Module 4

Constraint satisfaction problems

4.1 Instructional Objective

- The students should understand the formulation of constraint satisfaction problems
- Given a problem description, the student should be able to formulate it in terms of a constraint satisfaction problem, in terms of constraint graphs.
- Students should be able to solve constraint satisfaction problems using various algorithms.
- The student should be familiar with the following algorithms, and should be able to code the algorithms
    - Backtracking
    - Forward checking
    - Constraint propagation
    - Arc consistency and path consistency
    - Variable and value ordering
    - Hill climbing

The student should be able to understand and analyze the properties of these algorithms in terms of

- time complexity
- space complexity
- termination
- optimality

- Be able to apply these search techniques to a given problem whose description is provided.
- Students should have knowledge about the relation between CSP and SAT

At the end of this lesson the student should be able to do the following:

- Formulate a problem description as a CSP
- Analyze a given problem and identify the most suitable search strategy for the problem.
- Given a problem, apply one of these strategies to find a solution for the problem.

Lesson 9

Constraint satisfaction problems - I

4.2 Constraint Satisfaction Problems

Constraint satisfaction problems or **CSP**s are mathematical problems where one must find states or objects that satisfy a number of *constraints* or criteria. A constraint is a restriction of the feasible solutions in an optimization problem.

Many problems can be stated as constraints satisfaction problems. *Here are some examples:*

Example 1: The n-Queen problem is the problem of putting n chess queens on an $n \times n$ chessboard such that none of them is able to capture any other using the standard chess queen's moves. The colour of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

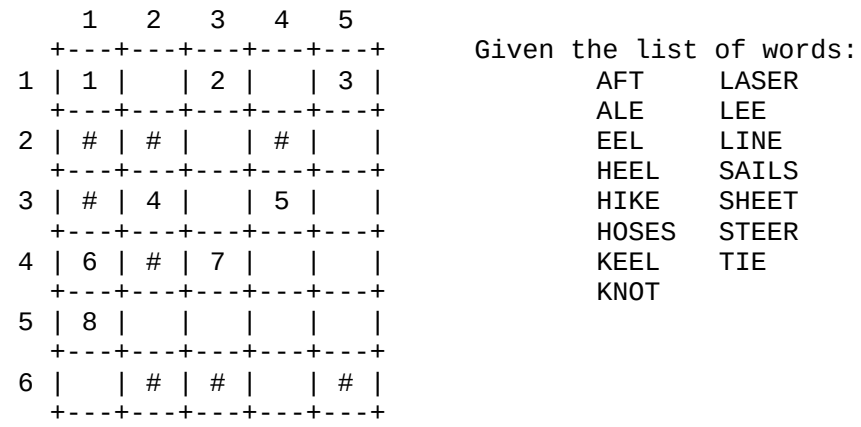
The problem was originally proposed in 1848 by the chess player Max Bazzel, and over the years, many mathematicians, including Gauss have worked on this puzzle. In 1874, S. Gunther proposed a method of finding solutions by using determinants, and J.W.L. Glaisher refined this approach.

The eight queens puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 **unique** solutions. The following table gives the number of solutions for *n* queens, both unique and distinct.

n:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	285,053
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	2,279,184

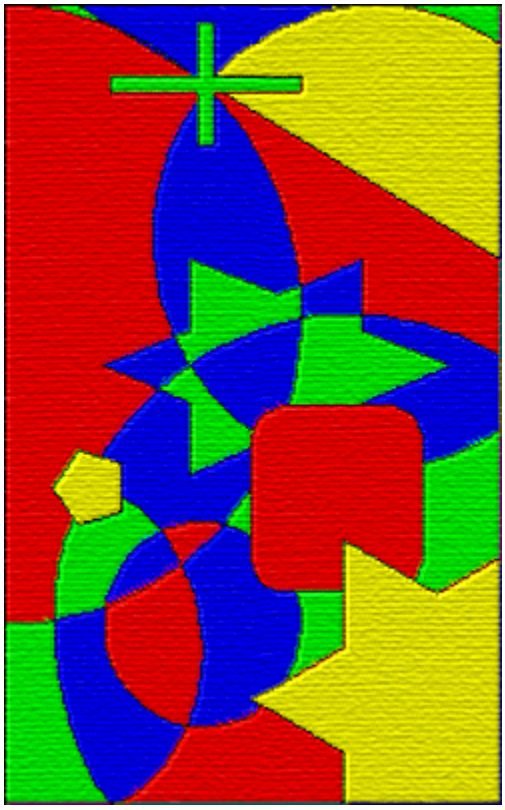
Note that the 6 queens puzzle has, interestingly, fewer solutions than the 5 queens puzzle!

