Chapter 5 - Constraint Satisfaction Problems

Russell, S., & Norvig, P. (2022). *Artificial Intelligence - A Modern Approach* (4th global ed.). Pearson.

Contents

- 1. Defining Constraint Satisfaction Problems
- 2. Constraint Propagation: Inference in CSPs
- 3. Backtracking Search for CSPs
- 4. Local Search for CSPs

Constraint Satisfaction Problems

- We saw that domain-specific heuristics could estimate the cost of reaching the goal from a given state, but that from the point of view of the search algorithm, each state is atomic, or indivisible—a black box with no internal structure. For each problem we need domain-specific code to describe the transitions between states.
- Constraint Satisfaction Problems:
  - We treat states as more than just little black boxes leads to new search methods and a deeper understanding of problem structure.
  - We break open the black box by using 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**.

1. Defining Constraint Satisfaction Problems

- A constraint satisfaction problem consists of three components, $\mathcal{X}$, $\mathcal{D}$, and $\mathcal{C}$:
  - $\mathcal{X}$ is a set of variables, $\{X_1, ..., X_n\}$.
  - $\mathcal{D}$ is a set of domains, $\{D_1, ..., D_n\}$, one for each variable.
  - $\mathcal{C}$ is a set of constraints that specify allowable combinations of values.
- A domain, D_i, consists of a set of allowable values, $\{v_1, ..., v_k\}$, for variable X_i. Different variables can have different domains of different sizes.
  - For example, a Boolean variable would have the domain $\{true, false\}$.
- Each constraint C_j 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 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 X_1 and X_2 both have the domain $\{1, 2, 3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as $<(X_1, X_2), \{(3, 1), (3, 2), (2, 1)\}>$ or as $<(X_1, X_2), X_1 > X_2>$.

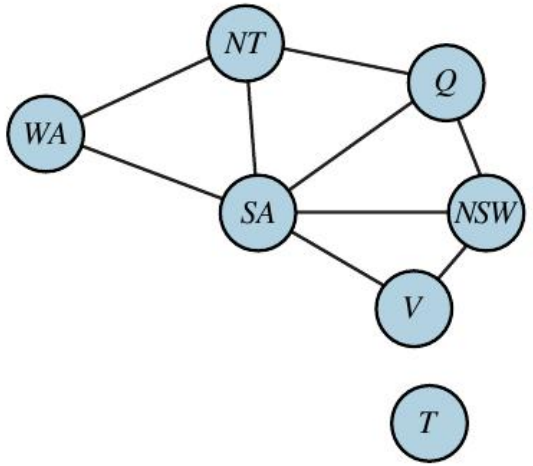
1. Defining Constraint Satisfaction Problems

- **Assignments**
  - CSPs deal with **assignments** of values to variables, $\{X_i = v_i, X_j = v_j, ...\}$.
- **Consistent Assignments**
  - An assignment that does not violate any constraints is called a **consistent** or legal assignment.
- **Complete Assignments**
  - A complete assignment is one in which every variable is assigned a value.
- **A Solution to a CSP**
  - A solution 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.

1. Defining Constraint Satisfaction Problems

Example Problem: Map Coloring

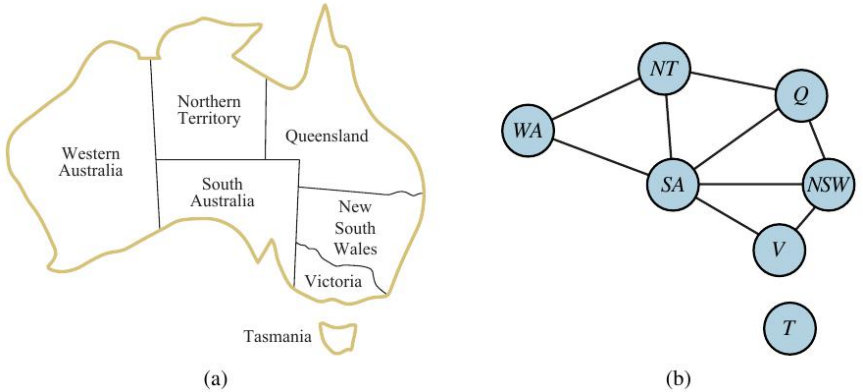
- We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.
- It can be helpful to visualize a (binary) CSP as a **constraint graph**. The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



- **Figure 5.1** (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.

1. Defining Constraint Satisfaction Problems

Example Problem: Map Coloring



- Variables
  - $\mathcal{X} = \{WA, NT, Q, NSW, V, SA, T\}$
- Domain of every variable is the set $D_i = \{red, green, blue\}$
- The constraints require neighboring regions to have distinct colors.
  - $\mathcal{C} = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$
    - Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle(SA, WA), SA \neq WA\rangle$

1. Defining Constraint Satisfaction Problems

Example Problem: Job-Shop Scheduling

- 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 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.

