Fall Semester, 2009 **B.Y. Choueiry**

CSCE 896-004: Basic Algorithms and Concepts in Constraint Processing

Homework 5

FC-CBJ with Static and Dynamic Variable Ordering

Assigned: Tuesday, Oct 6, 2009

Due: Friday, Oct 16, 2009

Total value: 70 points for undergrads 85 points for graduates. Penalty of 10 points for lack of clarity.

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 homework is to first implement the hybrid FC-CBJ by mixing the procedures investigated in the previous two homeworks; then to test the performance of the hybrid with the four static and dynamic variable heuristics implemented in previous homework.

- *Minimal width ordering:* (15 points)
  Bonus for undergraduate, mandatory for graduate students.
  Implement the *static* variable-ordering heuristic called *width* (10 points) and include it in the experiments (5 points).

- Implementing FC-CBJ with static variable-ordering heuristics (one solution and all solutions). (20 points)

- Implementing FC-CBJ with dynamic variable-ordering heuristics (one solution and all solutions). (20 points)

- Reporting the results obtained on the simple examples of Homework 2 and the random instances of Homework 3. (10 points)

  Note that the results are reported by reporting the number of constraint checks, nodes visited and the time taken for solving the CSP by finding a single solution or reporting that the problem cannot be solved. The process is applied for the four ordering heuristics.

- Comparing your results on BT, CBJ, FC (static and dynamic), FC-CBJ (static and dynamic) for each instance, comment *as thoroughly and critically as possible* given the restricted number of instances tested on the performance of:
  - the algorithms, (5 points)
  - the variable ordering heuristics, and (5 points)
  - the static and dynamic variable orderings. (5 points)
  - whether we are looking for one or all solutions. (5 points)

1 Variable Ordering: Width

The notion of a the *width* of a graph is an important notion in Graph Theory with many applications in Compute Science (CS) in general. In Theoretical of Databases, Bayesian Networks, Model Checking, and Constraint Processing, various tractability results are drawn based on the notion of bounded width (more precisely, tree width and hyper width) of the (hyper)graphs representing the corresponding computational problems. Without going into much details now, we hope to address that next semester in the course CSCE 990 Advanced Constraint Processing, we would like to implement the very simple procedure that orders the variables for a CSP (i.e., the nodes of the constraint graph) according to the minimum width ordering. The same (fortunately!) polynomial-time algorithm can be used to compute the width of the graph and to find the corresponding ordering. In backtrack-based search, this ordering has to be used as, of course, a static ordering.

1.1 Definition of width of a graph

First, let us revise the notion of the width of a graph.

- When you order the nodes of a graph linearly (as a vertical chain), the *parents* of a node are the nodes that are its neighbors in the graph *and* appearing before it in the ordering.

- Given an ordering, *the width of a node in the ordering* is defined as the number of its parents in the ordering.

- Given an ordering, *the width of the ordering* is defined as the maximum width of all nodes in the ordering.
- Now, *the width of a graph* is defined as the minimum width of all its possible orderings.

If a graph has n vertices, it has $n!$ possibly orderings. So, computing the width of the graph may appear as a daunting task. Fortunately, there is a sound (i.e., correct) and efficient (i.e., polynomial time) algorithm for computing the width of the graph. It runs in $\mathcal{O}(n^2)$

1.2 Algorithm

Below, we explain in plain English how the algorithm operates. Compute the degree of all the nodes in the graph. Set the k, the current value of the width, to zero. Increment the value of the current width and repeat the following until no nodes are left in the graph. While there are still nodes in the graph, choose the node with the smallest degree in the graph (breaking ties lexicographically), set the current value of the width to the degree of this node, remove the node from the graph deleting all its incident edges and decrementing the degree of the neighbors of the removed nodes. Loop over all the remaining nodes, iteratively removing all those nodes whose degree is smaller than or equal to the value of the current width. Every time a node is removed, remove its incident edges and decrement the degree of its adjacent nodes). When there are no nodes left in the graph, the width of the graph is equal to the value of the current width and the ordering of minimal width is the *reverse* of the order in which the nodes were removed. Below is the pseudocode of the algorithm:

```
WIDTH (graph)
1: Remove from the graph all nodes that are not connected to any others
2: k ≠ 0
3: while there are nodes left in the graph do
4:    k ≠ k + 1
5:    while there are nodes connected to k or less other nodes do
6:       remove them from the graph along with the edges incident to them
7:    end while
8: end while
9: RETURN(k, the nodes in their elimination order)
```

Note that the minimum width ordering of the nodes is the reverse of their elimination order.

2 Implementing FC-CBJ

Below are some comments and indications to help you in your implementation of FC-CBJ.

2.1 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.
- 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.

2.2 Running the Code

Your procedure(s) for FC-CBJ 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, FC or FC-CBJ by passing parameters to the program. Your program should support BT and CBJ from the previous homeworks in addition to FC and FC-CBJ. You are required to implement the following flags to specify the algorithm and the ordering heuristic:

- **-aBT** for backtrack search
- **-aCBJ** for conflict directed backtrack search
- **-aFC** for forward checking
- **-aFCCBJ** for forward checking with conflict directed backtracking
- **-uLX** for lexicographical ordering heuristic
- **-uLD** for least domain ordering heuristic
- **-uDEG** for degree domain ordering heuristic
- **-uDD** for domain degree domain ordering heuristic
- **-uW** for width ordering heuristic
- **-udLD** for dynamic least domain ordering heuristic
- **-udDEG** for dynamic degree domain ordering heuristic
- **-udDD** for dynamic 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 results in deduction of substantial amount of points.

2.3 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.

2.4 Modularity of the Code

FC and FC-CBJ follow the same procedure, with the difference that FC-CBJ performs the backtracking by possibly jumping over several levels. For this purpose, FC-CBJ maintains additional data structures that FC does not.

Avoid implementing two separate procedures. Design your code carefully to have single FC procedure, and extend it to FC-CBJ by switching the procedures within FC that implement FC-CBJ. Invest some time to decide a good modularity level of your code that enables you to have single FC procedure.

The advantage of this design will be visible during debugging. The algorithms might seem simple, but there is substantial amount of detail that needs to be carefully handled. In case you implement two separate procedures, you might end up debugging the same errors twice: once for FC and once for FC-CBJ.

The above is a mere recommendation. You are not obliged to abide by it. No points will be deducted if you implement two separate procedures for FC and FC-CBJ.