Catholic University of Louvain

Project 2 : Propagation

LINGI2365 - Constraint Programming

Auteurs :
Vanwelde Romain (3143-10-00)
Crochelet Martin (2236-10-00)

Groupe 7

Superviseurs :
Pr. Yves Deville
François Aubry

7 mars 2014

Table des matières

1 Questions

1.1 Explain in detail the time and space complexities of the DC3 algorithm. What is the time complexity if all the constraints are domain consistent when algorithm DC3 is called ?

With the notations :

— $e = \#C$ where C is the set of constraints that set up the problem and e, the number of constraints.

— $d = \max_{1 \leq i \leq n}(\#D(x))$ where $D(x)$ is the domain of the variable x and thus d, the maximum size of the domains.

— r that is defined as the maximum arity of a constraint for the CSP.

And the supposition that each variable is involved in at least one constraint (and that the constraints might have an arity superior to two - otherwise we would reduce to AC3).

Analysing the algorithm shows us that DC3 keeps a queue that contains the set of constraints for whom the domain consistency is no longer guaranteed. In the worst case, we can easily imagine that every constraint of the problem will find it's place in that queue. Knowing that the number of constraints for a CSP is defined as e, we can deduce that the worst-case space complexity involved with the DC3 algorithm is $O(e)$ (Note that this is independent of for data-structures used by the propagate method)

Concerning the time complexity, we will split our analysis into two different parts : first, we will analyse the time-complexity of the standard CDC algorithm since the DC3 one in a basic instance of CDC. Lastly, we will tackle the analysis of the DC3-specific propagate method.

For a non binary CSP, we begin by observing that a constraint can be put at most $r \cdot d$ times in the queue (maximum arity of a constraint time the maximum size of a domain) and that at most, since we iterate over that queue to call propagate, the propagate method will be executed $e \cdot r \cdot d$ times (the number of constraints times the maximum number of times a constraint can be put into the queue).

Furthermore, we can remark that the DC3-specific propagate method iterates over each variables of a constraint and over the domain of that constraint $O(r \cdot d)$ This iteration executes a code verifies for each value if it is still consistent with the constraint and construct the delta set. This implies that the algorithm has to iterates over all other variables of the constraint to check if a possible assignment exists for the current choice of assignment : the execute runs in $O(r \cdot d^{r-1})$ and is executed at most r times. The time-complexity of the DC3-specific propagate method is thus $O(r^2 \cdot d^r)$.

We can then deduce that the overall complexity of DC3 is the product of the two complexities and equals $O(e \cdot r \cdot d \cdot r^2 \cdot d^r) = O(e \cdot r^3 \cdot d^{r+1})$

1.2 In the algorithm CDC (5.1 in the book), what would be the effect of moving the instruction of line 21 (Q -= c) after line 16 ? Would the algorithm still be correct ?

The algorithm would not be correct since it would never terminate. Indeed, moving the line just after line 16 would remove the constraint from the queue and then re add it into the queue since the enqueueCDC would contain the constraint itself. The algorithm would iterate forever over the constraints and never leave the loop.

1.3 Consider the constraint X - a = Y .

1.3.1 Show that it is domain consistent if and only if D(X) - a = D(Y).

This directly derives from the definition of domain consistent : suppose that a value that belongs to X generates a value $X' = X - a$ that does not belong to the domain of Y then the domain would not be consistent since the definition of domain consistent is that for all values that belong to the domain of the variable, there exists at least one value in the domains of the other variables of the constraint that satisfies the constraint.

1.3.2 What is the definition of bound consistency for this constraint ?

The bound consistency is a weaker form of domain consistency in the sense that it only considers the bounds of the domain. This means that we actually have two conditions : one on the minima of the variables and one on the maxima :

— on the minima : $\min_{x \in D(X)}(x) - a = \min_{y \in D(Y)}(y)$
— on the maxima : $\max_{x \in D(X)}(x) - a = \max_{y \in D(Y)}(y)$

1.3.3 Is this constraint automatically bound consistent if it is domain consistent ?

Indeed, if the constraint is domain consistent, this means that the bound are also consistent since they belong to the domain and that all the values in the domain satisfies the constraint. Bound consistency is a weaker form of domain consistency.

1.3.4 Is this constraint automatically domain consistent if it is bound consistent ?

No the implication if false in this sense. Indeed for a constraint such as $x^2 = y$, and the domains $[0, 1, 2, 3, 4, 5, 6, 7, 8]$ for both variables, bound consistency would return the domains : $\{x := [0, 1, 2], y := [0, 1, 2, 3, 4]\}$ while domain consistency would give the domains : $\{x := [0, 1, 2], y := [0, 1, 4]\}$.

