Artificial Intelligence-BSCE-306L

Module 3:

Local and Adversarial Search

Dr . Saurabh Agrawal

Faculty Id: 20165

School of Computer Science and Engineering

VIT, Vellore-632014

Tamil Nadu, India

Outline

- **Local Search Algorithms**
- Hill-Climbing Search
- Simulated Annealing
- Genetic Algorithm
- **Adversarial Search**
- Game Trees
- Elementary two-players games
- Minimax Evaluation
- tic-tac-toe
- Minimax with Alpha-Beta Pruning

Local Search Algorithms

- The search algorithms that we have seen so far are designed to explore search spaces systematically.
- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path.
- When a goal is found, the path to that goal also constitutes a solution to the problem.
- In many problems, however, the path to the goal is irrelevant.
- The same general property holds for many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications, network optimization, vehicle routing, and portfolio management.

Local Search Algorithms

❑If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all.

❑**Local search algorithms operate using** a single **current node (rather than multiple paths) and generally move only to neighbors** of that node.

❑Typically, the paths followed by the search are not retained.

❑ Although locals earch algorithms are not systematic, although they have two key advantages:

1. **They use very little memory—usually a constant amount;**
2. **They can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.**

Local Search Algorithms

- In addition to finding goals, local search algorithms are useful for solving pure **optimization problems, in which the aim is to find the best state according to an objective function.**
- For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no “goal test” and no “path cost” for this problem.

Local Search Algorithms

- To understand local search, we find it useful to consider the **state-space landscape (as** in Figure).
- A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function).
- If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum; if elevation corresponds** to an objective function, then the aim is to find the highest peak—a **global maximum. (You** can convert from one to the other just by inserting a minus sign.)
- Local search algorithms explore this landscape. A **complete local search algorithm always finds a goal if one exists;** an **optimal algorithm always finds a global minimum/maximum.**

