CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

- http://www.ultimateaiclass.com/
- https://moodle.cis.fiu.edu/
- HW1 was due on Tuesday 10/3
  - Remember that you have up to 4 late days to use throughout the semester.
- HW2 out today, due 10/17
- Midterm on 10/19
  - Review during half of class on 10/17

Kuka

Kuka

- The clash, which looks to have been subject to extensive editing and probably wouldn't stand up to ITTF rules, saw Kuka steam ahead only for Boll to fight back to an 11-9 wine once he realised the robot couldn't cope with net calls, awkward bounces and other ping pong quirks.

Robot-soccer

Robot Soccer Goes Big Time

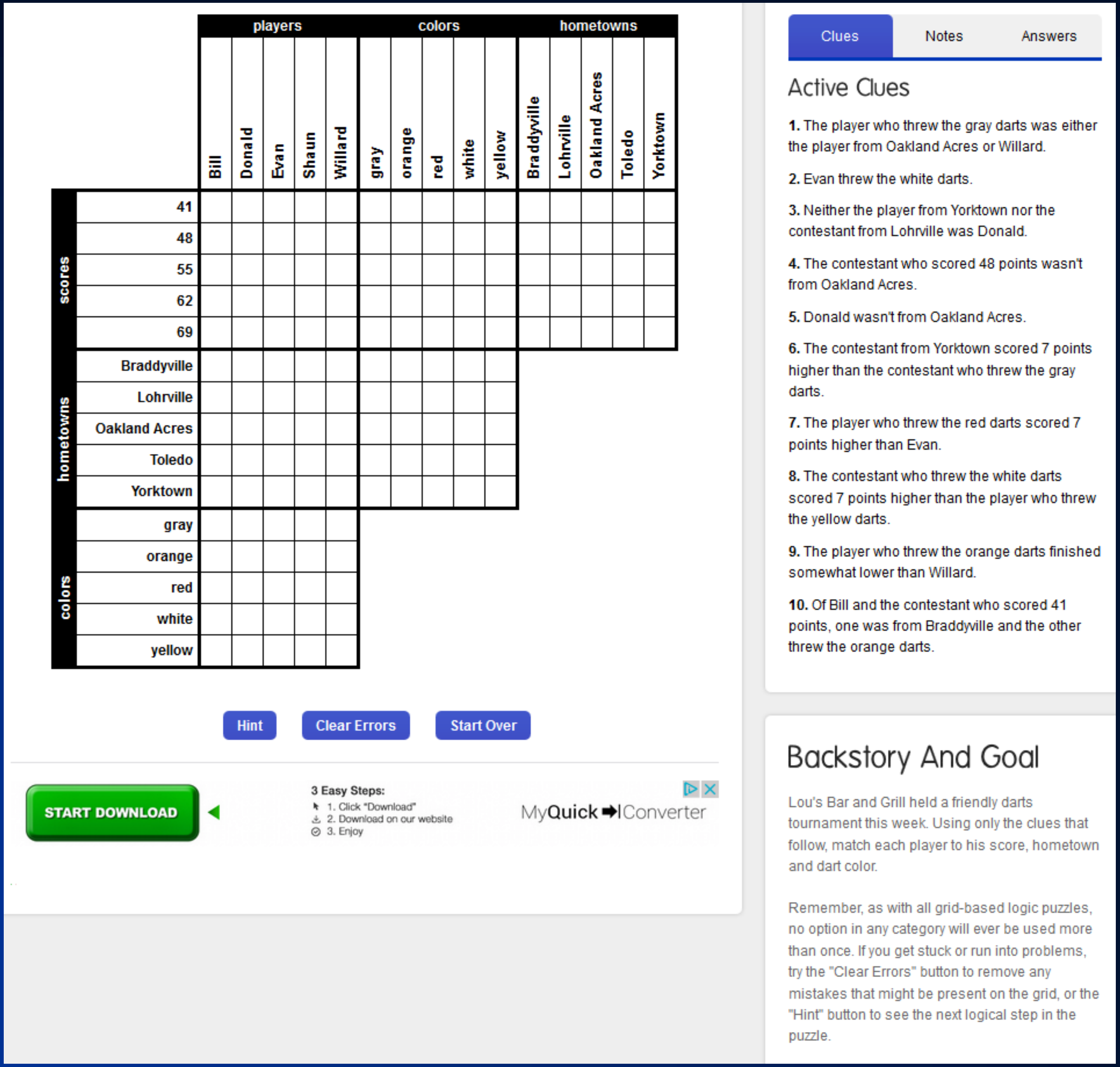
Upcoming lectures

- 10/5 (today): Finish CSP, start logic (propositional logic)
- 10/10: Wrap up logic (first-order logic, logical inference), start optimization (integer optimization)
- 10/12: Continue optimization (integer, linear optimization)
- 10/17: Wrap up optimization (nonlinear optimization), midterm review
- 10/19: Midterm
- Planning lecture will be after midterm on 10/26.

HW2

- Out today due 10/17
- Several exercises from textbook
- Logic puzzles that you must formulate models for as search/optimization problems using two different approaches (e.g., could be CSP, logical inference, integer programming). You can solve them using built-in Python solver libraries (e.g., for CSP and ILP) or build your own solver (possibly for extra credit). Open-ended question and many possible correct answers and approaches.

Quizzle



		players					colors					hometowns				
		Bill	Donald	Evan	Shaun	Willard	gray	orange	red	white	yellow	Braddyville	Lohrville	Oakland Acres	Toledo	Yorktown
scores	41															
	48															
	55															
	62															
	69															
hometowns	Braddyville															
	Lohrville															
	Oakland Acres															
	Toledo															
	Yorktown															
colors	gray															
	orange															
	red															
	white															
	yellow															

