by Patrick
November 14th, 2019

weeSeepy®: variables and constraints

Introduction

We now describe the implementation of constrained integer variables and constraints in weeSeepy. These two are closely intertwined and it is difficult to explain them in a linear way. Our description will have forward references, backward references and duplications. Consequently one should expect that when reading this document two passes may be required (similar to a two pass compilation). The approach taken in implementing weeSeepy is rather mechanical and as simple as possible (and no more). That is, one would hope that a second year CS undergraduate would be able to read and understand most of what is going on.

Overview

There are essentially three objects of interest, namely variables, constraints and the problem. The problem is a place-holder for the constraints and variables of the problem being modelled. A problem contains a list of constraints, a list of variables and a queue of constraints that are awaiting revision, where the revision of a constraint is the processing of all the necessary propagation methods on that constraint. The solving process then repeatedly selects a variable from the list of variables, attempts to instantiate that variable and then propagate the consequences of that decision.

Associated with each constraint is a collection of propagation methods that filter the domain of variables in that constraint when a change in those variables takes place. These methods are specialised for the type of changes that are possible. Each constraint has associated with it four *activity* lists. One list contains the variables in the sope of this constraint that have been instantiated, another contains variables in the scope of this constraint that have had the upper bound of their domain reduced, one list for variables that have lower bound increases and one list for variables that have had some value removed from between the lower and upper bounds of its domain. Therefore when a constraint comes to be revised its activity lists tells us exactly what propagation methods to apply to that constraint and the arguments to be used in those propagation methods.

A variable has a domain, a list of constraints acting on this variable and a list of integer deletions that have recently been made to this variable. The only thing that can be done to a variable is to reduce its domain. This can be done by instantiating the variable, increasing the domain's lower bound, decreasing the domain's upper bound or removing some value from between the domain's bounds. When one of these events takes place the variable is then added to the appropriate activity lists of the constraints of that variable, and the constraints of the variable are added to the revision queue.

Therefore the wheel continues to turn: search selects a variable for instantiation, this changes the domain of that variable, consequently the constraints of that variable are added to the revision queue, search then calls propagation, this fires off individual constraint propagations that in turn reduce the domains of variables and adds constraints to the revision queue. Although this might appear chaotic, it might also appear mechanical and rather simple to implement and understand.

IntVar

A constrained integer variable, IntVar, is essentially a place-holder for a reversible domain (implemented as a sparse set, with reversible integers used as pointers to domain limits) and the set of constraints

in which the variable participates (implemented as an ArrayList, and we call this *the constraints of the variable*). When the domain of a variable is reduced the constraints of the variable are then revised, i.e. the constraints of the variable are added to the problem's revision queue pending revision. The only change that can be made to a variable is the reduction of its domain and this takes place via constraint propagation: via removal of a value, instantiation, reduction in upper bound or increase in lower bound. This is shown in Listing 1.

The remove method (lines 1 to 6) removes a value x from the domain of **this** variable and if this changes the domain the result **true** is returned and assigned to the boolean variable changed (line 2). If a change took place and propagation is enabled (this is explained later) then that change is processed via a call to the changeRemoval method (lines 25 to 39). The removal of the value can have one of the four effects.

1. Removal of the value leaves the domain as a singleton (line 26).
2. The value removed is greater than the current upper bound (line 27).
3. The value removed is less than the current lower bound (line 28).
4. The value removed falls between the upper and lower bounds (line 29 to 38).

In case 4 above (line 29 of Listing 1) the value removed is added to the linked list of deletions of that variable (line 30) and this is later used when revising the constraints of this variable. For each constraint c in the constraints of the variable (line 31) that constraint is added to the revision queue (of the problem) if it is not already on it (lines 32 to 35) and this variable is added to the constraint's list of variables that have had case 4 removals from their domains (line 36) and this is later used during revision of the constraint (and is explained in the next section).

