by Patrick
November 6th, 2019

weeSeepy®: Reification

Introduction

A reified constraint is a constraint that has a boolean variable, call it b, associated with it. If the boolean variable is true the constraint is propagated, if it is false the opposite of the constraint is propagated and if the boolean variable can be both true or false (i.e. b has not yet been instantiated) then neither C nor ¬C are propogated. More succinctly, $b \iff C$. In [7] Jefferson, Moore, Nightingale and Petrie give a *GAC propagation algorithm for reify* as Algorithm 3. This is reproduced below as Algorithm 1.

Algorithm 1: GAC propagation algorithm for reify

```
1 reify(b, C)
2 begin
3     if b = {true, false} then
4         if ¬sat(C) then b ≠ true
5         if ¬sat(¬C) then b ≠ false
6
7     if b then propgate(C)
8     if ¬b then propgate(¬C)
```

There are some interesting things to note with regard to Algorithm 1. First, if we reify constraint C we must have a definition of the *opposite* of C. For example, if $C \equiv x \leq y$ then the opposite of C is $\neg C \equiv x > y$. Second, the boolean variable b must be a constrained variable, at the very least it must be reversible. Also, although not made explicit, at line 6 of Algorithm 1 it could be that $b = \emptyset$. If this is so then an exception will be thrown corresponding to a failure to find any satisfying assignments to the variables in the scope of either C (at line 4) or its opposite (at line 5). And finally, if at line 6 b remains equal to $\{true, false\}$ then neither line 7 nor line 8 will be executed and the domains of the variables in the scope of C and its opposite remain unchanged.

Reification in weeSeepy®

The reification implemented maintains a weaker form of consistency than GAC, i.e the level of consistency maintained is that defined within the propagate method of the constraint that is reified. The implementation of reification is described in the code segment in Listing 1.

```
1   public void reify(){
2       b = new IntVar("b",0,1,pb);
3       b.constraints.add(this);
4       isReified = true;
5       isOpposite = false;
6       opposite = this.makeOpposite();
7       opposite.b = b;
8       opposite.isReified = true;
9       opposite.isOpposite = true;
10      opposite.opposite = this;
11  }
12
13  boolean localRevision(){
14      boolean satisfied = true;
15      pb.propagationOn = false;
16       pb.pushWorld();
17        try {propagate();}
18        catch (CPException e) {satisfied = false;}
19       pb.popWorld();
20      pb.propagationOn = true;
21      return satisfied;
22  }
23
24  void reifiedPropagation(){
25      Constraint c = isOpposite ? opposite : this;
26      if (b.contains(0) && b.contains(1)){
27          if (!c.localRevision()) b.remove(1);
28          if (!c.opposite.localRevision()) b.remove(0);
29      }
30      if (b.isInstantiated() && b.getValue() == 1) c.propagate();
31      if (b.isInstantiated() && b.getValue() == 0) c.opposite.propagate();
32  }
```

Listing 1: Reification in weeSeepy®.

Given a constraint C, that constraint can be refied via a call to C.reify(), therefore in the code segment of lines 1 to 11 (in Listing 1) constraint C is refered to as **this**. In line 2 the constraint integer variable b corresponds to the reification variable of the same name in Algorithm 1, and is associated with this constraint, where the value 1 corresponds to true and 0 to false. This constraint is then added to the list of constraints associated with the reifying variable b (line 3) such that if and when the domain of b changes the constraints that have that variable in its scope are revised[1]. This constraint is marked as reified (line 4) and as not being an opposite (line 5). Its opposite is then created (line 6) where the makeOpposite method is defined for this constraint. Lines 7 to 10 are essentially the flip side of lines 2 to 6: the reification variable b is also associated with the opposite of this (line 7), the opposite is marked as reified (line 8), as being an opposite (line 9, the converse of line 5) and finally (line 10) the opposite of the opposite is of course this!

In Algorithm 1, lines 4 and 5, the sat method determines if the varaibles in the scope of a constraint can be instantiated such that the constraint is satisfied, whereas in our implementation we test if the constraint can be revised in isolation from all other constraints. We call this a local revision, and is realised by the localRevision method (the code in lines 13 to 22). Essentially the method starts by suppressing the chain of propagations that may result from domain reductions of the variables in the scope of this constraint (line 15). We then enter a new world, via the call to pushWorld (line 16). The constraint is then propagated in isolation (line 17) and if this results in a domain wipeout the resulting variable $satisfied$ is set to false (line 18). We then undo any effects of the call to propagate (made in line 17) via the call to popWorld (line 19). We then turn propagation back on (line 20) and finaly return the result of the local revision (line 21). The localRevision method is similar in spirit to forward checking [5][2] and the concept of worlds and disconnection from the revision queue is similar in principal to the local computation spaces in [9].