3 Easy Steps:
1. Click "Download"
2. Download on our website
3. Enjoy

Active Clues

1. The player who threw the gray darts was either the player from Oakland Acres or Willard.

2. Evan threw the white darts.

3. Neither the player from Yorktown nor the contestant from Lohrville was Donald.

4. The contestant who scored 48 points wasn't from Oakland Acres.

5. Donald wasn't from Oakland Acres.

6. The contestant from Yorktown scored 7 points higher than the contestant who threw the gray darts.

7. The player who threw the red darts scored 7 points higher than Evan.

8. The contestant who threw the white darts scored 7 points higher than the player who threw the yellow darts.

9. The player who threw the orange darts finished somewhat lower than Willard.

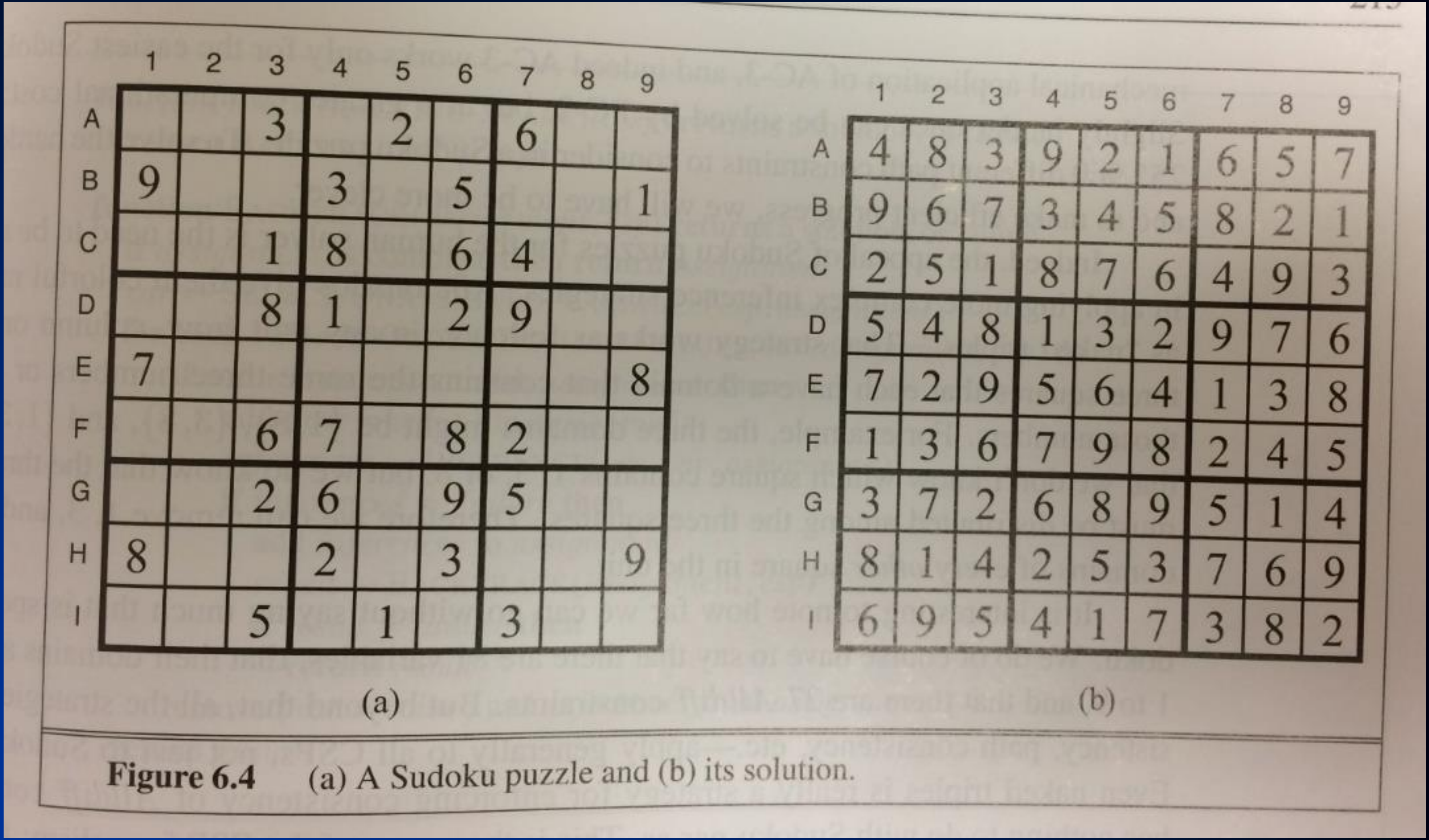
10. Of Bill and the contestant who scored 41 points, one was from Braddyville and the other threw the orange darts.

Backstory And Goal

Lou's Bar and Grill held a friendly darts tournament this week. Using only the clues that follow, match each player to his score, hometown and dart color.

Remember, as with all grid-based logic puzzles, no option in any category will ever be used more than once. If you get stuck or run into problems, try the "Clear Errors" button to remove any mistakes that might be present on the grid, or the "Hint" button to see the next logical step in the puzzle.

Example problem: Sudoku



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

(a)

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

(b)

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

CSP applications

- Examples of simple problems that can be modeled as a constraint satisfaction problem include:
  - Eight queens puzzle
  - Map coloring problem
  - Sudoku, Crosswords, Futoshiki, Kakuro (Cross Sums), Numbrix, Hidato and many other logic puzzles
- These are often provided with tutorials of ASP, Boolean SAT and SMT solvers. In the general case, constraint problems can be much harder, and may not be expressible in some of these simpler systems.
- "Real life" examples include automated planning and resource allocation. An example for puzzle solution is using a constraint model as a Sudoku solving algorithm.

Variations on the CSP formalism

- Standard variant: **discrete, finite domains**
  - E.g., map coloring and job-shop coloring
  - The 8-queens problem 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 Di={1,…,8}
- Discrete domain can be **infinite**, such as the set of integers or strings
  - Can no longer enumerate all combinations of values.
  - Instead use **constraint language** that can understand constraints such as T1 + d1 <= T2 directly, without enumerating the set of pairs of allowable values for (T1,T2)

Variations on the CSP formulation

- Special solution algorithms (which we will see shortly) exist for **linear constraints** on integer variables—that is, constraints such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables.

Variations on the CSP formulation

- 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 requires 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. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on.

CSP variations

- The simplest type of constraint is a **unary constraint**, which restricts the value of a single variable. For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green; 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
- We can also describe higher-order constraints, such as asserting that the value of Y is between X and Z, with the ternary constraint *Between*(X,Y,Z)

Constraint graph

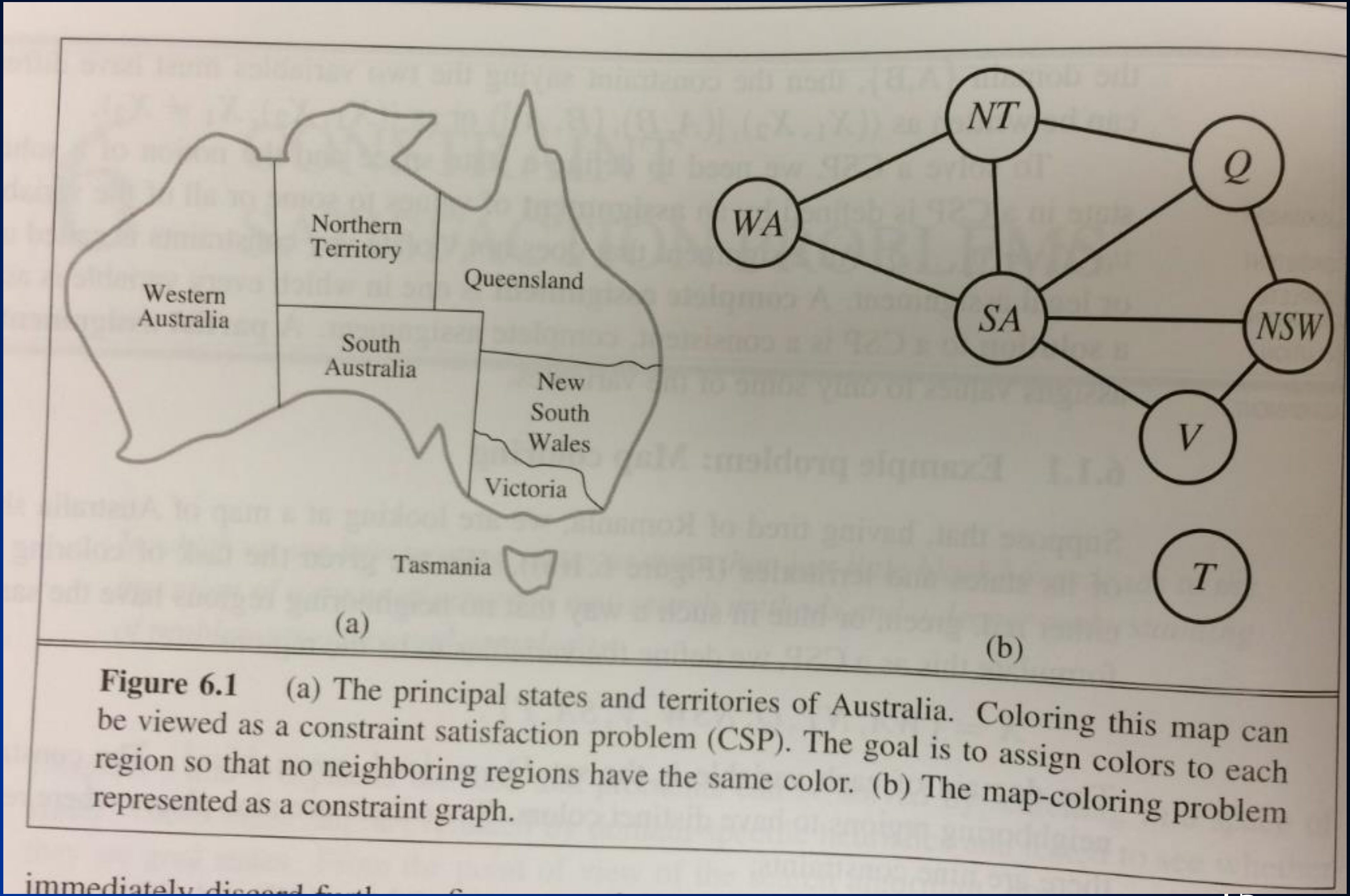
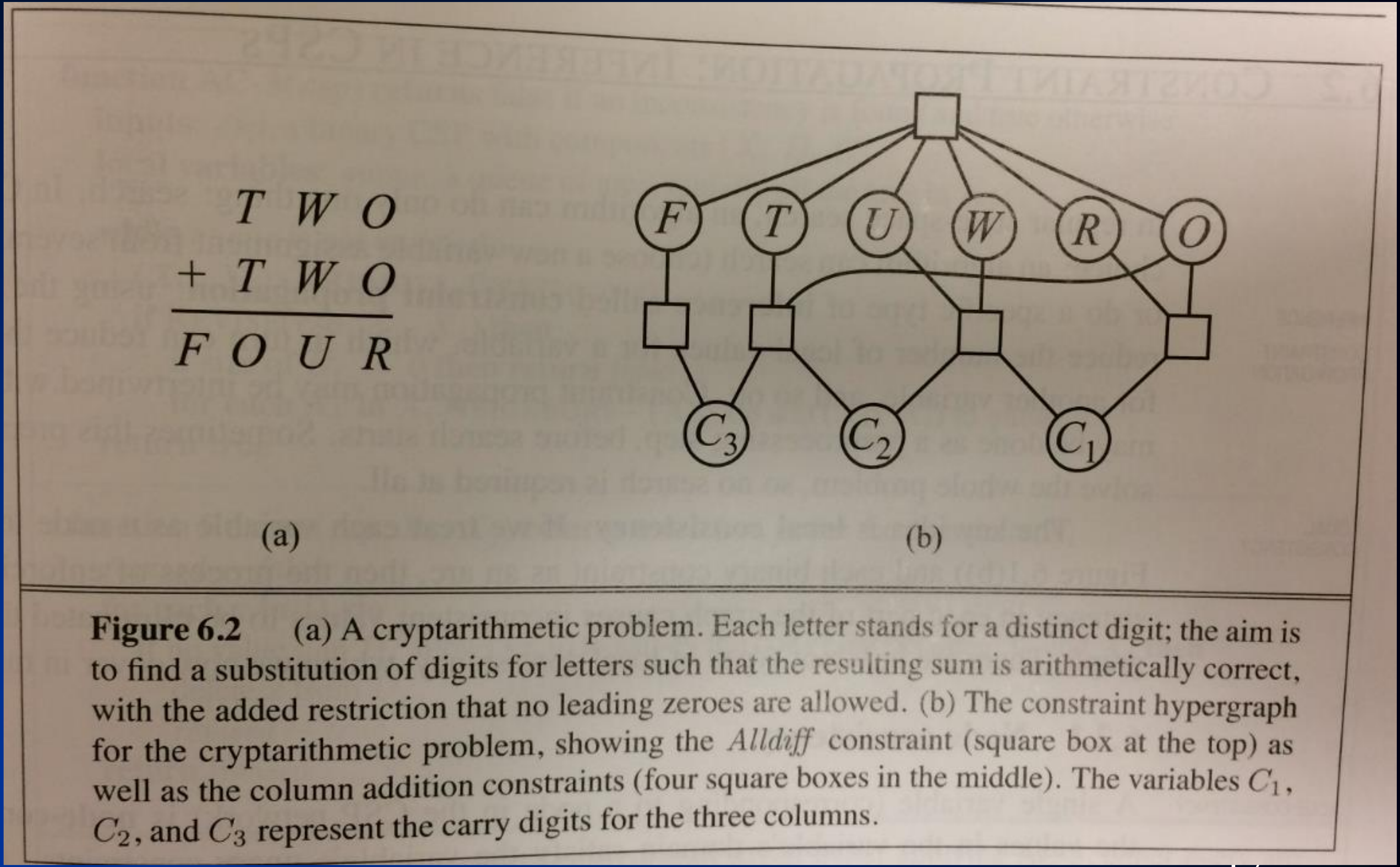