Note that the removal of a value (line 2) may leave the domain empty. When this happen, the call to domain.remove(x) will throw an exception. Since this removal will happen via constraint propagation, the exception is caught and processed when revising constraints, as seen in Listing 3. Similarly, an exception can be thrown when a variable is instantiated (line 8) and when its bounds change (lines 14 and 20).

A variable is instantiated with the value x (lines 7 to 11) by removing all values from the domain other than x. Therefore, if an attempt is made to instantiate a variable with a value x that is *not* in its domain the domain becomes empty and an exception is thrown (this all happens at line 8), otherwise line 9 then processes the change due to instantiation via a call to changeInstantiate. In lines 41 to 49 for each of the constraints c of the variable (line 42 to 48), the constraint c is enqueued (lines 43 to 46) and this variable is added to the list of variables of constraint c that have become instantiated (line 47).

A variable's upper bound can be reduced via a call to method removeAbove (lines 13 to 17 of Listing 1). All values above x are removed from the domain (line 20) and if this results in a change (and propagation is enabled) a call is made to changeUPB (lines 51 to 61). First, changeUPB determines if the domain is now a singleton and if so diverts control to the instantiate method (line 52). Otherwise (and similar to before) for each of the constraints c of this variable (lines 54 to 60) the constraint is enqueued and this variable is added to the list of variables on that constraint that have had an upper bound reduction (line 59). Methods removeBelow (lines 19 to 23) and changeLWB (lines 63 to 73) are similar to removeAbove and changeUPB.

```
    public boolean remove(int x){
        boolean changed = domain.remove(x);
        if (changed && pb.propagationOn) changeRemoval(x);
        return changed;
    }

    public boolean instantiate(int x){
        boolean changed = domain.removeAllBut(x);
        if (changed && pb.propagationOn) changeInstantiate();
        return changed;
    }

    public boolean removeAbove(int x){
        boolean changed = domain.removeAbove(x);
        if (changed && pb.propagationOn) changeUPB();
        return changed;
    }

    public boolean removeBelow(int x){
        boolean changed = domain.removeBelow(x);
        if (changed && pb.propagationOn) changeLWB();
        return changed;
    }

    void changeRemoval(int value){
        if (size() == 1) changeInstantiate();
        if (value > max()) changeUPB();
        if (value < min()) changeLWB();
        if (min() < value && value < max()){
            deletions.add(value);
            for (Constraint c : constraints){
                if (!c.onQueue){
                    pb.revisionQueue.add(c);
                    c.onQueue = true;
                }
                if (!c.removals.contains(this)) c.removals.add(this);
            }
        }
    }

    void changeInstantiate(){
        for (Constraint c : constraints){
            if (!c.onQueue){
                pb.revisionQueue.add(c);
                c.onQueue = true;
            }
            if (!c.instantiations.contains(this)) c.instantiations.add(this);
        }
    }

    void changeUPB(){
        if (size() == 1) changeInstantiate();
        else
            for (Constraint c : constraints){
                if (!c.onQueue){
                    pb.revisionQueue.add(c);
                    c.onQueue = true;
                }
                if (!c.newUPB.contains(this)) c.newUPB.add(this);
            }
    }

    void changeLWB(){
        if (size() == 1) changeInstantiate();
        else
            for (Constraint c : constraints){
                if (!c.onQueue){
                    pb.revisionQueue.add(c);
                    c.onQueue = true;
                }
                if (!c.newLWB.contains(this)) c.newLWB.add(this);
            }
    }
```

Listing 1: IntVar class in weeSeepy®.

Constraints

Listing 2 shows part of the implementation of the Constraint class. Actual constraints are then extensions of this super class. Associated with a constraint is a problem (line 3) and a list of variables (line 4). This is often called *the scope of the constraints*, i.e. the variables that participate in the constraint. This list is created when the constraint is created and remains unaltered throughout the life of the constraint. The constraint has four other linked lists (line 5)[1]. Each of these is a list of constrained integer variables. For example, if a variable has its upper bound changed then that variable is added to the newUPB list of this constraint. This information is then exploited when revising this constraint. Similarly when a variable in the scope of the constraint is instantiated it is added to the list instantiations, when a value is removed the variable is added to the list of removals of this constraint and when the lower bound increases the variable is added to the list newLWB.

