北京大学

超越经典搜索

Beyond Classical Search

·AIPKU·

内容来自：Artificial Intelligence A Modern Approach
Stuart Russel Peter Norvig Chapter 4

人工智能引论第六课

主讲人：李文新

前言

- Classical Searching problems: the **solution is a sequence of actions.**

- Sections 4.1 and 4.2 cover algorithms that perform purely **local search** in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state.

- These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it. The family of local search algorithms includes methods inspired by statistical physics (**simulated annealing**) and evolutionary biology (**genetic algorithms**).

内容提要

- 局部搜索算法和最优化问题
  - 爬山法、模拟退火搜索、局部束搜索、遗传算法
- 连续空间中的局部搜索
- 使用不确定动作的搜索
  - 不稳定的吸尘器世界、与或搜索树、不断尝试
- 使用部分可观察信息的搜索
  - 无观察信息的搜索、有观察信息的搜索、求解部分可观察环境中的问题、部分可观察环境中的Agent
- 联机搜索Agent和未知环境
  - 联机搜索问题、联机搜索Agent、联机局部搜索、联机搜索中的学习

Path is irrelevant， local search with a single node

- 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, **the path to the goal is irrelevant**. For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added. 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 operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node.

little memory for large / optimization problems

- Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very **little memory**—usually a constant amount; and (2) they can often find reasonable solutions **in large or infinite (continuous) state** spaces for which systematic algorithms are unsuitable.

- 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**. Many optimization problems do not fit the "standard" search model introduced in Chapter 3. 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.

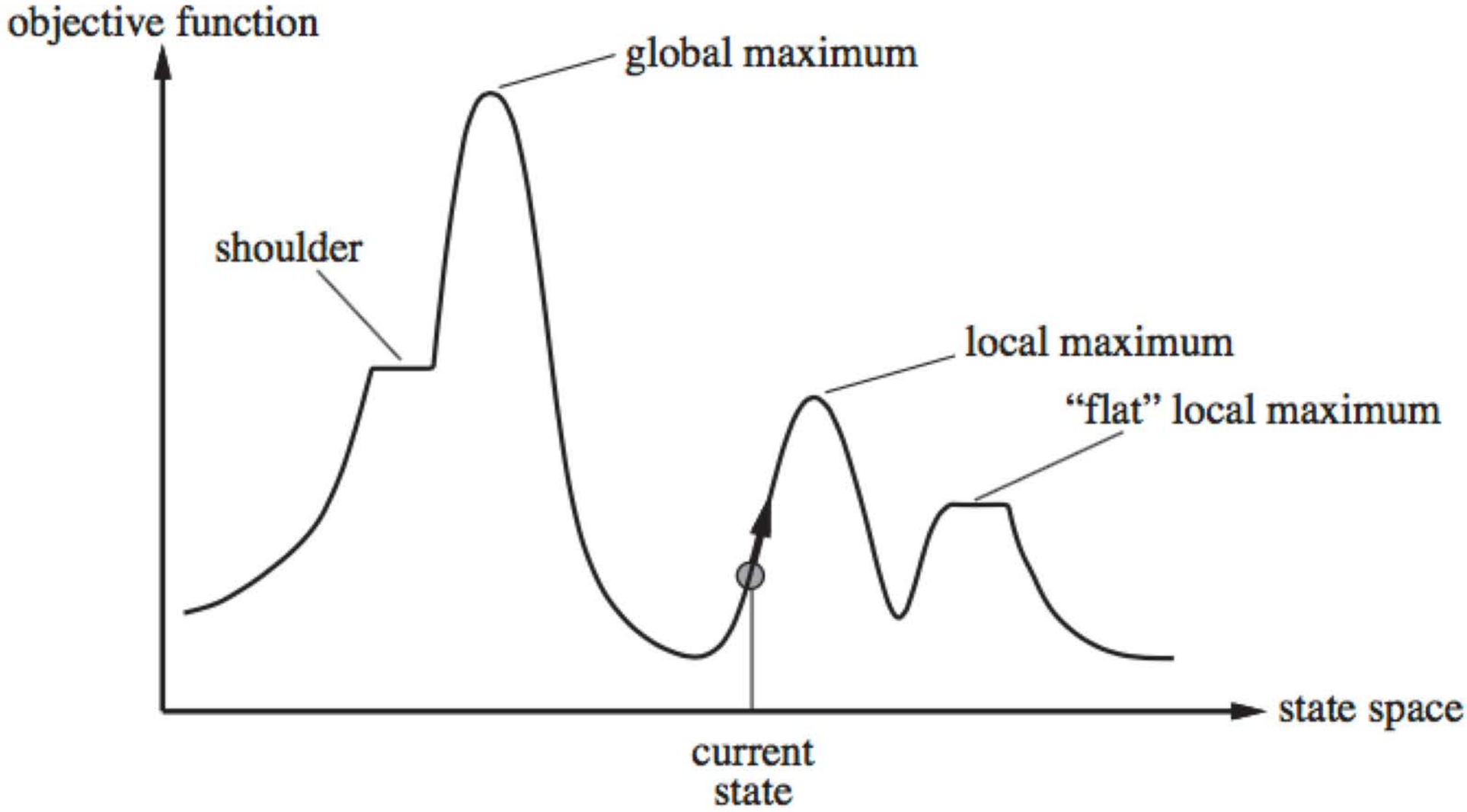


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

- A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

爬山法 The hill-climbing search algorithm

爬山法The **hill-climbing** search algorithm (**steepest-ascent** version). 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.

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.