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

CSP variations

- 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, all variables in a row or column must satisfy an *Alldiff* constraint. Another example is provided by **cryptarithmetic puzzles**. Each letter represents a different digit. For the example problem this would be represented as the global constraint *Alldiff*(F,T,U,W,R,O).

Cryptarithmetic problem



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

(a)

F
T
U
W
R
O
C_3
C_2
C_1

(b)

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.

Cryptarithmetic problem

- The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:
  - …

Cryptarithmetic problem

- The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:
  - $O + O = R + 10 * C10$
  - $C10 + W + W = U + 10*C100$
  - $C100 + T + T = O + 10 * C1000$
  - $C1000 = F$
- Where C10, C100, and C1000 are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. 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.

CSP variations

- Alternatively, homework exercise asks you to prove every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced, so we could transform any CSP into one with only binary constraints; this makes the algorithms simpler.
- There are however two reasons why we might prefer a global constraint such as *Alldiff* rather than a set of binary constraints. First, it is easier and less error-prone to write the problem description using *Alldiff*. Second, it is possible to design special-purpose inference algorithms for global constraints that are not available for a set of more primitive constraints, which we will describe.

CSP variations

- 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, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one.

CSP variations

- 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/integer/nonlinear programming problems do this kind of optimization.

Inference in CSPs

- In regular state-space search, an algorithm can only do 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.

CSP inference

- The key idea is **local consistency**. If we treat each variable as a node in a graph (like for the Australia constraint graph) and each binary constraint as an arc (edge), then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are several different types of local consistency: node consistency, arc consistency, path consistency, *k*-consistency.

Arc consistency

- A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable’s binary constraints. For formally, Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi,Xj). 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 explicitly as …

Arc consistency

- A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. For formally, Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi,Xj). 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 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.

Arc consistency

- How about for the Australia map-coloring problem?

Arc consistency

- 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 value value for the other variable. So applying arc consistency has no effect on the domains of either variable.

