AIES Notes: Constraint Satisfaction Problem [Lecture 7]

In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.

Constraint Satisfaction Problem

There are kind of problem for which **identifying the goal** is important rather than **finding the path to the goal.** Where the goal itself is important, which is known as **identification** problem. There are wide variety of identification problems which can be solved more efficiently if we use a **factored representation** for each **state:** a set of **variables,** each of which has a **value.**

A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **Constraint Satisfaction Problem**, or CSP. CSP search algorithms take advantage of the structure of states and use **general-purpose rather than problem-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.

Defining Constraint Satisfaction Problems

A constraint satisfaction problem consists of three components, X,D, and C:

X is a set of variables, $\{X_1, \ldots, X_n\}$.
D is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.
C is a set of constraints that specify allowable combinations of values.

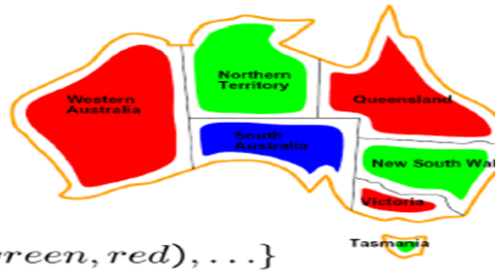
Each domain D_i consists of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consists of a pair <scope, rel>, where scope is a tuple of variables that participate in the constraint and *rel* is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation.

Example problem: Map coloring

- Variables: $WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T$
- Domain: $D = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors

$$WA \neq NT$$

$$(WA, NT) \in \{(red, green), (red, blue), (green, red), \ldots\}$$



- Solutions are assignments satisfying all constraints, e.g.:

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

Why formulate a problem as a CSP? One reason is that the CSPs yield a **natural representation** for a wide variety of problems; if you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique. In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space.

Constraint Graph

It can be helpful to visualize a CSP as a constraint graph, as shown in Figure 6.1(b). The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