1. Defining Constraint Satisfaction Problems

Example Problem: Job-Shop Scheduling

- We 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:

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

- Next, we represent **precedence constraints** between individual tasks. Whenever a task T_1 must occur before task T_2, and task T_1 takes duration d_1 to complete, we add an arithmetic constraint of the form

  $T_1 + d_1 \leq T_2$

1. Defining Constraint Satisfaction Problems

Example Problem: Job-Shop Scheduling

- 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
  - $Axle_F + 10 \leq Wheel_{RF}$; $AxleF + 10 \leq Wheel_{LF}$;
  - $Axle_B + 10 \leq Wheel_{RB}$; $AxleB + 10 \leq Wheel_{LB}$.
- 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):
  - $Wheel_{RF} + 1 \leq Nuts_{RF}$; $Nuts_{RF} + 2 \leq Cap_{RF}$;
  - $Wheel_{LF} + 1 \leq Nuts_{LF}$; $Nuts_{LF} + 2 \leq Cap_{LF}$;
  - $Wheel_{RB} + 1 \leq Nuts_{RB}$; $Nuts_{RB} + 2 \leq Cap_{RB}$;
  - $Wheel_{LB} + 1 \leq Nuts_{LB}$; $Nuts_{LB} + 2 \leq Cap_{LB}$.

1. Defining Constraint Satisfaction Problems

Example Problem: Job-Shop Scheduling

- 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:

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

- 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 $X + d_X \leq Inspect$.
- 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:

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

1. Defining Constraint Satisfaction Problems
Variations on the CSP Formalism

- Types of domains
  - Discrete, finite domain
    - Map-coloring, Scheduling with time limits, The 8-queens problem where the variables $Q_1, ..., Q_8$ correspond to the queens in columns 1 to 8, and the domain of each variable specifies the possible row numbers for the queen in that column, $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$. The constraints say that no two queens can be in the same row or diagonal.
  - Discrete, infinite domain (such as the set of integers or strings)
    - With infinite domains, we must use implicit constraints like $T_1 + d_1 \leq T_2$ rather than explicit tuples of values.
  - Continuous domain
    - Common in the real world
    - The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities.
    - Linear programming problems can be solved in time polynomial in the number of variables.
    - Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on. These problems constitute an important area of applied mathematics.

1. Defining Constraint Satisfaction Problems
Variations on the CSP Formalism

- Type of constraints
  - Unary constraint: which restricts the value of a single variable.
    - $<(SA), SA \neq green>$
  - Binary constraint: relates two vairables.
    - $<(SA, WA), SA \neq WA>$
    - A **binary CSP** is one with only unary and binary constraints; it can be represented as a constraint graph.
  - We can also define higher-order constraints. The ternary constraint:
    - $<(X, Y, Z), X < Y < Z>$
  - Global constraint: A constraint involving an arbitrary number of variables is called a global constraint. (The name is traditional but confusing because a global constraint need not involve all the variables in a problem).
    - One of the most common global constraints is ***Alldiff***, which says that all of the variables involved in the constraint must have different values. In Sudoku problems, all variables in a row, column, or 3 × 3 box must satisfy an *Alldiff* constraint.

1. Defining Constraint Satisfaction Problems
Variations on the CSP Formalism

- **Cryptarithmetic** puzzles
  - Each letter in a cryptarithmetic puzzle represents a different digit.
  - This would be represented as the global constraint *Alldiff*(*F, T, U, W, R, O*).
  - The addition constraints on the four columns of the puzzle can be written as the following *n*-ary constraints:
    - $O + O = R + 10C_1$
    - $C_1 + W + W = U + 10C_2$
    - $C_2 + T + T = O + 10C_3$
    - $C_3 = F$,

