Fall Semester, 2014 **B.Y. Choueiry**
CSCE 421/821: Foundations of Constraint Processing

Homework 2

Implementation of Arc Consistency

Assigned: Wednesday, Sep 17, 2014

Due: Wednesday, Sep 24, 2014: Initial AC1. Wednesday, Oct 1, 2014: Full Assignment.

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

Contents

The goal of this homework is to implement two basic algorithms, namely AC1 and AC3, for enforcing the arc-consistency property on a finite, binary CSP [2, 1]. You will also learn to use Excel to report your results, compute averages, and generate plots of your results.

The various components of these algorithms will be used extensively by other components of the CSP Solver that you will build during the semester. As such, you should pay special attention to the following two issues:

1. The data structures of the algorithms that you implement need to be totally different from the data structures that store the components of a CSP instances (i.e., variable, values, domains). The latter can be accessed and checked, but should not be modified by any of the functions or procedures of any solver algorithm.

2. Keep your implementation as modular as possible so that the various functions can be reused, with the proper input, by more than one component of your solver. For instance, the function CHECK can be used by arc consistency, path consistency, look-ahead in search, etc.

Again, you are advised to do this homework carefully as it will provide the building-blocks to the following homework and perhaps even your project. The various components of the homework will address the following issues:

- Implementing the data structures of the AC1/AC3 algorithm. Those are:
  - The *current-domains* for copying and modifying (i.e., filtering) the domains of the variables and 5 points
  - `Queue` for storing the directed edges to be revised. 5 points
- Implement a node-consistency algorithm. 3 points
- Implement the arc-consistency algorithm AC1. 20 points
- Implement the arc-consistency algorithm AC3. 10 points
- *Performance reporting:* Reporting the results obtained for running NC *then* AC1/AC3 on the benchmark problems. 15 points
- *Performance reporting:* Reporting the results obtained for running NC *then* AC1/AC3 on randomly generated problems from the entry **17c** in the page `http://cse.unl.edu/~choueiry/CSPTestInstances/`. Use the Excel sheet provided on the wiki to store the results, generate average values and plot your results. The instances provided include 50 instances for each constraint tightness value, which will allow you to 'see' the so-called 'phase transition.' 20 points
- Your observations. 2 points
- *Initial AC1 results* (**Due Wednesday, Sep 24, 2014**): 10 points
  Fill the Excel sheet provided on the wiki for AC1 on the benchmark problems and make it available to your classmates. These are only initial results, and the results can be updated.
- Fill the Excel sheet provided on the wiki for AC3 and make it available. 10 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 extensively discuss it with colleagues, but please do your own implementation. *If you receive help from anyone, you must clearly acknowledge it.* Always acknowledge sources of information (URL, book, class notes, etc.).
- *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.
- You are encouraged to put *your results* (in terms of number constraint checks, number of values removed, filtering effectiveness, 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.
- 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 AC1/AC3

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 AC1/AC3 in this homework.

- `problem`: A pointer to the CSP instance being solved.
- (optional) `time-setup`: 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 the instance.

- `cc`: The number of times a constraint is accessed to check whether two variable-value pairs are consistent. This number should not include the number of times a universal constraint is accessed: When no constraint exists between two variables, #CC should not be incremented.

- `fval`: The sum of the number of values removed by AC1/AC3 from the domains of all variables.

- `iSize`: The initial size of the CSP is the number of combinations of variable-value pairs for all variables, it is the product of the initial domain sizes: $\Pi_{V_i \in \mathcal{V}} |D_{V_i}|$. (Remember, the product rule in Discrete Math.) If those values are too large, you may want to report the logarithm:
$$\ln(\Pi_{V_i \in \mathcal{V}} |D_{V_i}|) = \Sigma_{V_i \in \mathcal{V}} \ln(|D_{V_i}|).$$

- `fSize`: If the problem is arc consistent, the size of the filtered CSP after enforcing arc consistency $\Pi_{V_i \in \mathcal{V}} |D'_{V_i}|$. If those values are too large, you may want to report the logarithm:
$$\ln(\Pi_{V_i \in \mathcal{V}} |D_{V_i}|) = \Sigma_{V_i \in \mathcal{V}} \ln(|D'_{V_i}|).$$
All arc-consistency algorithms should result in the same CSP, and thus the same size. If the problem is not arc consistent, then this value is not defined (`false`, `undetermined`, `undefined`).

- `fEffect`: If the problem is arc consistent, the value of filtering effectiveness if given by `filter` as
$$\ln(\Pi_{V_i \in \mathcal{V}} \frac{|D_{V_i}|}{|D'_{V_i}|}) = \Sigma_{V_i \in \mathcal{V}} (\ln \frac{|D_{V_i}|}{|D'_{V_i}|}) = \Sigma_{V_i \in \mathcal{V}} (\ln |D_{V_i}| - \ln |D'_{V_i}|),$$
otherwise it is `false`/`undetermined`.

  Using the logarithm allows us to 'curb' or 'reduce' the numerical imprecision due to too large or too small numbers. I am hoping that it works, I am not that it does, but it is worth trying.