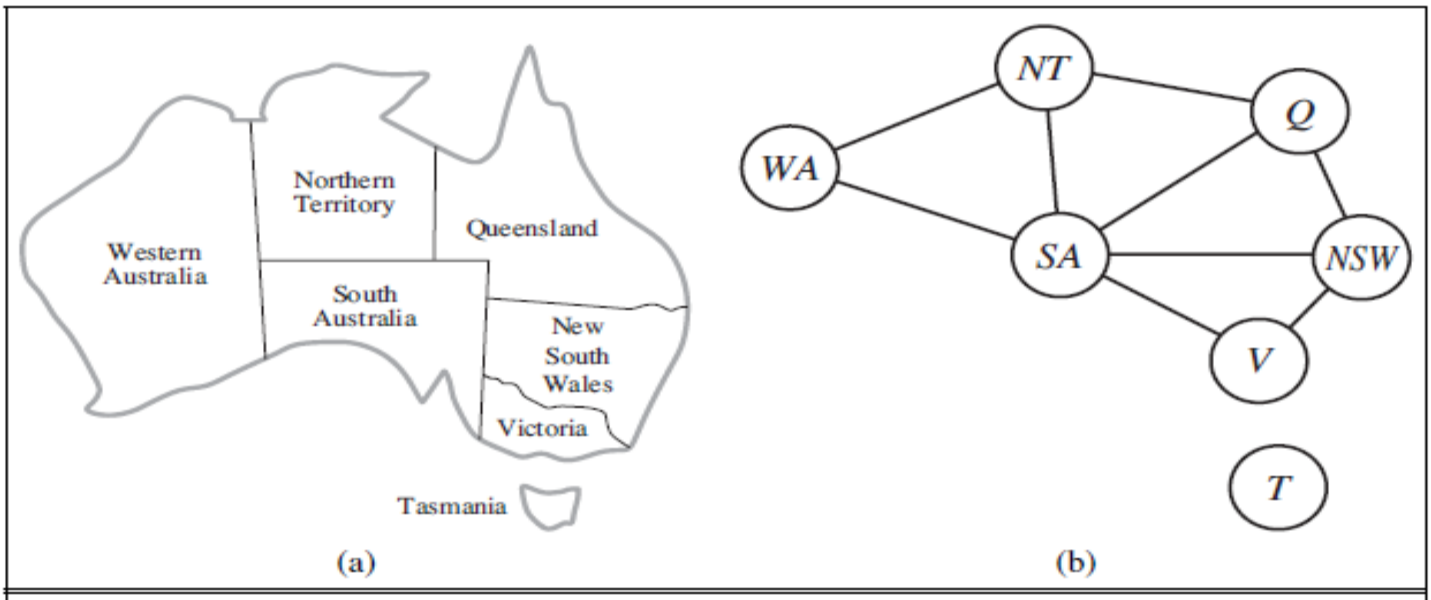


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Backtracking Search for CSPs

Our seemingly reasonable but naive formulation ignores crucial property common to all CSPs: **commutativity**. A problem is commutative if the order of application of any given set of actions has no effect on the outcome. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we need only consider a single variable at each node in the search tree. For example, at the root node of a search tree for coloring the map of Australia, we might make a choice between SA=red, SA=green, and SA=blue, but we would never choose between SA=red and WA=blue. With this restriction, the number of leaves is $\mathbf{d^n}$, as we would hope.

The term **backtracking search** is used for a **depth-first search** that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

The Big Idea

■ **Idea 1:** Only consider a single variable at each point
- Variable assignments are commutative, so fix ordering
- I.e., [WA = red then NT = green] same as [NT = green then WA = red]
- Only need to consider assignments to a single variable at each step
- How many leaves are there?

■ **Idea 2:** Only allow legal assignments at each point
- I.e. consider only values which do not conflict previous assignments
- Might have to do some computation to figure out whether a value is ok
- "Incremental goal test"

■ Depth-first search for CSPs with these two improvements is called backtracking search (useless name, really)

