CONSTRAINT PROPAGATION AND BACKTRACKING-BASED SEARCH

A brief introduction to mainstream techniques of constraint satisfaction

ROMAN BARTÁK

Charles University, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic

e-mail: `roman.bartak@mff.cuni.cz`

„Were you to ask me which programming paradigm is likely to gain most in commercial significance over the next 5 years I'd have to pick Constraint Logic Programming (CLP), even though it's perhaps currently one of the least known and understood."

Dick Pountain, BYTE, February 1995

Introduction

What is a constraint programming, what are its origins and why is it useful?

Constraint programming is an emergent software technology for declarative description and effective solving of combinatorial optimization problems in areas like planning and scheduling. It represents the most exciting developments in programming languages of the last decade and, not surprisingly, it has recently been identified by the ACM (Association for Computing Machinery) as one of the strategic directions in computer research. Not only it is based on a strong theoretical foundation but it is attracting widespread commercial interest as well, in particular, in areas of modelling heterogeneous optimisation and satisfaction problems.

What is a constraint?

A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. A constraint thus restricts the possible values that variables can take; it represents some partial information about the variables of interest. For instance, "the circle is inside the square" relates two objects without precisely specifying their positions, i.e., their co-ordinates. Now, one may move the square or the circle and he or she is still able to maintain the relation between these two objects. Also, one may want to add anther object, say triangle, and to introduce another constraint, say "square is to the left of the triangle". From the user (human) point of view, everything remains absolutely transparent.

Constraints arise naturally in most areas of human endeavour. The three angles of a triangle sum to 180 degrees, the sum of the currents floating into a node must equal zero, the position of the scroller in the window scrollbar must reflect the visible part of the underlying document. These are some examples of constraints which appear in the real world. Constraints can also be heterogeneous and so they can bind unknowns from different domains, for example the length (number) with the word (string). Thus, constraints are a natural medium for people to express problems in many fields.

We all use constraints to guide reasoning as a key part of everyday common sense. "I can be there from five to six o'clock", this is a typical constraint we use to plan our time. Naturally, we do not solve one constraint only but a collection of constraints that are rarely independent. This complicates the problem a bit, so, usually, we have to give and take.

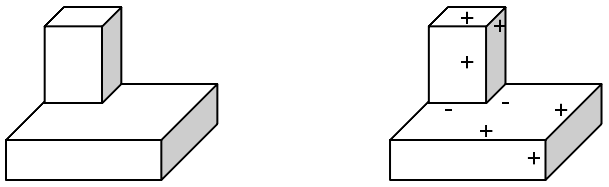
Constraints naturally enjoy several interesting properties:

- constraints may specify *partial* information, i.e., the constraint need not uniquely specify the values of its variables, (constraint X>2 does not specify the exact value of variable X, so X can be equal to 3, 4, 5 etc.)
- constraints are *heterogeneous*, i.e., they can specify the relation between variables with different domains (for example X = length(Y))
- constraints are *non-directional*, typically a constraint on (say) two variables X, Y can be used to infer a constraint on X given a constraint on Y and vice versa, (X=Y+2 can be used to compute the variable X using X:=Y+2 as well as the variable Y using Y:=X-2)
- constraints are *declarative*, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship,
- constraints are *additive*, i.e., the order of imposition of constraints does not matter, all that matters at the end is that the conjunction of constraints is in effect,
- constraints are *rarely independent*, typically constraints in the constraint store share variables.

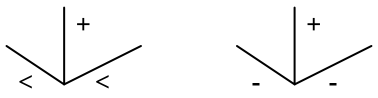
A bit of history ...

Constraints have recently emerged as a research area that combines researchers from a number of fields, including Artificial Intelligence, Programming Languages, Symbolic Computing and Computational Logic. Constraint networks and constraint satisfaction problems have been studied in Artificial Intelligence starting from the seventies (Montanary, 1974), (Waltz, 1975). Systematic use of constraints in programming has started in the eighties (Gallaire, 1985), (Jaffar, Lassez, 1987).

The constraint satisfaction origins from Artificial Intelligence where the problems like scene labelling was studied (Waltz, 1975). The scene labelling problem is probably the first constraint satisfaction problem that was formalised. The goal is to recognise the objects in the scene by interpreting lines in the drawings. First, the lines or edges are labelled, i.e., they are categorised into few types, namely convex (+), concave (-) and occluding edges (<). In some advanced systems, the shadow border is recognised as well.



There are a lot ways how to label the scene (exactly 3^n, where n is a number of edges) but only few of them has any 3D meaning. The idea how to solve this combinatorial problem is to find legal labels for junctions satisfying the constraint that the edge has the same label at both ends. This reduces the problem a lot because there are only a very limited number of legal labels for junctions.



... and some applications.

Constraint programming has been successfully applied in numerous domains. Recent applications include computer graphics (to express geometric coherence in the case of scene analysis), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), operations research problems (like optimisation problems), molecular biology (DNA sequencing), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts), etc.

Current research in this area deals with various foundational issues, with implementation aspects, and with new applications of constraint programming.

What does the constraint programming deal with?

Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (conditions, properties, requirements) which must be satisfied by the solution.

Work in this area can be tracked back to research in Artificial Intelligence (Montanary, 1974), (Waltz, 1975) and Computer Graphics (Sutherland, 1963), (Borning, 1981) in the sixties and seventies. Only in the last two decades, however, there has emerged a growing realisation that these ideas provide the basic for a powerful approach to programming (Gallaire, 1985), (Jaffar, Lassez, 1987), modelling, and problem solving and that different efforts to exploit these ideas can be unified under a common conceptual and practical framework, *constraint programming*.

Currently there can be seen two branches of Constraint Programming research which arise from distinct bases and, thus, use different approaches to solve constraints. Constraint Programming roofs both of them.

- **Constraint Satisfaction**

Constraint Satisfaction arose from the research in Artificial Intelligence (combinatorial problems, search) and Computer Graphics (SKETCHPAD system by Sutherland, expressing geometric coherence in the case of scene analysis). The Constraint Satisfaction Problem (CSP) is defined by:

- a finite set of variables,
- a function which maps every variable to a finite domain,
- a finite set of constraints.

Each constraint restricts the combination of values that a set of variables may take simultaneously. A solution of a CSP is an assignment to each variable a value from its domain satisfying all the constraints. The task is to find one solution or all solutions.

Thus, the CSP is a combinatorial problem which can be solved by search. There exists a trivial algorithm that solves such problems or finds that there is no solution. This algorithm generates all possible combinations of values and, then, it tests whether the given combination of values satisfies all constraints or not (consequently, this algorithm is called *generate and test*). Clearly, this algorithm takes a long time to run so the research in the area of constraint satisfaction concentrate on finding algorithms which solve the problem more efficiently, at least for a given subset of problems.

- **Constraint Solving**

Constraint Solving differs from Constraint Satisfaction by using variables with infinite domains like real numbers. Also, the individual constraints are more complicated, e.g., non-linear equalities. Consequently, the constraint solving algorithms uses the algebraic and numeric methods instead of combinations and search. However, there exists an approach which discretizes the infinite domain into finite number of components and, then, applies the techniques of constraint satisfaction. This tutorial does not cover constraint solving techniques.

Give me some examples

There are a lot of toy problems that can be solved using constraint programming naturally. Among them, the graph (map) colouring, N-queens, and crypto-arithmetic have a privilege position.

N-Queens

The N-queens problem is a well know puzzle among computer scientists. Given any integer N, the problem is to place N queens on squares in an N*N chessboard satisfying the constraint that no two queens threaten each other (a queen threatens any other queens on the same row, column and diagonal).

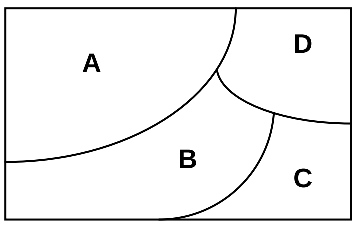
A typical way how to model this problem is to assume that each queen is in different column and to assign a variable R_i (with the domain 1...N) to the queen in the i-th column indicating the position of the queen in the row. Now, its is easy to express "no-threatening" constraints between each couple R_i and R_j of queens:

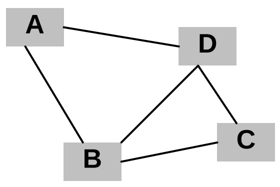
$$i \neq j \Rightarrow (R_i \neq R_j \ \& \ |i-j| \neq |R_i - R_i|)$$

Graph (map) colouring

Another problem which is often used to demonstrate potential of constraint programming and to explain concepts and algorithms for the CSP is the colouring problem. Given a graph (a map) and a number of colours, the problem is to assign colours to those nodes (areas in the map) satisfying the constraint that no adjacent nodes (areas) have the same colour assigned to them.

This problem is modelled naturally by labelling each node of the graph with a variable (with the domain corresponding to the set of colours) and introducing the non-equality constraint between each two variables labelling adjacent nodes.





Crypto-arithmetic

Last but not least example of using constraint techniques is a crypto-arithmetic problem. In fact, it is a group of similar problems. Given a mathematical expression where letters are used instead of numbers, the problem is to assign digits to those letters satisfying the constraint that different letters should have different digits assigned and the mathematical formulae holds. Here is a typical example of the crypto-arithmetic problem:

SEND + MORE = MONEY

The problem can be modelled by identifying each letter with a variable (with domain 0...9), by direct rewriting the formulae to an equivalent arithmetic constraint:

```
                 1000*S + 100*E + 10*N + D
+                1000*M + 100*O + 10*R + E
=  10000*M +  1000*O + 100*N + 10*E + Y
```

and by adding auxiliary constraints $S \neq 0$ and $M \neq 0$.

There are also other toy problems that can be solved using constraint programming techniques like Zebra (five houses puzzle), a crossword puzzle, or mastermind.

What about practical applications?

Of course, the constraint programming is not popular because of solving toy problems but because of its potential to model and solve real-life problems naturally and efficiently. Constraint programming can also serve as a roof for combination of different approaches, like integer programming and operation research.

The number of companies exploiting constraint technology increases each year. Here is a list of some of the well-known companies among them:

> British Airways, SAS, Swissair, French railway authority SNCF, Hong Kong International Terminals, Michelin, Dassault, Ericsson etc.

Also, there are a lot of companies providing solutions based on constraints like PeopleSoft, i2 Technologies, InSol, Vine Solutions or companies providing constraint-based tools like ILOG, IF Computer, Cosytec, SICS, or PrologIA.

The constraint programming techniques can lend a hand to many real-life problems. Among other things:

- time-tabling
  - workforce management
  - course scheduling
  - stuff scheduling
    - nurse scheduling
    - crew rostering problem (Italian Railway Company)
- planning and scheduling
  - transport planning
  - on-demand manufacturing
    - car sequencing
- resource allocation
  - forest treatment scheduling
  - well activity scheduling (Saga Petroleum a.s.)
  - airport counter allocation (Cathay Pacific Airways Ltd)
- analysis and synthesis of analogue and digital circuits
- option trading analysis
- cutting stock
- DNA sequencing
- chemical hypothetical reasoning
- warehouse location
- network configuration

Constraint Satisfaction

How to tackle constraint satisfaction problems?

Constraint Satisfaction Problems (CSPs) have been a subject of research in Artificial Intelligence for many years. The pioneering works on networks of constraints were motivated mainly by problems arising in the field of picture processing (Waltz, 1975), (Montanari,1974). AI research, concentrated on difficult combinatorial problems, is dated back to sixties and seventies and it has contributed to considerable progress in constraint-based reasoning. Many powerful algorithms were designed that became a basis of current constraint satisfaction algorithms.

Constraints, an ultimate anti NP-Hard weapon?

Most problems that the constraint programming concerns belong to the group that conventional programming techniques find hardest. Time needed to solve such problems using unconstrained search increases exponentially with the problem size.

Consider the simple problem of harbour which needs to schedule the loading and unloading of 10 ships using only 5 berths. You can solve this problem by trying all permutations of ships in berths, calculating the cost of each alternative and selecting the optimal schedule. This means exploring 5^{10} (about 10 million) alternatives in the worst case. Assuming that your computer can try an alternative every millisecond, then the whole problem is solved in around 3 hours. A decade later, the business has been good and the harbour has expanded to 10 berths and 20 ships. Now, finding the optimal schedule using the same method means trying 10^{20} alternatives, which will take more than 3000 million years on the same computer. Even a thousand times faster accelerator card does not help there. Fortunately, one does not need to explore all the alternatives. There are many criteria for choosing the berth for a particular ship, for example some berths are too small for some ships and it is not possible to load or unload two ships in the same berth at the same time etc. By embracing these constraints, the search space reduces dramatically and it makes the problem tractable. Also, in many real life problems the optimal solution is not necessarily required and a near to optimal solution is enough in many cases. For example, it is possible to break up the harbour into two parts, each with 5 berths and 10 ships, and solve this split problem in 6 hours using the above "brute force" method.

What is a CSP and its solution?

A *Constraint Satisfaction Problem* (CSP) consists of:

- a set of *variables* $X=\{x_1,...,x_n\}$,
- for each variable x_i, a finite set D_i of possible values (its *domain*),
- and a set of *constraints* restricting the values that the variables can simultaneously take.

Note that values need not be a set of consecutive integers (although often they are). They need not even be numeric.

A *solution to a CSP* is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied. We may want to find:

- just one solution, with no preference as to which one,
- all solutions,
- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables; in this case we speak about *Constraint Optimisation Problem* (COP).

Solutions to a CSP can be found by searching systematically through the possible assignments of values to variable. Search methods divide into two broad classes, those that traverse the space of partial solutions (or partial value assignments), and those that explore the space of complete value assignments (to all variables) stochastically.