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

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

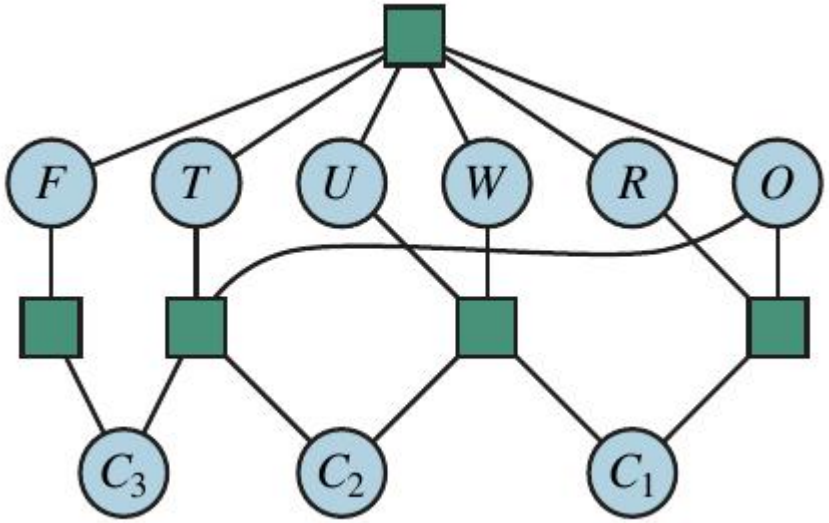
1. Defining Constraint Satisfaction Problems

Variations on the CSP Formalism

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

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

  - $O + O = R + 10C_1$
  - $C_1 + W + W = U + 10C_2$
  - $C_2 + T + T = O + 10C_3$
  - $C_3 = F$,



1. Defining Constraint Satisfaction Problems

Variations on the CSP Formalism

- Every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced.
  - This means that we could transform any CSP into one with only binary constraints—which certainly makes the life of the algorithm designer simpler.
- Another way to convert an *n*-ary CSP to a binary one is the **dual graph** transformation:
  - Create a new graph in which there will be one variable for each constraint in the original graph, and one binary constraint for each pair of constraints in the original graph that share variables.

1. Defining Constraint Satisfaction Problems

Variations on the CSP Formalism

- **Dual graph**
  - Consider a CSP with the variables $\mathcal{X} = \{X, Y, Z\}$, each with the domain $\{1, 2, 3, 4, 5\}$, and with the two constraints

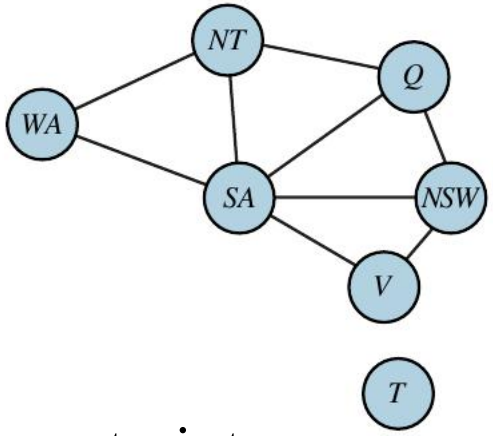
$$C_1 : <(X, Y, Z), X + Y = Z> \text{ and } C_2 : <(X, Y), X + 1 = Y>.$$

  - Then the dual graph would have the variables $\mathcal{X} = \{C_1, C_2\}$, where the domain of the C_1 variable in the dual graph is the set of $\{(x_i, y_j, z_k)\}$ tuples from the C_1 constraint in the original problem, and similarly the domain of C_2 is the set of $\{(x_i, y_j)\}$ tuples.
  - The dual graph has the binary constraint $<(C_1, C_2), R_1>$, where R_1 is a new relation that defines the constraint between C_1 and C_2 ; in this case it would be $R_1 = \{((1, 2, 3), (1, 2)), ((2, 3, 5), (2, 3))\}$.

2. Constraint Propagation: Inference in CSPs

- An atomic state-space search algorithm makes progress in only one way: by expanding a node to visit the successors.
- A CSP algorithm has choices.
  - It can generate successors by choosing a new variable assignment, or
  - It can do a specific type of inference called **constraint propagation**: 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.

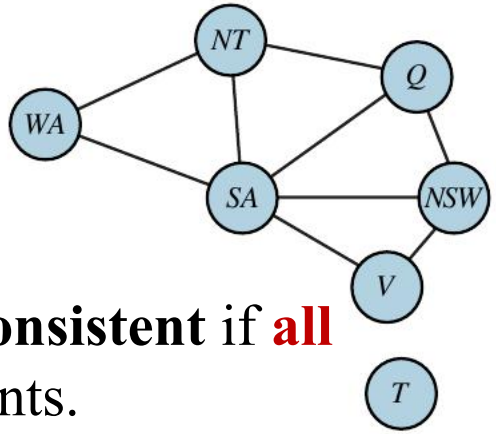
2. Constraint Propagation: Inference in CSPs



- 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.

2. Constraint Propagation: Inference in CSPs

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.
- As mentioned earlier, it is also possible to transform all *n*-ary constraints into binary ones. Because of this, some CSP solvers work with only binary constraints, expecting the user to eliminate the other constraints ahead of time.