8 皇后例题 估值函数 h 为互相攻击对

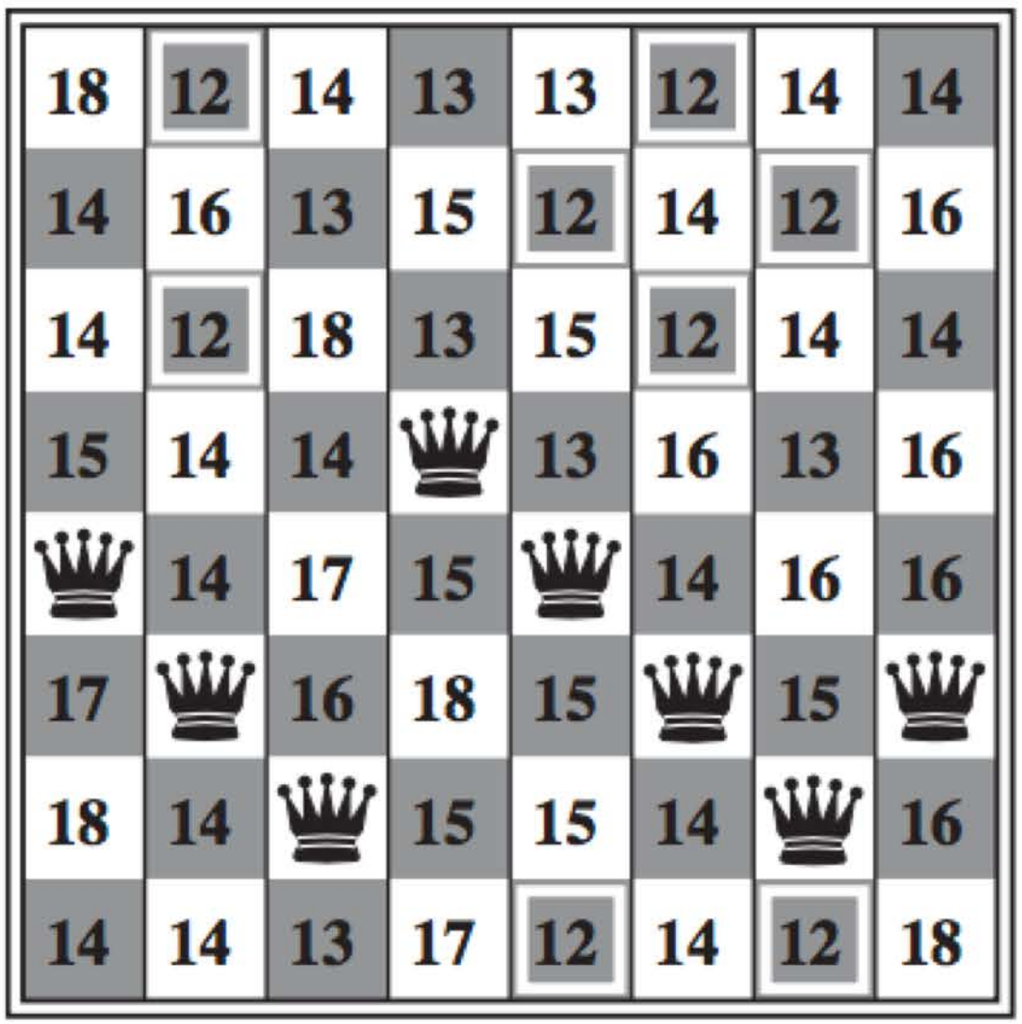
- **8-queens problem** ，**complete-state formulation**，8 queens on the board , one per column

- The heuristic cost function **h** is the number of pairs of queens that are attacking each other, either directly or indirectly.

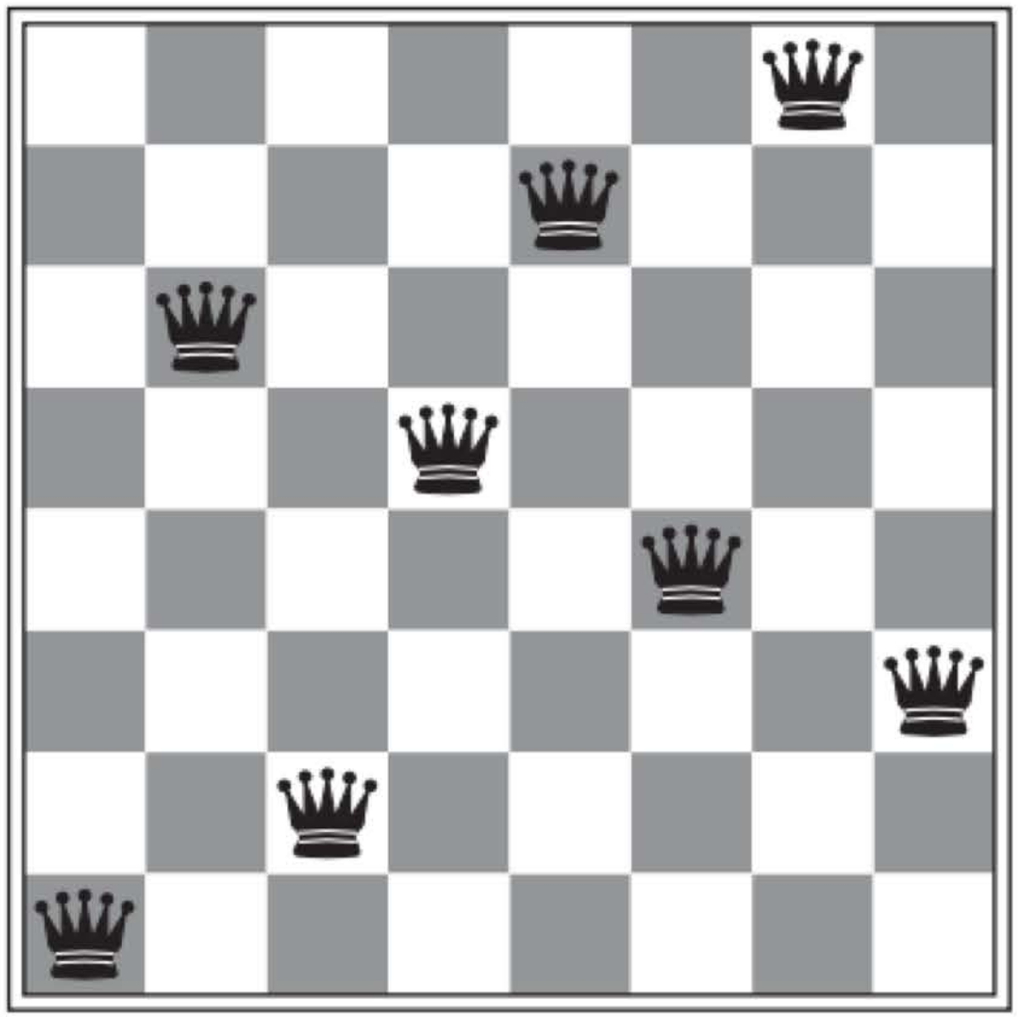
- moving a single queen to another square in the same column

- each state has 8 × 7 = 56 successors

(a)估值函数相邻状　（b）陷入局部最优



(a)



(b)

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.