The *reasons* for choosing to represent and solve a problem as a CSP rather than, say as a mathematical programming problem are twofold.

- First, the representation as a CSP is often much closer to the original problem: the variables of the CSP directly correspond to problem entities, and the constraints can be expressed without having to be translated into linear inequalities. This makes the formulation simpler, the solution easier to understand, and the choice of good heuristics to guide the solution strategy more straightforward.
- Second, although CSP algorithms are essentially very simple, they can sometimes find solution more quickly than if integer programming methods are used.

What is going on?

This tutorial is intended to give a basic grounding in constraint satisfaction problems and some of the algorithms used to solve them. In general, the tasks posed in the constraint satisfaction problem paradigm are computationally intractable (NP-hard).

Systematic search algorithm

A CSP can be solved using *generate-and-test paradigm* (GT) that systematically generates each possible value assignment and then it tests to see if it satisfies all the constraints. A more efficient method uses the *backtracking paradigm* (BT) that is the most common algorithm for performing systematic search. Backtracking incrementally attempts to extend a partial solution toward a complete solution, by repeatedly choosing a value for another variable.

Consistency techniques

The late detection of inconsistency is the disadvantage of GT and BT paradigms. Therefore various consistency techniques for constraint graphs were introduced to prune the search space. The consistency-enforcing algorithm makes any partial solution of a small sub-network extensible to some surrounding network. Thus, the inconsistency is detected as soon as possible. The consistency techniques range from simple *node-consistency* and the very popular *arc-consistency* to full, but expensive *path consistency*.

Constraint propagation

By integrating systematic search algorithms with consistency techniques, it is possible to get more efficient constraint satisfaction algorithms. Improvements of backtracking algorithm have focused on two phases of the algorithm: moving forward (*forward checking* and *look-ahead* schemes) and backtracking (*look-back* schemes).

Variable and value ordering

The efficiency of search algorithms which incrementally attempts to extend a partial solution depends considerably on the order in which variables are considered for instantiations. Having selected the next variable to assign a value to, a search algorithm has to select a value to assign. Again, this ordering effects the efficiency of the algorithm. There exist various general heuristics for dynamic or static ordering of values and variables.

Reducing search

The problem of most systematic search algorithms based on backtracking is the occurrence of many "backtracks" to alternative choices, which degrades the efficiency of the system. In some special cases, it is possible to completely eliminate the need for backtracking. Also, there exist algorithms which reduce the backtracking by choosing a special variable ordering.

Constraint optimisation

A typical real-life problem is not only about finding a solution satisfying all the constraints. Frequently, some optimisation is involved and the customers are asking for good solutions, whatever “good” mean. The optimisation nature of the problem can be encoded by an objective function defined in terms of problem variables. Many constraint satisfaction algorithms can be then extended to solve optimisation problems. The most widely used technique is called branch-and-bound.

Systematic Search

How to solve constraint satisfaction problems by search?

Many algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is insoluble. Thus the systematic search algorithms are sound and complete. The main disadvantage of these algorithms is that they take a very long time to do so.

There are two main classes of systematic search algorithms:

- algorithms that search the *space of complete assignments*, i.e., the assignments of all variables, till they find the complete assignment that satisfies all the constraints, and
- algorithms that *extend a partial consistent assignment to a complete assignment* that satisfies all the constraints.

In this section we present basic representatives of both classes. Although these algorithms look simple and non-efficient they are very important because they make the foundation of other algorithms that exploit more sophisticated techniques like propagation or local search.

Generate and Test (GT)

Generate-and-test method originates from the mathematical approach to solving combinatorial problems. It is a typical representative of algorithms that search the space of complete assignments.

First, the GT algorithm generates some complete assignment of variables and, then, it tests whether this assignment satisfies all the constraints. If the test fails, i.e., there exists any unsatisfied constraint, then the algorithm tries another complete assignment. The algorithm stops as soon as a complete assignment satisfying all the constraints is found, this is the solution of the problem, or all complete assignments are explored, i.e., the solution does not exist

The GT algorithm search systematically the space of complete assignments, i.e., it explores each possible combination of the variable assignments. The number of combinations considered by this method is equal to the size of the Cartesian product of all the variable domains.

Algorithm GT:

```
procedure GT(Variables, Constraints)
  for each Assignment of Variables do  % generator
    if consistent(Assignment, Constraints) then
       return Assignment
  end for
  return fail
end GT

procedure consistent(Assignment, Constraints) % test
  for each C in Constraints do
    if C  is not satisfied by Assignment then
      return fail
  end for
  return true
end consistent
```

The above algorithm schema is parameterised by the procedure for generation of complete assignments of variables. The pure form of GT algorithm uses trivial generator that returns all complete assignments in some specified order. This generator assumes that the variables and values in domains are ordered in some sense. The first complete assignment is generated by assigning first value from respective domain to each variable. The next complete assignment is derived from given assignment by finding such variable X that all following variables (in given order) are labelled by the last value from their respective domains (let Vs be the set of these variables). Then, the generator assigns next value from the domain D_x to the variable X, a first value from respective domains to each variable in Vs and the rest variables hold their values. If such variable X does not exist, i.e., all variables are labelled by last values in their respective domains, then the algorithm returns *fail* indicating that there is no other assignment.

Pure generator of complete assignments for GT:

```
procedure generate_first(Variables)
  for each V in Variables do
    label V by the first value in D_V
  end for
end generate_first

procedure generate_next(Assignment)
  find first X in Assignment such that
    all following variables are labelled by
    the last value from their respective domains
    (name the set of these variables Vs)
  if X is labelled by the last value then
    return fail
  label B by next value in D_X
  for each Y in Vs do
    assign first value in D_Y to Y
  end for
end generate_next
```

Example:

```
Domains: D_X=[1,2,3], D_Y=[a,b,c], D_Z=[5,6]
First assignment: X/1, Y/a, Z/5
Other assignments are generated in the following order:
     X/1,Y/a,Z/6
     X/1,Y/b,Z/5
     X/1,Y/b,Z/6
     ...
     X/3,Y/c,Z/6
```

Disadvantages: The pure generate-and-test approach is not very efficient because it generates many wrong assignments of values to variables which are rejected in the testing phase. In addition, the generator leaves out the conflicting instantiations and it generates other assignments independently of the conflict (a blind generator). There are two ways how to improve the pure GT approach:

- The generator of assignments is smart, i.e., it generates the next complete assignment in such a way that the conflict found by the test phase is minimised. This is a basic idea of stochastic algorithms based on local search that are not covered by this tutorial.
- Generator is merged with the tester, i.e., the validity of the constraint is tested as soon as its respective variables are instantiated. In fact, this method is used by the backtracking approach.

Backtracking (BT)

The most common algorithm for performing systematic search is backtracking. Backtracking incrementally attempts to extend a partial assignment that specifies consistent values for some of the variables, toward a complete assignment, by repeatedly choosing a value for another variable consistent with the values in the current partial solution.

Backtracking can be seen as a merge of generate and test phases from the GT approach. In the BT method, variables are instantiated sequentially and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial assignment violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial assignment violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test, however, its running complexity for most nontrivial problems is still exponential.

The basic form of backtracking algorithm is called *chronological backtracking*. If this algorithm discovers an inconsistency then it always backtracks to the last decision, therefore chronological.

Algorithm (chronological) BT:

```
procedure BT(Variables, Constraints)
  BT-1(Variables,{},Constraints)
end BT

procedure BT-1(Unlabelled, Labelled, Constraints)
  if Unlabelled = {} then return Labelled
  pick first X from Unlabelled
  for each value V from D_X do
    if consistent({X/V}+Labelled, Constraints) then
       R <- BT-1(Unlabelled-{X}, {X/V}+Labelled, Constraints)
       if R # fail then return R
    end if
  end for
  return fail % backtrack to previous variable
end BT-1

procedure consistent(Labelled, Constraints)
  for each C in Constraints do
    if all variables from C are Labelled then
      if C is not satisfied by Labelled then
        return fail
  end for
  return true
end consistent
```

Again, the above algorithm schema for chronological backtracking can be parameterised. It is possible to plug-in various procedures for choosing the unlabelled variable (variable ordering) and for choosing the value for this variable (value ordering). It is also possible to use more sophisticated consistency test that discovers inconsistencies earlier then the above procedure. We will discuss all these possibilities later.

Disadvantages: There are three major drawbacks of the standard (chronological) backtracking scheme.

- The first drawback is **thrashing**, i.e., repeated failure due to the same reason. Thrashing occurs because the standard backtracking algorithm does not identify the real reason of the conflict, i.e., the conflicting variables. Therefore, search in different parts of the space keeps failing for the same reason. Thrashing can be avoided by *backjumping*, sometimes called *intelligent backtracking*, i.e., by a scheme on which backtracking is done directly to the variable that caused the failure.
- The other drawback of backtracking is having to perform **redundant work**. Even if the conflicting values of variables are identified during the intelligent backtracking, they are not remembered for immediate detection of the same conflict in a subsequent computation. The methods to resolve this problem are called *backchecking* or *backmarking*. Both algorithms are useful methods for reducing the number of compatibility checks. There is also a backtracking based method that eliminates both of the above drawbacks of backtracking. This method is traditionally called *dependency-directed backtracking* and is used in truth maintenance systems. It should be noted that using advanced techniques adds other expenses to the algorithm that has to be balanced with the overall advantage of using them.
- Finally, the basic backtracking algorithm still **detects the conflict too late** as it is not able to detect the conflict before the conflict really occurs, i.e., after assigning the values to the all variables of the conflicting constraint. This drawback can be avoided by applying consistency techniques to *forward check* the possible conflicts.

Backjumping (BJ)

In the above analysis of disadvantages of chronological backtracking we identified the thrashing problem, i.e., a problem with repeated failure due to the same reason. We also outlined the method to avoid thrashing, called backjumping (Gaschnig, 1979).

The control of backjumping is exactly the same as backtracking, except when backtracking takes place. Both algorithms pick one variable at a time and look for a value for this variable making sure that the new assignment is compatible with values committed to so far. However, if BJ finds an inconsistency, it analyses the situation in order to identify the source of inconsistency. It uses the violated constraints as a guidance to find out the conflicting variable. If all the values in the domain are explored then the BJ algorithm backtracks to the most recent conflicting variable. This is the main difference from the BT algorithm that backtracks to the immediate past variable.

The following example shows the advantage of backjumping over chronological backtracking. It displays a board situation in the typical 8-queens problem. We have allocated first five queens by respective columns (each queen is in different row) and now we are looking for a consistent column position for the 6th queen. Unfortunately, each position is inconsistent with the assignment of the first five queens so we have to backtrack. The chronological backtracking backtracks to the Queen 5 and it finds another column for this queen (column H). However, it is still impossible to place the Queen 6 because, in fact, the conflict is with the Queens 4.

The backjumping is more "intelligent" in discovering the real conflict. The numbers in row 6 indicate the assigned queens that the corresponding squares are incompatible with. It is possible at this stage to realise that changing the value of Queen 5 will not resolve the conflict. The closest queen that can resolve the conflict is Queen 4 because then there is a chance that column D can be used for Queen 6.

	A	B	C	D	E	F	G	H
1	Q							
2			Q					
3					Q			
4		Q						
5				Q				
6	1	3,4	2,5	4,5	3,5	1	2	3
7								
8								

The question is how to identify the most recent conflicting variable in general.

level	Violated constraints			
	C_1	C_2	C_3	C_4
1		X		
2	X		X	-O-
3	X	--O--		
4			O	
5	O			
6				
7	X	X	X	X
value	**A**		**B**	

The above figure sketches the situation when a value is being assigned to the 7th variable. There are two possible values, A and B, and there exist two violated constraints for both values. The figure shows which variables are bound by respective constraints. These variables are marked by "X" and "O". For each constraint, a variable at the highest level is selected as the closest conflicting variable (to currently labelled variable). This variable is marked "O". We have to choose this variable because if its value is changed then it could be possible to satisfy the constraint. Now, we choose a conflicting level for each value of the 7th variable. It is the minimal level of conflicting variables for constraints violated by given value of the variable (marked "-O-"). This is because we need all constraints to be satisfied, e.g., even if the value of 5th variable is changed to satisfy the constraint C_1 we need to satisfy the constraint C_2 as well, and consequently the value of 3rd variable has to be changed too. Finally, the conflicting level for the 7th variable is found as the maximum of conflicting levels for each value of the variable (marked "--O--"). There is a direct consequence of the above method, namely, if there exist a consistent value for the variable then the algorithm jumps to the previous level upon backtracking.

