by Patrick
August 29th, 2019

weeSeepy®: Reversible Domains

Introduction

weeSeepy® is a simple CP toolkit (inspired by choco [4] and minCP [2]) that supports only constrained integer variables (IntVar). At heart, these constrained integer variables are mere wrappers for a domain represented as an enumerated set of integers. To allow backtracking domains must be reversible. That is, search may select a variable and instantiate it to some value and then perform constraint propagation resulting in the removal of values from the domains of other variables. If and when that instantiation is undone we must reverse out the effects of the instantiation, returning values to the domains of variables. Therefore we require two things: (a) an efficient way of representing potentially sparse sets and (b) an efficient way of reversing out change.

Sparse sets

Domains of variables are represented as sparse sets, as described in [1], in [3] and used in miniCP [2]. Figure 1 is an attempt to illustrate the idea of a sparse set realised in java listings that follow.

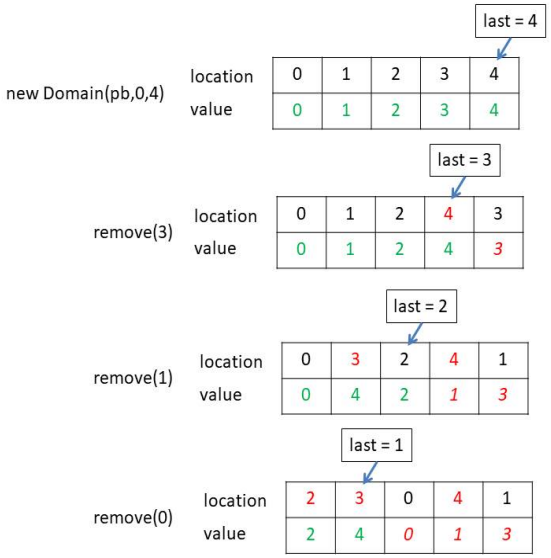

Figure 1: A brief history of the domain {0..4}.

Looking at Figure 1, a domain is represented using two arrays, location and value, where location[i] gives the position of the integer i in the array value. Therefore value[[location[i]] = i. Our figure shows a domain {0..4} initially with location[0] = 0, location[1] = 1 all the way up to location[4] = 4. Also we have value[0] = 0 and so on to value[4] = 4. In addition we have a pointer called last, that gives us the location (the position) of the last value in the domain.

In our picture, we remove the value 3 from the domain. To do this we find the location of the value 3 and this is location[3]. We then swap the value[location[3]] with what was in our last position value[location[last]] and update the locations of the changed values. We then decrement last so it now has the value 3. More generally, when we remove a value from the set we swap it with the last element in the set, update locations and values and then decrement the last pointer. Consequently all values that have been removed from the set are located to the right of last, i.e. value[last+1] to value[n-1] have been removed, and all of the remaining elements of the set reside in value[0] to value[last]. To determine if a value x is in the domain we then have a simple test: $x \in domain \iff location[x] \leq last$ and this is done in $O(1)$. To enumerate the values remaining in the domain we generate the set $\{value[i] \mid 0 \leq i \leq last\}$. Our illustration then continues to show what happens when we remove the value 1 then remove the value 0. In our picture a red entry corresponds to a value removed from the domain, green if it is in the domain.

The Domain of a Variable

We now show how we implement the domain of a variable. Looking at Listing 1, we allow our domains to run from a lower bound lwb to an upperbound upb (line 6), and use the arrays value and location (lines 4 and 5) to hold values and locations of values zero-based. That is, value x is in the domain if and only if location[x-lwb] $\leq$ last. This is realised via function contains in lines 21 to 25 in Listing 2. The pointer to the position of the last value in the domain is held in the reversible variable last (line 7, Listing 1) and this is described later on. Lines 10 to 22 is the constructor for a Domain with respect to a lower bound and an upperbound. A swap function is given in lines 1 to 6 of Listing 2, such that swap(A,i,j) will swap the contents of A[i] with that in A[j].

```
1  public class Domain {
2
3      Problem pb;
4      int[] value;
5      int[] location;
6      int lwb, upb, n;
7      ReversibleInt last, min, max;
8
9
10     public Domain(Problem pb,int lwb, int upb){
11         this.pb   = pb;
12         this.lwb  = lwb;
13         this.upb  = upb;
14         n         = upb - lwb + 1;
15         value     = new int[n];
16         location  = new int[n];
17         last      = new ReversibleInt(pb,n-1);
18         min       = new ReversibleInt(pb,lwb);
19         max       = new ReversibleInt(pb,upb);
20         for (int i=0;i<n;i++) value[i] = i;
21         for (int i=0;i<n;i++) location[i] = i;
22     }
```

Listing 1: The Reversible Domain of an IntVar (part 1).

