Course: Introduction to AI

Prof. Ahmed Guessoum

The National Higher School of AI

Chapter 6

Constraint Satisfaction Problems

Outline

- Defining Constraint Satisfaction Problems
  - Example problem: Map colouring
  - Example problem: Job-shop scheduling
  - Variations on the CSP formalism
- Constraint Propagation: Inference in CSPs
  - Node consistency
  - Arc consistency
  - Path consistency
  - *K*-consistency
  - Global constraints
  - Sudoku example

Outline

- Backtracking Search for CSPs
  - Variable and value ordering
  - Interleaving search and inference
  - Intelligent backtracking: Looking backward
- Local Search for CSPs
- The Structure of Problems

Constraint Satisfaction Problems

- Chapters 3 and 4: 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.
- The search algorithm “sees” each state as atomic, or indivisible—a black box with no internal structure.
- In this chapter, each state is seen as a set of variables, each of which has a value.
- A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- A problem described this way is called a **Constraint Satisfaction Problem**, or CSP.
- CSP search algorithms use *general-purpose* rather than *problem-specific* heuristics to solve complex problems.

Defining CSPs

- A constraint satisfaction problem consists of three components (sets): X, D, and C:
  - X is a set of variables, $\{X_1, . . . ,X_n\}$.
  - D is a set of domains, $\{D_1, . . . ,D_n\}$, one for each variable.
  - C is a set of constraints that specify allowable combinations of values.
  - Each domain Di consists of a set of allowable values, $\{v_1, . . . ,v_k\}$ 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.

Defining CSPs

- A relation can be represented as:
  - an explicit list of all tuples of values that satisfy the constraint, **or**
  - an abstract relation that supports two operations:
    - testing if a tuple is a member of the relation and
    - enumerating the members of the relation.
- E.g., if X_1 and X_2 both have the domain {A,B}, then the constraint that they must have different values can be written as $\langle (X_1, X_2), [(A,B), (B,A)] \rangle$ or as $\langle (X_1, X_2), X_1 \neq X_2 \rangle$.

Defining CSPs

- To solve a CSP, we need to define a state space and the notion of a solution.
- A CSP state 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.
- 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 colouring

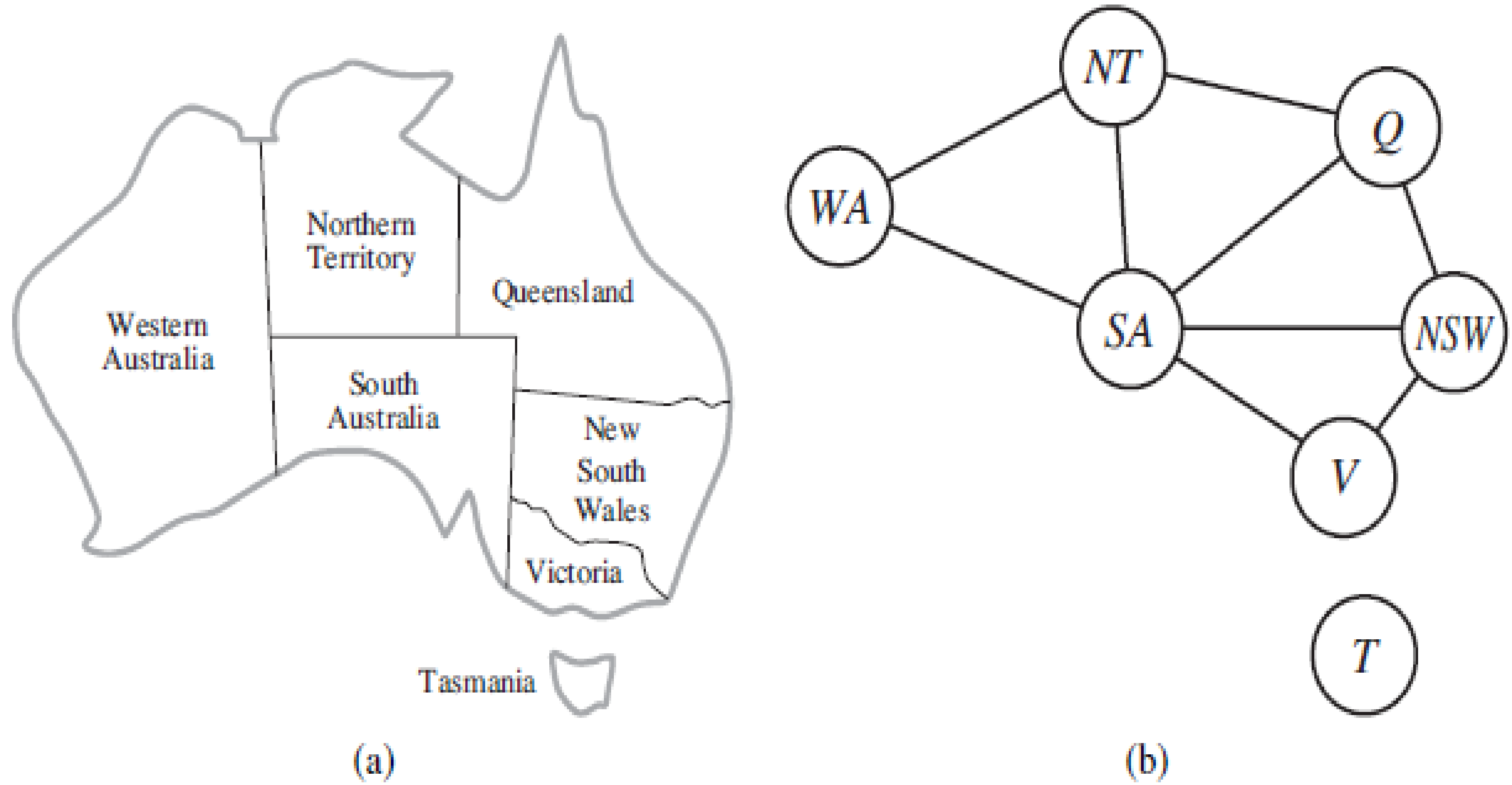


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Map colouring problem

- **Task**: colour each region of the map either *Red, Green*, or *Blue* so that no neighbouring regions have same colour.
- CSP formulation:
  - The variables are the regions $X = \{WA, NT, Q, NSW, V, SA, T\}$.
  - Domain of each variable: is the set $D_i = \{red, green, blue\}$.
  - There are nine constraints: $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$
- N.B.: $SA \neq WA$ can be fully enumerated in turn as {(red, green), (red, blue), (green,red), (green,blue), (blue,red), (blue,green)}

Map colouring problem

- A CSP can be visualized as a **constraint graph** (see previous figure).
  - The graph nodes correspond to problem variables.
  - A link connects any two variables that participate in a constraint.
- Australia map colouring problem: many possible solutions to this, e.g.

