Artificial Intelligence

[week #4]

Local Search

Hilmy. A. T

hilmi.tawakal@gmail.com

Finding solution with search

- **classical search:**
  - Design to explore search space sistematically
  - observable, deterministic, known environments
  - the solution is a sequence of actions.
- **What if:**
  - 10^{10000} state
  - state space initially unknown
  - Partially observe
  - cannot predict exactly what percept it will receive

Local Search

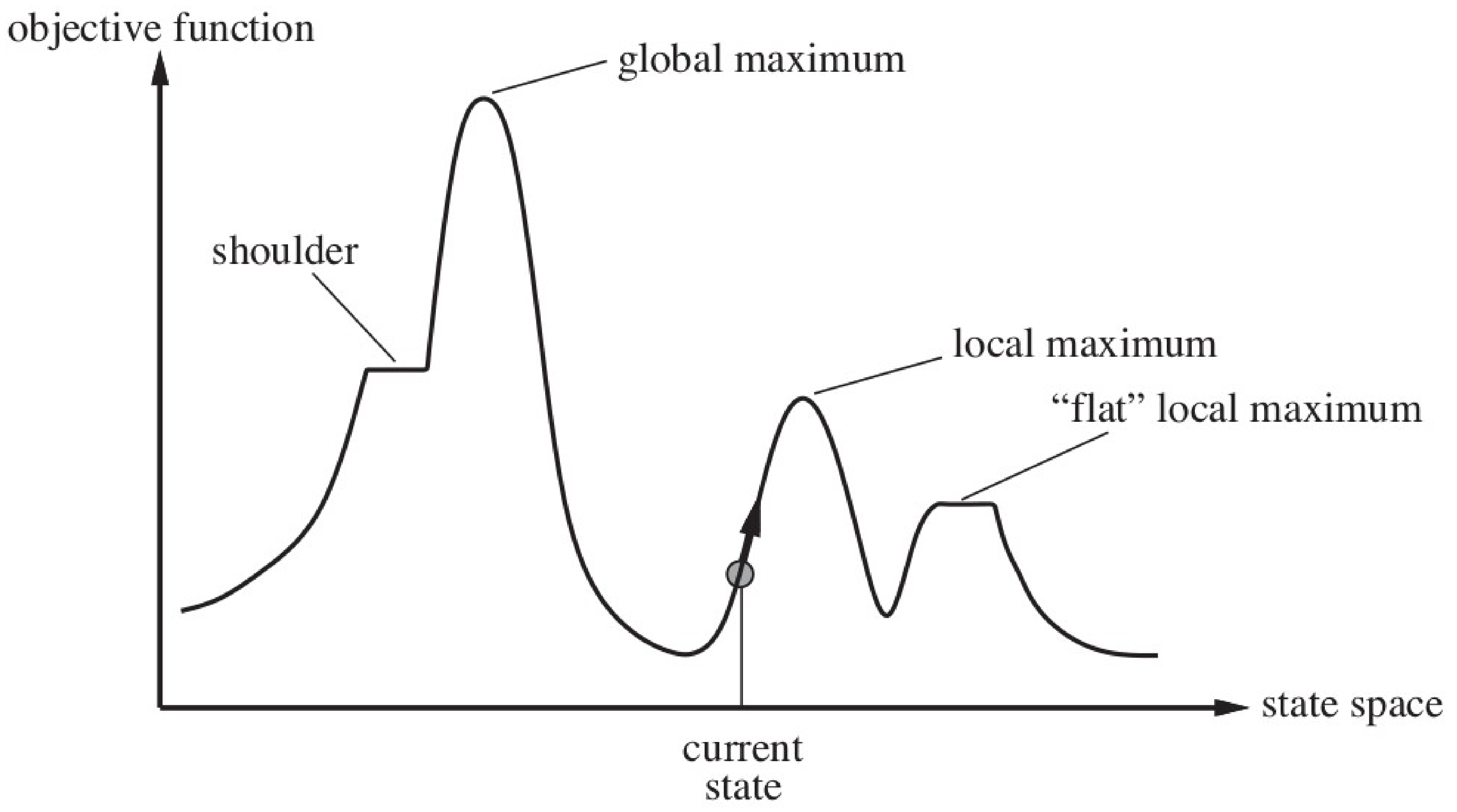
- Evaluating and modifying one or more current states
- Rather than systematically exploring paths from an initial state
- Suitable for problems in which all that matters is the solution state
- Not the path cost to reach it (remember n-queens problem?)

Local Search

- Operate using a single current node (rather than multiple paths)
- Although local search algorithms are not systematic, they have two key advantages:
  1) use very little memory
  2) can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- Useful for solving pure optimization problems

  → find the best state according to an objective function

Finding Maximum



Hill-climbing Search

- Simply a loop that continually moves in the direction of increasing value—that is, uphill.
- Terminates when it reaches a “peak” where no neighbor has a higher value.
- Does not maintain a search tree, only record the state and the value of the objective function
- Does not look ahead beyond the immediate neighbors of the current state.