There also exist less expensive methods of finding conflicting level, for example graph-based backjumping that jumps to the most recent variable that constraints the current variable (the 5th variable in the above example). However, graph-based backjumping behaves in the same way as chronological backtracking if each variable is constrained by every other variable (like in the N-Queens problem). Fortunately, for many problems, the constraint graphs are not complete

Algorithm BJ:

```
procedure BJ(Variables, Constraints)
  BJ-1(Variables,{},Constraints,0)
end BJ

procedure BJ-1(Unlabelled, Labelled, Constraints, PreviousLevel)
  if Unlabelled ={} then return Labelled
  pick first X from Unlabelled
  Level <- PreviousLevel+1
  Jump <- 0
  for each value V from Dx do
    C <- consistent({X/V/Level}+Labelled, Constraints, Level)
    if C = fail(J) then
      Jump <- max {Jump, J}
    else
      Jump <- PreviousLevel
      R <- BJ-1(Unlabelled-{X},{X/V/Level}+Labelled,
                                      Constraints, Level)
      if R # fail(Level) then return R
                         % success or backtrack to past level
    end if
  end for
  return fail(Jump) % backtrack to conflicting variable
end BJ-1

procedure consistent(Labelled, Constraints, Level)
  J <- Level
  NoConflict <- true
  for each C in Constraints do
    if all variables from C are Labelled then
      if C is not satisfied by Labelled then
        NoConflict <- false
        J <- min {{J}+ max{L | X in C & X/V/L in Labelled & L<Level}}
      end if
  end for
  if NoConflict then return true
  else return fail(J)
end consistent
```

Backmarking (BM)

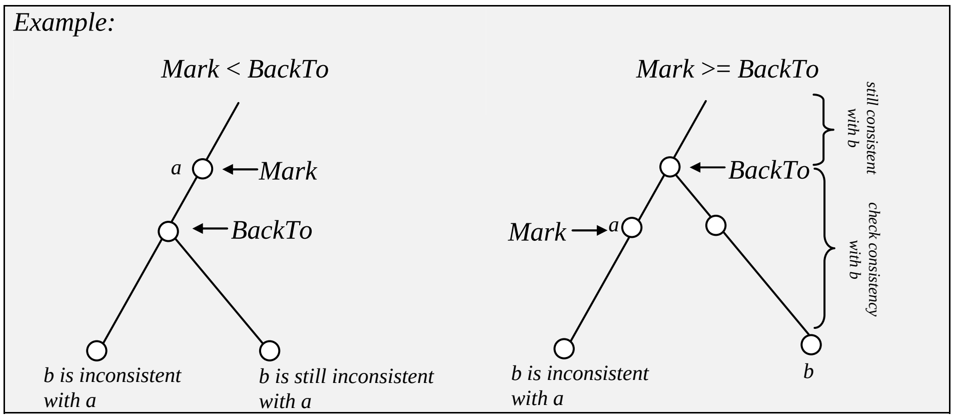
In the above analysis of disadvantages of chronological backtracking we identified the problem with redundant work, i.e., even if the conflicting values of variables are identified during the intelligent backtracking, they are not remembered for immediate detection of the same conflict in a subsequent computation. We mentioned two methods to resolve this problem, namely backchecking (BC) and backmarking (BM).

Both backchecking and its descendent backmarking are useful algorithms for reducing the number of compatibility checks. If the backchecking finds that some label Y/b is incompatible with any recent label X/a then it remembers this incompatibility. As long as X/a is still committed to, the Y/b will not be considered again.

Backmarking (Haralick, Elliot, 1980) is an improvement over backchecking that avoids some redundant constraint checking as well as some redundant discoveries of inconsistencies. It reduces the number of compatibility checks by remembering for every label the incompatible recent labels. Furthermore, it avoids repeating compatibility checks which have already been performed and which have succeeded.

To simplify the description of backmarking algorithm we assume working with binary CSPs (note, that it is not restriction because each CSP can be converted to equivalent binary CSP – see References). The idea of backmarking is as follows. When trying to extend a search path by choosing a value for a variable X, backmarking marks the individual level, *Mark*, in the search tree at which an inconsistency is detected for each value of X. If no inconsistency is detected for a value, its *Mark* is set to the level above the level of the variable X. In addition, the algorithm also remembers the highest level, *BackTo*, to which search has backed up since the last time X was considered. Now, when backmarking next considers a value V for X, the *Mark* and *BackTo* levels can be compared. There are two cases:

- *Mark* < *BackTo*. If the level at which V failed before is above the level to which we have backtracked, we know, without further constraint checking, that V will fail again. The value it failed against is still there.
- *Mark* >= *BackTo*. If since V last failed we have backed up to or above the level at which V encountered failure, we have to test V. However, we can start testing values against V at level *BackTo* because the values above that level are unchanged since we last successfully tested them against V.



The following figure demonstrates how the values of global variables (arrays) *Mark* and *BackTo* are computed. We use the 8-Queens problems again and the board shows the same situation as in the backjumping example. Note, that computating of *Mark* value is similar to finding the conflicting level in backjumping and that backjumping can naturally be combined with backmarking to improve the efficiency without additional overhead. The board situation shows the values of *Mark* (the number at each square) and *BackTo* at the state when all the values for Queen 6 have been rejected, and the algorithm backtracks to Queen 5 (therefore *BackTo*(6)=5). If and when all the values of Queen 5 are rejected, both *BackTo*(5) and *BackTo*(6) will be changed to 4.

	A	B	C	D	E	F	G	H	*BackTo*
1	Q								1
2	1	1	Q						1
3	1	2	1	2	Q				1
4	1	Q							1
5	1	4	2	Q					1
6	1	3	2	4	3	1	2	3	5
7									1
8									1

Algorithm BM:

```
procedure BM(Variables, Constraints)
  INITIALIZE(Variables)
  BM-1(Variables,{},Constraints,1)
end BM

procedure INITIALIZE(Variables)
  for each X in Variables do
    BackTo(X) <- 1
    for each V from $D_X$ do
      Mark(X,V) <- 1
    end for
  end for
end INITIALIZE

procedure BM-1(Unlabelled, Labelled, Constraints, Level)
  if Unlabelled ={} then return Labelled
  pick first X from Unlabelled  % now, the order is fixed
  for each value V from $D_X$ do
    if Mark(V,X) >= BackTo(X) then
      if consistent(X/V, Labelled, Constraints, Level) then
        R <- BM-1(Unlabelled-{X}, Labelled+{X/V/Level},
                                    Constraints, Level+1)
        if R # fail then return R  % success
      end if
    end if
  end for
  BackTo(X) <- Level-1
  for each Y in Unlabelled do
    BackTo(Y) <- min {Level-1, BackTo(Y)}
  end for
  return fail % backtrack to recent variable
end BM-1

procedure consistent(X/V, Labelled, Constraints, Level)
  for each Y/VY/LY in Labelled such that LY>=BackTo(X) do
    % in increasing order of LY
    if X/V is not compatible with Y/VY using Constraints then
      Mark(X,V) <- LY
      return fail
    end if
  end for
  Mark(X,V) <- Level-1
  return true
end consistent
```

Further Reading

A nice survey on depth-first search techniques is (Dechter, Frost, 1998); also the book (Dechter, 2003) contains nicely written chapters on search technique in constraint satisfaction. Backjumping has been introduced in (Gaschnig, 1979); its further improvement called dynamic backtracking was proposed in (Ginsberg, 1993). Backmarking is described in (Haralick, Elliot, 1980).

In addition to complete depth-first search techniques, there also exist many incomplete techniques like depth-bounded search (Beldiceanu *et al*, 1997), credit search (Cheadle *et al*, 2003) or iterative broadening (Ginsberg, Harvey, 1990).

Recently, techniques for recovery from a failure of value ordering heuristic called discrepancy search became popular thanks to good practical applicability. These are techniques like limited discrepancy search (Harvey, Ginsberg, 1995), improved limited discrepancy search (Korf, 1996), discrepancy-bounded depth first search (Beck, Perron, 2000), interleaved depth-first search (Meseguer, 1997), and depth-bounded discrepancy search (Walsh, 1997). A survey can be found in (Harvey, 1995) or (Barták, 2004).

Consistency Techniques[1]

Can constraints be used more actively during constraint satisfaction?

Consistency techniques were first introduced for improving the efficiency of picture recognition programs, by researchers in artificial intelligence (Montanari, 1974), (Waltz, 1975). 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 assignments at a very early stage, and thus cut short the search for consistent assignment. These techniques have since proved to be effective on a wide variety of hard search problems.

Example:

Let $A<B$ be a constraint between the variable A with the domain $D_A=3..7$ and the variable B with the domain $D_B=1..5$.

Visibly, for some values in D_A there does not exist a consistent value in D_B satisfying the constraint $A<B$ and vice versa. Such values can be removed from respective domains without loss of any solution, i.e., the reduction is safe. We get reduced domains $D_A=\{3,4\}$ and $D_B=\{4,5\}$.

Note, that this reduction does not remove all inconsistent pairs necessarily, for example $A=4$, $B=4$ is still in domains, but for each value of A from D_A it is possible to find a consistent value of B and vice versa.

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

In the following, we expect that a binary CSP is represented as a constraint graph where each node is labelled by the variable and the edge between two nodes corresponds to the binary constraint binding the variables that label the nodes connected by the edge. Unary constraint can be represented by the cycle edge.

Node Consistency (NC)

The simplest consistency technique is referred to as node consistency (NC).

Definition: The node representing a variable X in a constraint graph is **node consistent** if and only if for every value V in the current domain D_X of X, each unary constraint on X is satisfied. A **CSP is node consistent** if and only if all variables are node consistent, i.e., for all variables all values in its domain satisfy the constraints on that variable.

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

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

Algorithm NC:

```
procedure NC(G)
  for each variable X in nodes(G) do
    for each value V in the domain D_X do
      if unary constraint on X is inconsistent with V then
        delete V from D_X
    end for
  end for
end NC
```

Arc Consistency (AC)

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 (AC).

Definition: An arc (V_i,V_j) is **arc consistent** if and only for every value x in the current domain of V_i which satisfies the constraints on 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, then it does not automatically mean that (V_j,V_i) is also consistent. A **CSP is arc consistent** if and only if every arc (V_i,V_j) in its constraint graph is arc 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 a 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:

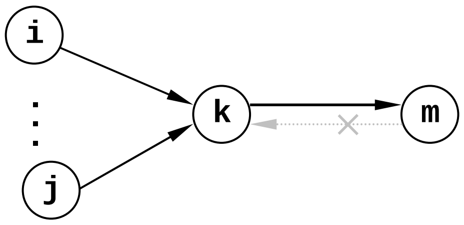
```
procedure REVISE(Vi,Vj)
  DELETED <- false
  for each X in Di do
    if there is no such Y in Dj such that (X,Y) is consistent, i.e.,
              (X,Y) satisfies all the constraints on Vi, Vj then
      delete X from Di
      DELETED <- true
    end if
  end for
  return DELETED
end REVISE
```

To make every arc of the constraint graph consistent, i.e., to make a corresponding CSP arc 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 easiest way how to establish arc consistency is to apply the REVISE procedure to all arcs repeatedly till the domain of any variable changes. The following algorithm, known as **AC-1**, does exactly this (Mackworth, 1977).

Algorithm AC-1:

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

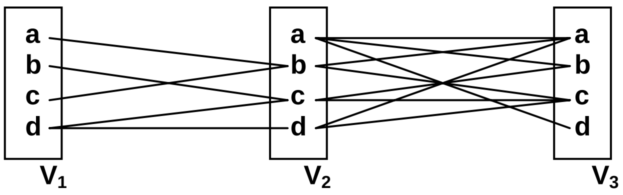
The AC-1 algorithm is not very efficient because the successful revision of even a single arc in some iteration forces all the arcs to be revised again in the next iteration, even though only a small number of them are really affected by this revision. Visibly, the only arcs affected by the reduction 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** (Mackworth, 1977), 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(G)
  Q <- {(Vi,Vj) in arcs(G), i#j}
  while not empty Q do
    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}
    end if
  end while
end AC-3
```

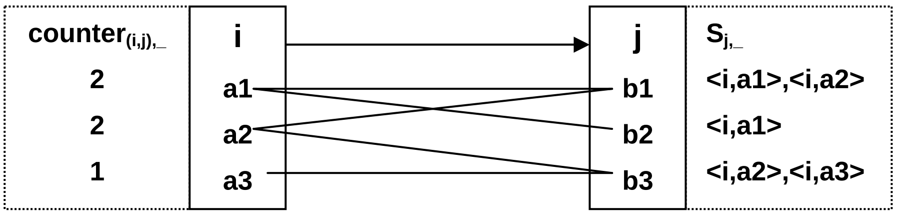
When the algorithm AC-3 revises the edge for the second time it re-tests many pairs of values which are already known (from the previous iteration) to be consistent or inconsistent respectively and which are not affected by the reduction of the domain. The idea behind AC-3 is based on the notion of *support*; a value is supported if there exists a compatible value in the domain of every other variable. When a value V is removed from the domain of the variable X, it is not always necessary to examine all the binary constraints $C_{Y,X}$. Precisely, we can ignore those values in D_Y which do not rely on V for support (in other words, those values in D_Y that are compatible with some other value in D_X other than V). The following figure shows these dependencies between pairs of values. If we remove the value *a* from the domain of variable V_2 (because it has no support in V_1), we do not need to check values *a, b, c* from the domain of V_3 because they all have other supports in the domain of V_2. However, we have to remove the value *d* from the domain of V_3 because it lost the only support in V_2.



Checking pairs of values again and again is a source of potential inefficiency. Therefore the algorithm **AC-4** (Mohr, Henderson, 1986) was introduced to refine handling of edges (constraints). The algorithm works with individual pairs of values using support sets for each value. First, the algorithm AC-4 initialises its internal structures which are used to remember pairs of consistent (inconsistent) values of incidental variables (nodes) - structure $S_{i,a}$ representing set of supports. This initialisation also counts "supporting" values from the domain of incidental 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(G)
  Q <- {}
  S <- {}  % initialize each element in the structure Sx,v
  for each arc (Vi,Vj) in arcs(G) do
    for each a in Di do
      total <- 0
      for each b in Dj do
        if (a,b) is consistent according to the constraint Ci,j then
          total <- total + 1
          Sj,b <- Sj,b union {<i,a>}
        end if
      end for
      counter[(i,j),a] <- total
      if counter[(i,j),a] = 0 then
        delete a from Di
        Q <- Q union {<i,a>}
      end if
    end for
  end for
  return Q
end INITIALIZE
```



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

Algorithm AC-4:

```
procedure AC-4(G)
  Q <- INITIALIZE(G)
  while not empty Q do
    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>}
      end if
    end for
  end while
end AC-4
```

Directional Arc Consistency (DAC)

In the above definition of arc consistency we mentioned a directional nature of arc consistency (for arc). Nevertheless, the arc consistency for CSP is not directional at all as each arc is assumed in both directions in the AC-x algorithms. Although, the node and arc consistency algorithms seem easy they are still stronger than necessary for some problems, for example, for enabling backtrack-free search in CSPs which constraints form trees. Therefore yet simpler concept was proposed to achieve some form of consistency, namely, directional arc consistency (DAC) that is defined under total ordering of the variables.