内容提要 – 局部搜索算法和最优化问题

- **Local maxima** • **Ridges** • **Plateaux**:

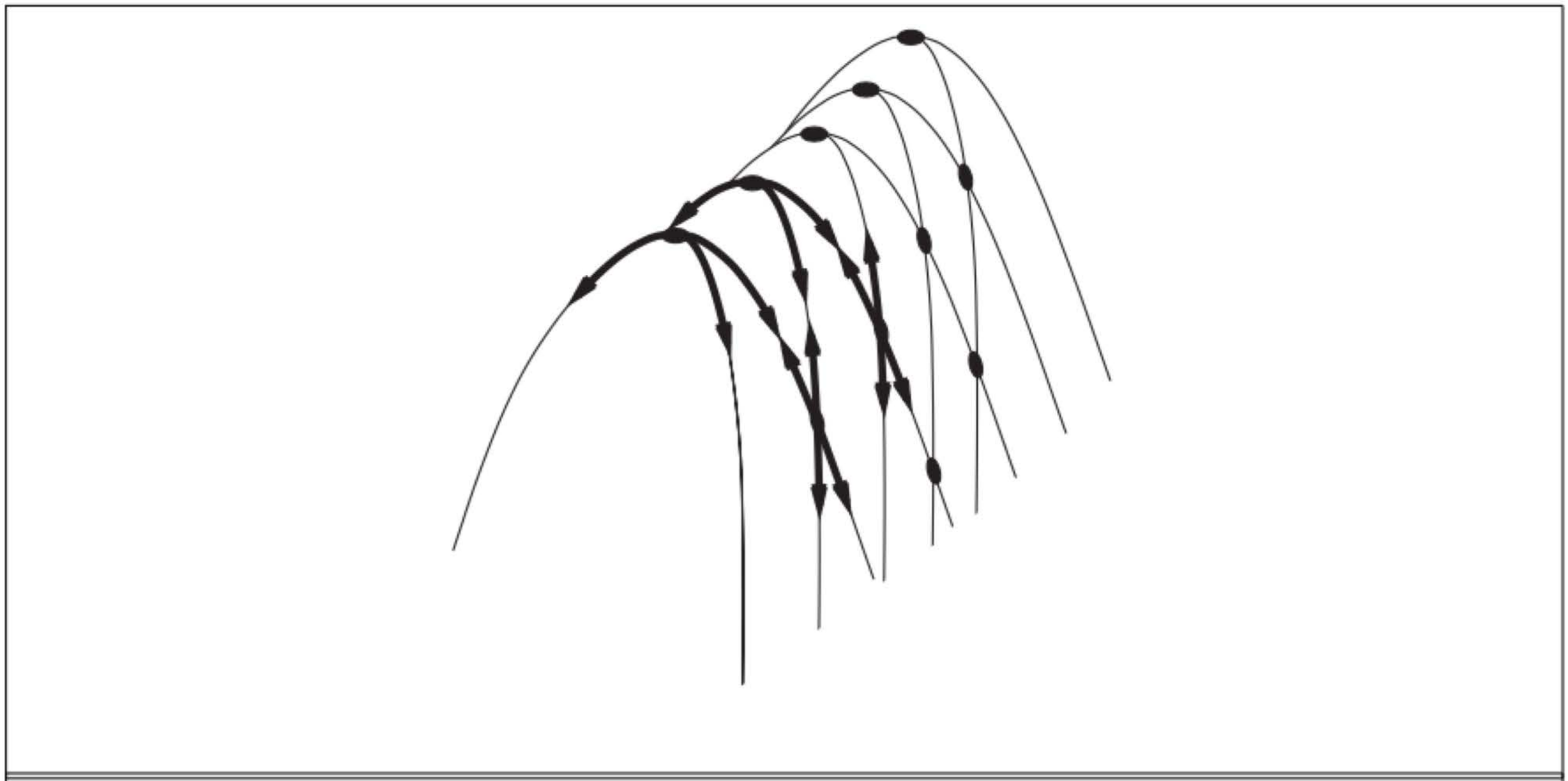


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point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

sideways move

- 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.

- to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4.1? an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed.

- For example, we could allow up to, say, **100 consecutive sideways** moves in the 8-queens problem. This raises the percentage of problem instances **solved** by hill climbing **from 14% to 94%.** Success comes at a cost: the algorithm **averages roughly 21 steps** for each **successful** instance and **64 for each failure**.

爬山法各种变种

- Many variants of hill climbing have been invented. **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. **好邻居里随机找一个**

- **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. **随机找一个好邻居**

- 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. **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, until a goal is found. It is trivially complete with probability approaching , because it will eventually generate a goal state as the initial state. **找不到解再从随机位置开始**

random-restart hill climbing is very effective

- 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 ≈ 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 ≈ 1.06 iterations are needed on average and (1 × 21) + (0.06/0.94) × 64 ≈ 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.

success depends on shape of state-space landscape

- The success of hill climbing depends very much on the **shape of the state-space landscape**: if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.

- On the other hand, many real problems have a landscape that looks more like a widely scattered family of balding porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, *ad infinitum*.

- **NP-hard problems** typically have an exponential number of local maxima to get stuck on. Despite this, a reasonably good local maximum can often be found after a small number of restarts.

内容提要 – 模拟退火搜索

- **模拟退火搜索 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.

内容提要 – 模拟退火搜索

- **Two ideas**

- 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.

- **A ping-pong ball**. 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.

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**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow next.\text{VALUE} - current.\text{VALUE}$
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

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.

- **Simulated annealing** was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks. In Exercise 4.4, you are asked to compare its performance to that of random-restart hill climbing on the 8-queens puzzle.

内容提要 – 模拟退火搜索

- 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.

内容提要 – 局部束搜索

- 局部束搜索 **Local beam search**

- Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The **local beam search** algorithm[3] keeps track of **k** states rather than just one. It begins with **k** randomly generated states. At each step, all the successors of all **k** states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the **k** best successors from the complete list and repeats.