A constraint must implement the abstract methods. The propagateInit method (line 33) is called the first time the constraint is revised. The propagate method is a general propagation method that ignores any of the detailed changes that have happened to variables in the scope of the constraint. Methods propagateRemoval, propagateUPB, propagateLWB and propagateInstantiate (lines 35 to 41) perform domain filtering when, respectively, a value is removed from a specified variable's domain (line 35), the upper bound of a specified variable has decreased (line 37), the lower bound of a specified variable has increased (line 39), or a specified variable has been instantiated (line 41).

A constraint is entailed when the constraint holds and no further changes to the domain of the variables in the scope of that constraint can result in any other subsequent changes. An example of this might be the less than constraint $x < y$. If the domain of x is $\{1, 2, 3\}$ and the domain of y is $\{4, 5, 6\}$ the constraint is entailed because the less than relation holds across all values in the domains and any change to x or y will not affect this constraint. By default, a safe option might be to define isEntailed to deliver false.

The makeOpposite method is used in reification, where we have a reification variable b for this constraint c and $b \iff c$. Therfore in the event that b is false we need to consider the opposite of c. For example, the opposite of the lessThan constraint would be greaterThanOrEqual. If a constraint is never reified the makeOpposite method can safely deliver null, and if half-reification is an option makeOpposite should deliver the newly created constraint Null.

The method isLegal by default might deliver true, and is only ever called when weeSeepy is run with assertions enabled, i.e. when debugging takes place. When this happens isLegal should check that the desired level of consistency for this constraint holds once it has been fully revised (i.e. after all propagations in its activity lists have been performed). Ideallly, one should implement this method without any concern for its computational cost.

[1] ... referred to as activity lists in our overview

```
1  public abstract class Constraint {
2
3      public Problem pb;
4      public ArrayList<IntVar> variables;
5      public LinkedList<IntVar> removals, newUPB, newLWB, instantiations;
6      public boolean onQueue, init, isReified, posted, isOpposite;
7      public Constraint opposite;
8      public IntVar b; // refication boolean variable
9      String name;
10
11
12     public Constraint(Problem pb){
13         this.pb        = pb;
14         variables      = new ArrayList<IntVar>();
15         removals       = new LinkedList<IntVar>();
16         newUPB         = new LinkedList<IntVar>();
17         newLWB         = new LinkedList<IntVar>();
18         instantiations = new LinkedList<IntVar>();
19         onQueue        = false;
20         init           = true;
21         isReified      = false;
22         posted         = false;
23         opposite       = null;
24         b              = null;
25     }
26
27     //
28     // code deleted: revise, reify, localRevision, reifiedPropagation
29     //
30
31     public abstract void propagate();
32
33     public abstract void propagateInit();
34
35     public abstract void propagateRemoval(IntVar x, int value);
36
37     public abstract void propagateUPB(IntVar x);
38
39     public abstract void propagateLWB(IntVar x);
40
41     public abstract void propagateInstantiate(IntVar x);
42
43     public abstract boolean isEntailed();
44
45     public abstract Constraint makeOpposite();
46
47     public abstract boolean isLegal();
48
49 }
```

Listing 2: Constraint class (part 1) in weeSeepy®.

In part 2 of the definition of Constraint, Listing 3, we have the revise method where a single constraint (this) is revised. That is, all the required propagation methods associated with this constraint are applied. First, if the constraint is reified then reifiedPropagation is performed (lines 2 to 6) and if this constraint is consistent true is returned (line 5) otherwise one of the reification variables has a domain wipeout and false is returned (line 4). Note that when a reified constraint is revised propagation is suppresssed, consequently the effects of this revsion are not propagated through to other variables. This suppression is done via the conditioning of lines 3, 9, 15 and 21 of Listing 1.

