CS4402 - Constraint Programming

Practical 2: Solving

Student number: 130018883 Words: 1434

---

1. A Forward Checking Constraint Solver

The first design choice was in which language to write the solver. After some preliminary work in JavaScript I wasn't happy with how the language suited the task, and so moved over to Python, which I found much more suitable.

I decided to represent the variables and constraints as tuples, as these are immutable, and the list of variables and constraints we are given will not change throughout the problem. I used a domain to represent the domain of possible values, as these would be changed as the solution progressed.

The problem class I am using for this practical is graph colouring. Variable will represent nodes in a graph, a binary constraint will exist containing two variables if there is an edge connecting their respective nodes. I used an integer list to represent distinct colours. For example, the basic problem I built my solver around can be seen in Figure 1 below:

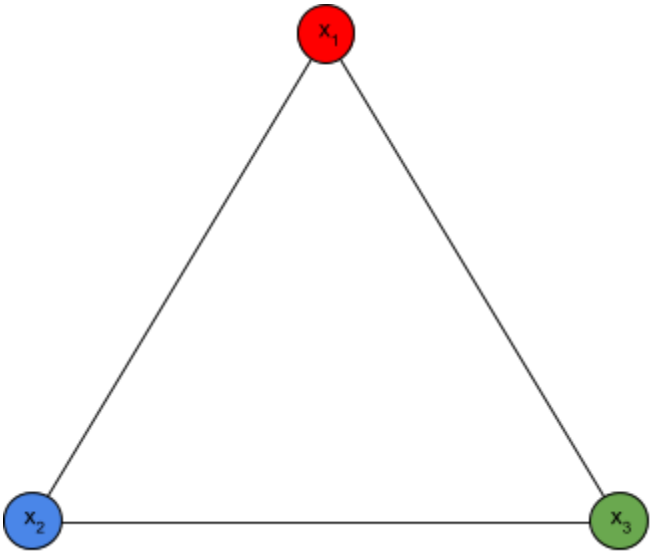

Figure 1

For this problem, the representation can be seen in Figure 2:

Variables:	(x_1, x_2, x_3)
Domain:	[1,2,3]
Constraints:	((x1,x2),(x1,x3),(x2,x3))

Figure 2

I then took the constraints and passed them into a function that returned a tuple of arcs, which introduced direction. I also took the variables and domains, and generated a dictionary called 'dom' of key-value pairs where the key represents the variable, and its associated value is the possible values that this variable may take. In this way values can be removed from each variables domain individually, and the domain of any variable can be looked up by reference.

For the purposes of this problem, the implementation of constraints meant writing a function that takes a single arc and the 'dom' dictionary, and revises the arc based on the inequality constraint to remove any variables that would violate the constraint. I also added a similar function to revise the arc with the equality constraint, to test that the solver can handle alternate constraints.

I also decided to make an ordered list of the variables to act as a queue for value assignment. Although it wasn't entirely necessary, I found it helpful, and looking forward to adding heuristics I decided that re-ordering this queue would be a good way of implementing variable ordering heuristics.

The forward check function takes depth as an argument, and tries to assign a value to the variable who is at that depth in the queue. Firstly it backs up the domain by making a deep copy, so that pruning can be undone if no assignments are possible. For each possible value, it then revises all future arcs incident on the variable in question. if consistent, it calls itself recursively at the next depth or returns if at the end of the queue.

I initially returned True if a solution was found, and False if the function ran out of possible valid assignments. A logical expression on this return value was then used afterwards to call the appropriate function to either display the solution, or a message that none was found.

As I had written each function independently, they did not have access to variables local to the main function, and so I had to alter the forward check function to take the necessary information as parameters.

2. Heuristics

The next step was to add heuristics to assist the search, and hopefully reach solution more quickly. I decided to implement two static heuristics: maximum degree and maximum cardinality, as well as the dynamic, smallest domain first heuristic.

I added some additional arguments to the forward check function to account for the heuristic and heuristic type. If the type is 'static' the heuristic is applied at a depth of 0 only, and if 'dynamic' it is called at each depth level

2.1 Maximum degree

For this heuristic I wrote a function that takes the queue and list of constraints as an argument, and returns a sorted list. Within this I wrote a function that takes a single variable as an argument, and returns its cardinality based on the constraints. This is then used as the key to sort the list. This must also be reversed, as the list is sorted by ascending order of the key for each list item.

2.2 Maximum cardinality

To implement this, I wrote another function that again takes the queue and constraints as arguments and returns a sorted list to be the new queue. Here I define an empty list, and move one element from the queue to the new list. Then, for each item remaining in the queue I calculate it's cardinality, and construct a list of these cardinalities with corresponding indices. I find the indice of the largest entry in this list, remove the corresponding variable from the queue and add it to the new list. This is repeated until the queue is empty. The new, ordered list is then returned.