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

Example problem: Map colouring

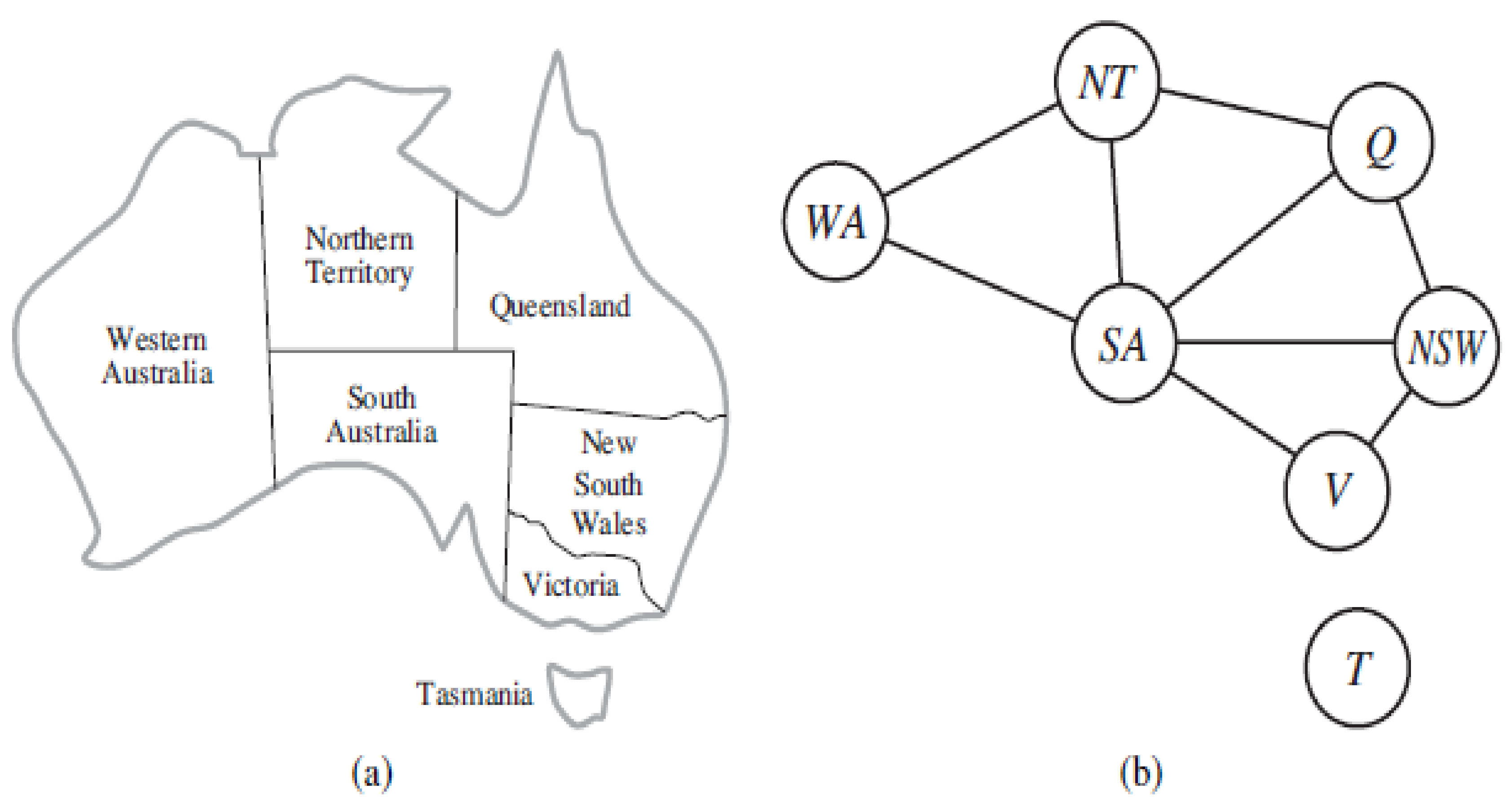


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Problem formulation as a CSP

- Why formulate a problem as a CSP?
  - CSPs have a natural representation for a wide variety of problems.
  - If a CSP-solving system is available, it is often easier to solve a problem using it than design a custom solution using another search technique.
  - CSP solvers can be faster than state-space searchers because they can quickly eliminate large parts of the search space. E.g. if {SA=blue}
    - ➔ none of the five neighbouring variables can take value blue. ➔ for constraint propagation *blue* is never considered.
    - Other search techniques: consider 3^5 =243 assignments for the five neighbouring vars; ➔
  - In CSP only 2^5 =32 assignments; a reduction of 87%.

Problem formulation as a CSP

- In regular state-space search: one can only ask if a specific state is a goal or not.
- With CSPs:
  - once a partial assignment is found not to be a solution, one can
    - immediately <u>discard further refinements</u> of the partial assignment.
    - <u>see *why* the assignment is not a solution</u>—i.e. which variables violate a constraint—so attention can be focused on the variables that matter.
- ➔ many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP. (chapter on Planning)

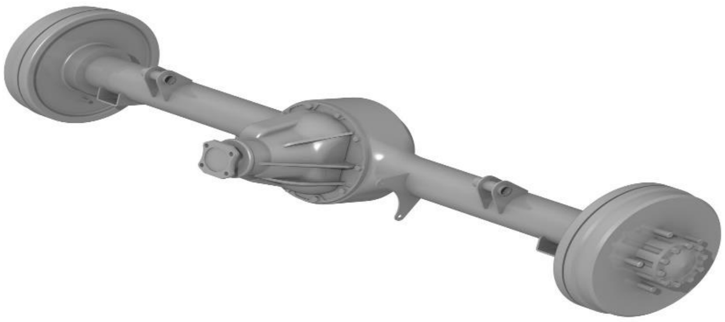
Example problem: Job-shop scheduling

- Problem of scheduling the day's work (in terms of job/tasks) in a factory, subject to various constraints.
- Many of these problems are solved with CSP techniques.
- E.g. scheduling the assembly of a car.
  - The job is composed of tasks;
  - Each <u>task can be modeled as a variable</u>, whose <u>value is the time the task starts</u>, expressed as an integer number of minutes;
  - Constraints can state, for instance:
    - that one task must occur before another. E.g., a wheel must be installed before the hubcap is put on;
    - that only so many tasks can go on at once;
    - the amount of time it takes to complete.

Example: car assembly scheduling

- Suppose the problem consists of (only) 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.
- The tasks can be represented with 15 variables:

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



Example: car assembly scheduling

- The value of each variable: the time the task starts.
- **Precedence constraints** between individual tasks:
  - Whenever a task T1 must occur before task T2, and task T1 takes duration d1 to complete, we add an arithmetic constraint: $T1 + d1 \leq T2$.
- In our example,
  - the axles must be placed before the wheels
  - it takes 10 minutes to install an axle. ➔

$$Axle_F + 10 \leq Wheel_{RF} \; ; \quad Axle_F + 10 \leq Wheel_{LF} \; ;$$
$$Axle_B + 10 \leq Wheel_{RB} ; \quad Axle_B + 10 \leq Wheel_{LB} \; ;$$

Example: car assembly scheduling

- For each wheel: affix it (1 minute), then tighten the nuts (2 minutes), and attach the hubcap (1 minute):

  $Wheel_{RF} + 1 \le Nuts_{RF}$; $Nuts_{RF} + 2 \le Cap_{RF}$;
  $Wheel_{LF} + 1 \le Nuts_{LF}$; $Nuts_{LF} + 2 \le Cap_{LF}$;
  $Wheel_{RB} + 1 \le Nuts_{RB}$; $Nuts_{RB} + 2 \le Cap_{RB}$;
  $Wheel_{LB} + 1 \le Nuts_{LB}$; $Nuts_{LB} + 2 \le Cap_{LB}$.

- Suppose: there are 4 workers to install wheels, but they share one tool to put the axle in place.
  ➔ We need a **disjunctive constraint** to say that $Axle_F$ and $Axle_B$ must not overlap in time; one must precede the other; so:
  (AxleF + 10 ≤ AxleB) **or** (AxleB + 10 ≤ AxleF)

Example: car assembly scheduling

- The *inspection* comes last and takes 3 minutes. ➔ For each variable except *Inspect,* we add a constraint of the form $X + d_X \leq Inspect$.
- Suppose there is a requirement to get the whole assembly done in 30 minutes.
  - Can be achieved by limiting the domain of all variables: $D_i = \{1, 2, 3, . . . , 27\}$.
- This 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.

Variations on the CSP Formalism

- Simplest kind of CSP: variables have **discrete**, **finite domains**. E.g.:
  - Map-colouring
  - Scheduling with time limits.
  - 8-Queens problem: each of the variables Q_1, … ,Q_8 takes a value in $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- A discrete domain can be **infinite**. E.g. if no deadline on the job-scheduling problem, there would be an infinite number of start times for each variable.
- For variables with infinite domains:
  - The constraints cannot be described by enumerating all combinations of possible values.
  - Need for a **constraint language**. E.g. to express a constraint like $T_1 + d_1 \leq T_2$ on variables T_1 and T_2.