[1] However, a constraint can only be reified once and a reification variable can be associated with only one constraint.

[2] ... an observation made by Ciaran McCreesh.

The reifiedPropagation method in lines 24 to 32 of Listing 1 is the realisation of Algorithm 1. Line 25 determines if this constraint is an opposite constraint or not. If it is an opposite constraint then we set c to be the opposite of this, otherwise we set c to be this constraint. This is done to simplify the code in lines 26 to 32. Lines 26 to 29 correspond to lines 3 to 6 in the Algorithm, and lines 30 and 31 correspond to lines 7 and 8 in the Algorithm. Note that an exception can be thrown if b becomes empty (at line 29) or if propagation in line 30 or 31 results in a domain wipeout.

Half reification

Our implementation is full reification, i.e. $b \iff C$. However, a weaker alternative is half-reification, where we have $b \implies C$ [3]. We can do this by via the Null constraint. The Null constraint does nothing, i.e. its propagtion methods are empty. Therefore we override the makeOpposite method of the constraint C and set its opposite to be $newNull(pb)$ [3].

Or

With reification we can now implement disjuncion, i.e. a constraint such as Or(C1,C2) where either C1 is true, C2 is true or both are true. The implementation of this is shown in the code segment of Listing 2. The simple case of $C1 \vee C2$ is given in lines 1 to 9. First, both C1 and C2 are reified (lines 3 and 4), the constraints are posted into the problem (lines 6 and 7) along with the summation constraint of line 8. That is, line 8 ensure that $C1.b + C2.b \geq 1$. The more general case is given in lines 11 to 24 where an array of constraints C is passed as an argument, and the summation constraint is posted in line 23[4].

```
1   public Or(Problem pb, Constraint C1, Constraint C2){
2       super(pb);
3       C1.reify();
4       C2.reify();
5       name = "Or";
6       pb.post(C1);
7       pb.post(C2);
8       pb.post(new SumGEQ(pb,new int[]{1,1},new IntVar[]{C1.b,C2.b},1));
9   }
10
11  public Or(Problem pb, Constraint[] C){
12      super(pb);
13      int n = C.length;
14      int[] a = new int[n];
15      IntVar[] b = new IntVar[n];
16      for (int i=0;i<n;i++){
17          C[i].reify();
18          pb.post(C[i]);
19          a[i] = 1;
20          b[i] = C[i].b;
21      }
22      name = "Or";
23      pb.post(new SumGEQ(pb,a,b,1));
24  }
```

Listing 2: Disjunction in weeSeepy®.

[3]The code for the Null constraint is trivial and is not shown.

[4]Our summation constraint is based on the ideas in [6] and will be described in a later note.

Two tests for Or

We now present two tests for our Or constraint. Our first test is simple and shows how propagation behaves on a disjunction. Our second example is taken from CSPLib [1].

Listing 3 The variable x can take values in the range 0 to 9 (line 6) and constraints are imposed (lines 8 and 9) such that $x < 3 \vee x > 4$. Lines 12 and 13 show the problem, and this is illustrated in Figure 1. Above the first line of stars we see the variables of the problem: the array x, the constants 3 and 4 and the reification variables (both called b). We then see the two reified constraints and the summation constraint that produces the or. The constraints are then propagated and the problem shown again (lines 15 and 17) and as Figure 1 shows, nothing changes. We then remove all values below 3, call propagation again and show the problem (lines 19 to 22). As illustrated, the domain of x is now in the range 5 to 9 inclusive (and is in a strange order due to the sparse set implementation) and the reification variables are both instantiated.