- At first sight, a local beam search with **k** states might seem to be nothing more than running **k** random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. *In a local beam search,* ***useful information is passed among the parallel search threads****.* In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

内容提要 – 局部束搜索

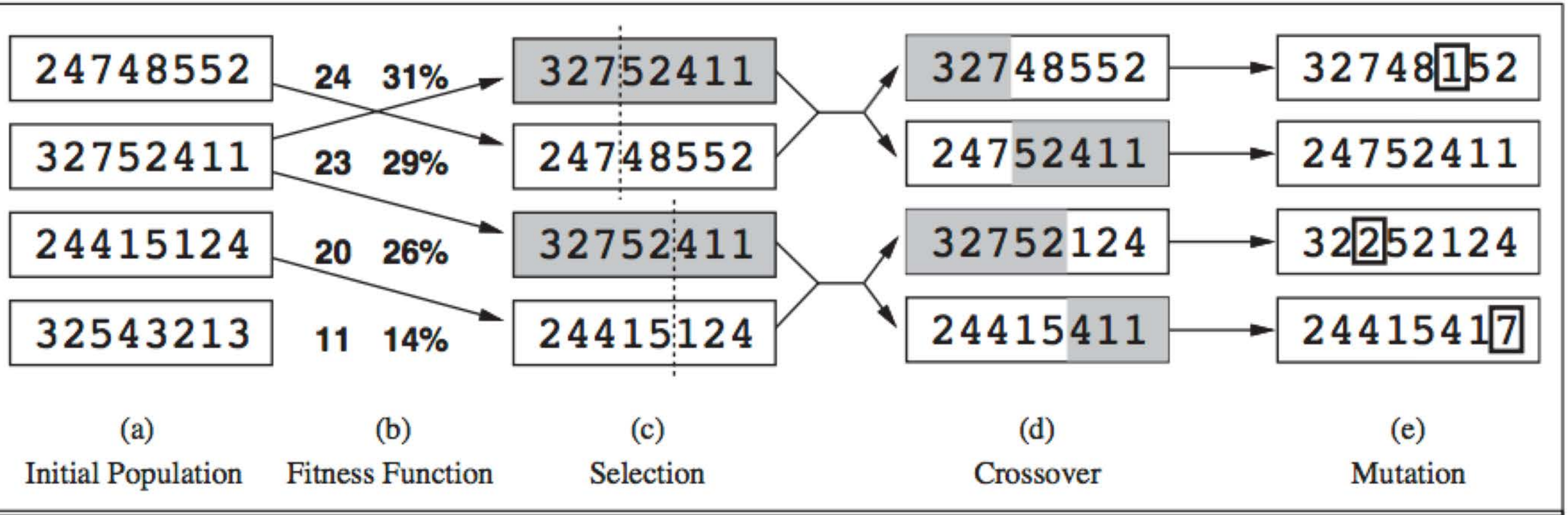
- 局部束搜索 **Local beam search**

- In its simplest form, local beam search can suffer from a lack of diversity among the **k** states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search**, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the best **k** from the the pool of candidate successors, stochastic beam search chooses **k** successors at random, with the probability of choosing a given successor being an increasing function of its value. Stochastic beam search bears some resemblance to the process of natural selection, whereby the "successors" (offspring) of a "state" (organism) populate the next generation according to its "value" (fitness).

内容提要 – 遗传算法

- 遗传算法 **Genetic algorithms**

- 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. The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with **sexual** rather than **asexual reproduction**.



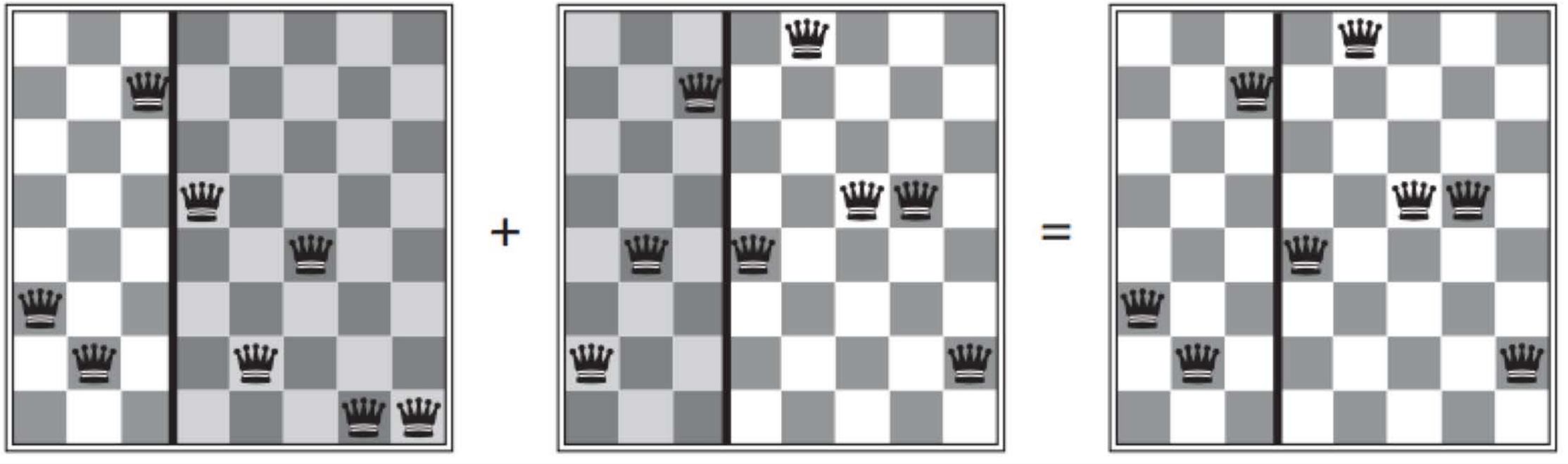


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

```
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))
```

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.

内容提要 – 遗传算法

- The theory of genetic algorithms explains how this works using the idea of a **schema**, which is a substring in which some of the positions can be left unspecified. **模式**

- It can be shown that if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema within the population will grow over time. **具有较好模式的人数会增加**

- Genetic algorithms work best when schemata correspond to **meaningful** components of a solution.

- This suggests that successful use of genetic algorithms requires careful engineering of the **representation**.

内容提要 – 遗传算法

- In practice, genetic algorithms have had a widespread impact on optimization problems, such as **circuit layout** and **job-shop scheduling**. At present, it is **not clear** whether the appeal of genetic algorithms arises from their performance or from their æsthetically pleasing origins in the theory of evolution.
- **Much work remains** to be done to identify the conditions under which genetic algorithms perform well.

北京大学

对抗搜索
Adversarial Search

·AIPKU·

内容来自：Artificial Intelligence A Modern Approach
Stuart Russel　Peter Norvig　Chapter 5
人工智能引论第六课
主讲人：李文新

内容提要

- 博弈
- 博弈中的优化决策
  - 极大极小算法、多人博弈时的最优决策、αβ剪枝
- 不完美的实时决策
  - 评估函数、截断搜索、向前剪枝、搜索与查表
- 随机博弈（机会博弈中的评估函数）
- 部分可观察的博弈（军旗、牌类）

内容提要 - 博弈

- *In which we examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.*

- 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**).

