by Patrick
November 27th, 2019

weeSeepy®: greater than, sum and the max constraint

Introduction

I now present a number of constraints, showing their actual implementation in weeSeepy.

GreaterThan ($x > y + c$)

Listing 1 shows the implementation of the constraint $x > y + c$ where x and y are constrained integer variables and c is a natural number ($c \in \mathbb{N}$).

```
1  public class GreaterThan extends BinaryConstraint {
2
3      int c;
4
5      public GreaterThan(Problem pb,IntVar x,IntVar y,int c){
6          super(pb,x,y);
7          name = "GreaterThan";
8          this.c = c;
9          assert c >= 0 : "c must be positive " + this;
10     }
11
12     public GreaterThan(Problem pb,IntVar x,IntVar y){
13         super(pb,x,y);
14         name = "GreaterThan";
15         c = 0;
16     }
17
18     public GreaterThan(Problem pb,IntVar x,int y){
19         super(pb,x,new IntVar("anon",y,y,pb));
20         name = "GreaterThan";
21         c = 0;
22     }
23
24     public void propagate(){
25         x.removeBelow(y.min()+1+c);
26         y.removeAbove(x.max()-1-c);
27     }
28
29     public void propagateInit(){propagate();}
30     public void propagateUPB(IntVar v){propagate();}
31     public void propagateLWB(IntVar v){propagate();}
32     public void propagateInstantiate(IntVar v){propagate();}
33     public void propagateRemoval(IntVar v, int value){}
34
35     public boolean isEntailed(){return x.min() > y.max()+c;}
36
37     public Constraint makeOpposite(){return new LessThanOrEqual(pb,x,y,c);}
38
39     public boolean isLegal(){return x.max() > y.max() + c && x.min() > y.min() + c;}
40 }
```

Listing 1: Greater Than constraint in weeSeepy®.

The constraint $x > y + c$ is created via the constructor in lines 5 to 8, with the assertion in line 9 that c must be a natural number. Llines 12 to 16 construct the constraint $x > y$. For the constraint $x > y$, where x is a constrained integer variable and y is a constant, we use the constructor in lines 18 to 22

where in line 19 an anonymous IntVar is created with a domain that is the singleton $\{y\}$. The propagate method (lines 24 to 27) increases the lower bound of x to be at most $\underline{y}+c+1$ and the upper bound of y to be at least $\overline{x}-c-1$. This is an $O(1)$ operation, consequently the other propagation methods in lines 29 to 32 call this simple propagate method. The removal of a value between the bounds of a variable has no effect (line 33). The constraint is entailed if $\underline{x} > \overline{y}+c$ (line 35), the opposite of $x > y+c$ is $x \leq y+c$ (line 37) and the constraint has been legally revised if $\overline{x} > \overline{y}+c \ \wedge \ \underline{x} > \underline{y}+c$ (line 39).

SumGEQ ($\sum a_i.x_i \geq c$)

The constraint implemented is a simplified version of that described in [2] and is simplified in that it allows only positive coefficients and fine-grained propagation is not considered, i.e. reacting to changes to individual variables is ignored. The implementation of the constraint is presented in Listing 2.

```
1  public class SumGEQ extends Constraint {
2
3      int c, n;
4      int[] a;
5      IntVar[] x;
6
7      public SumGEQ(Problem pb, int[] a, IntVar[] x, int c){
8          super(pb);
9          this.c = c;
10         this.a = a;
11         this.x = x;
12         this.n = x.length;
13         name   = "SumGEQ (with c = "+ c +"): ";
14         addVariables(x);
15         assert allPositive() : "all coeffs must be > 0 "+ this;
16     }
17
18     public void propagate(){
19         int sumOfMaxs = 0;
20         for (int i=0;i<n;i++) sumOfMaxs = sumOfMaxs + a[i] * x[i].max();
21         int slack = c - sumOfMaxs;
22         for (int j=0;j<n;j++){
23             int S_j = slack + a[j] * x[j].max(); // max slack on a[j]*x[j].max()
24             x[j].removeBelow(S_j/a[j]); // can have no less than this!
25         }
26     }
27
28     public void propagateInit(){propagate();}
29     public void propagateUPB(IntVar x){propagate();}
30     public void propagateLWB(IntVar x){propagate();}
31     public void propagateInstantiate(IntVar x){propagate();}
32     public void propagateRemoval(IntVar x, int value){}
33
34     public boolean isEntailed(){return false;}
35
36     public Constraint makeOpposite(){return new Null(pb);}
37
38     public boolean isLegal(){return true;}
39
40     private boolean allPositive(){
41         for (int b : a)
42             if (b<=0) return false;
43         return true;
44     }
45 }
```

Listing 2: SumGEQ constraint in weeSeepy®.