2. Constraint Propagation: Inference in CSPs

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 arc-consistent 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

$$<(X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9)\}>.$$

  - To make X arc-consistent with respect to Y, we reduce X's domain to $\{0, 1, 2, 3\}$. To make Y arc-consistent with respect to X, then Y's domain becomes $\{0, 1, 4, 9\}$, and the whole CSP is arc-consistent.

2. Constraint Propagation: Inference in CSPs
Arc Consistency

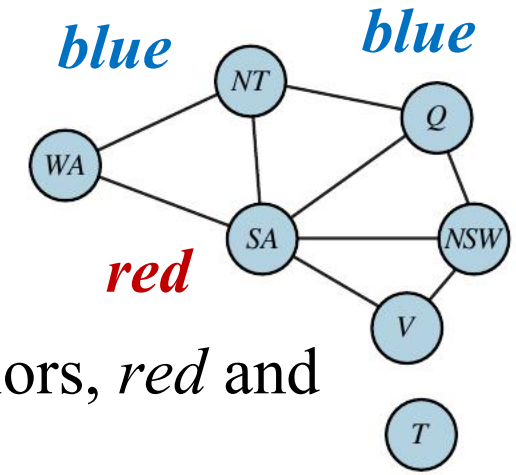
```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  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
```

(Each binary constraint becomes two arcs, one in each direction.)

- **Figure 5.3** The arc-consistency algorithm AC-3.
- After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved.
- The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it was the third version developed in the paper.

2. Constraint Propagation: Inference in CSPs

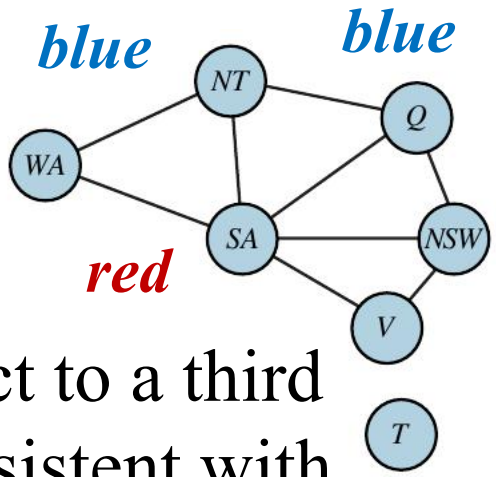
Path Consistency



- Suppose we are to color the map of Australia with just two colors, *red* and *blue*.
  - Arc consistency does nothing because every constraint can be satisfied individually with *red* at one end and *blue* at the other. 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 colors for them alone.
  - Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints). To make progress on problems like map coloring, we need a stronger notion of consistency.
- **Path consistency** tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.

2. Constraint Propagation: Inference in CSPs

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.

2. Constraint Propagation: Inference in CSPs

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.

2. Constraint Propagation: Inference in CSPs

k-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.
- Now suppose we have a CSP with *n* nodes and make it strongly *n*-consistent. We can then solve the problem as follows:
  - First, we choose a consistent value for X_1 .
  - We are then guaranteed to be able to choose a value for X_2 because the graph is 2-consistent, for X_3 because it is 3-consistent, and so on.
  - For each variable X_i, we need only search through the *d* values in the domain to find a value consistent with $X_1, ..., X_{i-1}$. The total run time is only $O(n^2 d)$.

2. Constraint Propagation: Inference in CSPs

k-Consistency

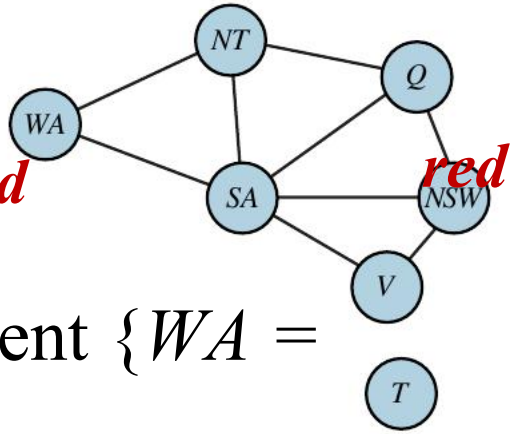
- Constraint satisfaction is NP-complete in general, and any algorithm for establishing *n*-consistency must take time exponential in *n* in the worst case.
- Worse, *n*-consistency also requires space that is exponential in *n*.
- In practice, determining the appropriate level of consistency checking is mostly an empirical science.
  - Computing 2-consistency is common, and 3-consistency less common.

2. Constraint Propagation: Inference in CSPs
Global Constraints

- A **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables).
- Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far.
  - For example, the *Alldiff* constraint says that all the variables involved must have distinct values. 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.