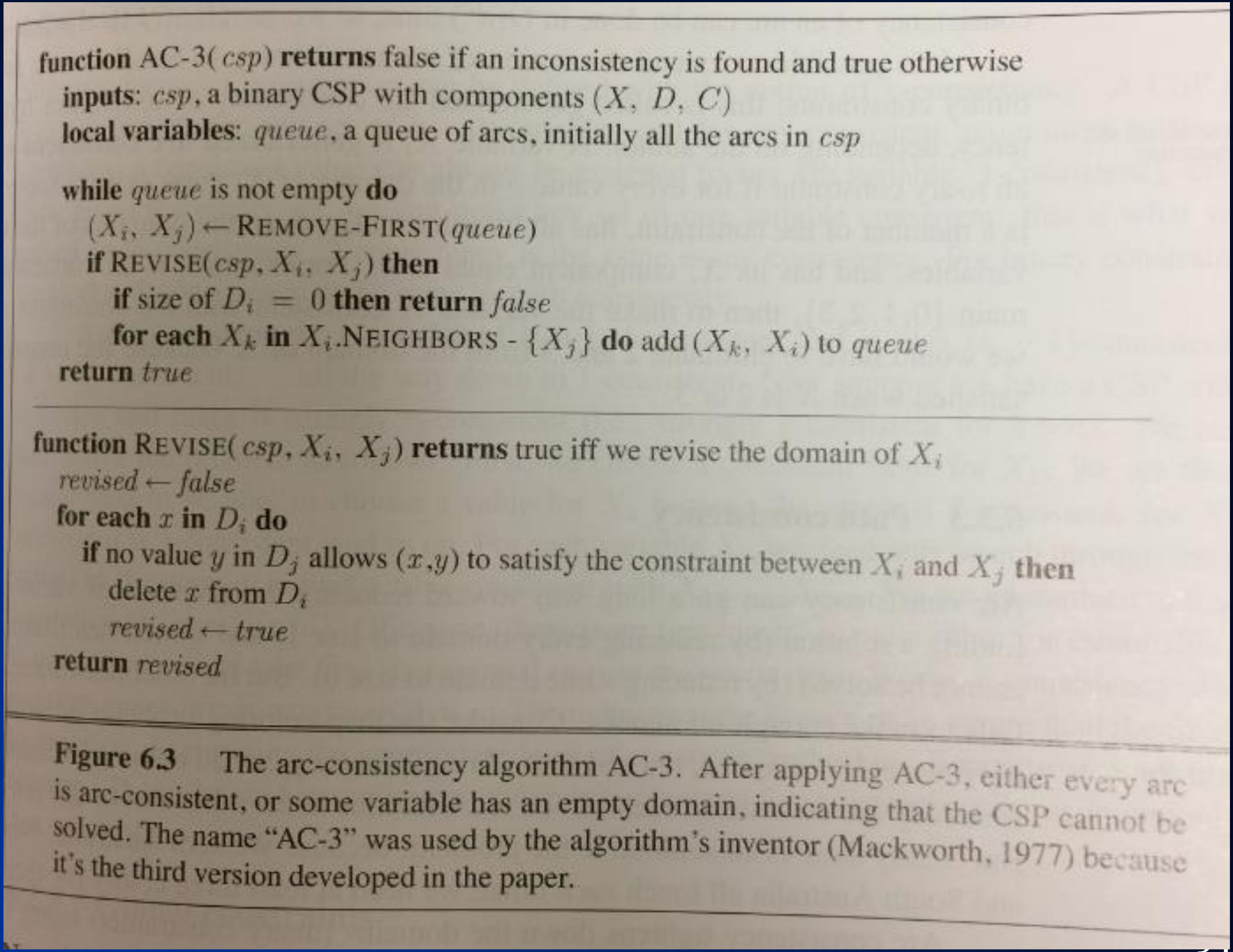
Arc consistency

- The most popular algorithm for arc consistency is called AC-3. To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider (Actually the order of consideration is not important, so the data structure is really a set, but tradition calls it a queue). Initially the queue contains all the arcs in the CSP. (Each binary constraint becomes two arcs, one in each direction.) AC-3 then pops off an arbitrary arc (Xi,Xj) from the queue and makes Xi arc-consistent with respect to Xj. If this leaves Di unchanged, the algorithm just moves to the next arc. But if this revises Di (makes the domain smaller), then we add to the queue all arcs (Xk,Xi) where Xk is a neighbor of Xi. We need to do that because the change in Di might enable further reductions in the domains of Dk, even if we have previously considered Xk.

Arc consistency

- If Di is revised down to nothing, then we know the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, we keep checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, we are left with a CSP that is equivalent to the original CSP—they both have the same solutions—but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains.

