Local Search Algorithms

Vijay Kumar Meena
Assistant Professor
KIIT University

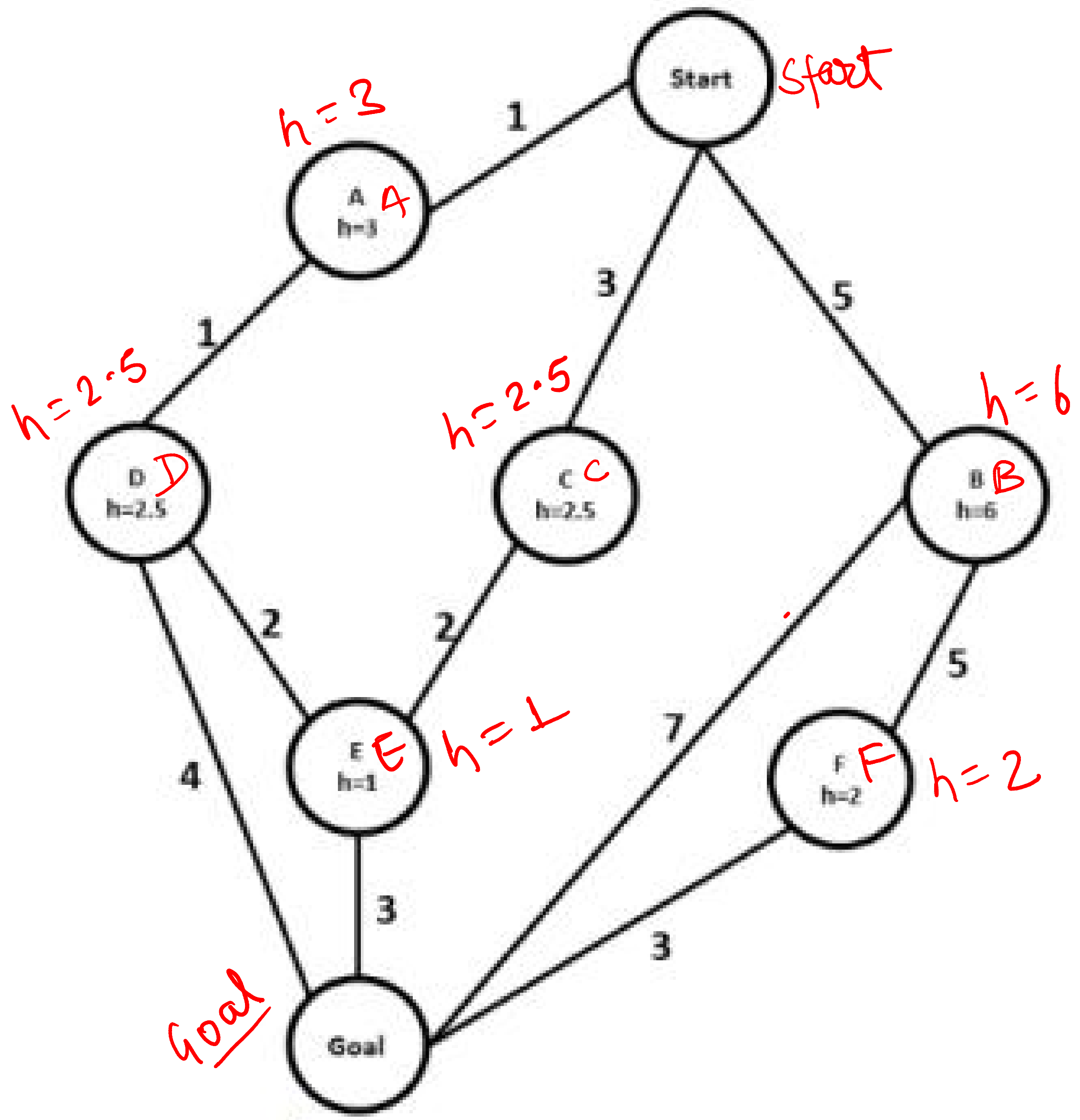
Acknowledgement

- These slides are taken (as it is or with some modifications) from various resources including -
  - Artificial Intelligence: A Modern Approach (AIMA) by Russell and Norvig.
  - Slides of Prof. Mausam (IITD)
  - Slides of Prof. Rohan Paul (IITD)
  - Slides of Prof. Brian Yu (Harvard University)