Example 2: A crossword puzzle: We are to complete the puzzle



The numbers 1,2,3,4,5,6,7,8 in the crossword puzzle correspond to the words that will start at those locations.

Example 3: A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using k colors, so that no two neighboring countries have the same color. Example of a four color map is shown below:



The **four color theorem** states that given any plane separated into regions, such as a political map of the countries of a state, the regions may be colored using no more than four colors in such a way that no two adjacent regions receive the same color. Two regions are called *adjacent* if they share a border segment, not just a point. Each region must be contiguous: that is, it may not consist of separate sections like such real countries as Angola, Azerbaijan, and the United States.

It is obvious that three colors are inadequate: this applies already to the map with one region surrounded by three other regions (even though with an even number of surrounding countries three colors are enough) and it is not at all difficult to prove that five colors are sufficient to color a map.

The four color theorem was the first major theorem to be proved using a computer, and the proof is not accepted by all mathematicians because it would be infeasible for a human to verify by hand. Ultimately, one has to have faith in the correctness of the compiler and hardware executing the program used for the proof. The lack of

mathematical elegance was another factor, and to paraphrase comments of the time, "a good mathematical proof is like a poem — this is a telephone directory!"

Example 4: The **Boolean satisfiability problem (SAT)** is a decision problem considered in complexity theory. An instance of the problem is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true?

In mathematics, a formula of propositional logic is said to be **satisfiable** if truth-values can be assigned to its variables in a way that makes the formula true. The class of satisfiable propositional formulas is NP-complete. The propositional satisfiability problem (SAT), which decides whether or not a given propositional formula is satisfiable, is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design and verification.

The problem can be significantly restricted while still remaining NP-complete. By applying De Morgan's laws, we can assume that NOT operators are only applied directly to variables, not expressions; we refer to either a variable or its negation as a *literal*. For example, both x_1 and not(x_2) are literals, the first a *positive* literal and the second a *negative* literal. If we OR together a group of literals, we get a *clause*, such as (x_1 *or* not(x_2)). Finally, let us consider formulas that are a conjunction (AND) of clauses. We call this form conjunctive normal form. Determining whether a formula in this form is satisfiable is still NP-complete, even if each clause is limited to at most three literals. This last problem is called 3CNFSAT, 3SAT, or 3-satisfiability.

On the other hand, if we restrict each clause to at most two literals, the resulting problem, 2SAT, is in P. The same holds if every clause is a Horn clause; that is, it contains at most one positive literal.

Example 5: A cryptarithmetic problem: In the following pattern

```
  S E N D
  M O R E
=========
M O N E Y
```

we have to replace each letter by a distinct digit so that the resulting sum is correct.

All these examples and other real life problems like time table scheduling, transport scheduling, floor planning etc are instances of the same pattern, captured by the following definition:

A **Constraint Satisfaction Problem** (CSP) is characterized by:

- a *set of variables* {x1, x2, .., xn},
- for each variable xi a *domain* Di with the possible values for that variable, and

- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n, a value in Di for xi so that all constraints are satisfied.

A CSP can easily be stated as a sentence in first order logic, of the form:

```
(exist x1)..(exist xn) (D1(x1) & .. Dn(xn) => C1..Cm)
```

4.3 Representation of CSP

A CSP is usually represented as an undirected graph, called ***Constraint Graph*** where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyperarcs.

A constraint can affect any number of variables form 1 to n (n is the number of variables in the problem). If all the constraints of a CSP are binary, the variables and constraints can be represented in a constraint graph and the constraint satisfaction algorithm can exploit the graph search techniques.

The conversion of arbitrary CSP to an equivalent binary CSP is based on the idea of introducing a new variable that encapsulates the set of constrained variables. This newly introduced variable, we call it an encapsulated variable, has assigned a domain that is a Cartesian product of the domains of individual variables. Note, that if the domains of individual variables are finite than the Cartesian product of the domains, and thus the resulting domain, is still finite.

Now, arbitrary n-ary constraint can be converted to equivalent unary constraint that constrains the variable which appears as an encapsulation of the original individual variables. As we mentioned above, this unary constraint can be immediately satisfied by reducing the domain of encapsulated variable. Briefly speaking, n-ary constraint can be substituted by an encapsulated variable with the domain corresponding to the constraint.

This is interesting because any constraint of higher arity can be expressed in terms of binary constraints. Hence, binary CSPs are representative of all CSPs.