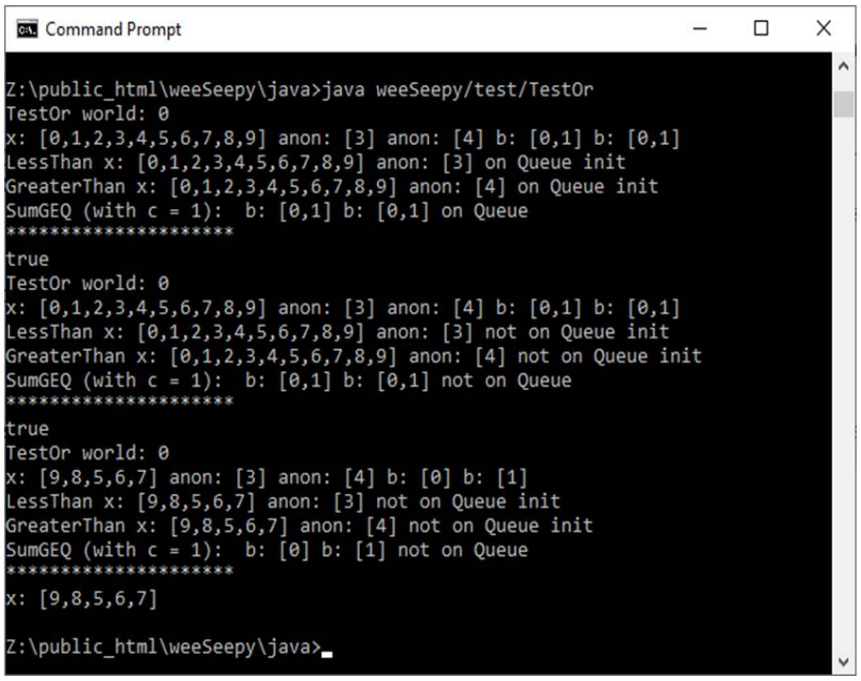

Figure 1: Command line output of TestOr.java.

```
1  public class TestOr {
2
3      public static void main(String[] args){
4          Problem pb = new Problem("TestOr");
5
6          IntVar x = pb.intVar("x",0,9);
7
8          Constraint c1 = new LessThan(pb,x,3);
9          Constraint c2 = new GreaterThan(pb,x,4);
10         new Or(pb,c1,c2);
11
12         pb.show();
13         System.out.println("********************");
14
15         System.out.println(pb.propagate());
16         pb.show();
17         System.out.println("********************");
18
19         x.removeBelow(3);
20         System.out.println(pb.propagate());
21         pb.show();
22         System.out.println("********************");
23         System.out.println(x);
24
25     }
26 }
```

Listing 3: A test for Or in weeSeepy®.

Listing 4 Our next test comes from CSPLib problem number prob009 [10], a problem also used in [8] to investigate general reification. In its pure form, we are given a large square and a number of smaller squares. The smaller squares have to be placed inside the larger square such none of the smaller squares overlap. In our problem we allow the containg shape to be a rectangle. This was done so that we can test that our model can determine when one square must be beside (or above) another. For example, if we have a 7 by 5 rectangle and two squares of size 3, propagation should determine that the squares must be placed beside one another, rather than one on top of the other.

A sample problem instances, named p01.txt, is shown in Figure 2, where we have a rectangullar pallet of size 11 by 9 rectangle and 11 squares to be packed into that rectangle. The squares are of sizes 2, 3, 1, 2, 1, 5, 2, 1, 6, 2 and 1 respectively. In the same figure we have a solution to this problem. The solution gives for each of the 11 squares the x and y coordinates of its bottom left corner and the size of that square. Also shown are measures of the search effort: nodes is the number of variable instantiations, fails is the number of backtracks and cpuTime is milliseconds in search. A picture of this solution is shown in Figure 3. We now present the model.

Listing 4 reads in a problem in lines 5 to 14, such that maxX and maxY (lines 7 nd 8) are the size of the rectangular pallet. In lines 10 to 13 we read in the sizes of the squares to be packed (sizes are held in array size). Line 16 declares a problem and lines 18 and 19 declare our decision variables, the x and y coordinates of the bottom left hand corner for each square. These constrained integer variables are then created in lines 21 to 24. The constraints of the model are created and posted in lines 26 to 34, where we post for every pair of squares (i,j) the disjunction that the i^{th} square is to the left, below, to the right or above the j^{th} square. This is then posted in the Or constraint in line 33.

In lines 41 to 46 we create a variable ordering heuristic (voh) for the problem, where search selects variables with the smallest domain first. This has an unexpected effect. Squares that are large will have small domains in there coordinating variables. Therefore choosing a variable with a small domain in preference to a variable with a larger domain will tend to correspond to a fitting of large squares before smaller squares, i.e. somewhat like a first-fit-decreasing heuristic. In line 48 we perform a backtracking search to find a first solution[5], and if a solution is found it is printed out (lines 50 to 54). The run time

[5]I really must change bt to solve. It's that naming problem again.

statistics are then dispayed (lines 55 and 56). If the variable ordering heuristic is not used the run time increases from about 60 milliseconds to 60 seconds and 727,838 nodes, i.e. it is a thousand times slower.

