Author (all other notes): Nikhil Sharma

Author (Bayes' Nets notes): Josh Hug and Jacky Liang, edited by Regina Wang

Author (Logic notes): Henry Zhu, edited by Peyrin Kao

Credit (Machine Learning and Logic notes): Some sections adapted from the textbook *Artificial Intelligence: A Modern Approach*.

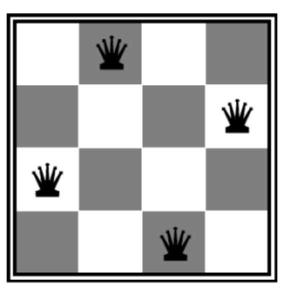
Last updated: August 26, 2023

Constraint Satisfaction Problems

In the previous note, we learned how to find optimal solutions to search problems, a type of **planning problem**. Now, we'll learn about solving a related class of problems, **constraint satisfaction problems** (CSPs). Unlike search problems, CSPs are a type of **identification problem**, problems in which we must simply identify whether a state is a goal state or not, with no regard to how we arrive at that goal. CSPs are defined by three factors:

1. *Variables* - CSPs possess a set of N variables $X_1,...,X_N$ that can each take on a single value from some defined set of values.
2. *Domain* - A set $\{x_1,...,x_d\}$ representing all possible values that a CSP variable can take on.
3. *Constraints* - Constraints define restrictions on the values of variables, potentially with regard to other variables.

Consider the N-queens identification problem: given an $N \times N$ chessboard, can we find a configuration in which to place N queens on the board such that no two queens attack each another?



We can formulate this problem as a CSP as follows:

1. *Variables* - X_{ij}, with $0 \leq i, j < N$. Each X_{ij} represents a grid position on our $N \times N$ chessboard, with i and j specifying the row and column number respectively.

2. *Domain* - $\{0,1\}$. Each X_{ij} can take on either the value 0 or 1, a boolean value representing the existence of a queen at position (i,j) on the board.

3. *Constraints* -

   - $\forall i,j,k\ (X_{ij},X_{ik}) \in \{(0,0),(0,1),(1,0)\}$. This constraint states that if two variables have the same value for i, only one of them can take on a value of 1. This effectively encapsulates the condition that no two queens can be in the same row.
   - $\forall i,j,k\ (X_{ij},X_{kj}) \in \{(0,0),(0,1),(1,0)\}$. Almost identically to the previous constraint, this constraint states that if two variables have the same value for j, only one of them can take on a value of 1, encapsulating the condition that no two queens can be in the same column.
   - $\forall i,j,k\ (X_{ij},X_{i+k,j+k}) \in \{(0,0),(0,1),(1,0)\}$. With similar reasoning as above, we can see that this constraint and the next represent the conditions that no two queens can be in the same major or minor diagonals, respectively.
   - $\forall i,j,k\ (X_{ij},X_{i+k,j-k}) \in \{(0,0),(0,1),(1,0)\}$.
   - $\sum\limits_{i,j} X_{ij} = N$. This constraint states that we must have exactly N grid positions marked with a 1, and all others marked with a 0, capturing the requirement that there are exactly N queens on the board.

Constraint satisfaction problems are **NP-hard**, which loosely means that there exists no known algorithm for finding solutions to them in polynomial time. Given a problem with N variables with domain of size $O(d)$ for each variable, there are $O(d^N)$ possible assignments, exponential in the number of variables. We can often get around this caveat by formulating CSPs as search problems, defining states as **partial assignments** (variable assignments to CSPs where some variables have been assigned values while others have not). Correspondingly, the successor function for a CSP state outputs all states with one new variable assigned, and the goal test verifies all variables are assigned and all constraints are satisfied in the state it's testing. Constraint satisfaction problems tend to have significantly more structure than traditional search problems, and we can exploit this structure by combining the above formulation with appropriate heuristics to hone in on solutions in a feasible amount of time.

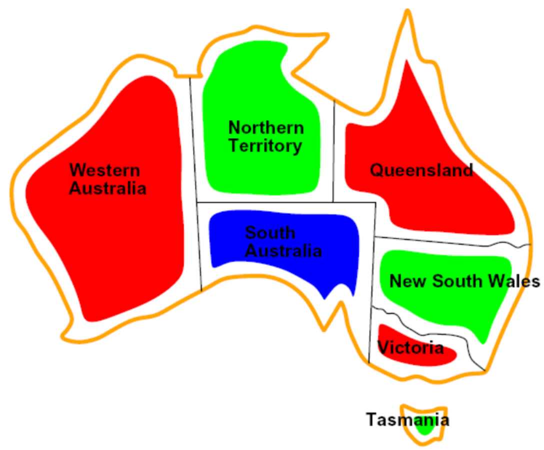
Constraint Graphs

Let's introduce a second CSP example: map coloring. Map coloring solves the problem where we're given a set of colors and must color a map such that no two adjacent states or regions have the same color.