Example 2 revisited: We introduce a variable to represent each word in the puzzle. So we have the variables:

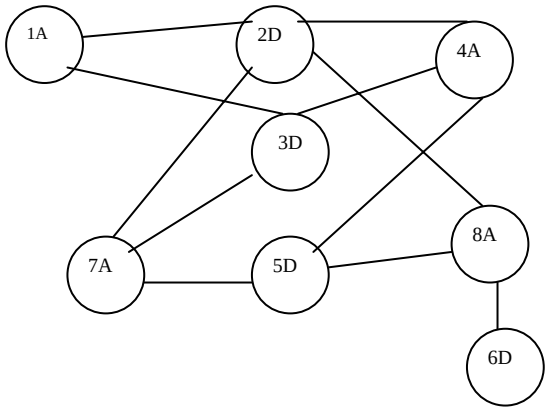
```
VARIABLE | STARTING CELL | DOMAIN
=============================================
1ACROSS  | 1             | {HOSES, LASER, SAILS, SHEET, STEER}
4ACROSS  | 4             | {HEEL, HIKE, KEEL, KNOT, LINE}
7ACROSS  | 7             | {AFT, ALE, EEL, LEE, TIE}
8ACROSS  | 8             | {HOSES, LASER, SAILS, SHEET, STEER}
2DOWN    | 2             | {HOSES, LASER, SAILS, SHEET, STEER}
3DOWN    | 3             | {HOSES, LASER, SAILS, SHEET, STEER}
5DOWN    | 5             | {HEEL, HIKE, KEEL, KNOT, LINE}
6DOWN    | 6             | {AFT, ALE, EEL, LEE, TIE}
```

The domain of each variable is the list of words that may be the value of that variable. So, variable 1ACROSS requires words with five letters, 2DOWN requires words with five letters, 3DOWN requires words with four letters, etc. Note that since each domain has 5 elements and there are 8 variables, the total number of states to consider in a naive approach is 5^8 = 390,625.

The constraints are all binary constraints:

```
        1ACROSS[3] = 2DOWN[1]  i.e. the third letter of 1ACROSS must be
equal to the first letter of 2DOWN
        1ACROSS[5] = 3DOWN[1]
        4ACROSS[2] = 2DOWN[3]
        4ACROSS[3] = 5DOWN[1]
        4ACROSS[4] = 3DOWN[3]
        7ACROSS[1] = 2DOWN[4]
        7ACROSS[2] = 5DOWN[2]
        7ACROSS[3] = 3DOWN[4]
        8ACROSS[1] = 6DOWN[2]
        8ACROSS[3] = 2DOWN[5]
        8ACROSS[4] = 5DOWN[3]
        8ACROSS[5] = 3DOWN[5]
```

The corresponding graph is:



4.4 Solving CSPs

Next we describe four popular solution methods for CSPs, namely, *Generate-and-Test*, *Backtracking*, *Consistency Driven, and Forward Checking*.

4.4.1 Generate and Test

We generate one by one all possible complete variable assignments and for each we test if it satisfies all constraints. The corresponding program structure is very simple, just nested loops, one per variable. In the innermost loop we test each constraint. In most situation this method is intolerably slow.

4.4.2 Backtracking

We order the variables in some fashion, trying to place first the variables that are more highly constrained or with smaller ranges. This order has a great impact on the efficiency of solution algorithms and is examined elsewhere. We start assigning values to variables. We check constraint satisfaction at the earliest possible time and extend an assignment if the constraints involving the currently bound variables are satisfied.

Example 2 Revisited: In our crossword puzzle we may order the variables as follows: 1ACROSS, 2DOWN, 3DOWN, 4ACROSS, 7ACROSS, 5DOWN, 8ACROSS, 6DOWN. Then we start the assignments:

```
   1ACROSS      <- HOSES
    2DOWN       <- HOSES     => failure, 1ACROSS[3] not equal to
2DOWN[1]
                <- LASER     => failure
                <- SAILS
     3DOWN      <- HOSES     => failure
                <- LASER     => failure
                <- SAILS
      4ACROSS   <- HEEL      => failure
                <- HIKE      => failure
                <- KEEL      => failure
                <- KNOT      => failure
                <- LINE      => failure, backtrack
     3DOWN      <- SHEET
      4ACROSS   <- HEEL
       7ACROSS  <- AFT       => failure
        ..............................
```

What we have shown is called *Chronological Backtracking*, whereby variables are unbound in the inverse order to the the order used when they were bound. *Dependency Directed Backtracking* instead recognizes the cause of failure and backtracks to one of the causes of failure and skips over the intermediate variables that did not cause the failure.

