UNIT – II

CHAPTER – 7

Constraint Satisfaction Problems

Introduction

- In previous chapters we have explored the idea that problems can be solved by searching in a **space of states**.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- An alternative representation like , a **factored representation** for each state is used in this chapter, i.e. 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.

Defining Constraint Satisfaction (CSP) Problems

- A constraint satisfaction problem consists of three components, X, D, and C:
  a. X is a set of variables, $\{X_1, \ldots, X_n\}$.
  b. D is a set of domains, $\{D_1, \ldots, D_n\}$, one for each variable.
  c. 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.
- For example, if X_1 and X_2 both have the domain *{A,B}*, then the constraint saying the two variables must have different values can be written as $<(X_1,X_2), [(A,B), (B,A)]>$ or as $<(X_1,X_2), X_1 \neq X_2>$

Defining Constraint Satisfaction Problems contd…

- To solve a CSP, we need to define a state space and the notion of a solution.
- Each state in a CSP is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_j = v_j , \ldots\}$.
- 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, and a **solution** to a CSP is a consistent.
- A **partial assignment** is one that assigns values to only some of the variables.

Example problem: Map coloring

- The figure given here is the map of Australia showing each of its states and territories.
- Here the task is to color each region either *red, green, or blue* in such a way that no 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 each 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:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\} .$$

Example problem: Map coloring contd…

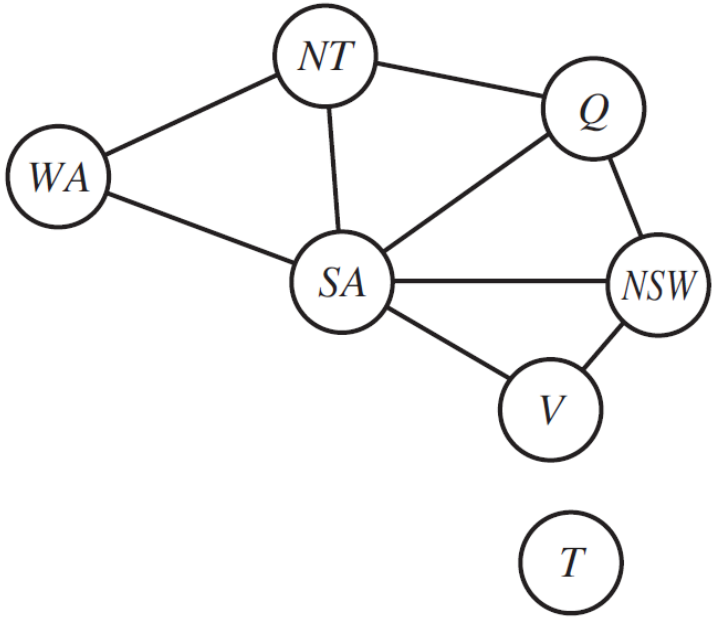
Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

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

There are many possible solutions to this problem, such as

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

It can be helpful to visualize a CSP as a **constraint graph as in figure.** The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.



Why formulate a problem as CSP?

- CSPs yield a natural representation for a wide variety of problems.
- CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space.
- 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 would have to consider $\mathbf{3^5} = 243$ assignments. With CSP we never have to consider blue as a value, so we have only $2^5 = 32$ assignments to look at, a reduction of 87%.
- In regular state-space search we can only ask: is this specific state a goal? Whereas in CSPs, once we find out that a partial assignment is not a solution, it immediately discards further refinements of the partial assignment. Also, 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 regular state-space search can be solved quickly when formulated as a CSP.

Example problem: Job Shop Scheduling

- Factories have the problem of scheduling a day's worth of jobs, subject to various constraints.
- In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the **assembly of a car**.
- The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

Example problem: Job Shop Scheduling contd…

- 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
  - Inspect the final assembly.
- We can represent the tasks with 15 variables:

$$X = \{Axle_F, Axle_B, Wheel_{RF}, Wheel_{LF}, Wheel_{RB}, Wheel_{LB}, Nuts_{RF}, Nuts_{LF}, Nuts_{RB}, Nuts_{LB}, Cap_{RF}, Cap_{LF}, Cap_{RB}, Cap_{LB}, Inspect\}$$