So Far…

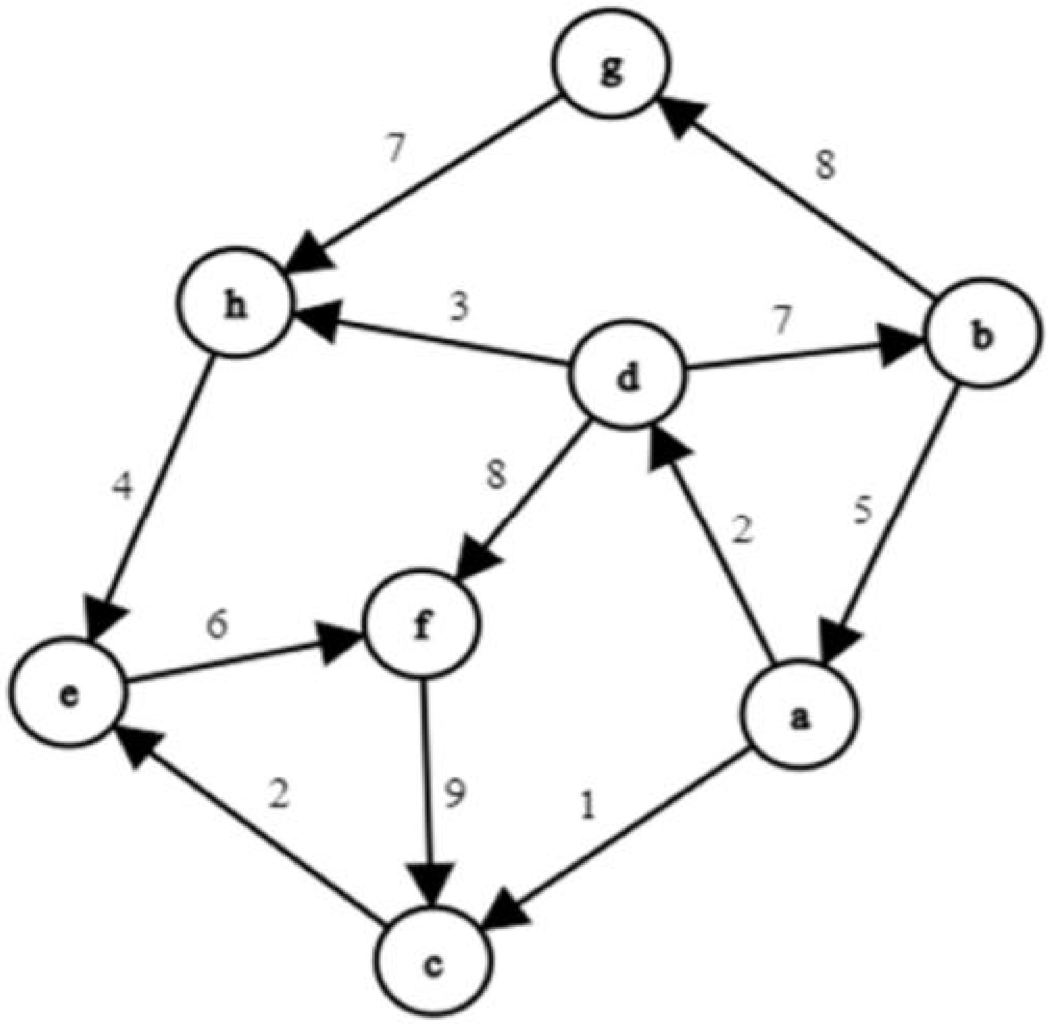
Exercise

- Write down the order in which states are being explored by each of following search algorithms?
  - Depth First Search
  - Breadth First Search
  - Iterative Deepening Search
  - Uniform Cost Search
  - Greedy Best First Search
  - A* Search



Exercise

3. Consider the state-space graph in the following figure and the heuristic values given in the table. Consider **d** as the start node and **c** as the goal node. Is the heuristic function admissible? Is it consistent? Explain your reasoning. [3+3]



n	d	a	b	e	f	g	h	c
h(n)	12	1	5	10	8	24	11	0

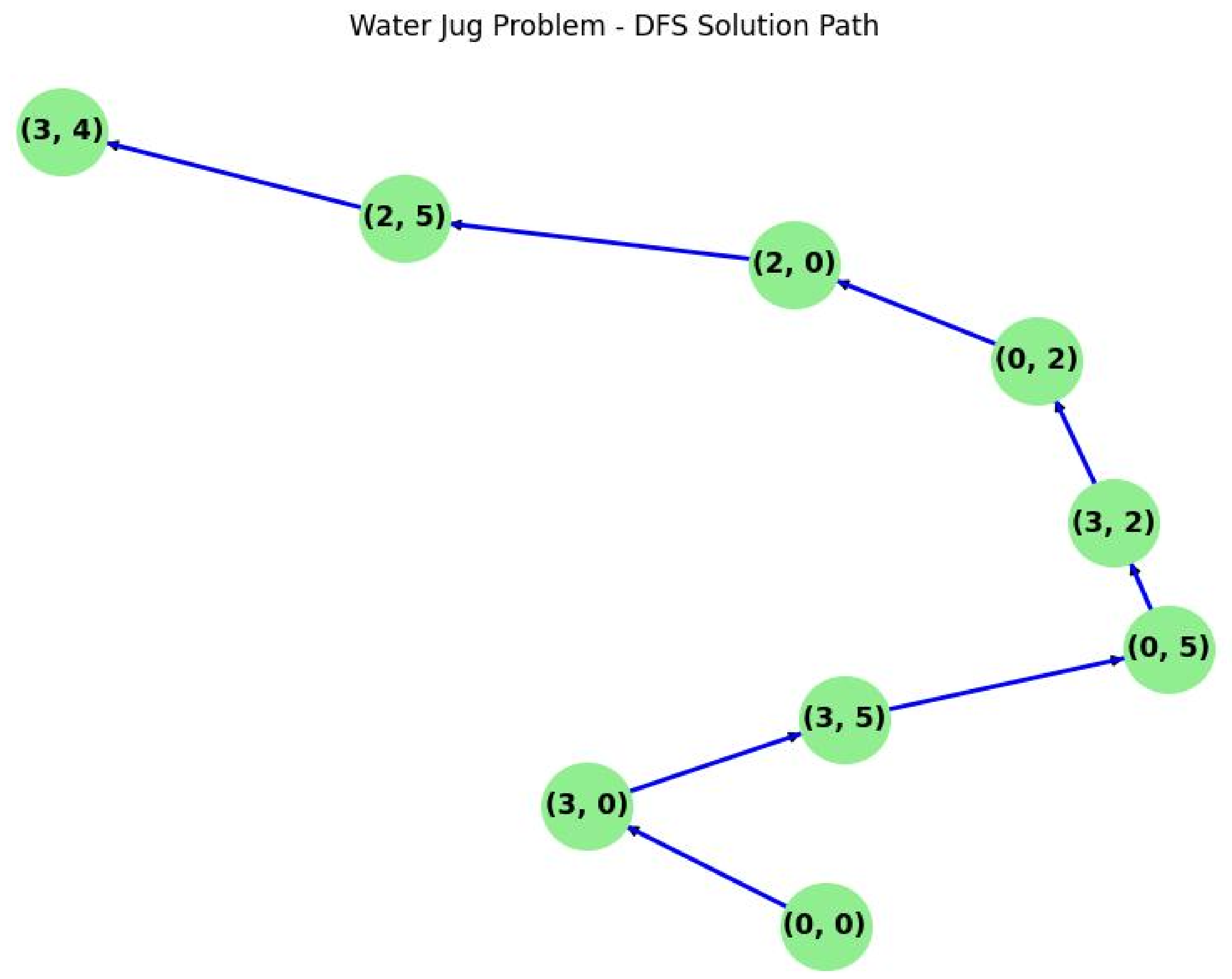
Exercise: Water Jug Problem

- You are given two jugs, one with a capacity of X liters and the other with a capacity of Y liters.
- You need to measure exactly Z liters of water using these two jugs.
- The allowed operations are:
  - Fill one of the jugs.
  - Empty one of the jugs.
  - Pour water from one jug into another until one jug is either full or empty.
- For instance, given two jugs with capacities of 3 liters and 5 liters, and a goal of measuring 4 liters, the search for a solution begins from the initial state and moves through various possible states by filling, emptying, and pouring the water between the two jugs.
- Model Water Jug Problem as a search problem and find solution using DFS.

Solution: Water Jug Problem

- We represent each state as a pair (x, y) where:
  - x is the amount of water in the 3-liter jug.
  - y is the amount of water in the 5-liter jug.
- The initial state is (0, 0) because both jugs start empty, and the goal is to reach any state where either jug contains exactly 4 liters of water.
- The following operations define the possible transitions from one state to another:
  - Fill the 3-liter jug: Move to (3, y).
  - Fill the 5-liter jug: Move to (x, 5).
  - Empty the 3-liter jug: Move to (0, y).
  - Empty the 5-liter jug: Move to (x, 0).
  - Pour water from the 3-liter jug into the 5-liter jug: Move to (max(0, x - (5 - y)), min(5, x + y)).
  - Pour water from the 5-liter jug into the 3-liter jug: Move to (min(3, x + y), max(0, y - (3 - x))).