Definition: A **CSP is directional arc consistent** under an ordering of variables if and only if every arc (V_i,V_j) in its constraint graph such that i<j according to the ordering is arc consistent.

Notice the difference between AC and DAC, in AC we check every arc (V_i,V_j) while in DAC only the arcs (V_i,V_j) where i<j are considered. Consequently, the arc consistency is stronger than directional arc consistency, i.e., arc consistent CSP is also directional arc consistent but not vice versa (directional arc consistent CSP is not necessarily arc consistent as well).

The algorithm for achieving directional arc consistency is easier and more efficient than AC-x algorithms. In fact, each arc is examined exactly once in the following algorithm DAC-1.

Algorithm DAC-1:

```
procedure DAC-1(G)
  for j = |nodes(G)| to 1 by -1 do
    for each arc (Vi,Vj) in arcs(G) such that i<j do
      REVISE(Vi,Vj)
    end for
  end for
end DAC-1
```

The DAC-1 procedure potentially removes fewer redundant values than the algorithms already mentioned which achieve AC. However, DAC-1 requires less computation than procedures AC-1 to AC-3, and less space than procedure AC-4. The choice of achieving AC or DAC is domain dependent. In principle, more values can be removed through constraint propagation in more tightly constraint problems. Thus AC tends to be worth achieving in more tightly constrained problems.

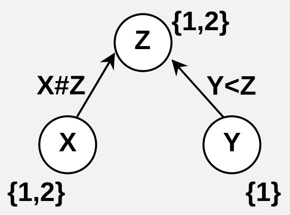
The directional arc-consistency is sufficient for backtrack-free solving of CSPs which constraints form trees. In this case, it is possible to order the nodes (variables) starting from the tree root and concluding at tree leaves. If the graph (tree) is made directional arc-consistent using this order then it is possible to find the solution by assigning values to variables in the same order. The directional arc-consistency guarantees that for each value of the root (parent) we will find consistent values in daughter nodes and so on till the values of the leaves. Consequently, no backtracking is necessary to find a complete consistent assignment.

From DAC to AC

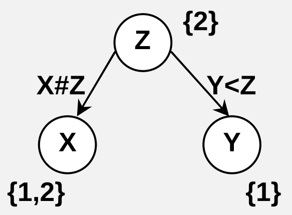
Notice, that a CSP is arc-consistent if, for any given ordering < of the variables, this CSP is directional arc-consistent under both < and its reverse. Therefore, it is tempting to believe (wrongly) that arc-consistency could be achieved by running DAC-1 in both directions for any given <. The following simple example shows that this belief is a fallacy.

Example:

If the DAC-1 is applied to the following graph using variable ordering X,Y,Z, the domains of respective variables do not change.



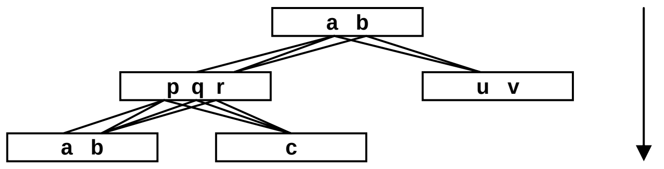
Now, if the DAC-1 is applied using reverse order Z,Y,X, the domain of variable Z changes only but the resulting graph is still not arc-consistent (the value 2 in D_X is inconsistent with Z).



Nevertheless, in some cases it is possible to achieve arc-consistency by running DAC algorithm in both directions for particular ordering of variables. In particular, if DAC is applied to a tree graph using the ordering starting from the root and concluding at leaves and, subsequently, the DAC is applied in the opposite direction then we achieve full arc-consistency. In the above example, if we apply DAC under the ordering Z,Y,X (or Y,X,Y) and, subsequently, in opposite direction, we get an arc-consistent graph.

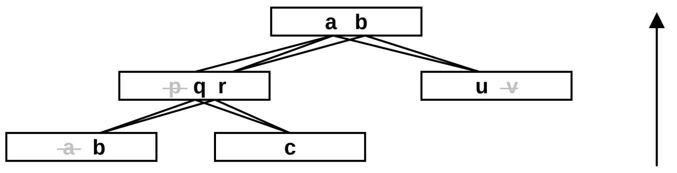
Proposition: If DAC is applied to a tree graph using the ordering starting from the root and concluding at leaves and, subsequently, the DAC is applied in the opposite direction, then we achieve full arc-consistency.

Proof: If the first run of DAC is finished then all values in any parent node are consistent with some assignment of daughter nodes. In other words, for each value of the parent node there exists at least one support (consistent value) in each daughter node.



Now, if the second run of the DAC is performed in the opposite direction and some value is removed from a node then this value is not a support of any value of the parent node (this is the reason why this value is removed, it has no support in the parent node and, consequently, it is not a support of any value from the parent node). Consequently, removing a value from some node does not evoke losing support of any value of the parent node.

The conclusion is that each value in some node is consistent with any value in each daughter node (first run) and with any value in the parent node (second run) and, therefore the graph is arc consistent.



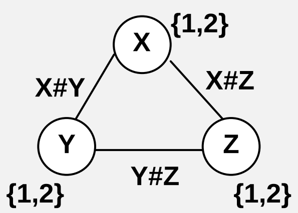
Q.E.D.

Is Arc-Consistency sufficient?

Achieving 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? Or can we at least prove that the solution exists?

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 (this holds for AC and DAC as well). If any domain becomes empty, then the CSP has no solution. Otherwise, the answer to above questions 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 does not exist any labelling that satisfies all the constraints.

A CSP after achieving arc-consistency:

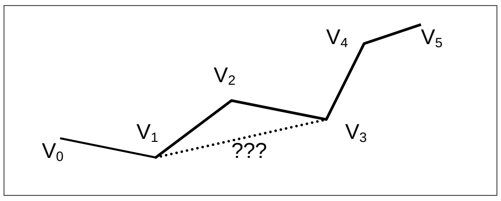
1) domain size for each variable becomes one => exactly one solution exists
2) any domain becomes empty => no solution exists
3) otherwise => ???

Path Consistency (PC)

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. This is the main idea of path-consistency.

Definition: A path (V_0, V_1,..., V_m) in the constraint graph for CSP is **path consistent** if and only if for every pair of values x in D_0 and y in D_m that satisfies all the constraints on V_0 and V_m there exists a label for each of the variables V_1,..., V_{m-1} such that every binary constraint on the adjacent variables V_i, V_{i+1} in the path is satisfied. A **CSP is path consistent** if and only if every path in its graph is path consistent.

Note carefully that the definition of path consistency for the path (V_0, V_1,..., V_m) does not require the values x_0, x_1,..., x_m to satisfy all the constraints between variables V_0, V_1,..., V_m, In particular, the variables V_1, V_3 are not adjacent variables in the path (V_0, V_1,..., V_m), so the values x_1, x_3 needs not satisfy the constraint between V_1, V_3.



Naturally, a path consistent CSP is arc consistent as well because an arc is equivalent to the path of length 1. In fact, to make the arc (V_i,V_j) arc-consistent one can make the path (V_i,V_j,V_i) path-consistent. Consequently, path consistency implies arc consistency. However, the reverse implication does not hold, i.e., arc consistency does not imply the path consistency as the above example shows (if we make the graph path consistent, we discover that the problem has no solution). Therefore, path consistency is stronger than arc consistency.

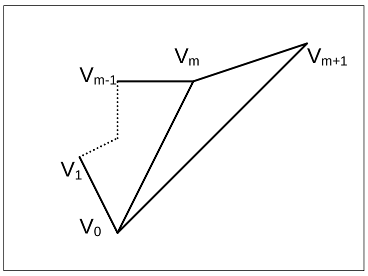
There is an important proposition about path consistency that simplifies maintaining path consistency. In 1974, Montanary pointed out that if every path of length 2 is path consistent then the graph is path consistent as well. Consequently, we can check only the paths of length 2 to achieve full path consistency.

Proposition: A CSP is path consistent if and only if all paths of length 2 are path consistent.

Proof: Path consistency for paths of length 2 is just a special case of full path consistency so the implication *path-consistent => path-consistent for paths of length 2* (1) is trivially true.

The other implication *path-consistent <= path-consistent for paths of length 2* (2) can be proved using induction on the length of the path.

1. **Base Step:** When the length of path is 2 then the above implication (2) holds (trivial).
2. **Induction Step:** Assume that the implication (2) is true for all paths with length between 2 and some integer *m*. Pick any two variables V_0 and V_{m+1} and assume that x_0 in D_0 and x_{m+1} in D_{m+1} are two values that satisfy all the constraints on V_0 and V_{m+1}. Now pick any *m* variables V_1,..., V_m. There must exist some value x_m in D_m such that all the constraints on the V_0, V_m and V_m, V_{m+1} are satisfied (according to the base step). Finally, there must exists a label for each of the variables V_1,..., V_{m-1} such that every binary constraint on the adjacent edges in the path (V_0, V_1,..., V_m) is satisfied (according to the base step; we can assume that x_m satisfies all unary constraints on V_m). Consequently, every binary constraint on the adjacent edges in the path (V_0, V_1,..., V_{m+1}) is also satisfied and the path (V_0, V_1,..., V_{m+1}) is path-consistent.



Q.E.D.

Algorithms which achieve path-consistency remove not only inconsistent values from the domains but also inconsistent pairs of values from the constraints (remind that we are working with binary CSPs). The binary constraint is represented here by a {0,1}-matrix where value 1 represents legal, consistent pair of values and value 0 represents illegal, inconsistent pair of values. For uniformity, both the domain and the unary constraint of a variable X is also represented using the {0,1}-matrix. In fact, the unary constraint on X is represented in the form of a binary constraint on (X, X).

Example:

The binary constraint A>B+1 on variables A and B with respective domains $D_A=\{3,4,5\}$ and $D_B=\{1,2,3\}$ can be represented by the following {0,1}-matrix:

$$\begin{array}{c|ccc|}
{}^{B}_{A} & 1 & 2 & 3 \\
3 & 1 & 0 & 0 \\
4 & 1 & 1 & 0 \\
5 & 1 & 1 & 1
\end{array}$$

The domain of variable A, $D_A=\{3,4,5\}$, and the unary constraint A>3 can also be represented by the {0,1}-matrix (in this case, the matrix is diagonal):

$$\begin{array}{c|ccc|}
{}^{A}_{A} & 3 & 4 & 5 \\
3 & 0 & 0 & 0 \\
4 & 0 & 1 & 0 \\
5 & 0 & 0 & 1
\end{array}$$

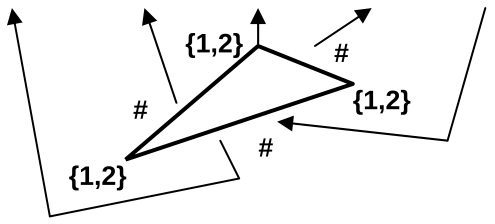
Note, that we need to know the exact order of variables in the binary constraint as well the order of values in respective domains.

Now, using the matrix representation it is easier to compose constraints. This *constraint composition* is a kernel of the path consistency algorithms because to achieve path consistency in path (X,Y,Z) we can compose the constraint on (X,Y) with the constraint on (Y,Z) and make an intersection of this composition with the constraint on (X,Z). In fact, the composition of two constraints is equivalent to multiplication of {0,1}-matrices using binary operations AND, OR instead of *, + and the intersection of the matrices corresponds to performing AND operation on respective elements of the matrices. Therefore we use * to mark the composition operation and & to mark the intersection. More formally, let $C_{X,Y}$ be a {0,1}-matrix representing constraint on X and Y. Then we can make the path (X,Y,Z) path consistent by the following assignment:

$$C_{X,Z} \leftarrow C_{X,Z} \,\&\, C_{X,Y} * C_{Y,Y} * C_{Y,Z}$$

Example:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \& \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$



Naturally, the composition operation has to be performed for all pairs (X, Z) and for all intermediary nodes Y. Similarly to arc consistency, to make every path of the constraint graph consistent, i.e., to make the corresponding CSP path consistent, it is not sufficient to execute this composition operation for each path (X,Y,Z) just once. Once a domain of a variable/constraint is reduced then it is possible that some previously revised path has to be revised again, because some pairs of values become incompatible due to missing value of intermediary node. The easiest was how to establish path consistency is to apply the composition operations to all paths repeatedly till the domain of any variable/constraint changes. The following naive algorithm **PC-1** does exactly this (Mackworth, 1977).

