北京大學

约束满足问题
Constraint Satisfaction Problems

·AIPKU·

内容来自：Artificial Intelligence A Modern Approach
Stuart Russel Peter Norvig Chapter 6
人工智能引论第七课
主讲人：李文新

内容提要

1. 定义约束满足问题
   - 地图着色问题、作业调度问题、CSP的形式化
2. 约束传播：CSP中的推理
   - 结点相容、弧相容、路径相容、k-相容、全局约束、数独
3. CSP的回溯搜索
   - 变量和取值顺序、搜索与推理交错进行、智能回溯：向后看
4. CSP局部搜索

1. 定义约束满足问题

- 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.

定义约束满足问题

- **A relation** can be represented as an explicit **list of all** tuples of values that satisfy the constraint, or as **an abstract relation**

- 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, complete assignment.

- A **partial assignment** is one that assigns values to only some of the variables.

Example problem: Map coloring

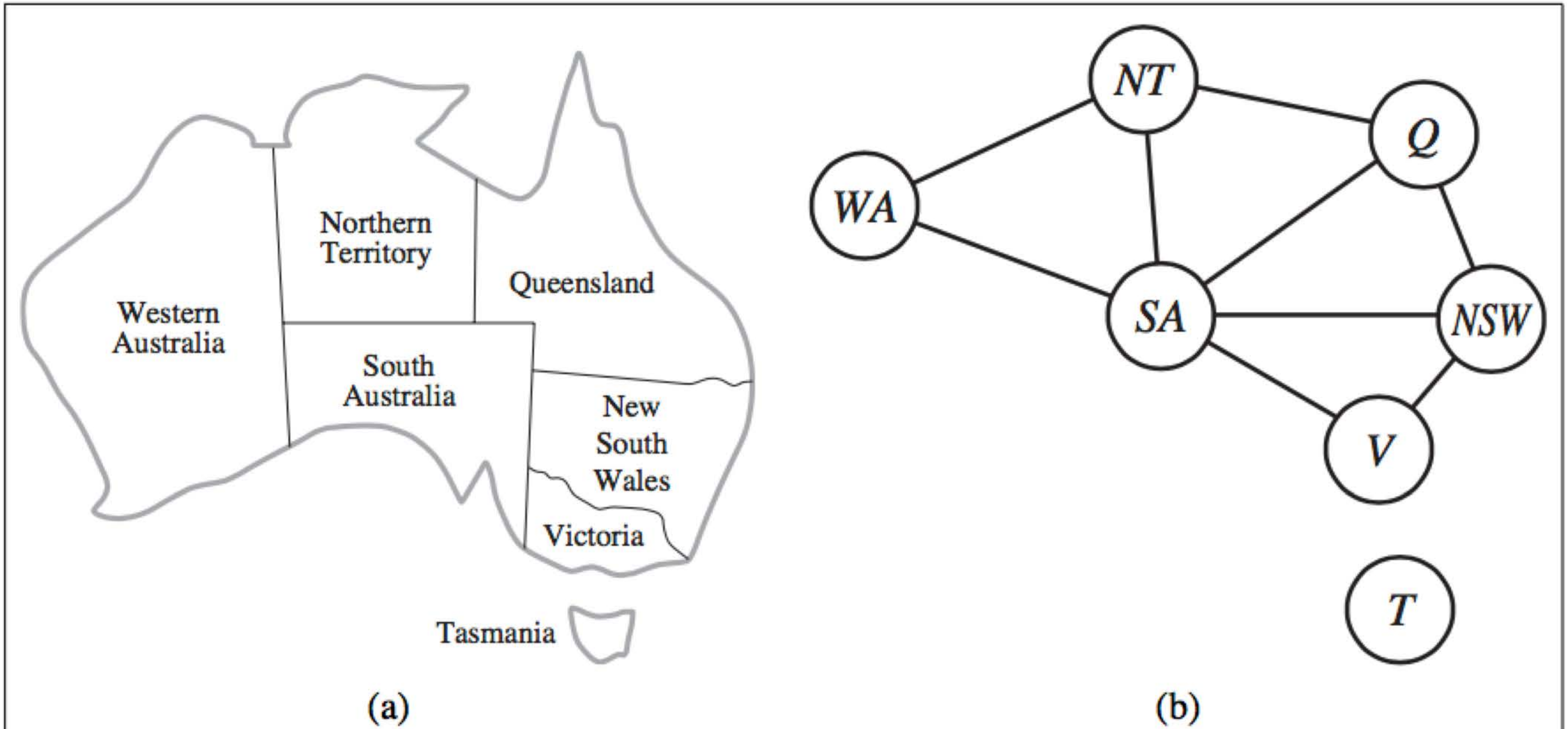


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Example problem: Map coloring

- Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories (Figure 6.1(a)). We are given the task of coloring 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

- $\boldsymbol{X} = \{\boldsymbol{WA, NT, Q, NSW, V, SA, T}\}$.

- The domain of each variable is the set $\boldsymbol{D_i} = \{$**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

- **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. For example, once we have chosen {***SA* = blue**} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue. Without taking advantage of constraint propagation, a search procedure would have to consider $\mathbf{3^5 = 243}$ assignments for the five neighboring variables; with constraint propagation we never have to consider blue as a value, so we have only $\mathbf{2^5 = 32}$ assignments to look at, a reduction of 87%.

Example problem: Map coloring

- In regular state-space search we can only ask: is this specific state a goal? No? What about this one?

- With CSPs, once we find out that **a partial assignment is not** a solution, we can **immediately discard** further refinements of the partial assignment.

- Furthermore, we can see ***why*** the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter.

- As a result, many problems that are intractable for 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轮毂罩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

- 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:
- ***X*** = {*AxleF,AxleB,WheelRF,WheelLF,WheelRB,WheelLB,NutsRF, 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 we represent **precedence constraints** between individual tasks. Whenever a task $\boldsymbol{T_1}$ must occur before task $\boldsymbol{T_2}$, and task $\boldsymbol{T_1}$ takes duration $\boldsymbol{d_1}$ to complete, we add an arithmetic constraint of the form $\boldsymbol{T_1 + d_1 \leq T_2}$.

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
  - **AxleF + 10 ≤ WheelRF ; AxleF + 10 ≤ WheelLF ;**
  - **AxleB + 10 ≤ WheelRB; AxleB + 10 ≤ WheelLB .**
- 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):
  - **WheelRF +1≤NutsRF; WheelLF +1≤NutsLF;**
  - **WheelRB + 1 ≤ NutsRB; WheelLB + 1 ≤ NutsLB;**
  - **NutsRF +2≤CapRF; NutsLF +2≤CapLF;**
  - **NutsRB + 2 ≤ CapRB; NutsLB + 2 ≤ CapLB .**

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 and Axle must not overlap in time; either one comes first or the other does:

- (**AxleF +10≤AxleB**) **or** (**AxleB +10≤AxleF**).