Local Search Algorithms

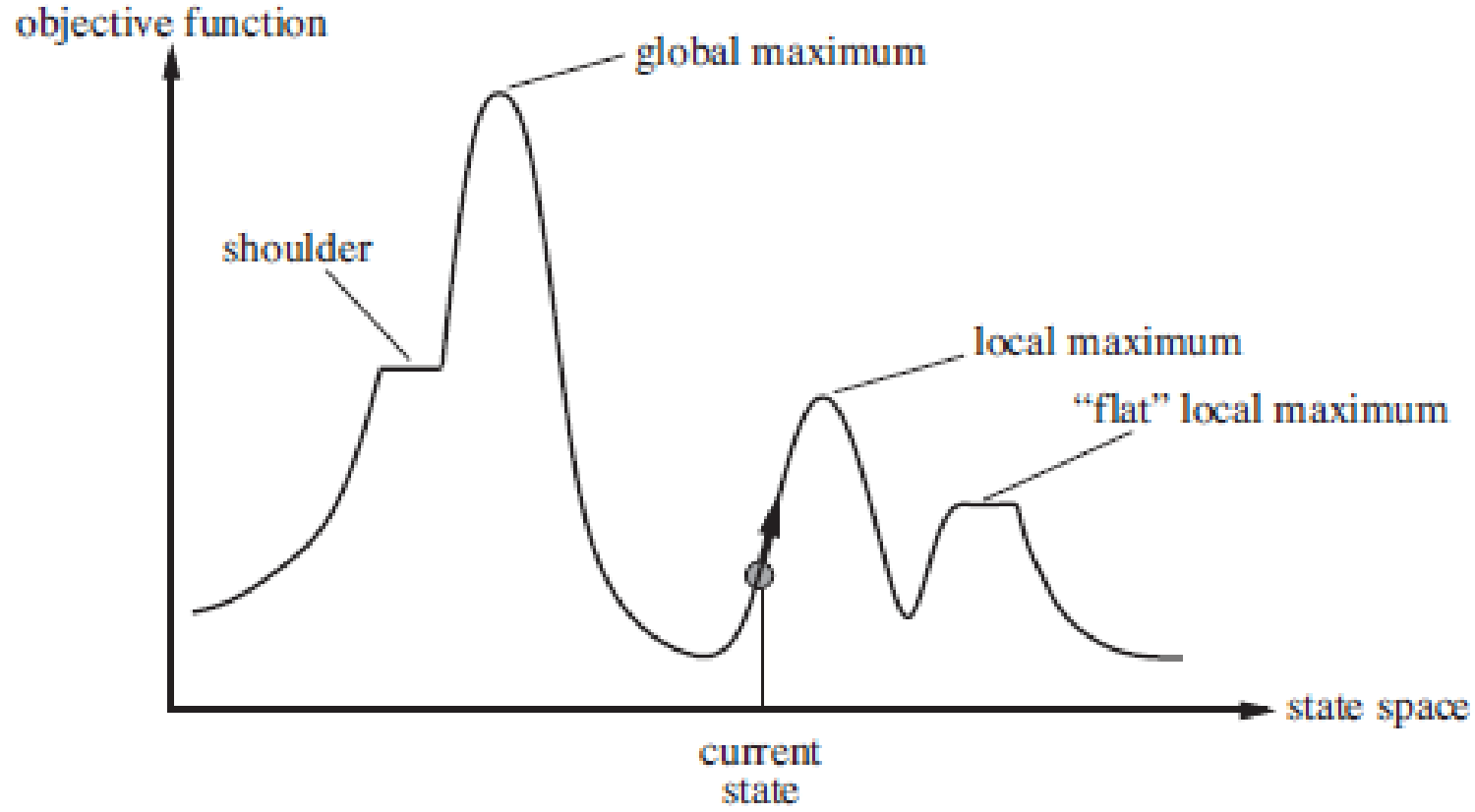

Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-Climbing Search

- The hill-climbing search algorithm (steepest-ascent version) is shown in Figure.
- It is simply a loop that continually moves in the direction of increasing value—that is, uphill.
- It terminates when it reaches a “peak” where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.
- This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia.

Hill-Climbing Search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Hill-Climbing Search

- To illustrate hill climbing, we will use the 8-queens problem.
- Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column.
- The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8×7=56 successors).
- The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
- The global minimum of this function is zero, which occurs only at perfect solutions. Figure 4.3(a) shows a state with h=17.
- The figure also shows the values of all its successors, with the best successors having h=12.
- Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

Hill-Climbing Search

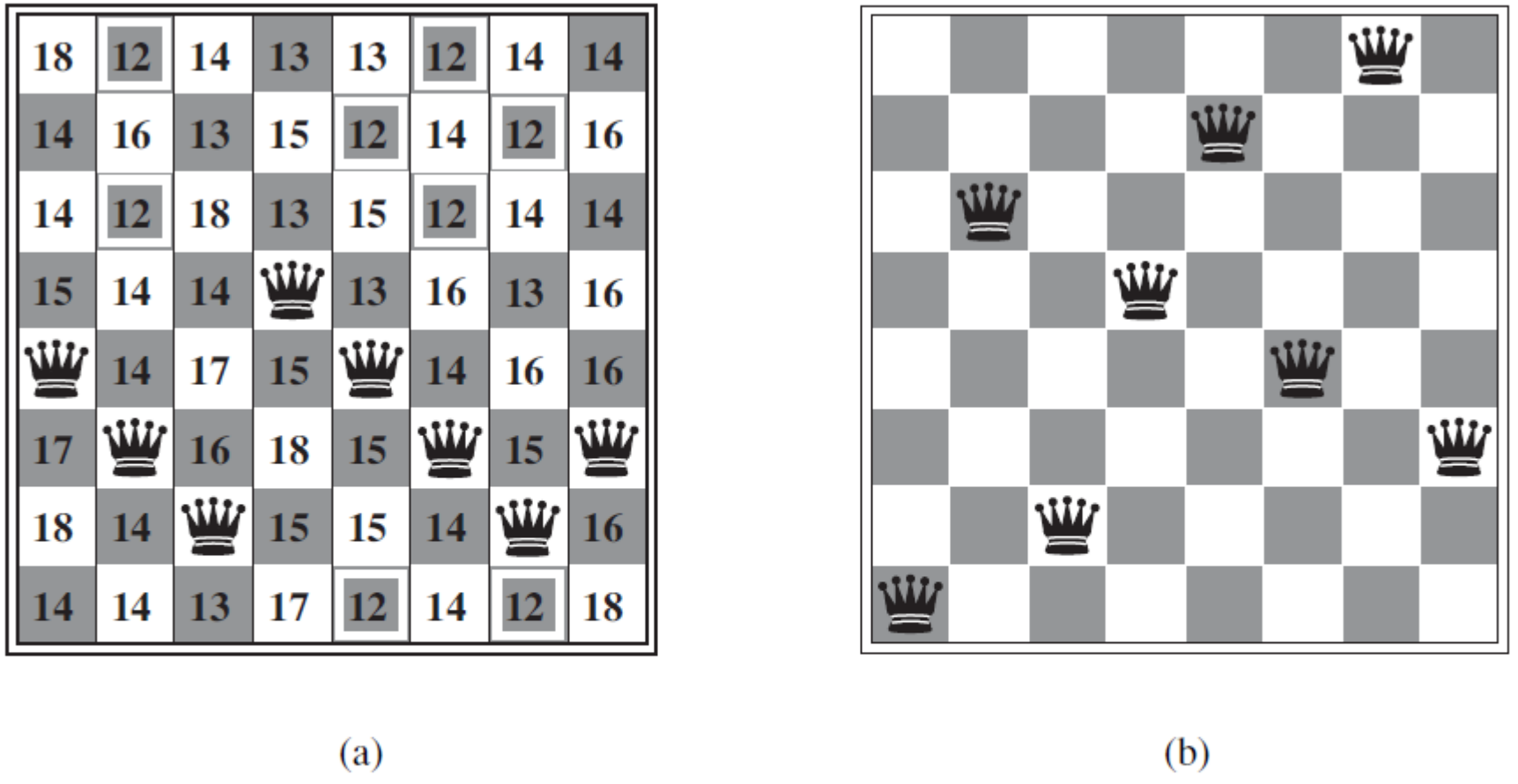

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Hill-Climbing Search

❑Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.

❑Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well.

❑Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state.

