# Constraint-Satisfaction Problems

Constraint-Satisfaction Problems use a factored representation for each state, that is each state consists of a set of variables which can be assigned with some values. When the set of variables are assigned with values which do not violate any constraint then the CONSTRAINT-SATISFACTION problem has been resolved.

Constraint-Satisfaction Search algorithms use general purpose techniques rather than problem-specific techniques.

A screenshot of a cell phone

Description automatically generatedConstrains satisfaction problems can be represented as constraint graphs where the nodes represent the variables of the problems and the edges represent the constraint between variables (nodes).

A close up of a map

Description automatically generatedIf a problem can be represented as a constraint satisfaction problem then it is better to use a constraint-satisfaction search algorithm rather then a simple state-search algorithm. Indeed, while state-search algorithms are capable of understanding whether a given node is a terminal node or not, constraint-satisfaction problems are able to cut large portions of the search space by exploiting the constraints. For example, making use of the constraints of the problem, constraint-satisfaction problems can understand whether the current partial assignment may lead to the solution of the problem or not.

For example, once we have chosen {SA = blue} in the Australia problem, we can conclude that none of the five neighbouring variables can take on the value blue. Without taking advantage of constraint propagation, a search procedure would have to consider 35 = 243 assignments for the five neighbouring variables; with constraint propagation we never have to consider blue as a value, so we have only 25 = 32 assignments to look at, a reduction of 87%.

## Types of Constraints

1. Unary Constraint => restricts the value of a single variable
2. Binary Constraint => relates two variables
3. Ternary Constraint => related three variables
4. N-Ary Constraint => relates n constraints
5. Global Constraint => a constraint that relates an arbitrary number of variables. An example is Alldiff , which says that all of the variables involved in the constraint must have different values

Any Constraint Satisfaction Problem involving N-Ary Constraints can be reduced to a Binary CSP and so can be represented as a CONSTRAINT GRAPH.

A close up of a map

Description automatically generated

In alternative, in order to reduce the number of binary constraints it is possible to relate relation variables with each other when they have sharded variables (this way the number of binary relations halves). This is called DUAL GRAPH TRANSFORMATION.

Let's say your problem has the following constraints:

* C1, which involves x, y and z:
  + x + y = z
* C2, which involves x and y:
  + x < y

with the following domains:

* x :: [1,2,3]
* y :: [1,2,3]
* z :: [1,2,3]

The author says that you need to create 2 more variables, one for each constraint. They are defined as follows:

* c1 = < x, y, z >
* c2 = < x, y >

The domains of c1 and c2 are defined so that they don't violate C1 and C2, i.e.:

* c1 :: [ <1,2,3>, <2,1,3>, <1,1,2>]
* c2 :: [<1,2>, <2,3>, <1,3>]

c1 and c2 will be the nodes of the dual graph, but first you need to define a constraint between them, i.e. R1:

* R1: "the 1st and the 2nd element of c1 (x and y) must be equal to the 1st and the 2nd element of c2 respectively" (actually you could split it in two simpler constraints)

Furthermore, we can divide the constraints into:

1. Absolute Constraints => their violation leads to a NON SOLUTION. They cannot be violated
2. Preference Constraints => their violation does not lead to a NON SOLUTION. They can be violated

Many real-world CSPs include preference constraints PREFERENCE CONSTRAINTS indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one. Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a constraint optimization problem, or COP. CONSTRAINT OPTIMIZATION PROBLEM Linear programming problems do this kind of optimization

## Inference in CSP through CONSTRAINT PROPAGATION

While search algorithms can only SEARCH, CSP algorithms can do two things:

1. Search => assigning a value to a variable
2. CONSTRAINT PROPAGATION => exploiting the constraints to reduce the number of values that variables can be assigned with. In other words, reducing the domain of variables

Constraint Propagation can be achieved through different techniques that belong to the category of LOCAL CONSISTENCIES.

These LOCAL CONSISTENCY techniques are the following:

1. Node Consistency
2. Arc Consistency
3. Path Consistency
4. K-Consistency

## Node Consistency

A single variable (corresponding to a node in the CSP network) is node-consistent if all the values in the variable’s domain satisfy the variable’s unary constraints.

For example, in the variant of the Australia map-colouring problem (Figure 6.1) where South Australians dislike green, the variable SA starts with domain {red, green, blue}, and we can make it node consistent by eliminating green, leaving SA with the reduced domain {red, blue}. We say that a network is node-consistent if every variable in the network is node-consistent.