- This looks like a more complicated constraint, combining **arithmetic** and **logic**. But it still reduces to a set of pairs of values that **AxleF** and **AxleF** can take on.

Example problem: Job-shop scheduling

- 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:

- $D_i = \{1,2,3,\ldots,27\}$.

- This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with **thousands of variables**. In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques are used, as discussed **in Chapter 11**.

types of variables

- The simplest kind of CSP involves variables that have **discrete**, **finite domains**. Map-coloring problems and scheduling with time limits are both of this kind. The 8-queens problem described in Chapter 3 can also be viewed as a finite-domain CSP, where the variables **Q1,...,Q8** are the positions of each queen in columns **1,...,8** and each variable has the domain $\boldsymbol{D_i}$ = {**1,2,3,4,5,6,7,8**}.

- A discrete domain can be **infinite**, such as the set of integers or strings

- Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research.

- **For example,** the scheduling of experiments on **the Hubble Space Telescope r**equires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical, precedence, and power constraints.

types of constraints

- In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint (一元)** , 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; we can express that with the unary constraint ⟨(SA),SA /= green⟩

- A **binary constraint** relates two variables. For example, SA /= NSW is a binary constraint. A binary CSP is one with only binary constraints; it can be represented as a constraint graph, as in Figure 6.1(b).

- We can also describe higher-order constraints, such as asserting that the value of Y is between X and Z, with the **ternary 三元 constraint Between(X, Y, Z)**.

global constraint - Alldiff

- A constraint involving an arbitrary number of variables is called a **global constraint**. (The name is traditional but confusing because it 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 (see Section 6.2.6), all variables in a row or column must satisfy an **Alldiff** constraint.

preference constraints

- The constraints we have described so far have all been **absolute** constraints, violation of which rules out a potential solution.
- Many real-world CSPs include **preference constraints** indicating which solutions are **preferred**.
- For example, Prof. R might **prefer** teaching in the morning. A schedule that has Prof. R teaching at 2 p.m. would still be an **allowable** but would **not** be an **optimal** one.
- **Preference constraints** can often be encoded as **costs** on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1.
- With this formulation, CSPs with preferences can be solved with optimization search methods, either **path-based** or **local**. We call such a problem a **constraint optimization problem**, or **COP**. Linear programming problems do this kind of optimization.

2. Constraint Propagation： Inference in CSPs

- **约束传播：CSP中的推理**

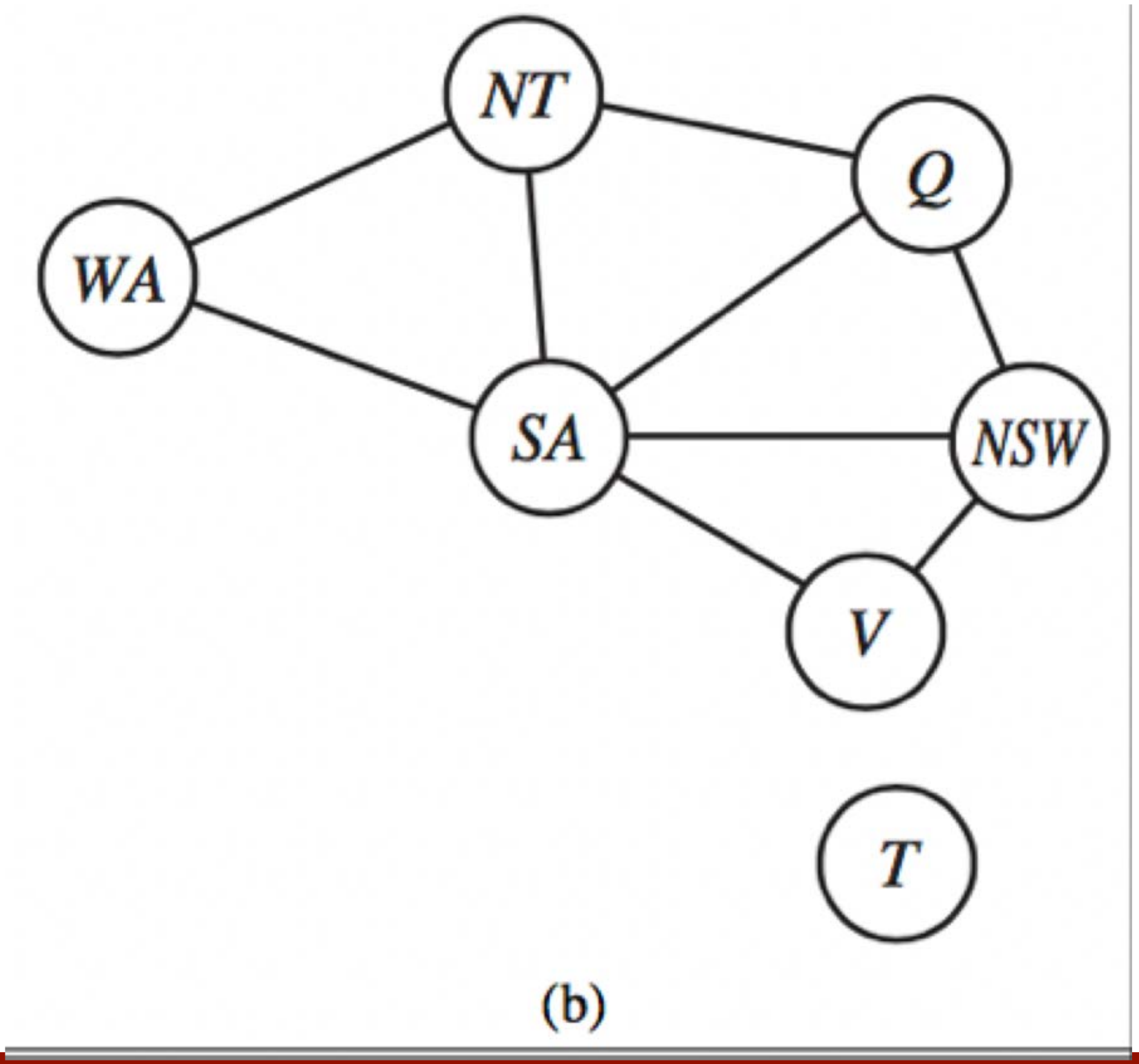
- In regular state-space search, an algorithm can do only one thing: **search**.

- In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or 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. 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.

Constraint Propagation： Inference in CSPs

- The key idea is **local consistency**. If we treat each variable as a node in a graph (see Figure 6.1(b)) 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, which we now cover in turn.**



(b)

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 the variant of the Australia **map-coloring** problem (Figure 6.1) 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 network is node-consistent if every variable in the network is node-consistent.**

- It is always possible to eliminate all the unary constraints in a CSP by running node consistency.

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 network 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 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}. If we also make Y arc-consistent with respect to X , then Y ' s domain becomes {0, 1, 4, 9} and the **whole CSP is arc-consistent.**

```
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
```

Figure 6.3 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's the third version developed in the paper.