Informally, the constraint reasoning is as follows. Assume that you are out to lunch with (for the sake of exposition) three of your generous friends at a restaurant with a fixed price menu and that each of your

friends declares how much money they have in their pockets. Assume the total bill will be T_{bill}. You can then work out the very least amount that you will have to pay. Assume that each friend of yours pays as much as possible (maybe even more than T_{bill}). Therefore we sum up the amount of money that your friends have and we call that T_{sum}. In the event of maximum generosity you will have to pay no more than $max(0, T_{bill} - T_{sum})$. This can then be done in turn for each of your friends.

In Listing 2 lines 7 to 16 we have the constructor, where the array a is of positive integers (line 15 and lines 40 to 44) and x is an array of constrained integer variables. The propagate method (lines 18 to 26) is used in all of the methods (lines 28 to 31) except propagateRemoval (line 32) which does nothing. The propagate method starts by summing up the maximum amount that can be made, i.e. $\sum a[i].\overline{x[i]}$ (lines 19 and 20). We then compute *the slack* (line 21) and this is sometimes called *the float*. In the for loop (lines 22 to 25) we compute the slack for a variable (line 23) and then update the lower bound of that variable accordingly (line 24). In lines 34 to 38 we make the minimal amount of coding effort to implement the constraint.

```
1  public void propagate(){
2      int sumOfMins = 0;
3      for (int i=0;i<n;i++) sumOfMins = sumOfMins + a[i] * x[i].min();
4      int slack = c - sumOfMins;
5      for (int j=0;j<n;j++){
6          int S_j = slack + a[j] * x[j].min(); // max slack on a[j]*x[j].min()
7          x[j].removeAbove(S_j/a[j]); // can have no more than this!
8      }
9  }
```

Listing 3: SumLEQ constraint propagation method in weeSeepy®.

Obviously, there is much more that can be made of this constraint, as specified in [2], in particular making it fine-grained and incremental and allowing for negative coefficients. This would be a nice student exercise or part of a larger student project. The SumLEQ constraint ($\sum a_i.x_i \leq c$) propagation method is shown in Listing 3 and is symmetric to SumGEQ. SumEQ ($\sum a_i.x_i = c$) is essentially a combination of the two, i.e. $\sum a_i.x_i = c \equiv \sum a_i.x_i \geq c \ \wedge \ \sum a_i.x_i \leq c$.

Max ($x = max(v_0, \ldots, v_{n-1})$)

The maximum constraint can logically be thought of as

$$x = max(v_0, \ldots, v_{n-1}) \ \equiv \ \bigwedge x \geq v_i \wedge \bigvee x = v_i$$

where all variables are constrained integer variables. Due to our familiarity with the max function in conventional programming languages, it is easy to forget that in this case *things work both ways.* That is, if the domain changes on the left hand side of the relation, domains might also change on the right hand side. The implementation of this constraint is given in Listings 4, 5 and 6. The constraint maintains domain consistency on the variable x and bound consistency on the variables in the array v.

```
1  public class Max extends Constraint {
2
3      IntVar x;
4      IntVar[] v;
5      ReversibleInt  indexOfLargest;
6      ReversibleInt  largestMax;
7      ReversibleInt  largestInst;
8      ReversibleInt  largestMin;
9      ReversibleBool entailed;
10     ReversibleInt[] support; // support[x] = i where v[i] contains x
11     int n, m, xLwb, xUpb;
12
13     public Max(Problem pb,IntVar x,IntVar[] v){
14         super(pb);
15         this.x           = x;
16         this.v           = v;
17         this.n           = v.length;
18         name             = "Max";
19         largestInst      = new ReversibleInt(pb,Integer.MIN_VALUE);
20         largestMin       = new ReversibleInt(pb,Integer.MIN_VALUE);
21         largestMax  = new ReversibleInt(pb,Integer.MIN_VALUE);
22         indexOfLargest   = new ReversibleInt(pb,-1);
23         entailed         = new ReversibleBool(pb,false);
24         xLwb             = x.getLwb();
25         xUpb             = x.getUpb();
26         m                = xUpb - xLwb + 1;
27         support          = new ReversibleInt[m];
28
29         addVariables(v);
30         addVariable(x);
31         for (int i=0;i<m;i++) support[i] = new ReversibleInt(pb,0);
32     }
```

Listing 4: Max (part 1) constraint in weeSeepy®.