Solution: Water Jug Problem



Local search

- So far we have solved problems which are in observable, deterministic and known environment where the solution is a sequence of actions.
- 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 goal is found, the *path* to that goal also constitutes a *solution* to the problem.
- In many problems however, the path to 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

- If path to goal node does not matter, only finding goal node matters then we can consider different class of search algorithms that does not worry about paths at all.
- **Local search** algorithms operate using a single **current node** rather than multiple paths and generally move only to neighbours of that node only.
- The paths followed by Local search algorithms are not retained therefore they have no memory of past decisions.

Local search

- Although local search algorithms are not systematic, they have two key advantages –
  - They use very little memory --- Usually a constant amount of memory to keep track of current node
  - They can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic search 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 we have talked about.
- 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

- State-Space for local search can be seen as a landscape which has a location (defined by the current state) and an elevation (defined by the value of the heuristic cost function or objective function).
- If elevation is defined as a **cost function** then our goal is to minimize the cost therefore **global minima of landscape** will give the **optimal solution**.
- If elevation is defined as an **objective function** then our goal is to **maximize the objective value** or we have to find global maxima in the landscape.
- 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 Landscape

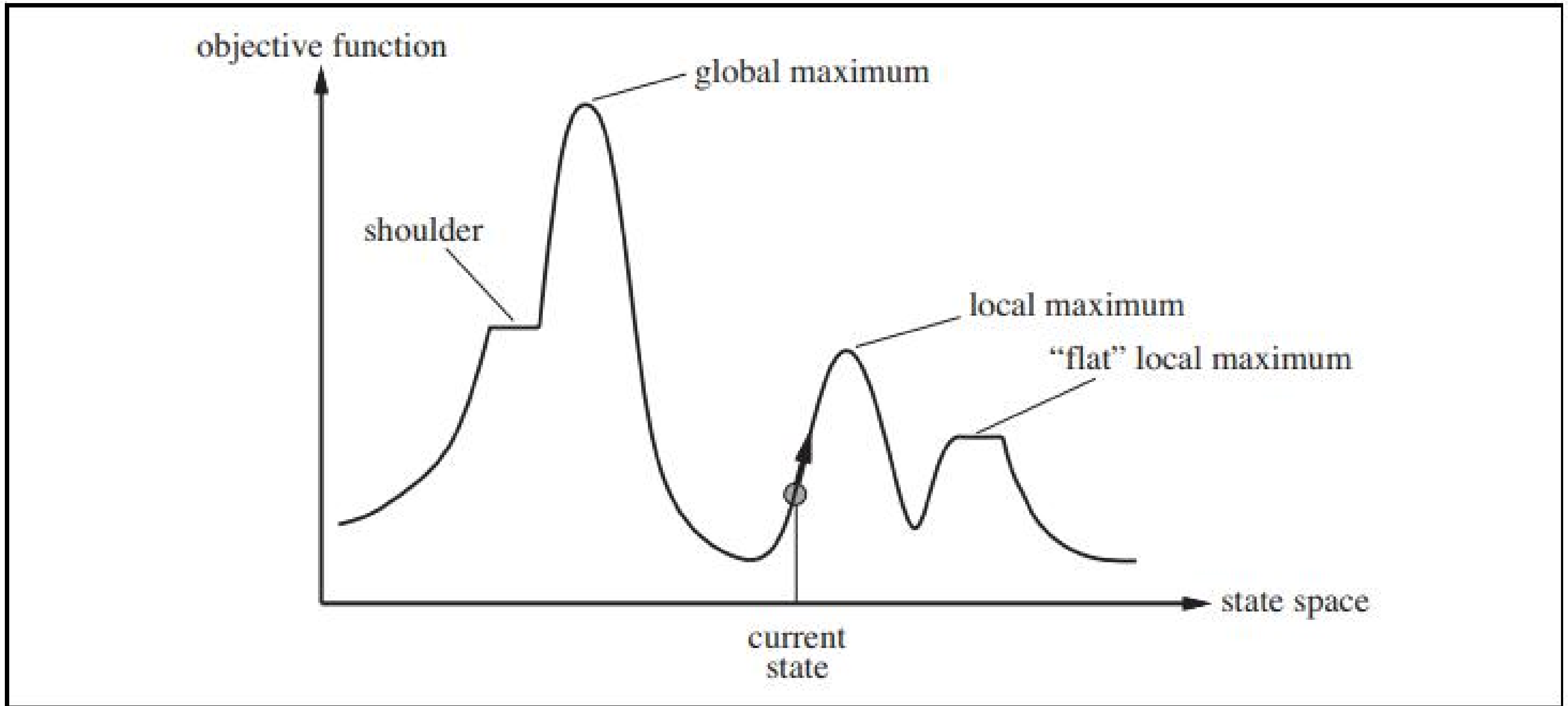
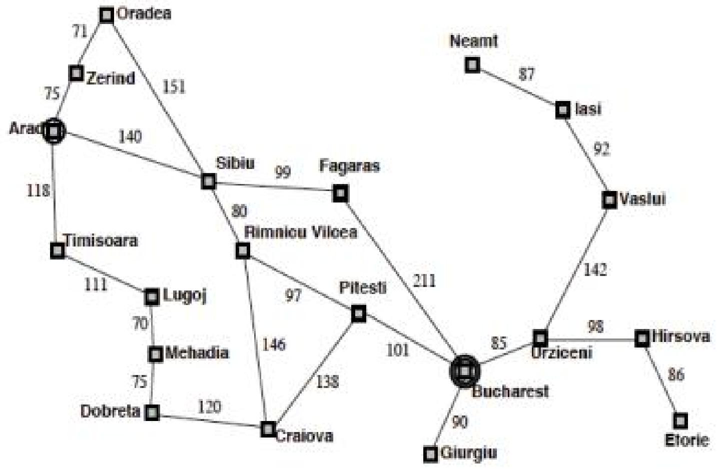


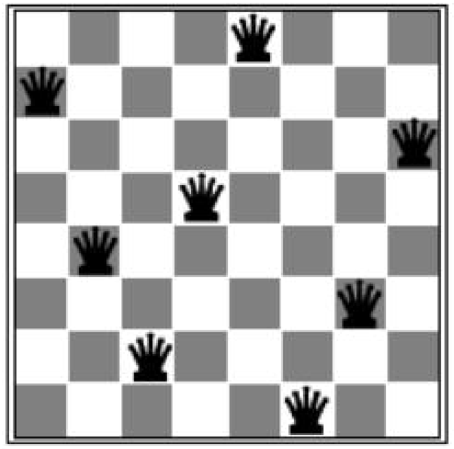
Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Planning vs optimization

- **Planning**
  - Explore paths in the search space and want the optimal path to the goal
  - When a goal is found, the path to the goal constitutes the solution.