Arc consistency



```
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.

- 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 (Xk,Xi) can be inserted in the queue only d times because Xi has at most d values to delete. Checking consistency of an arc can be done in O(d^2) time, so we get O(cd^3) total worst-case time.
  - Note that the newer "AC-4" algorithm runs in O(cd^2) worst-case time but can be slower than AC-3 on average cases.

Arc consistency

- It is possible to extend the notion of arc consistency to handle n-ary rather than just binary constraints; this is called generalized arc consistency or sometimes *hyperarc consistency*. A variable Xi is **generalized arc consistent** with respond to an n-ary constraint if for every value v in the domain of Xi there exists a tuple of values that is a member of the constraint, has all its values taken from the domains of the corresponding variables, and has its Xi component equal to v. For example, if all variables have the domain {0,1,2,3}, then to make the variable X consistent with the constraint $X < Y < Z$, we would have to eliminate 2 and 3 from the domain of X because the constraint cannot be satisfied when X is 2 or 3.

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 WA, NT and SA 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 achieves path consistency in much the same way that AC-3 achieves arc consistency.

Path consistency

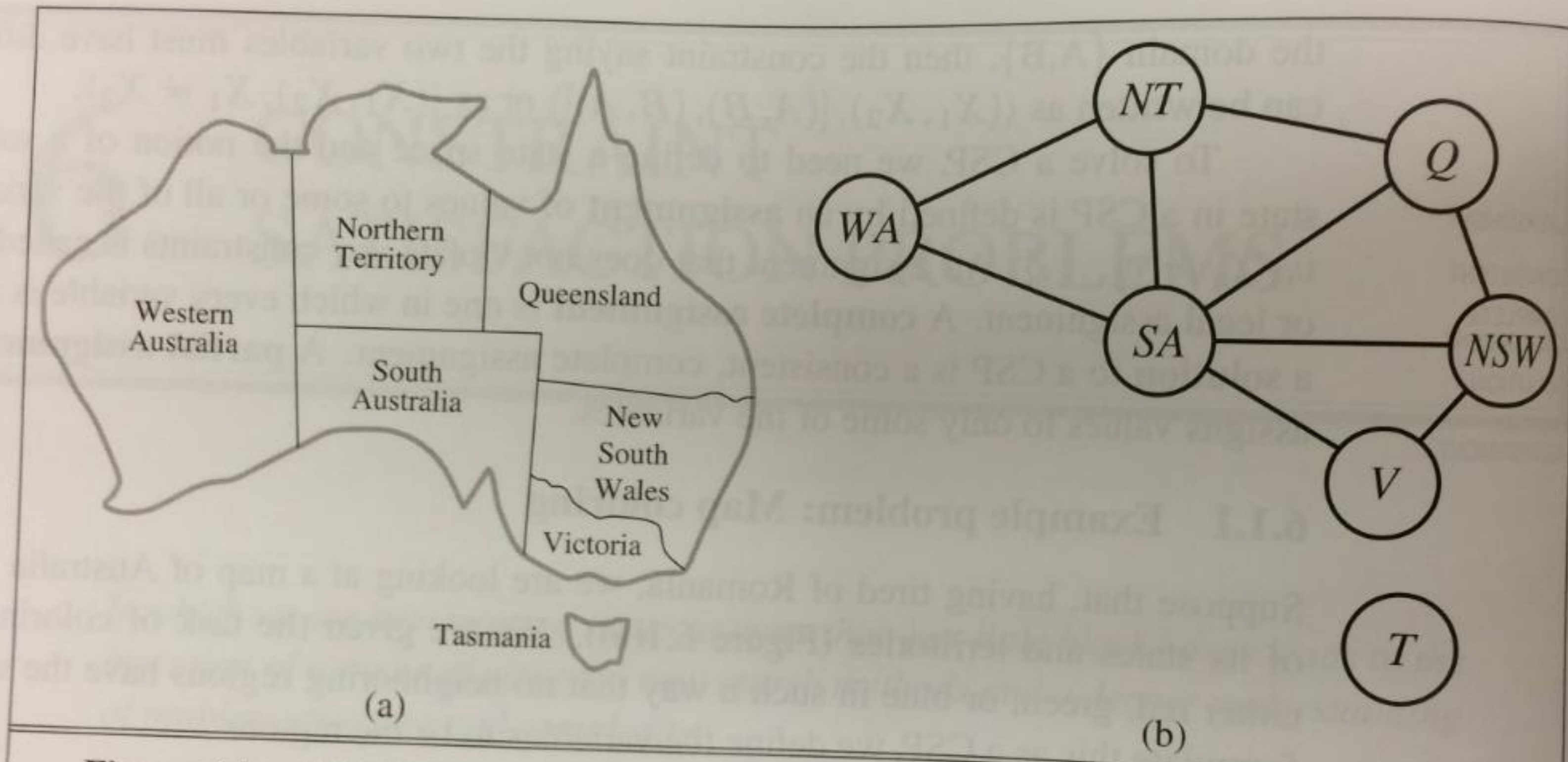


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

immediately discard furth

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.

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* constraints work 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

- 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

- This method can detect the inconsistency in the assignment {WA=red, NSW = red}. Notice that the variables SA, NT, and Q are effectively connected by an *Alldiff* constraint because each pair must have two different colors. After applying AC-3 with the partial assignment, the 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. There are more complex inference algorithms for *Alldiff* that propagate more constraints but are more computationally expensive to run.

Global constraints

- 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 number 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, fi 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 one example has the domain {2,3,4,5,6}, the values 5 and 6 can be deleted from each domain.

Global constraints

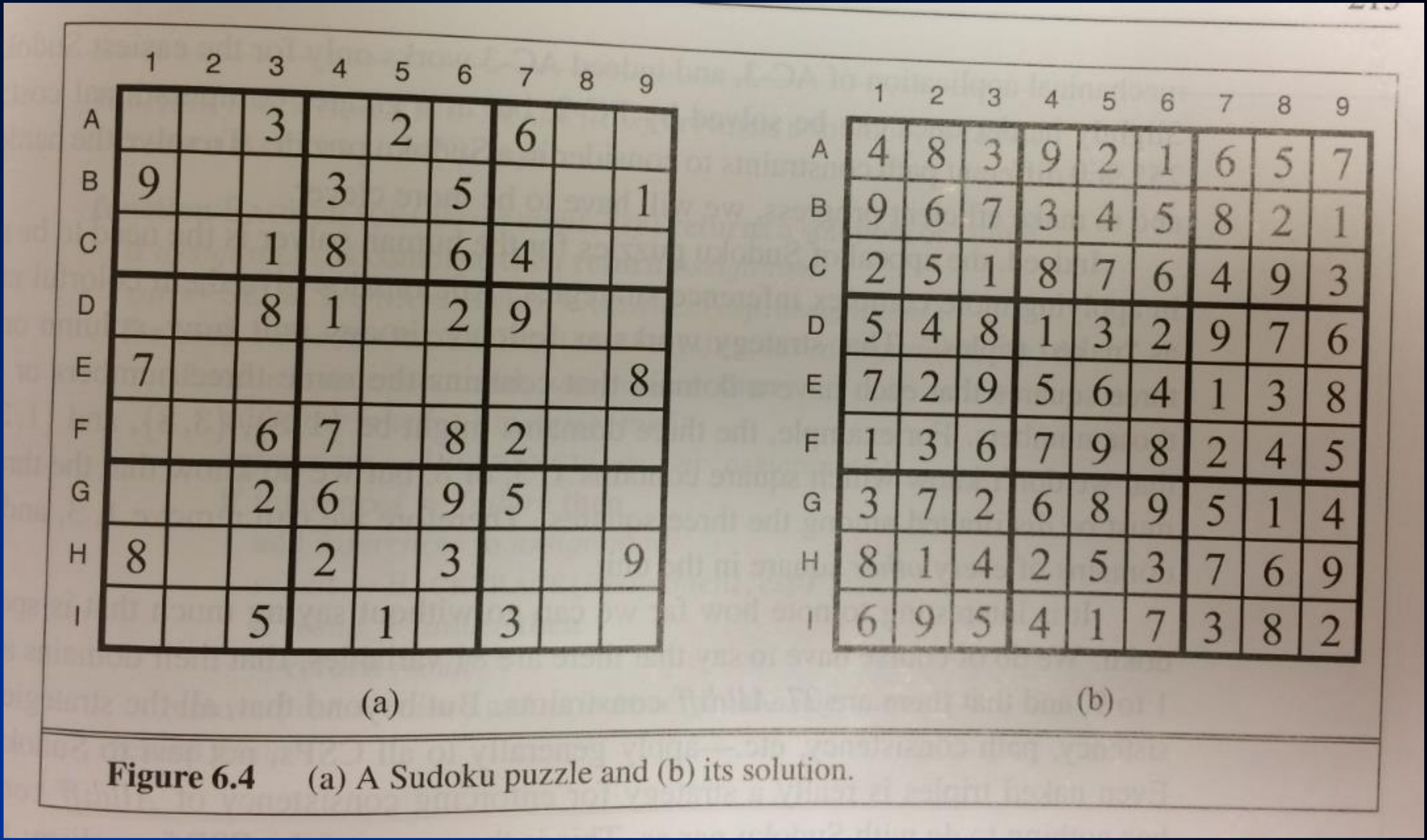
- For large resource-limited problems with integer values—such as logistical problems involving moving thousands of people in hundreds of vehicles—it is usually not possible to represent the domain of each variable as a large set of integers and gradually reduce that set by consistency-checking methods. Instead, domains are represented by upper and lower bounds and ar emanaged 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 number of passengers on each flight are then D1 = [0,165] and D2 = [0,385]. Now suppose we have the additional constraint that two flights must carry 420 people: F1 + F2 = 420. Propagating bound constraints, we reduce the domains to: D1=[35,165] and D2=[255,385].

Global constraints

- We say that a CSP is **bound 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 81squares, 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 3x3 box. 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 or 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*(A4,A5,A6,B4,B5,B6,C4,C5,C6)