内容提要 - 博弈

- Games, **are interesting *because* they are too hard to solve**. For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes (although the search graph has "only" about 10^{40} distinct nodes). Games, like the real world, therefore require the **ability to make *some* decision even when calculating the *optimal* decision is infeasible.**

- Game-playing research has therefore spawned a number of interesting ideas on how to make the **best possible use of time.**

内容提要 - 博弈

- 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.

内容提要 - 博弈

- 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:

- • S0: 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**.

- • U T I L I T Y (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. A **zero-sum game**

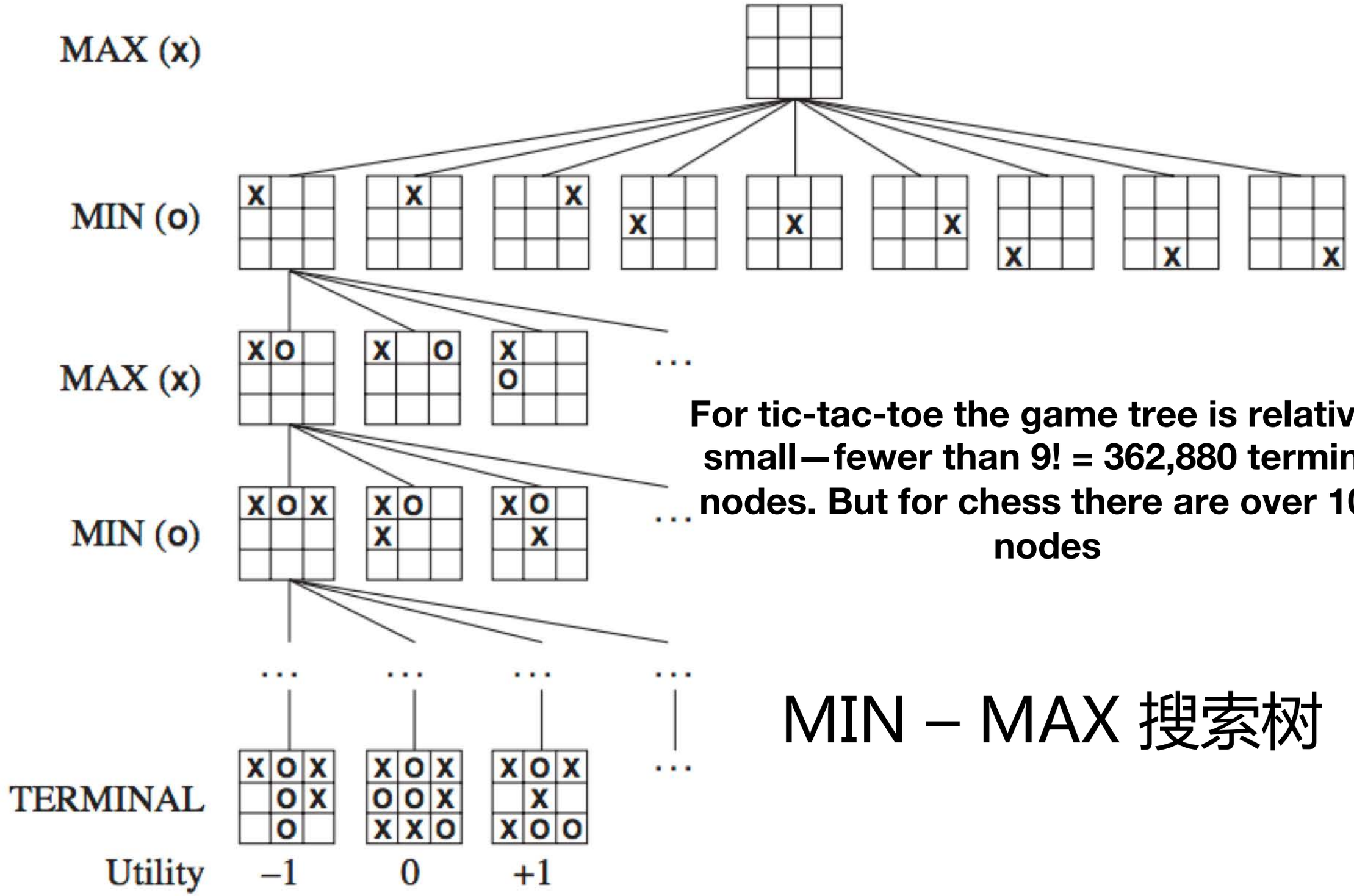


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities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