❑For example, from the state in Figure 4.3(a), it takes just five steps to reach the state in Figure 4.3(b), which has h=1 and is very nearly a solution.

❑Unfortunately, hill climbing often gets stuck for the following reasons:

1. **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. Figure 4.1 illustrates the problem schematically. More concretely, the state in Figure 4.3(b) is a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.

Hill-Climbing Search

1. **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. Figure 4.1 illustrates the problem schematically. More concretely, the state in Figure 4.3(b) is a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.
2. **Ridges:** a ridge is shown in Figure 4.4. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
3. **Plateaux:** a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. (See Figure 4.1.) A hill-climbing search might get lost on the plateau.

Hill-Climbing Search

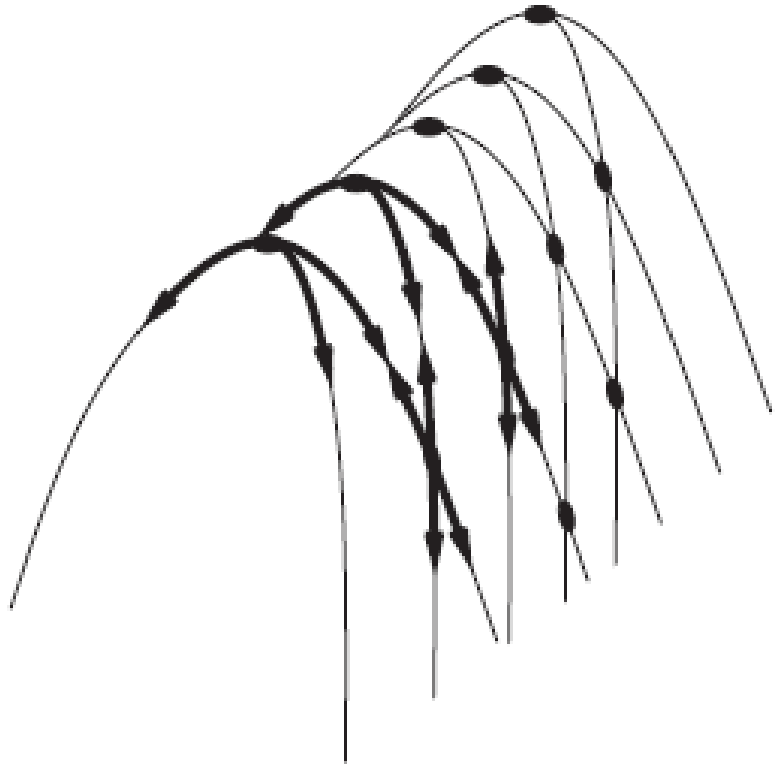

Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Hill-Climbing Search

- In each case, the algorithm reaches a point at which no progress is being made.
- Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances.
- It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with $8^8 \approx 17$ million states.

Hill-Climbing Search

❑**Many variants of hill climbing have been invented.**

1. **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
   - This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
2. **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
   - This is a good strategy when a state has many (e.g., thousands) of successors.
   - The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima.

Hill-Climbing Search

❑**Many variants of hill climbing have been invented.**

3. **Random-restart hill climbing** adopts the well-known adage, "If at first you don't succeed, try, try again."
   - It conducts a series of hill-climbing searches from randomly generated initial states,1 until a goal is found.
   - It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is 1/p. For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success).
   - The expected number of steps is the cost of one successful iteration plus $(1-p)/p$ times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, $1/0.94 \approx 1.06$ iterations are needed on average and $(1\times21)+(0.06/0.94)\times64 \approx 25$ steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.2

Simulated Annealing

- A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.
- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.
- Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness.
- Simulated annealing is such an algorithm.
- In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.

Simulated Annealing

- To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface.
- If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum.
- The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum.
- The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

Simulated Annealing

- The innermost loop of the simulated-annealing algorithm (Figure 4.5) is quite similar to hill climbing.
- Instead of picking the best move, however, it picks a random move.
- If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the “badness” of the move—the amount ΔE by which the evaluation is worsened.
- The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated Annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"

    current ← MAKE-NODE(problem.INITIAL-STATE)
    for t = 1 to ∞ do
        T ← schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.VALUE – current.VALUE
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE/T)
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.

Genetic Algorithm

- A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.
- GAs begin with a set of k randomly generated states, called the population.
- Each state, or individual, is POPULATION represented as a string over a finite alphabet—most INDIVIDUAL commonly, a string of 0s and 1s.
- For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2 8 = 24$ bits.
- Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We demonstrate later that the two encodings behave differently.)
- Figure 4.6(a) shows a population of four 8-digit strings representing 8-queens states.