- *Alldiff*(A4,A5,A6,B4,B5,B6,C4,C5,C6)
- …

Sudoku example

- Let us see how far arc consistency can take us. Assume that the *Alldiff* constraints have been expanded into binary constraints (such as A1 != A2) so that we can apply the AC-3 algorithm directly. Consider the variable E6—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. This leaves E6 with a domain of {4}; in other words, we know the answer for E6. Now consider I6. 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. 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.

Sudoku example

- Of course, Sudoku would soon lose its appeal of 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.

Sudoku example

- Indeed, the appeal of Sudoku puzzles for the human solver is the need to be resourceful in applying more complex inference strategies. Aficionados give them colorful names, such as “naked triples.” That 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. For example, the three domains might be {1,8}, {3,8}, and {1,3,8}. From that we don’t know which square contains 1, 3, or 8, but we do know that the three numbers must be distributed among the three squares. Therefore we can remove 1, 3, and 8 from the domains of every *other* square in the unit.

Sudoku example

- 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 *per se*. 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.

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; next we will look at local search algorithms over complete assignments.

Backtracking search for CSPs

- We could apply standard depth-limited search. A state would be a partial assignment, and an action would be adding *var = value* to the assignment. But for a CSP with n variables of domain size d, we quickly notice something terrible: the branching factor at the top level is *nd* because any of *d* values can be assigned to any of *n* variables. At the next level, the branching factor is *(n-1)d*, and so on for *n* levels. We generate a tree with n!*d^n leaves, even though there are only d^n possible complete assignments!

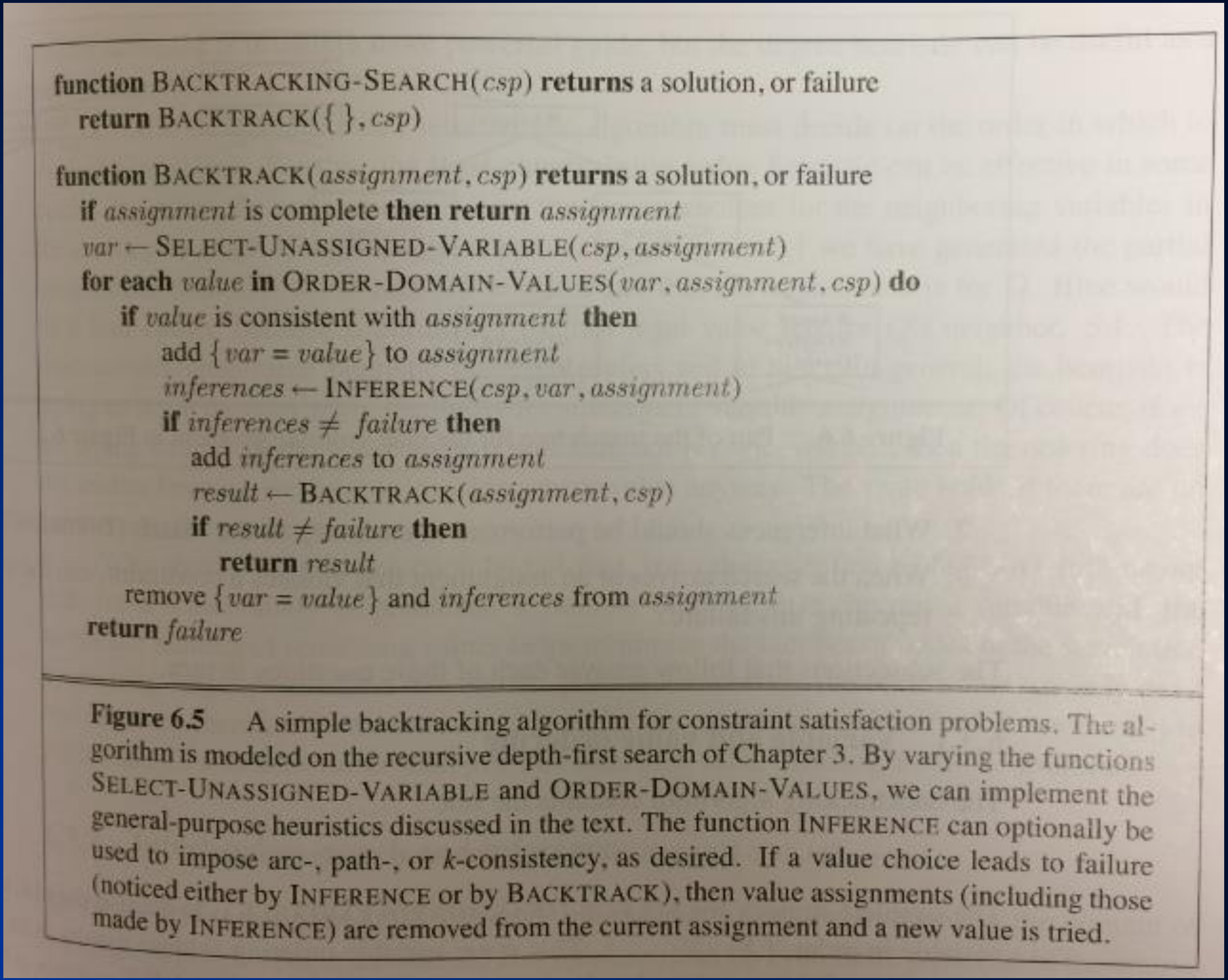
Backtracking search for CSPs

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

Backtracking search for CSPs

- 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 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, where we have assigned variables in the order WA, NT, Q,… Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

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, assignment)
  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, assignment)
        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.

Backtracking search for CSPs

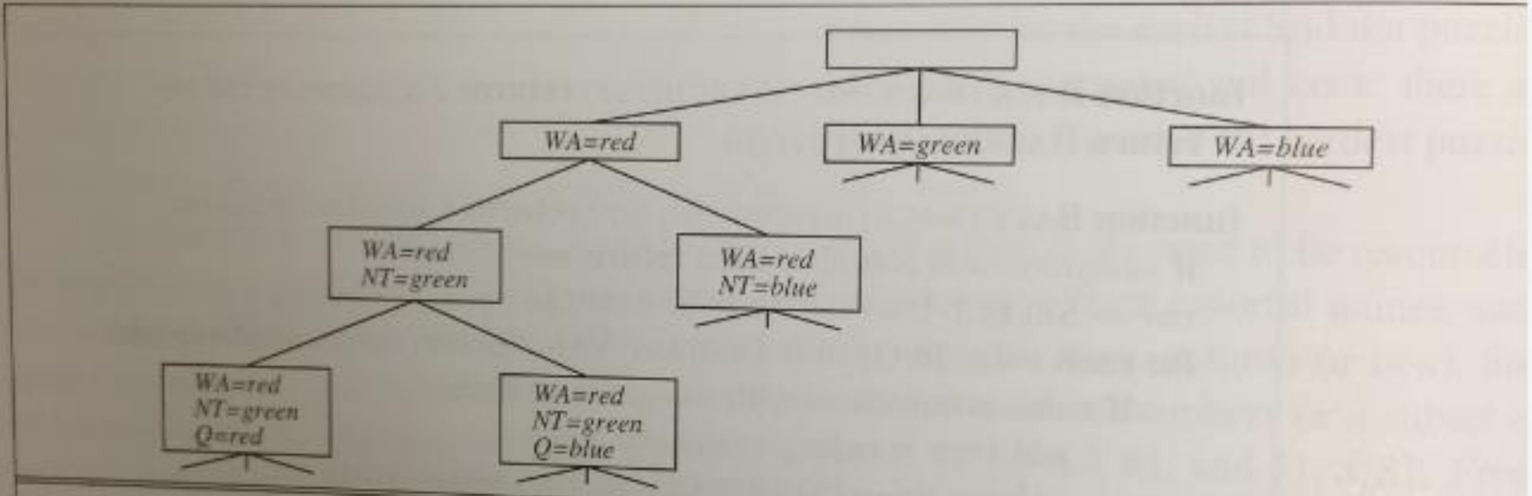


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

Backtracking search for CSPs

- Previously we improved 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, using them to answer 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, assignment)
- The simplest strategy 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, there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all *forced*. This intuitive idea—choosing the variable with 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.

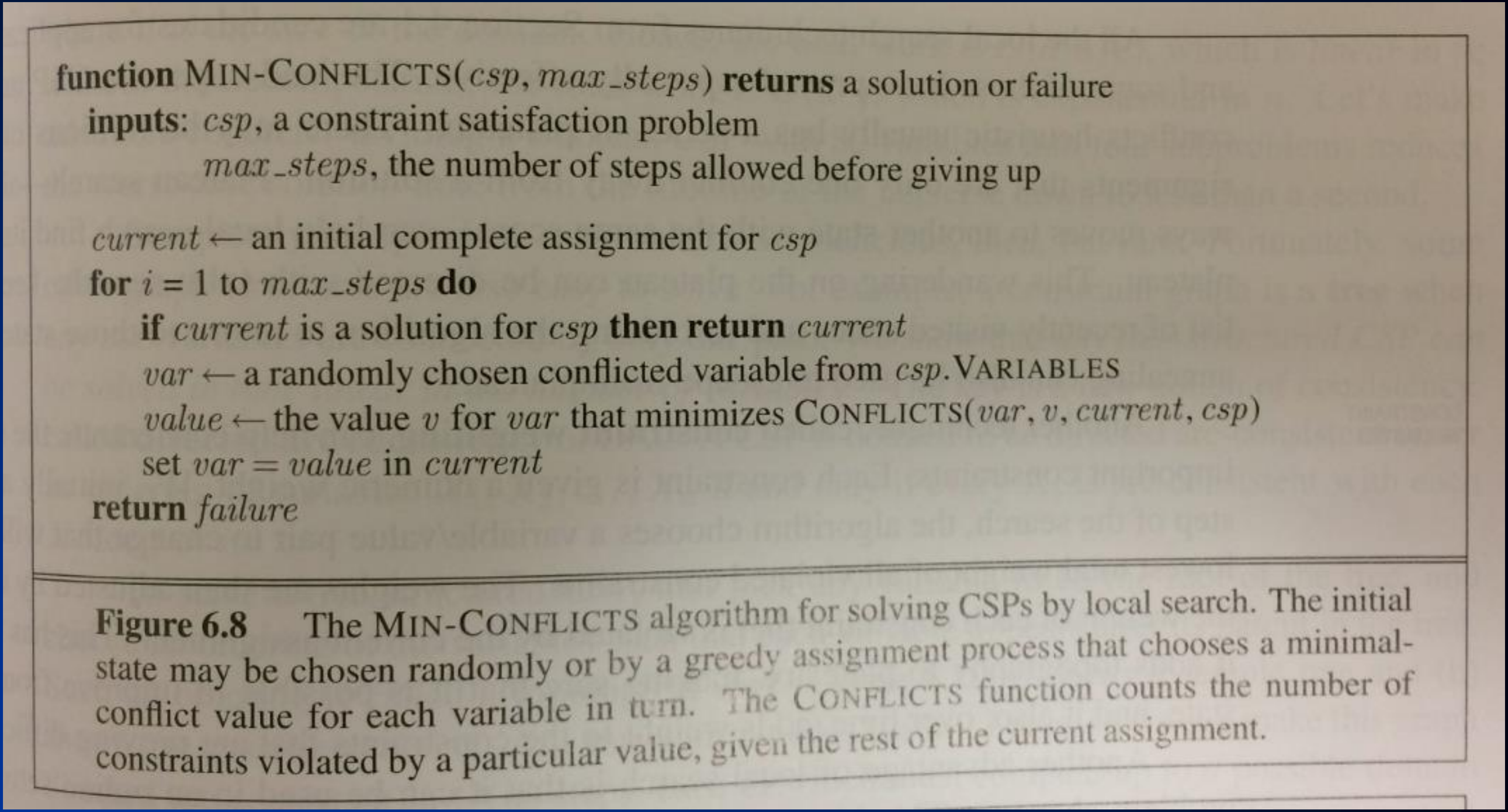
Variable and value ordering

- 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.

Local search for CSPs

- Local search algorithms 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, 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.

Local search – min conflicts



