1. Introduction (1.-Introduction)

Constraint Satisfaction Problem

1. Introduction

In a **Constraint Satisfaction Problem (CSP)** every state of a problem is described by a set of variables each of which has a value. A problem is said to be solved when each variable has a value that satisfies all the constraints on the variable.

CSP search algorithms take advantage of the structure of states and use general rather than domain-specific heuristics to enable the solution of complex problems.

The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

CSPs have the additional advantage that the actions and transition model can be deduced from the problem description.

A constraint satisfaction problem consists of three components, $\mathcal{X}$, $\mathcal{D}$, and $\mathcal{C}$:
$\mathcal{X}$ is a set of variables, $\{X_1, \ldots, X_n\}$.
$\mathcal{D}$ is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.

A domain, D_i, consists of a set of allowable values, $\{v_1, v_2, \cdots, v_k\}$, for variable X_i.

- For example, a Boolean variable would have the domain $\{true, false\}$. Different variables can have different domains of different sizes.

Each constraint C_j consists of a pair $\langle scope, rel \rangle$, where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.

- For example, if X1 and X2 both have the domain $\{1, 2, 3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as $\langle (X1, X2), \{(3, 1), (3, 2), (2, 1)\} \rangle$ or as $\langle (X_1, X_2), X_1 > X_2 \rangle$.

A constraint involving an arbitrary number of variables is called a **global constraint**. Unary Constraint has one variable and Binary Constraint has two variables.

CSPs deal with assignments of values to variables, $\{X_i = v_i, X_j = v_j, \cdots\}$. An assignment that does not violate any constraints is called a **consistent or legal assignment**.

A **complete assignment** is one in which every variable is assigned a value, and a solution to a CSP is a consistent, complete assignment.

A **partial assignment** is one that leaves some variables unassigned, and a partial solution is a partial assignment that is consistent.

Solving a CSP is an NP-complete problem in general, although there are important subclasses of CSPs that can be solved very efficiently.

Example problem: Map coloring

Given a map of Australia showing each of its states and territories the task is to assign color; either red, green, or blue; to each region in such a way that no two neighboring regions have the same color.

To formulate this as a CSP, we define the variables to be the regions:
$x = \{WA, NT, Q, NSW, V, SA, T\}$.

The domain of every variable is the set $\{D_i = red, green, blue\}$.

The constraints require neighboring regions to have distinct colors.

Since there are nine places where regions border, there are nine constraints:

$$\begin{aligned} C = \{&SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ &WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}. \end{aligned}$$

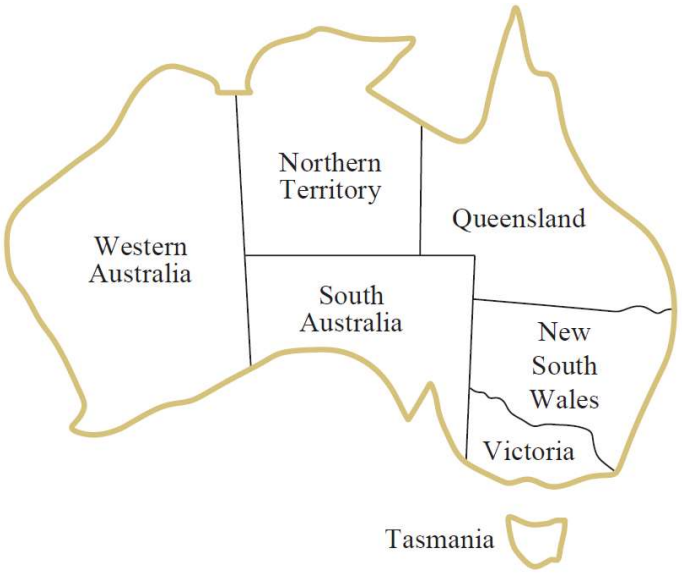
Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