Arc consistency

- **The complexity of AC-3** can be analyzed as follows. Assume a CSP with **n** variables, each with domain size at most **d**, and with **c** binary constraints (arcs). Each arc ($\mathbf{X_k}$,$\mathbf{X_i}$) can be inserted in the queue only **d** times because $\mathbf{X_i}$ has at most **d** values to delete. Checking consistency of an arc can be done in O($\mathbf{d^2}$) time, so we get O($\mathbf{cd^3}$) total **worst-case time.**

- On the other hand, arc consistency can do nothing for the Australia map-coloring problem. Consider the following inequality constraint on (**SA, WA**):

- **{(red , green), (red , blue), (green , red), (green , blue), (blue , red), (blue , green)}** .

- No matter what value you choose for SA (or for WA), there is a valid value for the other variable. So applying arc consistency has no effect on the domains of either variable.

Path consistency

- Arc consistency can go a long way toward reducing the domains of variables, sometimes **finding a solution** (by reducing every domain to size 1) and sometimes finding that the CSP **cannot be solved** (by reducing some domain to size 0).

- But for other networks, arc consistency fails to make enough inferences. Consider the map-coloring problem on Australia, but with only two colors allowed, red and blue. Arc consistency can do nothing because every variable is already arc consistent: each can be red with blue at the other end of the arc (or vice versa).

- 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.

Path consistency

- 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.

- A two-variable set {**Xi,Xj**} is path-consistent with respect to a third variable **Xm** if, 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.**

Path consistency

- Let’ s see how path consistency fares in coloring the Australia map with two colors. We will make the set {**WA, SA**} path consistent with respect to **NT** .
- We start by enumerating the consistent assignments to the set. In this case, there are only two: {**WA = red, SA = blue**} and {**WA = blue, SA = red**}.
- We can see that with both of these assignments **NT** can be neither red nor blue (because it would conflict with either **WA** or **SA**). Because there is no valid choice for **NT** , we eliminate both assignments, and we **end up with no valid assignments** for {**WA, SA**}. Therefore, we know that there can be no solution to this problem.
- The **PC-2** algorithm (Mackworth, 1977) achieves path consistency in much the same way that **AC-3** achieves **arc consistency.**

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 **kth** 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 networks,
- **3-consistency** is the same as **path consistency**.

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** (i.e., strongly k-consistent for k = n). We can then solve the problem as follows: First, we choose a consistent value for **X1**. We are then guaranteed to be able to choose a value for **X2** because the graph is 2-consistent, for **X3** because it is 3-consistent, and so on. For each variable **Xi**, we need only search through the **d** values in the domain to find a value consistent with **X1 , . . . , Xi−1** . We are guaranteed to find a solution in time **O(n^2d).**

- Of course, there is no free lunch: 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**. The **memory** issue is even **more severe** than the **time**. In practice, determining the appropriate level of consistency checking is mostly an empirical science. It can be said practitioners **commonly** compute **2-consistency** and less commonly 3-consistency.

Global constraints

- Remember that 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 (as in the cryptarithmetic problem above and Sudoku puzzles be- low).
- 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**.

Global constraints - Alldiff

- 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.

Global constraints - Alldiff

- This method can detect the inconsistency in the assignment {**WA = red , NSW = red** } for Figure 6.1. 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 domain of each variable is reduced to {**green, blue**}. That is, we have three variables and only two colors, so the **Alldiff** constraint is violated.
- **Thus, a simple consistency procedure for a higher-order constraint is sometimes more effective than applying arc consistency to an equivalent set of binary constraints.**

Global constraints - Atmost

- Another important higher-order constraint is the **resource constraint**, sometimes called the **atmost** constraint.

- For example, in a scheduling problem, let **P1, . . . , P4** denote the numbers of personnel assigned to each of four tasks. The constraint that no more than **10** personnel are assigned in total is written as **Atmost(10,P1,P2,P3,P4)**. We can detect an inconsistency simply by checking the sum of the minimum values of the current domains;

- for example, if each variable has the domain {**3,4,5,6**}, the Atmost constraint cannot be satisfied. We can also enforce consistency by deleting the maximum value of any domain if it is not consistent with the minimum values of the other domains. Thus, if each variable in our example has the domain {**2, 3, 4, 5, 6**}, the values **5** and **6** can be deleted from each domain.

Global constraints - bounds propagation

- For **large resource-limited problems** with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually **not possible** to represent the domain of each variable **as a large set of integers** and gradually reduce that set by consistency-checking methods.

- Instead, domains are represented by **upper and lower bounds** and are managed by **bounds propagation**.

- For example, in an airline-scheduling problem, let' s suppose there are two flights, **F1** and **F2**, for which the planes have capacities **165** and **385**, respectively. The initial domains for the numbers of passengers on each flight are then

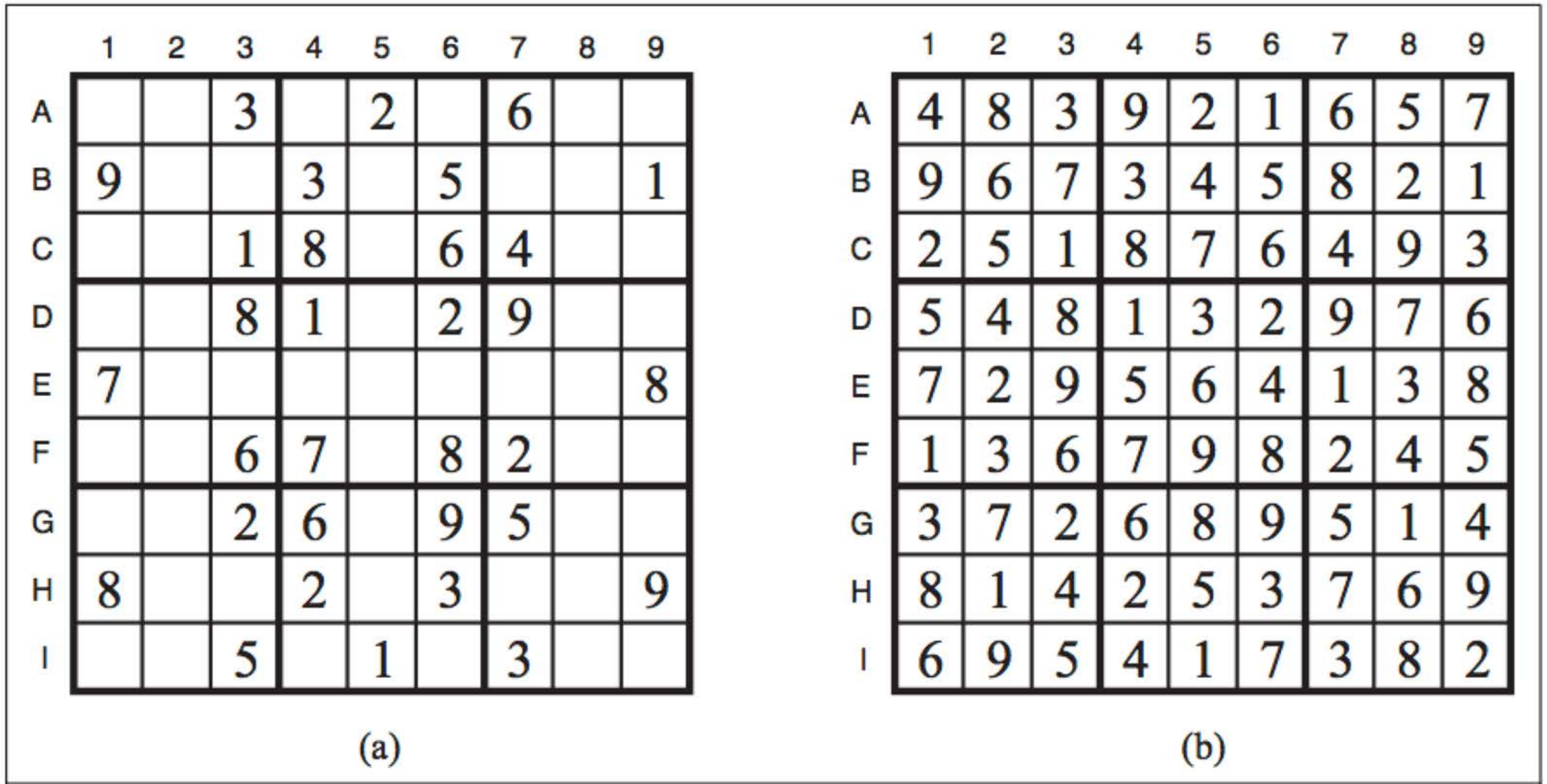
- **D1 = [0,165]** and **D2 = [0,385]** .

