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 last week, due 10/17
- Midterm on 10/19
  - Covering search (uninformed, informed, local, adversarial, CSP), logic, and optimization
  - Review during half of class on 10/17

Upcoming lectures

- 10/5: Continue CSP
- 10/10: Wrap up CSP, start logic (propositional logic, first-order logic)
- 10/12: Wrap-up logic (logical inference), start 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.
  - Possible class cancellations on 10/24 and 11/7. May have TA review exams and homeworks during these lectures.

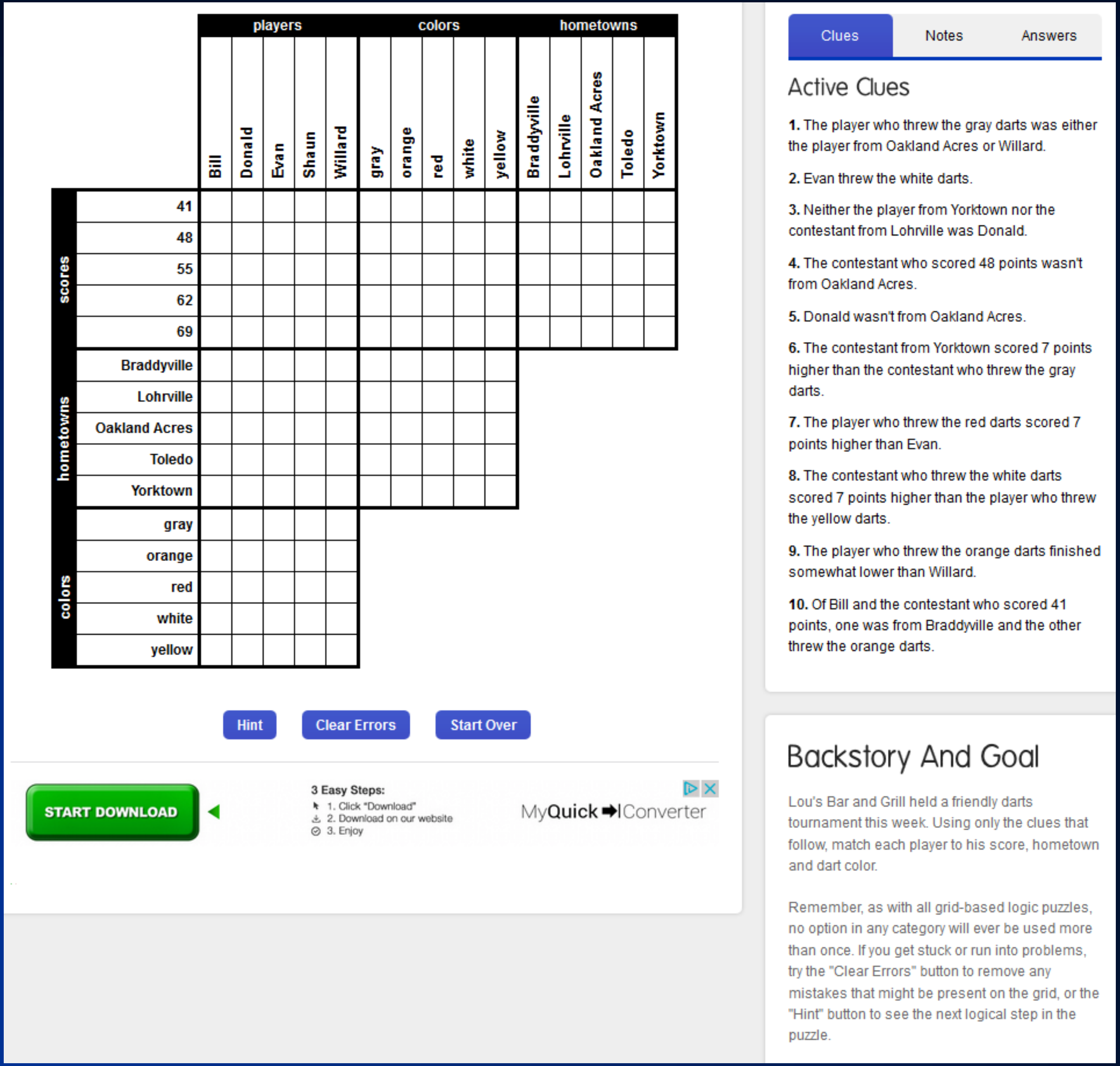
HW1

- Will be back before midterm
- Received 29 on moodle (33 students enrolled)
- Will be lenient regarding late days for HW1 due to the hurricane

HW2

- Out last week 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.
- http://www.logic-puzzles.org/

Quizzle



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.