## Arc Consistency

A variable in a CSP is arc-consistent if every value in its domain satisfies the variable’s binary constraints.

More formally, Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi, Xj ).

A network is arc-consistent if every variable is arc consistent with every other variable.

Remember that it is not always possible to reduce the domain of a variable by using arc-consistency.

Remember that while Xi can be arc-consistent with Xj, Xj may not be arc-consistent with Xi.

A screenshot of a social media post

Description automatically generated

The analysis of the time complexity of the AC-3 algorithm is the following.

Assume a CSP with n variables, each with domain size at most d, and with c binary constraints (arcs). Each arc (Xk, Xi) can be inserted in the queue only d times because Xi has at most d values to delete. Therefore, in the worst case the queue will be filled with O(dc) arcs. The REVISE function has a time complexity of O(d2) because in the worst case both Di and Dj have d values and each outer loop (d of them) will run d times. In conclusion, the time complexity of AC-3 is O(dc) x O(d2) = O(d3c).

A variable Xi is **generalized arc consistent** with respect to an n-ary constraint if for every value v in the domain of Xi there exists a tuple of values that is a member of the constraint which has its Xi component equal to v and all other component values must be values that belong to the domain of the corresponding variable. If this does not occur for a value v of Xi then the value v can be deleted from the domain of Xi.

## Path Consistency

While Arc-Consistency can be a very useful tools in some kinds of CSPs like establishing if a CSP cannot be solved (by reducing some domain to size 0) or by resolving it (by reducing every domain to size 1), in some other problems it may not be enough to reduce the domain of variables.

Consider the map-colouring problem on Australia, but with only two colours allowed, red and blue. Arc consistency can do nothing because every variable is already arc consistent: each can be red with blue at the other end of the arc (or vice versa). But clearly there is no solution to the problem: because Western Australia, Northern Territory and South Australia all touch each other, we need at least three colours for them alone.

This is why we need a higher local consistency called PATH CONSISTENCY.

Path-Consistency consists of creating all possible 2-variables sets where the variables inside each set are linked by a binary constraint. Then it needs to be established whether a 2-variables set if PATH CONSISTENT with another variable which must be linked to both of the variables in the 2-variables set by a binary constraint.

A two-variable set {Xi, Xj} is path-consistent with respect to a third variable Xm if, for every assignment {Xi = a, Xj = b} consistent with the constraints on {Xi, Xj}, there is an assignment to Xm that satisfies the constraints on {Xi, Xm} and {Xm, Xj}. This is called path consistency because one can think of it as looking at a path from Xi to Xj with Xm in the middle.

Let’s see how path consistency fares in colouring the Australia map with two colours. We will make the set {WA, SA} path consistent with respect to NT. We start by enumerating the consistent assignments to the set. In this case, there are only two: {WA = red, SA = blue} and {WA = blue, SA = red}. We can see that with both of these assignments NT can be neither red nor blue (because it would conflict with either WA or SA). Because there is no valid choice for NT, we eliminate both assignments, and we end up with no valid assignments for {WA, SA}. Therefore, we know that there can be no solution to this problem.

## K-Consistency

K-Consistency techniques are higher forms of local consistencies.

Node consistency corresponds to 1-Consistency, Arc-Consistency corresponds to 2-Consistency and Path Consistency corresponds to 3-Consistency.

A CSP is k-consistent if, for any set of k − 1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable.

If a set of k-1 variables is k consistent with a variable Kth and there are some variables among the k-1 variables that are not linked to the Kth variable then they do not influence the consistency as they do not constrain in any way the values that Kth can have.

In other words, If a set of k-1 variables is k consistent with a variable Kth and n variables are added to the set which are not linked to the Kth variable by a binary constraint then the set becomes (k+n) consistent with the Kth variable.

A CSP is **STRONGLY K-CONSISTENT** if it is k-consistent and is also (k − 1)-consistent, (k − 2)-consistent, ... all the way down to 1-consistent.

## Global Constraints

As we have already said Alldiff is a GLOBAL CONSTRAINT.

One simple form of inconsistency detection for Alldiff constraints works as follows: if m variables are involved in the constraint, and if they have n possible distinct values altogether, and m>n, then the constraint cannot be satisfied.