■ Backtracking search is the basic uninformed algorithm for CSPs

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

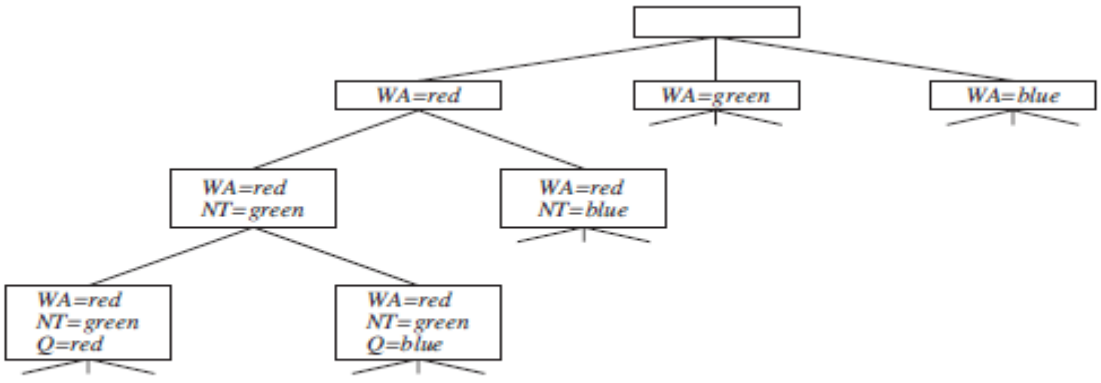


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

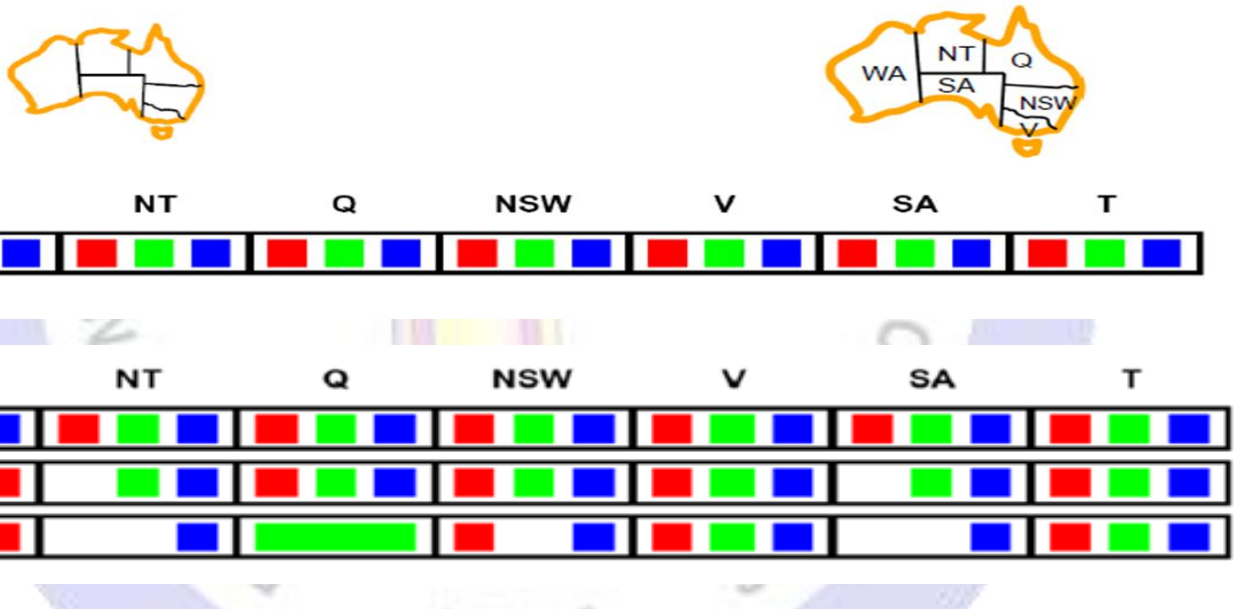
Improving Backtracking

■ General-purpose ideas give huge gains in speed

- **Ordering:**
  - Which variable should be assigned next?
  - In what order should its values be tried?
- **Filtering:** Can we detect inevitable failure early?
- **Structure:** Can we exploit the problem structure?

Filtering: Forward Checking

- Idea: Keep track of remaining legal values for unassigned variables (using immediate constraints)
- Idea: Terminate when any variable has no legal values



	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA*=*red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	(R)	B	(G)	R B	R G B	B	R G B
After *V*=*blue*	(R)	B	(G)	R	(B)		R G B

Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$ is assigned, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$ is assigned, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

Filtering: Constraint Propagation

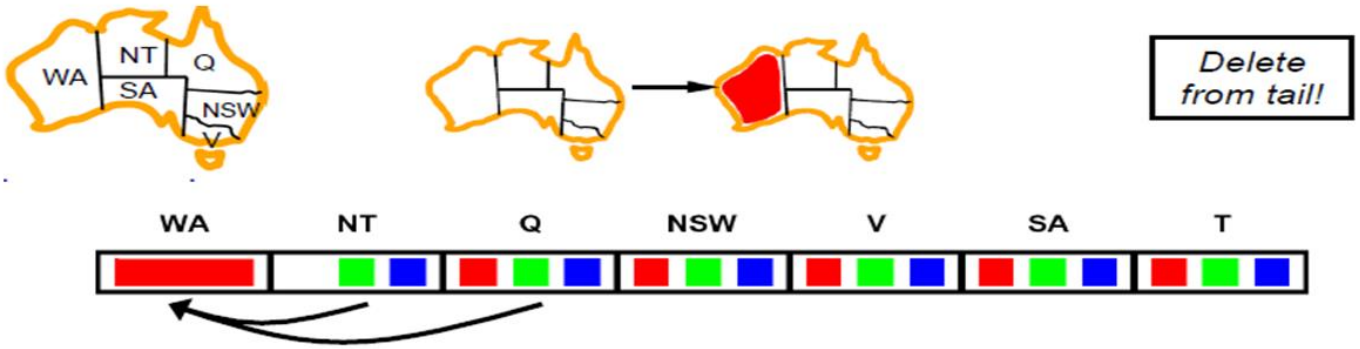
■ Forward checking propagates information from assigned to unassigned variables, but **doesn't provide early detection for all failures**:

- ■ NT and SA cannot both be blue!
- ■ Why didn't we detect this yet?