MIN-MAX

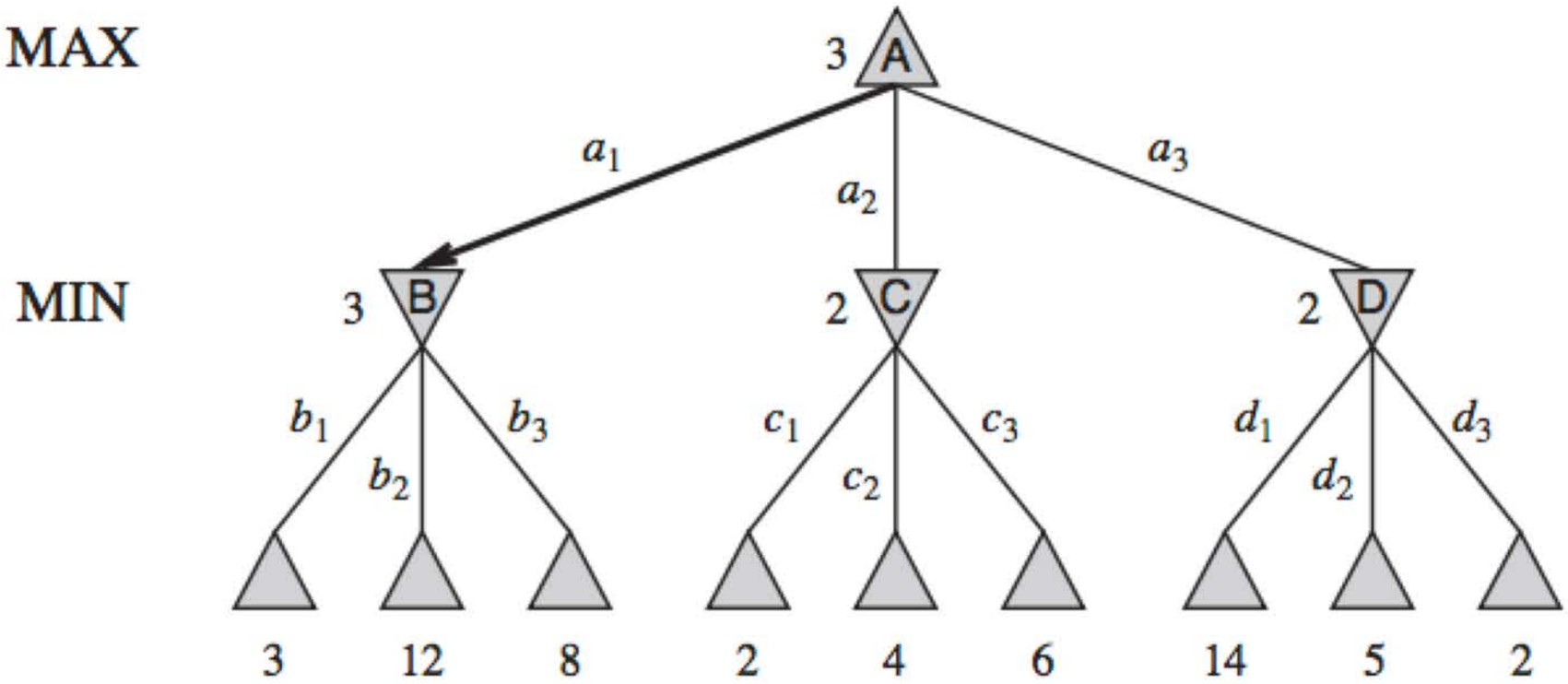


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state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