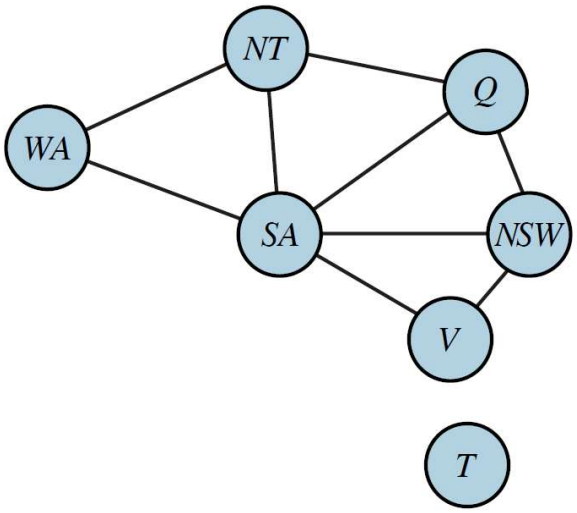
$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}.$$

There are many possible solutions to this problem, such as

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}.$$

We can visualize a binary CSP, with only unary and binary constraints, as a constraint graph as shown in below.





The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.

Why formulate a problem as a CSP?

One reason is that the CSPs yield a natural representation for a wide variety of problems; it is often easy to formulate a problem as a CSP.

Another is that years of development work have gone into making CSP solvers fast and efficient.

A third is that a CSP solver can quickly prune large swathes of the search space that an atomic state-space searcher cannot.

- For example, once we have chosen $SA = blue$ in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue.

A search procedure that does not use constraints would have to consider $3^5 = 243$ assignments for the five neighboring variables; with constraints we have only $2^5 = 32$ assignments to consider, a reduction of 87%.

In atomic state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment violates a constraint, we can immediately discard further refinements of the partial assignment.

Furthermore, we can see why the assignment is not a solution—we see which variables violate a constraint— so we can focus attention on the variables that matter. As a result, many problems that are intractable for atomic state-space search can be solved quickly when formulated as a CSP.

Example problem: Job-shop scheduling

Consider a small part of the car assembly, consisting of 15 tasks:

- install axles (front and back),
- affix all four wheels (right and left, front and back),
- tighten nuts for each wheel,

- affix hubcaps, and
- inspect the final assembly.

We can represent the tasks with 15 variables:

$$\begin{aligned}\mathcal{X} = \{&Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF},\\ &Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}.\end{aligned}$$

In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write

$$\begin{aligned}Axle_F + 10 \leq Wheel_{RF}; \quad Axle_F + 10 \leq Wheel_{LF};\\ Axle_B + 10 \leq Wheel_{RB}; \quad Axle_B + 10 \leq Wheel_{LB}.\end{aligned}$$

Next we say that for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

$$\begin{aligned}Wheel_{RF} + 1 \leq Nuts_{RF}; \quad Nuts_{RF} + 2 \leq Cap_{RF};\\ Wheel_{LF} + 1 \leq Nuts_{LF}; \quad Nuts_{LF} + 2 \leq Cap_{LF};\\ Wheel_{RB} + 1 \leq Nuts_{RB}; \quad Nuts_{RB} + 2 \leq Cap_{RB};\\ Wheel_{LB} + 1 \leq Nuts_{LB}; \quad Nuts_{LB} + 2 \leq Cap_{LB}.\end{aligned}$$

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a disjunctive constraint to say that AxleF and AxleB must not overlap in time; either one comes first or the other does:

$$(Axle_F + 10 \leq Axle_B) \quad \textbf{or} \quad (Axle_B + 10 \leq Axle_F).$$

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except Inspect we add a constraint of the form

$$X + d_X \leq \text{Inspect}$$

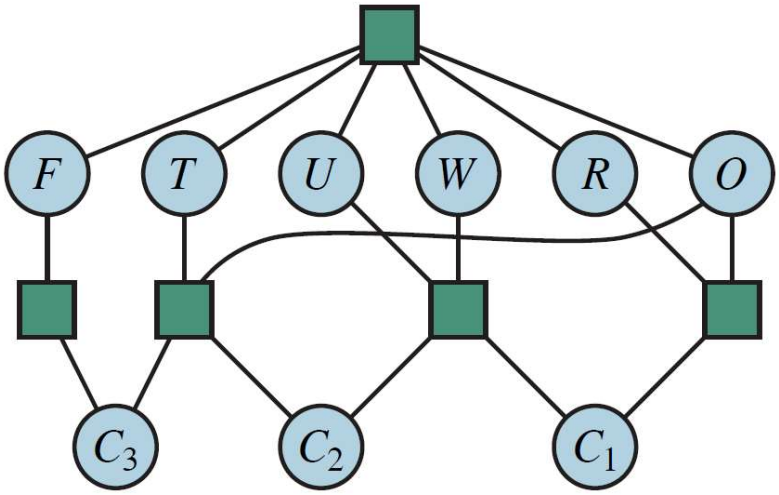
Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