2 Problems

2.1 AC3 propagator

2.1.1 What is the definition of domain consistency for this constraint

Intuitively, the domain consistency is the set of values of the domain that have a support (meaning a value that satisfies the constraint) in the domain of the other variables of the constraint. In this case, the domain consistency and bound consistency are equivalent under the assumption that the domain for X and Y are ordonable values. Indeed, since the constraint is linear, the condition on the bounds of the domains is sufficient to provoke a domain consistency. We have then the conditions :

— $\min_{x \in D(X)}(x) \geq \min_{y \in D(Y)}(y) + a$

— $\max_{x \in D(X)}(x) \geq \max_{y \in D(Y)}(y) + a$

Note that the $\geq$ plays an important role as well : in the previous case of constraint (same constraint but with equality), the domain consistency was different from the bound consistency. Indeed, the equality forces the values in the domain of Y to follow a linear function of the values of X while in the case of the $\geq$, the values of Y "just" have to belong to a subset of the plane that is defined by the minima and maxima of the domain of X.

2.1.2 Are domain consistency and bound consistency equivalent for this constraint ? Prove why they are, or why they are not.

See previous question.

2.1.3 What do the lines _x.addMax(this) and _y.addMin(this) in post do ?

Since we are using bound consistency (equivalent to domain consistency in this case), we can implement the CDC algorithm by using two different queues of propagation : one for the maxima and one for the minima). This is a simple optimization that allows to reconsider only the constraints that have been "touched" by the modification of the maxima *or* minima of a domain. The two lines in questions defines those queues and initialises them by putting all the constraints of the CSP into the queues. This corresponds to the initCDC method of the algorithm 5.1 of the book.

2.1.4 Explain why the propagate method is correct.

This implementation begins by updating the min and max of the domains of the variables and, if one of the domain boils down to zero, the propagation returns a failure. Otherwise, the propagation will test if the current assignment satisfies the constraint and, if it's the case, it returns a success. In the last case, the implementation will return a suspend meaning that the current assignment does not satisfies the constraint but that there are still values in the domain that have to be tried.

Furthermore, the propagate method has important side effects : it reduces the domain of the variables in order to accelerate the search by removing bounds of the domains. The specifications demand the new domain to be a subset of the previous one and that it still contains all the possible solutions for the CSP. This is easy to see considering the fact that the propagate method only removes values (via update$Max|Min$) that are proved to not satisfy the constraint.

2.2 The channeling constraint

2.2.1 Give the definition of domain consistency for this constraint

The definition of domain consistency for this constraint is the following one :

$$\forall x \; \epsilon X, \exists y \; \epsilon Y \; \bullet \; \text{x[i]} = (\text{y==i}) \;\&\; \forall y \; \epsilon Y, \exists x \; \epsilon X \; \bullet \; \text{x[i]} = (\text{y==i})$$

2.2.2 Implement an AC5 propagator achieving domain consistency for this constraint.

We had to implement method `valRemove` and `valBind`.
A important fact to notice for the next of the exercise is that the array can only contain one true value since the Y variable is an integer.

For the valRemove method, we first determine if the value removed is from Y domain or from one of the X_i domains.
If the value has been removed from the Y domain, we can directly determine that $\_X[value] == 0$. (Thanks to the constraint).
If we remove the value from one of the X_i domains, we can deduce more informations depending on the value. **If value equals 0**, we know that the only remaining value to that domain is 1. Then, we can thus deduce the only remaining value we can attribute to the variable Y which is the index of the X_i variable binded. We can furthermore bind all others X_i varaibles to 0. **If value equals 1**, we know that the only remaining value to that domain is 0, and we can remove the index of the X_i variable binded from the Y domain.

```
Outcome<CP> valRemove(var<CP>{int} z, int val) {
    if (z.getId() == _Y.getId()) {
        if (_X[val].bindValue(0) == Failure) return Failure;
    } else {
        // Since the value removed is zero, the associated var is directly
            binded to 1
        // All the others of the boolean table have to be at zero
        if (val == 0){
            forall (u in _X.getRange()){
                if (_X[u].getId() == z.getId()){
                     if (_Y.bindValue(u) == Failure) return Failure;
                     if (_X[u].bindValue(1) == Failure) return Failure;
                } else {
                     if (_X[u].bindValue(0) == Failure) return Failure;
                }
            }
        // If the value removed is one, we bind zero to the associated var
        } else if (val == 1) {
            forall (u in _X.getRange()){
                if (_X[u].getId() == z.getId()){
                     if (_X[u].bindValue(0) == Failure) return Failure;
                     if (_Y.removeValue(u) == Failure) return Failure;
                }
            }
        }
    }
    return Suspend;
}
```