■ **Constraint propagation** propagates from constraint to constraint

Consistency of an Arc

- An arc X → Y is consistent iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint



- What happens?
- Forward checking = Enforcing consistency of each arc pointing to the new assignment

---

Arc Consistency of a CSP

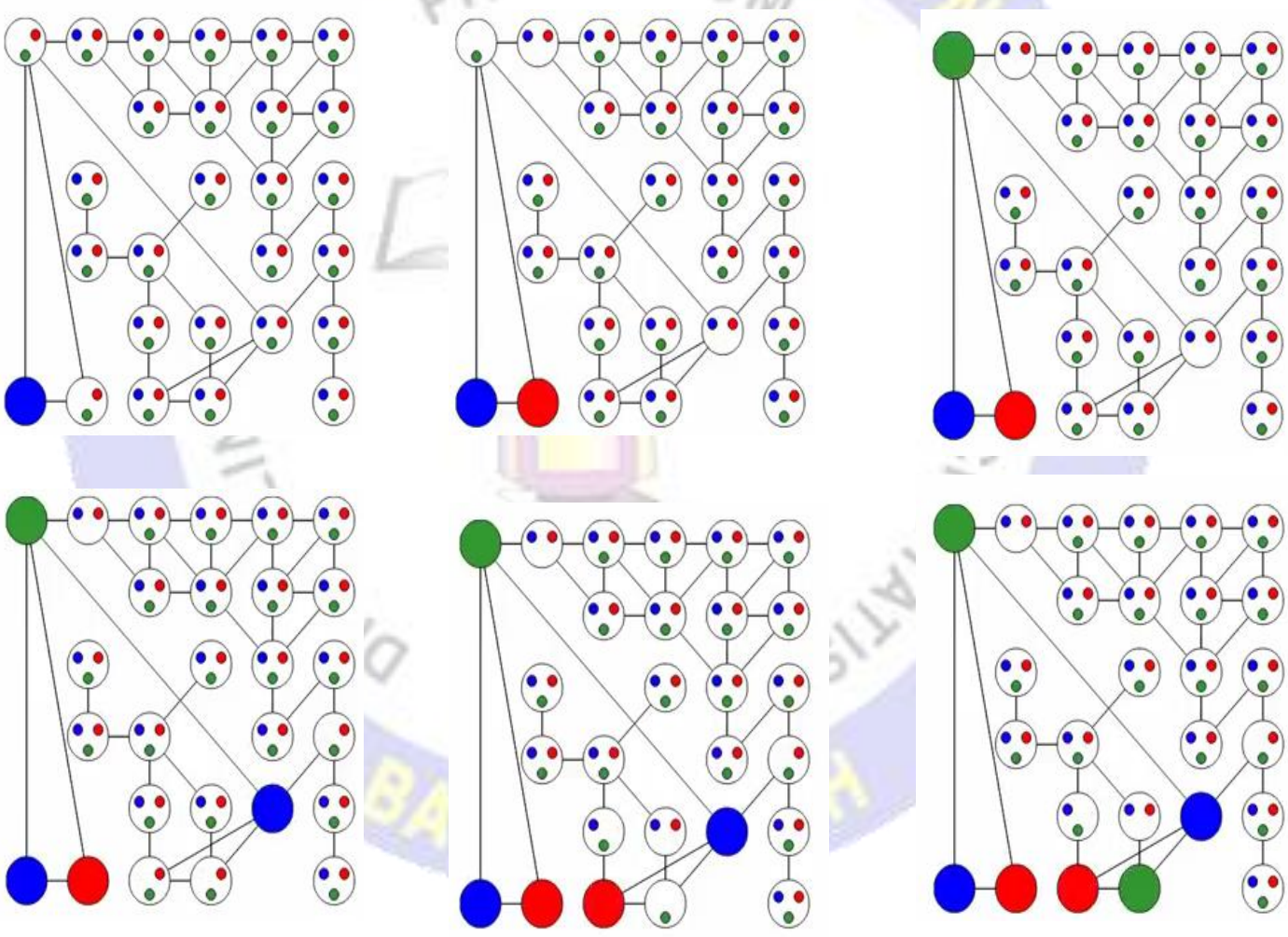
- A simple form of propagation makes sure all arcs are consistent:

- If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- What's the downside of enforcing arc consistency?
- Can be run as a preprocessor or after each assignment

Ordering: Minimum Remaining Value (MRV)

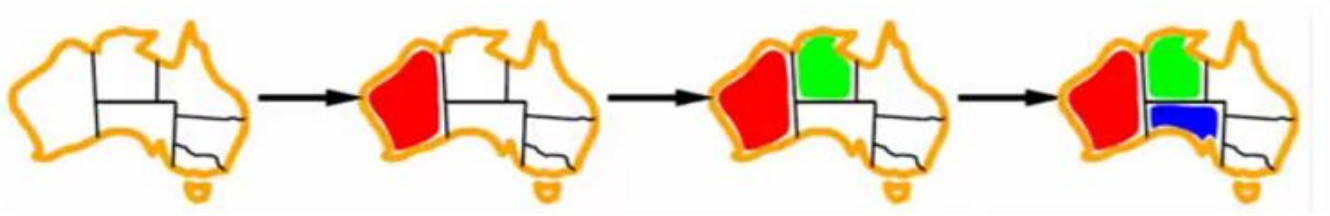
This static variable ordering seldom results in the most efficient search. For example, after the assignments for WA=red and NT =green in, there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced. This intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum-remaining-values (MRV) heuristic**. It also has been called the "**most constrained variable**" or "**fail-first**" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately avoiding pointless searches through other variables.

MRV Example:



The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.

- Minimum remaining values (MRV):
  - Choose the variable with the fewest legal values

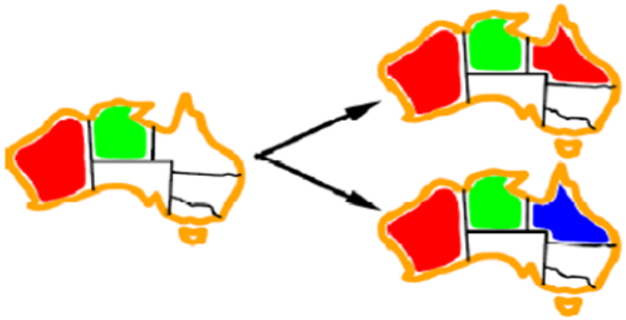


- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering

Ordering: Least Constraining Value

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the least-constraining-value heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure 6.1 we have generated the partial assignment with WA=red and NT =green and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue. In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments. Of course, if we are trying to find all the solutions to a problem, not just the first one, then the ordering does not matter because we have to consider every value anyway. The same holds if there are no solutions to the problem.

- Given a choice of variable:
  - Choose the *least constraining value*
  - The one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this!

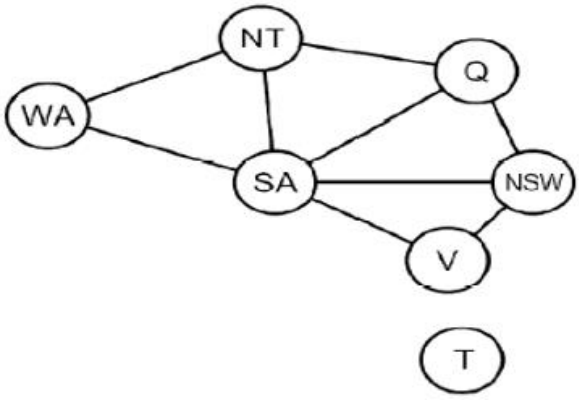


- Why least rather than most?
- Combining these heuristics makes 1000 queens feasible

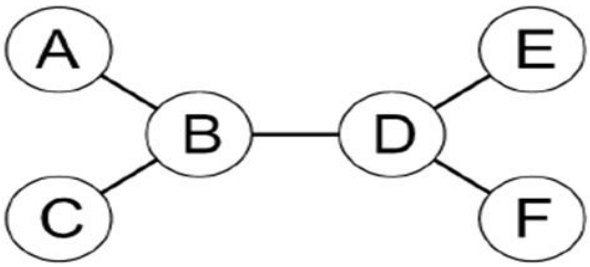
Why should variable selection be fail-first, but value selection be fail-last? It turns out that, for a wide variety of problems, a variable ordering that chooses a variable with the minimum number of remaining values helps minimize the number of nodes in the search tree by pruning larger parts of the tree earlier. For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

