School of Computer Science
University of St Andrews
2020-21
CS4402
Constraint Programming
Practical 2: Constraint Solver Implementation

This Practical comprises 50% of the coursework component of CS4402. It is due on Friday 27th 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. A description of the .csp file format is in the **appendix** of this document.

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/

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`. Considerations:

- 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 (see your lecture notes):
    - 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**.

Your source code should be well commented.

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. Instrument your solver to record three measures:

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

For each of the provided instances, report the measures and discuss how the techniques compare. Then, for each problem class, use the generators to create new instances and:

- Langford's problem: Gradually increase the two parameters and record how the performance of your solver in its various configurations changes with respect to the three recorded measures.
    - Is there a relationship between the parameters and any of the three recorded measures?

- Perform a similar set of experiments and analysis for the n-queens problem as n increases.
- Construct a set of sudoku instances (there are various generators easily available online) and solve them with your solver in its various configurations. Does the difficulty of solving these instances vary significantly?

Report

Your report should have the following sections:

- **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 empirical evaluation described above.

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 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.
- Marks above 17 will be awarded according to the quality of the implementation, and the depth and quality of the empirical analysis and report.

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

Appendix - .csp Input format

We will explain the simple input format with an example, 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 that defines its scope (recall from lectures that the scope of a constraint is the variables it involves). As per the comment below the variables are indexed from 0. It then lists all the acceptable combinations of 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 parenthesis 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….