2.3 Smallest domain first

Here, I wrote a function that takes the queue, the 'dom' dictionary, and the depth as arguments. I make a copy of the queue to be manipulated, and look only at the list from indice 'depth' to the end; i.e. ignore already assigned variables. I construct a list of the size of the domain of each of these variables by looking them up in the 'dom'. The index of the minimum value of this list is then taken to identify which unassigned variable has the smallest domain. I swap this item in the queue with the item at the current depth, so that is immediately taken as the variable to be assigned.

3. Empirical Evaluation

For this section, I decided to take a larger graph with more connections to test the solver. This can be seen in Figure 3 below:

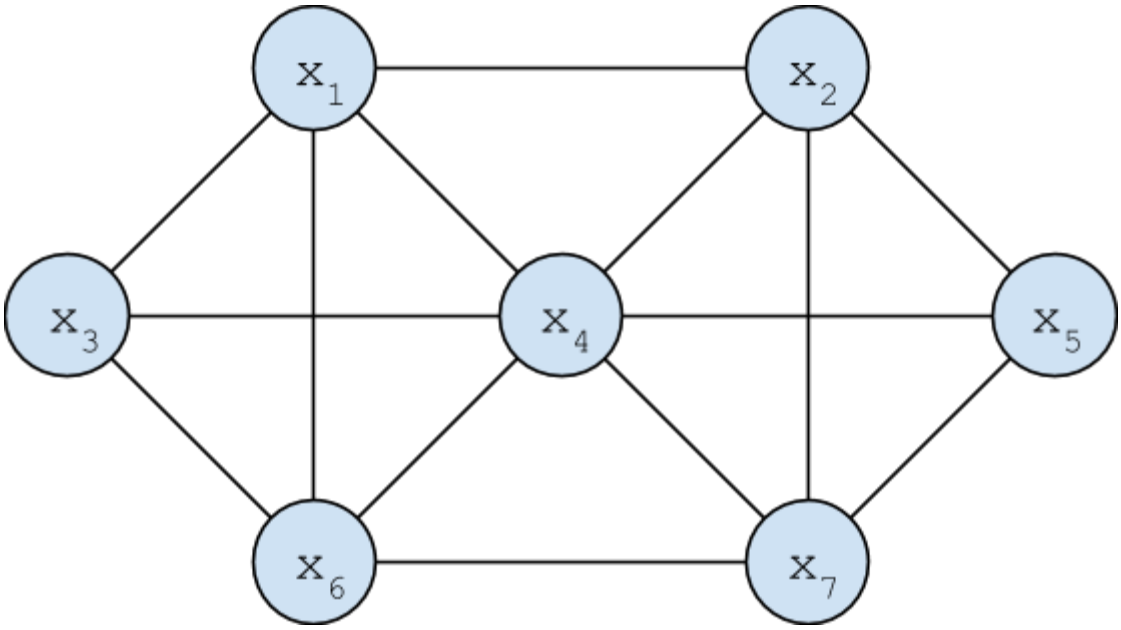

Figure 3

I then initialised the problem with the necessary variables, constraints, and a domain of four values as four colours are needed to colour this graph.

I decided that time taken to execute the function would be the best way to evaluate the heuristic, and so used a Python module that calculates this. I needed to add a utility that converted my forward check function into one with no arguments for this purpose, so added the wrapper function.

For each heuristic I ran the forward check 1,000 times, and took the average time of these trials. The results are displayed in Figure 4:

Heuristic:	Average time take:
Maximum degree	0.000387
Maximum cardinality	0.000540
Smallest domain first	0.000501

Figure 4

As can be seen, the maximum degree heuristic gives the best results, with smallest domain first outperforming maximum cardinality.

Interesting, each heuristic arrives at a different (although equally valid solution). The `dWaySolverTest.py` file returns a single solution for each heuristic, so the solutions can be seen first hand.

4. Extenstion: Two way solver

For this extension, I used the same code as the earlier part, but had to adapt the forward check function so that it used 2-way branching as opposed to d-way branching.

I began by removing one value from the domain of the variable at the current depth, and storing the other options as backup. I then re-assigned to domain of the variable to a singleton list of this value, and checked all arcs containing future variables for consistency. As long as this consistency is held, the forward check function is called on the next depth level, much the same as the d-way solver had done.

If this consistency failed however, then the value taken is no good. At this point I check if the length of the backup list is greater than 0. If so, we have more values to try, so the domain of the variable is re-assigned as the backup list; i.e. all values except the one we originally tried. The forward check function is then called at the same depth level with the updated domains.

However, if the length of the backup list is 0, we've run out of values for this variable, and the solver has failed. Here we return False instead of True to signify the failure.

I performed the same trials as outlined in section 3 for comparison, and the results are listed in Figure 5 below:

Heuristic:	Average time take:
Maximum degree	0.000110
Maximum cardinality	0.000254
Smallest domain first	0.000221

Figure 5

The relationship between the heuristics is the same: maximum degree performs best, followed by smallest domain first, and finally maximum cardinality. However, in each case the time is greatly reduced, highlighting that the 2-way branching offers a much more efficient method of solution.