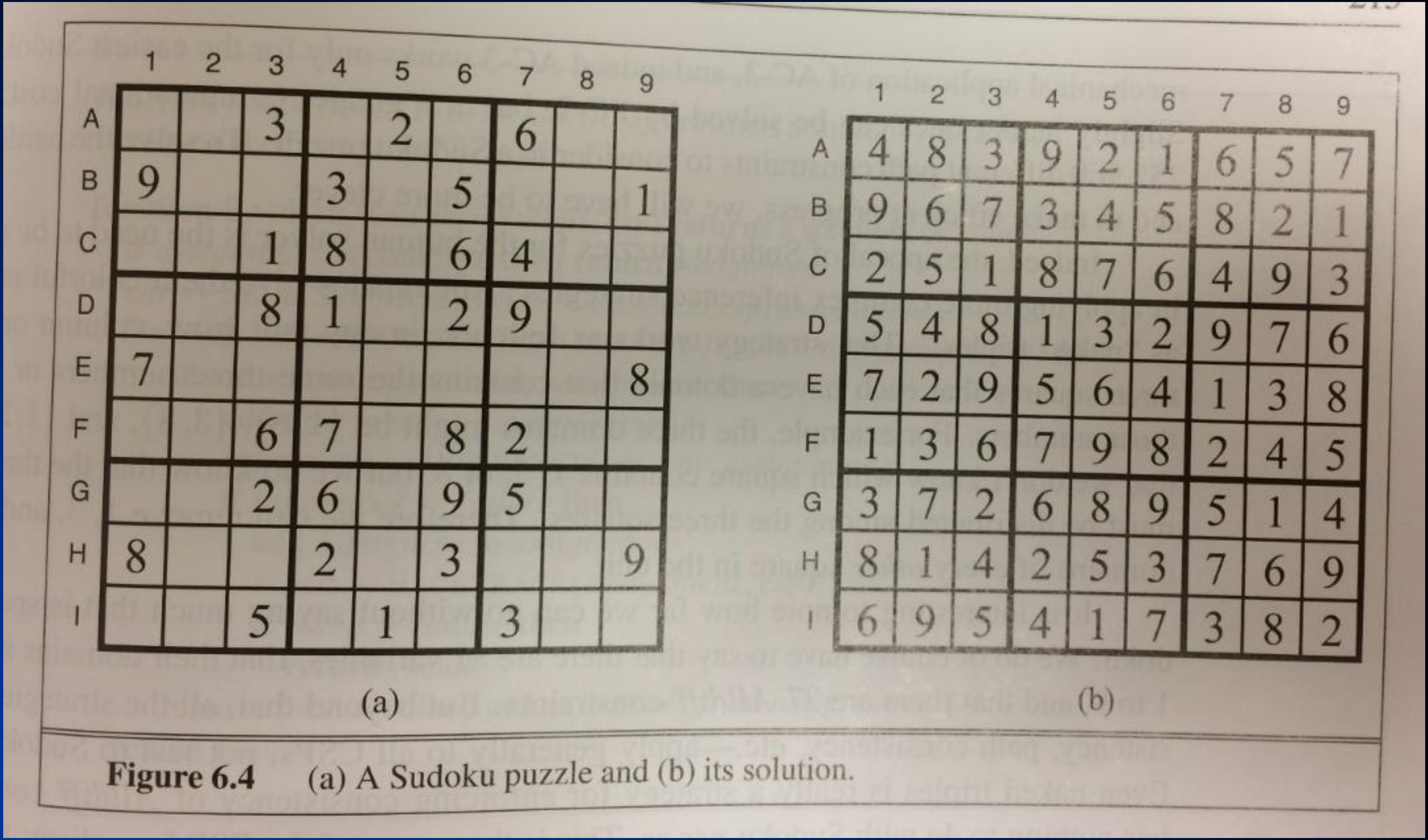
Backstory And Goal

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.

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!

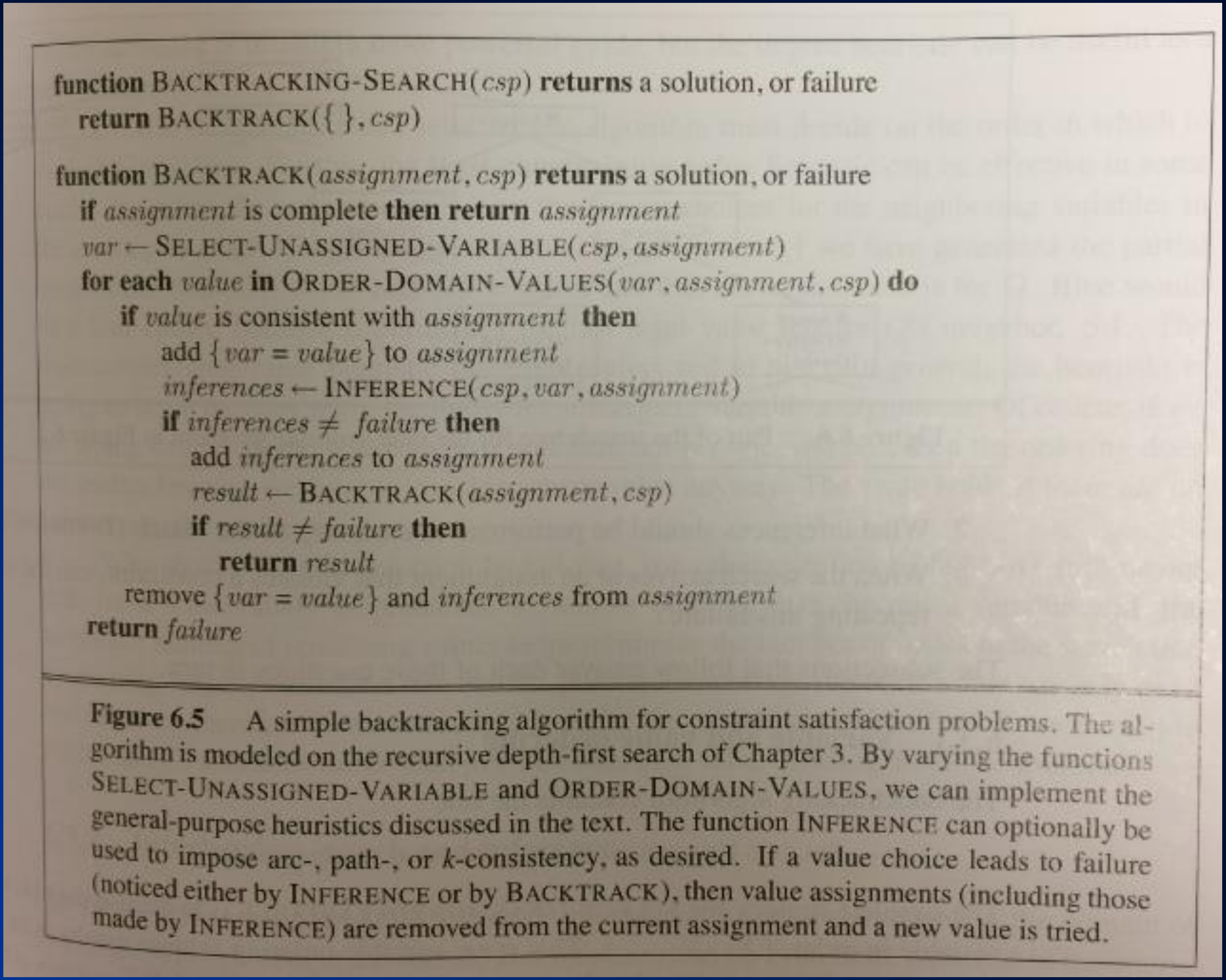
Backtracking search for CSPs

- 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

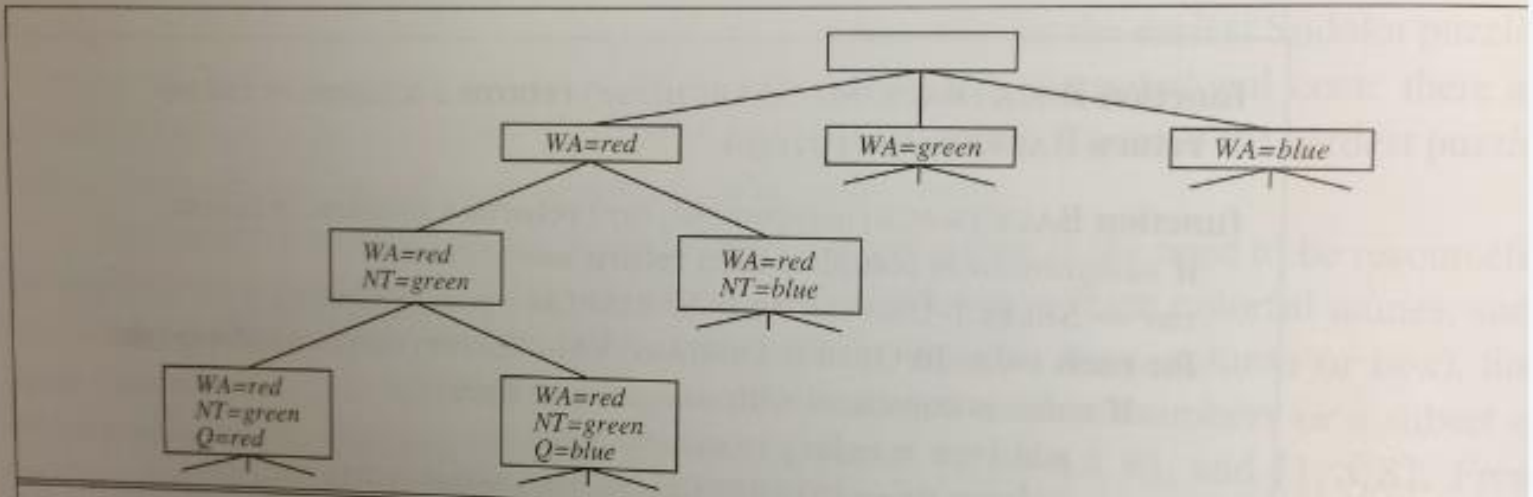


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

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.