Problem Structure

- Tasmania and mainland are independent subproblems
- Identifiable as connected components of constraint graph
- Suppose each subproblem has c variables out of n total
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



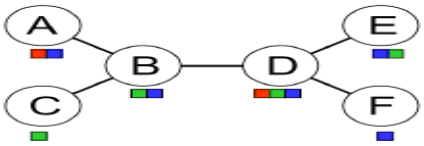
Tree Structure CSPs

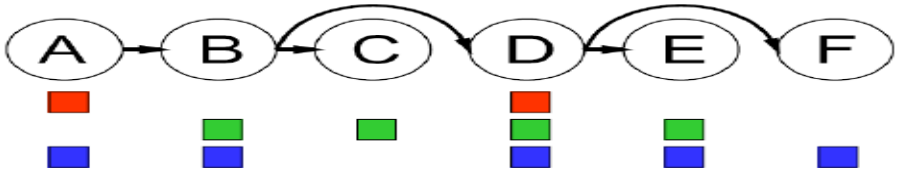


- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$
- This property also applies to probabilistic reasoning (later): an important example of the relation between syntactic restrictions and the complexity of reasoning.

Tree-Structured CSPs

- Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering





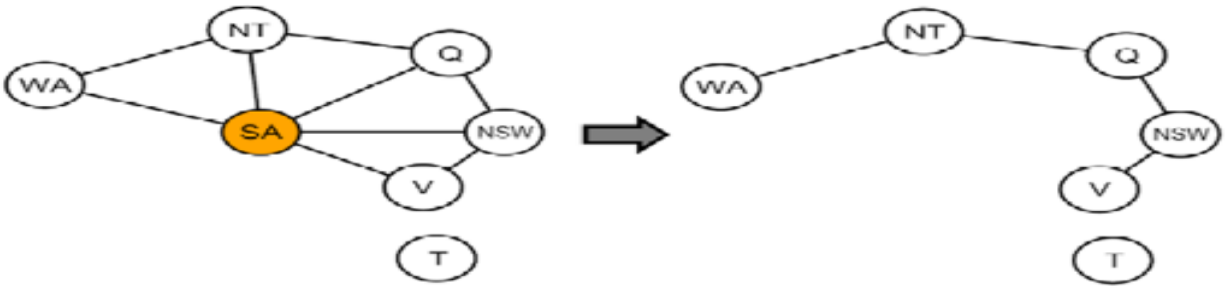
- For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
- For i = 1 : n, assign X_i consistently with Parent(X_i)
- Runtime: O(n d^2) (why?)

Tree-Structured CSPs

- Why does this work?
- Claim: After processing the right k nodes, given any satisfying assignment to the rest, the right k can be assigned (left to right) without backtracking
- Proof: Induction on position

- Why doesn't this algorithm work with loops?
- Note: we'll see this basic idea again with Bayes' nets

Nearly Tree-Structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime O((d^c) (n-c) d^2), very fast for small c

Summary

• Constraint satisfaction problems (CSPs) represent a state with a set of variable/value pairs and represent the conditions for a solution by a set of constraints on the variables. Many important real-world problems can be described as CSPs.

• A number of inference techniques use the constraints to infer which variable/value pairs are consistent and which are not. These include node, arc, path, and k-consistency.

• Backtracking search, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.

• The minimum-remaining-values and degree heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The least-constraining- value heuristic helps in deciding which value to try first for a given variable. Backtracking occurs when no legal assignment can be found for a variable. Conflict-directed back jumping backtracks directly to the source of the problem.

• Local search using the min-conflicts heuristic has also been applied to constraint satisfaction problems with great success.

• The complexity of solving a CSP is strongly related to the structure of its constraint graph. Tree-structured problems can be solved in linear time. Cut set conditioning can reduce a general CSP to a tree-structured one and is quite efficient if a small cut set can be found. Tree decomposition techniques transform the CSP into a tree of subproblems and are efficient if the tree width of the constraint graph is small.