Global constraints - bounds consistent

- Now suppose we have the additional constraint that the two flights together must carry 420
- people: **F1 + F2 = 420**. Propagating bounds constraints, we reduce the domains to **D1 = [35, 165] and D2 = [255, 385]** .
- We say that a CSP is **bounds consistent** if for every variable **X**, and for both the lower- bound and upper-bound values of **X**, there exists some value of **Y** that satisfies the constraint between **X** and **Y** for every variable **Y** . This kind of bounds propagation is widely used in practical constraint problems.

Sudoku example

- The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not recognize it. A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3 × 3 box (see Figure 6.4). A row, column, or box is called a **unit**.



	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 6.4 (a) A Sudoku puzzle and (b) its solution.

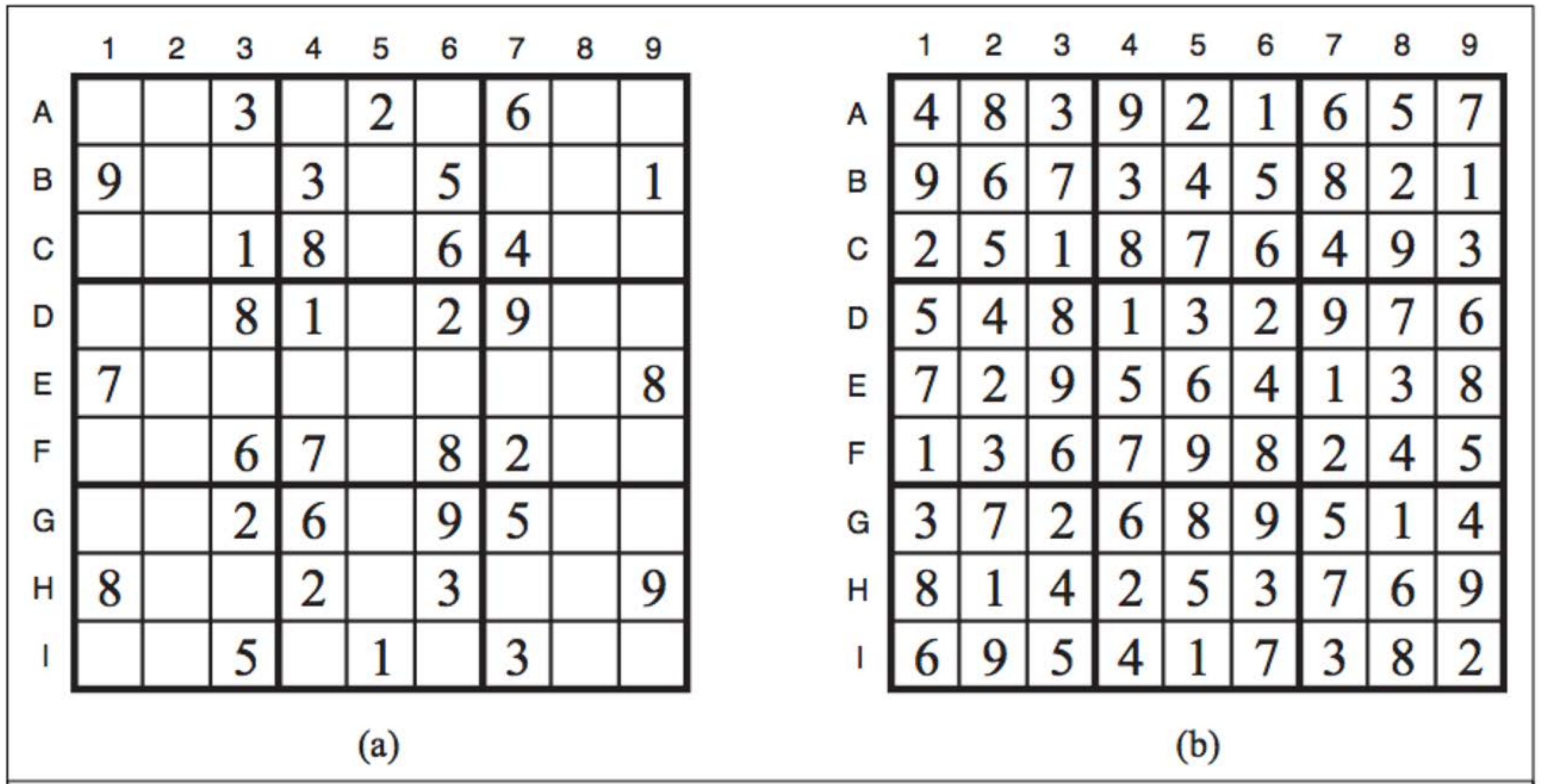
Sudoku example

- The Sudoku puzzles that are printed in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, even the hardest Sudoku problems yield to **a CSP solver** in less than 0.1 second.
- A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names **A1** through **A9** for the top row (left to right), down to **I1** through **I9** for the bottom row.
- The empty squares have the domain {**1, 2, 3, 4, 5, 6, 7, 8, 9**} and the prefilled squares have a domain consisting of a single value.
- In addition, there are 27 different **Alldiff** constraints: one for each row, column, and box of 9 squares.