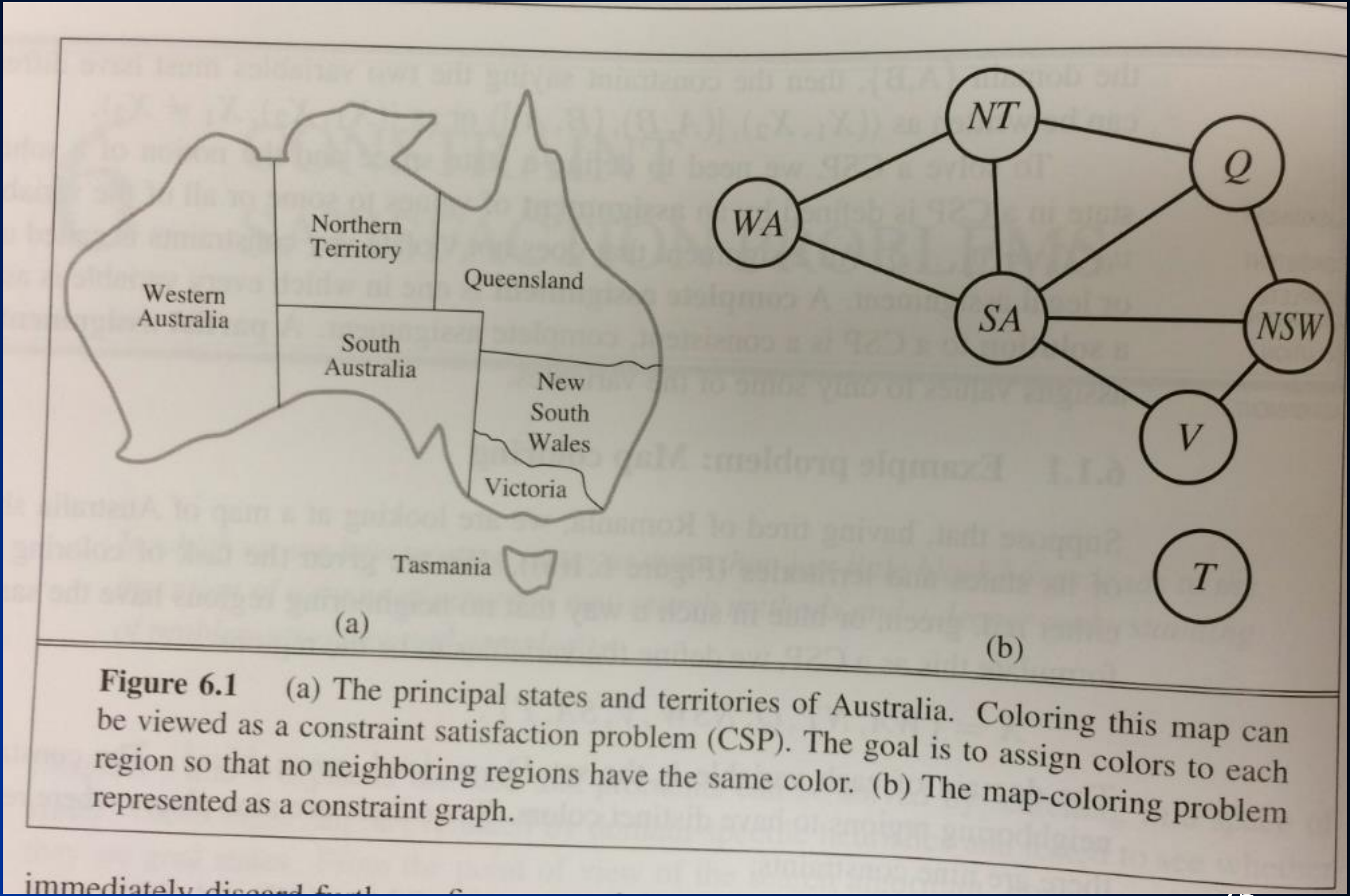
Variable and value ordering

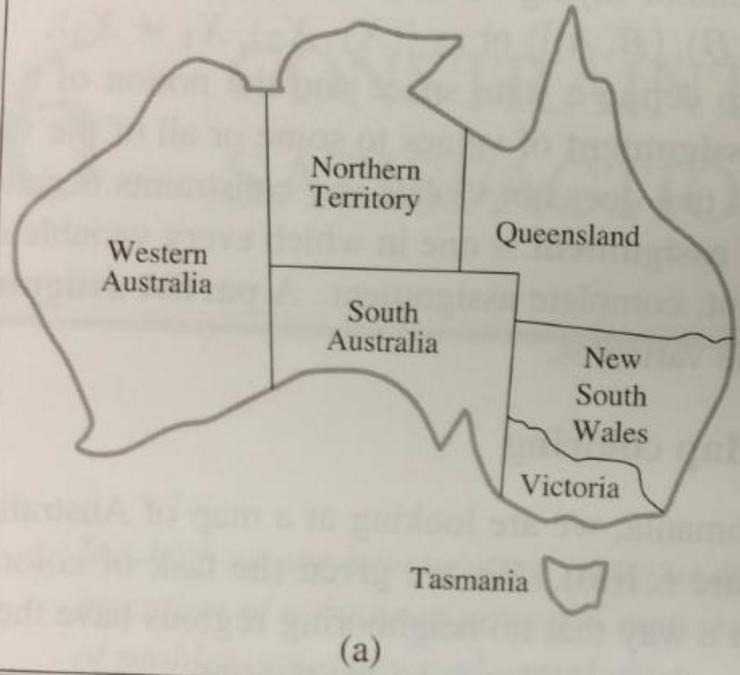
- If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables. The MRV heuristic usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

Variable and value ordering

- The MRV heuristic doesn't help at all in choosing the first region to color in Australia, because initially every region has three legal colors. In this case, the **degree heuristic** comes in handy. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. For the Australia map, 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.