Variations on the CSP Formalism

- CSPs with infinite domains are also common:
  - E.g. the scheduling of experiments on the Hubble Space Telescope.
  - Field of Operations Research.
  - The best-known category of such CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities.
- CSPs based on the types of constraints:
  - **Unary constraint**: simplest type; restricts the value of a constraint to a single variable. E.g. <(SA), SA ≠ green>.
  - **Binary constraint:** relates two variables. E.g. SA ≠ NSW.
    - A binary CSP is one with only binary constraints and can be represented as a constraint graph.

Variations on the CSP Formalism

- A **ternary constraint** involves three variables, e.g. *between(X, Y,Z)*.
- A **global constraint** involves any number of variables.
  - E.g. *Alldiff* CSP: all the variables involved in the constraint must have different values.
  - E.g. Sudoku: all variables in a row or column or box must satisfy an *Alldiff* constraint.
  - **Cryptarithmetic** (*Alldiff*) puzzles: each letter in the puzzle represents a different digit. Eg. *Alldiff (F, T,U,W,R,O).*

```
      T W O
   +  T W O
   ------------
  = F O U R
```

Variations on the CSP Formalism

- The addition constraints for this puzzle can be written as:

$$O + O = R + 10 * C_1$$

$$C_1 + W + W = U + 10 * C_2$$

$$C_2 + T + T = O + 10 * C_3$$

$$C_3 = F$$

where C_1, C_2, and C_3 are auxiliary variables representing the digit (1 or 0) carried over into the 10s, 100s, or 1000s.

These constraints can be represented in a **constraint hypergraph**.

- A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints.
- Note that **any CSP can be transformed into one with only binary constraints.**

Variations on the CSP Formalism

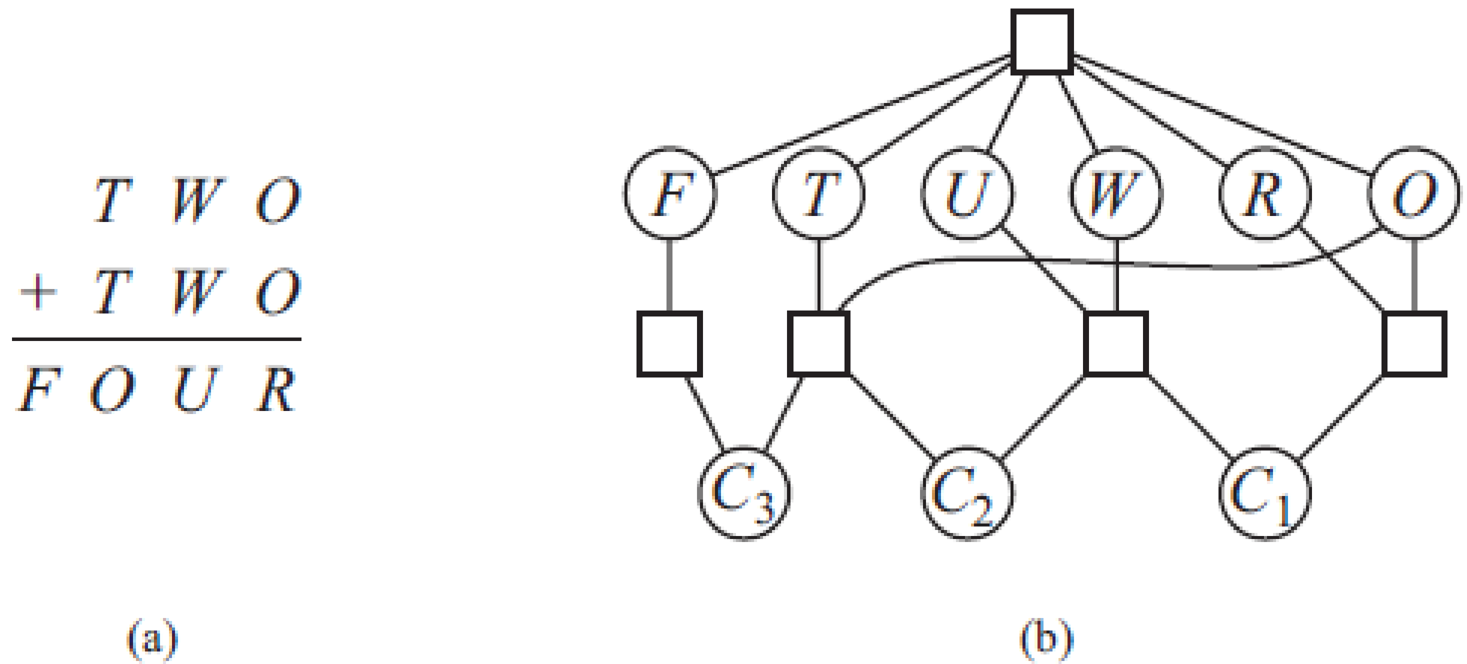


Figure 6.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic 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 C_1, C_2, and C_3 represent the carry digits for the three columns.

Variations on the CSP Formalism

- **Preference constraints** indicate which solutions are preferred. E.g., in a university class-scheduling problem:
  - There are absolute constraints that no professor can teach two classes at the same time.
  - Preference constraints may also be allowed : Prof. A prefers teaching in the morning, but Prof. B prefers teaching in the afternoon.

  ➔ A schedule that has Prof. A teaches at 2 p.m. would be an allowable solution but not an optimal one.
- Preference constraints can often be encoded as costs on individual variable assignments
  - E.g., assigning an afternoon slot for Prof. A costs 2 points against the overall objective function, but a morning slot costs 1.
  - As such, 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, as in Linear Programming problems.

Outline

- Defining Constraint Satisfaction Problems
  - Example problem: Map colouring
  - Example problem: Job-shop scheduling
  - Variations on the CSP formalism
- Constraint Propagation: Inference in CSPs
  - Node consistency
  - Arc consistency
  - Path consistency
  - *K*-consistency
  - Global constraints
  - Sudoku example

Constraint Propagation: Inference in CSPs

- In CSPs there is a choice:
  - an algorithm can search (choose a new variable assignment from several possibilities)
  - and/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.
- The key idea is **local consistency** which is of different types: Node consistency, Arc consistency, Path consistency, *K*-consistency.

Node consistency

- A single variable (a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints.
  - In variant of Australia map-colouring problem where South Australians dislike green, the variable SA starts with domain {red , green, blue}, and is made node-consistent by eliminating green ➔ SA then has the reduced domain {red , blue}.
  - A network is node-consistent if every variable in it 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 variable X_j if $\forall d_i \in D_i, \exists d_j \in D_j\ s.t.$

$$binary\ constraint\ on\ arc(X_i, X_j) is\ satisfied$$

  - E.g., consider the constraint Y = X^2 where the domain of X and Y is the set of digits {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.
    - This constraint can be written 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 becomes arc-consistent

Arc consistency

- Note that arc consistency can do nothing for the Australia map-colouring problem.
  - E.g., 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.

Arc consistency: AC-3

- AC-3: most popular algorithm for arc consistency.
- AC-3 maintains a queue of arcs to consider to make every variable arc-consistent. (Actually, the order of consideration is not important, so the data structure is really a set, but traditionally called a queue.)
- Initially, the queue contains all the arcs in the CSP.
- AC-3 then pops off an arbitrary arc (X_i, X_j) from the queue and makes X_i arc-consistent with respect to X_j .
- If this leaves D_i unchanged,
  - then the algorithm just moves on to the next arc.
  - else, D_i (makes the domain smaller), so we add to the queue all arcs (X_k, X_i) where X_k is a neighbour of X_i. We need to do that because the change in D_i might enable further reductions in the domains of D_k, even if we have previously considered X_k.

Arc consistency: AC-3

- If D_i is revised down to the empty set,
  - then we know the whole CSP has no consistent solution, and AC-3 returns failure.
  - else, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue.
    - At this point, we end up with a CSP that is equivalent to the original CSP—they both have the same solutions.
    - The arc-consistent CSP will (in most cases) be faster to search because its variables have smaller domains.
- It is possible to extend the notion of arc consistency to handle n-ary (not just binary) constraints.
- Generalised arc consistency (or hyperarc consistency).