In Listing 4 we have the state variables for the constraint. A number of reversible integers are used (lines 5 to 10): indexOfLargest (line 5) gives the location of the variable in array v that has the largest maximum value and that value is stored in largestMax (line 6), largestInst (line 7) is the largest value assigned to any of the instantiated variables in the array v, largestMin (line 8) is the value of the largest minimum of the variables in v, support is an array of reversible integers such that support[val] gives the index of the first variable in v that has in its domain the value val (if we assume that xLwb is zero) and this supports the value val in the domain of variable x. Since we do not assume that xLwb is zero, more correctly support[val - xLwb] is the index of the first supporting variable in v that contains the value val. Initially support[i] is zero. If val is unsupported, support[val - xLwb] will be set to n, and if at some time the value y is removed from v[support[y - xLwb]] we then search for a support on the right of that location. In some regards this is similar to the approach used in AC2001 [3, 1]. In line 11, n is the number of variables in v, m is the number of values in the domain of variable x, xLwb is the lower bound of variable x when that variable was created and xUpb is the upper bound of x at the time of creating variable x (and these are used as offsets, see above). Lines 13 to 32 give the constructor for the constraint.

```
1   public void propagate(){
2       findLargestMax();
3       findLargestMin();
4       x.removeAbove(largestMax.getValue());
5       x.removeBelow(largestMin.getValue());
6       for (Integer val : x)
7           if (!supported(val)) x.remove(val);
8       for (IntVar y : v) y.removeAbove(x.max());
9   }
10
11  public void propagateInit(){propagate();}
12
13  public void propagateRemoval(IntVar y, int val){
14      if (!y.equals(x) && !supported(val)) x.remove(val);
15  }
16
17  public void propagateUPB(IntVar y){
18      if (y.equals(x)){
19          for (IntVar z : v) z.removeAbove(x.max());
20          findLargestMax();
21      }
22      if (!y.equals(x)){
23          if (y.equals(v[indexOfLargest.getValue()])){
24              findLargestMax();
25              x.removeAbove(largestMax.getValue());
26          }
27          for (int val : x)
28              if (val > y.max() && !supported(val)) x.remove(val);
29      }
30  }
31
32  public void propagateLWB(IntVar y){
33      if (!y.equals(x)) x.removeBelow(y.min());
34  }
35
36  public void propagateInstantiate(IntVar y){
37      if (y.equals(x)){
38          if (x.max() < largestMax.getValue()){
39              for (IntVar w : v) w.removeAbove(x.getValue());
40              findLargestMax();
41          }
42          if (uniqueLargest()){
43              v[indexOfLargest.getValue()].instantiate(x.getValue());
44              largestInst.setValue(x.getValue());
45          }
46      }
47      if (!y.equals(x)){
48          x.removeBelow(y.getValue());
49          if (y.equals(v[indexOfLargest.getValue()]) && y.getValue()<largestMax.getValue()){
50              findLargestMax();
51              x.removeAbove(largestMax.getValue());
52          }
53          largestInst.setValue(Math.max(largestInst.getValue(),y.getValue()));
54          for (int val : x) if (!supported(val)) x.remove(val);
55      }
56      entailed.setValue(x.isInstantiated() && x.getValue() == largestInst.getValue());
57  }
```

Listing 5: Max (part 2) constraint in weeSeepy®.

Listing 5 gives the propagation methods for the constraint and Listing 6 gives the utilities for those methods. The first method, propagate (lines 1 to 9) is called when the constraint is first revised (and method propagateInit (line 11) is called). We find the largest maximum and largest minimum in v (lines 2 and 3) by calling the methods in Listing 6 (lines 13 to 23 and lines 25 to 28). The method findLargest scans the array v to find the index and value of the variable in v with the largest maximum and this is stored in the reversible variables largestMax and indexOfLargestMax (Listing 6 lines 21 and 22). The method findLargestMin (Listing 6 lines 25 to 28) finds the value of the largest minimum in v and stores this in the reversible variable largestMin. The propagate method then removes from x all values greater