$$D_i = \{0, 1, 2, 3, \cdots 30\}$$

Cryptarithmetic puzzle

Each letter in a cryptarithmetic puzzle represents a different digit, which can be represented as the global constraint Alldiff(F,T,U,W,R,O).

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

$$\begin{aligned} O + O &= R + 10 \cdot C_1 \\ C_1 + W + W &= U + 10 \cdot C_2 \\ C_2 + T + T &= O + 10 \cdot C_3 \\ C_3 &= F \,, \end{aligned}$$

where C_1, C_2, and C_3 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column.

These constraints can be represented in a constraint hypergraph, such as the one shown in above figure towards right.

A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints — constraints involving n variables.

The constraints we have described so far have all been **absolute constraints**, violation of which rules out a potential solution.

Many real-world CSPs include **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 but would not be an optimal one.

CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constrained optimization problem, or COP**.

Constraint Propagation:

The process of using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

Local Consistency.

If we treat each variable as a node in a graph and each binary constraint as an edge, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency

- **Node consistency**: A single variable (corresponding to a node in the CSP graph) 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-coloring problem 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 graph is node-consistent if every variable in the graph is node-consistent***.

    - It is easy to eliminate all the unary constraints in a CSP by reducing the domain of variables with unary constraints at the start of the solving process.

- **Arc consistency**: A variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints. More formally, X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). A graph is arc-consistent if every variable is arcconsistent with every other variable.
    - For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of decimal digits. We can write this constraint explicitly as $\langle (X, Y), (0, 0), (1, 1), (2, 4), (3, 9) \rangle$. To make X arc-consistent with respect to Y, we reduce X's domain to {0, 1, 2, 3}. If we also make Y arc-consistent with respect to X, then Y's domain becomes {0, 1, 4, 9}.
    - The most popular algorithm for enforcing arc consistency is called AC-3.

AC-3 algorithm

To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. Initially, the queue contains all the arcs in the CSP. (Each binary constraint becomes two arcs, one in each direction.) AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue and makes X_i arc-consistent with respect to X_j.

If this leaves D_i unchanged, the algorithm just moves on to the next arc. But if this revises D_i (makes the domain smaller), then we add to the queue all arcs (X_k, X_i) where X_k is a neighbor of X_i. We need to do that because the change in D_i might enable further reductions in D_k, even if we have previously considered X_k.

If D_i is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure.

Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will be faster to search because its variables have smaller domains.

In some cases, it solves the problem completely (by reducing every domain to size 1) and in others it proves that no solution exists (by reducing some domain to size 0).

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

Path-Consistency

A two-variable set $\{X_i, X_j\}$ is path-consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints (if any) on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. The name refers to the overall consistency of the path from X_i to X_j with X_m in the middle.

Let's see how path consistency fares in coloring the Australia map with two colors.

- 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

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 k^{th} variable.

- 1-consistency says that, given the empty set, we can make any set of one variable consistent: this is what we called node consistency.
- 2-consistency is the same as arc consistency.
- For binary constraint graphs, 3-consistency is the same as path consistency.

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

Remember that a global constraint is one involving an arbitrary number of variables (but not necessarily all variables).

- For example, the Alldiff constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem above and Sudoku puzzles).

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 coloring problem of Australia.

- 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 domains of SA, NT, and Q are all reduced to {green, blue}.
- That is, we have three variables and only two colors, so the Alldiff constraint is violated.

Thus, a simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constraints.

Resource constraint / 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.

Bounds propagation.

We say that a CSP is bounds-consistent if for every variable X, and for both the lower-bound and upper-bound values of X, there exists some value of Y that satisfies the constraint between X and Y for every variable Y. This kind of bounds propagation is widely used in practical constraint problems.

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 flights F1 and F2 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]