Arc consistency: AC-3

```
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) ← 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 ← 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
```

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.

- It is also possible to transform all n-ary constraints into binary ones.
- It is common to define CSP solvers that work with only binary constraints. Assumption made in the sequel.

Path consistency

- Arc consistency can go a long way toward reducing the domains of variables, sometimes every domain to one value or a domain as empty set.
- For other networks, arc consistency fails to make enough inferences. E.g. map-colouring problem on Australia.
- Arc consistency tightens down the domains (unary constraints) using the arcs (binary constraints).
- To make progress on problems like map colouring, 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.

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 on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$.
- This is called **path consistency** because it can be seen as looking at a path from X_i to X_j with X_m in the middle.
- E.g. colouring the Australia map with two colours:
  - Let us make the set {WA,SA} path consistent with respect to NT.
  - Let us start by enumerating the consistent assignments to the set. 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, so no valid assignments for {WA,SA}.

➔ There can be no solution to this problem.

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.
- Special cases of k-consistency:
  - 1-consistency is the same as 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.

K-consistency

- Suppose we have a CSP with n nodes and it is strongly n-consistent.
- The problem can then be solved as follows:
  - First, choose a consistent value for X1.
  - Then one is guaranteed to be able to choose a value for X2 because the graph is 2-consistent,
  - .. And for X3 because it is 3-consistent, and so on.
  - For each variable Xi, we need only search through the d values in its domain D_i to find a value consistent with X1, . . . ,Xi−1.
- We are guaranteed to find a solution in time $O(n^2 d)$.
- However, any algorithm for <u>establishing n-consistency</u> must take <u>time exponential</u> in n in the worst case. <u>Same for space complexity</u>.
- In practice, practitioners commonly compute 2-consistency and less commonly 3-consistency.

Global constraints

- Remember: a **global constraint** is one involving an arbitrary number of variables (but not necessarily all variables).
- These constraints can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far.
  - E.g. ***Alldiff* problems**: all variables must have distinct values
  - A simple detection of inconsistency for *Alldiff* constraints: if it is an *m*-variable constraint, and there are *n* possible distinct values altogether, and $m > n$, then the constraint cannot be satisfied.
  - ➔ simple algorithm:
    1. Remove any variable X with singleton value *v*
    2. Remove *v* from the domains of the remaining variables
    3. Keep repeating steps 1 and 2; if empty domain reached or number of variables > number of available values, then **inconsistency detected**.
  - E.g. {WA=red , NSW=red} and SA, NT, Q to be coloured!

Example problem: Map colouring

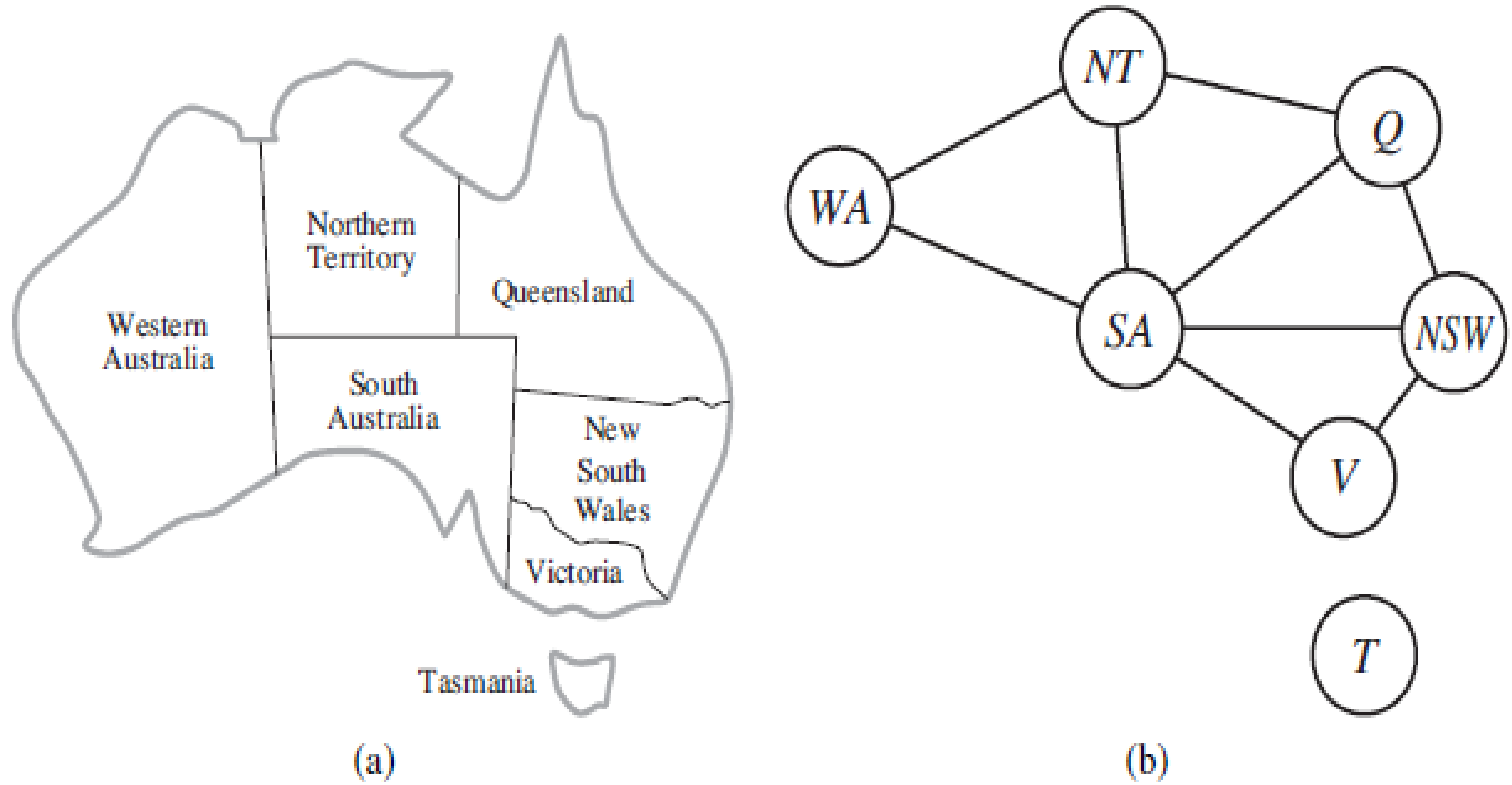


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Global constraints

- Other global constraint: **resource constraint, also** called the *atmost constraint*.
  - E.g., in a scheduling problem, let P1, . . . , P4 denote the numbers of workers assigned to each of four tasks.
  - Constraint: no more than 10 workers are assigned in total is written as *Atmost(10, P1, P2, P3, P4).*
  - One can detect inconsistency: check the sum of the minimum values of the current domains of the variables.
    - E.g. if variables domain is {3, 4, 5, 6}, then *Atmost* constraint cannot be satisfied.
  - Can also enforce consistency: delete the max value of any domain if it is not consistent with the min values of the other domains.
    - E.g. if variables domain is {2, 3, 4, 5, 6}, then the values 5 and 6 can be deleted from each domain.

Global constraints

- For large resource-limited problems with integer values:
  - E.g. logistics problems of moving thousands of people in hundreds of vehicles
- Not possible to represent the domain of a variable as a large set of integers and gradually reduce it by consistency-checking methods.
- Domains are represented by upper and lower bounds and **bounds propagation** is performed.
  - E.g., airline-scheduling problem: suppose for two flights F1 and F2, the planes have capacities 165 and 385, resp.
  - The initial domains for the numbers of passengers on each flight are then D1 = [0, 165] and D2 = [0, 385] .
  - Suppose the additional constraint: 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] .

Sudoku example

	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

- Sudoku board: 81 squares, some of which are initially filled with digits from 1 to 9.
- Puzzle: fill in all the remaining squares such that no digit appears twice in any row, column, or 3×3 box.
- A row, column, or box is called a **unit**.
- ➔ can be seen as a CSP with 81 variables, one for each square.
- Variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row.
- Empty squares have the domain {1, 2, 3, 4, 5, 6, 7, 8, 9}
- Prefilled squares have a domain consisting of a single value.