2. Constraint Propagation: Inference in CSPs

Global Constraints



- This method can detect the inconsistency in the assignment $\{WA = red, NSW = red\}$.
  - 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.

2. Constraint Propagation: Inference in CSPs

Sudoku

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 5.4 (a) A Sudoku puzzle and (b) its solution.

- A Sudoku puzzle can be considered a CSP with 81 variables, one for each square.
- We use the variable names *A*1 through *A*9 for the top row (left to right), down to *I*1 through *I*9 for the bottom row.
- The empty squares have the domain {1, 2, 3, 4, 5, 6, 7, 8, 9} and the pre-filled squares have a domain consisting of a single value.
- In addition, there are 27 different *Alldiff* constraints, one for each unit (row, column, and box of 9 squares):
  - *Alldiff*(A1, A2, A3, A4, A5, A6, A7, A8, A9)
    Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)
    …
    Alldiff(A1, B1,C1, D1, E1, F1, G1, H1, I1)
    Alldiff(A2, B2,C2, D2, E2, F2, G2, H2, I2)
    …|
    Alldiff(A1, A2, A3, B1, B2, B3,C1,C2,C3)
    Alldiff(A4, A5, A6, B4, B5, B6,C4,C5,C6)
    …

Assume that the *Alldiff* constraints have been expanded into binary constraints (such as A1 ≠ A2) so that we can apply the AC-3 algorithm directly.

3. Backtracking Search for CSPs

- Sometimes we can finish the constraint propagation process and still have variables with multiple possible values. In that case we have to **search** for a solution.
- Backtracking search algorithms: work on partial assignments.
- Local search algorithms: over complete assignments.

3. Backtracking Search for CSPs

- Consider how a standard depth-limited search could solve CSPs.
  - **A state would be a partial assignment, and an action would extend the assignment.**
- For a CSP with n variables of domain size d we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n.
- The branching factor at the top level would be nd because any of d values can be assigned to any of n variables.
- At the next level, the branching factor is $(n-1)d$, and so on for n levels. So the tree has $n!d^n$ leaves, even though there are only d^n possible complete assignments!

3. Backtracking Search for CSPs

- A crucial property of CSPs: **commutativity**.
  - A problem is commutative if the order of application of any given set of actions does not matter.
  - In CSPs, it makes no difference if we first assign *NSW* = *red* and then *SA* = *blue*, or the other way around. Therefore, we need only consider a single variable at each node in the search tree.
  - With this restriction, the number of leaves is d^n. At each level of the tree we do have to choose which variable we will deal with, but we never have to backtrack over that choice.

3. Backtracking Search for CSPs

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

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

- **Figure 5.5** A simple backtracking algorithm for constraint satisfaction problems.
- The algorithm is modeled on the recursive depth-first search of Chapter 3.
- If a value choice leads to failure, then value assignments are retracted and a new value is tried.
- It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to extend each one into a solution via a recursive call.