The following is an easy way to do dependency directed backtracking. We keep track at each variable of the variables that precede it in the backtracking order and to which it is connected directly in the constraint graph. Then, when instantiation fails at a variable, backtracking goes in order to these variables skipping over all other intermediate variables.

Notice then that we will backtrack at a variable up to as many times as there are preceding neighbors. [This number is called the *width* of the variable.] The time complexity of the backtracking algorithm grows when it has to backtrack often. Consequently there is a real gain when the variables are ordered so as to minimize their largest width.

4.4.3 Consistency Driven Techniques

Consistency techniques effectively rule out many inconsistent labeling at a very early stage, and thus cut short the search for consistent labeling. These techniques have since proved to be effective on a wide variety of hard search problems. The consistency techniques are deterministic, as opposed to the search which is non-deterministic. Thus the deterministic computation is performed as soon as possible and non-deterministic computation during search is used only when there is no more propagation to done. Nevertheless, the consistency techniques are rarely used alone to solve constraint satisfaction problem completely (but they could).

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

4.4.3.1 Node Consistency

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

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

4.4.3.2 Arc Consistency

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

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

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

The following algorithm does precisely that.

Algorithm REVISE

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

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

Algorithm AC-1

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

This algorithm is not very efficient because the succesfull revision of even one arc in some iteration forces all the arcs to be revised again in the next iteration, even though only a small number of them are really affected by this revision. Visibly, the only arcs affected by the reduction of the domain of V_k are the arcs (V_i,V_k). Also, if we revise the

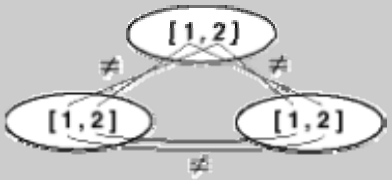
arc (V_k,V_m) and the domain of V_k is reduced, it is not necessary to re-revise the arc (V_m,V_k) because non of the elements deleted from the domain of V_k provided support for any value in the current domain of V_m. The following variation of arc consistency algorithm, called AC-3, removes this drawback of AC-1 and performs re-revision only for those arcs that are possibly affected by a previous revision.

Algorithm AC-3

```
procedure AC-3
  Q <- {(Vi,Vj) in arcs(G),i#j};
  while not Q empty
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      Q <- Q union {(Vi,Vk) such that (Vi,Vk) in
arcs(G),i#k,i#m}
    endif
  endwhile
end AC-3
```

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

Example:



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

Algorithm INITIALIZE

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

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

Algorithm AC-4

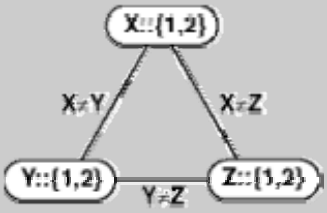
```
procedure AC-4
  Q <- INITIALIZE;
  while not Q empty
    select and delete any pair <j,b> from Q;
    for each <i,a> from Sj,b do
      counter[(i,j),a] <- counter[(i,j),a] - 1;
      if counter[(i,j),a]=0 & a is still in Di then
        delete a from Di;
        Q <- Q union {<i,a>};
      endif
    endfor
  endwhile
end AC-4
```

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

Maintaining arc consistency removes many inconsistencies from the constraint graph but is any (complete) instantiation of variables from current (reduced) domains a solution to the CSP? If the domain size of each variable becomes one, then the CSP has exactly one solution which is obtained by assigning to each variable the only possible value in its

domain. Otherwise, the answer is no in general. The following example shows such a case where the constraint graph is arc consistent, domains are not empty but there is still no solution satisfying all constraints.

Example:



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

4.4.3.3 Path Consistency (K-Consistency)

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

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

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

A node representing variable V_i is **restricted path consistent** if it is arc-consistent, i.e., all arcs from this node are arc-consistent, and the following is true: For every value a in the domain D_i of the variable V_i that has *just one supporting value* b from the domain of 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 algorithm for making graph restricted path consistent can be naturally based on AC-4 algorithm that counts the number of supporting values. Although this algorithm removes more inconsistent values than any arc-consistency algorithm it does not eliminate the need for 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 complexity of the algorithm for obtaining n-consistency in a n-node constraint graph

is also exponential. If the graph is (strongly) K-consistent for K<n, then in general, backtracking cannot be avoided, i.e., there still exist inconsistent values.

