School of Computer Science
University of St Andrews
2022-23
CS4402
Constraint Programming
Practical 2: Constraint Solver Implementation

This Practical comprises 50% of the coursework component of CS4402. It is due on Friday 25th November at 21:00 (NB MMS is the definitive source for deadlines and weights). The deliverables comprise:

- A report, the contents of which are specified below.
- The Java source code for the constraint solver you will implement.
    - Include a readme file explaining how to compile and run your submission. *Do not submit* source code that requires a particular IDE to be run.
- The instance files for any CSP instances you use in addition to those provided to test your solver.

The practical will be marked following the standard mark descriptors as given in the Student Handbook (see link below).

Problem Specification

This practical is to design, implement in Java, and test empirically a constraint solver for **binary constraints**. Your solver should employ **2-way branching**, and it should implement both the **Forward Checking** and the **Maintaining Arc Consistency** algorithms. It should support two *variable* ordering strategies:

- **Ascending** (i.e. in the order specified in the supplied .csp file), and
- **Smallest-domain first**

It should support **ascending** *value* ordering.

Supplied Files

Accompanying this specification you will find the following Java files, which you should extend and/or modify to produce your submission:

- `BinaryConstraint.java`
- `BinaryCSP.java`
- `BinaryCSPReader.java`
- `BinaryTuple.java`

You will need to add further classes, e.g. a `Solver` class hierarchy, in order to complete your submission.

In addition, you will find ten `.csp` files (a format that can be read by the

`BinaryCSPReader`), which contain instances of three problem classes that you should use to test your solver. Note that the `.csp` format assumes that variable domains are specified simply as an integer range. You may wish to extend this.

There are also three Generator Java source files that were used to generate these instances, and can be used to generate more. The Sudoku generator produces the constraints only for the Sudoku puzzle – you will need to edit the domains in the generated .csp file to provide the clues for a particular instance. See the two provided Sudoku instances for examples. Langford's Number Problem may be unfamiliar to you. Its description can be found at CSPLib entry 24: http://www.csplib.org/Problems/prob024/

Input format

We will explain the simple input format with an example. We will use the n-queens problem, with n=4 (4Queens.csp).

Comments start with two slashes, like in the C programming language:

```
// 4-Queens.  This line is a comment
```

The first line of the file has a single number, expressing the number of variables your problem has.

```
// Number of variables:

4
```

Following the number of variables, you will have one line per variable, describing its domain. As we have 4 variables, we will have one line per variable. For example, the first line describes the domain of the first variable, expressed as 0, 3. This domain is `{0,1,2,3}`, equivalent to `int(0..3)` in Essence Prime.

```
// Domains of the variables: 0.. (inclusive)

0, 3

0, 3

0, 3

0, 3
```

Finally an arbitrary list of binary constraints. A constraint starts with a header, and then lists all the possible values those variables can take. In this example:

```
// constraints (vars indexed from 0, allowed tuples):

c(0, 1)

0, 2

0, 3

1, 3

2, 0

3, 0

3, 1
```

The header starts with the letter c, and between parentheses appear the two variables that are constrained. In our case, variable 0 and variable 1. From here you can infer that variables start indexing by 0. Then the list of valid values is specified. It can be read that the valid list of values is:

- v0 = 0 and v1 = 2, or
- v0 = 0 and v1 = 3, or
- v0 = 1 and v1 = 3,
- and so on….

Output Format

The format of the solver's **standard output** (stdout) will need to be the following:

- The first line will be a single integer which denotes the number of search nodes.
- The second line will be a single integer which denotes the number of arc revisions done.
- The third and any subsequent lines will represent the solution found. Each one will contain an integer, which will be the assignment to the corresponding variable. The order of the variables will be the same as in the input format.

Continuing with the 4 queens example, the output:

```
8

12

1

3

0

2
```

Will represent that the solver has explored 8 nodes, done 12 arc revisions and the queens are respectively located in the second row, fourth row, first row and third row.

This output should go into the standard output (stdout) stream, but feel free to use standard error (stderr) in any way you want. Note that only one solution is required from the solver in the output, but feel free to explore ways to either count or find all solutions.

Artifact

As marking will be supported by automatic tests, there are a few requirements that your submission should follow:

- Your code should be compiled and included in a self-executable JAR file.
- This JAR file should be included in the root directory of your submission.