3. Backtracking Search for CSPs

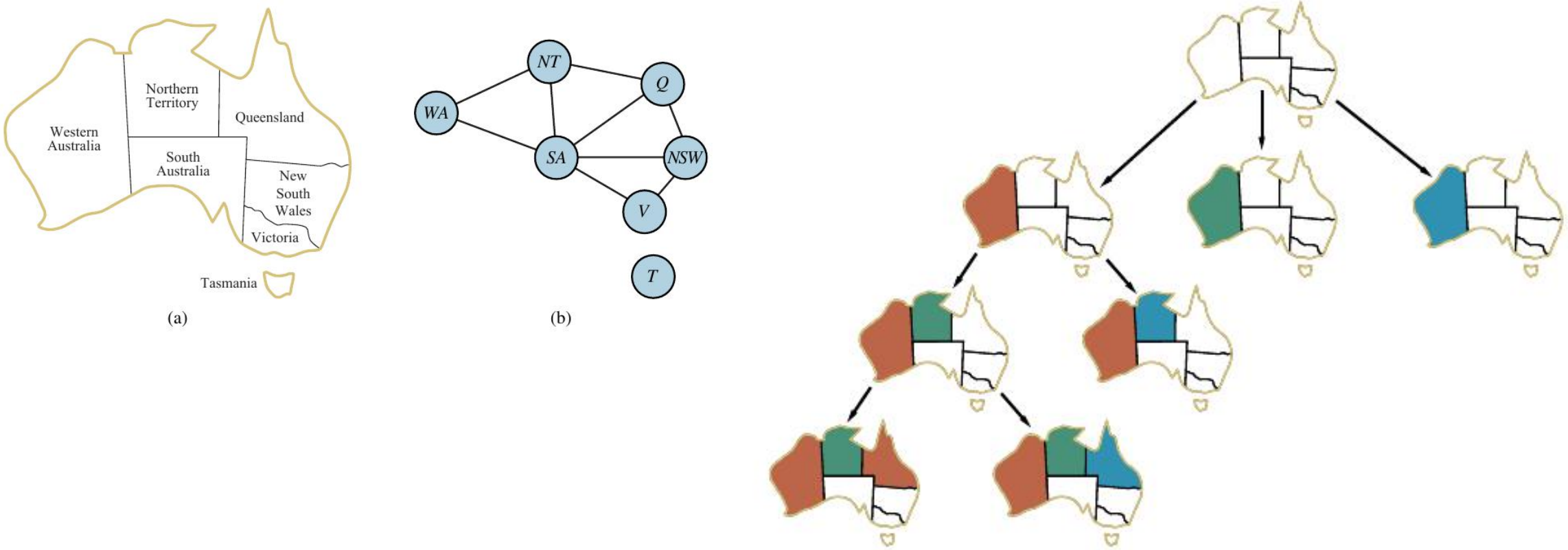
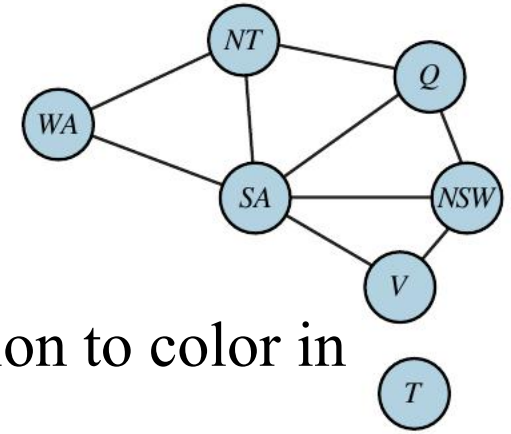


Figure 5.6 Part of the search tree for the map-coloring problem in Figure 5.1.

3. Backtracking Search for CSPs
Variable and Value Ordering

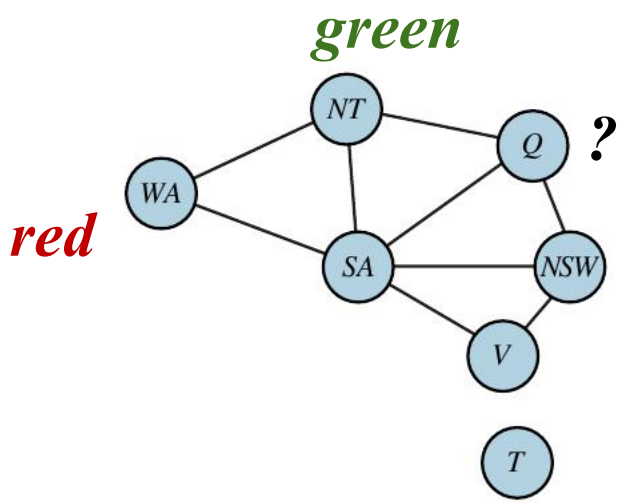
- ***Which variable should be assigned next ?***
  var ← SELECT-UNASSIGNED-VARIABLE(*csp, assignment*)
  - Static ordering: choose the variables in order, $\{X_1, X_2, ...\}$
  - Random ordering
  - **Minimum-Remaining-Values** (MRV) heuristic ("**most constrained variable**" or **"fail-first"** heuristic):
    - Choose the variable with the fewest "legal" values.
    - Picks a variable that is most likely to cause a failure soon. 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.
    - The MRV heuristic usually performs better than a random or static ordering, sometimes by orders of magnitude, although the results vary depending on the problem.

3. Backtracking Search for CSPs
Variable and Value Ordering



- 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.
  - *SA* is the variable with highest degree, 5; the other variables have degree 2 or 3, except for *T*, which has degree 0.

3. Backtracking Search for CSPs
Variable and Value Ordering



- ***In what order should its values be tried?***
  ORDER-DOMAIN-VALUES(*csp, var, assignment*)
- Once a variable has been selected, the algorithm must decide on the order in which to examine its values.
- **Least-Constraining-Value** heuristic: (**"fail-last"**)
  - It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.
  - For example, suppose that 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.

3. Backtracking Search for CSPs

Variable and Value Ordering

- Why should variable selection be fail-first, but value selection be fail-last?
  - Every variable has to be assigned eventually, so by choosing the ones that are likely to fail first, we will on average have fewer successful assignments to backtrack over.
  - 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.