Genetic Algorithm

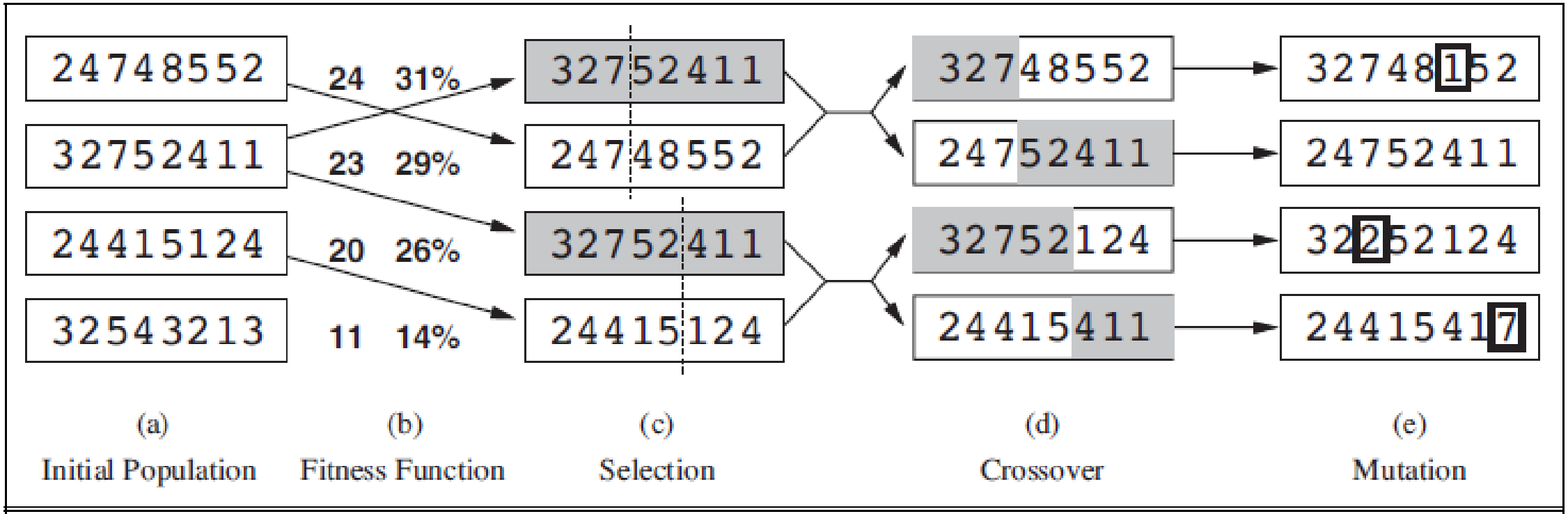

Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic Algorithm

- The production of the next generation of states is shown in Figure 4.6(b)–(e). In (b), each state is rated by the objective function, or (in GA terminology) the fitness function.
- A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of non-attacking pairs of queens, which has a value of 28 for a solution.
- The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.
- In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b).
- Notice that one individual is selected twice and one not at all.
- For each pair to be mated, a crossover point is chosen randomly from the positions in the string.
- In Figure 4.6, the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.

Genetic Algorithm

- In (d), the offspring themselves are created by crossing over the parent strings at the crossover point.
- For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent.
- The 8-queens states involved in this reproduction step are shown in Figure 4.7. The example shows that when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state.
- It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

Genetic Algorithm

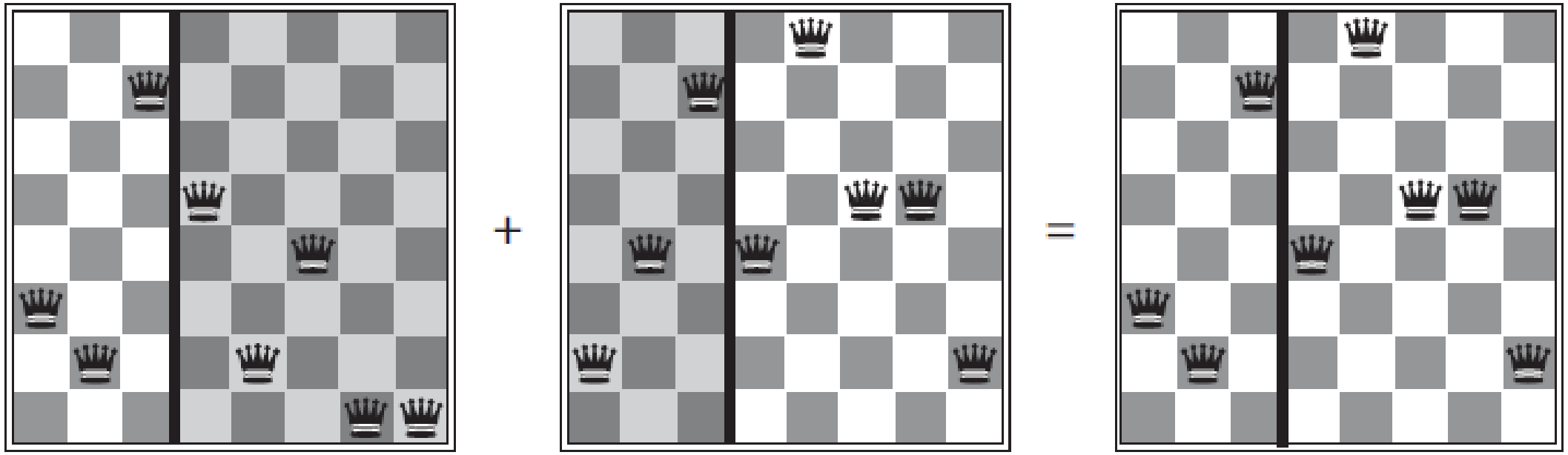

Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

Genetic Algorithm

- Finally, in (e), each location is subject to random **mutation with a small independent** probability.
- One digit was mutated in the first, third, and fourth offspring.
- In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.
- Figure 4.8 describes an algorithm that implements all these steps.