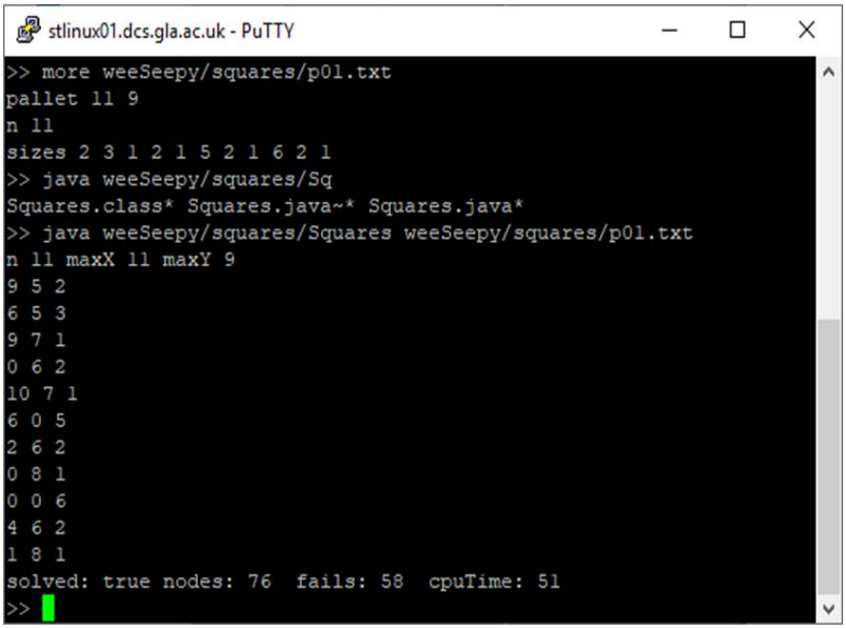

Figure 2: Command line output of Squares.java.

```
1  public class Squares {
2
3      public static void main(String[] args) throws IOException {
4
5          Scanner sc = new Scanner(new File(args[0]));
6          sc.next(); // pallet
7          int maxX = sc.nextInt();
8          int maxY = sc.nextInt();
9          sc.next(); // n
10         int n = sc.nextInt();
11         sc.next(); // sizes
12         int[] size = new int[n];
13         for (int i=0;i<n;i++) size[i] = sc.nextInt();
14         sc.close();
15
16         Problem pb = new Problem("Squares");
17
18         IntVar[] x = new IntVar[n];
19         IntVar[] y = new IntVar[n];
20
21         for (int i=0;i<n;i++){
22             x[i] = pb.intVar("x_" + i,0,maxX-size[i]);
23             y[i] = pb.intVar("y_" + i,0,maxY-size[i]);
24         }
25
26         for (int i=0;i<n-1;i++)
27             for (int j=i+1;j<n;j++){
28                 Constraint[] C = new Constraint[4];
29                 C[0] = new GreaterThanOrEqual(pb,x[j],x[i],size[i]); // i left of j
30                 C[1] = new GreaterThanOrEqual(pb,y[j],y[i],size[i]); // i below j
31                 C[2] = new GreaterThanOrEqual(pb,x[i],x[j],size[j]); // i right of j
32                 C[3] = new GreaterThanOrEqual(pb,y[i],y[j],size[j]); // i above j
33                 new Or(pb,C);
34             }
35
36         //
37         // NOTE: the x/y variables for squares of large size will habe relatively small
38         //       domains, consequently the SDF variable ordering heuristic is leaning
39         //       towards the "first fit decreasing" bin packing heuristic!
40         //
41         ArrayList<IntVar> decVars = new ArrayList<IntVar>();
42         for (int i=0;i<n;i++){
43             decVars.add(x[i]);
44             decVars.add(y[i]);
45         }
46         pb.voh = new SDF(decVars); // variable ordering heuristic is smallest domain first
47
48         boolean satisfied = pb.bt();
49
50         if (satisfied){
51             System.out.println("n "+ n +" maxX "+ maxX +" maxY "+ maxY);
52             for (int i=0;i<n;i++)
53                 System.out.println(x[i].getValue() +" "+ y[i].getValue() +" "+ size[i]);
54         }
55         System.out.println("solved: "+ satisfied +" nodes: "+ pb.nodes
56                            +"  fails: "+ pb.fails +"  cpuTime: "+ pb.cpuTime);
57     }
58
59 }
```

Listing 4: (im)Perfect squares in weeSeepy®.