Algorithm PC-1:

```
procedure PC-1(Vars, Constraints)
  n <- |Vars|
  Y(n) <- Constraints    % we use the {0,1}-matrix representation
  % Y(k)(i,j) represents a matrix for constraint Ci,j in k-th step
  repeat
    Y(0) <- Y(n)
    for k=1 to n do
      for i=1 to n do
        for j=1 to n do
          Y(k),(i,j) <- Y(k-1),(i,j) &
                        Y(k-1),(i,k)* Y(k-1),(k,k)* Y(k-1),(k,j)
  until Y(n)=Y(0)
  Constraints <- Y(n)
end PC-1
```

The basic idea of PC-1 is as follows: for every variable V_k, pick every constraint $C_{i,j}$ from the current set of constraints Y^k and attempt to reduce it by means of relations composition using $C_{i,k}$, $C_{k,k}$ and $C_{k,j}$. After this is done for all variables, the set of constraints is examined to see if any constraint in it has changed. The whole process is repeated as long as some constraints have been changed. Note that $Y^k(i,j)$ represents the constraint $C_{i,j}$ in the set Y^k and that Y^k is only used to build Y^{k+1}.

Like AC-1, PC-1 is very inefficient because even a small change in one constraint will cause the whole set of constraints to be re-examined. Moreover, PC-1 is also very memory consuming as many arrays Y^k are stored. Therefore improved algorithm **PC-2** (Mackworth, 1977) was introduced in which only relevant constraints are re-examined.

Similarly to AC algorithms we first introduce a procedure for path revision that restricts a constraint $C_{i,j}$ using $C_{i,k}$ and $C_{k,j}$. The procedure returns TRUE, if the constraint domain is changed, and FALSE otherwise.

Algorithm REVISE PATH:

```
procedure REVISE_PATH((i,k,j), C)
  Temp <- Ci,j & (Ci,k * Ck,k * Ck,j)
  if (Temp = Ci,j) then return FALSE
  else
    Ci,j <- Temp
    return TRUE
  end if
end REVISE_PATH
```

Note, that we do not need revise path in both directions if $C_{i,j} = C^T_{j,i}$, i.e., if only one {0,1}-matrix is used to represent the constraints $C_{i,j}$ and $C_{j,i}$ (C^T is the transposition of the matrix C, i.e., rows and columns are interchanged). This is because the following deduction holds:

$$(C_{i,j} \,\&\, C_{i,k} * C_{k,k} * C_{k,j})^T = C^T_{i,j} \,\&\, (C_{i,k} * C_{k,k} * C_{k,j})^T = C^T_{i,j} \,\&\, C^T_{k,j} * C^T_{k,k} * C^T_{i,k} = C_{j,i} \,\&\, C_{j,k} * C_{k,k} * C_{k,i}$$

Now, we can use some ordering of variables and examine only paths (i,k,j) such that i=<j. Note, that there is no condition about *k* and, therefore, we do not restrict ourselves to some form of directional path consistency.

Finally, if the constraint $C_{i,j}$ is reduced in REVISE_PATH, we want to re-examine only the relevant paths. Because of above discussion about variable ordering there are two cases when the constraint $C_{i,j}$ is reduced, namely i<j and i=j.

- If i<j then all paths which contain (i,j) or (j,i) are relevant with the exception of (i,i,j) and (i,j,j) because $C_{i,j}$ will not be restricted by these paths as a result of itself being reduced.
- If i=j, i.e., the restricted path was (i,k,i), then all paths with i in it need to be re-examined, with the exception of (i,i,i) and (k,i,k). This is because neither $C_{i,i}$ nor $C_{k,k}$ will be further restricted (it was the variable V_k which has caused $C_{i,i}$ to be reduced).

The following algorithm RELATED_PATHS returns paths relevant to a given path (i,k,j). Note that *n* is equal to the number of variables in the CSP (and the numbering of variables starts in 1).

Algorithm RELATED PATHS:

```
procedure RELATED_PATHS((i,k,j))
  if (i<j) then
    return {(i,j,p) | i<=p<=n & p#j} U
           {(p,i,j) | 1<=p<=j & p#i} U
           {(j,i,p) | j<p<=n} U
           {(p,j,i) | 1<=p<i}
  else  % i.e. i=j
    return {(p,i,r) | 1<=p<=r<=n} - {(i,i,i),(k,i,k)}
  end if
end RELATED_PATHS
```

Now, it is easy to write PC-2 algorithm whose structure is very similar to AC-3 algorithm. The algorithm starts with the queue of all paths to be revised and as soon as a constraint is reduced, the relevant paths are added to the queue. As we mentioned above, the algorithm assumes ordering < among variables to further decrease the number of checked paths. Remind that this is because the reduction $C_{i,k} * C_{k,k} * C_{k,j}$ is equivalent to the reduction $C_{j,k} * C_{k,k} * C_{k,i}$.

Algorithm PC-2:

```
procedure PC-2(Vars, Constraints)
  n <- |Vars|
  Q <- {(i,k,j) | 1<=i<=j<=n & i#k & k#j}
  while Q # {} do
    select and delete any path (i,k,j) from Q
    if REVISE_PATH((i,k,j),Constraints) then
      Q <- Q U RELATED_PATHS((i,k,j))
  end while
end PC-2
```

The PC-2 algorithm is far away efficient than the PC-1 algorithm and it has also smaller memory consumption than PC-1.

Directional Path Consistency (DPC)

Similarly to weakening arc-consistency to directional arc-consistency we can weaken path-consistency to directional path consistency. The reason for doing this is also the same as in DAC. Sometimes, it is sufficient to achieve directional path-consistency which is computationally less expensive than achieving full path-consistency.

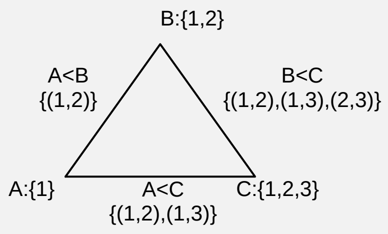
Definition: A **CSP is directional path consistent** under an ordering of variables if and only if for every two variables V_i and V_j each path (V_i,V_k,V_j) in its constraint graph such that k>i and k>j according to the ordering is path consistent.

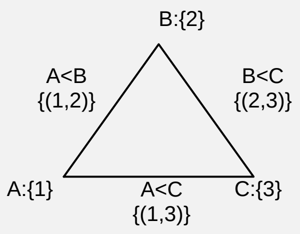
Again, notice the difference between PC and DPC. In PC we check every path (V_i,V_k,V_j) while in DPC only the paths (V_i,V_k,V_j) where k>i and k>j are considered. Consequently, the path consistency is stronger than directional path consistency; however, it is less expensive to achieve directional path consistency. The following example shows that path consistency is strictly stronger than directional path consistency, i.e., PC removes more inconsistent values than DPC. It also shows that DPC can be even weaker then AC. However, DPC is at least as strong as DAC because if path (V_i,V_k,V_i) where i<k is path-consistent then also the arc (V_i,V_k) is arc-consistent.

Example:

This CSP is directional path consistent under the ordering A,B,C of variables. However, this graph is not path consistent.

This is the same CSP after achieving full path consistency.





Similarly to DAC, the algorithm for achieving directional path-consistency is easier and more efficient than the PC algorithms. Again, the algorithm DPC-1 goes through the variables in the descending order (according to the ordering <) and each path is examined exactly once.

Algorithm DPC-1:

```
procedure DPC-1(Vars, Constraints)
  n <- |Vars|
  Q <- {(i,j) | i<j & Ci,j in Constraints}
  for k = n to 1 by -1 do
    for i = 1 to k-1 do
      for j = i to k do
        if (i,k) in Q & (j,k) in Q then
          Ci,j <- Ci,j & (Ci,k * Ck,k * Ck,j)
          Q <- Q + (i,j)
        end if
      end for
    end for
  end for
end DPC-1
```

Why not path-consistency?

Path consistency removes more inconsistencies from the constraint graph than arc-consistency but it has also many disadvantages. Here are three main reasons why path-consistency algorithms are almost never implemented in commercial CSP-solving systems:

- The ration between the complexity of PC and the simplification factor brings path-consistency far less interesting than the one brought by arc-consistency.
- PC algorithms are based on elimination of pairs of values assignments. This imposes that constraints should have an extensive representation ({0,1}-matrix) from which individual pairs can be deleted. Such a representation is often unacceptable for the implementation of real-world problems for which intensive representations are much more concise and efficient.
- Finally, enforcing path-consistency has the major drawback of bringing some modifications to the connectivity of the constraint graph by adding some edges to this graph (i.e., if a path consistency for (V_i,V_k,V_j) is enforced and there is no constraint between V_i and V_j then a new constraint between these two variables appears)

Restricted Path Consistency (RPC)

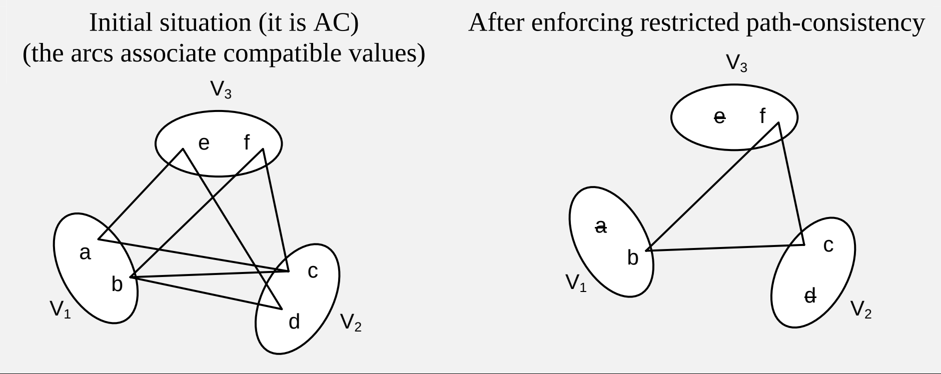
Because of above mentioned reasons, Pierre Berlandier (1995) introduced a new level of partial consistency which is situated between arc and path consistency. This level, half way between AC and PC, is called restricted path-consistency (RPC).

The procedure for enforcing restricted path-consistency turns the above three drawbacks of PC by their incompleteness: path-consistency checking is engaged for a given assignment pair if and only if the deletion of this pair implies the deletion of one of its elements. Such situation occurs when a given assignment pair represents the only support for one of the assignments with regard to some constraint. The algorithm for making a graph restricted path consistent can be naturally based on AC-4 algorithm that counts the number of supporting values.

Definition: 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 incidental variable V_j there exists a value "c" in the domain of other incidental 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 restricted path consistency removes at least the same number of inconsistent pairs as the arc-consistency does and also some pairs beyond. The following example demonstrates such case.

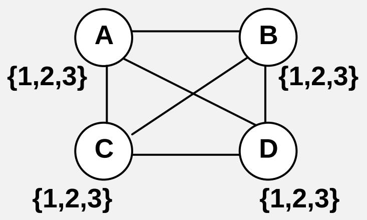
Example:


Initial situation (it is AC)
(the arcs associate compatible values)
After enforcing restricted path-consistency
V_3
e
f
a
b
c
d
V_1
V_2
V_3
~~e~~
f
~~a~~
b
c
~~d~~
V_1
V_2

Path-Consistency still not sufficient?

Enforcing path consistency removes more inconsistencies from the constraint graph than arc-consistency but is it sufficient now? The answer is unfortunately the same as for arc-consistency, i.e., achieving path-consistency still does not imply neither that any (complete) instantiation of variables from current (reduced) domains is a solution to the CSP nor that the solution exists. The following example shows such a case where the constraint graph is path consistent, domains are not empty but there is still no solution satisfying all constraints.

Example:



This constraint graph (the constraints are inequalities between respective variables) is path-consistent but there does not exist any labelling that satisfies all the constraints.

A CSP after achieving path-consistency:

- domain size for each variable becomes one => exactly one solution exists
- any domain becomes empty => no solution exists
- otherwise => ???

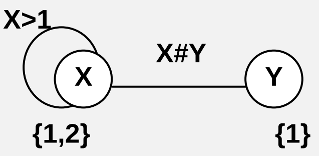
K-consistency

Because path-consistency is still not sufficient to solve the CSP in general, there remains a question whether there exists any consistency technique that can solve the CSP problem completely. Let us first define a general notion of consistency that covers node, arc, and path consistencies.

Definition: A constraint 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 K-th variable. Then there exists a value for this K-th variable that satisfies all the constraints among these K variables. A constraint graph is **strongly K-consistent** if it is J-consistent for all J<=K.

Visibly, strongly K-consistent graph is K-consistent as well. However, the reverse implication does not hold in general as the following example shows.

Example:



This constraint graph is 2-consistent but it is not 1-consistent because the value 1 of variable X does not satisfy the unary constraint X>1. Consequently, the graph is not strongly 2-consistent.

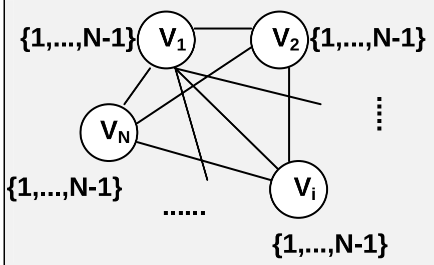
K-consistency is a general notion of consistency that covers all above mentioned consistencies (with the exception of RPC). In particular:

- node consistency is equivalent to strong 1-consistency,
- arc-consistency is equivalent to strong 2-consistency, and
- path-consistency is equivalent to strong 3-consisetncy.

Algorithms exist for making a constraint graph strongly K-consistent for K>2 but in practice they are rarely used because of efficiency issues. Although these algorithms remove more inconsistent values than any arc-consistency algorithm they do not eliminate the need for search 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 time complexity of the algorithm for obtaining N-consistency in an N-node constraint graph is exponential. If the graph is (strongly) K-consistent for K<N, then in general, backtracking (search) cannot be avoided, i.e., there still exist inconsistent values.

Example:



This constraint graph with inequality constraints between each pair of variables is strongly K-consistent for each K<N, where N is a number of nodes (variables). However, there does not exist any labelling that satisfies all the constraints.