Constraint graph





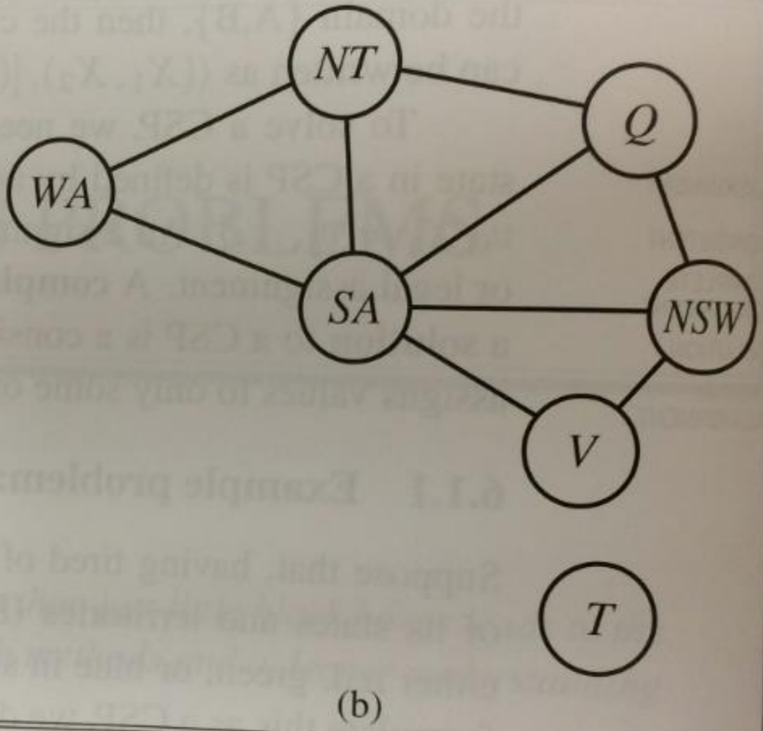


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Variable and value ordering

- 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 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.

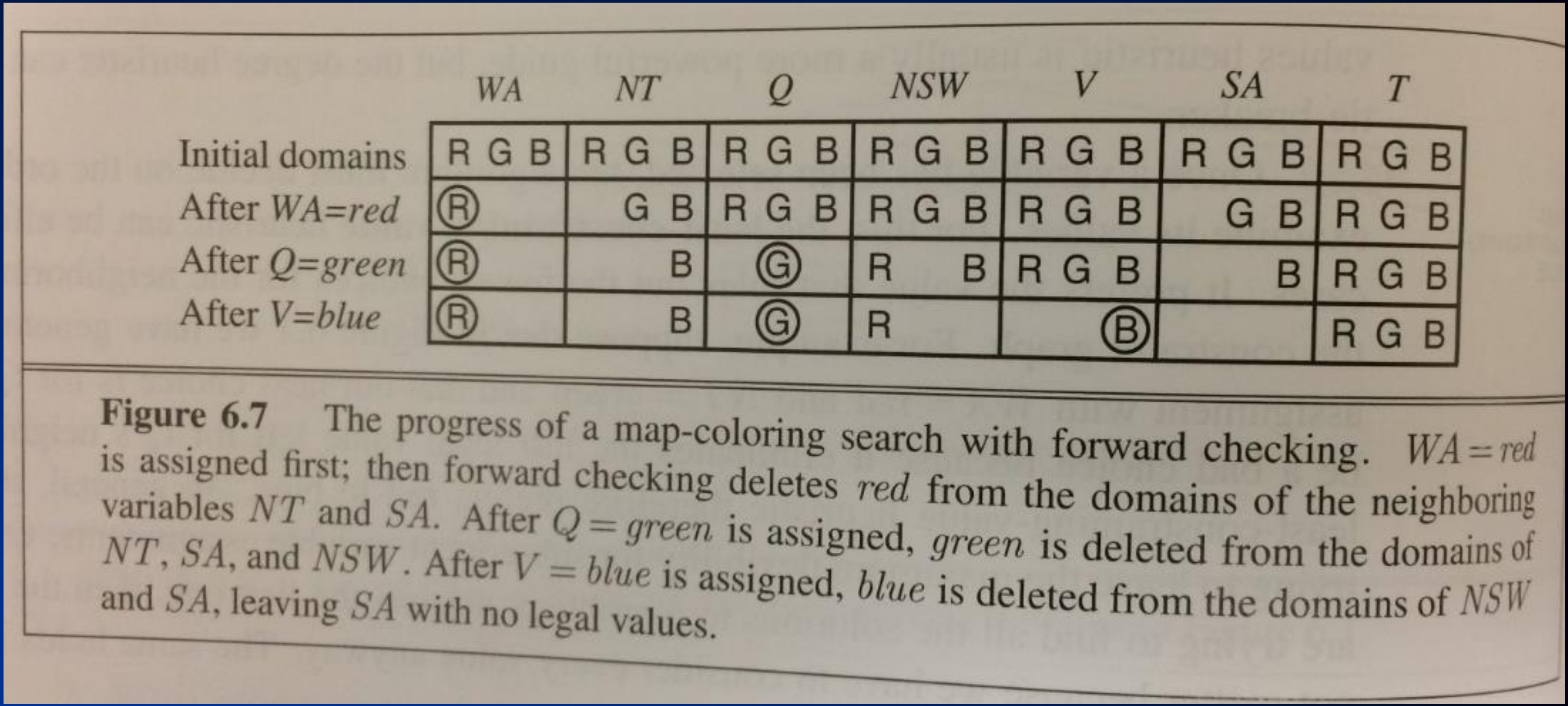
Interleaving search and inference

- We have seen how AC-3 can infer reductions in the domain of variables *before* 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 procedure 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

- Figure 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.

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

Interleaving search and inference

- For many problems the search will be more effective if we combine the MRV heuristic with forward checking. Consider after assigning {WA=red}. Intuitively, it seems that that assignment constrains its neighbors, NT and SA, so we should handle those variables next, and then all the other variables will fall into place. That’s exactly what happens with MRV: NT and SA have two values, so one of them is chosen first, then the other, then Q, NSW, and V in order. Finally T still has three values, and any one of them works. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

Interleaving search and inference

- 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. 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.

Interleaving search and inference

- The algorithm called MAC (**Maintaining Arc Consistency**) 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.

Intelligent backtracking

- The BACKTRACKING-SEARCH algorithm has a very simple policy for what to do when a branch of the search fails: back up the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited.
- Consider what happens when we apply this with a fixed variable ordering Q, NSW, V, T, SA, WA, NT. Suppose we have generate the partial assignment {Q = red, NSW = green, V=blue, T=red}. When we try the next variable SA, we see that every value violates a constraint. We back up to T and try a new color for Tasmania! Obviously this is silly—recoloring Tasmania cannot possibly resolve the problem with South Australia.