than largestMax and all values below largestMin (lines 4 and 5), i.e. variable x is topped and tailed. We then remove from the domain of x all values that have no support in v (lines 6 and 7) via the call to the method supported (defined in Listing 6 lines 37 to 45). The supported method searches for a variable in v that contains the value val, i.e. a variable that supports the potential instantiation of x with the value val. The search does not start at v[0] but at the most recent support for that value and that is support[val - xLwb] (Listing 6 line 38). If a support is found (Listing 6 line 39) the index of that support is saved (Listing 6 line 40) and the method returns true otherwise no support was found and support[val - xLwb] is set to n (Listing 6 line 43) and false is returned. Therefore lines 6 and 7 of Listing 5 achieve domain consistency for x. In line 8 we remove from the domain of variables in v all values greater than the largest value in the domain of x (and there must exist some variable in v that contains that largest value otherwise lines 6 to 7 would have removed it from the domain of x).

```
1   public boolean isEntailed(){
2       return x.isInstantiated() && x.getValue() == largestInst.getValue();
3   }
4
5   public Constraint makeOpposite(){return new Null(pb);}
6
7   public boolean isLegal(){
8       for (int val : x)
9           if (!supported(val)) return false;
10      return x.max() == largestMax.getValue();
11  }
12
13  void findLargestMax(){
14      int largest = Integer.MIN_VALUE;
15      int index   = 0;
16      for (int i=0;i<n;i++)
17          if (v[i].max() > largest){
18              largest = v[i].max();
19              index = i;
20          }
21      largestMax.setValue(largest);
22      indexOfLargest.setValue(index);
23  }
24
25  void findLargestMin(){
26      for (int i=0;i<n;i++)
27          largestMin.setValue(Math.max(largestMin.getValue(),v[i].min()));
28  }
29
30  boolean uniqueLargest(){
31      int largest = largestMax.getValue();
32      for (int i=indexOfLargest.getValue()+1;i<n;i++)
33          if (v[i].max() == largest) return false;
34      return true;
35  }
36
37  boolean supported(int val){
38      for (int i=support[val - xLwb].getValue();i<n;i++)
39          if (v[i].contains(val)){
40              support[val - xLwb].setValue(i);
41              return true;
42          }
43      support[val - xLwb].setValue(n);
44      return false;
45  }
```

Listing 6: Max (part 3) constraint in weeSeepy®.

The method propagateRemoval (lines 13 to 15) is called when variable y loses a value between its upper and lower bounds. If y is actually the variable x we do nothing, otherwise variable y might be a support for the value val in the domain of x. Consequently if y is not variable x and if that value is now unsupported then it is removed from the domain of x (line 14).

Method propagateUPB (lines 17 to 30) is called when the upper bound of variable y is reduced. If y is in fact variable x (line 18) we remove from all variables in v values that are above the upper bound of x (line19) and then find a new largest maximum in v (line 20). On the other hand, if y is not the variable x (line 22) then it is some variable in v, and if that variable had the largest maximum value (line 23) we then find a new largest maximum (line 24) via a call to the findLargestMax method (defined in Listing 6 lines 13 to 23) and crop the domain of the x variable accordingly (line 25). Since variable y has lost values, if any of those lost values were supports for values in x then those values must be removed from the domain of x (lines 27 and 28).

Method propagateLWB (lines 32 to 34) is called when the lower bound of variable y increases. If the variable y is the variable x then nothing happens. However, if the variable y is one of the variables in v then we can remove from the domain of x all values less than the new lower bound of y, i.e. the new value for x must be at least the new lower bound of y (line 33).

The propagateInstantiate method (lines 36 to 57) is called when the variable y is instantiated. If that variable is in fact the variable x (lines 37 to 46) and x has been instantiated to a value less than its earlier upper bound then we can crop variables in v and must then find a new largest maximum in v (lines 38 to 41). If the largest maximum in v is unique (see Listing 6 lines 30 to 35) then we can enforce the instantiation of that variable (lines 42 to 45). However, if the variable y is some variable in v (line 47) we can remove from x all values below the value assigned to y, as x must take at least that value (line 48). If y is in v and y was the largest variable in v and y is not assigned its largest value (line 49) then we need to find a new largest maximum (line 50) and then crop the domain of x (line 51). We then record the largest instantiation in v (line 53) and remove all unsupported values in x (line 54). Before finishing, the method determines if the constraint is entailed (line 56) i.e. if x is instantiated and the largest variable in v takes the same value.

```
1  public class TestMax {
2
3      public static void main(String[] args){
4          Problem pb = new Problem("TestMax");
5
6          int n = 3;
7          IntVar x = pb.intVar("x",0,2);
8          IntVar[] v = new IntVar[n];
9          for (int i=0;i<n;i++) v[i] = pb.intVar("v_"+ i,0,3);
10         pb.post(new Max(pb,x,v));
11
12         System.out.println(x +" "+ v[0] +" "+ v[1] +" "+ v[2]);
13         System.out.println("**********************");
14
15         pb.propagate();
16         System.out.println(x +" "+ v[0] +" "+ v[1] +" "+ v[2]);
17         System.out.println("**********************");
18
19         v[1].instantiate(1);
20         pb.propagate();
21         System.out.println(x +" "+ v[0] +" "+ v[1] +" "+ v[2]);
22         System.out.println("**********************");
23
24         x.instantiate(1);
25         pb.propagate();
26         System.out.println(x +" "+ v[0] +" "+ v[1] +" "+ v[2]);
27         System.out.println("**********************");
28
29         v[0].remove(1);
30         pb.propagate();
31         System.out.println(x +" "+ v[0] +" "+ v[1] +" "+ v[2]);
32         System.out.println("**********************");
33     }
34 }
```