If this is the first time the constraint has ever been revised the propagateInit method (lines 7 to 11) is called (and this might create internal state information for the constraint). If consistent then this drops through to line 12. We now apply all of the required propagation methods to this constraint, via lines 12 to 36. Lines 13 to 20 process the removals of values from the domains of variables in the scope of this constraint: in line 14 a variable v is selected and deleted from the list of variables that have had removals, lines 15 to 19 iterates over the values that have been removed from variable v and applies the

propagateRemoval method (line 17) and this may throw an exception due to a domain wipeout so this is caught at line 18. Lines 21 to 25 processes the list of variables that have reductions in their upper bounds, lines 26 to 31 process variables that have had increases in their lower bound and lines 31 to 35 process the instantiation of variables. Within the while loop of lines 12 to 36 the lists of removals, newUPB, newLWB and instantiations can grow as well as shrink, e.g. a call to propagateRemoval might result in some other variable in this constraint being added to the list newUPB. However, isince propagation only removes values from domains this chaotic iteration will terminate in a fixed point or will terminate with some exception being thrown and caught (lines 9, 18, 24, 34).

```
1   public boolean revise(){
2       if (isReified){
3           try {reifiedPropagation();}
4           catch (CPException e) {return false;}
5           return true;
6       }
7       if (init){
8           try {propagateInit();}
9           catch (CPException e) {return false;}
10          init = false;
11      }
12      while (removals.size()>0||newUPB.size()>0||newLWB.size()>0||instantiations.size()>0){
13          while (!removals.isEmpty()){
14              IntVar v = removals.remove();
15              while (!v.deletions.isEmpty()){
16                  int x = v.deletions.remove();
17                  try {propagateRemoval(v,x);}
18                  catch (CPException e) {return false;}
19              }
20          }
21          while (!newUPB.isEmpty()){
22              IntVar v = newUPB.remove();
23              try {propagateUPB(v);}
24              catch (CPException e) {return false;}
25          }
26          while (!newLWB.isEmpty()){
27              IntVar v = newLWB.remove();
28              try {propagateLWB(v);}
29              catch (CPException e) {return false;}
30          }
31          while (!instantiations.isEmpty()){
32              IntVar v = instantiations.remove();
33              try {propagateInstantiate(v);}
34              catch (CPException e) {return false;}
35          }
36      }
37      return true;
38  }
```

Listing 3: Constraint class (part 2) in weeSeepy®.

The Problem

A problem is an object that holds everything together. It is analogous to the Model + Solver in choco. The Problem (pb is a typical name) holds a list of constrained integer variables, a list of all the constraints in the model, a trail used by reversible variables and the revision queue i.e. a queue of constraints that are awaiting revision.

```
1   public Constraint post(Constraint c){
2       if (!c.posted){
3           constraints.add(c);
4           revisionQueue.add(c);
5           for (IntVar x: c.variables)
6               x.constraints.add(c);
7           c.onQueue = true;
8           c.pb = this;
9           c.posted = true;
10      }
11      return c;
12  }
13
14  public boolean propagate(){
15      boolean consistent = true;
16      while (consistent && !revisionQueue.isEmpty()){
17          Constraint c = revisionQueue.peek();
18          consistent = c.revise();
19          if (consistent) {
20              revisionQueue.remove();
21              c.onQueue = false;
22              assert c.isLegal() : c;
23          }
24          if (!consistent) flushRevisionQueue();
25      }
26      assert cleanConstraints() : "End of propagate and constraint queue is not clean";
27      return consistent;
28  }
```

Listing 4: Problem class in weeSeepy®.