- The value of each variable is the time that the task starts.
- Next **precedence constraints** between individual tasks is to be represented.
- Whenever a task T1 must occur before task *T2*, and task *T1* takes duration *d1* to complete, then add an arithmetic constraint of the form : $T1 + d1 \leq T2$

Example problem: Job Shop Scheduling contd…

- In current example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, which can be presented as follows:

  $Axle_F + 10 \leq Wheel_{RF}$ $\quad\quad$ $Axle_F + 10 \leq Wheel_{LF}$

  $Axle_B + 10 \leq Wheel_{RB}$ $\quad\quad$ $Axle_B + 10 \leq Wheel_{LB}$

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

- 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 $Axle_F$ and $Axle_B$ must not overlap in time; either one comes first or the other does: $(Axle_F + 10 \leq Axle_B)$ **or** $(Axle_B + 10 \leq Axle_F)$

- Let the inspection comes last and takes 3 minutes. For every variable except Inspect add a constraint of the form $X + d_X \leq Inspect$

- Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. It can be achieved by limiting the domain of all variables as : $D_i = \{1, 2, 3, \ldots, 27\}$

Variations on the CSP formalism

- The domain can be combination of continuous or discrete and finite or infinite
- The simplest kind of CSP involves variables that have **discrete**, **finite domains**
  E.g. Map-coloring problems and scheduling with time limits
- A discrete domain can be **infinite**, such as the set of integers or strings.
  E.g. If no limit in job-scheduling problem, there would be an infinite number of start times for each variable.

- If the domains are infinite, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

- Instead, a **constraint language** must be used that understands constraints $T1 + d1 \leq T2$ directly, without enumerating the set of pairs of allowable values for (T_1, T_2)

- CSP with **continuous domains** are common in the real world and are widely studied in the field of operations research

Variations on the CSP formalism Cntd…

In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints.

1. The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green then it can expressed with the unary constraint as c = <(SA),SA = green>

2. A **binary constraint** relates two variables. For example, c = <(SA, NSW), (SA = NSW)>. A binary CSP is one with only binary constraints; it can be represented as a constraint graph shown previously.

3. Higher-order constraints such as asserting that the value of Y is between X and Z is the ternary constraint and represented as Between(X, Y,Z).

4. A constraint involving an arbitrary number of variables is called a **global constraint**.

Variations on the CSP formalism Cntd…

- Cryptarithmetic puzzle problem: It involves **global constraint** *Alldiff(X)*
- Representation of this puzzle as CSP

  X = {F, T, U, W, R, O}

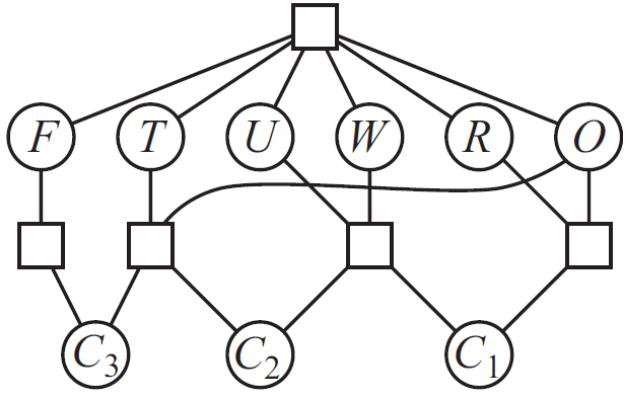
  D = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

  c = Alldiff(X)

- Each letter stands for a distinct digit
- The aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed.

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

- The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle).
- The variables C1, C2, and C3 represent the carry digits for the three columns



Constraint Propagation: Inference in CSPs

- Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts.
- Sometimes this preprocessing can solve the whole problem, so no search is required at all.
- The key idea is **local consistency** i.e. if each variable is treated as a **node** in a graph and each binary constraint as an **arc**, 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 as follows
  - Node Consistency
  - Arc Consistency
  - Path Consistency
  - K-Consistency

Constraint Propagation: Inference in CSPs Cntd…

1. 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 map-coloring problem, assume South Australian dislike green, then 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.