Listing 7: A test for the max constraint in weeSeepy®.

Testing

Testing the constraints is important and non-trivial. Test cases and small applications have been implemented to do this for all the constraints, along with the embedding of assertions within the code. Many of the weeSeepy test cases have been coded again in choco to allow a back to back comparison. An example of a test for the max constraint is given in Listing 7 and the output from that test is shown in Figure 1. Two additional models were coded up for this small test, one in choco the other using the primitive encoding shown in Listing 8 where lines 1 to 7 replace line 10 in Listing 7. A further test was implemented to enumerate all solutions to a problem that uses the max constraint, again with the two encodings of the max constraint and a choco model.

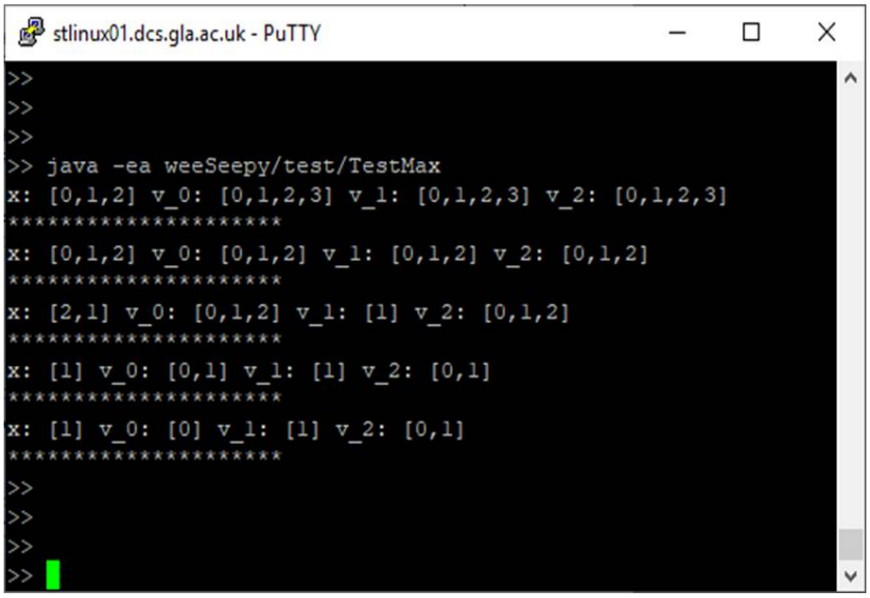

Figure 1: Command line output of TestMax.java.

```
1        Constraint[] c = new Constraint[n];
2        for (int i=0;i<n;i++)
3            c[i] = new Equals(pb,x,v[i]);
4        new Or(pb,c);
5
6        for (int i=0;i<n;i++)
7            pb.post(new GreaterThanOrEqual(pb,x,v[i]));
```

Listing 8: Alternative encoding of max in weeSeepy®.

Outroduction

Often, implementing a constraint requires very little coding. Most often, coding a constraint requires a great amount of care, attention and rigorous testing. The constraints implemented so far are: Equals, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual, Max, NotEquals, Null, Or, SumEQ, SumGEQ and SumLEQ. The Or constraint required reified variables and the summation constraints, along with the opposite of constraints. The Max constraint is an example of a fine-grained implementation of a constraint (and this might be repeated in the sum constraints). What has become obvious, from this experience, is that creating a useful constraint catalog is a big job.

References

[1] C. Bessière and J. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 309–315, 2001.

[2] 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.

[3] Y. Zhang and R. H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pages 316–321, 2001.