Intelligent backtracking

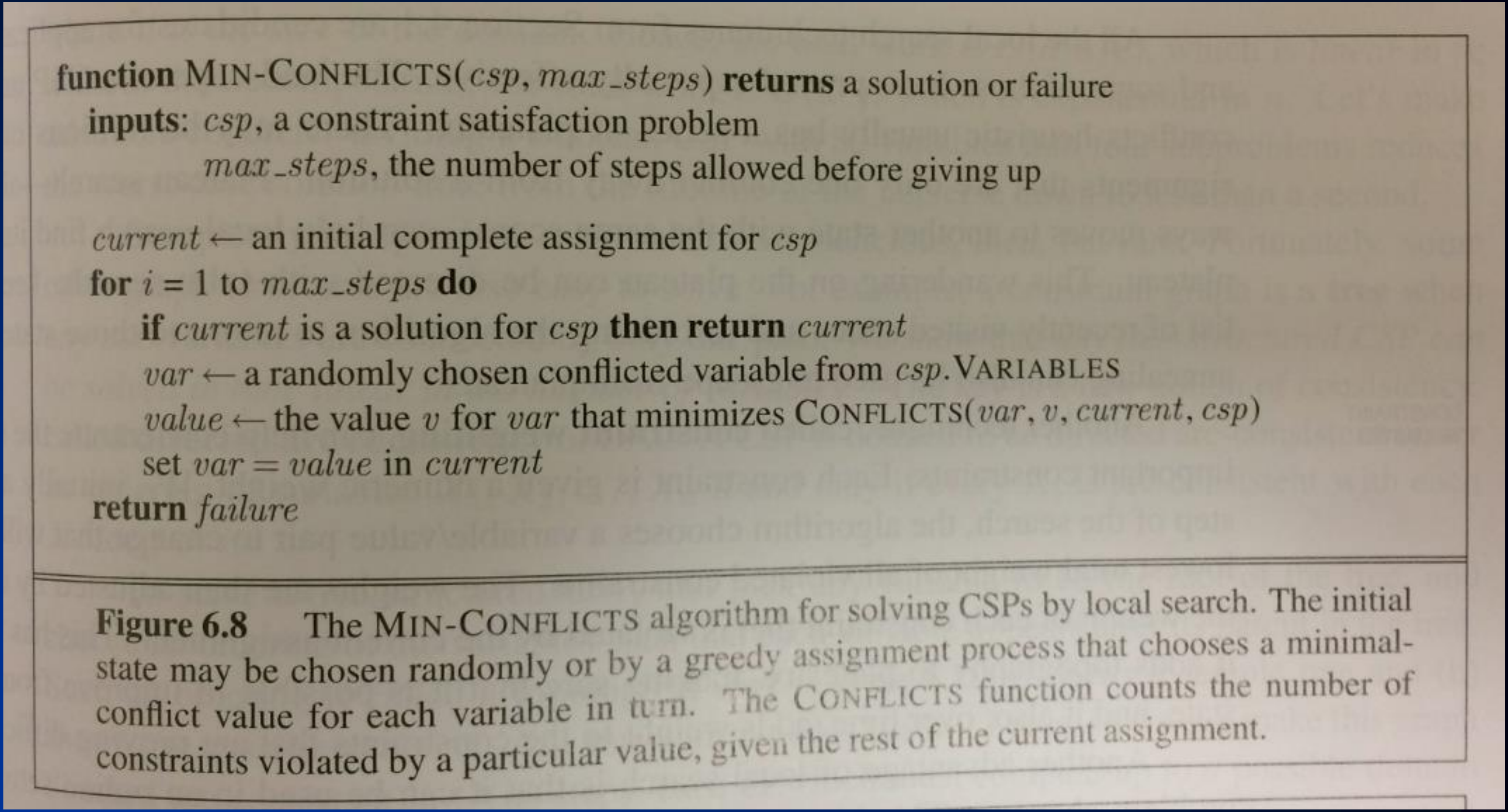
- A more intelligent approach to backtracking is to backtrack to a variable that might fix the problem—a variable that was responsible for making one of the possible values of SA impossible. To do this, we will keep track of a set of assignments that are in conflict with some value for SA. The set (in this case {Q=red, NSW=green, V =blue}) is called the **conflict set** for SA. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, 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.

- Consider again the partial assignment {WA=red,NSW=red} (which is inconsistent). Suppose we try T=red next and then assign NT, Q, V, SA, We know that no assignment can work for these last four variables, so eventually we run out of values to try at NT. Now the question is, where to backtrack? Backjumping cannot work, because NT *does* have values consistent with the preceding assigned variables-NT doesn’t have a complete conflict set of preceding variables that caused it to fail. We know, however, that the four variables NT, Q, V, and SA, *taken together*, failed because of a set of preceding variables, which must be those variables that directly conflict with the four. This leads to a deeper notion of the conflict set for a variable such as NT: it is that set of preceding variables that caused NT, *together with any subsequent variables*, to have no consistent solution. In this case, the set is WA and NSW, so the algorithm should backtrack to NSW and skip over Tasmania. A backjumping algorithm that uses conflict sets defined in this way is called **conflict-directed backjumping**.

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. 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.

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

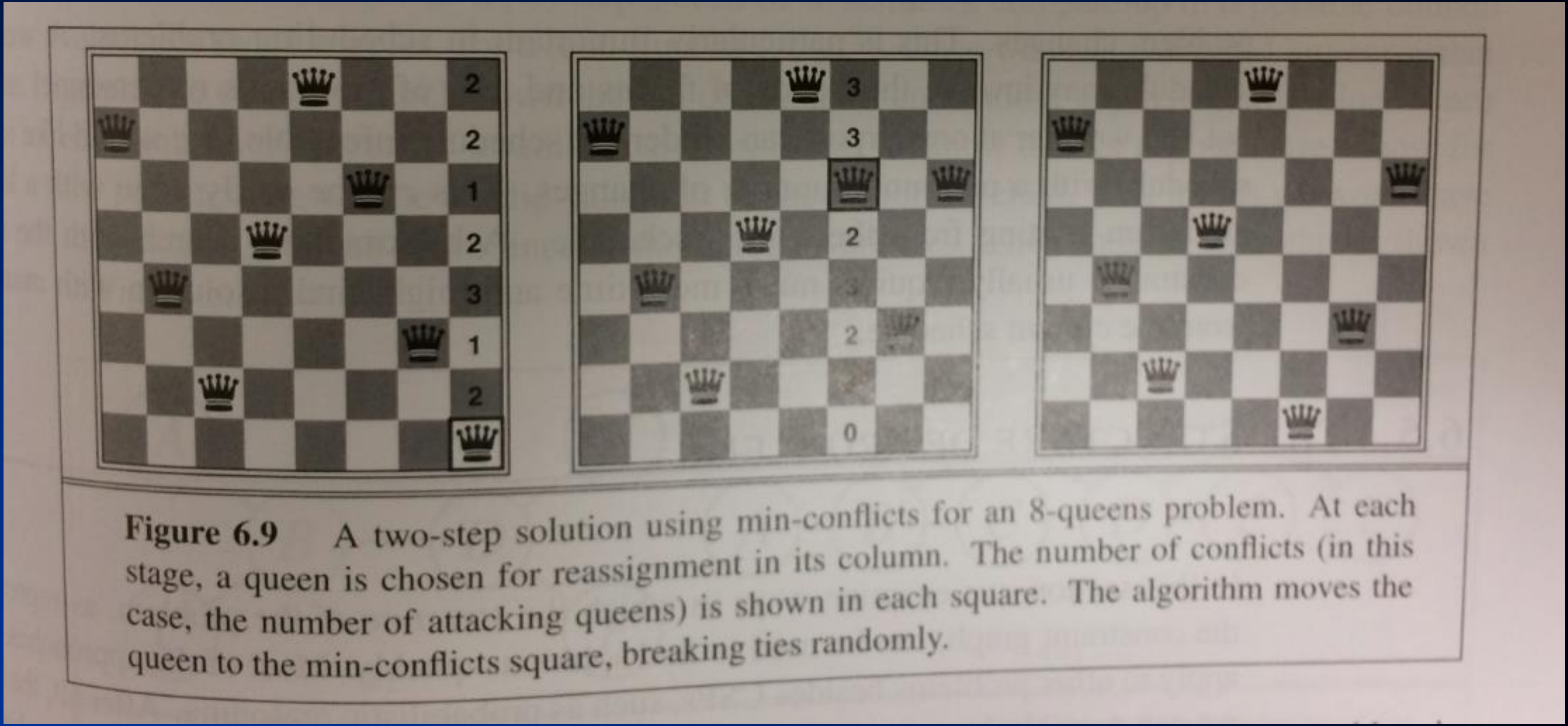


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking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

Local search for CSPs

- 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. 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.