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

```
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 算法的效率分析

- 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 (see page 87). 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.

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.

- 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

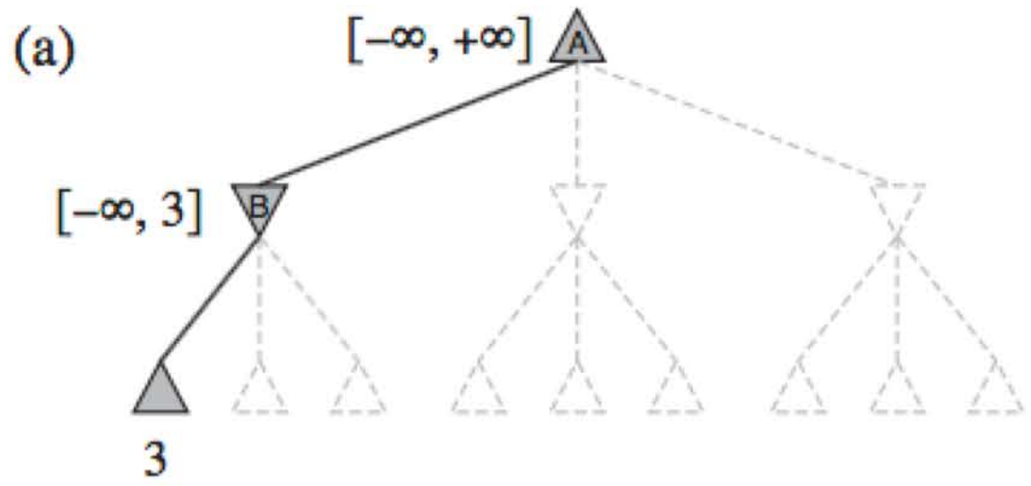
- MINIMAX(root) = max(min(3,12,8),min(2,x,y),min(14,5,2))

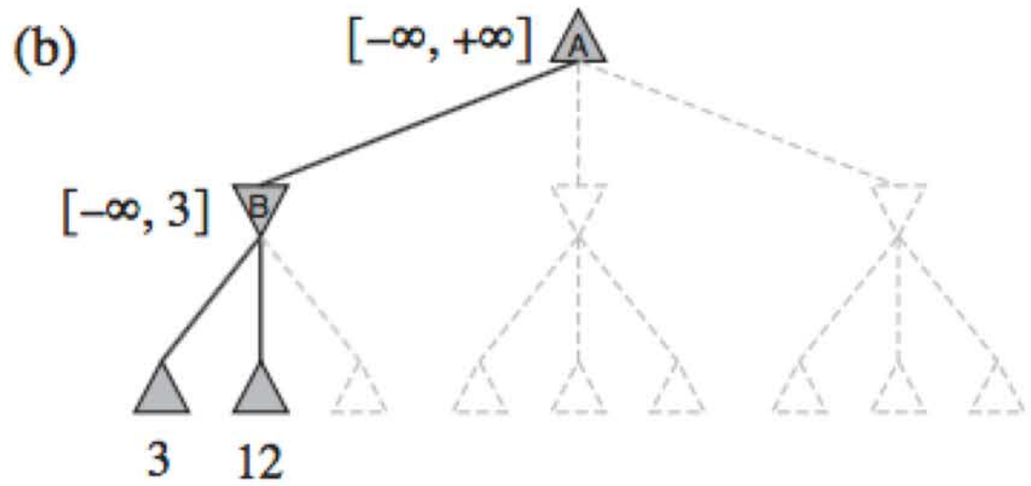
- = max(3,min(2,x,y),2)

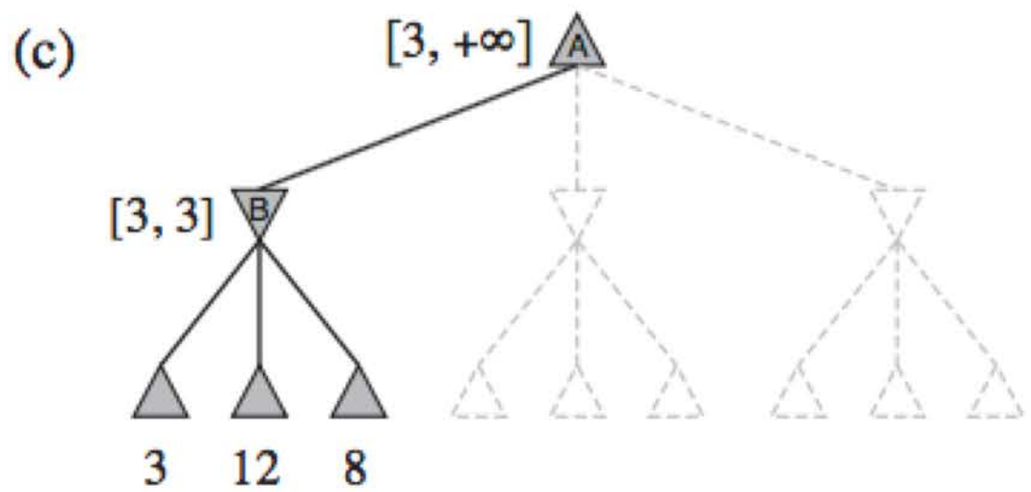
- = max(3,z,2) where z = min(2,x,y) ≤ 2 = 3.

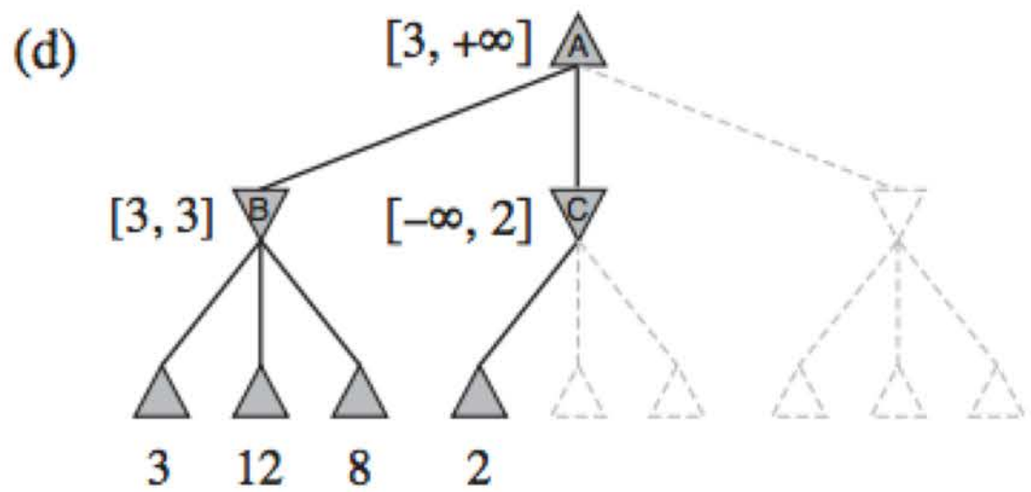
- In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

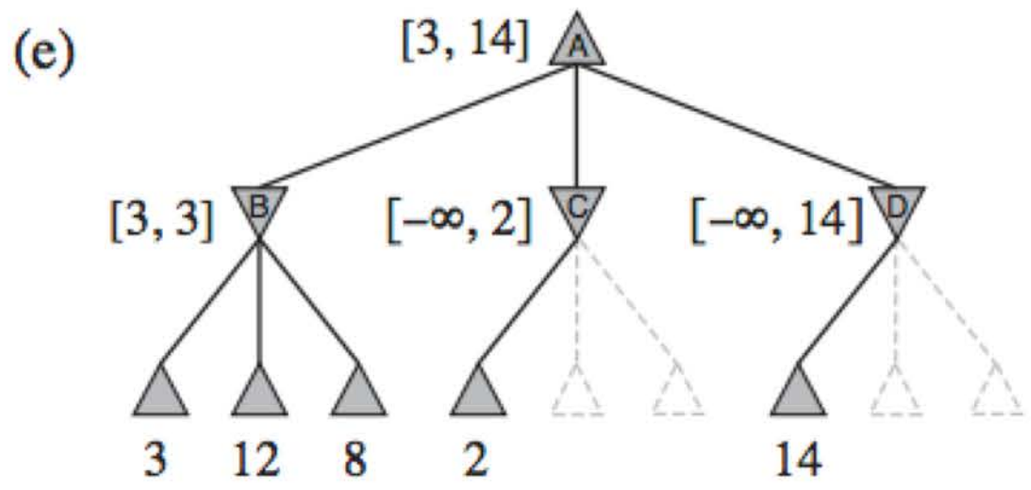
alpha–beta pruning

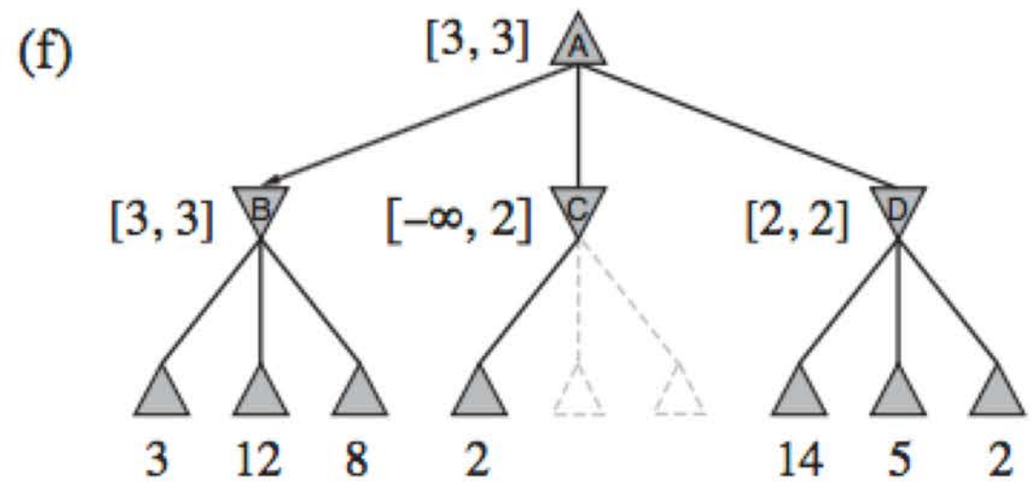












alpha–beta pruning

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.

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. The complete algorithm is given in Figure 5.7. We encourage you to trace its behavior when applied to the tree in Figure 5.5.

```
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).

Move ordering

- This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

- If this can be done, then it turns out that alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for minimax. This means that the effective Sqrt(b) instead of b—for chess, about 6 instead of 35. Put another way, alpha–beta can solve a tree roughly twice as deep as minimax in the same amount of time. If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b. For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case $O(b^{m/2})$ result.

内容提要 - 不完美的实时决策

- Claude Shannon’ s paper *Programming a Computer for Playing Chess* (1950) proposed instead that programs should cut off the search earlier and apply a **heuristic evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves.