```
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 – min conflicts

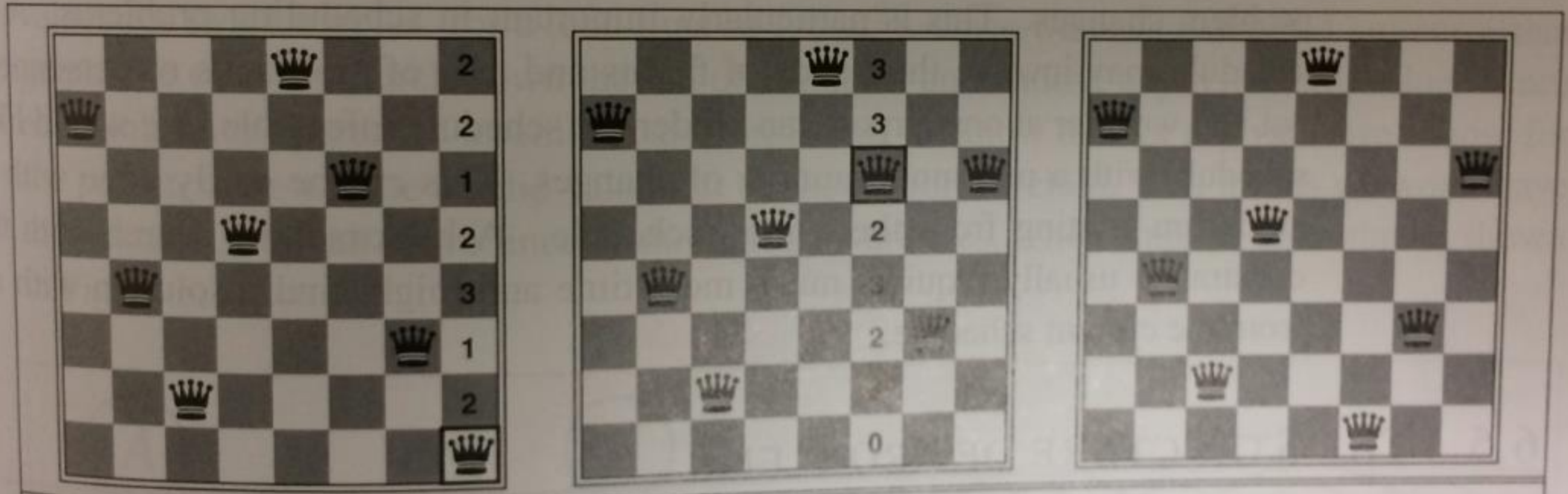
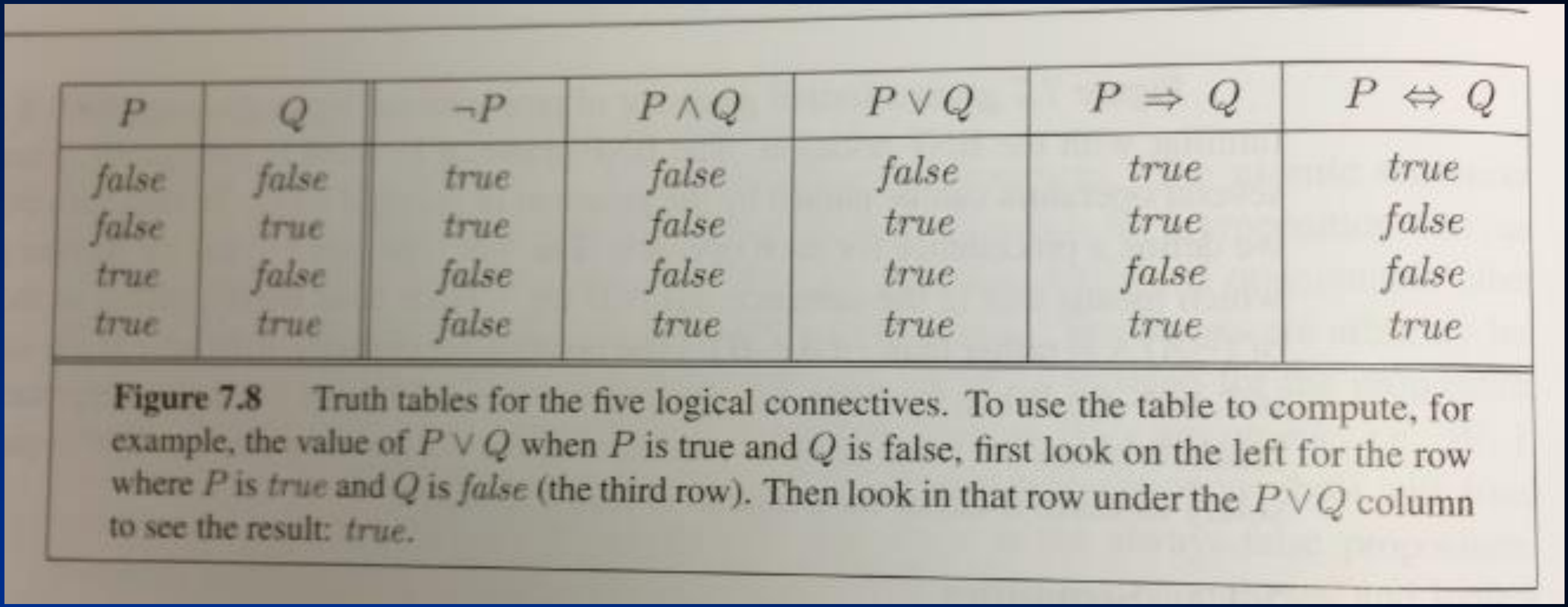


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking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