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.

Logical agents

- The problem-solving (search) agents “know things,” but only in a very limited, inflexible sense. For example, the transition model for the 8-puzzle—knowledge of what the actions do—is hidden inside the domain-specific code of the RESULT function. It can be used to predict the outcome of actions but not to deduce that two tiles cannot occupy the same space or that states with odd parity cannot be reached from states with even parity, etc. The atomic representations used by problem-solving agents are also very limiting. In a partially observable environment, an agent’s only choice for representing what it knows about the current state is to list all possible concrete states—a hopeless prospect in large environments.

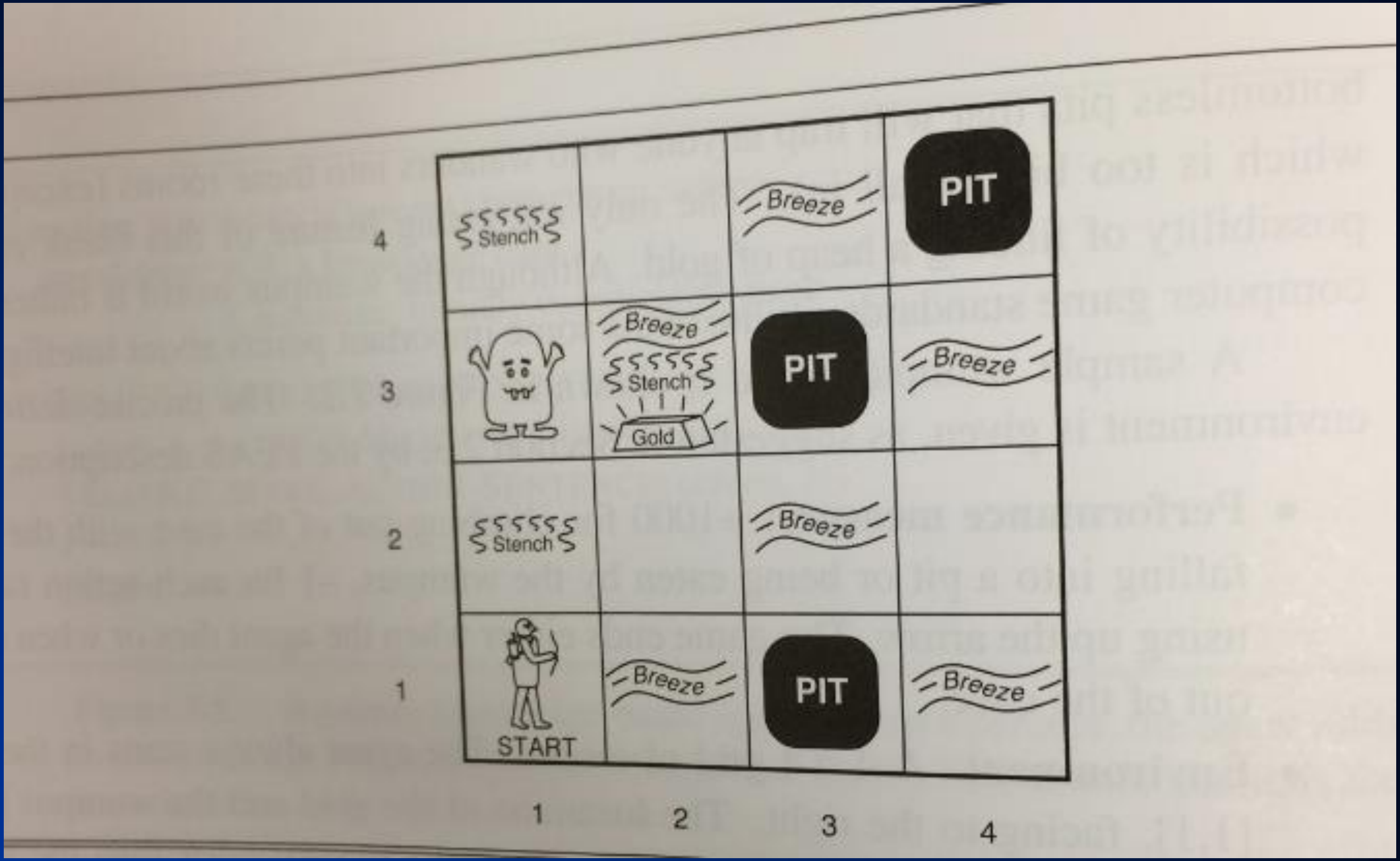
Logical agents

- Constraint satisfaction introduced the idea of representing states as assignments of values to variables; this is a step in the right direction, enabling some parts of the agent to work in a domain-independent way and allowing for more efficient algorithms. We now take this step to its logical conclusion—we develop **logic** as a general class of representations to support knowledge-based agents. Such agents can combine and recombine information to suit myriad purposes. Often this process can be quite far removed from the needs of the moment—as when a mathematician proves a theorem or an astronomer calculates the earth’s life expectancy. Knowledge-based agents can accept new tasks in the form of explicitly-described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge.

Logical agents

- The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

Wumpus world



Wumpus world

- **Performance measure**: +1000 for climbing out of the cave with the gold, -1000 for falling into a pit or being eaten by the wumpus, -1 for each action taken and -10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment**: A 4x4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

Wumpus world

- **Actuators**: The agent can move *Forward*, *TurnLeft* by 90 degrees, or *TurnRight* by 90 degrees. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].

Wumpus world

- **Sensors**: The agent has five sensors, each of which gives a single bit of information:
  - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
  - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
  - In the square where the goal is, the agent will perceive a *Glitter*.
  - When an agent walks into a wall, it will perceive a *Bump*.
  - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.
- The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench*, *Breeze*, *None*, *None*, *None*].

Wumpus world

- Consider a knowledge-based wumpus agent exploring the environment in the next figure. We use an informal knowledge representation language consisting of writing down symbols in a grid. The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively in square [1,1].
- The first percept is [*None*,*None*,*None*,*None*,*None*], from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK.

