Consistency Techniques

[node consistency] [arc consistency] [path consistency]

Consistency techniques were first introduced for improving the efficiency of picture recognition programs, by researchers in artificial intelligence [Waltz]. Picture recognition involves labelling all the lines in a picture in a consistent way. The number of possible combinations can be huge, while only very few are consistent. Consistency techniques effectively rule out many inconsistent labellings at a very early stage, and thus cut short the search for consistent labellings. These techniques have since proved to be effective on a wide variety of hard search problems.

Notice that consistency techniques are deterministic, as opposed to the search which is non-deterministic. Thus the deterministic computation is performed as soon as possible and non-deterministic computation during search is used only when there is no more propagation to done. Nevertheless, the consistency techniques are rarely used alone to solve constraint satisfaction problem completely (but they could).

In binary CSPs, various consistency techniques for constraint graphs were introduced to prune the search space. The consistency-enforcing algorithm makes any partial solution of a small subnetwork extensible to some surrounding network. Thus, the potential inconsistency is detected as soon as possible.

Node Consistency

The simplest consistency technique is refered to as node consistency and we mentioned it in the section on binarization of constraints. The node representing a variable V in constraint graph is **node consistent** if for every value x in the current domain of V, each unary constraint on V is satisfied.

If the domain D of a variable V containts a value "a" that does not satisfy the unary constraint on V, then the instantiation of V to "a" will always result in immediate failure. Thus, the node inconsistency can be eliminated by simply removing those values from the domain D of each variable V that do not satisfy unary constraint on V.

Algorithm NC

```
procedure NC
  for each V in nodes(G)
    for each X in the domain D of V
      if any unary constraint on V is inconsistent with X
      then
        delete X from D;
      endif
    endfor
  endfor
end NC
```

Arc Consistency

If the constraint graph is node consistent then unary constraints can be removed because they all are satisfied. As we are working with the binary CSP, there remains to ensure consistency of binary constraints. In the constraint graph, binary constraint corresponds to arc, therefore this type of consistency is called arc consistency.

Arc (V_i,V_j) is **arc consistent** if for every value x the current domain of V_i there is some value y in the domain of V_j such that V_i=x and V_j=y is permitted by the binary constraint between V_i and V_j. Note, that the concept of arc-consistency is directional, i.e., if an arc (V_i,V_j) is consistent, than it does not automatically mean that (V_j,V_i) is also consistent.

Clearly, an arc (V_i,V_j) can be made consistent by simply deleting those values from the domain of V_i for which there does not exist corresponding value in the domain of D_j such that the binary constraint between V_i and V_j is satisfied (note, that deleting of such values does not eliminate any solution of the original CSP). The following algorithm does precisely that.

Algorithm REVISE

```
procedure REVISE(Vi,Vj)
  DELETE <- false;
  for each X in Di do
    if there is no such Y in Dj such that (X,Y) is consistent,
    then
       delete X from Di;
       DELETE <- true;
    endif;
  endfor;
  return DELETE;
end REVISE
```

To make every arc of the constraint graph consistent, it is not sufficient to execute REVISE for each arc just once. Once REVISE reduces the domain of some variable V_i, then each previously revised arc (V_j,V_i) has to be revised again, because some of the members of the domain of V_j may no longer be compatible with any remaining members of the revised domain of V_i. The following algorithm, known as **AC-1**, does precisely that.

Algorithm AC-1

```
procedure AC-1
  Q <- {(Vi,Vj) in arcs(G),i#j};
  repeat
    CHANGE <- false;
    for each (Vi,Vj) in Q do
      CHANGE <- REVISE(Vi,Vj) or CHANGE;
    endfor
  until not(CHANGE)
end AC-1
```

This algorithm is not very efficient because the succesfull revision of even one arc in some iteration forces all the arcs to be revised again in the next iteration, even though only a small number of them are really affected by this revision. Visibly, the only arcs affected by the reduction of the domain of V_k are the arcs (V_i,V_k). Also, if we revise the arc (V_k,V_m) and the domain of V_k is reduced, it is not necessary to re-revise the arc (V_m,V_k) because non of the elements deleted from the domain of V_k provided support for any value in the current domain of V_m. The following variation of arc consistency algorithm, called AC-3, removes this drawback of AC-1 and performs re-revision only for those arcs that are possibly affected by a previous revision.

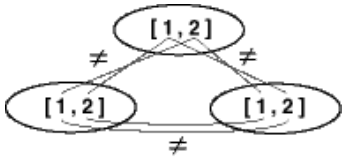
Algorithm AC-3

```
procedure AC-3
  Q <- {(Vi,Vj) in arcs(G),i#j};
  while not Q empty
    select and delete any arc (Vk,Vm) from Q;
```

```
    if REVISE(Vk,Vm) then
      Q <- Q union {(Vi,Vk) such that (Vi,Vk) in arcs(G),i#k,i#m}
    endif
  endwhile
end AC-3
```

When the algorithm AC-3 revises the edge for the second time it re-tests many pairs of values which are already known (from the previous iteration) to be consistent or inconsistent respectively and which are not affected by the reduction of the domain. As this is a source of potential inefficiency, the algorithm **AC-4** was introduced to refine handling of edges (constraints). The algorithm works with indiviual pairs of values as the following example shows.

Example:



First, the algorithm AC-4 initializes its internal structures which are used to remember pairs of consistent (inconsistent) values of incidental variables (nodes) - structure $S_{i,a}$. This initialization also counts "supporting" values from the domain of incindental variable - structure $counter_{(i,j),a}$ - and it removes those values which have no support. Once the value is removed from the domain, the algorithm adds the pair <Variable,Value> to the list Q for re-revision of affected values of corresponding variables.