4.4.4 Forward Checking

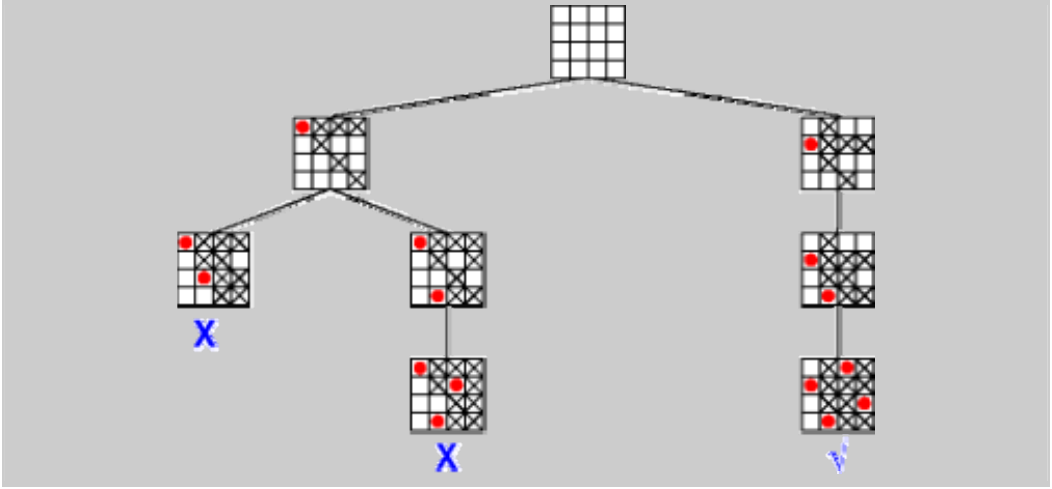
Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables. We speak about restricted arc consistency because forward checking checks only the constraints between the current variable and the future variables. 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. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

Algorithm AC-3 for Forward Checking

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

Forward checking detects the inconsistency earlier than simple backtracking and thus it allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. This reduces the search tree and (hopefully) the overall amount of work done. But it should be noted that forward checking does more work when each assignment is added to the current partial solution.

Example: (4-queens problem and FC)



Forward checking is almost always a much better choice than simple backtracking.

4.4.5 Look Ahead

Forward checking checks only the constraints between the current variable and the future variables. 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).

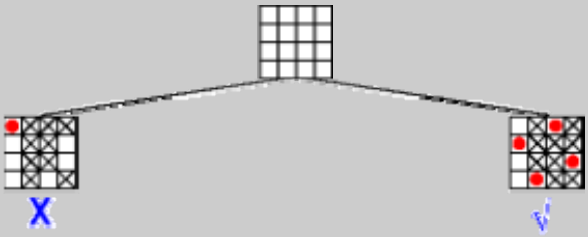
The advantage of look ahead is that it detects also the conflicts between future variables and therefore allows branches of the search tree that will lead to failure to be pruned earlier than with forward checking. Also as with forward checking, whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no necessary.

Algorithm AC-3 for Look Ahead

```
procedure AC3-LA(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent
    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,i>cv}
      consistent <- not Dk empty
    endif
  endwhile
  return consistent
end AC3-LA
```

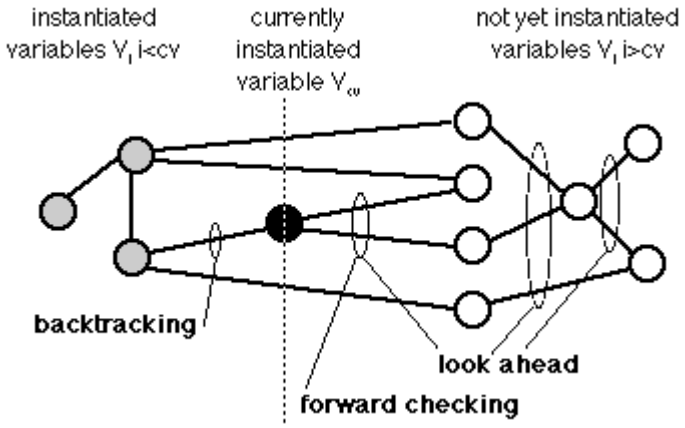
Look ahead prunes the search tree further more than forward checking but, again, it should be noted that look ahead does even more work when each assignment is added to the current partial solution than forward checking.

Example: (4-queens problem and LA)



4.4.6 Comparison of Propagation Techniques

The following figure shows which constraints are tested when the above described propagation techniques are applied.



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.