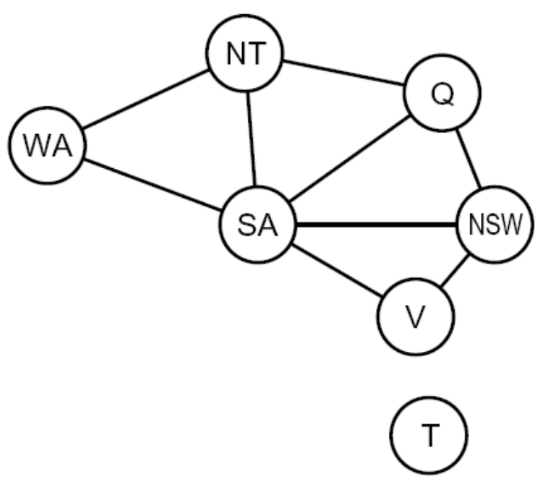
Constraint satisfaction problems are often represented as constraint graphs, where nodes represent variables and edges represent constraints between them. There are many different types of constraints, and each is handled slightly differently:

- *Unary Constraints* - Unary constraints involve a single variable in the CSP. They are not represented in constraint graphs, instead simply being used to prune the domain of the variable they constrain when necessary.
- *Binary Constraints* - Binary constraints involve two variables. They're represented in constraint graphs as traditional graph edges.
- *Higher-order Constraints* - Constraints involving three or more variables can also be represented with edges in a CSP graph, they just look slightly unconventional.

Consider map coloring the map of Australia:



The constraints in this problem are simply that no two adjacent states can be the same color. As a result, by drawing an edge between every pair of states that are adjacent to one another, we can generate the constraint graph for the map coloring of Australia as follows:



The value of constraint graphs is that we can use them to extract valuable information about the structure of the CSPs we are solving. By analyzing the graph of a CSP, we can determine things about it like whether it's sparsely or densely connected/constrained and whether or not it's tree-structured. We'll cover this more in depth as we discuss solving constraint satisfaction problems in more detail.

Solving Constraint Satisfaction Problems

Constraint satisfaction problems are traditionally solved using a search algorithm known as **backtracking search**. Backtracking search is an optimization on depth first search used specifically for the problem of constraint satisfaction, with improvements coming from two main principles:

1. Fix an ordering for variables, and select values for variables in this order. Because assignments are commutative (e.g. assigning $WA = Red,\ NT = Green$ is identical to $NT = Green,\ WA = Red$), this is valid.

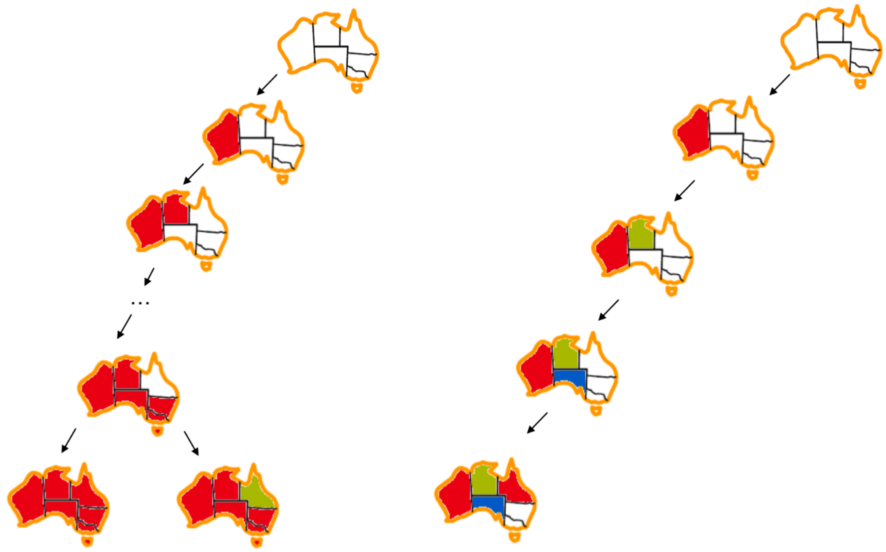
2. When selecting values for a variable, only select values that don't conflict with any previously assigned values. If no such values exist, backtrack and return to the previous variable, changing its value.