- The output should follow the guidelines in the **Output Format** section.
- The submission should include an easy way to rebuild the JAR file, using for example a bash script, a Makefile script, or similar tools.
- Finally, the executable should adhere to the following command line parameters: `file.csp Algorithm VarOrder ValOrder`

  where:

  - `File.csp` is the path to the .csp file to be solved.
  - `Algorithm` can be either: `fc, mac`
  - `VarOrder` can be either: `asc, sdf`
  - `ValOrder` can only be `asc`

  For example, the execution:

  ```
  $ java -jar P2.jar csps/4queen.csp fc sdf asc
  ```

  Will try to solve the 4 queens problem (located in csps/4queen.csp) using Forward Checking, with the Smallest Domain First Variable ordering and the ascending value ordering.

Basic Solver Design

In designing your solver, you will need to decide upon a suitable representation for variables and domains. A partial implementation of binary extensional constraints is provided in `BinaryConstraint` and `BinaryTuple`. Some considerations follow:

- To implement both Forward Checking and Maintaining Arc Consistency your design must support domain pruning via arc revision. It must also support a mechanism by which domain pruning is undone upon backtracking.

- The main additional consideration for Maintaining Arc Consistency over Forward Checking is the queue. Again, careful implementation is required here. Take care:
  - To establish arc consistency **before search**.
  - To **initialise the queue correctly** subsequently.

- Your implementation of the smallest-domain first heuristic should access the current state of the pruned domains to make its decisions.

- Take care in implementing two-way branching that arc revision is performed on **both branches**, and make sure to restore the domains correctly for subsequent value selection.

- The **smallest-domain first** variable heuristic can reach a tie, which should be broken. To break the tie of two or more variables having the same domain size, select the smaller index (that is, the first one of those specified in the .csp file).

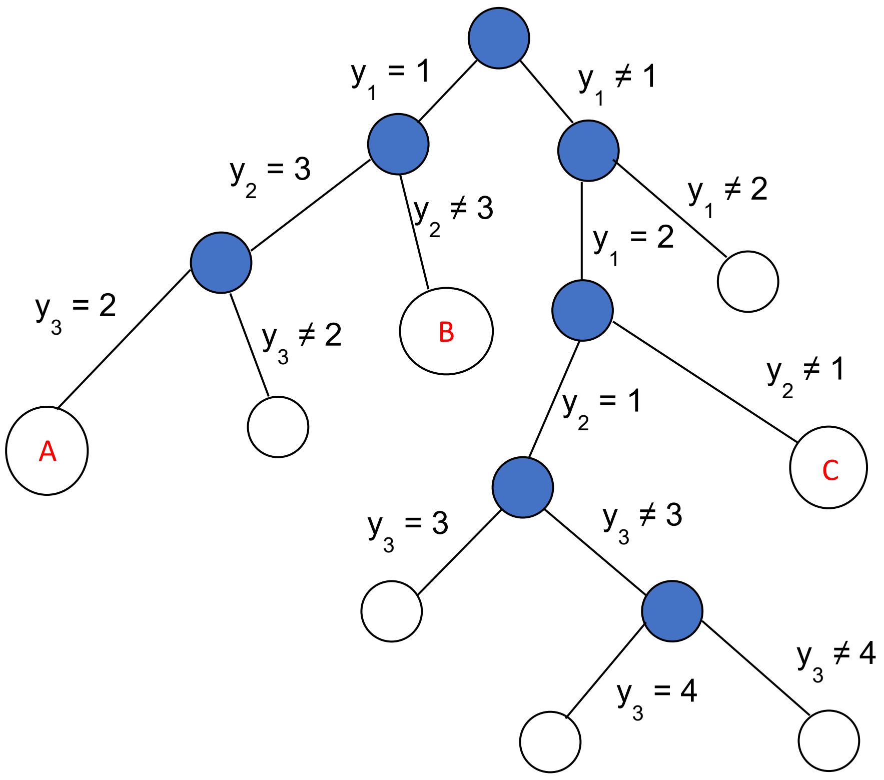
- When counting **arc revisions**, remember that arcs have a direction.