Sudoku example

- 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)
- …

Sudo



	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

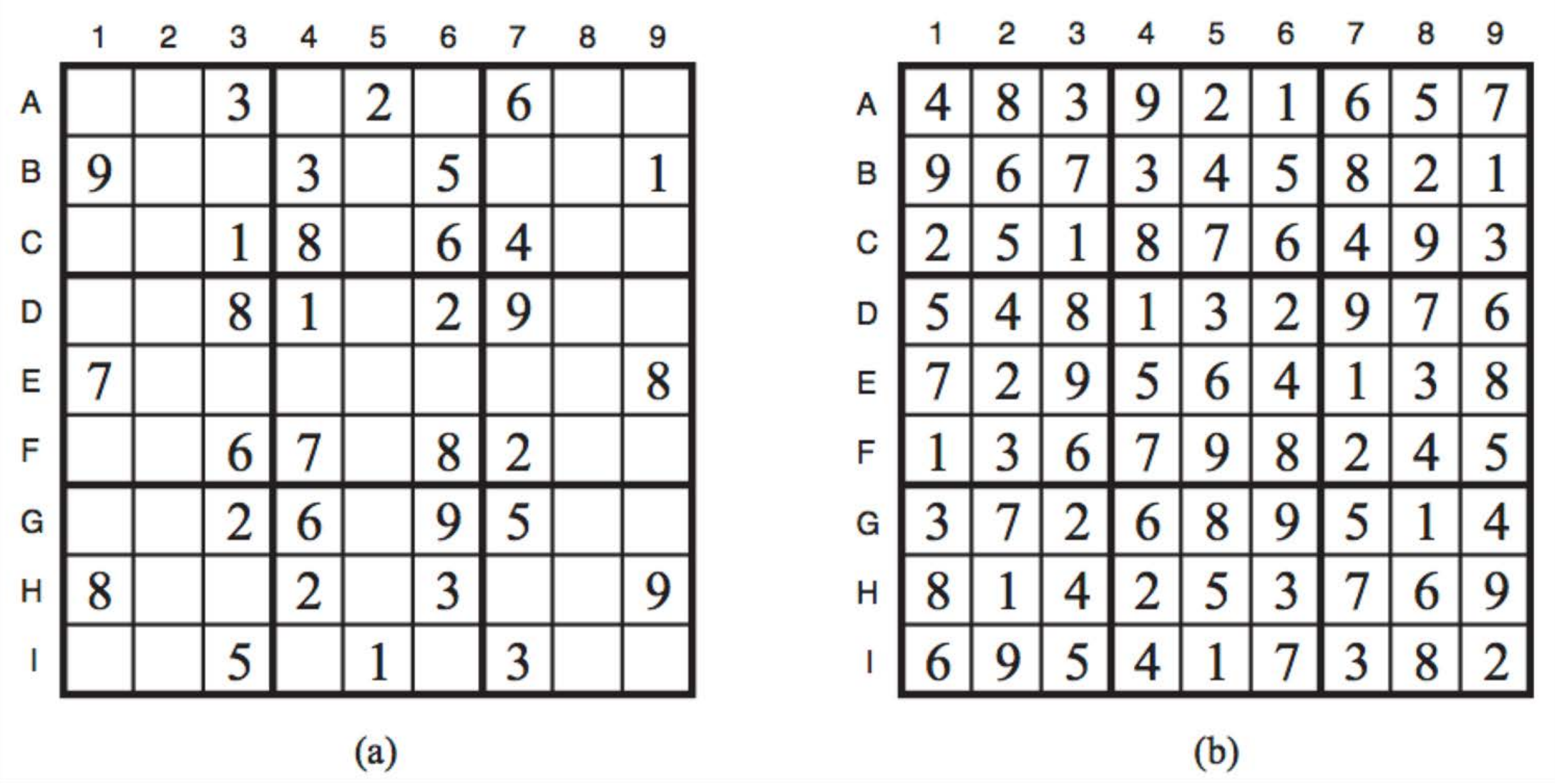
(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

- Consider variable **E6** from Figure 6.4(a)—the empty square between the 2 and the 8 in the middle box. From the constraints in the box, we can remove not only **2** and **8** but also **1** and **7** from E6’ s domain. From the constraints in its column, we can eliminate **5, 6, 2, 8, 9,** and **3**. That leaves E6 with a domain of {**4**}; in other words, we know the answer for E6 .
- Now consider variable **I6** —the square in the bottom middle box surrounded by 1, 3, and 3. Applying arc consistency in its column, we eliminate **5, 6, 2, 4** (since we now know E6 must be 4), **8, 9**, and **3**. We eliminate **1** by arc consistency with I5 , and we are left with only the value **7** in the domain of I6 .

Sudoku example

- Now there are 8 known values in column 6, so arc consistency can infer that **A6** must be **1**. Inference continues along these lines, and eventually, AC-3 can solve the entire puzzle—all the variables have their domains reduced to a single value, as shown in Figure 6.4(b).



(a)

	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		

(b)

	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

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

AC-3 works only for the easiest Sudoku

- Of course, Sudoku would soon lose its appeal if every puzzle could be solved by a mechanical application of AC-3, and indeed **AC-3 works only for the easiest Sudoku puzzles**. Slightly harder ones can be solved by **PC-2**, but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle.

- To solve the hardest puzzles and to make efficient progress, we will **have to be more clever.**

the power of the CSP formalism

- It is interesting to note how far we can go **without saying much that is specific to Sudoku**.

- We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 Alldiff constraints.

- But beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of Alldiff constraints and **has nothing to do with Sudoku**.

- This is the power of the **CSP formalism**: for each new problem area, we only need to define the problem in terms of constraints; then the **general constraint-solving mechanisms can take over.**

3. Backtracking Search for CSPs

- **Sudoku** problems are designed to be solved by inference over constraints. But many other CSPs cannot be solved by inference alone; there comes a time when we must search for a solution.
    - In this section we look at **backtracking search algorithms** that work on **partial assignments**;
    - in the next section we look **at local search algorithms over complete assignmen**ts.
- We could apply a standard **depth-limited search** (from Chapter 3).
- A state would be a partial assignment, and an action would be adding **var = value** to the assignment.