Function remove, lines 7 to 19 in Listing 2, allows us to remove the value x from the domain. The first thing is to determine if that value is actually in the domain (line 8). In line 9 we get the zero-based value and call it v and the zero-based value in the last position (line 10) and call it w. Since last is a reversible variable we get hold of its value via its getValue method and change it via the setValue method (more of this later). We then swap the contents of value[last.getValue()] with the contents of value[location[v]] (line 11), do the same with the locations of these values (lines 12 and 13) and then downdate the last pointer (line 14). If the domain is now empty an exception is thrown (line 15) and is safely caught

```
1   void swap(int[] array,int i,int j){
2       int temp = array[i];
3       array[i] = array[j];
4       array[j] = temp;
5   }
6
7   boolean remove(int x){
8       if (x < lwb || x > upb || !contains(x)) return false;
9       int v = x - lwb;
10      int w = value[last.getValue()];
11      swap(value,last.getValue(),location[v]);
12      location[w] = location[v];
13      location[v] = last.getValue();
14      last.setValue(last.getValue() - 1);
15      if (isEmpty()) throw new CPException("Domain is empty.");
16      if (x == min.getValue()) min.setValue(getNewMin());
17      if (x == max.getValue()) max.setValue(getNewMax());
18      return true;
19  }
20
21  boolean contains(int x){
22      if (x < lwb || x > upb) return false;
23      int v = x - lwb;
24      return location[v] <= last.getValue();
25  }
```

Listing 2: Removing a value from a domain.

elsewhere (more on that later). It might be that the value x that was removed was the smallest value in the domain and if so we find a new minumum value (line 16) via the method getNewMin (lines 5 to 10 in Listing 5). Method getNewMin linearly scans the location array (lines 7 and 8 in Listing 5) from low to high until it finds an entry that is in the domain (line 8) or falls off the end of the set of remain values (which should not happen and results in the largest integer possible being returned in line 9). Similarly a new maximum may be requested at line 17 and result in a call to getNewMax (Listing 5 lines 14 to 19). Note, that if a value is actually removed the method delivers true (line 18) and will later be used as indication that constraint propagation may be required.

```
1   boolean removeAbove(int x){
2       boolean changed = false;
3       for (int v=max();v>x;v--)
4           changed = remove(v) || changed;
5       if (isEmpty()) throw new CPException("Domain is empty.");
6       return changed;
7   }
```

Listing 3: Removing all values above a given value x.

Frequently we want to remove all values in the domain of a variable that are greater than some value x, for example in a less than or greater than constraint. This is done via the removeAbove method (lines 1 to 7 in Listing 3). In lines 3 and 4 we scan downwards from the largest value in the domain (got from reversible variable max via the method max in line 21 of Listing 5) removing all values that are greater than x (via the remove method). Line 4 also records if a value was actually removed and this is returned at line 6 again to let us know if constraint propagation may be required. If the domain is emptied an exception is thrown (line 5) and safely caught elsewhere.

A variable is considered to be instantiated when its domain has been reduced to a singleton. Therefore to instantiate a variable to a value x we remove from the domain of that variable all values but x. This is done via the method removeAllBut (lines 1 to 14 of Listing 4). We check to see if x really is in the domain (line 2) and if that value is not in the domain then the domain will become empty and an exception is

```
1   boolean removeAllBut(int x){
2         if (x < lwb || x > upb || !contains(x))
3             throw new CPException("The value "+ x +" is not in the domain.");
4         if (size() == 1) return false;
5         int v = x - lwb;
6         int w = value[0];
7         swap(value,0,location[v]);
8         location[w] = location[v];
9         location[v] = 0;
10        last.setValue(0);
11        min.setValue(x);
12        max.setValue(x);
13        return true;
14    }
```

Listing 4: Instantiating a variable.