Further Reading

Consistency techniques make the core of constraint satisfaction technology. The basic arc consistency and path consistency algorithms (AC-1,2,3, PC-1,2) are described in (Mackworth, 1997), their complexity study can be found in (Mackworth, Freuder, 1985). Algorithm AC-4 with the optimal worst-case time complexity has been proposed in (Mohr, Henderson, 1986). Its improvement called AC-6 that decreases memory consumption and improves average time complexity was proposed in (Bessiere, 1994). This algorithm has been further improved to AC-7 in (Bessiere, Freuder, Regin, 1999). AC-5 is a general schema for AC algorithms that can collapse to both AC-3 and AC-4. It is described in (Van Hentenryck *et al*, 1992). Recently, optimal versions of AC-3 algorithms have been independently proposed, namely AC-3.1 (Zhang, Yap, 2001) and AC-2001 (Bessiere, Regin, 2001).

Mohr and Henderson (1986) proposed an improved algorithm for path consistency PC-3 based on the same idea as AC-4. However, this algorithm is not sound – a correction called PC-4 is described in (Han, Lee, 1988). Algorithm PC-5 using the ideas of AC-6 is described in (Singh, 1995). Restricted path consistency that is a half way between AC and PC is described in (Berlandier, 1995).

There also exist other consistency techniques going beyond the k-consistency scheme like inverse consistencies (Verfaillie et al, 1999), neighbourhood inverse consistencies (Freuder, Elfe, 1996), or singleton consistencies (Prosser et al, 2000). Stronger consistency techniques are usually not used in practice due to their time and space complexity and most constraint solvers are built around (generalized) arc consistency. Filtering power is improved there via so called *global constraints* that encapsulate several “simple” constraints and typically achieve a stronger pruning. A typical representative of global constraints is all-different by Regin (1994).

Constraint Propagation

Can we combine depth-first search and consistency techniques?

In the previous chapters we presented two rather different schemes for solving the CSP: systematic search with chronological backtracking as its representative and consistency techniques. The systematic search was developed for general applications, and does not use constraints to improve the efficiency (backjumping and backmarking are two improvements that try to exploit constraints to reduce the search space). Opposite, the consistency techniques reduce the search space using constraints till the solution is found. Neither systematic search nor consistency techniques prove themselves to be efficient enough to solve the CSP completely. Therefore a third possible schema was introduced that embeds a consistency algorithm inside a search algorithm. Such schemas are usually called *look-ahead strategies* and they are based on idea of reducing the search space through constraint propagation.

As a skeleton we use a simple backtracking algorithm that incrementally instantiates variables and extends a partial assignment that specifies consistent values for some of the variables, toward a complete assignment, by repeatedly choosing a value for another variable. In order to reduce the search space, some consistency technique is applied to the constraint graph after assigning a value to the variable. Depending on the degree of consistency technique we get various constraint satisfaction algorithms.

Backtracking, once more

Even simple backtracking (BT) performs some kind of consistency technique and it can be seen as a combination of pure generate & test and a fraction of arc consistency. The BT algorithm tests arc consistency among already instantiated variables, i.e., the algorithm checks the validity of constraints considering the partial instantiation. Because the domains of instantiated variables contain just one value, it is enough to check only those constraints/arcs containing the last instantiated variable. If any domain is reduced then the corresponding constraint is not consistent and the algorithm backtracks to a new instantiation.

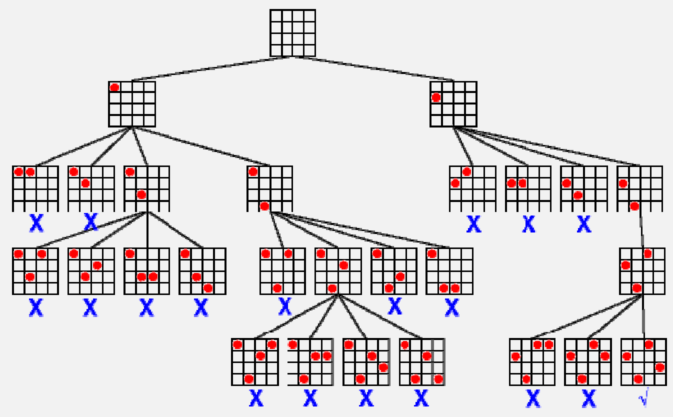
The following procedure AC-BT is called each time a new value is assigned to some variable V_{cv} (cv is the consecutive number of the variable in the order of instantiating variables).

Algorithm AC for Backtracking:

```
procedure AC-BT(cv)
  Q <- {(Vi,Vcv) in arcs(G),i<cv};
  consistent <- true;
  while not Q empty & consistent do
    select and delete any arc (Vk,Vm) from Q;
    consistent <- not REVISE(Vk,Vm)
  end while
  return consistent
end AC-BT
```

The BT algorithm detects the inconsistency as soon as it appears and, therefore, it is far away efficient than the simple generate & test approach. But it has still to perform too much search because it waits till the inconsistency really appears.

Example: (4-queens problem and BT)



As we demonstrated with backjumping and backmarking strategies, the BT algorithm can be easily extended to backtrack to the conflicting variable and, thus, to incorporate some form of look-back scheme or intelligent backtracking. Nevertheless, this adds some additional expenses to the algorithm and it seems that preventing possible future conflicts is more reasonable than recovering from them.

Forward Checking

Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency between instantiated variables, it performs arc consistency between pairs of a not-yet instantiated variable and an instantiated variable. Therefore, it maintains the invariance that for every un-instantiated variable there exists at least one value in its domain which is compatible with the values of instantiated variables.

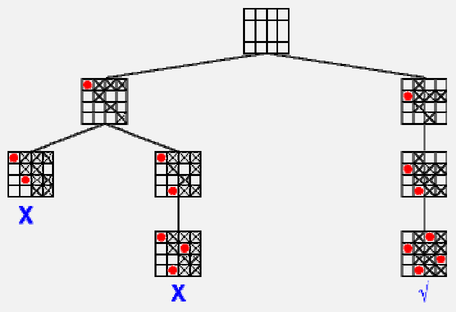
The forward checking algorithm is based on the following idea. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. Notice, that we check only the constraints/arcs between the future variables and the currently instantiated variable. The reason is that the satisfaction of other constraints between future variable and already instantiated variables does not change. If the domain of a future variable becomes empty, then it is known immediately that the current partial assignment is inconsistent. Consequently, forward checking allows branches of the search tree that will lead to a failure to be pruned earlier than with chronological backtracking. Note also that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so checking the assignment against the past assignments is no longer necessary.

Algorithm AC for Forward Checking:

```
procedure AC-FC(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent do
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      consistent <- not empty Dk
    end if
  end while
  return consistent
end AC-FC
```

Notice, that in AC-BT algorithm we use procedure REVISE as a consistency test and in AC-FC we have to test the emptiness of the domain. This is because the procedure REVISE is applied to domains containing exactly one value in AC-BT. Consequently, if the domain is reduced (REVISE returns True) then the domain becomes empty. In AC-FC, the reduction of the domain does not mean necessarily that the domain is empty (because the domain of a future variable can contain more than one value) so we have to test emptiness explicitly.

Example: (4-queens problem and FC)



Forward checking detects the inconsistency earlier than chronological backtracking and thus it reduces the search tree and (hopefully) the overall amount of work done. But it should be noted that forward checking does more work when an instantiated variable is added to the current partial assignment. Nevertheless, forward checking is still almost always a better choice than simple backtracking.

Partial Look Ahead

The more computational effort is spent on the problem reduction the more inconsistencies can be removed. Consequently, less search is necessary to find the solution. Forward checking performs only the checks of constraints between the current variable and the future variables. Now, we can extend this consistency checking to even latter variables that have not a direct connection with already instantiated variables, using directional arc-consistency. This algorithm is called **DAC-Look Ahead** or **Partial Look Ahead**.

Remind that directional arc-consistency requires some total ordering of the variables. For simplicity reasons, we will use the reverse ordering of variables from the backtracking skeleton. In practice, this is not necessary and any different orderings can be used. However, in such case, the consistency of current variable with previously assigned variables has to be checked as well.

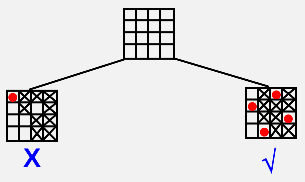
Algorithm DAC for Partial Look Ahead:

```
procedure DAC-LA(cv)
  for i=cv+1 to n do
    for each arc (Vi,Vj) in arcs(G) such that i>j & j>=cv do
      if REVISE(Vi,Vj) then
        if empty Di then return fail
    end for
  end for
  return true
end DAC-LA
```

Notice, that we check directional arc-consistency only between the future variables and between the future variables and the current variable. The reason is that the constraints between the future and past variables are not influenced by assigning a value to the current variable and therefore it is not necessary to re-check these constraints.

Partial Look Ahead checks more constraints than Forward Checking and, thus, it can find more inconsistencies than FC as the following example shows.

Example: (4-queens problem and PLA)



Full Look Ahead

In above paragraphs we showed that using directional arc-consistency can remove more values from domains of future variables than forward checking. So why not to perform full arc consistency that will further reduces the domains and removes possible conflicts? This approach is called **(Full) Look Ahead** or **Maintaining Arc Consistency** (MAC).

Similarly to partial look ahead, in full look ahead we check the constraints between the future variables and between the future variables and the current variable. However, now the constraints are checked in both directions so even more inconsistencies can be detected. Again, whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so checking the instantiated variable against the past assignments is not necessary. Also, it is not necessary to check constraints between the future and past variables because of the same reason as with partial look ahead.

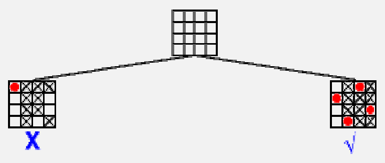
The full look ahead procedure can use arbitrary arc-consistency algorithm. In the following procedure we use the AC-3 algorithm. Notice that we start checking arc consistency with the queue containing the arcs from the future variables to the current variable only. This is because only these arcs/constraints are influenced by assigning a value to the current variable.

Algorithm AC-3 for Full Look Ahead:

```
procedure AC3-LA(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent do
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      Q <- Q union {(Vi,Vk) | (Vi,Vk) in arcs(G),i#k,i#m,i>cv}
      consistent <- not empty Dk
    end if
  end while
  return consistent
end AC3-LA
```

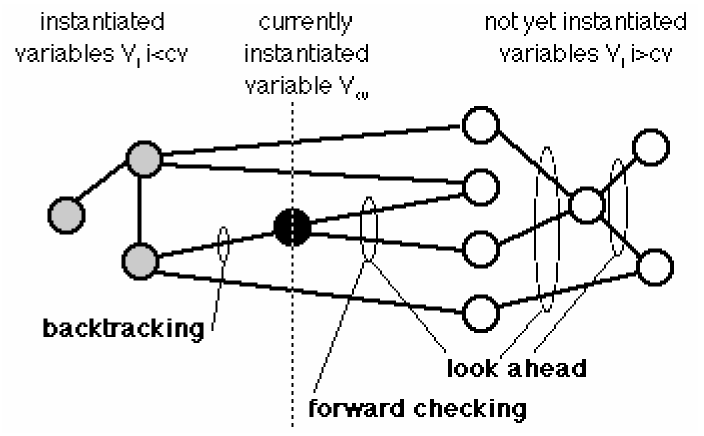
The advantage of full look ahead is that it allows branches of the search tree that will lead to a failure to be pruned earlier than with forward checking and with partial look ahead. However, it should be noted again that full look ahead does even more work when an instantiated variable is added to the current partial assignment than forward checking and partial look ahead.

Example: (4-queens problem and LA)



Comparison of propagation techniques

The constraint propagation methods can be easily compared be exploring which constraints are being checked when a value is assigned to current variable V_{cv}. The following figure shows which constraints are tested when the above described propagation techniques are applied. Note that in partial look ahead the same arcs are checked as in full look ahead. However, in partial look ahead each arc is checked exactly once.


instantiated variables V_i i<cv
currently instantiated variable V_{cv}
not yet instantiated variables V_i i>cv
backtracking
look ahead
forward checking

More constraint propagation at each node will result in the search tree containing fewer nodes, but the overall cost may be higher, as the processing at each node will be more expensive. In one extreme, obtaining strong n-consistency for the original problem would completely eliminate the need for search, but as mentioned before, this is usually more expensive than simple backtracking. Actually, in some cases even the full look ahead may be more expensive than simple backtracking. That is the reason why forward checking and simple backtracking are still used in applications.

Search Orders and Search Reduction

Can we further influence efficiency of solving algorithms?

In the previous chapter we presented few search algorithms for constraint satisfaction. All of these algorithms require the order in which variables are to be considered for labelling as well as the order in which the values are assigned to the variable on backtracking. Note, that decisions about these orderings could affect the efficiency of the constraint satisfaction algorithm dramatically. For example, if a right value is chosen for each variable during labelling then the problem is solved completely without backtracking. Of course, this is an artificial case but in most cases we can choose ordering which can reduce the number of backtracks required in a search. In look-ahead algorithms, the ordering of variables could affect the amount of search space pruned.

Both topics of search orders and reduction of search space are discussed in this chapter

Variable Ordering[2]

Experiments and analysis of several researchers have shown that the ordering in which variables are chosen for instantiation can have substantial impact on the complexity of backtrack search. The ordering may be either

- a *static ordering*, in which the order of the variables is specified before the search begins, and it is not changed thereafter, or
- a *dynamic ordering*, in which the choice of next variable to be considered at any point depends on the current state of the search.