- **Optimization**
  - Unimportant how the goal was reached.
  - Only reaching the goal matters.
  - 8-queens, VLSI layout etc.



Local Search Methods

- We have just one solution in mind, and we look for alternatives in the vicinity of that solution

- Generic Local Search Algorithm

1. Start from an initial configuration X_0
2. Repeat until *satisfied*:
   (a) Generate the *set of neighbors* of X_i and evaluate them
   (b) *Select* one of the neighbors, X_{i+1}
   (c) The selected neighbor becomes the current configuration

Variations with choice of *next states, selection* of next state and *satisficing* criteria.

Hill-climbing search

- Hill-Climbing Search is a local search algorithm which tries to maximize the objective function for a state.
- At any step, Hill-climbing moves in the direction of increasing objective value i.e. at each step, it selects node whose objective value is higher than current node.
- If it can’t find neighbouring node with higher objective value then it terminates.

Hill-climbing search

- Hill climbing does not maintain a search tree. It only maintains a node data structure which holds the value regarding current state and objective function value.
- Hill climbing does not look ahead beyond the immediate neighbours 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: Steepest-Ascent Version

```
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
```

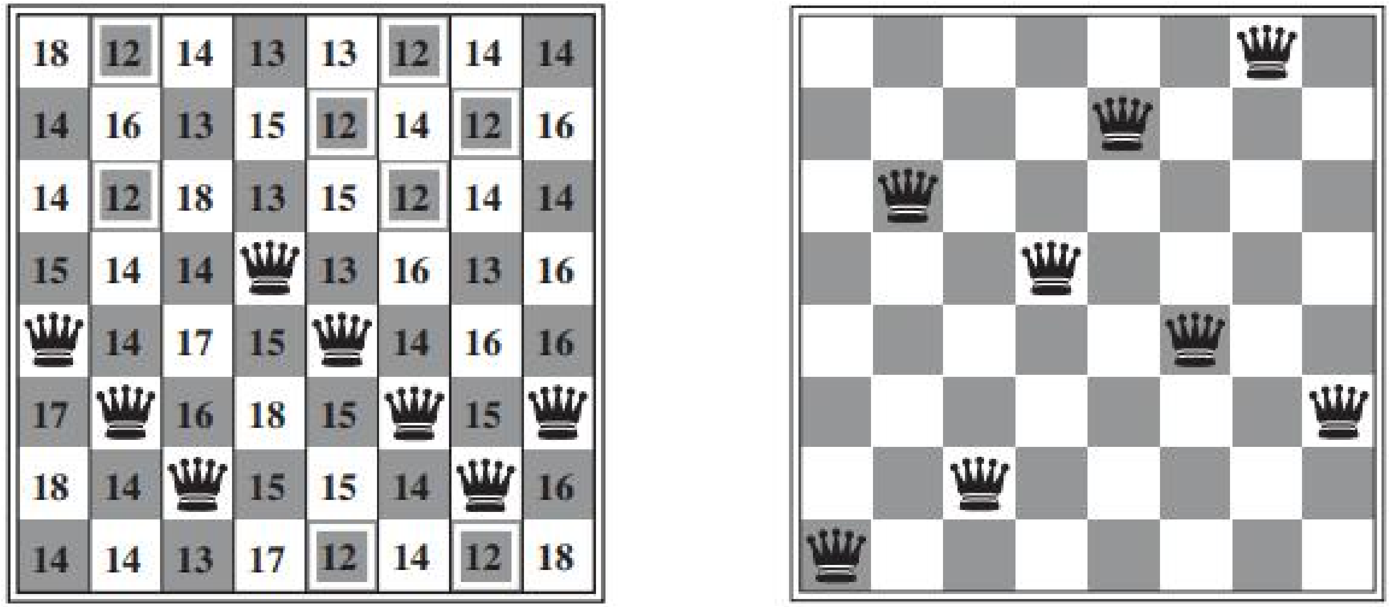
Hill-Climbing Search: 8-queen problem

- Suppose you are given a 8x8 chessboard and you have place 8-queens on the board such that no queen attack each other.
- How can we convert it to an optimization problem?
  - Think of a state
  - Function or criteria that we need to optimize.

Hill-Climbing Search: 8-Queen Problem

- State Space
  - All 8 queens on the board in some configuration.
- Successor Function or Transition Model
  - Move a single queen to another square in the same column.
  - Each state has 8 x 7 = 56 successors.
- Objective or Evaluation or heuristic cost function h(n)
  - The number of pairs of queens that are attacking each other either directly or indirectly.
  - We want to minimize this heuristic value.
  - The global minimum of this function is zero, which occurs only at perfect solutions.

Hill-climbing: 8-queen problem



(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.

Hill-Climbing Search

- Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one i.e. break the ties randomly.
- Hill climbing is sometimes called greedy local search because it grabs a good neighbour 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.

Hill-climbing: 8-queen problem

- Hill climbing only takes five steps to go from state on left (h = 17) to state on right (h = 1) which is nearly an optimal solution.