This leads to the following simple algorithm: First, remove any variable in the constraint that has a singleton domain, and delete that variable’s value from the domains of the remaining variables. Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

This method can detect the inconsistency in the assignment {WA = red, NSW = red} for Figure 6.1. Notice that the variables SA, NT, and Q are effectively connected by an Alldiff constraint because each pair must have two different colors. After applying AC-3 with the partial assignment, the domain of each variable is reduced to {green, blue}. That is, we have three variables and only two colors, so the Alldiff constraint is violated.

Another important higher-order constraint is the resource constraint, sometimes called the atmost constraint.

For example, in a scheduling problem, let P1,...,P4 denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as Atmost(10, P1, P2, P3, P4).

We can detect an inconsistency simply by checking the sum of the minimum values of the current domains; for example, if each variable has the domain {3, 4, 5, 6}, the Atmost constraint cannot be satisfied.

We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain {2, 3, 4, 5, 6}, the values 5 and 6 can be deleted from each domain.

For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency-checking methods.

Instead, domains are represented by upper and lower bounds and are managed by **BOUNDS PROPAGATION**.

For example, in an airline-scheduling problem, let’s suppose there are two flights, F1 and F2, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then

D1 = [0, 165] and D2 = [0, 385] .

Now suppose we have the additional constraint that the two flights together must carry 420 people: F1 + F2 = 420.

Propagating bounds constraints, we reduce the domains to D1 = [35, 165] and D2 = [255, 385] .

A CSP is said to be BOUND CONSISTENT if for every value that is in the range of the bound of a variable X satisfy the constraints with another variable Y, that is we can choose a value in the range of bound Y that together with the value chosen for X satisfy the constraint.

## Backtracking Search for CSPs

Using Local consistency techniques is not enough to resolve most of the real-world CSPs. Thus, it is not possible to resolve a CSP by means of only inferences by constraints but it is needed to do some kind of SEARCH.

In CSPs, SEARCH means to add a value to a variable that is not currently assigned to the CURRENT PARTIAL ASSIGNMENT. Therefore, SEARCH in CSPs consists of starting with an EMPTY ASSIGNMENT and arriving to the SOLUTION through several PARTIAL ASSIGNMENTS that step by step increment their size by 1 because a new variable is assigned and added to the current assignment.

A CRUCIAL PROPERTY of CSPs is **COMMUTATIVITY.** A problem is commutative when the order of actions does not matter. Therefore, a CSPs are commutative because the order through which the variables are assigned does not matter.

Given that a CSP consists of n variables with a domain of side d each then the total number of partial assignments is equal to dn. If CSPs were not commutative then CSP search algorithms should try every possible sequence of variable assignments each of them could be assigned with d values. Such algorithms would produce nd nodes at the first level, (n-1)d nodes at the next level and so on, producing n!dn leaves which are way more than the possible COMPLETE ASSIGNMENTS of the problem itself.

Therefore, CSP-Search Algorithms must deal with only one variable for each level of the SEARCH TREE.

A picture containing screenshot

Description automatically generated

The term backtracking search is used for a depth-first search that chooses values for BACKTRACKING SEARCH one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

A screenshot of a cell phone

Description automatically generated

It is possible to make BACKTRACKING-SEARCH more efficient for a specific CSP by tuning the functions SELECT-UNASSIGNED VARIABLE, ORDER-DOMAIN VALUES and INFERENCE.

Indeed, we can add sophistication to the backtracking search by answering these questions:

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
2. What inferences should be performed at each step in the search (INFERENCE)?
3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

## Variable and Value ordering

It is possible to choose which variable to assign next according to several heuristics.

The most common ones are the following:

1. Static Ordering => choose the next unassigned variable in order {X1, X2,...}
2. Random Ordering => choose the next unassigned variable randomly
3. Minimum-Remaining Values (MRV)
4. Degree Heuristic

The **MINIMUM-REMAINING VALUE** heuristic consists of choosing the unassigned variable with the least number of values in its domain as the next variable to be assigned. This heuristic is based on the very simple idea that given that the current partial assignment cannot lead to a solution then by choosing the unassigned variable with the least number of values in its domain as the next variable to be assigned will be the fastest way to understand that the current assignment is wrong.

In other words, this heuristic is the most effective way to reduce the SEARCH SPACE because at each level chooses the variable that will contribute the least to the BRANCHING FACTOR.