Constraint Propagation: Inference in CSPs Cntd…

2. 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)
- For example, consider the constraint $Y = X^2$ where the domain of both X and Y is the set of digits
- This constraint can explicitly be written as <(X, Y), {(0, 0), (1, 1), (2, 4), (3, 9)}>
- To make *X* arc-consistent with respect to Y, The domain of X is reduced to {0, 1, 2, 3}
- Similarly, Y 's domain becomes {0, 1, 4, 9} to make Y arc-consistent with respect to X, and hence whole CSP is arc-consistent.

Constraint Propagation: Inference in CSPs Cntd…

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary CSP with components $(X,\ D,\ C)$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST(*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 $\leftarrow$ *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 $\leftarrow$ *true*
  return *revised*

Simplified form of Pseudocode of AC-3

Steps of AC-3

1.Initialize:

Start with a queue of all arcs (directed variable pairs) in the CSP.

2.Iterate through the queue:

While the queue is not empty:

i. Remove an arc (X_i, X_j) from the queue.

ii. Apply the **Revise** procedure:

- Check if the domain of X_i can be reduced by removing values that have no valid support in X_j.
- If the domain of X_i is reduced:

Add all neighboring arcs of X_i (excluding X_j) back to the queue.

3.Termination:

1. If any domain becomes empty, the CSP is unsolvable.
2. Otherwise, the CSP is arc-consistent.

Constraint Propagation: Inference in CSPs Cntd…

3. Path consistency

- Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints).
- A stronger notion of consistency is needed to make progress on problems like map coloring.
- **Path consistency** tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.
- A set of two-variable *{Xi ,Xj}* is path-consistent with respect to a third variable Xm iff, 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.

Constraint Propagation: Inference in CSPs Cntd…

- **4. k-consistency**
- Stronger forms of propagation can be defined with the notion of 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.
- 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.