In Listing 4 a constraint can be posted into the problem via the post method (lines 1 to 12). The constraint is added to the list of constraints and to the revision queue (lines 3 and 4). Lines 5 and 6 add to each of the variables x in the scope of constraint c the constraint c. Therefore c is now in the list of constraints of variable x (line 6). Note that this is all done conditionally (line 2) such that a constraint can be posted at most once.

Method propagate (lines 14 to 28) is similar in spirit to Mackworth's AC3 algorithm. The constraints in the revisionQueue are revised. In lines 16 to 25, a constraint is taken (but not removed from) the head of the revisionQueue and is revised (lines 17 and 18). If the revision was successful, leaving the constraint consistent, then the constraint is dequeued (lines 20 and 21) and if assertions are enabled line 22 is executed. If the revision resulted in an inconsistency (at line 18) then the revision queue is flushed (line 24). That is, we remove each constraint from revisionQueue, empty each of the constraints' newUPB list, newLWB list, instantiations list and for each variables in the removals list clear out the variables deletions and remove that variable from the removals list (i.e. we clear out the action lists). At line 26 we have again an assertion and that is that all constraints are clean. A constraint is dirty (not clean) if at this point it is on the revision queue or still has updates pending (non-empty action lists).

The search for a solution is done via the solve method attached to the Problem class. This is shown in Listing 5. The algorithm is essentially depth first search (dfs). Method solve (lines 1 to 5) attempts to make the problem consistent by propagating all of the constraints and if this succeeds calls dfs (line 2). In lines 7 to 15 the dfs method uses the problem's variable ordering heuristic to select a variable for instantiation (line 8) and if none are selected then all variables are assumed to be instantiated and a solution exception is thrown (line 9) and this is caught at line 3, with the solve method returning true. Otherwise dfs iterates over the values in the domain of the selected variable x (lines 10 to 15), going into a new world (line 11), instantiating that variable (line 12) and if this results in consistent propagation a recursive call is made to dfs (line 13). On return dfs pops out of this world (line 14). Note that if a solution exception is not thrown by dfs then the execution of line 2 in solve falls through to line 4 and false is returned.

```
1     public boolean solve() {
2         try {if (propagate()) dfs();}
3         catch (SolutionException e){return true;}
4         return false;
5     }
6
7     private void dfs() {
8         IntVar v = voh.select();
9         if (v == null) throw new SolutionException();
10        for (int x : v){
11            pushWorld();
12            v.instantiate(x);
13            if (propagate()) dfs();
14            popWorld();
15        }
16    }
```

Listing 5: First solution search in weeSeepy®.

How it goes

We now have a quick look at an example problem and the implementation of a constraint.

```
1  public class GCol {
2
3      public static void main(String[] args) throws IOException {
4
5          String s = "";
6          Scanner sc = new Scanner(new File(args[0]));
7          while (sc.hasNext() && !s.equals("p")) s = sc.next();
8          sc.next();
9          int n = sc.nextInt();
10         int m = sc.nextInt();
11         int[][] A = new int[n][n];
12         while (sc.hasNext()){
13             s     = sc.next(); // skip "edge"
14             int i = sc.nextInt() - 1;
15             int j = sc.nextInt() - 1;
16             A[i][j] = A[j][i] = 1;
17         }
18         sc.close();
19
20         Problem pb = new Problem("Col");
21
22         int k = Integer.parseInt(args[1]);
23
24         IntVar[] v = new IntVar[n];
25
26         for (int i=0;i<n;i++) v[i] = pb.intVar("v_" + i,1,k);
27
28         for (int i=0;i<n-1;i++)
29             for (int j=i+1;j<n;j++)
30                 if (A[i][j] == 1) pb.post(new NotEquals(pb,v[i],v[j]));
31
32         boolean solved = pb.solve());
33         if (solved)
34            for (int i=0;i<n;i++) System.out.println(v[i]);
35     }
36
37 }
```

Listing 6: Naive graph colouring in weeSeepy®.