Genetic Algorithm

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

Adversarial Search- Game Trees

- Mathematically game theory views as any multiagent environment as a game, provided that the impact of each agent on the others is “significant,” regardless of whether the agents are cooperative or competitive.
- In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, zero-sum games of perfect information (such as chess).
- In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.
- For example, if one player wins a game of chess, the other player necessarily loses.
- It is this opposition between the agents’ utility functions that makes the situation adversarial.

Adversarial Search- Game Trees

- Games have engaged the intellectual faculties of humans—sometimes to an alarming degree—for as long as civilization has existed.
- For AI researchers, the abstract nature of games makes them an appealing subject for study.
- The state of a game is easy to represent, and agents are usually restricted to a small number of actions whose outcomes are defined by precise rules.
- Physical games, such as croquet and ice hockey, have much more complicated descriptions, a much larger range of possible actions, and rather imprecise rules defining the legality of actions.
- With the exception of robot soccer, these physical games have not attracted much interest in the AI community.

Adversarial Search- Game Trees

- We begin with a definition of the optimal move and an algorithm for finding it.
- We then look at techniques for choosing a good move when time is limited.
- Pruning allows us to ignore portions of the search tree that make no difference to the final choice, and heuristic evaluation functions allow us to approximate the true utility of a state without doing a complete search.

Adversarial Search- Game Trees- Two-Player Games

- We first consider games with **two players**, whom we call MAX and MIN for reasons that will soon become obvious.
- MAX moves first, and then they take turns moving until the game is over.
- At the end of the game, points are awarded to the winning player and penalties are given to the loser.
- A game can be formally defined as a kind of search problem with the following elements:

  - S_0: The **initial state**, which specifies how the game is set up at the start.
  - PLAYER(s): Defines which player has the move in a state.
  - ACTIONS(s): Returns the set of legal moves in a state.
  - RESULT(s, a): The **transition model**, which defines the result of a move.
  - TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.

Adversarial Search- Game Trees- Two-Player Games

❑A game can be formally defined as a kind of search problem with the following elements:

- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values $+1$, 0, or $\frac{1}{2}$. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to $+192$. A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either $0 + 1$, $1 + 0$ or $\frac{1}{2} + \frac{1}{2}$. "Constant-sum" would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of $\frac{1}{2}$.

Adversarial Search- Game Trees- Two-Player Games (tic-tac-toe)

- The initial state, ACTIONS function, and RESULT function define the **game tree for the** game—a tree where the nodes are game states and the edges are moves.
- Figure shows part of the game tree for tic-tac-toe (noughts and crosses).
- From the initial state, MAX has nine possible moves.
- Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

Adversarial Search- Game Trees- Two-Player Games (tic-tac-toe)