Sudoku example

- In addition, there are 27 different *Alldiff* constraints: one for each 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)*
  - . . .

Arc Consistency on Sudoku

- Applying arc consistency:
  - Assume that the *Alldiff* constraints have been expanded into binary constraints (e.g. A1 ≠ A2)
  - So we can apply the AC-3 algorithm directly.
  - In the example, consider variable E_6
    - From constraints in the box: we can remove 1, 2, 7, 8 from E_6's domain.
    - From constraints in its column: remove 5, 6, 2, 8, 9, and 3. ➔ E_6 has singleton domain {4}. E_6 now known.
  - Then consider variable I_6
    - Applying arc consistency in its column, remove 5, 6, 2, 4, 8, 9, and 3.
    - By arc consistency on row, remove 1 ➔ domain of I_6 is {7}.
    - Now we can infer that A6 must be 1.

Arc Consistency on Sudoku

- AC-3 works only for the easiest Sudoku puzzles.
- Slightly harder ones can be solved by PC-2 (for achieves path consistency), but at a greater computational cost: there are 255,960 different path constraints to consider in a Sudoku puzzle.
- To solve the hardest puzzles, need to apply more complex inference strategies.
- The **"naked triples" strategy** works as follows:
  - In any unit (row, column or box), find three squares that each have a domain that contains the same three numbers or a subset of those numbers. E.g. {1, 8}, {3, 8}, and {1, 3, 8}.
  - Here we don't know which square contains 1, 3, or 8.
  - BUT we know we can remove 1, 3, and 8 from the domains of every *other* square in the unit.

Sudoku and CSP

- Note that all that has been done is not specific to Sudoku (including the "naked triples" strategy)
- We do have to say that:
  - there are 81 variables,
  - their domains are the digits 1 to 9, and
  - there are 27 Alldiff constraints.
- Beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems.
- The <u>power of the CSP formalism</u>: 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.

Outline

- Defining Constraint Satisfaction Problems
  - Example problem: Map colouring
  - Example problem: Job-shop scheduling
  - Variations on the CSP formalism
- Constraint Propagation: Inference in CSPs
  - Node consistency
  - Arc consistency
  - Path consistency
  - K-consistency
  - Global constraints
  - Sudoku example

Outline

- Backtracking Search for CSPs
  - Variable and value ordering
  - Interleaving search and inference
  - Intelligent backtracking: Looking backward
- Local Search for CSPs
- The Structure of Problems

Backtracking Search for CSPs

- Some CSPs, e.g. Sudoku, are solved only by inference over constraints.
- But many other CSPs cannot be solved by inference alone; the need for search for a solution arises.
- Standard depth-limited search can be applied.
- A **state** would be a **partial assignment**, and an **action** would be adding *var = value* to the assignment.
- Problem:
  - For a CSP with n variables of domain size d, the branching factor at the top level is $n.d$
  - At the next level, the branching factor is $(n - 1)\,d$, and so on for n levels.

➔ We generate a tree with $n! \cdot d^n$ leaves, even though there are only d^n possible complete assignments!

Backtracking Search for CSPs

- The previous complexity is drastically simplified when considering that all CSPs are **commutative**.
- A problem is commutative if the order of application of any given set of actions has no effect on the outcome.
- In CSPs, when assigning values to variables, we reach the same partial assignment regardless of order.
- ➔ we need only consider a *single* variable at each node in the search tree.
- With this restriction, the number of leaves is d^n.
- **Backtracking search:** a DFS that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- The BACKTRACKING-SEARCH algorithm keeps only a single representation of a state and alters that representation rather than creating new ones, as seen in classical search.