Backtracking Search for CSPs

- A problem is **commutative 可交换的** 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 **d^n**

```
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 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

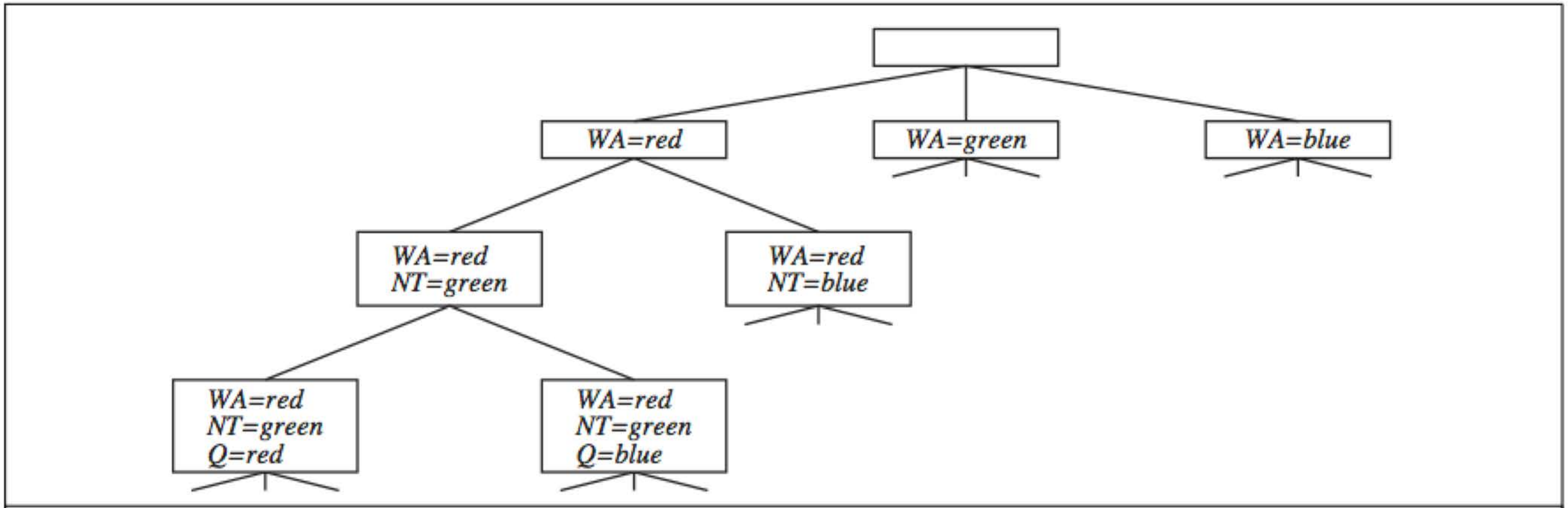


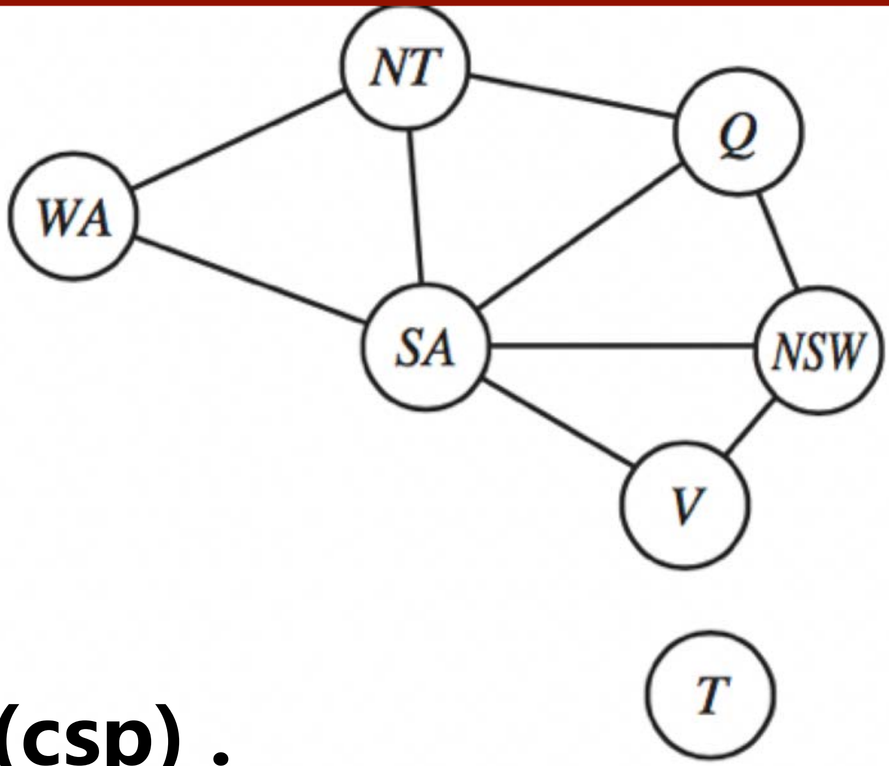
Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

- 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 algorithm is shown in Figure 6.5. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA, NT , Q, ...

Backtracking Search for CSPs

- Notice that BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones, as described on page 87.
- In Chapter 3 we improved the poor performance of uninformed search algorithms by supplying them **with domain-specific heuristic** functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently ***without*** **such domain-specific** knowledge. Instead, we can add some sophistication to the unspecified functions in Figure 6.5, 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**?

Variable and value ordering



- The backtracking algorithm contains the line

  var ← SELECT-UNASSIGNED-VARIABLE(csp) .

- The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, {**X1, X2, . . .**}. This static variable ordering **seldom** results in the most **efficient** search.
- For example, after the assignments for **WA** = **red** and **NT** = **green** in Figure 6.6, 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

Variable ordering - minimum-remaining-values (MRV)

- 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.
- The MRV heuristic **usually performs better** than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

Variable ordering - degree 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. 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.

- In Figure 6.1, **SA** is the variable with highest degree, **5**; the other variables have degree **2** or **3**, except for **T** , which has degree **0**. In fact, once **SA** is chosen, applying the degree heuristic solves the problem without any false steps—you can choose *any* consistent color at each choice point and still arrive at a solution with no backtracking. **The minimum-remaining- values heuristic is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.**

value 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.

Variable and value ordering

- **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.

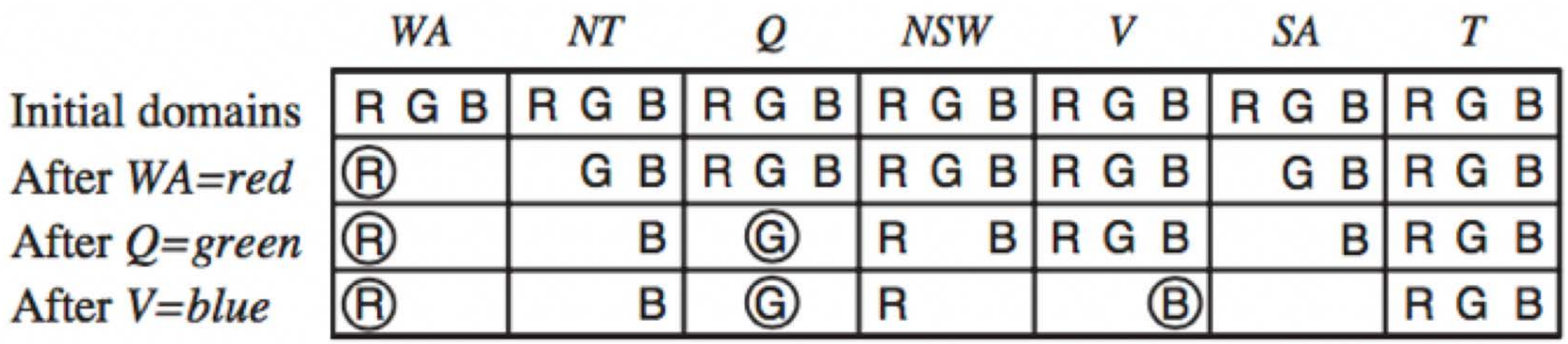
- **If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.**

Interleaving search and inference

- So far we have seen how AC-3 and other algorithms can infer reductions in the domain of variables *before* we begin the search. But inference can be even 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.

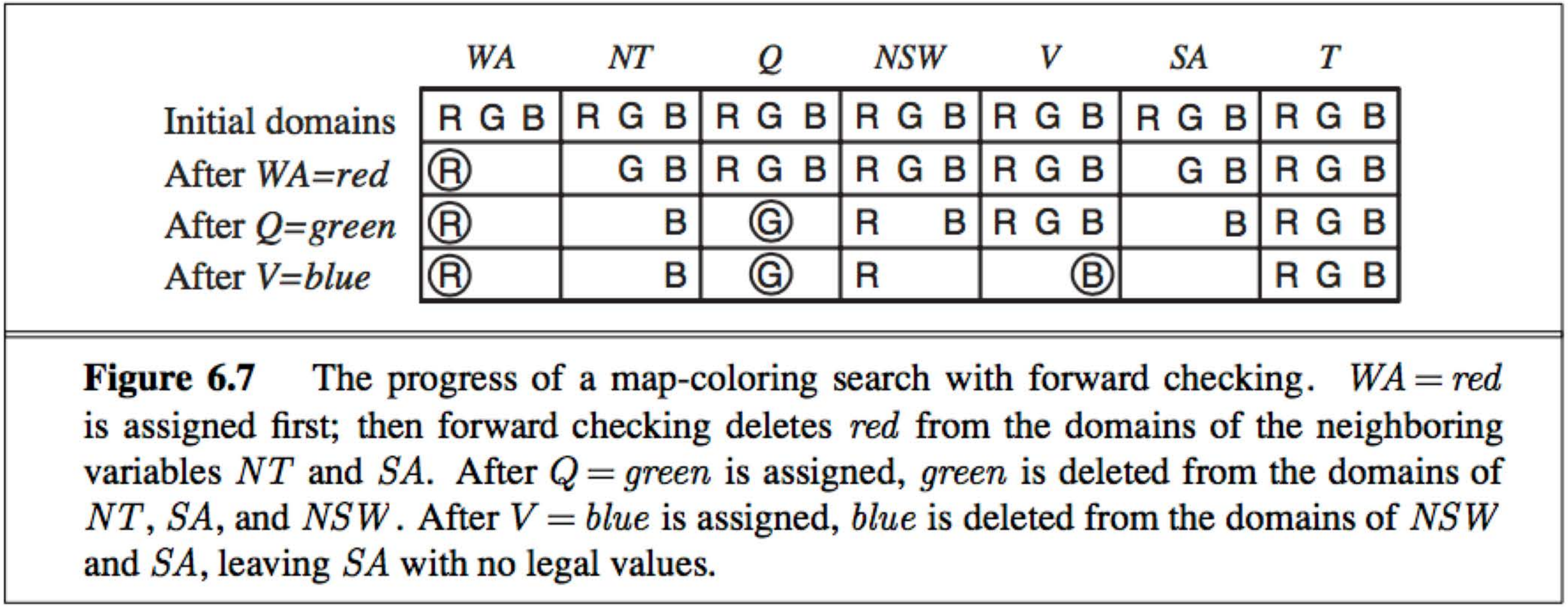