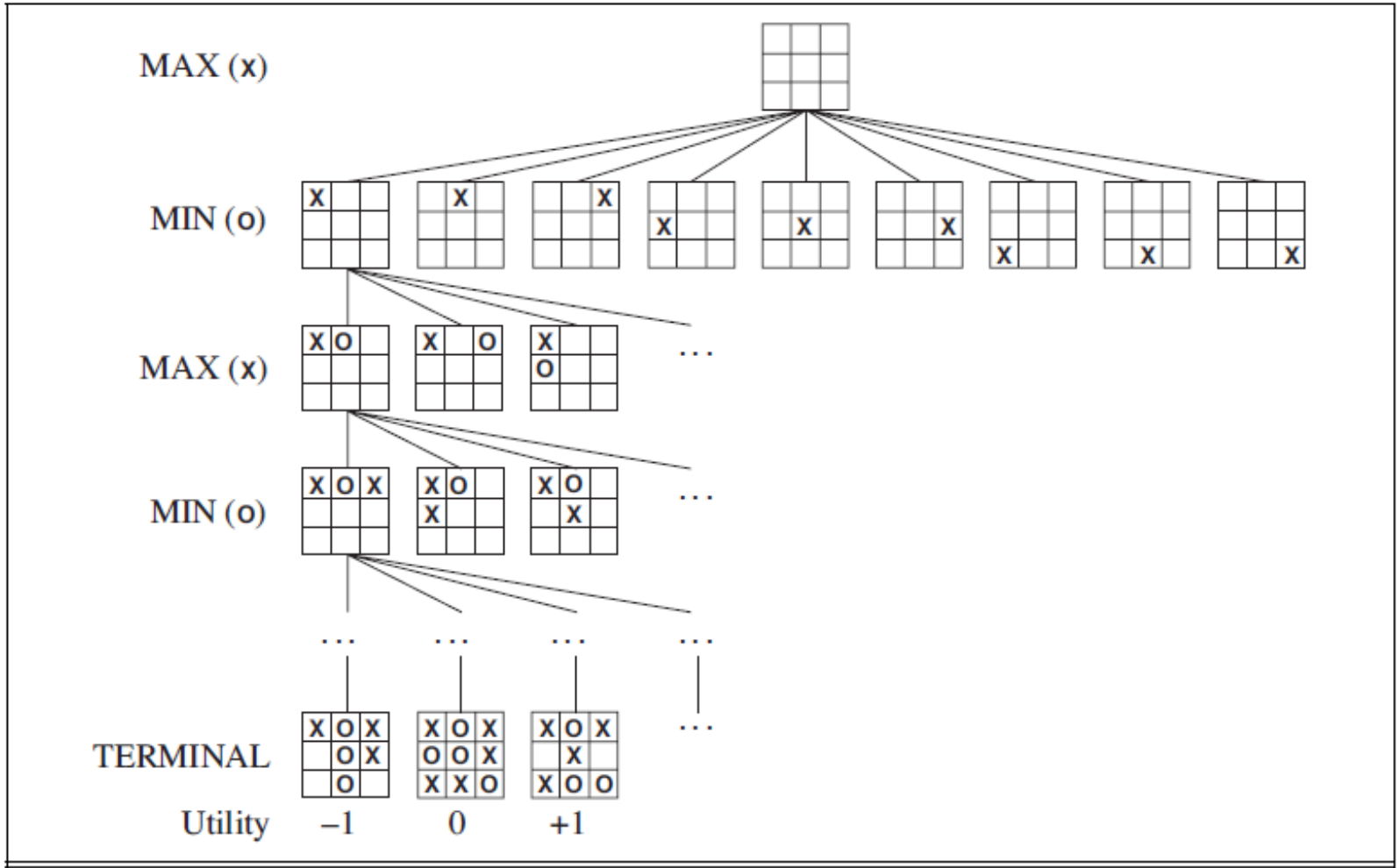

Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Adversarial Search- Game Trees- Two-Player Games (tic-tac-toe)

- For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes.
- But for chess there are over 1040 nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.
- But regardless of the size of the game tree, it is MAX's job to search for a good move.
- We use the term **search tree for a** tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

Min-Max Evaluation – Optimal Decisions in Games

- In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win.
- In adversarial search, MIN has something to say about it.
- MAX therefore must find a contingent **strategy, which specifies MAX's move in** the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to *those* moves, and so on.
- This is exactly analogous to the AND–OR search algorithm with MAX playing the role of OR and MIN equivalent to AND.
- Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.
- We begin by showing how to find this optimal strategy.

Min-Max Evaluation – Optimal Decisions in Games

- Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will switch to the trivial game in Figure.
- The possible moves for MAX at the root node are labeled a1, a2, and a3. The possible replies to a1 for MIN are b1, b2, b3, and so on.
- This particular game ends after one move each by MAX and MIN.
- In game parlance, we say that this tree is one move deep, consisting of two half-moves, each of which is called a ply.
- The utilities of the terminal states in this game range from 2 to 14.