- For the purposes of **search node counting:**
    - Both left and right branches count as search nodes.
    - A variable with a single element in its domain still needs to be assigned. Therefore, the left branch that assigns that variable will count as a search node.
    - A domain wipeout occurring by either arc revision or simply by removing the last element of the domain in a right branch should still be counted as a search node.

Your source code should be readable and well commented.

Node Count Example

This is an example that aims to clarify how search nodes should be counted.



Consider this search tree example. At the root node, there is nothing decided. Therefore, the search node count there is 0. Let's now consider the search in three different moments (A, B and C):

A - The search progresses by assigning $y_1 = 1$, $y_2 = 3$ and $y_3 = 2$. After the assignment, arc revision finds that the assignment is not consistent. At this point, before backtracking we have done 3 search nodes.
B – Here we have backtracked up until having only $y_1 = 1$. As $y_2 = 3$ has not worked out, we do a right branch and find that after removing 3 from the domain, arc revision empties the domain of a variable. At this point we have visited 5 search nodes

C - The last example is a situation where we have visited a total of 13 search nodes. At this point we do the right branch of y_2 , removing 1 from its domain and emptying it. Notice that at point C we have not yet backtracked and removed the value 2 from the domain of y_1, which is why that node is not counted in the total of 13.

Empirical Evaluation

Using the supplied instances and instance generators, design and run a set of experiments to compare the merits of Forward Checking and Maintaining Arc Consistency using both ascending variable ordering and smallest-domain first.

For each of the provided instances, report the following three measures:

- Time taken,
- Nodes in the search tree,
- Arc revisions.

In addition to that, discuss how the techniques compare and if they differ substantially in a given instance or problem class, explain why that is the case.

Problem Generators

In addition to the evaluation of the given instances, it is **required** that you support your analysis of each problem class by using the generators to create sets of new instances and for:

- Langford's problem: Keep changing the generator parameters and plot how the three recorded measures change with respect to the input values of the generator.
    - Can your intuition find a relationship between parameters and any of the three recorded measures?
    - Try solving with another technique. Do the same conclusions apply?
- N-queens problem: Analyze how the instances of this class behave in a similar way as you did with the Langford's problem.
- Regarding the Sudoku, generate a set of them and record the measurements. Remember that for creating them you will need to add the hints manually. You might reproduce some of the Sudoku puzzles from: https://www.sudokuwiki.org/Daily_Sudoku

Note that the difficulty rating of each puzzle is stated in the calendar.

  - Does any measure differ significantly between the instances?
  - Does the difficulty rating of the puzzle affect any of the measures?

Report

Your report, apart from the usual **Introduction** and **Conclusion** sections, should at least have the following ones:

- **Forward Checking**: Describe your implementation of the Forward Checking algorithm, including the data structures you used and your method of supporting domain pruning (and restoration following backtracking). Include here also an account of how you implemented binary branching.
- **Maintaining Arc Consistency**: Similarly, describe how you implemented this algorithm, and in particular how you manage the queue.
- **Empirical Evaluation**: Describe your experimental setup, and record and analyse the output of the full empirical evaluation described above. Include a table with the measures taken for all the solved instances.

Feel free to also describe any additional work that you have done, if any.

Marking

This practical will be marked following the standard mark descriptors as given in the Student Handbook. There follows further guidance as to what is expected:

- To achieve a mark of 7 or higher: A rudimentary attempt at Forward Checking only. Adequate evaluation and report.
- To achieve a mark of 11 or higher: A reasonable attempt at both Forward Checking and Maintaining Arc Consistency, mostly complete, or complete and with a few flaws. Reasonably well evaluated and reported.
- To achieve a mark of 14 or higher: A good attempt at both Forward Checking and Maintaining Arc Consistency, with at most only minor flaws. Well evaluated and reported.
- To achieve a mark of 17: A fully correct implementation of both algorithms, very well evaluated and reported.
- To achieve a mark greater than 17: In addition to the requirements for a mark of 17, evidence of an exceptional achievement in terms of technical challenge, new problems and/or additional features implemented to the solver such as other heuristics or constraints.

Pointers

Your attention is drawn to the following:

- Mark Descriptors:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html

- Lateness:
  https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html

- Good Academic Practice:
  https://info.cs.st-andrews.ac.uk/student-handbook/academic/gap.html