The pseudocode for how recursive backtracking works is presented below:

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
         add {var = value} to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         remove {var = value} from assignment
   return failure
```

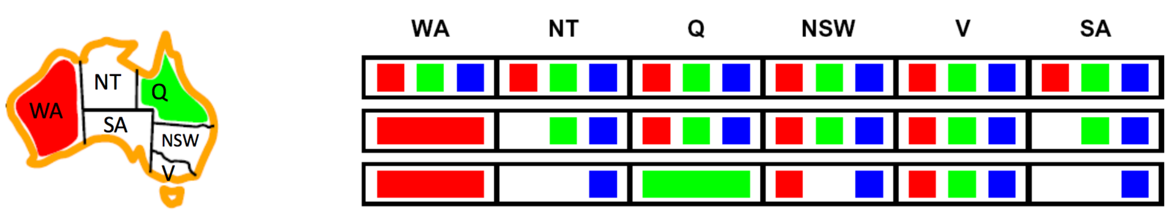
For a visualization of how this process works, consider the partial search trees for both depth first search and backtracking search in map coloring:



Note how DFS regretfully colors everything red before ever realizing the need for change, and even then doesn't move too far in the right direction towards a solution. On the other hand, backtracking search only assigns a value to a variable if that value violates no constraints, leading to a significantly less backtracking. Though backtracking search is a vast improvement over the brute-forcing of depth first search, we can get more gains in speed still with further improvements through filtering, variable/value ordering, and structural explotation.

Filtering

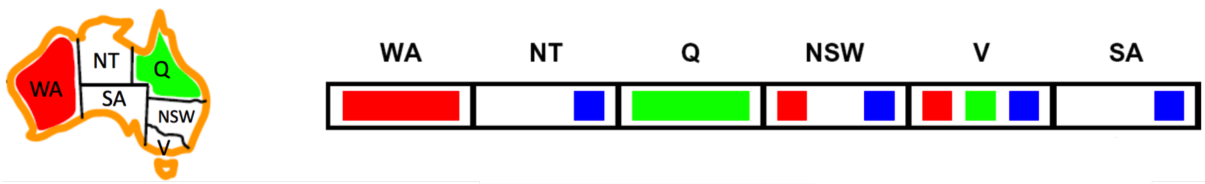
The first improvement to CSP performance we'll consider is **filtering**, which checks if we can prune the domains of unassigned variables ahead of time by removing values we know will result in backtracking. A naïve method for filtering is **forward checking**, which whenever a value is assigned to a variable X_i, prunes the domains of unassigned variables that share a constraint with X_i that would violate the constraint if assigned. Whenever a new variable is assigned, we can run forward checking and prune the domains of unassigned variables adjacent to the newly assigned variable in the constraint graph. Consider our map coloring example, with unassigned variables and their potential values:



Note how as we assign $WA = red$ and then $Q = green$, the size of the domains for NT, NSW, and SA (states adjacent to WA, Q, or both) decrease in size as values are eliminated. The idea of forward checking can be generalized into the principle of **arc consistency**. For arc consistency, we interpret each undirected edge of the constraint graph for a CSP as two directed edges pointing in opposite directions. Each of these directed edges is called an **arc**. The arc consistency algorithm works as follows:

- Begin by storing all arcs in the constraint graph for the CSP in a queue Q.
- Iteratively remove arcs from Q and enforce the condition that in each removed arc $X_i \longrightarrow X_j$, for every remaining value v for the tail variable X_i, there is at least one remaining value w for the head variable X_j such that $X_i = v, X_j = w$ does not violate any constraints. If some value v for X_i would not work with any of the remaining values for X_j, we remove v from the set of possible values for X_i.
- If at least one value is removed for X_i when enforcing arc consistency for an arc $X_i \longrightarrow X_j$, add arcs of the form $X_k \longrightarrow X_i$ to Q, for all unassigned variables X_k. If an arc $X_k \longrightarrow X_i$ is already in Q during this step, it doesn't need to be added again.
- Continue until Q is empty, or the domain of some variable is empty and triggers a backtrack.

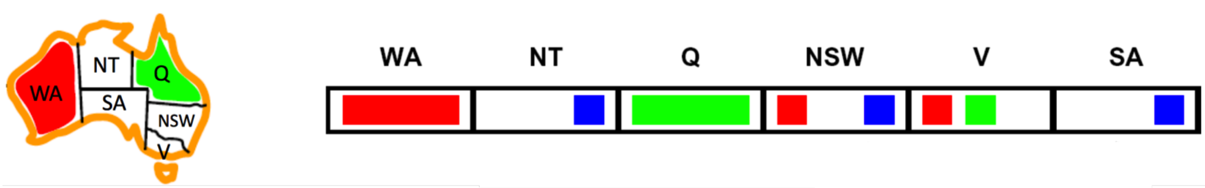
The arc consistency algorithm is typically not the most intuitive, so let's walk through a quick example with map coloring:



We begin by adding all arcs between unassigned variables sharing a constraint to a queue Q, which gives us

$$Q = [SA \rightarrow V, V \rightarrow SA, SA \rightarrow NSW, NSW \rightarrow SA, SA \rightarrow NT, NT \rightarrow SA, V \rightarrow NSW, NSW \rightarrow V]$$

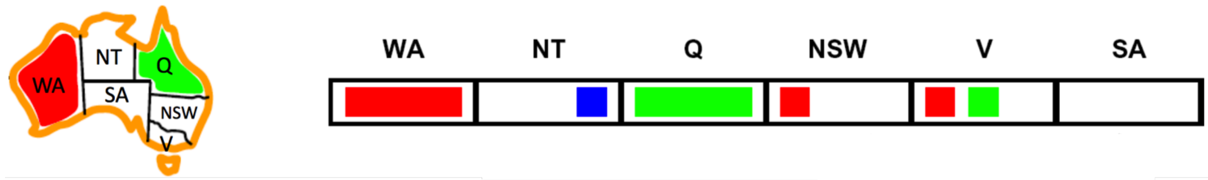
For our first arc, $SA \rightarrow V$, we see that for every value in the domain of SA, $\{blue\}$, there is *at least* one value in the domain of V, $\{red, green, blue\}$, that violates no constraints, and so no values need to be pruned from SA's domain. However, for our next arc $V \rightarrow SA$, if we set $V = blue$ we see that SA will have no remaining values that violate no constraints, and so we prune *blue* from V's domain.



Because we pruned a value from the domain of V, we need to enqueue all arcs with V at the head - $SA \rightarrow V$, $NSW \rightarrow V$. Since $NSW \rightarrow V$ is already in Q, we only need to add $SA \rightarrow V$, leaving us with our updated queue

$$Q = [SA \rightarrow NSW, NSW \rightarrow SA, SA \rightarrow NT, NT \rightarrow SA, V \rightarrow NSW, NSW \rightarrow V, SA \rightarrow V]$$

We can continue this process until we eventually remove the arc $SA \rightarrow NT$ from Q. Enforcing arc consistency on this arc removes *blue* from SA's domain, leaving it empty and triggering a backtrack. Note that the arc $NSW \rightarrow SA$ appears before $SA \rightarrow NT$ in Q and that enforcing consistency on this arc removes *blue* from the domain of NSW.



Arc consistency is typically implemented with the AC-3 algorithm (Arc Consistency Algorithm #3), for which the pseudocode is as follows:

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do                      tail      head
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
---------------------------------------------------------------------------
function REMOVE-INCONSISTENT-VALUES( Xi, Xj) returns true iff succeeds
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
         then delete x from DOMAIN[Xi];  removed ← true
   return removed
```

The AC-3 algorithm has a worst case time complexity of $O(ed^3)$, where e is the number of arcs (directed edges) and d is the size of the largest domain. Overall, arc consistency is more holistic of a domain pruning technique than forward checking and leads to fewer backtracks, but requires running significantly more computation in order to enforce. Accordingly, it's important to take into account this tradeoff when deciding which filtering technique to implement for the CSP you're attempting to solve.

As an interesting parting note about consistency, arc consistency is a subset of a more generalized notion of consistency known as **k-consistency**, which when enforced guarantees that for any set of k nodes in the CSP, a consistent assignment to any subset of $k-1$ nodes guarantees that the k^{th} node will have at least one consistent value. This idea can be further extended through the idea of **strong k-consistency**. A graph that is strong k-consistent possesses the property that any subset of k nodes is not only k-consistent but also $k-1, k-2, \ldots, 1$ consistent as well. Not surprisingly, imposing a higher degree of consistency on a CSP is more expensive to compute. Under this generalized definition for consistency, we can see that arc consistency is equivalent to 2-consistency.