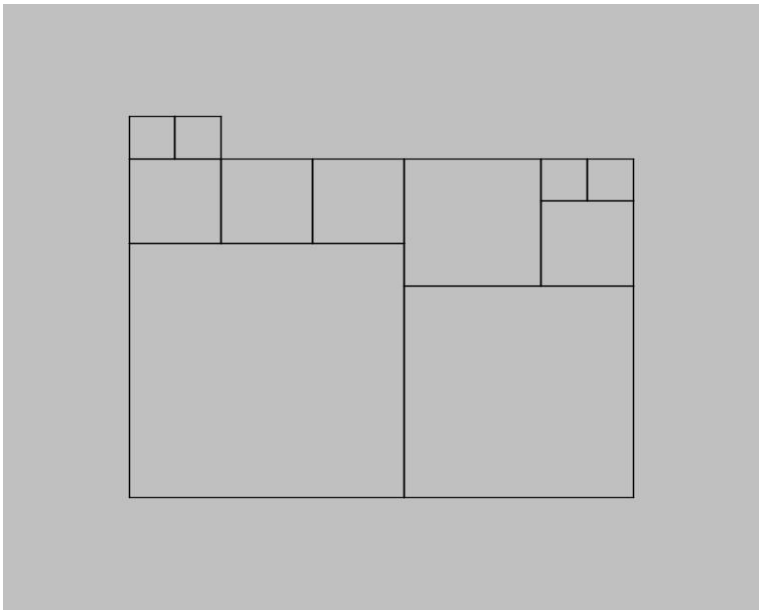

Figure 3: A packing of the squares in problem p01 into an 11 by 9 rectangle.

Outroduction

Via reification we have implemented disjunction. From this we can get implication: since $P \implies Q \equiv \neg P \vee Q$ implication can be realised by reifing the opposite of P and then creating an Or constraint. And of course, given implication we can engineer the biconditional.

Having to produce an opposite for a constraint might be used as an excuse for not reifing a constraint. However, Fages and Soliman [2] *"... show that several global constraints that were believed to be hard to negate can in fact be efficiently negated, and that entailment and disentailment can be efficiently tested."*.

In our first simple test of Or (Listing 3), we have $x \in \{0, ..., 9\}$ and $x < 3 \vee x > 4$. From the get go we would expect that propagation alone should deduce that $x \in \{0, 2, 5, ..., 9\}$. This is *constructive disjunction*. Given $C1 \vee C2$ construcive disjunction captures the values disallowed if $C1$ is propagated and the values disallowed if $C2$ is propagated and removes the intersection of these disallowed values from the shared variables. This tends to be expensive to do and rarely cost effective. However, recent work by Gotlieb, Marijan and Spieker [4] claims that this can ineed be done efficiently.

Although the actual code for reification is relatively small, it has been a challenge. Testing and debugging the Or constraint was very difficult[6], a fair amount of infrastructure had to be put in place to allow Or such as the SumGEQ constraint, implementation of the opposite of each constraint and working out how to turn off propagation in weeSeepy!

There is one thing that begs to be done and that is to compare weeSeepy's performance against choco. The model in Listing 4 should be coded up in choco and a quick comparison made.

References

[1] CSPLib: A problem library for constraints. http://www.csplib.org, 1999.

[6] Many thanks to James Trimble for helping me with this.

[2] F. Fages and S. Soliman. Reifying global constraints. HAL archives-ouvertes, Research Report 8084, 2012.

[3] T. Feydy, Z. Somogyi, and P. J. Stuckey. Half reification and flattening. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, pages 286–301, 2011.

[4] A. Gotlieb, D. Marijan, and H. Spieker. Stratified constructive disjunction and negation in constraint programming. *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, Nov 2018.

[5] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980.

[6] W. Harvey and J. Schimpf. Bounds consistency techniques for long linear constraints. In *In Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, pages 39–46, 2002.

[7] C. Jefferson, N. C. A. Moore, P. Nightingale, and K. E. Petrie. Implementing logical connectives in constraint programming. *Artif. Intell.*, 174(16-17):1407–1429, 2010.

[8] M. Z. Lagerkvist and C. Schulte. Propagator groups. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, pages 524–538, 2009.

[9] C. Schulte. Programming deep concurrent constraint combinators. In *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000, Boston, MA, USA, January 2000, Proceedings*, pages 215–229, 2000.

[10] H. Simonis. CSPLib problem 009: Perfect square placement.