Listing 6 is a simple weeSeepy model that naively solves a k colouring problem, where the input graph is in DIMACS format and the number of colours, k, is given on the command line. Lines 5 to 18 read in

the DIMACS graph and produce an adjacency matrix (lines 11 and 16). The number of colours is taken from the command line (line 22). A new problem is created (line 20). The vertices of the graph are represented as constrained integer variables, each with a domain 1 to k (lines 24 and 26). The constraints are then posted such that adjacent variables/vertices take different values/colours (lines 28 to 30). The problem is then solved (line 32) and if this was successful the solution found is printed (lines 33 and 34).

```
1  public class NotEquals extends BinaryConstraint {
2
3      public NotEquals(Problem pb,IntVar x, IntVar y){
4          super(pb,x,y);
5          name = "NotEquals";
6          assert !x.equals(y) : "x and y must be different "+ this;
7      }
8
9      public void propagate(){
10         if (x.isInstantiated()) y.remove(x.getValue());
11         if (y.isInstantiated()) x.remove(y.getValue());
12     }
13
14     public void propagateInit(){propagate();}
15
16     public void propagateRemoval(IntVar v, int value){}
17
18     public void propagateUPB(IntVar v){}
19
20     public void propagateLWB(IntVar v){}
21
22     public void propagateInstantiate(IntVar v){propagate();}
23
24     public boolean isEntailed(){
25         return x.max() < y.min() || x.min() > y.max();
26     }
27
28     public Constraint makeOpposite(){return new Equals(pb,x,y);}
29
30     public boolean isLegal(){
31         return (x.isInstantiated() && y.isInstantiated() && x.getValue() != y.getValue()) ||
32                (x.isInstantiated() && !y.contains(x.getValue())) ||
33                (y.isInstantiated() && !x.contains(y.getValue())) ||
34                (x.size() > 1 && y.size() > 1);
35     }
36 }
```

Listing 7: The not equals to constraint in weeSeepy®.

We now define the not equals constraint used in Listing 6 and this is given in Listing 7. Lines 3 to 7 define the constructor. The super class BinaryConstraint adds a pinch of syntactic sugar, such that the arguments x and y are added to the constraint automatically and an assertion is made that x and y are different. All the abstract methods of Listing 2 are implemented. The constraint is relatively simple, where we need only propagate when one of the variables is instantiated, and since there are only two variables it is cheap to determine if this is so. Consequently the main effort is taken by the propagate method (lines 9 to 12). In line 10, if x has been instantiated then we remove the value assigned to x from the domain of y, and line 11 is its mirror image (y is instantiated). The propagateInt method (line 14) calls propagate as does propagateInstantiate (line 22). When a value is removed between the upper and lower bounds nothing happens (line 16) and the same holds for changes in the bounds (lines 18 and 20). The constraint is entailed if the two domains are disjoint (line 24 to 26) and an incomplete but O(1) test is performed (line 25). The opposite of less than is greater than or equal (line 28) and this constraint is not listed here. The isLegal method tests all cases where the constraint is satisfied and and is consistent.

Outroduction

To restate the goals ... I wanted weSeepy to be simple, easy to implement and understand[2]. I wanted to make the core design decisions in three days and spend the rest of my time testing those decisions. So far, the decisions appear to be good. However, I am sure that weeSeepy is not as efficient as it might have been. But, I hope that it can be used for teaching (explaining how things work under the hood), for student projects (implement new constraints, search algorithms, heuristics, model and solve problems etc) and possibly for research (I have always tried to combine teaching with research). I need to make weeSeepy more widely available and publicise it and I need to prepare some teaching material (lectures) and possibly some exam questions based on weeSeepy.

[2] At present in weeSeepy a word count (wc) shows that 1,834 lines of java code make up weeSeepy, 646 lines are in the test directory, 152 in the graph colouring example and 1,283 lines in the (im)perfect squares directory (although 1,043 lines are taken up in that directory by Robert Sedgewick's StdDraw software).