thrown (line 2) and safely caught elsewhere. Otherwise we then swap the value x to the front of the set and make the last pointer point to location zero, thus removing all but the value x in $O(1)$ (lines 4 to 10). Finally, we have potentially a new minimum and maximum value and these are updated (lines 11 and 12) and at last we return true (a significant change has occurred).

```
1     int size(){return last.getValue() + 1;}
2
3     boolean isEmpty(){return size() == 0;}
4
5     int getNewMin(){
6         if (isEmpty()) throw new CPException("Domain is empty.");
7         for (int i=min.getValue()-lwb;i<n;i++)
8             if (location[i] <= last.getValue()) return i + lwb;
9         return Integer.MAX_VALUE;
10    }
11
12    int min(){return min.getValue();}
13
14    int getNewMax(){
15        if (isEmpty()) throw new CPException("Domain is empty.");
16        for (int i=max.getValue()-lwb;i>=0;i--)
17            if (location[i] <= last.getValue()) return i + lwb;
18        return Integer.MIN_VALUE;
19    }
20
21    int max(){return max.getValue();}
22
23    void reset(){
24        last.setValue(n-1);
25    }
26
27    int getLastValue(){
28        return value[last.getValue()] + lwb;
29    }
30 }
```

Listing 5: Utilities.

Listing 5 shows some other utility functions, such as finding the size (cardinality) of the set (line 1), delivering the smallest value in the domain (line 12), the largest value in the domain (line 21), reseting the domain (lines 23 to 25) and getting the last value in the unordered set (lines 27 to 29). Although not shown, the method removeBelow is a small edit distance from its symmetric partner removeAbove.

Reversible Variables

In the previous section we have seen that the pointer last is used as a boundary between values that have been removed from the domain and those that remain in the domain. Looking at Figure 1 if after our final removal (where we remove 0) we reset the last pointer to be equal to 3 we would return to the state immediately after the first removal, where 3 was removed from the domain. The domain would then be $[2, 4, 0, 1]$. That is, the values 1 and 0 are returned to the domain. However, the domain is reordered, but being a set this is acceptable. To allow the above magic we need last to be reversible, and we do this using reversible variables.

```
1  public class ReversibleInt extends ReversibleVar {
2
3      Problem pb;
4      int value;
5      Stack<Integer> whenChanged;
6      Stack<Integer> priorValue;
7
8
9      public ReversibleInt(Problem pb,int initialValue){
10         this.pb      = pb;
11         whenChanged = new Stack<Integer>();
12         priorValue  = new Stack<Integer>();
13         value       = initialValue;
14         whenChanged.push(pb.world);
15     }
16
17     int getValue(){return value;}
18
19     void setValue(int x){
20         if (value != x && whenChanged.peek() != pb.world){
21             pb.trail.peek().add(this);
22             whenChanged.push(pb.world);
23             priorValue.push(value);
24         }
25         value = x;
26     }
27
28     void restoreValue(){
29         whenChanged.pop();
30         value = priorValue.pop();
31     }
32 }
```

Listing 6: Reversible Integers

In weeSeepy® a ReversibleInt (Listing 6) works in conjunction with a Problem. A reversible variable has two stacks associated with it (lines 5 and 6) and has a current value, held in the integer value (declared in line 4, initialised in line 13, updated in line 25 and restored in line 30). We use the concept of a *world*, an idea proposed in the earilest versions of choco. A world might be thought of as a state, moving forwards in the search or backtracking. When we do something significant, such as selecting and instantiating a variable (followed by constraint propagation) we move into a new world. But the world is not an object but a collection of things that allow us to take a world view.

In the Problem, Listing 7, we have a trail (line 6) where that trail is a stack of lists of reversible variables. The world is just a simple integer. When we move into a new world, via a call to pushWorld (lines 13 to 16 in Listing 7) we create a new (empty) list of reversible variables (line 14) and push this list onto the trail and and then increment the world counter (line 15). When the value of a reversible variable is changed (lines 19 to 26 of Listing 6) we test if (a) it is indeed a new value that is being set and (b) that the last time this value changed was in a previous world. If these two tests (on line 20) are met we are in a new world and this reversible variable has changed value for the first time in that world. We then add this reversible variable onto the Problem's stack (line 21) and record in what world this

```
1     public Problem(String name){
2         this.name      = name;
3         variables      = new ArrayList<IntVar>();
4         constraints    = new ArrayList<Constraint>();
5         revisionStack = new Stack<Constraint>();
6         trail          = new Stack<ArrayList<ReversibleVar>>();
7         trace          = false;
8         world          = 0;
9         solutions      = 0;
10        firstProbe     = true;
11    }
12
13    void pushWorld(){
14        trail.push(new ArrayList<ReversibleVar>());
15        world++;
16    }
17
18    void popWorld(){
19        ArrayList<ReversibleVar> current = trail.pop();
20        for (ReversibleVar v : current) v.restoreValue();
21        world--;
22    }
```

Listing 7: The Problem and its trail of reversible variables

change took place (line 22) and store off the previous value of the reversible variable (line 23). We then unconditionally set the value of the reversible variable (line 25).