- One of the simplest forms of inference is called **forward checking**. Whenever a variable **X** is assigned, the forward-checking process establishes **arc consistency** for it: for each unassigned variable Y that is connected to **X** by a constraint, delete from **Y**’ s domain any value that is 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



	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*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

- Figure 6.7 shows the progress of backtracking search on the Australia CSP with forward checking.
- There are **two important points** to notice about this example.
- **First**, notice that after **WA = red** and **Q = green** are assigned, the domains of **NT** and **SA** are reduced to a single value;
- we have eliminated branching on these variables altogether by propagating information from **WA** and **Q**.
- A **second** point to notice is that after **V =blue**, the domain of **SA** is empty. Hence, forward checking has detected that the partial assignment {**WA=red, Q=green,V =blue**} is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.



	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*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		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.

- For many problems the search will be more effective if we combine the **MRV heuristic with forward checking**. Consider Figure 6.7 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 hap-pens 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.**

doesn’ t look ahead

- Although forward checking detects many inconsistencies, it does not detect all of them. The problem 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 6.7. It shows that when **WA** is 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.

Maintaining Arc Consistency (MAC)

- The algorithm called MAC (for **Maintaining Arc Consistency (MAC)**) detects this inconsistency.

- After a variable **Xi** 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 (**Xj,Xi**) for all **Xj** that are unassigned variables that are neighbors of **Xi.**

- 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.

- We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC’ s queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

4. Local Search for CSPs

- Local search algorithms (see Section 4.1) turn out to be effective in solving many CSPs. They use a complete-state formulation: **the initial state assigns a value to every variable, and the search changes the value of one variable at a time.**

- For example, in the 8-queens problem (see Figure 4.3), the initial state might be a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column. Typically, the initial guess violates several constraints. **The point of local search is to eliminate the violated constraints**

- In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic. The algorithm is shown in Figure 6.8 and its application to an 8-queens problem is diagrammed in Figure 6.9.

Local Search for CSPs

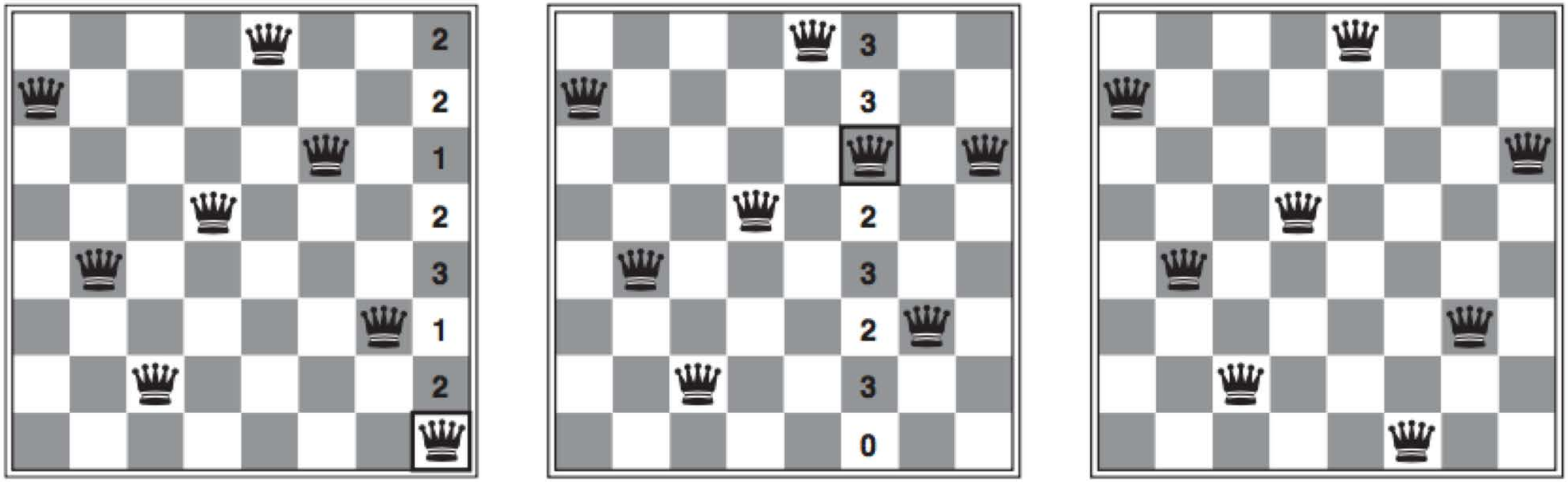