Wumpus world

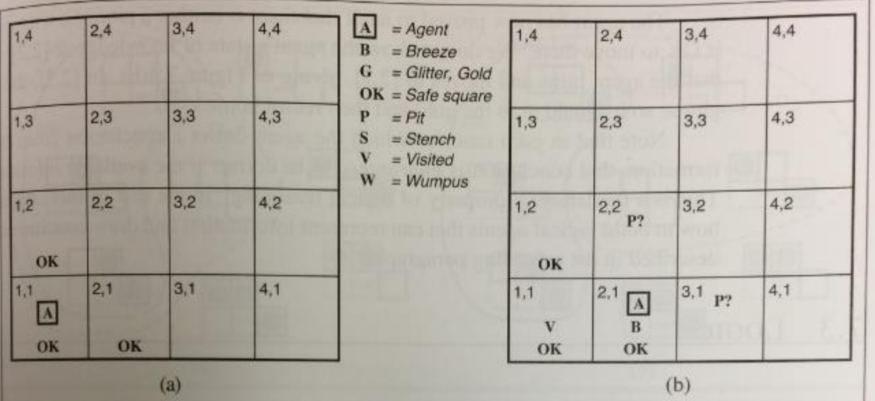


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None, None, None, None, None*]. (b) After one move, with percept [*None, Breeze, None, None, None*].

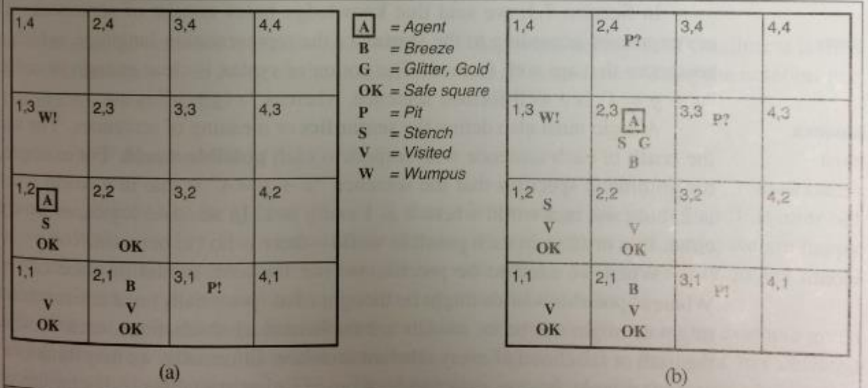


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench, None, None, None, None*]. (b) After the fifth move, with percept [*Stench, Breeze, Glitter, None, None*].

Wumpus world

- A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by “B”) in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation “P?” indicates a possible pit in those squares. At this point, there is only one known square that is OK and that as not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

Wumpus world

- The agent perceives a stench in [1,2], resulting in the state of knowledge shown in 7.4a. The stench in [1,2], means that there must be a wumpus nearby. But the wumpus cannot be in [1,1[, by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

Wumpus world

- The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us 74b. In [2,3], the agent detects a glitter, so it should grab the gold and then return home.
- Note that in each case for which the agent draws a conclusion from the available information, that conclusion is *guaranteed* to be correct if the available information is correct. This is a fundamental property of logical reasoning.

Logic

- Consider the situation in 7.3b: the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent’s knowledge of the rules of the wumpus world, constitute the **knowledge base** (KB). The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are 2^3=8 possible **models**. These eight models are shown in 7.5.