- In other words, the suggestion is to alter minimax or alpha–beta in two ways: replace the utility function by a heuristic evaluation function EVAL, which estimates the position’ s utility, and replace the terminal test by a **cutoff test** that decides when to apply EVAL. That gives us the following for heuristic minimax for state s and maximum depth d:

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN.} \end{cases}$$

不完美的实时决策 - Evaluation functions

- **First**, the **evaluation function** should **order** the *terminal* states in the **same** way **as** the **true utility function**: states that are wins must evaluate better than draws, which in turn must be better than losses. Otherwise, an agent using the evaluation function might err even if it can see ahead all the way to the end of the game.
- **Second**, the **computation** must **not** take too **long**! (The whole point is to search faster.)
- **Third**, for nonterminal states, the **evaluation** function should be strongly **correlated** with the **actual** chances of winning.

不完美的实时决策 - Evaluation functions

- The evaluation function cannot know which states are which, but it can return a single value that reflects the *proportion* of states with each outcome.
- For example, suppose our experience suggests that 72% of the states encountered in the two-pawns vs. one-pawn category lead to a win (utility +1); 20% to a loss (0), and 8% to a draw (1/2). Then a reasonable evaluation for states in the category is the **expected value**: (0.72 × +1) + (0.20 × 0) + (0.08 × 1/2) = 0.76.

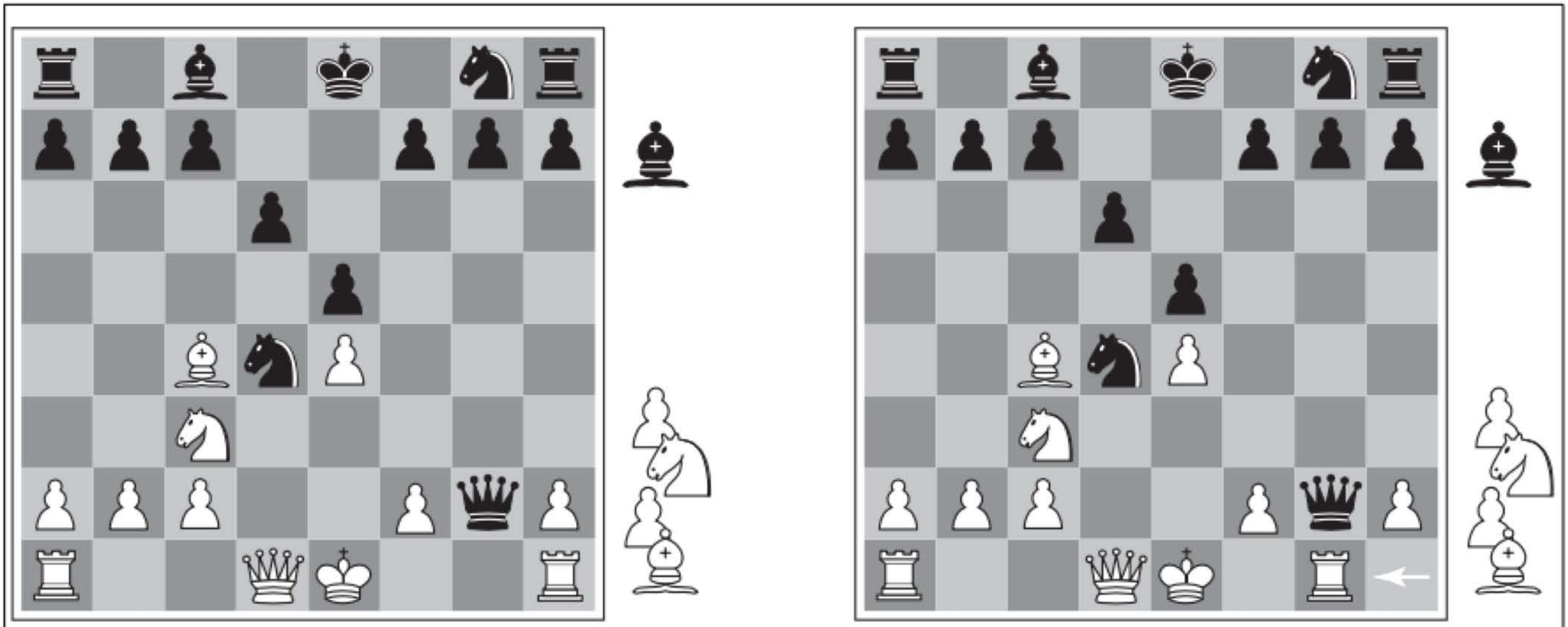
不完美的实时决策 - Evaluation functions

- Mathematically, this kind of evaluation function is called a **weighted linear function** because it can be expressed as

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s) \ ,$$

- Adding up the values of features seems like a reasonable thing to do, but in fact it involves a strong assumption: that the contribution of each feature is *independent* of the values of the other features. For example, assigning the value 3 to a bishop ignores the fact that bishops are more powerful in the endgame, when they have a lot of space to maneuver.

不完美的实时决策 - Evaluation functions



(a) White to move

(b) White to move

Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

不完美的实时决策 - Evaluation functions

- For this reason, current programs for chess and other games also use *nonlinear* combinations of features. For example, a pair of bishops might be worth slightly more than twice the value of a single bishop, and a bishop is worth more in the endgame (that is, when the *move number* feature is high or the *number of remaining pieces* feature is low).
- The astute reader will have noticed that the features and weights are *not* part of the rules of chess! They come from centuries of **human** chess-playing **experience**. In games where this kind of experience is not available, the weights of the evaluation function can be **estimated by the machine learning** techniques of Chapter 18. Reassuringly, applying these techniques to chess has confirmed that a bishop is indeed worth about three pawns.

不完美的实时决策 - **Cutting off search**

- The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. We replace the two lines in Figure 5.7 that mention TERMINAL-TEST with the following line:

- **if** CUTOFF-TEST(state, depth) **then return** EVAL(state)

- We also must arrange for some bookkeeping so that the current depth is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to **set a fixed depth limi**t so that CUTOFF-TEST(state, depth) returns true for all depth greater than some fixed depth d. (It must also return true for all terminal states, just as TERMINAL-TEST did.) The depth d is chosen so that a move is selected within the allocated time. A more robust approach is to apply iterative deepening. (See Chapter 3.) When time runs out, the program returns the move selected by the **deepest completed search**. As a bonus, iterative deepening also helps with **move ordering.**