Dynamic ordering is not feasible for all search algorithms, e.g., with simple backtracking there is no extra information available during the search that could be used to make a different choice of ordering from the initial ordering. However, in look-ahead algorithms, the current state includes the domains of the variables as they have been pruned by the current set of instantiations, and so it is possible to base the choice of next variable on this information.

Several heuristics have been developed and analysed for selecting variable ordering. The most common one is based on the "**FAIL-FIRST"** principle, which can be explained as:

"*To succeed, try first where you are most likely to fail.*"

In this method, the variable with the fewest possible remaining alternatives, i.e., **the variable with the smallest domain**, is selected for instantiation. Thus the order of variable instantiations is, in general, different in different branches of the tree, and is determined dynamically. This method is based on assumption that any value is equally likely to participate in a solution, so that the more values there are, the more likely it is that one of them will be a successful one.

The fail-first principle may seem slightly misleading, after all, we do not want to fail. The reason is that if the current partial solution does not lead to a complete solution, then the sooner we discover it the better. Hence encouraging early failure, if failure is inevitable, is beneficial in the long term. On the other end, if the current partial solution can be extended to a complete solution, then every remaining variable must be instantiated and the one with the smallest domain is likely to be the most difficult to find a value for (instantiating other variables first may further reduce its domain and lead to a failure). Hence the principle could equally well be stated as:

[2] partly taken from Barbara M. Smith: A Tutorial on Constraint Programming, TR 95.14, University of Leeds,1995

"*Deal with hard cases first: they can only get more difficult if you put them off.*"

This heuristic should reduce the average depth of branches in the search tree by triggering early failure.

Another heuristic, that is applied when all variables have the same number of values (their domains are of the same size), is to choose the variable which participates in most constraints (in the absence of more specific information on which constraints are likely to be difficult to satisfy, for instance). This heuristic follows also the principle of dealing with hard cases first and it is called **most constrained heuristics**.

There is also a heuristic for static ordering of variables that is suitable for simple backtracking. This heuristic says: choose the variable which has the **largest number of constraints with the past variables**. For instance, during solving graph colouring problem, it is reasonable to assign colour to the vertex which has common arcs with already coloured vertices so the conflict is detected as soon as possible.

Backtrack-free Search

In the above paragraphs we presented variable orderings which can noticeably improve the efficiency of backtrack search. The open question is:

Does there exist any variable ordering which can eliminate the need for backtracking at all?

Before answering this question we definite what is backtrack-free search first.

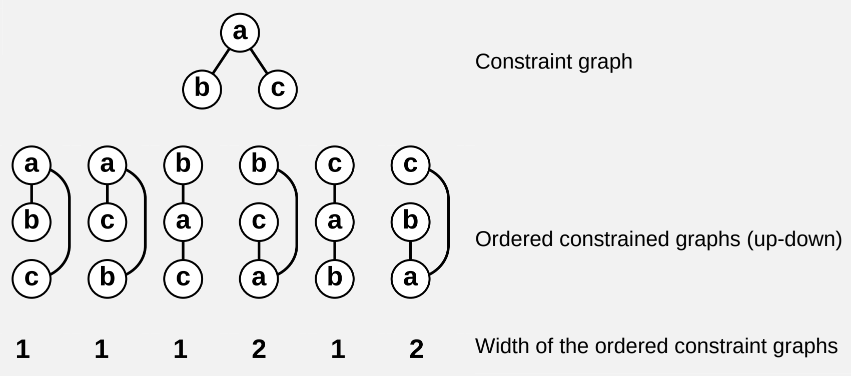
Definition: A search in CSP is **backtrack-free** if in a depth first search under an ordering of its variables for every variable that is to be labelled, one can always find for it a value which is compatible with all labels committed to so far.

If the ordering of variables is backtrack-free then we know that for each variable there exists a value compatible with the assignment of foregoing variables in the search and, therefore, no backtrack to change a value of foregoing variable is necessary. The following definitions and theorem show how to establish such backtrack-free ordering for strongly K-consistent constraint graphs.

Definition: An **ordered constraint graph** is a constraint graph whose vertices have been ordered linearly. The **width of the vertex** in an ordered constraint graph is the number of constraint arcs that lead from the vertex to the previous vertices (in the linear order). The **width of the ordered constraint graph** is the maximum width of any of its vertices and the **width of the constraint graph** is the minimum width of all the orderings of that graph. In general the width of a constraint graph depends upon its structure.

The following example demonstrates the meaning of above defined notions.

Example: (width of constraint graph)


a
b
c
Constraint graph
a b c
a c b
b a c
b c a
c a b
c b a
Ordered constrained graphs (up-down)
1 1 1 2 1 2
Width of the ordered constraint graphs

The width of the constraint graph is 1.

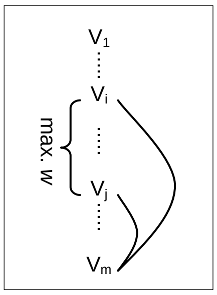
To find the width of the constraint graph we do not need to explore all possible variable orderings. The following procedure by Freuder finds a sequence of variables (vertices) which has the minimal width of the graph. In other words, the width of the ordered constraint graph defined by the returned ordering is the width of the constraint graph. The input to the procedure is a general graph and the output is a sequence of vertices which has the minimal width.

Algorithm Min Width Ordering:

```
procedure Min-Width-Ordering((V,E))
  Q <- {};
  while not V empty do
    N <- the node in V joined by the least number of edges in E;
         % in case of a tie, make arbitrary choice
    V <- V - {N};
    Remove all the edges from E which join V to other nodes in V;
    Q <- N:Q  % put N as a head of the sequence Q
  end while
  return Q;
end Min-Width-Ordering
```

Proposition: If a constraint graph is strongly K-consistent, and K>*w* where *w* is the width of the constraint graph, then there exists a search order that is backtrack free

Proof: The proof of the above proposition is straightforward. There exists an ordering of the graph such that the number of constraint arcs leading from any vertex of the graph to the previous vertices is at most *w*. Now if the variables are instantiated using this ordering, then whenever a new variable is instantiated, a value for this variable is consistent with all the previous assignments because:



(i) this value is to be consistent with the assignments of at most *w* other variables, and

(ii) the graph is strongly (*w*+1)-consistent.

Q.E.D.

Interestingly, all tree structured constrained graphs have width 1, so it is possible to find at least one ordering that can be used to solve the constraint graph without backtracking provided that the constraint graph is arc consistent.

Value Ordering[3]

Once the decision is made to instantiate a variable, it may have several values available. Again, the order in which these values are considered can have substantial impact on the time to find the first solution. However, if all solutions are required or there are no solutions, then the value ordering is indifferent.

A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than branches which lead to death ends. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

Suppose we have selected a variable to instantiate: how should we choose which value to try first? It may be that none of the values will succeed, in that case, every value for the current variable will eventually have to be considered, and the order does not matter. On the other hand, if we can find a complete solution based on the past instantiations, we want to choose a value which is likely to succeed, and unlikely to lead to a conflict. So, we apply the **"SUCCEED-FIRST"** principle.

One possible heuristic is to prefer those values that **maximise the number of options** available. Visibly, the algorithm AC-4 is good for using this heuristic as it counts the supporting values. It is possible to count "promise" of each value, that is the product of the domain sizes of the future variables after choosing this value (this is an upper bound on the number of possible solutions resulting from the assignment). The value with the highest promise should be chosen. It is also possible to calculate the percentage of values in future domains which will no longer be usable. The best choice would be the value with the lowest cost.

Another heuristic is to prefer the value (from those available) that leads to an easiest to solve CSP. This requires to estimate the difficulty of solving a CSP. One method by Dechter propose to convert a CSP

[3] partly taken from Barbara M. Smith: A Tutorial on Constraint Programming, TR 95.14, University of Leeds,1995

into a tree-structured CSP by deleting a minimum number of arcs and then to find all solutions of the resulting CSP (higher the solution count, easier the CSP).

For randomly generated problems, and probably in general, the work involved in assessing each value is not worth the benefit of choosing a value which will on average be more likely to lead to a solution than the default choice. In particular problems, on the other hand, there may be information available which allows the values to be ordered according to the principle of choosing first those most likely to succeed.

CC (Cycle-Cut set)

Above, we presented a theorem whose direct consequence is:

acyclic constraint graphs are globally consistent iff they are arc consistent.

Therefore, if we make the acyclic constraint graph arc-consistent then the graph is globally consistent as well so we can find the solution using backtrack-free search. Unfortunately, most constraint graphs contain cycles so it is not possible to use the sketched method directly. Nevertheless, after removing all cycles the method is applicable. Naturally, we do not need to remove all nodes (variables) on cycle, it is sufficient to cut the cycles by removing some nodes. A set of nodes which cut all the cycles in the graph is called a *cycle-cut set.*

And how to remove a variable from the constraint network? This is done by instantiating the variable and propagating this instantiation to neighbour variables via constraints. Then we can remove the variable from the network without influence to the rest of solution. Naturally, we may find later that this instantiation is in conflict with some other (cycle-cut set) variables so another instantiation should be tried. Finally, if we find a consistent instantiation of cycle-cut set then it is possible to extend this partial solution to a complete solution in a backtrack-free manner using the variable ordering described in the previous section. This observation makes the basis of the cycle-cut set algorithm whose formal definition follows.

Algorithm Cycle Cutset:

```
procedure cycle-cutset(G)
  C <- find-cycle-cutset(G);  % see next chapters
  while ex. labeling of variables in C satisfying all constraints do
    LC <- a(nother) labelling of variables in C satisfying all
          constraints;
    enforce DAC from C to the remaining variables;
    if all domains are non-empty then
      LR <- labelling of remaining variables (out of C)
            using backtrack-free search;
      return LC+LR;
    end if
  end while
  return fail;
end cycle-cutset
```

The major problem with the cycle-cut set method is its potential for thrashing. Because we are solving the cycle-cut set C independently of the rest of the network, the backtracking may occur to find another labelling for C caused by the same conflict.

MACE (MAC Extended)

To eliminate the potential sources of thrashing in the cycle-cut set algorithm, an extended version of MAC algorithm, MACE, was proposed using the same principle. MACE combines two basic ideas: instantiate less and propagate less to improve the efficiency of the constraint satisfaction algorithm. The algorithm instantiates only a subset of the CSP variables while maintaining only a partial arc consistent state of the constraint network. This partial arc-consistent state is guaranteed to extend to a fully arc-consistent state. The gain in efficiency is twofold. Instantiating a smaller number of variables aims at reducing the number of backtracks (and, accordingly, the number of constraint-checks, nodes visited, values deleted, etc.).

MACE uses the idea of cycle-cut set of a constraint graph while it removes its major drawback, thrashing, by maintaining arc consistency.

Algorithm MACE (idea):

```
1) enforce arc consistency (if it is not possible then return fail)
2) partition the variables into two sets:
      a cycle-cut set C and the set of variables which are not involved
      in any cycle U
      (note, that U is not necessarily the complement of C)
3) disconnect U from the graph
4) while C#0 do
    4a) set value to a variable in C
    4b) enforce arc consistency
        if failed then backtrack to 4a (or to previous variable)
    4c) disconnect singleton variables (add them to U)
    4d) disconnect "cycle-free" variables (add them to U)
5) reconnect variables in U
   enforce directed arc consistency from C to U
6) conduct backtrack-free search for a complete solution
```

How to find a cycle-cut set?

MACE and CC need an algorithm to find a cycle-cut set. There is no known polynomial algorithm for finding the minimum cycle-cut set (we prefer a smaller cycle-cut set because it has to labelled completely). Fortunately, there are several heuristics which yield a good cycle-cut set at a reasonable cost.

(i) The simplest heuristic sorts first the vertices in *decreasing order of their degree*. Then, starting with the vertex with the highest degree, as long as the graph still has cycles, add the vertex to the cycle-cut set and remove it, together with all the edges involving it, from the graph.

(ii) A smaller cycle-cut set can be obtained if, before adding a vertex to the cycle-cut set, we check whether the vertex is part of any cycle or not.

(iii) The third heuristic determines dynamically the number of cycles in which each vertex is involved and adds to the cycle-cut set at each step the vertex participating in the most cycles.

Visibly, second and third heuristic yields smaller cut sets than the first heuristic. Based on real-life observations, the third heuristic yields slightly smaller cycle-cut sets which translate sometimes into small gains in efficiency, but no major improvement upon the second heuristic. In case of large problems probably the second heuristic is the best choice, because of its lower cost.

Algorithm Find Cycle Cutset (using heuristic (i)):

```
procedure find-cycle-cutset(G)
  (V,E) = G;
  Q <- order elements in V by descending order of their degrees
       in the constraint graph G;
  CC <- {};
  while the graph G is cyclic do
    V <- first element in Q;
    CC <- CC + V;
    Q <- Q - V;
    remove V and edges involving it from the constraint graph G;
  end while
  return CC;
end find-cycle-cutset
```

Constraint Optimisation

How to find an optimal solution satisfying the constraints?

Till now we have presented the constraint satisfaction algorithms for finding one or all solutions satisfying all the constraints, i.e., all solutions were equally good. However, in many real-life applications, we do not want to find any solution but a good solution. The quality of solution is usually measured by some application dependent function called an *objective function*. The goal is to find such an assignment that satisfies all the constraints and minimise or maximise the objective function respectively. Such problems are called *Constraint Optimisation Problems (COP).*

Definition: A **Constraint Optimisation Problem (COP)** consists of a standard CSP and an objective function *f* which maps every solution (complete labelling of variables satisfying all the constraints) to a numerical value. The task is to find such a solution that is optimal regarding the objective function, i.e., it minimises or maximises respectively the objective function.