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
```

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.

Example: Search for Colouring the Map of Australia CSP

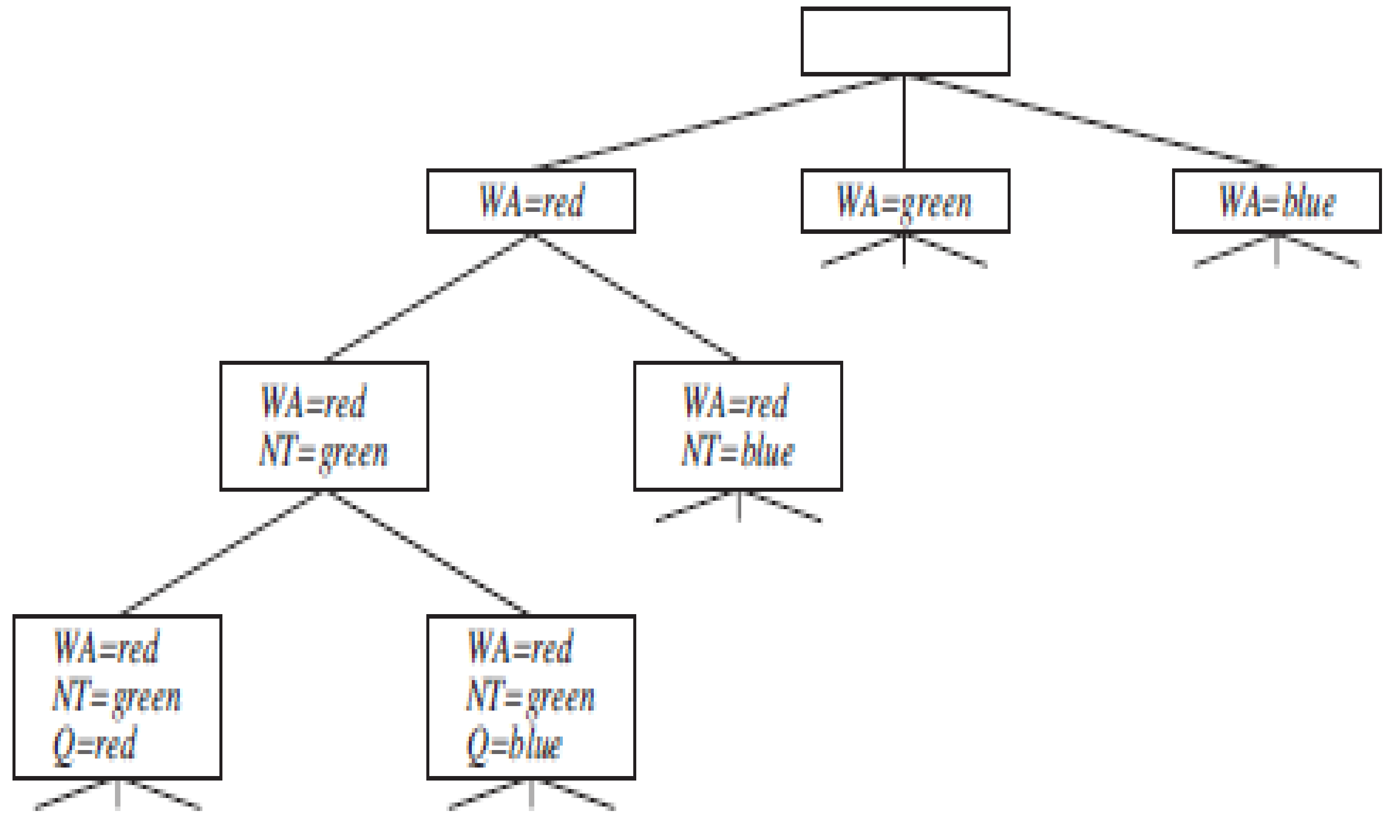


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

Improving the CSP Solving Efficiency

- In classical search, domain-specific heuristics were used.
- We can solve CSPs efficiently *without* such domain-specific knowledge.
- Instead, some sophistication is added to the functions used in the BACKTRACKING-SEARCH algorithm:
  - Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
  - What inferences should be performed at each step in the search (INFERENCE)?
  - When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

Variable and value ordering

- In the backtracking algorithm, the simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, {X1, X2, . . .}.
- Usually does not result in the most efficient search.
  - After the assignments WA=red and NT =green, 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.
- Choosing the variable with the fewest “legal” values is called the **Minimum Remaining-Values** (MRV) heuristic (or also the “most constrained variable” or “fail-first” heuristic).
- If a variable X has no legal values left, MRV will select X and failure will be detected immediately.
- MRV usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more.

Variable and value ordering

- Note that in map colouring example, MRV doesn’t help at all in choosing the first region to color in Australia, initially every region having three legal colours.
- The **Degree Heuristic** attempts to reduce the branching factor on future choices by <u>selecting the variable that is involved in the largest number of constraints on other unassigned variables</u>.
  - In map example, SA has degree 5; the other variables have degree 2 or 3, except for T, which has degree 0.
  - So select SA which then makes the solution straightforward.
- *MRV is usually a more powerful guide, but the degree heuristic can be useful as a tie-breaker.*

Variable and value ordering

- Once a variable has been selected, the algorithm must decide on the order in which to examine its values.
- The **Least-Constraining-Value** heuristic can be effective in some cases.
- It prefers the value that rules out the fewest choices for the neighbouring variables in the constraint graph.
  - E.g., suppose we have generated the partial assignment {WA=red, 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 neighbour, SA.
  - The least-constraining-value heuristic therefore prefers red to blue.
- The least-constraining-value heuristic tries to leave the maximum flexibility for subsequent variable assignments.
- If we aim to find all the solutions to a problem, then the ordering does not matter because we have to consider every value.

Example problem: Map colouring

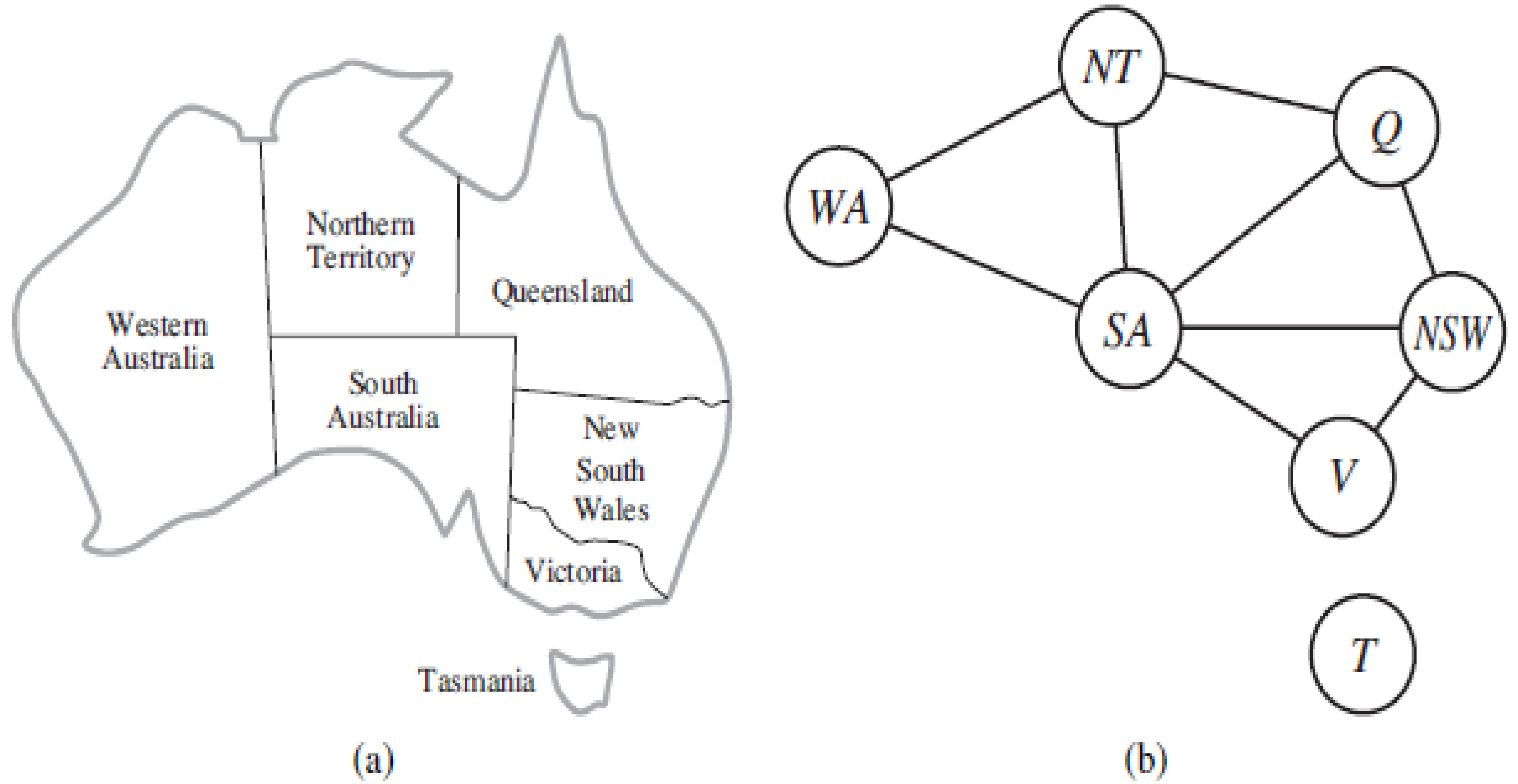


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Interleaving search and inference

- AC-3 and other algorithms can infer reductions in the domain of variables ***before*** we begin the search.
- **Inference** during a search: every time we make a choice of a value for a variable, we can infer new domain reductions on the neighbouring variables.
- **Forward checking**: a form of inference:
  - Establishes arc consistency when a variable X is assigned a value.
  - 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.
  - No need to do forward checking if arc consistency already done as a preprocessing step.

Forward checking for map colouring

- Oftentimes, search will be more effective if MRV heuristic and forward checking are combined. E.g.
  - Intuition: after assigning {WA=red}, this constrains its neighbours, NT and SA, so we should handle these variables next.
  - MRV with FC: after assigning {WA=red}, 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.
- FC detects many inconsistencies, but not all of them.
- Problem: FC makes current variable arc-consistent, but doesn't look ahead to make all the other variables arc-consistent. E.g.
  - When WA=red and Q=green, NT and SA are forced to be blue though they are neighbours!

Forward checking for map colouring