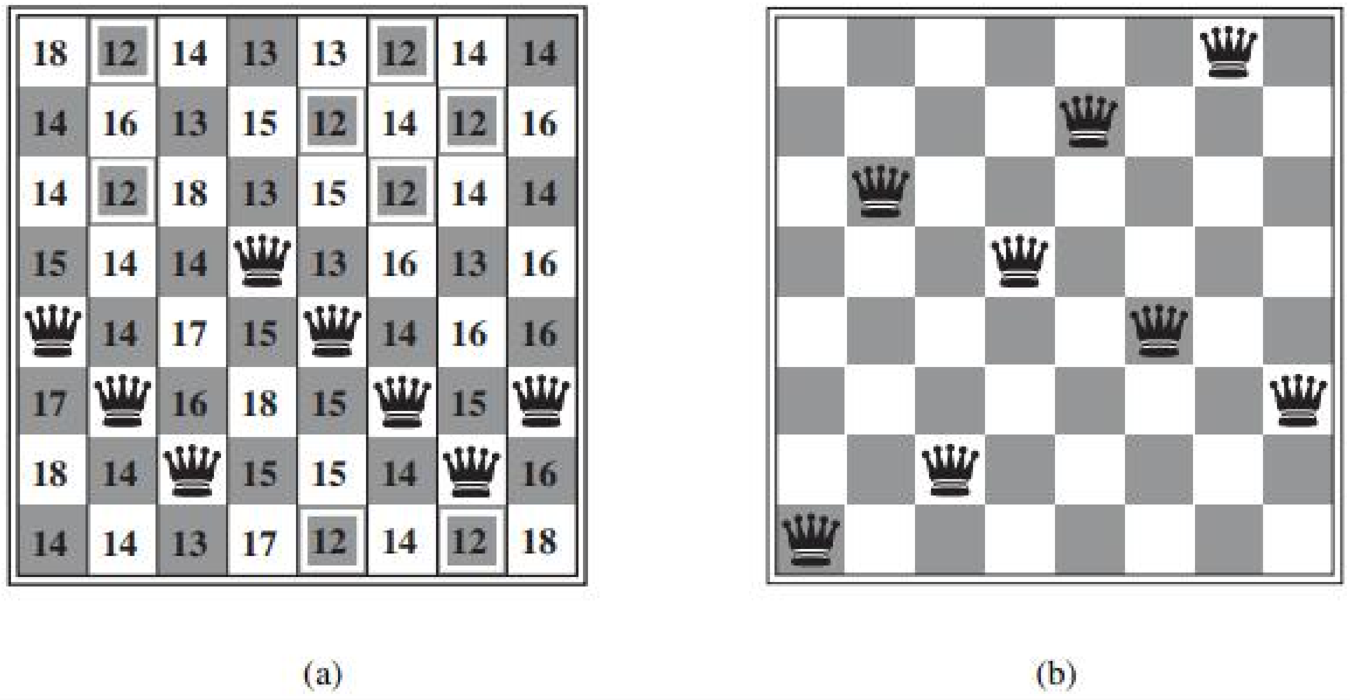


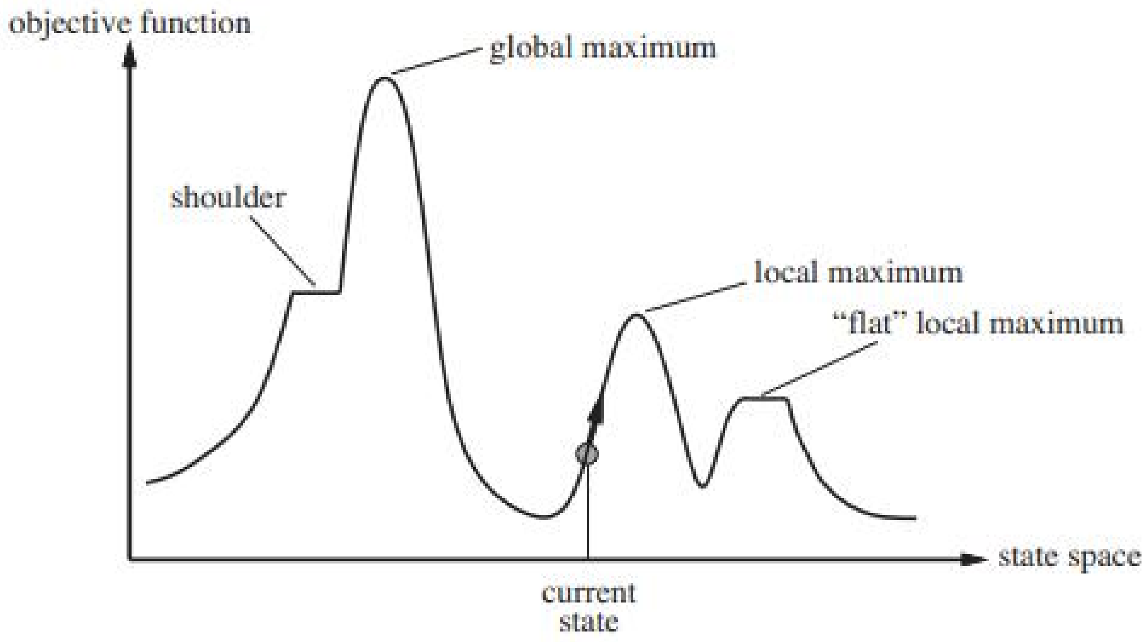
Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Hill-climbing: Limitations

The performance of Hill-climbing algorithms depends upon how state-space landscape is distributed. Even though Hill-climbing algorithms are very fast, they often gets stuck because of following reasons –

- **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.
- **Ridges** - A sequence of local maxima, that are not directly connected to each other. From each local maximum, all the available actions point downhill.
- **Plateau** – 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.

Hill-Climbing: Limitations



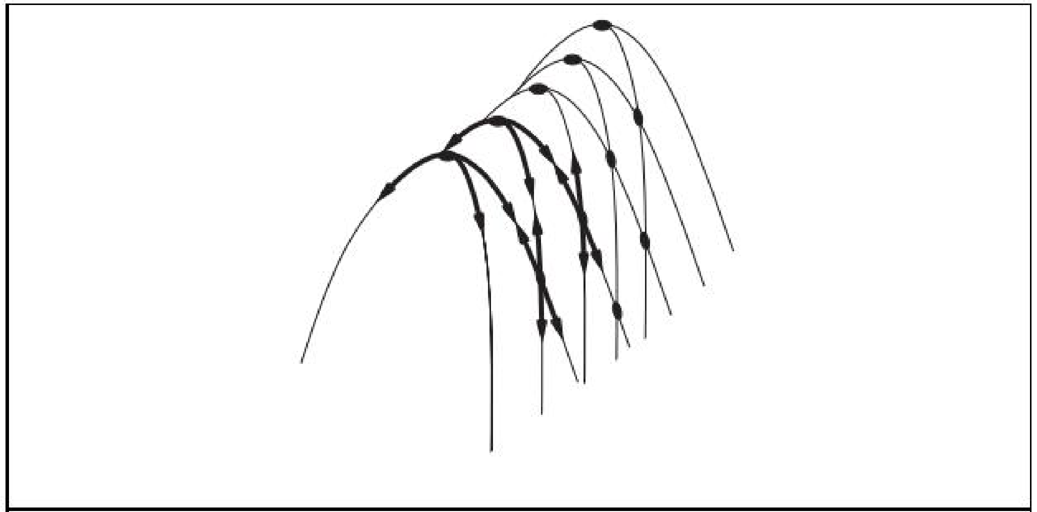


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point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

Hill-climbing: 8-queens problem

- Starting from a randomly generated 8-queens state, the steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances.
- But it works quickly, taking just 4 steps on average solve problem when it succeeds and only 3 steps when it gets stuck – not bad for a state space with $8^8 \approx 17\ million\ states$.
- Can we do something about it?

Hill-climbing: Sideways Moves Variant

- If Hill-climbing algorithm is stuck on plateau i.e. there are no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape from plateau, if the plateau is a shoulder.
- But we need to restrict number of sideways moves, otherwise algorithm might get stuck in an infinite loop.
- For 8-queens problem -
  - Allowing sideways moves with a limit of 100 improves percentage of problem instances solved by hill climbing from 14 to 94%.
  - Success comes at a cost: the algorithm takes roughly 21 steps on average for each successful instance to find the solution and 64 steps for each failure.