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

The following data structures are needed for the operation of the algorithms:

- `current-domain`: For each variable, some structure where to copy the initial domain (`initial-domain`?) stored in the CSP data structures (which should *not* be modified). This structure can be of the same type as the one storing the instance, but it should be a copy, not the same, because `current-domain` will be modified and `initial-domain` should not be modified. Note that other solvers we will implement will also require the use of `current-domain`.

In order to ensure that your programs return comparable results to help you in the debugging, I strongly recommend that you sort the domains of the variables lexicographically and iterate over them in that order. Otherwise, your results may slight differ in terms of #CC especially if the CSP is not arc consistent (depending on whether we encounter domain annihilation earlier or later).

- `Queue`: A data structure that holds directed edges corresponding to the constraints in the original problem. You could store the list (V_i, V_j, R_{ij}) to facilitate access to the constraint definition. Remember to store both (V_i, V_j, R_{ij}) and (V_j, V_i, R_{ij}) because the two 'directed' edges of a given constraint are handled in separation by the algorithm AC3. The cardinality of the initial `Queue` is $|\texttt{Queue}| = 2e$, where e is the number of constraints in the problem. Note that:
  - The queue in AC1 is static.
  - The queue in AC3 is dynamic: elements will be added and removed, which will allow AC3 to have a better worst case complexity in practice.

  You are free to implement `Queue` as you wish (e.g., a linked list, maphash, or other).

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 homework.

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.

- CHECK$((V_i, a), (V_j, b))$: a function that determines whether two variable-value pairs are consistent. The method should return `true` if it is, `false` otherwise. Further, it should increment #CC if the constraint C_{V_i,V_j} is not universal.
  Caution: for those using the Java parser, apparently there is a `Check` function available for all types of constraints, including the ones defined in intension. You may want to rely on using it.

- SUPPORTED$((V_i, a), V_j)$: a function that returns `true` as soon as it finds a support for (V_i, a) in R_{V_i,V_j}. *This function should stop* right after encountering the first support, otherwise, performance may be negatively affected. (Why look for more supports if you have one?) This function is not included in your textbook or reference material but it is included in `Revise`. I find it useful to factor out.

- REVISE(V_i, V_j): a function filters D_{V_i} given C_{V_i,V_j} so that V_i is arc consistent according to C_{V_i,V_j}. This function is the one that modifies D_{V_i}

- AC1 and AC3: Relying on the description in the book, the paper, Bartak's web site, and/or the slides, implement the algorithms AC1 and AC3.

  It is important to check whether D_{V_i} is annihilated every time D_{V_i} has been updated and *return AC1, AC3 at this point indicating that the problem is not arc consistent* and cannot possibly have a solution, and thus all processing should stop. We say that the problem is *globally inconsistent*. It is interesting to note that a cheap, polynomial time algorithm such as the arc-consistency algorithm AC1 or AC3 may sometimes be able to discover by operating *locally* that an instance of an NP-complete problem is *globally* inconsistent. Here is the entire 'power' of local consistency properties and their algorithms. The heart of CP.

  AC1 revises each directed arc in the `Queue`, one at a time, from the beginning until the end `Queue`, does not modify `Queue`, and restarts from scratch every time *any* domain has been revised.

  AC3 starts with the same queue as AC1. However, every time, it *removes* an element from `Queue` when it revises it. However, if the domain is filtered, then all directed edges that may be affected are added back into the `Queue` unless they are there already. You may enjoy figuring out an efficient way to determine whether or not a given a directed edge is already in `Queue`. However, at this stage of the course, performance is not so central. So do not lose too much time on this issue.

3.2 Input

Implement NC, AC1, and AC3. Note that the execution of AC1 and AC3 should always be preceded by NC. Your code should take the parameters specifying the algorithm and input filename. You are required to use the following flags to specify the algorithm and filename:

- **-f <filename>** for the file of the CSP instance
- **-a ac1** for AC1
- **-a ac3** for AC3

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

3.3 Output

As output, your solver should return the following information on the screen:

- `problem`: The name of the problem
- `time-setup`, (optional)
- `cc`
- `cpu`
- `fval`
- `iSize`
- `fSize`
- `fEffect`

You might consider adding a flag to your program to output this information in a comma/tab separated format, for ease of copying into excel.

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

```
Instance name: XXX
cc: XXX
cpu: XXX
fval: XXX
iSize: XXX
fSize: XXX
fEffect: XXX
```

where the XXX should be replaced with the computed values for the loaded instance.

4 Performance comparison

Finally, run your code on the binary CSP instances you loaded in Homework 1, measuring the parameters listed below for *both* AC1 and AC3. Note that running both AC1 and AC3 allows you to debug your code because both algorithms must find the same CSP for instances that are arc consistent. Handin your documented code and results (e.g., in an Excel sheet) as provided in the two sheets of the Excel file on the wiki. The first sheet should be used to report the results on the benchmark problems tested in Homework 1. The second sheet should be used to report the results of the randomly generated instances **17c**, which should be reported, then averaged and ploted.

Conclude with your observations.

References

[1] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[2] Alan K. Mackworth and Eugene C. Freuder. The Complexity of Some Polynomial Network Consistency algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, (25) 1:65–74, 1984.