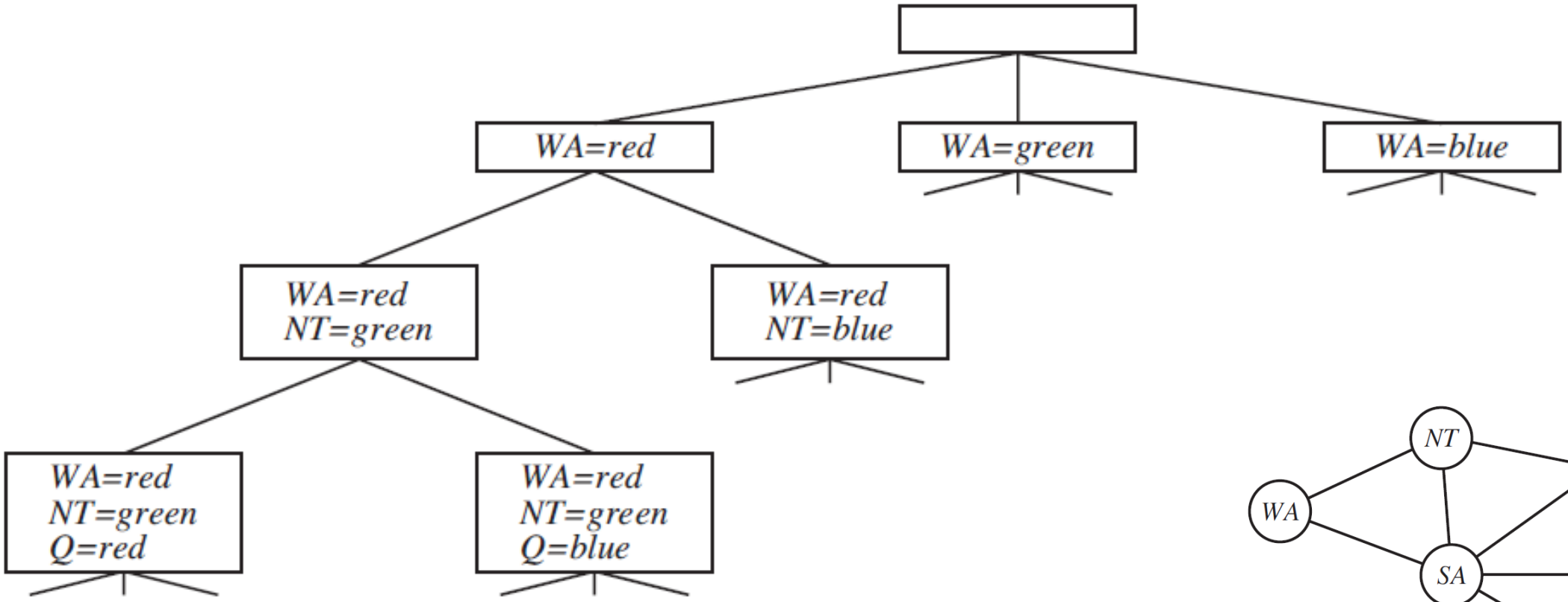
Backtracking Search for CSPs

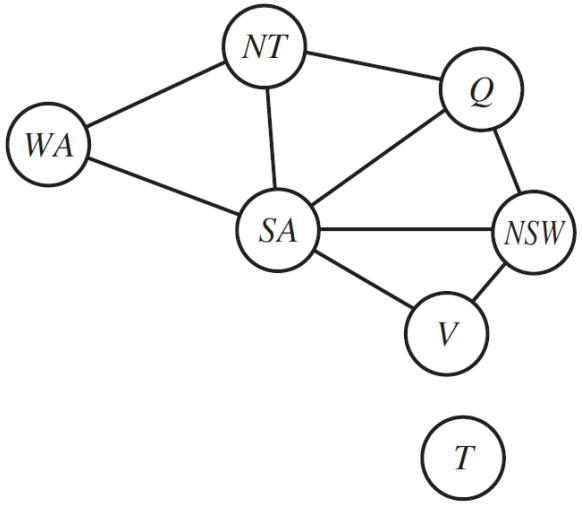
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return BACKTRACK({ }, csp)

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
```

Backtracking Search for Map coloring problem

Part of the search tree for the map-coloring problem





It continues further. The point where it is not possible to assign the value for any unassigned variable, that node will be killed and backtracking takes place.

Backtracking Search for CSPs Cntd…

The backtracking algorithm can be improvised by adding some sophistication to the unspecified functions in pseudocode and using them to address the following 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?

Backtracking Search for CSPs : **Variable and value ordering**

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
   - Choosing the variable with the fewest "legal" values—is called the **minimum remaining-values** (MRV) heuristic.
   - 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 i.e. 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.
   - 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. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

Backtracking Search for CSPs : **Variable and value ordering**

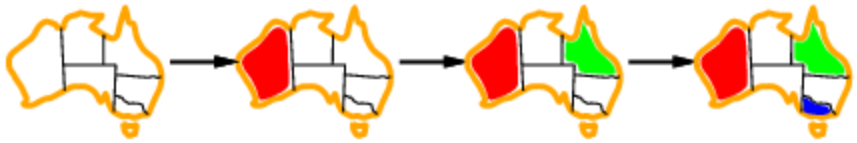
1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
   - Choosing the variable with the fewest "legal" values—is called the **Minimum Remaining-values** (MRV) heuristic.
   - The MRV heuristic doesn't help at all 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 i.e. 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.
   - 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. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

Interleaving search and inference

- Inference can be more powerful in the course of a search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.
- One of the simplest forms of inference is called **forward checking**.
- Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y ’s domain any value that is inconsistent with the value chosen for X.
- Because forward checking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

Interleaving search and inference: Forward Checking : Example

- Keep track of remaining legal values for unassigned variables
- Terminate search 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.

Interleaving search and inference: Forward Checking : Contd

- For many problems the search will be more effective if we combine the MRV heuristic with forward checking.
- Consider the same example as [revious, after assigning {WA=red}. Intuitively, it seems that that assignment constrains its neighbors, NT and SA, so we should handle those variables next, and then all the other variables will fall into place.
- That's exactly what happens with MRV: NT and SA have two values, so one of them is chosen first, then the other, then Q, NSW, and V in order. Finally T still has three values, and any one of them works.
- We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.
- The problem with forward checking 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. It shows that when WAis 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.
- But the algorithm called MAC (for **Maintaining Arc Consistency (MAC)**) detects this inconsistency.

Intelligent backtracking: **Looking backward**

- The BACKTRACKING-SEARCH algorithm has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited.
- A more intelligent approach to backtracking is to backtrack to a variable that might fix the problem
- To do this, we will keep track of a set of assignments that are in conflict with some value.
- The **backjumping** method backtracks to the *most recent* assignment in the conflict set
- This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.
- A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.
- On the other hand, **Constraint learning** is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**.
- No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.