	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.

Forward checking has detected that the partial assignment {WA=red, Q=green, V =blue} is inconsistent
➔ the algorithm will backtrack immediately.

Forward checking for map colouring

- The **Maintaining Arc Consistency (MAC)** algorithm detects the previous 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, only the arcs (Xj,Xi) are added for all neighbours Xj that are unassigned variables .
    - Then, 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 backtracking occurs.
- MAC is strictly more powerful than FC because FC does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, FC does not <u>recursively propagate constraints</u> when changes are made to the domains of variables.

Intelligent backtracking: Looking backward

- **Chronological backtracking** (BACKTRACKING-SEARCH algorithm seen previously): when a branch of the search fails: back up to the preceding variable and try a different value for it.
  - i.e. the *most recent* decision point is revisited.
- Example of shortcomings of this form of backtracking:
  - Suppose a fixed variable ordering Q, NSW, V , T, SA, WA, NT; and
  - Suppose we have generated the partial assignment {Q=red, NSW =green, V =blue, T =red}.
  - For SA, we see that every value violates a constraint.

➔ backtrack to T to try a different colour which does not make sense.

Intelligent backtracking: Backjumping

- Alternative backtracking: backtrack to a variable that was responsible for making one of the possible values of SA impossible.
  - We keep track of a set of assignments that are in conflict with some value of the variable of interest (here SA).
  - The set is called the **conflict set** for SA. (Here, {Q=red ,NSW =green, V =blue})
  - The **backjumping** method backtracks to the *most recent* assignment in the conflict set;
  - Here, backjumping would jump over Tasmania and try a new value for V.
- 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.

Example problem: Map colouring

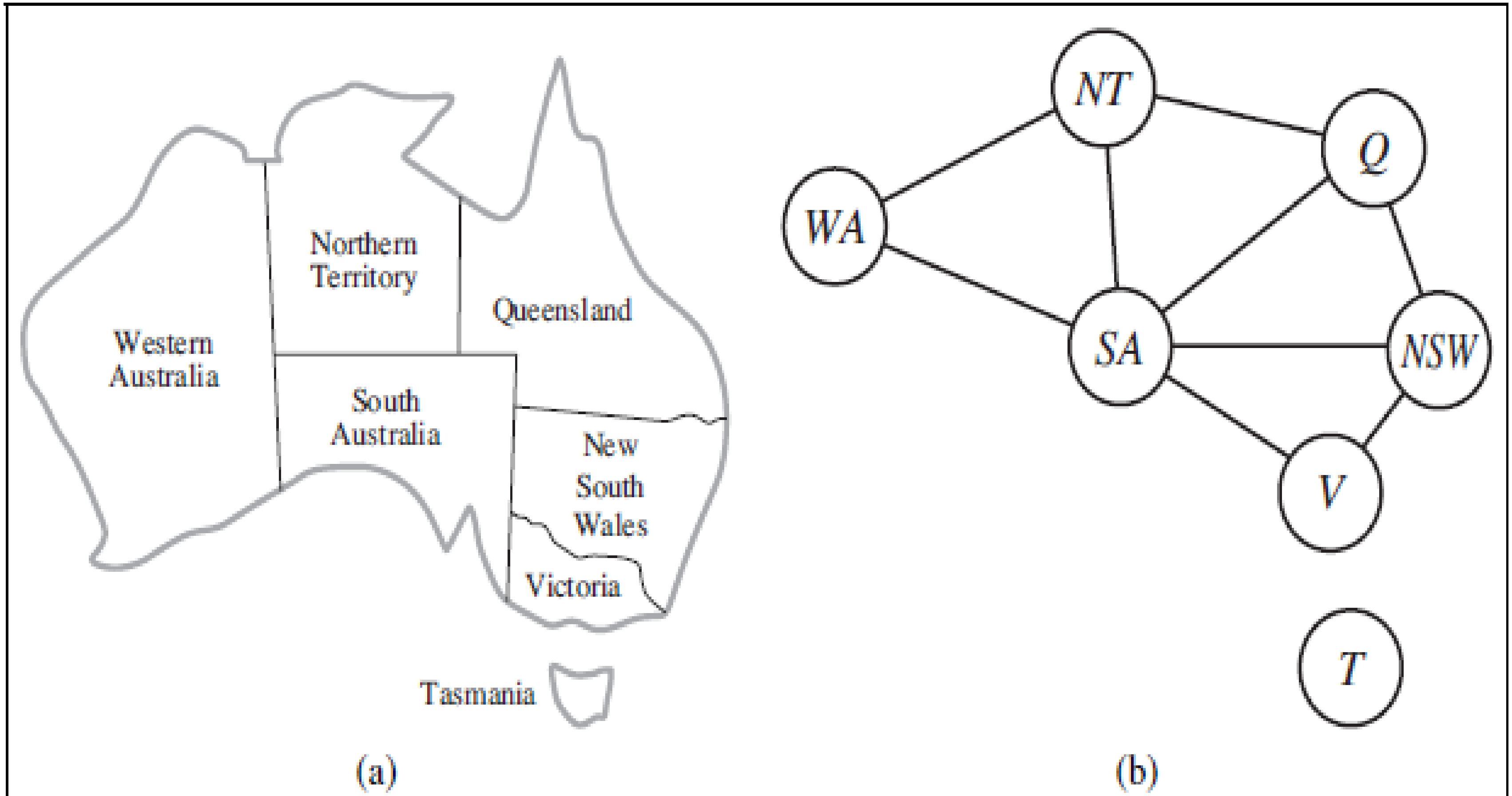


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Intelligent backtracking: Backjumping

- Note that FC can supply the conflict set with no extra work:
  - Whenever FC based on an assignment X=x deletes a value from Y's domain, it should add X=x to Y's conflict set.
  - If the last value is deleted from Y's domain, then the assignments in the conflict set of Y are added to the conflict set of X.
  - So, when we get to Y, we know immediately where to backtrack if needed.
- *Every* branch pruned by backjumping is also pruned by FC → simple backjumping is redundant in a FC search or, in a search that uses stronger consistency checking, such as MAC.
- More sophisticated forms of backjumping exist.

Constraint learning and no-good

- **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**.
- The no-good set is recorded, either by adding a new constraint to the CSP or by keeping a separate cache of no-goods.
  - Consider the state {WA = red, NT = green, Q = blue} in the bottom row of the following search tree.
  - Forward checking can tell us this state is a *no-good* because there is no valid assignment to SA.
  - Here, recording the no-good would not help, because once we prune this branch from the search tree, we will never encounter this combination again.
- Not the case if this branch started higher with assignments of other variables e.g. V and T where recording of no-good helps!

Example: Search for Colouring the Map of Australia CSP

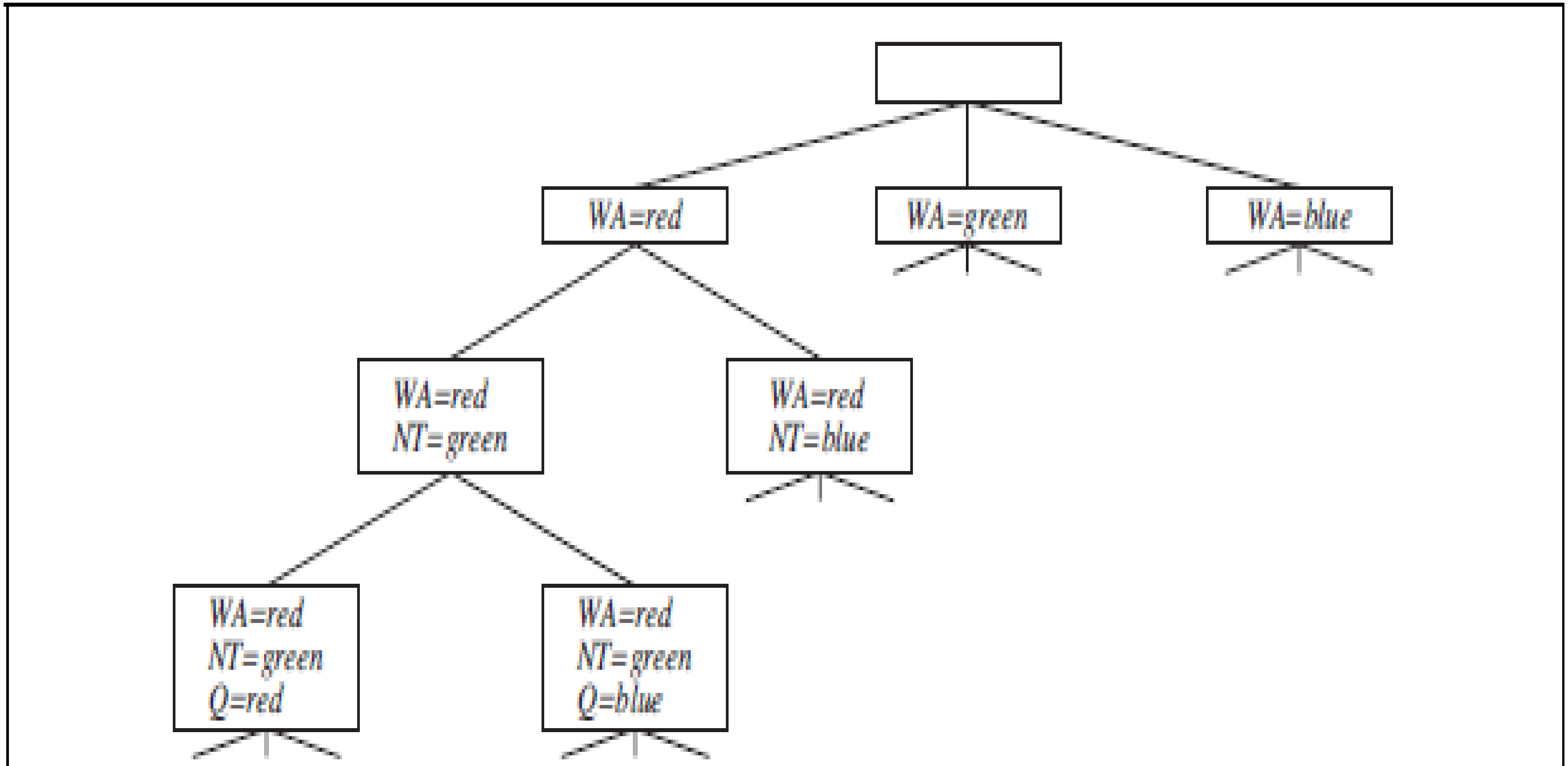


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

Local Search for CSPs

- Local search algorithms are 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.
  - E.g. 8-queens problem
  - The initial state could be a random configuration of 8 queens in 8 columns,
  - 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.

Local Search for CSPs: Min-Conflicts Algorithm

- 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** algorithm is very effective for many CSPs.
- On the n-queens problem, not counting 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).

Local Search for CSPs: Min-Conflicts Algorithm