In order to find the optimal solution, we potentially need to explore all the solutions of CSP and to compare their values using the optimisation function. Therefore techniques for finding or generating all solutions are more relevant to COP than techniques for finding a single solution.

Branch and Bound

Perhaps the most widely used technique for solving optimisation problems including COP **is branch-and-bound** (B&B) which is a well known method both in Artificial Intelligence and Operations Research. This method uses heuristic to prune the search space. This heuristics, we will call it *h*, is a function that maps assignments (even partial) to a numeric value that is an estimate of the objective function. More precisely, *h* applied to some partial assignment is an estimate of best values of the objective function applied to all solutions (complete assignments) that rise by extending this partial assignment. Naturally, the efficiency of branch and bound method is highly dependent on availability of good heuristic. In such a case, the B&B algorithm can prune the search sub-trees where the optimal solution does not settle. Note, that there are two possibilities when the sub-tree can be pruned:

- there is no solution in the sub-tree at all,
- all solutions in the sub-tree are sub-optimal only (they are not optimal).

Of course, the closer the heuristic is to the objective function, the larger sub-tree can be pruned. On the other hand, we need a reliable heuristic ensuring that no sub-tree where the optimal solution settles is not pruned. This reliability can be achieved easily if, in case of minimisation problem, the heuristic is an underestimate of the optimisation function, i.e., the value of the heuristic function is not higher than the value of the objective function. In case of maximisation problems, we require the heuristic to be an overestimate of the objective function. In both cases, we can guarantee soundness and completeness of the branch-and-bound algorithm.

Unfortunately, it is not easy to find a reliable and efficient heuristic and, sometimes, such heuristic is not available. In such a case, it is up to the user if he or she chooses:

- a more efficient heuristic with the risk of pruning a sub-tree with an optimal solution (consequently, sub-optimal solution is obtained), or
- a reliable but less efficient heuristic with longer time of computation.

There exist several modifications of branch-and-bound algorithm, we will present here the depth-first branch-and-bound method that is derived from the backtracking algorithm for solving a CSP. The algorithm uses two global variables for storing the current upper bound (we are minimising the optimisation function) and the best solution found so far. It behaves like chronological backtracking algorithm except that as soon as a value is assigned to the variable, the value of heuristic function is computed. If this value exceeds the bound, then the sub-tree under the current partial assignment is pruned immediately. Initially, the bound is set to (plus) infinity and during computation it records the value of the objective function for the best solution found so far.

Algorithm Branch & Bound:

```
procedure BB(Variables, Constraints)
  Bound <- infinity;   % looking for minimum of function f
  Best <- nil;         % best solution found so far
  BB-1(Variables,{},Constraints)
  return Best
end BT

procedure BB-1(Unlabelled, Labelled, Constraints)
  if Unlabelled = {} then
    if f(Labelled) < Bound then
      Bound <- f(Labelled);   % set new upper bound
      Best <- Labelled;       % remember new best solution
    end if
  else
   pick first X from Unlabelled
   for each value V from D_X do
     if consistent({X/V}+Labelled, Constraints)
        & h({X/V}+Labelled) < Bound then
       BB-1(Unlabelled-{X}, {X/V}+Labelled, Constraints)
     end if
   end for
end BB-1
```

The efficiency of B&B is determined by two factors:

- the above discussed quality of the heuristic function and
- whether a good bound is found early.

Notice that we set (plus) infinity as the initial bound in the algorithm. However, if the user knows the value of optimum or its approximation (an upper bound in case of minimisation) then he or she can set the initial bound to a "better" value closer to optimum. Consequently, the algorithm can prune more sub-trees sooner and it is much more efficient.

Observations of real-life problems show also that the “last step” to optimum, i.e., improving a good solution even more, is usually the most computationally expensive part of the solving process. Fortunately, in many applications, users are satisfied with a solution that is close to optimum if this solution is found early. Branch-and-bound algorithm can be modified to find sub-optimal solutions as well by using the second “acceptability” bound that describes the upper bound of acceptable solution. The only modification of the above B&B algorithm to use acceptability bound is in the part where a complete assignment is found. If the algorithm finds a solution that is better than the acceptability bound then this solution is accepted, i.e., it can be returned immediately to the user even if it is not proved to be optimal.

Algorithm Branch & Bound with acceptability bound:

```
procedure BB-2(Unlabelled, Labelled, Constraints)
  if Unlabelled = {} then
    if f(Labelled) < Bound then
      Bound <- f(Labelled);   % set new upper bound
      Best <- Labelled;       % remember new best solution
      if f(Labelled) =< Acceptability_Bound then
        return Best
      end if
    end if
  else
   ...       % this part is the same as in BB-1 procedure
end BB-1
```

Glossary

What does this notion mean?

Arc consistency (AC) - a method of removing values from the domain that does not satisfy binary constraints on the variable

Arity - a number of parameters

Backjumping (BJ) - a backtracking based method to avoid thrashing using direct jump to conflicting variable

Backmarking (BM) - a method of reducing the number of consistency checks (during backtracking) by remembering incompatible labels

Backtracking (BT) - a search method using come-back upon failure

Binarization - a process of converting n-ary constraint to binary constrains

Binary CSP - a CSP with binary constraints only

Boolean CSP - a CSP with binary domains only

Constraint - a logical relation among several unknowns

Constraint Logic Programming (CLP) - an extension of logic programming to work with constraints over a given domain

Constraint Optimisation Problems (COP) - a variant of CSP when an optimal solution, given some objective function, is being found

Constraint Satisfaction Problem (CSP) - a problem consisting of the set of variables, their domains and constraints among variables that must hold

Directional Arc Consistency (DAC) - a method of removing values from the domain that does not satisfy binary constraints on the variable under an ordering of variables

Directional Path Consistency (DPC) - a method of removing value pairs from constraints that are not consistent along a path under an ordering of variables

Forward checking (FC) - a method of preventing future conflicts during search by removing inconsistent values from variables connected to currently labelled variable

Full Look Ahead (FLA) - see Look Ahead

Generate and test (GT) - a method of problem solving by exploring all possibilities and testing whether they satisfy the constraints

K-consistency - a method of removing inconsistent values by checking all constraints between K variables

Look ahead (LA) - a method of preventing future conflicts during search by maintaining arc consistency between unlabelled variables

Maintaining Arc Consistency (MAC) - see Look Ahead

Node consistency (NC) - a method of removing values from the domain that does not satisfy unary constraints on the variable

Partial Look Ahead (PLC) - a method of preventing future conflicts during search by maintaining directional arc consistency between unlabelled variables

Path consistency (PC) - a method of removing value pairs from constraints that are not consistent along a path

Restricted Path Consistency (RPC) - a method, half way between AC and PC, that checks path consistency only if one supporting value remains

SAT - a problem of satisfaction of logical formula

Strong K-consistency - a method of removing inconsistent values by checking all constraints between K or less variables

Thrashing - a repeated failure due to the same reason (during backtracking)

References

Where can I find more information?

On-line

On-line Guide to Constraint Programming
R. Barták, http://kti.mff.cuni.cz/~bartak/constraints, 1998.

Constraints Archive
http://4c.ucc.ie/web/archive/index.jsp

Association for Constraint Programming
http://slash.math.unipd.it/acp/

Books

Constraint Processing
R. Dechter, Morgan Kaufmann, 2003

Programming with Constraints: An Introduction
K. Marriott, P.J. Stuckey, MIT Press, 1998.

Constraint Satisfaction in Logic Programming
P. Van Hentenryck, MIT Press, 1989.

Foundations of Constraint Satisfaction
E. Tsang, Academic Press, 1993.

General Surveys

Constraint Logic Programming – A Survey
J. Jaffar & M.J. Maher, J. Logic Programming, 19/20:503-581, 1996.

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

A Tutorial on Constraint Programming
B.M. Smith, TR 95.14, University of Leeds, 1995.

The Origins

The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory
A. Borning, in ACM Transactions on Programming Languages and Systems 3(4): 252-387, 1981.

Logic Programming: Further Developments
H. Gallaire, in: IEEE Symposium on Logic Programming, Boston, IEEE, 1985.

Constraint Logic Programming
J. Jaffar & J.L. Lassez, in Proc. The ACM Symposium on Principles of Programming Languages, ACM, 1987.

Networks of constraints fundamental properties and applications to picture processing
U. Montanary, in: Information Sciences 7: 95-132, 1974.

Sketchpad: a man-machine graphical communication system
I. Sutherland, in Proc. IFIP Spring Joint Computer Conference, 1963.

Understanding line drawings of scenes with shadows
D.L. Waltz, in Psychology of Computer Vision, McGraw-Hill, New York, 1975.

Binarisation

On the conversion between Non-Binary and Binary Constraint Satisfaction Problems
F. Bacchus, P. van Beek, in Proc. National Conference on Artifical Intelligence (AAAI-98), Madison, Wisconsin, 1998.

On the equivalence of constraint satisfaction problems
F. Rossi, V. Dahr and C. Petrie, in Proc. European Conference on Artificial Intelligence (ECAI-90), Stockholm, 1990. Also MCC Technical Report ACT-AI-222-89.

Using auxiliary variables and implied constraints to model non-binary problems
B. Smith, K. Stergiou, T. Walsh, in Proc. National Conference on Artificial Intelligence (AAAI-00), Austin, Texas, 2000.

Encodings of Non-Binary Constraint Satisfaction Problems
K. Stergiou, T. Walsh, in Proc. National Conference on Artificial Intelligence (AAAI-99), Orlando, Florida, 1999.

Depth-First Search

Incomplete Depth-First Search Techniques: A Short Survey
Roman Barták. In Proceedings of CPDC 2004, Gliwice, Poland.

Discrepancy-Bounded Depth First Search
J. Christopher Beck and Laurent Perron. In Proceedings of CP-AI-OR, pp. 7-17, 2000.

Partial Search Strategy in CHIP
Nicolas Beldiceanu, Eric Bourreau, Peter Chan, and David Rivreau. In Proceedings of 2nd International Conference on Metaheuristics-MIC97, 1997.

ECLiPSe: An Introduction
Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen and Mark G. Wallace. Imperial College London, TR IC-Parc-03-1, 2003.

Backtracking algorithms for constraint satisfaction problems; a survey
R. Dechter, D. Frost, in Constraints, International Journal, 1998.

Performance Measurement and Analysis of Certain Search Algorithms
Gaschnig, J., CMU-CS-79-124, Carnegie-Mellon University, 1979.

Dynamic Backtracking
M.L. Ginsberg, in Journal of Artificial Intelligence Research 1, pages 25-46, 1993.

Iterative Broadening
Matthew L. Ginsberg and William D. Harvey. In Proceedings of National Conference on Artificial Intelligence (AAAI-90). AAAI Press, pp. 216-220, 1990.

Increasing tree search efficiency for constraint satisfaction problems
Haralick, R.M., Elliot, G.L., in: Artificial Intelligence 14:263-314, 1980.

Limited Discrepancy Search
W.D. Harvey and M.L. Ginsberg, in Proceedings of IJCAI95, pages 607-613, 1995.

Nonsystematic backtracking search
William D. Harvey. PhD thesis, Stanford University, 1995.

Limited discrepancy search
William D. Harvey and Matthew L. Ginsberg. In Proceedings of the 14th International Joint Conference on Artificial Intelligence, pp. 607-615, 1995.

Improved Limited Discrepancy Search
Richard E. Korf. In Proceedings of National Conference on Artificial Intelligence (AAAI-96). AAAI Press, pp. 286-291, 1996.

Interleaved Depth-First Search
Pedro Meseguer. In Proceedings of 15th International Joint Conference on Artificial Intelligence, pp. 1382-1387, 1997.

Depth-bounded Discrepancy Search
Toby Walsh. In Proceedings of 15th International Joint Conference on Artificial Intelligence, pp. 1388-1393, 1997.

Consistency techniques

Improving Domain Filtering using Restricted Path Consistency
P. Berlandier, in Proceedings of the IEEE CAIA-95, Los Angeles CA, 1995.

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

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

Refining the Basic Constraint Propagation Algorithm
Ch. Bessière and J.-Ch. Régin.. In Proceedings of IJCAI-01, 309-315, (2001).

Some practicable filtering techniques for the constraint satisfaction problem
R. Debruyne and C. Bessi`ere, in Proceedings of the 15th IJCAI, pages 412-417, 1997.

Neighborhood inverse consistency preprocessing
E. Freuder and C. D. Elfe, in Proceedings of the AAAI National Conference, pages 202-208, 1996.

Comments on Mohr and Henderson's path consistency algorithm
C. Han and C. Lee, in Artificial Intelligence 36, pages 125-130, 1988.

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

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

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

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

Singleton Consistencies
P. Prosser, K. Stergiou, T. Walsh, in Proc Principles and Practice of Constraint Programming (CP2000), pages 353-368, 2000.

A filtering algorithm for constraints of difference in CSPs
J.C. Régin, in AAAI-94, in Proceedings of the Twelfth National Conference on Artificial Intelligence, pages 362-367, 1994.

Path Consistency Revised
M. Singh, in Proceedings of the 7th IEEE International Conference on Tolls with Artificial Intelligence, pages 318-325, 1995.

A generic customizable framework for inverse local consistency
G. Verfaillie, D. Martinez, and C. Bessiere, in Proceedings of the AAAI National Conference, pages 169-174, 1999.

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

Making AC-3 an Optimal Algorithm
Y. Zhang and R. Yap.. In Proceedings of IJCAI-01, 316-321, (2001).