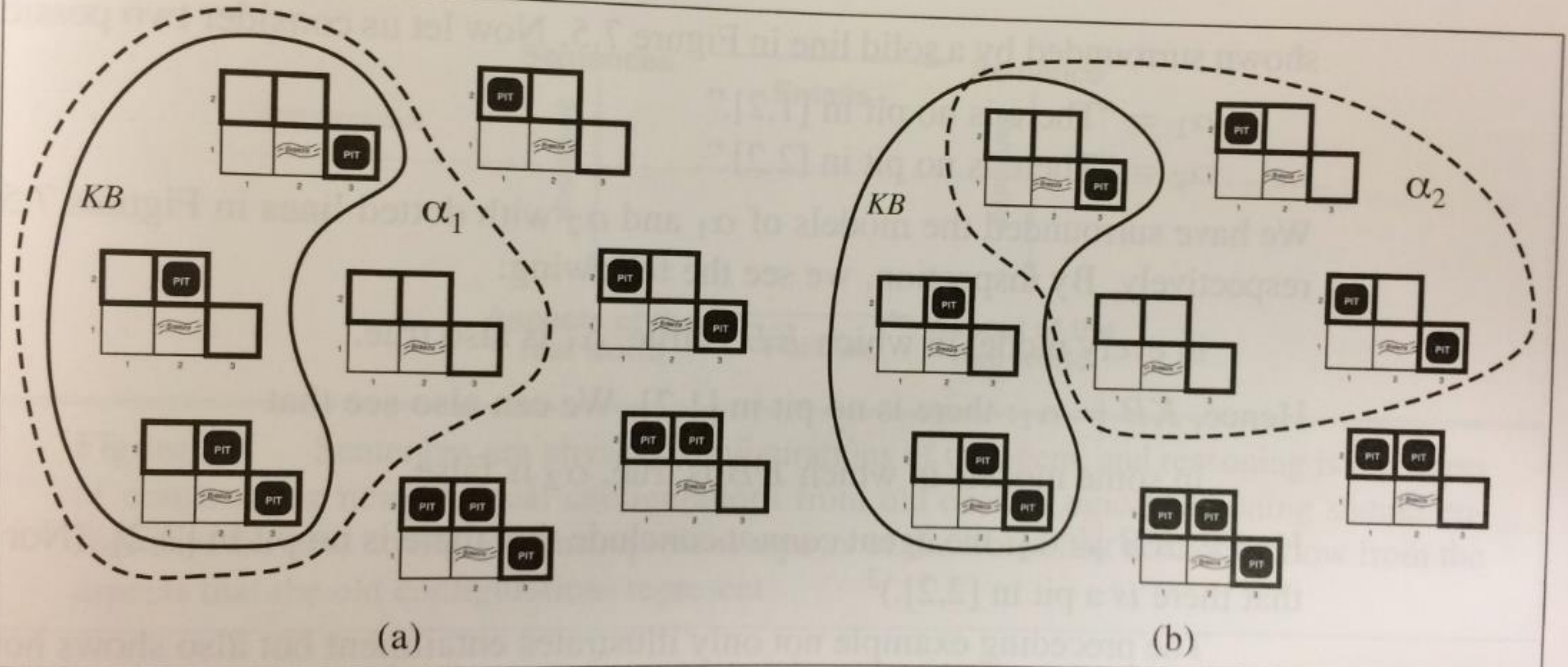


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

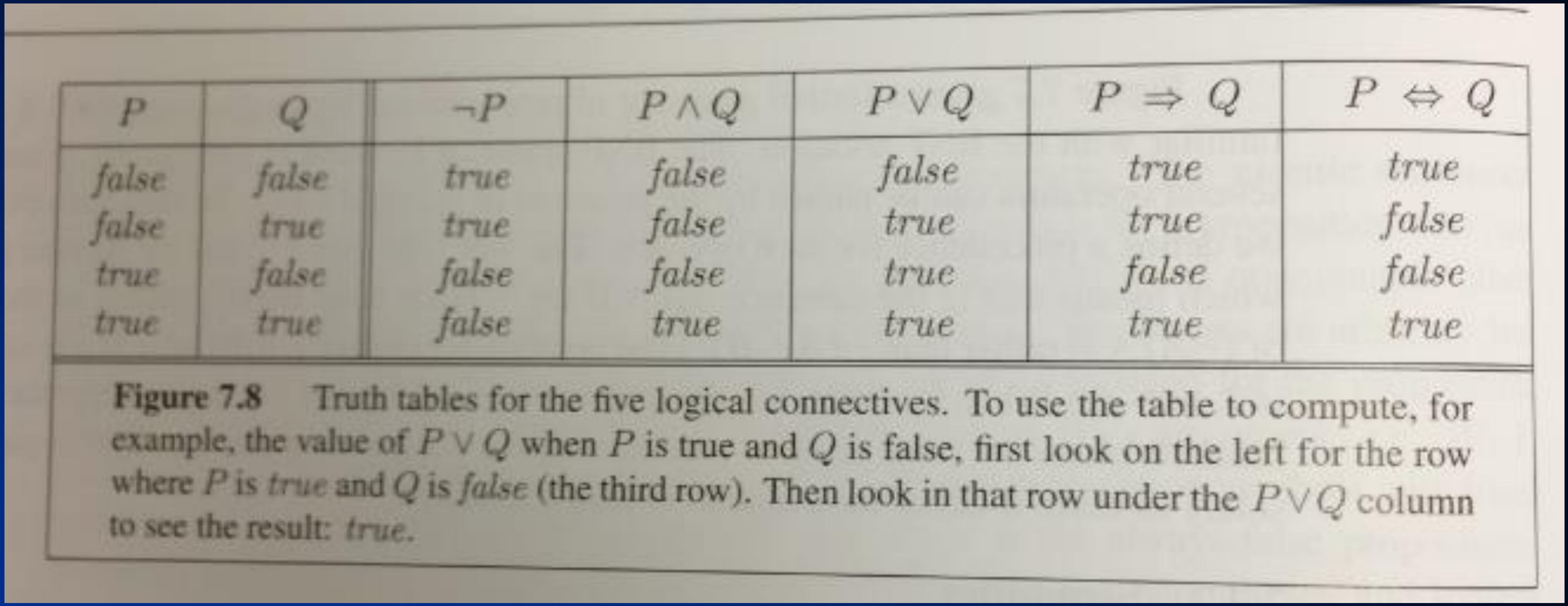
Logical agents

- The KB can be thought of a set of **sentences** or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows—for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are shown surrounded by a solid line in 7.5. Now let us consider two possible conclusions:
  - A1 = "There is no pit in [1,2]"
  - A2 = "There is no pit in [2,2]"
- A1 and A2 are surrounded with dotted lines in 7.5a and 7.5b. By inspection, we see the following:
  - In every model in which KB is true, A1 is also true.

Logical agents

- Hence, KB |= A1; there is no pit in [1,2]. We can also see that
  - In *some* models in which KB is true, A2 is false.
- Hence, KB !|= A2; the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)

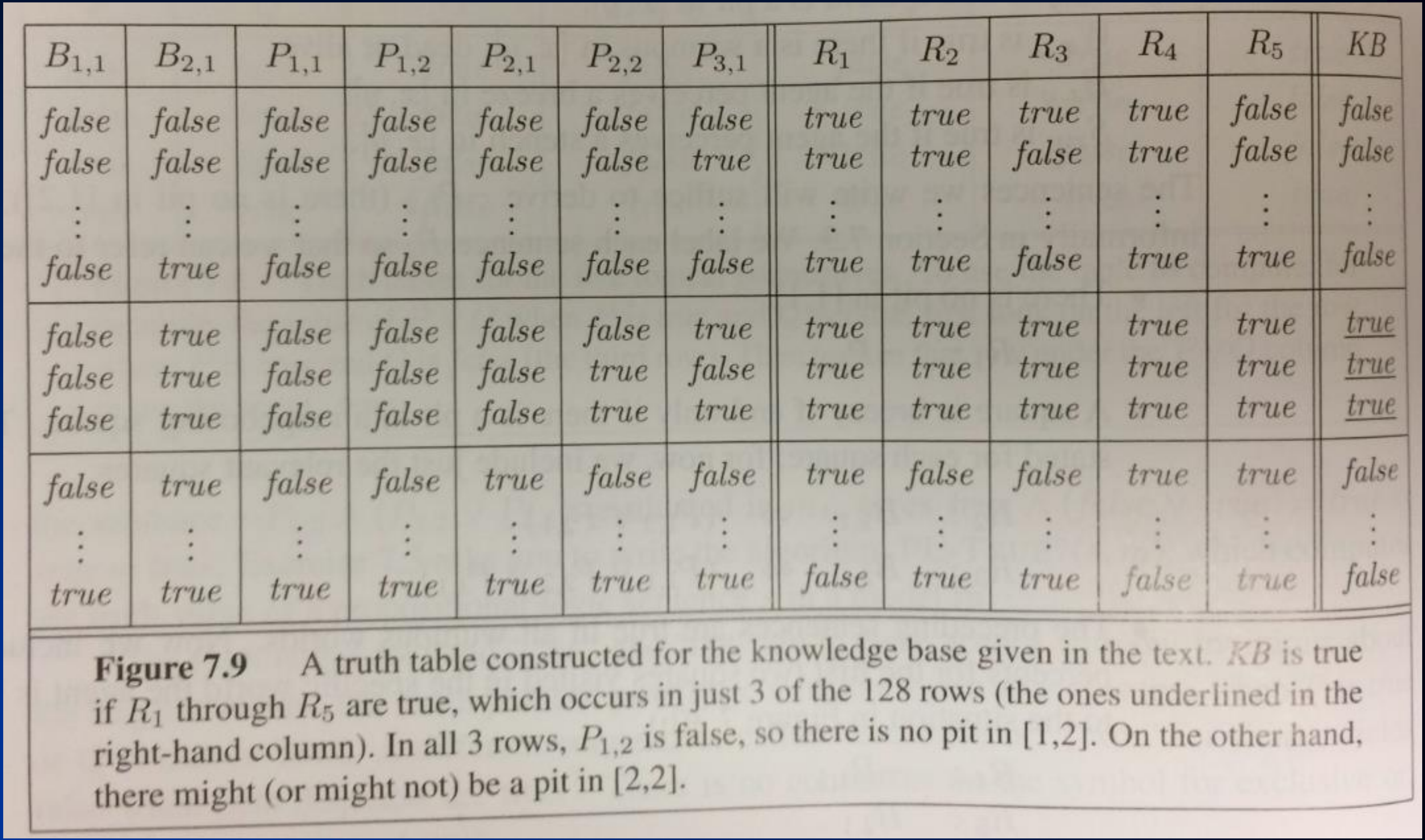
Propositional logic



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



$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*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 10 from Russel/Norvig
- HW1: out 9/5 was due on 10/3
- HW2: out last week due 10/17