Min-Max Evaluation – Optimal Decisions in Games

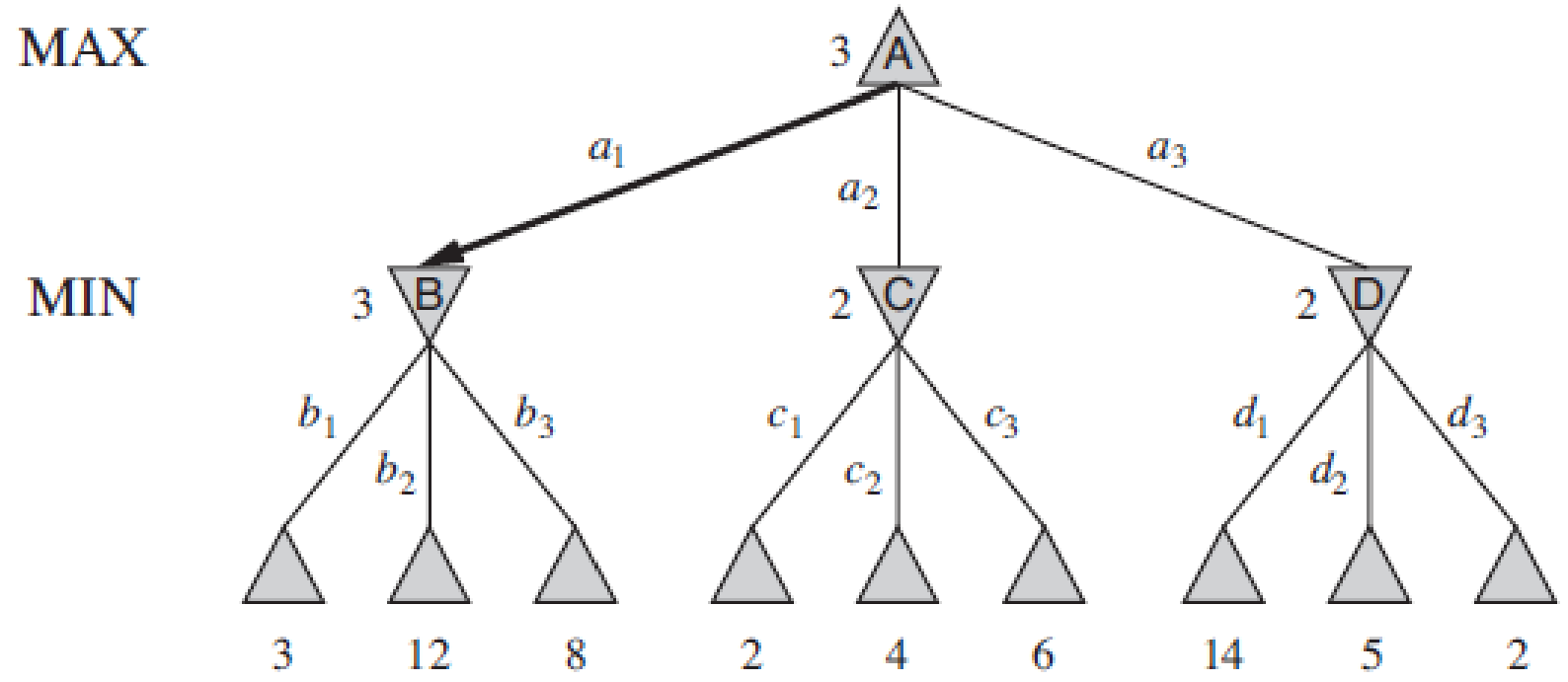

Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Min-Max Evaluation – Optimal Decisions in Games

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(n).
- The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game.
- Obviously, the minimax value of a terminal state is just its utility.
- Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.
- So we have the following:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Min-Max Evaluation – Optimal Decisions in Games

- Let us apply these definitions to the game tree in Figure 5.2.
- The terminal nodes on the bottom level get their utility values from the game's UTILITY function.
- The first MIN node, labeled B, has three successor states with values 3, 12, and 8, so its minimax value is 3.
- Similarly, the other two MIN nodes have minimax value 2.
- The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3.
- We can also identify the minimax decision at the root: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

Min-Max Evaluation – Optimal Decisions in Games

- This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the worst-case outcome for MAX.
- What if MIN does not play optimally?
- Then it is easy to show that MAX will do even better.
- Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

Min-Max Evaluation – Minimax Algorithm

- The minimax algorithm (Figure 5.3) computes the minimax decision from the current state.
- It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.
- The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.
- For example, in Figure 5.2, the algorithm first recurses down to the three bottomleft nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively.
- Then it takes the minimum of these values, 3, and returns it as the backedup value of node B. A similar process gives the backed-up values of 2 for C and 2 for D.
- Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

Min-Max Evaluation – Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

Min-Max Evaluation – Minimax Algorithm

- The minimax algorithm performs a complete depth-first exploration of the game tree.
- If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$.
- The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time.
- For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Minimax with Alpha-Beta Pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half.
- The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree.
- That is, we can borrow the idea of pruning to eliminate large parts of the tree from consideration.
- The particular technique we examine is called alpha–beta pruning.
- When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Minimax with Alpha-Beta Pruning

❑Consider again the two-ply game tree from Figure 5.2, Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process.

❑The steps are explained in Figure 5.5, The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

❑Another way to look at this is as a simplification of the formula for MINIMAX.

❑Let the two unevaluated successors of node C in Figure 5.5 have values x and y.

❑Then the value of the root node is given by

$$\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}$$

❑In other words, the value of the root and hence the minimax decision are *independent of the* values of the pruned leaves x and y.