The MRV heuristic usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

The **DEGREE-HEURISTIC** consists of selecting the variable that is involved in the largest number of constraints on other unassigned variables. It is based on the simple idea that by choosing the unassigned variable that is involved in the largest number of constraints with the other unassigned variables then the latter’s domains will be reduced in size because their domains will be constrained by the chosen variable. In other words, the branching factor will be reduced at each level.

Empirical data has shown the MRV heuristic performs better than the DEGREE-HEURISTIC in general.

It is possible to use the MRV heuristic and the Degree-heuristic in combination by using the Degree-Heuristic as a tie-braker (if there are more than one variables with same domain size then choose the variable which constraints the larger number of other unassigned variables).

As regards value ordering, one of the most effective heuristics is the LEAST-CONSTRAINING VALUE heuristic which consists of assigning a value to the variable which constraints as little as possible the neighbouring variables (nodes).

The explanation of the LEAST-CONSTRAINING VALUE heuristic is that while with the MRV or Degree heuristic we reduce the size of the SEARCH-SPACE with the Least-Constraining value heuristic we try to follow the path that seems more likely to bring us to the solution of the problem as our final aim is to find the SOLUTION as soon as possible.

## Interleaving Search with Inference

The most efficient way to use the local consistency techniques mentioned before is to interleave them with search.

One of the simplest ways is to use FORWARD CHECKING.

FORWARD CHECKING consists of running arc-consistency on the variables that are neighbouring with the variable that has been assigned by the search phase. By doing this way, it is possible to reduce the domain size of the neighbouring variables.

As it is possible to understand, FORWARD CHECKING is a very important tool to apply the MRV heuristic as through FORWARD CHECKING we can reduce step by step the domain of the variables.

Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after WA = red and Q = green are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. A second point to notice is that after V = blue, the domain of SA is empty. Hence, forward checking has detected that the partial assignment {WA = red, Q = green, V = blue} is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

A picture containing photo, wooden, many, old

Description automatically generated

Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it makes the current variable arc-consistent, but doesn’t look ahead and make all the other variables arc-consistent. For example, consider the third row of Figure 6.7. It shows that when WA is red and Q is green, both NT and SA are forced to be blue. Forward checking does not look far enough ahead to notice that this is an inconsistency: NT and SA are adjacent and so cannot have the same value. The algorithm called MAC (for Maintaining Arc Consistency (MAC)) detects this MAINTAINING ARC CONSISTENCY (MAC) inconsistency. After a variable Xi is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (Xj , Xi) for all Xj that are unassigned variables that are neighbors of Xi. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC’s queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

In other words, while FORWARD CHECKING makes all neighbouring nodes of the assigned node arc-consistent with the latter, the MAC goes beyond because in case a neighbouring node’s domain x has decreased in size then arc-consistency must be established between x and all his neighbouring nodes as well.

A screenshot of a cell phone

Description automatically generated

A screenshot of a social media post

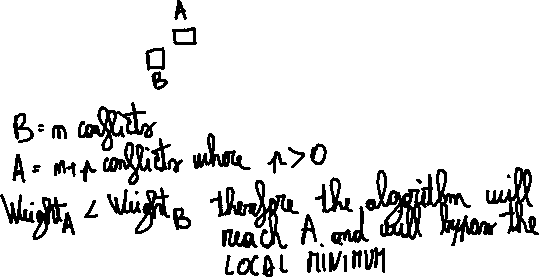
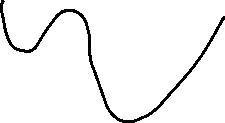
Description automatically generated

## A close up of a newspaper Description automatically generated

After having analysed constraint weighting I have come up with the following observations:

1. The algorithm will try to remove the constraints which result more difficult not to violate
2. When a local minimum is reached, it is possible to overcome it even by using the min-conflict heuristic because it may happen that the states that have an higher number of conflicts are matched with a lower weight. In other words, even if the local minimum has a lower number of conflicts, it may be matched with a high weight because it contains constraints that have been hard to get rid of.
3. The algorithm may overcome plateaux as well if side-moves are allowed

However, the second and third options above are not guaranteed to happen.



## Links:

* Why a CSP with n variables and n-strong consistent can be resolved in time complexity O(n2d) where d is the max number of the values contained in the domains
* Understanding BACKJUMPING
* Do chapter 6.5