```
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.

Min-Conflicts Algorithm applied to 8-Queens Problem

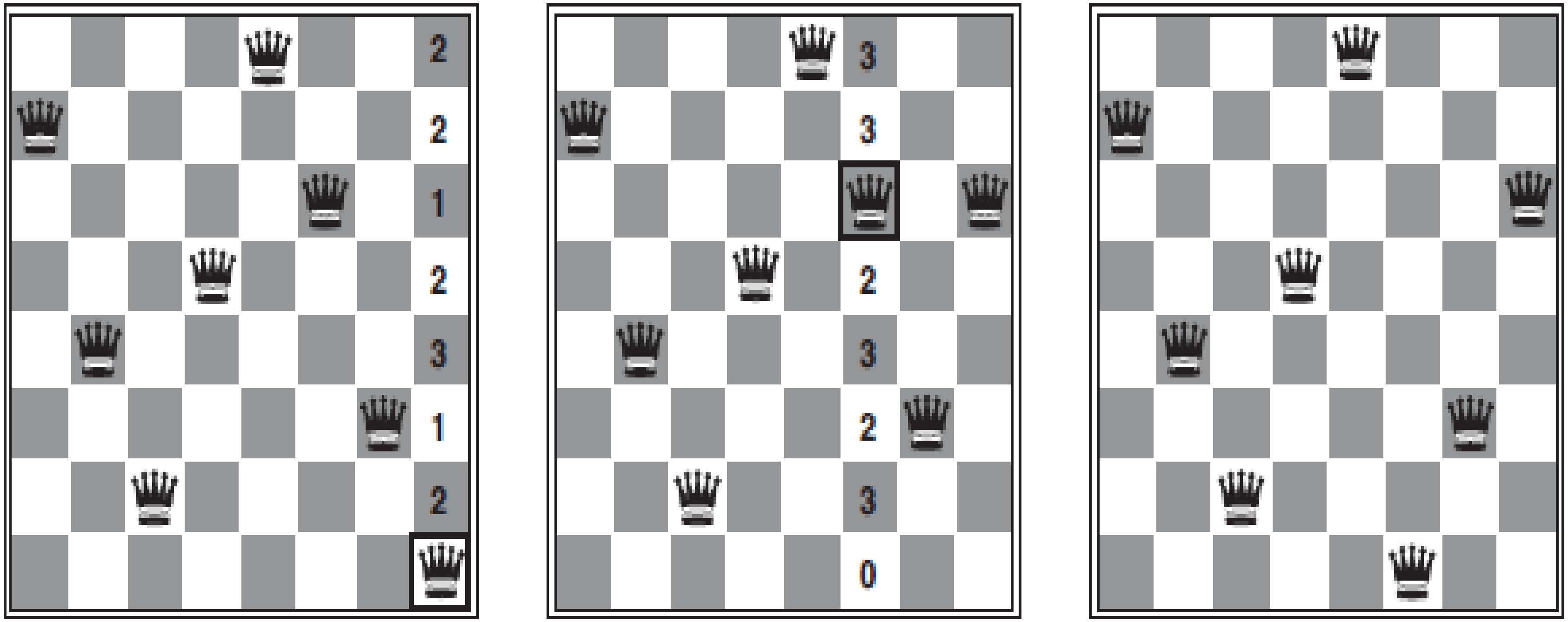


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking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

Local search for CSPs

- All the local search techniques studied are candidates for application to CSPs; some of those have proved especially effective.
- The landscape of a CSP under the <u>MinConflicts heuristic usually has a series of plateaux</u>.
- To get out of plateaux:
  - Allowing sideway moves
  - **Tabu Search:** keeping a small list of recently visited states and forbidding the algorithm to return to those states.
  - Simulated annealing

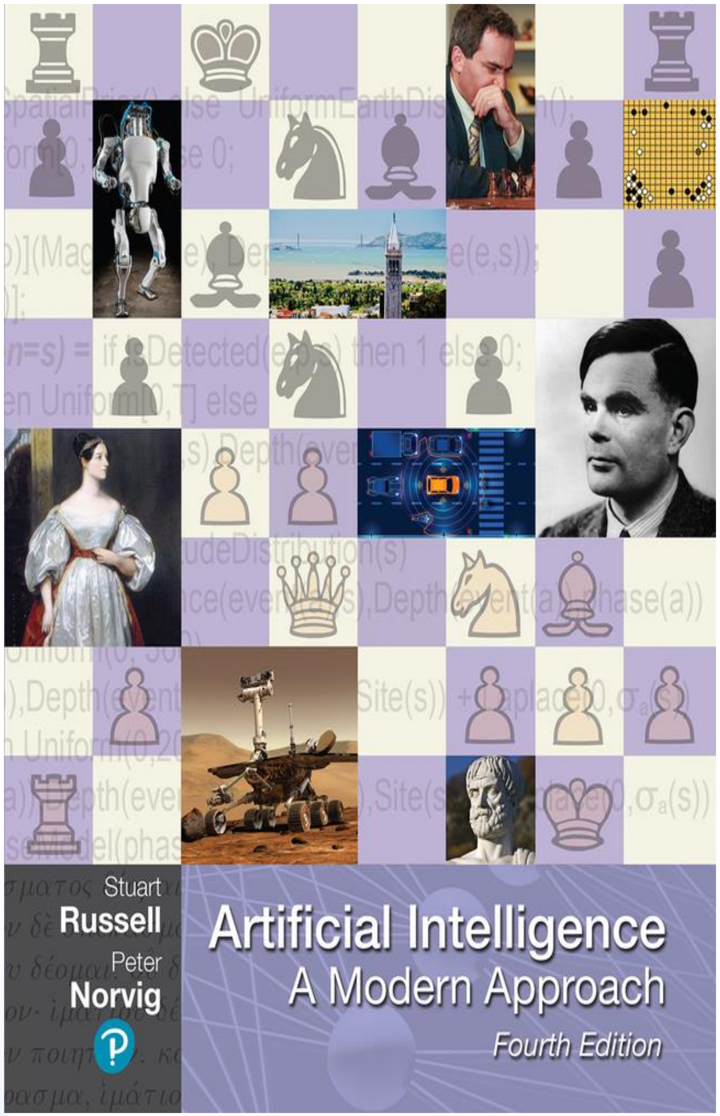
Local search for CSPs

- **Constraint weighting**, can help concentrate the search on the important constraints.
- Each constraint is given a numeric weight, W_i, initially all 1.
- At each step of the search, the algorithm <u>chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints</u>.
- The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.
- Two benefits:
  - It adds topography to plateaux, making sure that it is possible to improve from the current state, and
  - Over time, it adds weight to the constraints that are proving difficult to solve.

Local search for CSPs

- Other advantage of local search: it can be used in an online setting when the problem changes.
- This is particularly important in scheduling problems.
- Example: 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.

Slides based on the textbook



- Russel, S. and Norvig, P. (2020) Artificial Intelligence,

A Modern Approach (4th Edition), Pearson Education Limited.