Minimax with Alpha-Beta Pruning

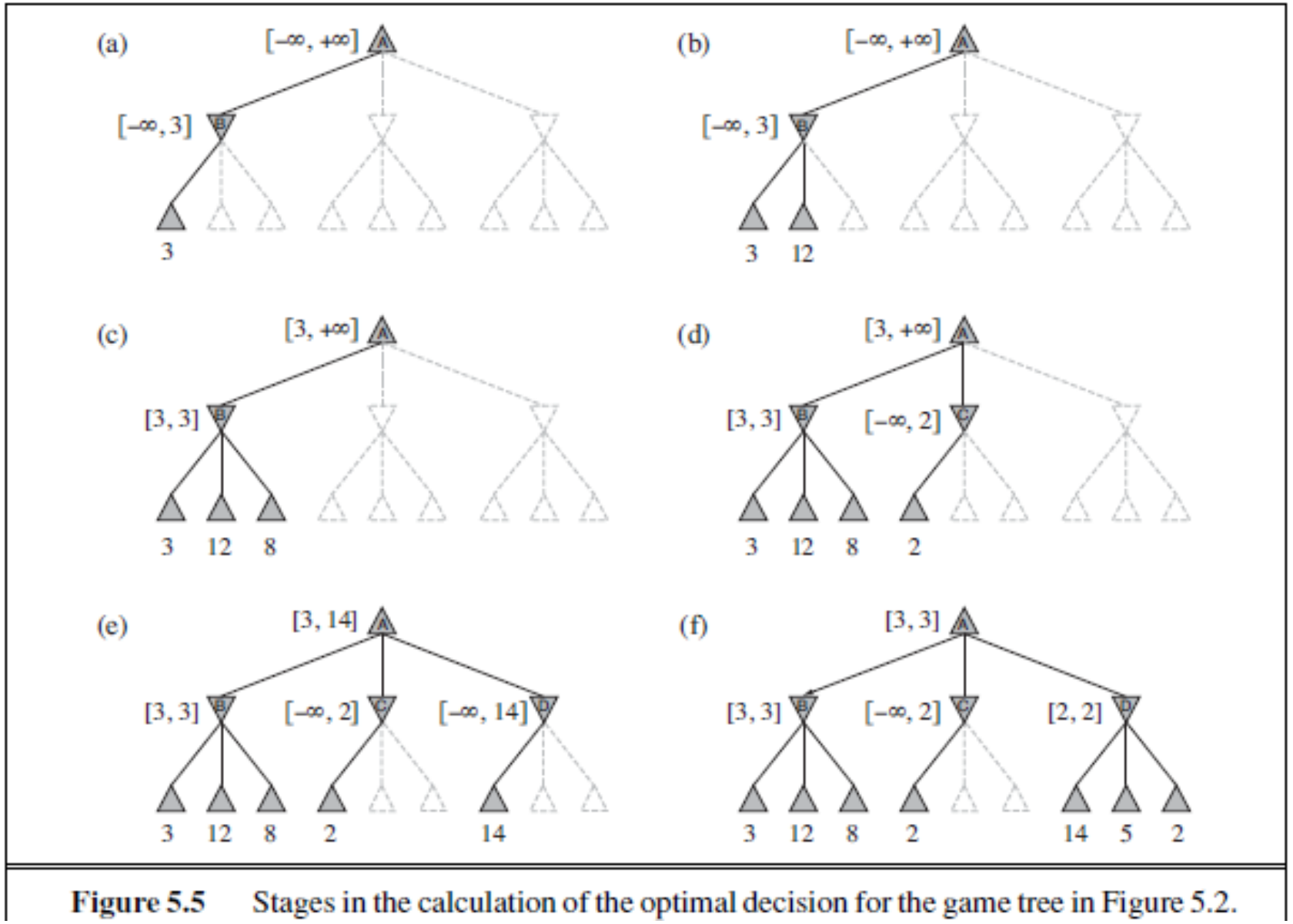

Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2.

Minimax with Alpha-Beta Pruning

Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Minimax with Alpha-Beta Pruning

- Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.
- The general principle is this: consider a node nsomewhere in the tree (see Figure 5.6), such that Player has a choice of moving to that node.
- If Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play.
- So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Minimax with Alpha-Beta Pruning

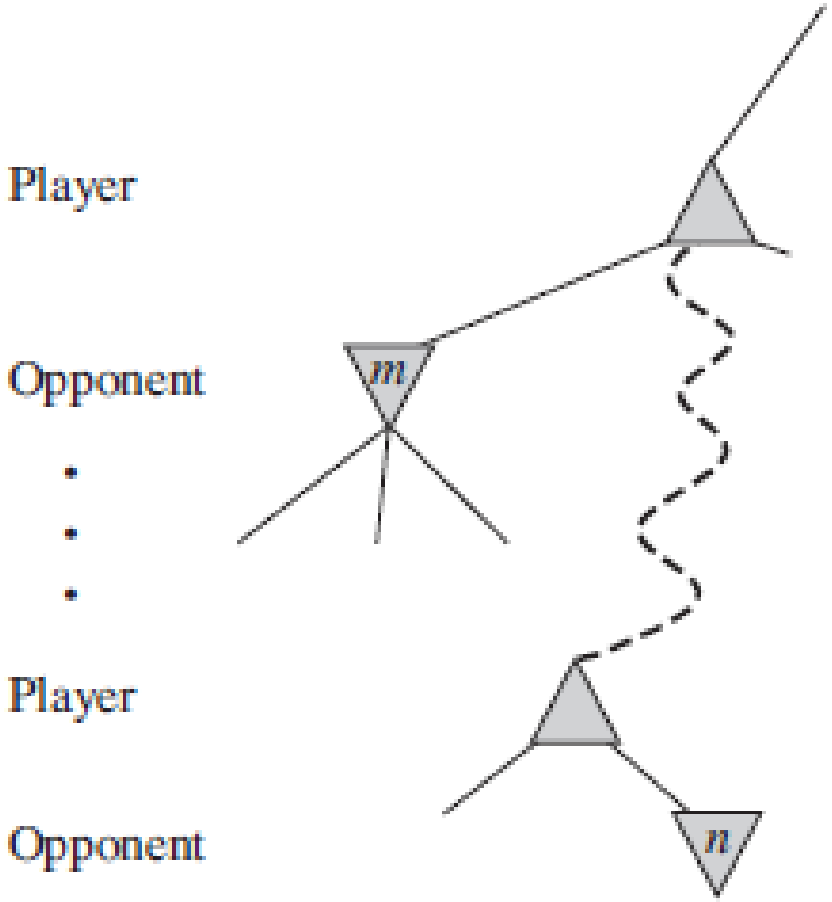

Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

Minimax with Alpha-Beta Pruning

❑Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree.

❑Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

→ α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

→ β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

❑Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

Minimax with Alpha-Beta Pruning

- The complete algorithm is given in Figure 5.7.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Note for Students

- This power point presentation is for lecture, therefore it is suggested that also utilize the text books and lecture notes.
- Also Refer the solved and unsolved examples of Text and Reference Books.