cheker provided in "Tool SolutionChecker" in the page: `http://www.cril.univ-artois.fr/~lecoutre/software.html#`. E.g., with the Tools2008.jar file, you can check with the SolutionChecker by running: 'java -cp Tools2008.jar abscon.instance.tools.SolutionChecker chain4-conflicts.xml 4 3 2 1'

The output format that web grader will check is in the following:

```
Instance name: XXX
variable-order-heuristic: LX|LD|DEG|UU
var-static-dynamic: static
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>
Number of solutiosn: XXX
```

where the XXX should be replaced with the corresponding values.

5 Performance comparison

Finally, run your code on the binary CSP instances you loaded in Homework 1, measuring the number of constraint checks #CC, number of nodes visited #NV, number of backtrack points #BT, and CPU time as defined in Section 2. Note that 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 (e.g., in an Excel sheet such as the one provided on the wiki) as in the table shown below. Conclude with your observations.