Hill-climbing variants

- **Stochastic Hill Climbing**
  - Instead of choosing best neighbouring state, choose a state randomly from all better neighbours.
  - The probability of selection can depend upon the quality of the neighbouring state.
  - The algorithm converges more slowly than steepest ascent hill-climbing (choosing best successor) but in some state space landscapes, it finds better solutions.
- **First-choice Hill Climbing**
  - Implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
  - Good strategy when a state has many successors.

Hill-climbing variants: random-restart

- The hill-climbing algorithms described so far are incomplete because they can get stuck to local maxima.
- Random-restart hill climbing performs a series of hill-climbing searches from randomly generated start states 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.

Random-restart hill climbing: 8-queens

- For 8-queens instances with no sideways moves allowed, $p \approx 0.14$ so we need roughly 7 iterations to find a goal.
- 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 $(1 * 21) + \left(\frac{0.06}{0.94}\right) * 64 \approx 25\ steps$.
- For n-queens problem, random-restart hill climbing is very effective. Even for three million queens, the algorithm can find solutions in under a minute.

Hill-climbing Conclusion

- The success of hill climbing depends very much on the shape of the state-space landscape.
- If there is a problem whose state-space landscape has few local maxima and plateaux then random-restart hill climbing will find a good solution very quickly.
- Even if state-space landscape is more scattered, hill-climbing algorithms can find a reasonably good local maximum in small number of steps.

Simulated Annealing

- A hill-climbing algorithm that never makes “downhill” moves toward states with lower objective value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum (or local minimum).
- On other hand, a purely random walk – that is selecting successor randomly – is complete but extremely inefficient.
- Simulated Annealing combines functionalities of both hill-climbing and random walk giving both complete and efficient algorithm.
- In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a low energy crystalline state.

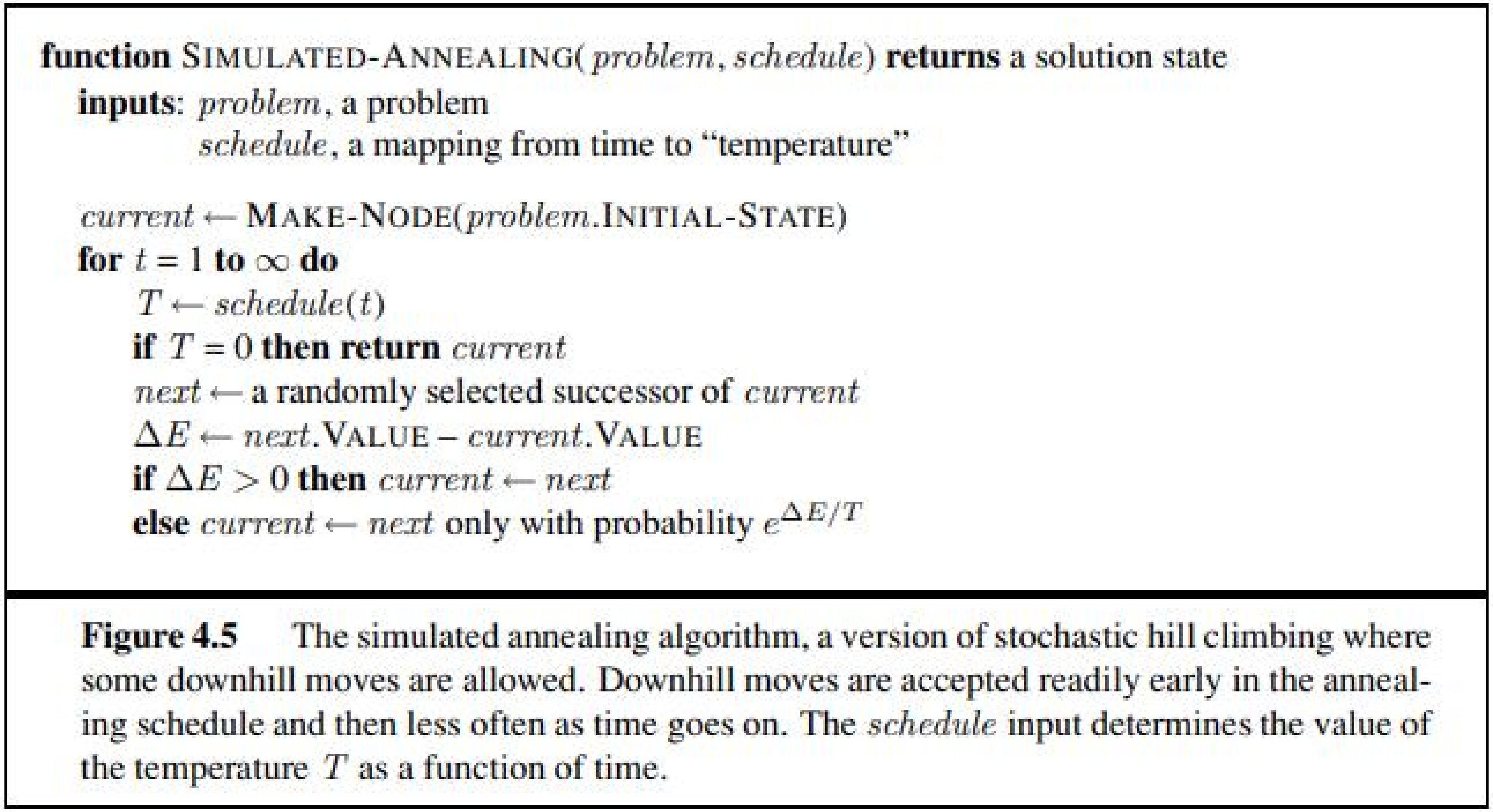
Simulated annealing

- Basic Ideas:
  - Like hill-climbing identify the quality of the local improvements
  - Instead of picking the best move, pick one randomly
  - Say the change in objective function is $\Delta E = neighbour.Value - current.Value$
  - If ΔE is positive, then move to that state
  - Otherwise –
    - Move to this state with probability proportional to ΔE
    - Thus: worse moves (very large negative ΔE) are executed less often
  - However, there is always a chance of escaping from local maxima
  - Over time, make it less likely to accept locally bad moves

Simulated Annealing

- Imagine letting a ball roll downhill on the function surface
  - This is like hill-climbing (for minimization)
- Now imagine function surface is vibrating while the ball rolls, gradually reducing the vibrations with time
  - This is like simulated annealing

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 $\infty$ do
      $T$ ← schedule($t$)
      if $T = 0$ then return current
      next ← a randomly selected successor of current
      $\Delta E$ ← next.VALUE – current.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.

Local Beam Search

- Hill climbing keeps only one node in memory which might seem to be an extreme reaction to memory limitations.
- Local Beam Search algorithm keeps track of ***k states*** rather than just one state.
- It begins with *k randomly* generated states and at each step, all the successors of all k states are generated.
- If any one is a goal then the algorithm returns the solution otherwise it selects *k best successors* from the list and repeats.