Algorithm INITIALIZE

```
procedure INITIALIZE
  Q <- {};
  S <- {};   % initialize each element of structure S
  for each (Vi,Vj) in arcs(G) do   % (Vi,Vj) and (Vj,Vi) are same elements
    for each a in Di do
      total <- 0;
      for each b in Dj do
        if (a,b) is consistent according to the constraint (Vi,Vj) then
          total <- total+1;
          Sj,b <- Sj,b union {<i,a>};
        endif
      endfor;
      counter[(i,j),a] <- total;
      if counter[(i,j),a]=0 then
        delete a from Di;
        Q <- Q union {<i,a>};
      endif;
    endfor;
  endfor;
  return Q;
end INITIALIZE
```

After the initialization, the algorithm AC-4 performs re-revision only for those pairs of values of incindental variables that are affected by a previous revision.

Algorithm AC-4

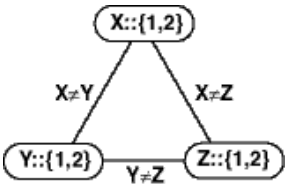
```
procedure AC-4
  Q <- INITIALIZE;
  while not Q empty
    select and delete any pair <j,b> from Q;
```

```
        for each <i,a> from Sj,b do
          counter[(i,j),a] <- counter[(i,j),a] - 1;
          if counter[(i,j),a]=0 & a is still in Di then
            delete a from Di;
            Q <- Q union {<i,a>};
          endif
        endfor
      endwhile
    end AC-4
```

Both algorithms, AC-3 and AC-4, belong to the most widely used algorithms for maintaining arc consistency. It should be also noted that there exist other algorithms AC-5, AC-6, AC-7 etc. but their are not used as frequently as AC-3 or AC-4.

Maintaining arc consistency removes many inconsistencies from the constraint graph but is any (complete) instantiation of variables from current (reduced) domains a solution to the CSP? If the domain size of each variable becomes one, then the CSP has exactly one solution which is obtained by assigning to each variable the only possible value in its domain. Otherwise, the answer is no in general. The following example shows such a case where the constraint graph is arc consistent, domains are not empty but there is still no solution satisfying all constraints.

Example:



This constraint graph is arc consistent but there is no solution that satisfies all the constraints.

K-consistency (Path Consistency)

Given that arc consistency is not enough to eliminate the need for backtracking, is there another stronger degree of consistency that may eliminate the need for search? The above example shows that if one extends the consistency test to two or more arcs, more inconsistent values can be removed.

A graph is **K-consistent** if the following is true: Choose values of any K-1 variables that satisfy all the constraints among these variables and choose any Kth variable. Then there exists a value for this Kth variable that satisfies all the constraints among these K variables. A graph is **strongly K-consistent** if it is J-consistsent for all J<=K.

Node consistency discussed earlier is equivalent to strong 1-consistency and arc-consistency is equivalent to strong 2-consistency (arc-consistency is usually assumed to include node-consistency as well). Algorithms exist for making a constraint graph strongly K-consistent for K>2 but in practice they are rarely used because of efficiency issues. The exception is the algorithm for making a constraint graph strongly 3-consistent that is usually refered as **path consistency**. Nevertheless, even this algorithm is too hungry and a weak form of path consistency was introduced.

A node representing variable V_i is **restricted path consistent** if it is arc-consistent, i.e., all arcs from this node are arc-consistent, and the following is true: For every value a in the domain D_i of the variable V_i that has *just one supporting value* b from the domain of incindental variable V_j there exists a value c in the domain of other incindental variable V_k such that (a,c) is permitted by the binary constraint between V_i and V_k, and (c,b) is permitted by the binary constraint between V_k and V_j.

The algorithm for making graph restricted path consistent can be naturally based on AC-4 algorithm that counts the number of supporting values. Although this algorithm removes more inconsistent values than any arc-consistency algorithm it does not eliminate the need for seach in general. Clearly, if a constraint graph containing n nodes is strongly n-consistent, then a solution to the CSP can be found without any search. But the worst-case complexity of

the algorithm for obtaining n-consistency in a n-node constraint graph is also exponencial. If the graph is (strongly) K-consistent for K<n, then in general, backtracking cannot be avoided, i.e., there still exist inconsistent values.

[node consistency] [arc consistency] [path consistency]

---

based on Vipin Kumar: Algorithms for Constraint Satisfaction Problems: A Survey, AI Magazine 13(1):32-44,1992

Further reading:

Consistency in networks of relations [AC1-3]
A.K. Mackworth, in Artificial Intelligence 8, pages 99-118, 1977.

The complexity of some polynomial network consistency algorithms for constraint satisfaction problems [AC1-3]
A.K. Mackworth and E.C. Freuder, in Artificial Intelligence 25, pages 65-74, 1985.

Arc and path consistency revised [AC4]
R. Mohr and T.C. Henderson, in Artificial Intelligence 28, pages 225-233, 1986.

Arc consistency for factorable relations [AC5]
M. Perlin, in Artificial Intelligence 53, pages 329-342, 1992.

A generic arc-consistency algorithm and its specializations [AC5]
P. Van Hentenryck, Y. Deville, and C.-M. Teng, in Artificial Intelligence 57, pages 291-321, 1992.

Arc-consistency and arc-consistency again [AC6]
C. Bessiere, in Artificial Intelligence 65, pages 179-190, 1994.

Using constraint metaknowledge to reduce arc consistency computation [AC7]
C. Bessiere, E.C. Freuder, and J.-R. Régin, in Artificial Intelligence 107, pages 125-148, 1999.