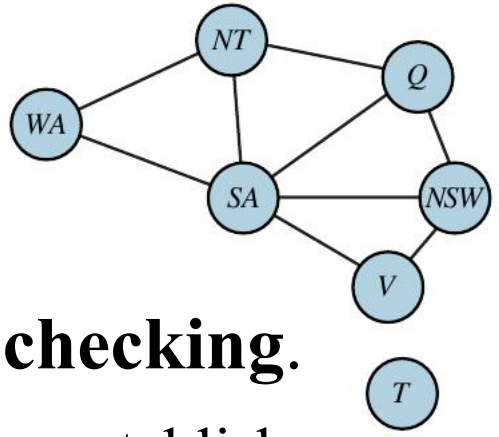
3. Backtracking Search for CSPs

Interleaving Search and Inference

- What inferences should be performed at each step in the search ? INFERENCE(*csp, var, assignment*)
  - 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*.

3. Backtracking Search for CSPs

Interleaving Search and Inference



- 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.

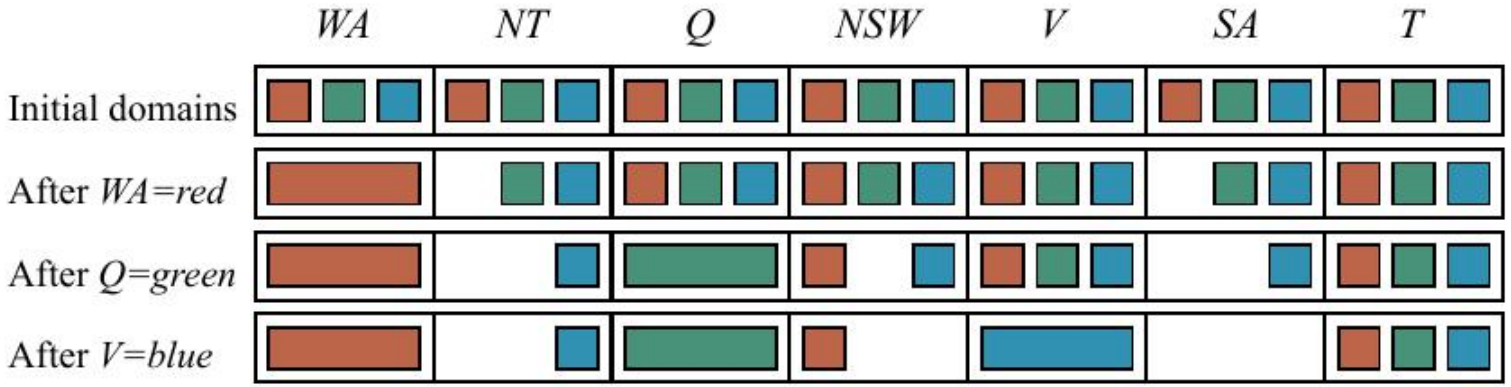
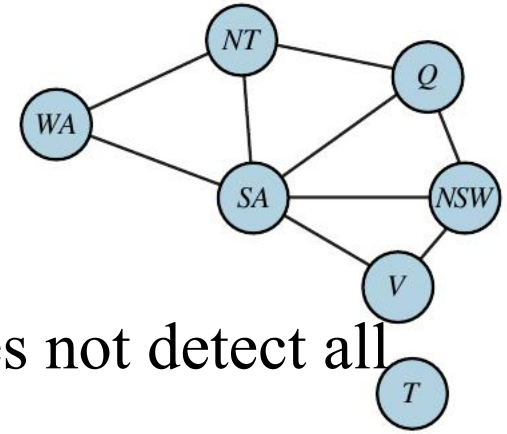


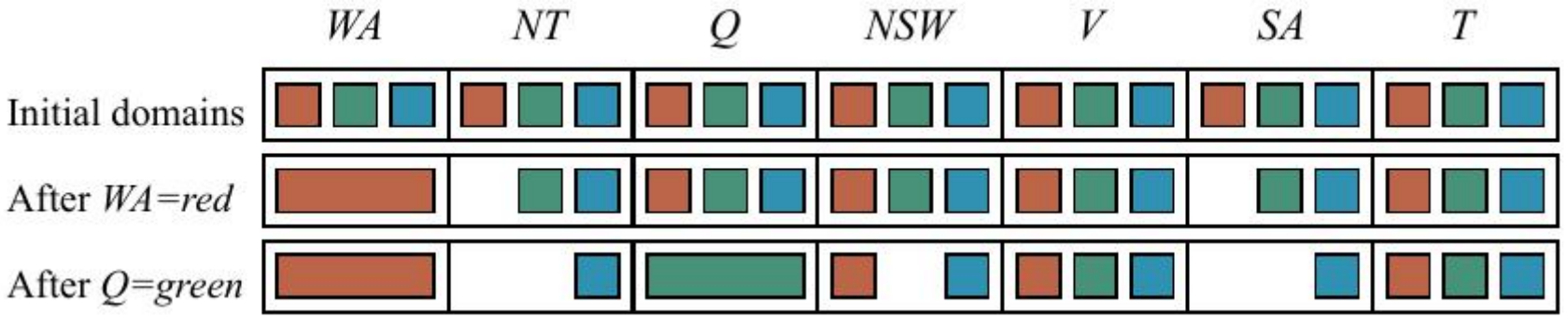
Figure 5.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After $Q = green$ is assigned, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After $V = blue$ is assigned, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