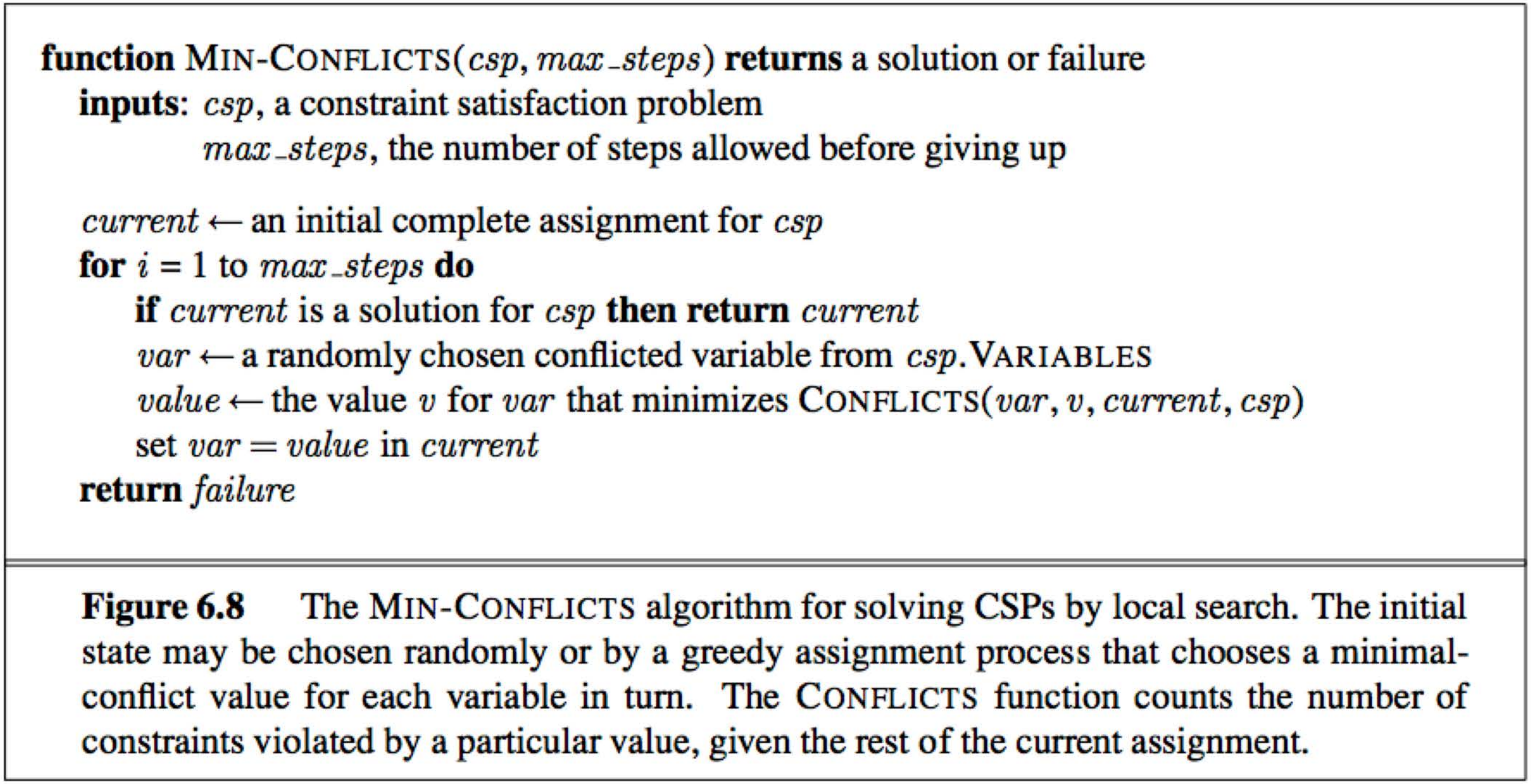


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

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($var$, $v$, $current$, $csp$)
      set $var = value$ in $current$
  return $failure$
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. 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.

Local Search for CSPs

- **Min-conflicts is surprisingly effective for many CSPs.**
- Amazingly, on the **n-queens problem**, if you don’ t count the initial placement of queens, the run time of min-conflicts is roughly ***independent of problem size*.** It solves even the *million*-queens problem in an average of 50 steps (after the initial assignment).
- This remarkable observation was the stimulus leading to a great deal of research in the 1990s on **local search** and the distinction **between easy and hard problems**, which we take up in Chapter 7. Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. **Min-conflicts also works well for hard problems.**
- For example, it has been used to schedule observations for the **Hubble Space Telescope**, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

Local Search for CSPs

- All the local search techniques from Section 4.1 are candidates for application to CSPs, and some of those have proved especially effective.

- The **landscape** of a CSP under the **min-conflicts** heuristic usually has **a series of plateaux.** There may be millions of variable assignments that are only one conflict away from a solution. **Plateau search**—allowing sideways moves to another state with the same score—can help local search find its way off this plateau.

- This wandering on the plateau can be directed with **tabu search**: keeping a small list of recently visited states and **forbidding** the algorithm to return to those states. Simulated annealing can also be used to escape from plateaux.

Local Search for CSPs

- Another **advantage of local search** is that it can be used in an **online** setting when the problem changes.
- This is particularly important in **scheduling problems**.
- A week’ s **airline schedule** may involve **thousands of flights** and **tens of thousands of personnel assignments**, but **bad weather** at one airport can render the schedule infeasible.
- We would like to **repair the schedule with a minimum number of changes.** This can be easily done with a **local search** algorithm starting from the current schedule.
- **A backtracking search** with the new set of constraints usually requires much **more time** and might find a solution with **many changes** from the current schedule.

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 backjumping** backtracks directly to the source of the problem.

SUMMARY

- **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. **Cutset conditioning** can reduce a general CSP to a tree-structured one and is quite efficient if a small cutset 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.