  → 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
```

Hill-climbing Search

- Sometimes called greedy local search
  - → grabs a good neighbor state without thinking ahead about where to go next.
- makes rapid progress toward a solution because it is usually quite easy to improve a bad state.
- Unfortunately, hill climbing often gets stuck by:
  - → Local maxima
  - → Ridges
  - → Plateaux

N-queens problem

- 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.

Overcome problems

- Enable sideway moves (overcome plateau)
- Allow up to, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage from 14% to 94%
- Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

Hill-climbing variation

- **Stochastic hill-climbing:** chooses at random from among the uphill moves
- **First-choice hill climbing:** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- **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

Simulated annealing

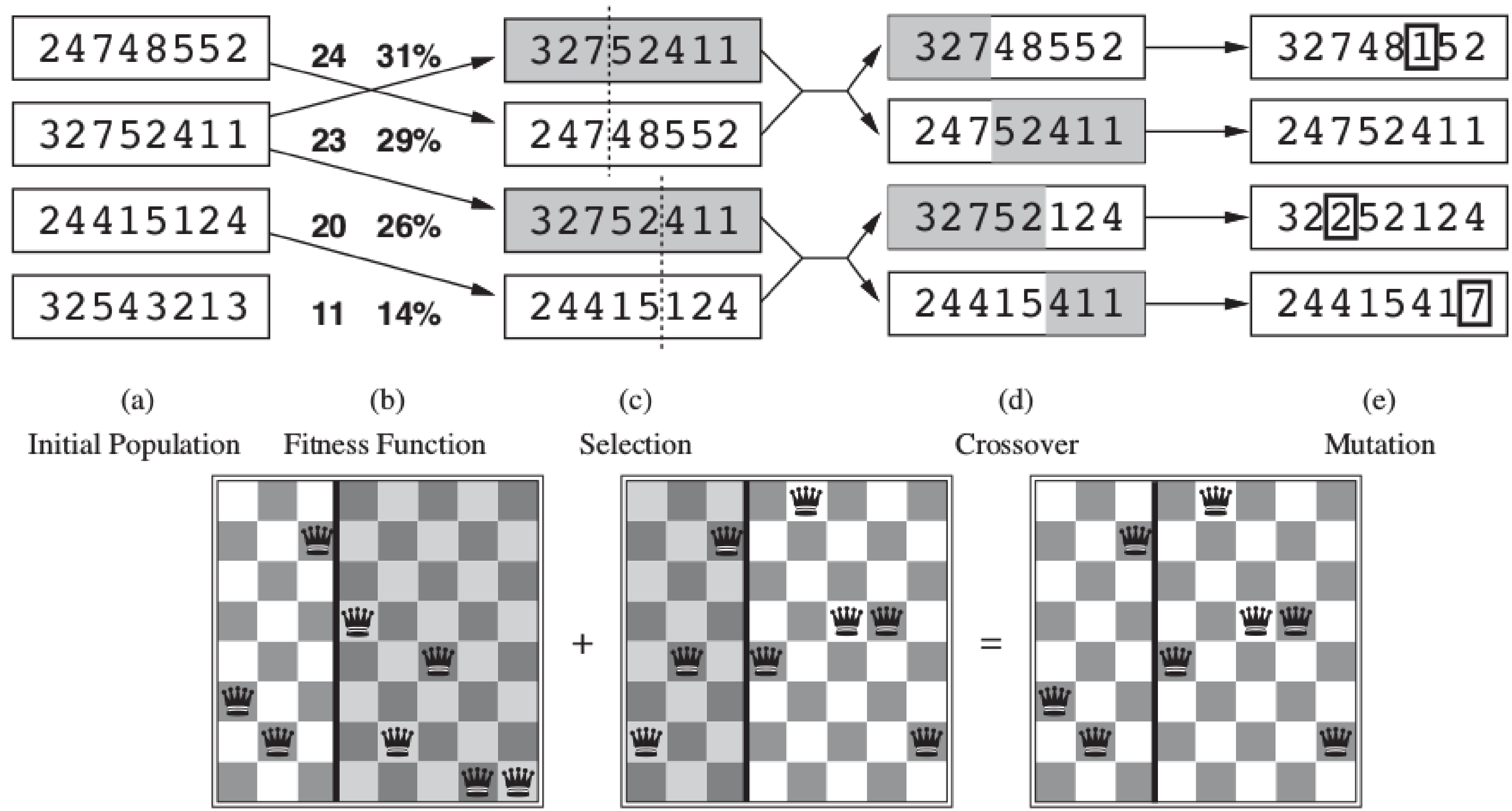
- A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.
- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is 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

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*.VALUE – *current*.VALUE
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

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))
```