3. Backtracking Search for CSPs

Interleaving Search and Inference



- Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it doesn't look ahead far enough.
  - For example, consider the *Q* = *green* row. We've left both *NT* and *SA* with *blue* as their only possible value, which is an inconsistency, since they are neighbors.



3. Backtracking Search for CSPs

Interleaving Search and Inference

- The algorithm called MAC (for **Maintaining Arc Consistency**) detects inconsistencies like this.
  - After a variable X_i 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 (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i.
  - 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.

4. Local Search for CSPs

- Local search algorithms turn out to be very effective in solving many CSPs.
  - They use a complete-state formulation where each state assigns a value to every variable, and the search changes the value of one variable at a time.

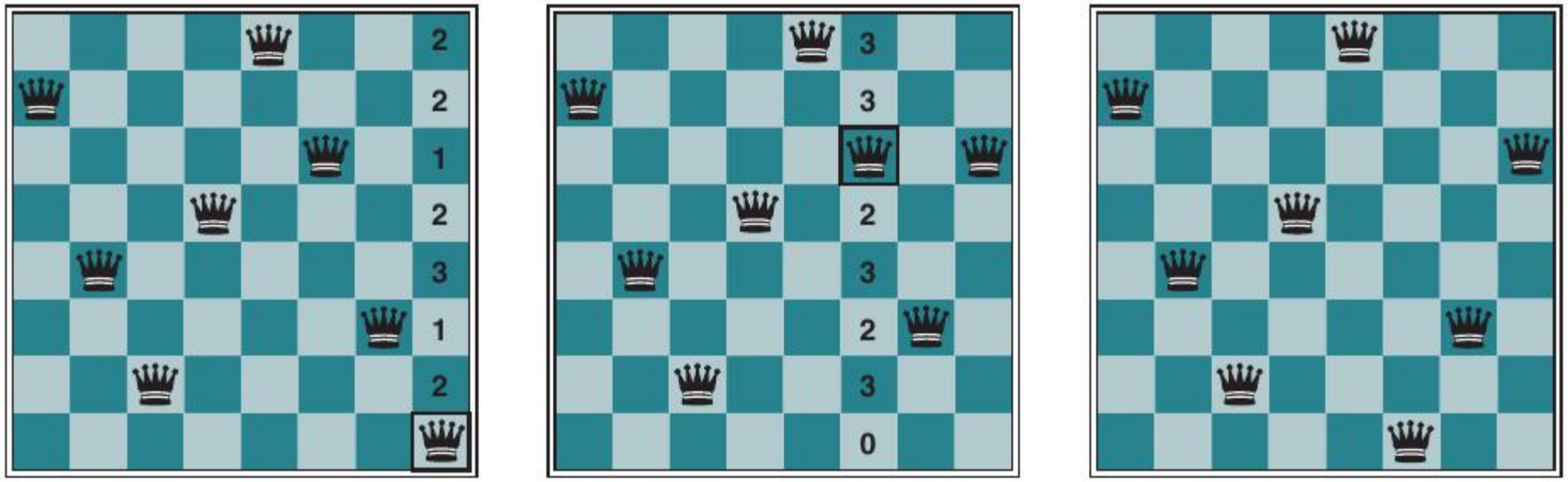


Figure 5.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

We start on the left with a complete assignment to the 8 variables; typically this will violate several constraints. We then randomly choose a conflicted variable, which turns out to be Q8, the rightmost column. We'd like to change the value to something that brings us closer to a solution; the most obvious approach is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic.

4. Local Search for CSPs

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
    inputs: csp, a constraint satisfaction problem
            max_steps, the number of steps allowed before giving up

    current ← an initial complete assignment for csp
    for i = 1 to max_steps do
        if current is a solution for csp then return current
        var ← a randomly chosen conflicted variable from csp.VARIABLES
        value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
        set var = value in current
    return failure
```

Figure 5.9 The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.