by Patrick
December 3d, 2019

weeSeepy®: searching for the next solution

Introduction

We might want to enumerate all the solutions to a problem, or maybe find the first hundred solutions. Therefore we might want to find a solution and then ask for the next solution, and so on. This might look something like Listing 1 where in a loop we find a solution (line 10), print it out (lines 11 and 12) then go find another (back to line 10). What follows, in Listing 2, is how this is implemented in weeSeepy.

```
 1  public class TestNextSolution {
 2
 3      public static void main(String[] args){
 4          Problem pb = new Problem("TestNextSolution");
 5
 6          int n = 3;
 7          IntVar[] v = new IntVar[n];
 8          for (int i=0;i<n;i++) v[i] = pb.intVar("v_"+ i,0,1);
 9
10          while (pb.nextSolution()){
11              for (IntVar y : v) System.out.print(" "+ y.getValue());
12              System.out.println();
13          }
14      }
15  }
```

Listing 1: Searching for the next solution in weeSeepy®, an example.

Searching for the next solution

The code in Listing 1 shows how we might search for all solutions to a (trivial) problem. In lines 6 to 8 we create three constrained integer variables each with a domain $\{0, 1\}$, no constraints are posted, and the while loop of lines 10 to 12 searches for all solutions and prints them out (in this case the 8 solutions solutions 000 to 111). The implementation of the method (within class Problem) nextSolution is shown in Listing 2

```
 1      public boolean nextSolution(){
 2          if (firstProbe){
 3              nodes = 0; fails = 0; cpuTime = System.currentTimeMillis();
 4              int n = variables.size();
 5              var   = new IntVar[n+1];
 6              val   = new int[n+1];
 7              Arrays.fill(var,null);
 8              Arrays.fill(val,Integer.MIN_VALUE);
 9              firstProbe = false;
10          }
11          reset();
12          boolean solved = false;
13          try {if (propagate()) probe();}
14          catch (SolutionException e) {solved = true;}
15          cpuTime = System.currentTimeMillis() - cpuTime;
16          return solved;
17      }
18
19      private void probe() {
20          IntVar v = null;
21          if (var[world] == null)
22              v = voh.select();
23          else
24              v = var[world];
25          if (v == null){
26              val[world - 1]++;
27              throw new SolutionException();
28          }
29          var[world] = v;
30          for (int x : v){
31              if (x >= val[world]){
32                  val[world] = x;
33                  pushWorld();
34                  v.instantiate(x);
35                  if (propagate()) probe();
36                  popWorld();
37              }
38          }
39          var[world] = null;
40          val[world] = Integer.MIN_VALUE;
41      }
42
43      public void reset(){
44          while (world > 0) popWorld();
45      }
```

Listing 2: Searching for the next solution in weeSeepy®, implementation.

The idea is as follows. We use a backtracking search to find a our first solution. In our simple problem (Listing 1) that would be v[0] = 0, v[1] = 0, v[2] = 0. This is recorded. When search is then restarted (next time round the while loop) we start by instantiating v[0] to 0, v[1] to 1 and v[2] to the next value in its domain that is greater than or equal to 0+1. The second call to nextValue will have v[0] = 0, v[1] = 1 and demand that the next value for v[2] be greater than or equal to 1+1. Since no such value exists, search will backtrack and attempt an instantiation of v[1] to 1. That is, we store the last branch of the search tree and the next probe takes this branch until hitting the leaf where it it takes the next value in the domain. This is similar to a (simplified) restarting search with nogood recording [2, 1].

Looking at Listing 2, in line 2 the global variable firstProbe is set to true when the problem is created. Array var is of constrained integer variables and val is of integers (lines 5 and 6) and are used to store the branch of search that leads to a solution, where var[i] is the variable selected at depth i and val[i] is the value assigned to that variable. Initially these are set in lines 7 and 8. By setting firstProbe to false in line 9 this ensures that the initialisation steps (lines 3 to 8) are performed once only. In line 11 the problem is reset, via the call to the reset method (lines 43 to 45). Therefore we start at the top of search with all reversible variables set back to their values in world zero.

The search algorithm is then called via lines 13 and 14, a combination of constraint propagation and search, via the call to the probe method (lines 19 to 41). In method probe, the integer world is the depth in search i.e. the length of the current branch of search, where pushWorld increments world and popWorld decrements world (as well as maintaining the reversible variables in the problem). If we are moving down a branch of search for the first time we select a variable dynamically (lines 21 and 22) otherwise we take a forced selection from a previous traversal down this branch (line 23 and 24). However, if no variable is selected we assume that all variables are instantiated, a solution is found, we are now in a world beyond all instantiations and we now set the value of the last variable to be one more than it was (thus its next instantiation must be greater or a backtrack is forced) and a solution exception is thrown (lines 25 to 28).

The variable selected is stored in the path (line 29) and we now search over that variable with values in its domain greater than or equal to the last value tried (lines 30 to 38). However, due to line 26, if the current variable corresponds to the leaf at the end of the branch we are interested only in values greater than that variable's last instantiation. If that last instantiation was the last value in the domain of that variable line 31 will continually fail and we drop through to lines 39 and 40 where the variable's history is cleared and backtracking takes place.

Our example again

Looking at our example (Listing 1) once more, at the top of search the branch history will be var[0] = null, var[1] = null, var[2] = null, val[0] = $-\infty$, val[1] = $-\infty$, val[2] = $-\infty$. Having printed out the first solution 000 the branch history will be var[0] = v[0], var[1] = v[1], var[2] = v[2], val[0] = 0, val[1] = 0, val[2] = 1. The next call to nextSolution terminates with solution 001 and branch history var[0] = v[0], var[1] = v[1], var[2] = v[2], val[0] = 0, val[1] = 0, val[2] = 2. The next solution is then 010 with path history var[0] = v[0], var[1] = v[1], var[2] = v[2], val[0] = 0, val[1] = 1, val[2] = 1.

Outroduction

I am sure that there will be more efficient ways to implement nextSolution. My implementation is simple (I believe), with a small overhead in space (to store pointers to variables and their values) and a small overhead in time (in accessing these).

References

[1] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, and J. Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, pages 20–38, 2019.

[2] G. Glorian, F. Boussemart, J. Lagniez, C. Lecoutre, and B. Mazure. Combining nogoods in restart-based search. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, pages 129–138, 2017.