ValRemove_Channeling.co

The valRemove method do similar things.
First, if the binded variable is Y, we can deduce the entire corresponding array.
If the binded varaible is on of the X_i variables, we can determine some informations depending of the value binded.
If the value is 1, we can determine that all the others X_i variables equals 0 and that the Y variable equals the index of the binded X_i variable.
If the value is 0, we can just remove the corresponding index of the domain of Y.

```
   Outcome<CP> valBind(var<CP>{int} z, int val) {
      // Binded var is Y
      if (z.getId() == _Y.getId()) {
         forall (u in _X.getRange()){
            // we can bind all the others vars of X
            if (_X[u].bindValue(u == val) == Failure) return Failure;
         }
      } else { // Binded var is one of the X vars
         forall (u in _X.getRange()){
            if (_X[u].getId() == z.getId()){
               // If val==0, we remove the index of Dom(Y)
               if (val == 0)
                  if (_Y.removeValue(u)== Failure) return Failure;
               // If val==1, we bind variable Y
               if (val == 1)
                  if (_Y.bindValue(u)== Failure) return Failure;

            } else if (_X[u].getId() != z.getId()){
                  // If val==1, we bind all other var of X to 0
                 if (val == 1)
                  if (_X[u].bindValue(0)== Failure) return Failure;
            }
         }
      }
      return Suspend;
   }
```

ValBind_Channeling.co

2.3 AC2001 propagator

2.3.1 Generic constraint

We first declare the two constraints variables and the two support structures.

```
abstract class AC2001Constraint extends UserConstraint<CP> {

   // constraint variables
   var<CP>{int} _x;
   var<CP>{int} _y;
   // support structures
   trail{int}[] _firstYSupport;
   trail{int}[] _firstXSupport;

   AC2001Constraint(var<CP>{int} x, var<CP>{int} Y) : UserConstraint<CP>()
      {
        ...
    }

...

}
```

AC2001Constraint_variables.co

In the constructor, we assign constraints variables, and we instantiate the support structure.

```
    AC2001Constraint(var<CP>{int} x, var<CP>{int} Y) : UserConstraint<CP>()
        {
        // assign constraint variables
        _x = x;
        _y = Y;

        // create support structures
        _firstYSupport = new trail {int}[_x.getMin().._x.getMax()](_x.
            getSolver());
        _firstXSupport = new trail {int}[_y.getMin().._y.getMax()](_y.
            getSolver());
    }
```

AC2001Constraint_constructor.co

Then, in the post method, we initialize support structures, we do the initial propagation, and we suscribe to some events.

```
    // the postAC2001 method
    Outcome<CP> post(Consistency<CP> cl) {
        // initialize support structures
        forall (u in _x.getMin().._x.getMax() : _x.memberOf(u))
            _firstYSupport[u] := _y.getMin();
        forall (v in _y.getMin().._y.getMax() : _y.memberOf(v))
            _firstYSupport[v] := _x.getMin();

         // do initial propagation
        Outcome<CP> resPropag = propagate();

         // subscribe your constraint to some events
        _x.addDomain(this);
        _y.addDomain(this);
        return resPropag;
    }
```

AC2001Constraint_post.co

The propagate method is the most important one. It propagates for X and Y. We iterate on all variables, and beginning at the firstSupportValue, we try to find the new one (which could remain the same). Once there are no more firstSupportValue, we remove the value of the corresponding domain.

```
    // the propagateAC2001 method
    Outcome<CP> propagate() {
         //propagate_varAC2001 for X
```

```
      forall (u in _x.getMin().._x.getMax() : _x.memberOf(u)){
         bool support = false;
         forall (v in _firstYSupport[u].._y.getMax())
            if (check(u,v)){
               support = true;
               _firstYSupport[u] := v;
               break;
            }
         if (!support && _x.removeValue(u) == Failure)
            return Failure;
      }

      //propagate_varAC2001 for Y
      forall (u in _y.getMin().._y.getMax() : _y.memberOf(u)){
         bool support = false;
         forall (v in _firstXSupport[u].._x.getMax())
            if (check(u,v)){
               support = true;
               _firstXSupport[u] := v;
               break;
            }
         if (!support && _y.removeValue(u) == Failure)
            return Failure;
      }
      return Suspend;
   }
```