Therefore, in class Problem (Listing 7) a call to pushWorld (lines 13 to 16) makes space available to trail reversible variables. A call to popWorld (lines 18 to 22) then removes a list of reversible variables from the trailing stack (line 19), and calls this current, and then iterates over the reversible variables in current and restores their value (line 20). The world counter is then decremented (line 21).

Looking again at our reversible variables (Listing 6) the restoreValue method (lines 28 to 31) throws away the record of the world that the change occured in (line 29) and gets the earlier value of the variable (line 30).

```
1  public class TestDomain {
2
3      public static void main(String[] args){
4          Problem pb = new Problem("TestDomain");
5          Domain d = new Domain(pb,0,4);
6          System.out.println("new Domain(pb,0,4) "+ d);
7          d.remove(3);
8          System.out.println("d.remove(3)        "+ d);
9          d.remove(1);
10         System.out.println("d.remove(1)        "+ d);
11         d.remove(0);
12         System.out.println("d.remove(0)        "+ d);
13         pb.pushWorld();
14         System.out.println("pushWorld()        "+ d);
15         d.remove(2);
16         System.out.println("d.remove(2)        "+ d);
17         pb.popWorld();
18         System.out.println("popWorld()         "+ d);
19         d.removeAllBut(2);
20         System.out.println("d.removeAllBut(2)  "+ d);
21     }
22 }
```

Listing 8: A small example

We give a small example of reversible domains in Figure 2, the output of the program given in Listing 8. The domain of d is initially [0,1,2,3,4] (line 5). The value 3 is removed (domain is now [0,1,2,4]), then

```
new Domain(pb,0,4) [0,1,2,3,4]
d.remove(3)        [0,1,2,4]
d.remove(1)        [0,4,2]
d.remove(0)        [2,4]
pushWorld()        [2,4]
d.remove(2)        [4]
popWorld()         [4,2]
d.removeAllBut(2)  [2]
```

Figure 2: Yet another history of the domain {0..4}.

1 is removed (domain is now [0,4,2]) and then 0 is removed (domain is now [2,4]). We now enter a new world (line 13). We then remove the value 2 (line 15, domain is now [4]). In line 17 we move back to the previous world via the call to popWorld and domain is now [4,2]. Finally we remove all values but 2 (line 19) and domain is [2].

Outroduction

It is apparent that to describe any individual part of a constraint programming toolkit we must first explain all other parts of that toolkit. To explain domains we need to tell the reader about reversible variables and to tell the reader about reversible variables we must introduce the Problem and worlds. What we have explained so far is non-trivial. Unfortunately, things are not going to get easier because we must soon introduce constrained integer variables, constraints and constraint propagation, and these are all defined in terms of each other. But this must all be worthwhile, so we might take a glance at the goal and that is to perform a chronological search that performs constraint propagation. This is the algorithm that Daniel Sabin and Gene Freuder called MAC, for Maintaining Arc Consistency [5] and we present it in Listing 9. In subsequent notes we will describe constrained integer variables (used in lines 8, 9, 10 and 12 in Listing 9) and constrainst and how they behave (line 13 Listing 9).

```
1   public boolean bt() {
2       try {if (propagate()) bt0();}
3       catch (SolutionException e) {return true;}
4       return false;
5   }
6
7   private void bt0() {
8       IntVar v = selectVar();
9       if (v == null) throw new SolutionException();
10      for (int x : v){
11          pushWorld();
12          v.instantiate(x);
13          if (propagate()) bt0();
14          popWorld();
15      }
16  }
```

Listing 9: A taster ... chronological backtracking

Acknowledgements

To quote Oscar Wilde *"Imitation is the sincerest form of flattery that mediocrity can pay to greatness."* The good things reported here are lifted from miniCP [2] and choco [4]. The bad decisions I claim as my own.

References

[1] P. Briggs and L. Torczon. An efficient representation for sparse sets. *LOPLAS*, 2(1-4):59–69, 1993.

[2] Laurent Michel, Pierre Schaus, Pascal Van Hentenryck. MiniCP: A lightweight solver for constraint programming, 2018. Available from `https://minicp.bitbucket.io`.

[3] V. le Clément, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-sets for domain implementation. In *TRICS - Techniques foR Implementing Constraint programming Systems, CP 2013 Workshop.*, 2013.

[4] C. Prud'homme, J.-G. Fages, and X. Lorca. Choco documentation, 2017.

[5] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the Eleventh European Conference on Artificial Intelligence, Amsterdam, The Netherlands, August 8-12, 1994.*, pages 125–129, 1994.