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** can help reducing this problem. Instead of choosing the best k from 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).

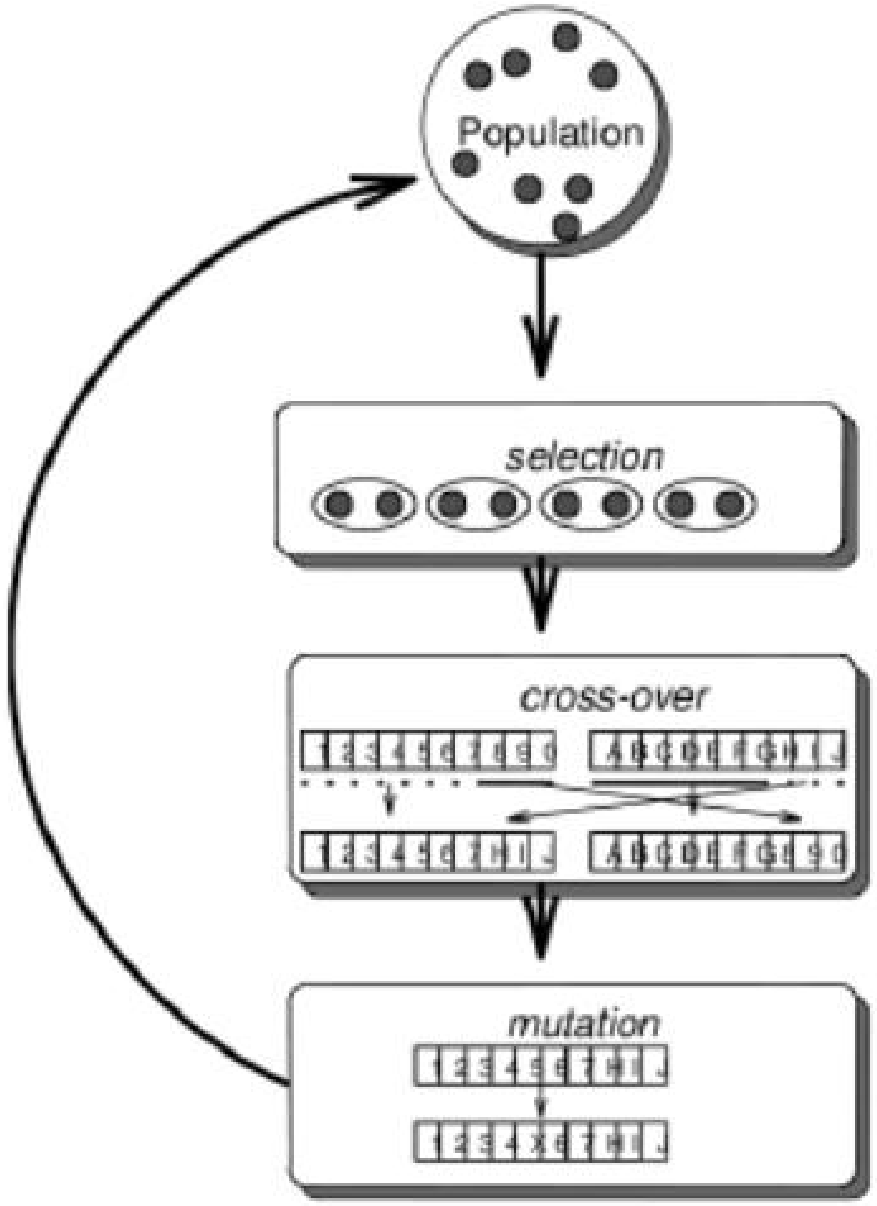
Genetic Algorithms (GA)

- A genetic algorithm is a variant of stochastic beam search in which successor states are generated by combining two parents states rather than by modifying a single state.
- Just like beam search, Genetic algorithms also starts with a set of k randomly generated states called **population.**
- Each state, or individual is represented as a string over a finite alphabet – most 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 $8 \times \log2\ 8 = 24$ bits.
- Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8.

Genetic Algorithms (GA)

- Each state is rated by the objective function or the **fitness function** in GA terminology.
- A fitness function should return higher values for better states. So for 8-queens problem, we can use the *number of non-attacking pairs of queens* as fitness function, which has a value of 28 for a solution.
- In genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score.