CSP summary

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

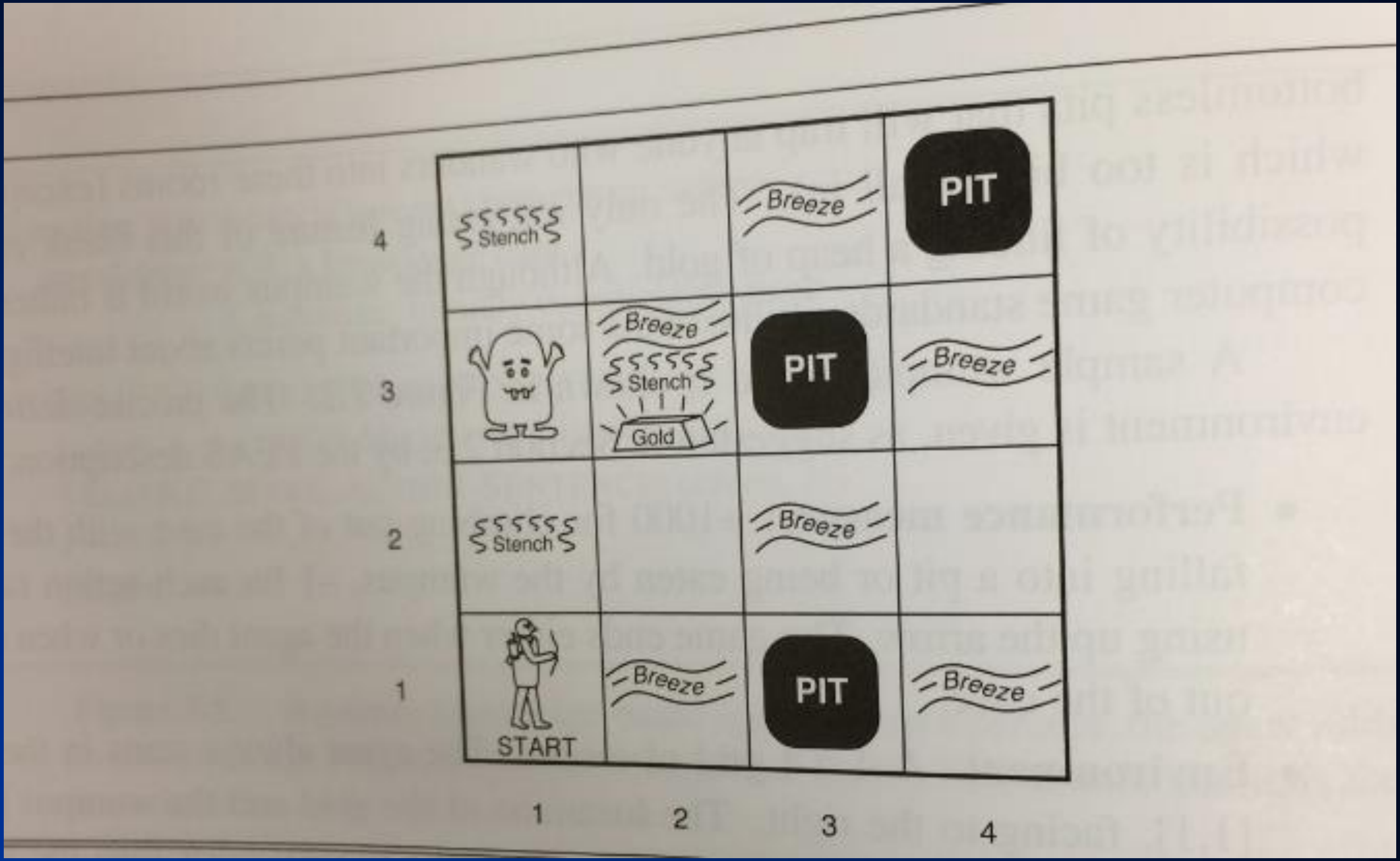
Propositional logic



P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

Wumpus world



Wumpus world

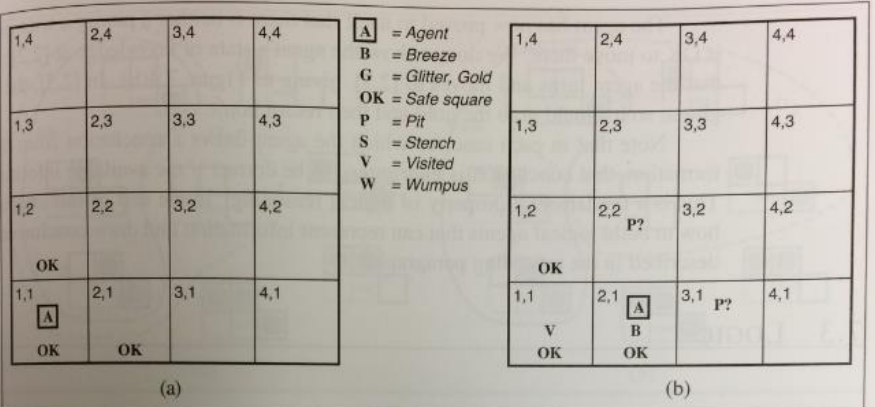


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None, None, None, None, None*]. (b) After one move, with percept [*None, Breeze, None, None, None*].

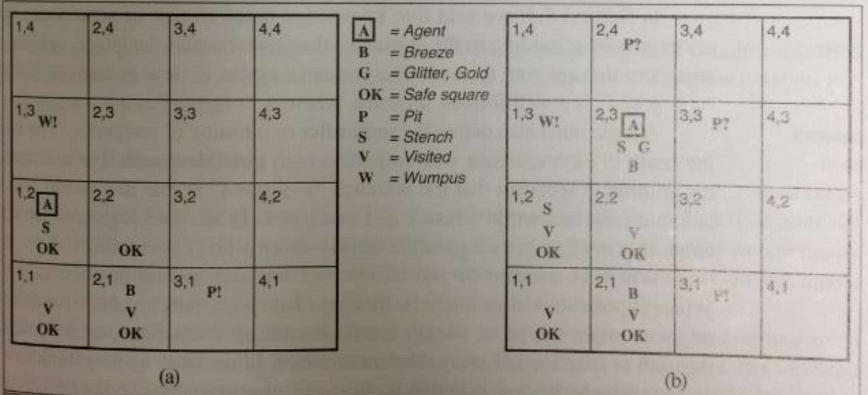
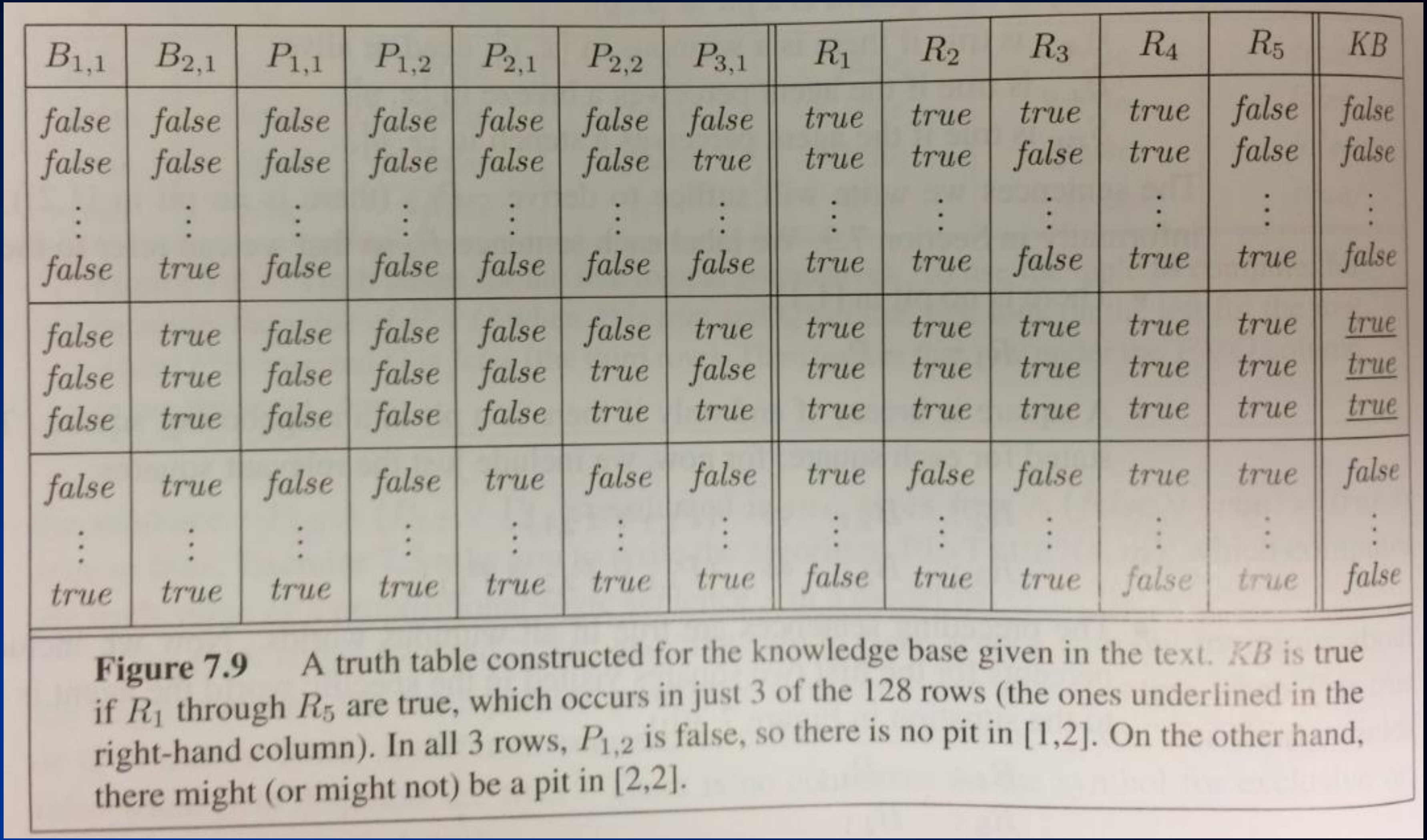


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench, None, None, None, None*]. (b) After the fifth move, with percept [*Stench, Breeze, Glitter, None, None*].

Wumpus world



$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	$\underline{\mathit{true}}$
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	$\underline{\mathit{true}}$
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	$\underline{\mathit{true}}$
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. KB is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

Homework for next class

- Chapters 9-10 from Jensen textbook.
- HW1: out 9/5 was due on 10/3
- HW2: out today due 10/17