AC2001Constraint_propagate.co

2.3.2 Specific constraints

Here is the method check for the first three constraints.

```
// You've got three constraints to implement here below


//doubleModulo Constraint : x mod k = y mod k (k is a constant)
class AC2001DoubleModulo extends AC2001Constraint{
     int _k;
     AC2001DoubleModulo(var<CP>{int} x, var<CP>{int} y,int k) :
        AC2001Constraint(x,y){
        _k = k;
     }
     boolean check(int a, int b) {return ((a%_k)==(b%_k)) ;}
}

//sum cstr //  x+y= k  (k is a constant)
class AC2001Sum extends AC2001Constraint{
     int _k;

     AC2001Sum(var<CP>{int} x,var<CP>{int} y,int k) : AC2001Constraint(x,y){
        _k=k;
     }
     boolean check(int a, int b) {return ((a+b) == _k); }
}
```

```

//distance positive |x-y| = k
class AC2001Dist extends AC2001Constraint {
  int _k;
  AC2001Dist(var<CP>{int} x, var<CP>{int} y, int k) : AC2001Constraint(x,y)
      {
     _k = k;
  }
  boolean check(int a, int b) { return (abs(a-b)==_k); }
}
```

AllConstraints.co

2.4 The AllDifferent constraint

2.4.1 Define forward-checking consistency for the decomposition of AllDiff in binary constraints.

A constraint c(x_m,x_n) is FC consistent wrt D(X) iff :
— if $\forall x_i \in \{x_m, x_n\} : D(x_i) = \{a_i\}$, then c($a_m$,$a_n$)
— if $\exists y \in \{x_m, x_n\}\ \forall x \in \{x_m, x_n\} \backslash \{y\}$: #D(x) = 1, then c is domain consistent wrt D(X)

Here, AllDiff is FC consistent if

$$\forall i \in \{1, k\}, \forall j \in \{1, k\}, i! = j \bullet \text{c}(x_m,x_n) \text{ is FC consistent.}$$

2.4.2 Implement a propagator ensuring a consistency for AllDiff equivalent to forward- checking consistency for the decomposition of AllDiff (. . .)

In the post method, we iterate on all the variables. Those who are not bind are subscribed to bind event.

```
   Outcome<CP> post(Consistency<CP> cl) {
      forall(i in 1.._X.getSize()) {
         // if _X[i] is not binded
         if (!_X[i].bound()){
            // We suscribe it to the bind event
            _X[i].addAC5Bind(this);
         }
      }
      return Suspend;
   }
```

AllDiffFC_post.co

In the valBind method, we remove from all variable's domain different from the one which have been binded, the value that have been binded.

```
   Outcome<CP> valBind(var<CP>{int} z, int val) {
      forall(i in 1.._X.getSize()) {
         if (z.getId() != _X[i].getId())
         {
            // we remove the value from the others domains
            if(_X[i].removeValue(val) == Failure)
               return Failure;
         }

      }
      return Suspend;
   }
```

AllDiffFC_valBind.co

2.4.3 Modify the Nqueens.co file provided into file NqueensFC.co such that it uses your implementation of alldifferent.

The only lines which have been modified are the lines who post the constraints. The constructor used creates a "fake" variable known as a view.

```
solve<m> {

  m.post(AllDiffFC(all(i in S) new var<CP>{int}(m,q[i], i)),onDomains);
  m.post(AllDiffFC(all(i in S) new var<CP>{int}(m,q[i], - i)),onDomains);
  m.post(AllDiffFC(q),onDomains);

} using {
   label(q);
   c := c + 1;
}
```

NQueensFC.co

2.4.4 Test both yours (NqueensFC.co) and Comet's (Nqueens.co) implementation of AllDifferent on the NQueens problem. Compare the number of failures and choices between the two implementations and for each n = 12 . . . 32 in a table.

n	12	13	14	15	16	17	18	19	20	21	22	23	24
NQueens	20	11	117	79	597	327	1.936	71	7.341	160	57.834	339	8.203
NQueensFC	34	32	208	143	1.123	752	4.508	381	22.220	943	162.268	2.510	38.902

n	25	26	27	28	29	30	31	32
NQueens	455	5.138	4.649	36.342	13.672	758.650	112.867	826.632
NQueensFC	4.507	34.458	39.943	252.503	126.158	4.574.951	1.087.399	7.040.698

Figure 1 – Choices

n	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
NQueens	39	13	270	176	1.358	714	4.540	172	17.921	399	135.923	847	19.603	1.150	12.032	12.211	89.509	35.369	1.837.227	288.305	2.057.346
NQueensFC	64	16	423	290	2.305	1.561	9.557	742	48.059	1.959	343.151	5.193	81.171	9.149	69.903	84.175	523.005	266.235	9.374.708	2.250.869	14.336.324

Figure 2 – Failures

2.4.5 What did you find in your comparative study ? Explain the results !

First, we observe that our implementation of allDifferent has each time a number of choices bigger than in the comet's implementation. Since this number is bigger, we will have to try more values before finding a good one. Thus, we will have more failures.