Genetic algorithms



Genetic algorithms

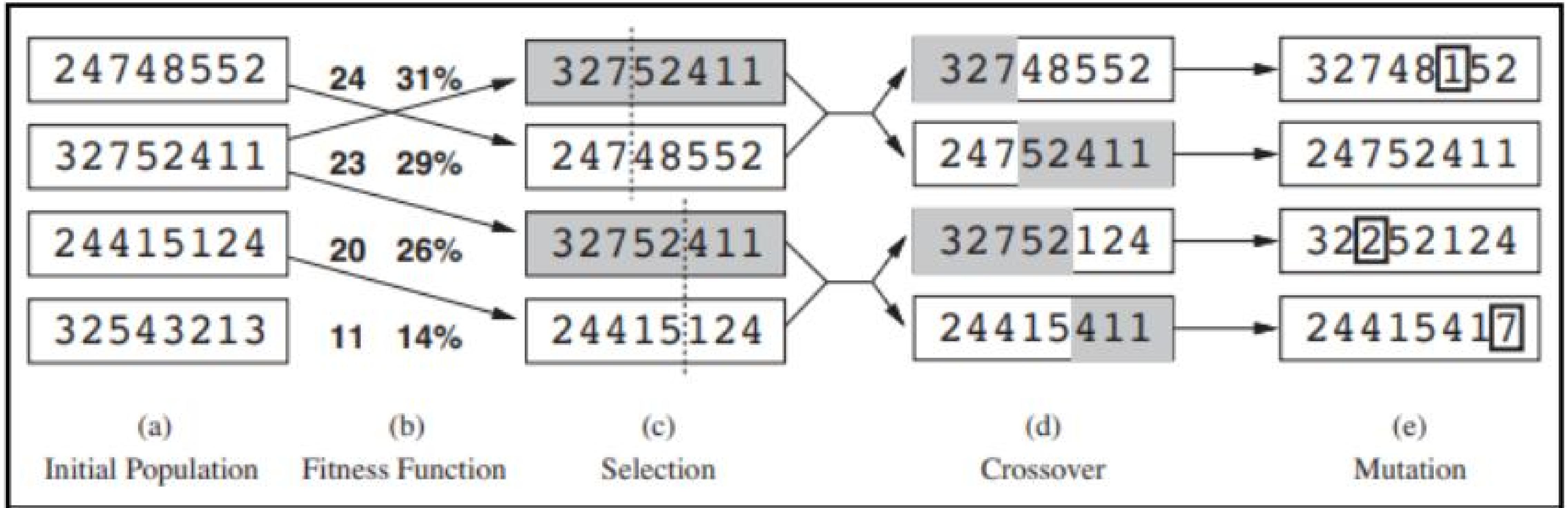
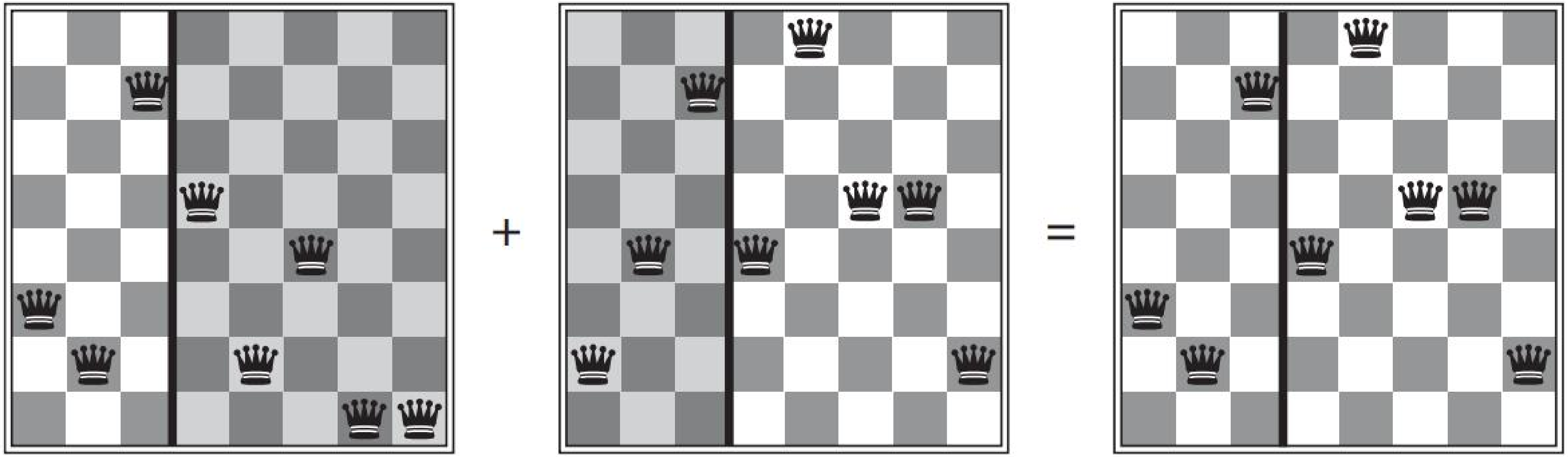
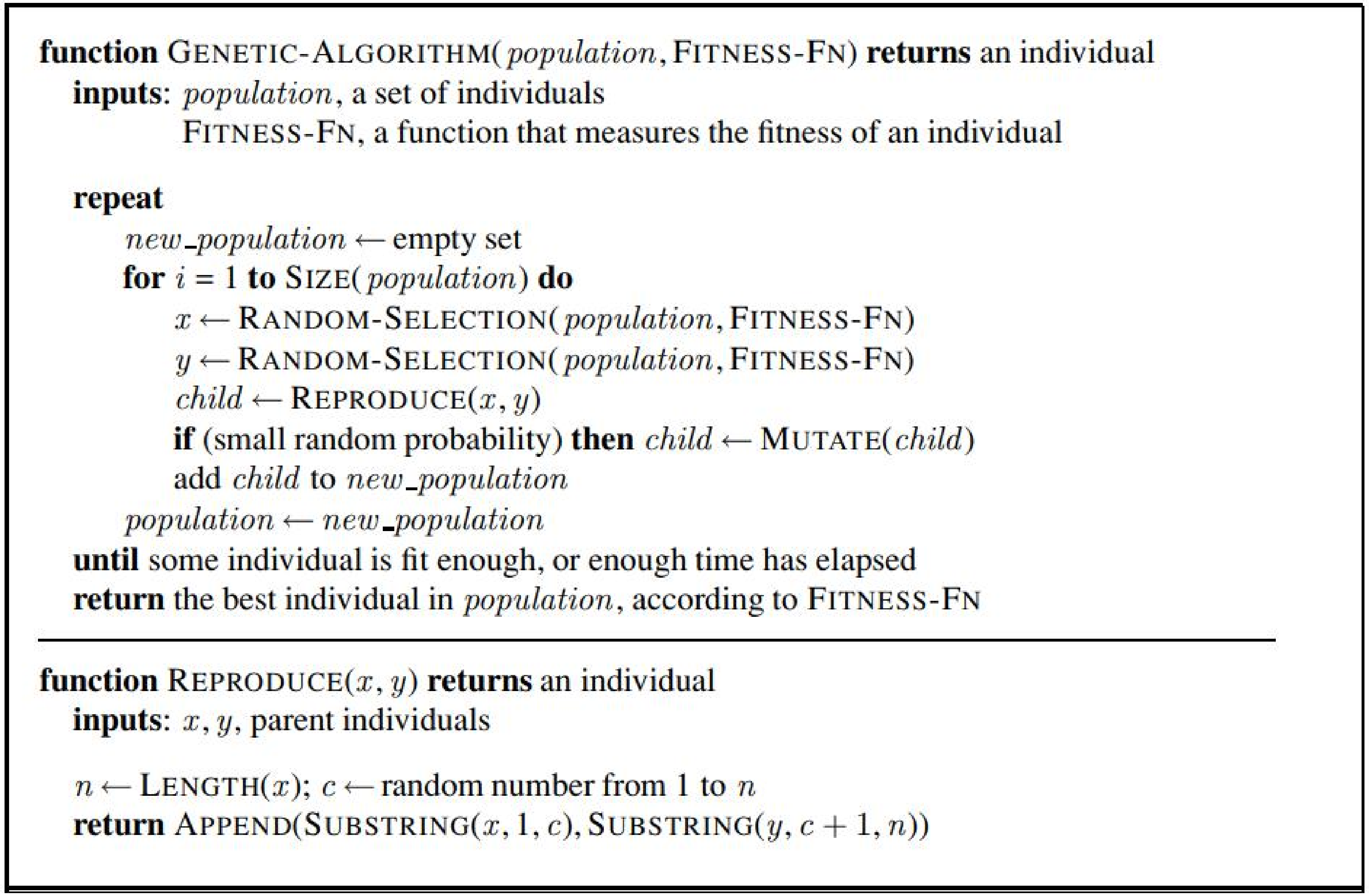


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Genetic algorithms



Genetic algorithms



```
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))
```

Genetic Algorithms

- Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads.
- 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 aesthetically pleasing origins in the theory of evolution.