CS4402 Practical 2 - Binary constraint solver

170008773

21st April 2018

1 Introduction

For this assignement a basic binary constraint solver was to be implemented using Forward Checking (FC) and 2-way branching. The solver was also required to be able to use both dynamic and static heuristics. Eventually the solver was extended to utilise Maininting Arc Consistency (MAC). An emperical comparison between the algorithms and several heuristics was also made. MAC with a dynamic, smallest domain heuristics performed best overall.

2 Desing and implementation

2.1 Variables and domains

Variables To make some of the necessary bookkeeping easier, such as keeping track of the domains, a `CSPVariable` class was implemented. This class keeps track of things such as the domain, whether the variable is assigned, the name of the variable and eventually what order it has in a possible static heuristic. The name of the variable also enables naming variables instead of merely asigning them a numeric id, which is favorable when moddeling more complex problems.

Domains The domains were originally modled as integer ranges but this was quickly changed to being moddled by a `Set` of integers. This was done for several reasons. Firstly this supports domains which are not continues ranges, which is required in certain problems. Secondly, this allowes for fast checking of containment, adding and removal of values which is instrumental for a constraint solver. Thirdly, this allows for easier extentions when more complex objects become desirable in domains.

2.2 Constraints vs. Arcs

Since the solver only had to support binary table constraints the constraints are modled as two variables and a set of tuples. The tuples representing the allowed value assignements, and stored in a list. This proved problematic for two reasons. Firstly there was the trouble of finding the correct constraints without having to check all of them, since this would be very expensive for a large amount of constraints. Secondly it introduced a lot of amiguity over the domain of which variable had to get pruned. This was eventually solved by to changes. Firstly the constraints we converted from bi-directional contraints into uni-dicrectional arcs. So for example that meant that the constraint $c(x, y) = \{\langle 1, 2\rangle\}$ would be stored as $c(x, y) = \{\langle 1, 2\rangle\}$ *and* $c(y, x) = \{\langle 2, 1\rangle\}$, with the implicit assumption that the second variable was always the one getting pruned. This allowed the constraints to be stored in a `Map<CSPVariable, Map<CSPVariable, BinaryArc>>`, meaning that we would index the arcs by the "origin" and then by the "destination" variable. This allowed for fast accesment of the relevant constrains and removed the ambiguity over which domain would have to be pruned, at the cost of extra storage space. It is important to note however that this does not imply extra work in upkeep of the domains since support is bi-directional by definition. It also meant that `hashCode` and `equals` methods had to be writen for all the classes in the containment chain, but this proved to be no problem. It also allowed for easy retreaval of all outgoing or incoming arcs, given a variable, which proved very useful in the implementation of MAC to be discussed later.

2.2.1 Pruning

The solver also needed a mechanic to undo the pruning of domains that was done by the arc revision upon backtracking. Since FC nor MAC ever backtracks more than one level at a time, a `Stack` was chosen to record the most recent domain changes. That way, upon backtracking, only the top of the stack had to be restored. The changes were recorded by having the `revise` method simply return a `Set` of all values that were removed from the domain. These values were then stored in a `Map<CSPVariable,Set<Integer>>` and then pushed onto the stack. Since `CSPVariable` already had its `hashCode` and `equals` methods implemented this was fairly simple.

2.3 Heuristics

2.3.1 Dynamic heuristics

To support the use of heuristics, a `PriorityQueue` was used to store the variables. The type of heuristics used was determined by which `Comparator` was used. Here a custom one was implemented for the dynamic smallest domain heuristic. This comparator simply chcked the sizes of the the domains of the two variables. It was chosen to make the dynamic smallest domain heuristic the default option.

2.3.2 Static heuristics

The static heuristics were implemented in much the same way as the dynamic, with the added complexity that a file containg the order has to be supplied at runtime. The heuristic is then read in in the format of `name,order` without any input checking, i.e. the reader assumes that all the variables in the problem are also present in the heuristic file. A Heuristic generator was also implemented, though the format of the heuristic files was designed to be readable and writable for both humans and machienes. In the heuristic generator, code for generating the heuristics for lexographic and maximum degrre variable ordering was also implemented.

2.4 MAC

2.5 Design limitations

2.6 testing

3 Eperical evaluation

3.1 Experiment setup

3.2 Results

3.2.1 Algorithms

3.2.2 Heuristics

3.3 Discussion

4 Conclusion

word count: f