CS 188: Artificial Intelligence

Local search

Instructor: Gholamreza Ghassem-Sani

Sharif University of Technology

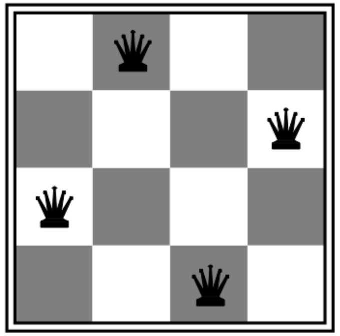
[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]

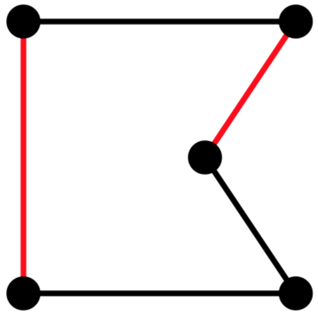
Generated and Test

- Algorithm
  1. Generate a (potential goal) state:
     - Particular point in the problem space, or
     - A path from a start state
  2. Test if it is a goal state
     - Stop if positive
     - go to step 1 otherwise
- Systematic or Heuristic?
  - It depends on “Generate”

Local search algorithms

- In many optimization problems, ***path*** is irrelevant; the goal state ***is*** the solution
- Then state space = set of "complete" configurations; find ***configuration satisfying constraints***, e.g., n-queens problem; or, find ***optimal configuration***, e.g., travelling salesperson problem





- In such cases, can use ***iterative improvement*** algorithms: keep a single "current" state, try to improve it
- Constant space, suitable for online as well as offline search

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

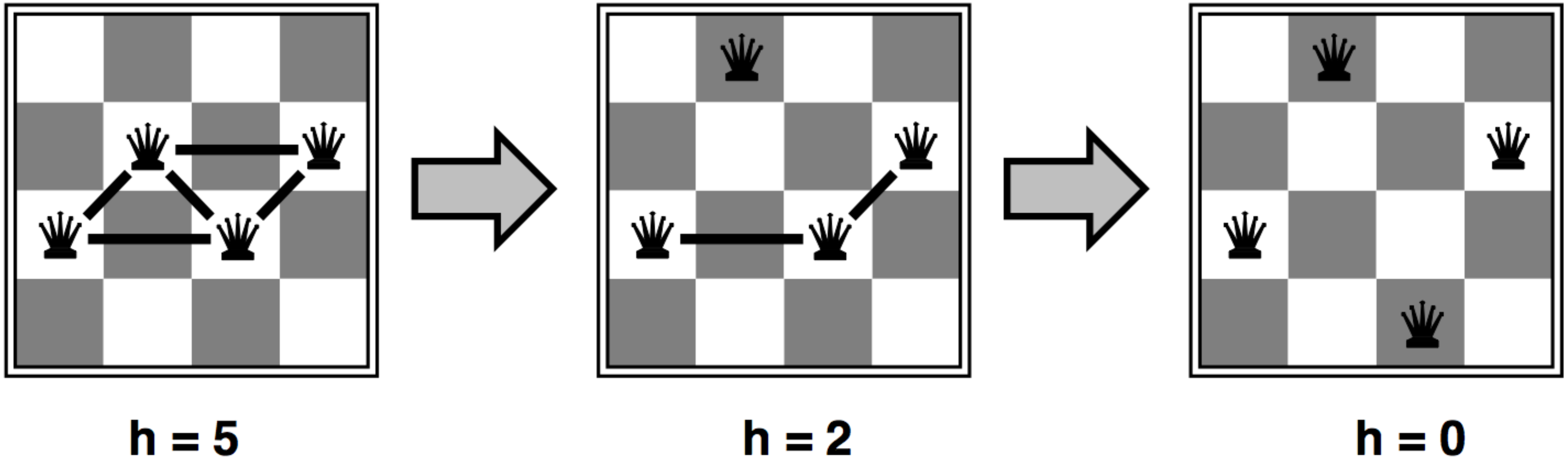
Hill Climbing

- Simple Hill Climbing
  - expand the current node
  - evaluate its children one by one (using the heuristic evaluation function)
  - choose the FIRST node with a better value

- Steepest Ascend Hill Climbing
  - expand the current node
  - Evaluate all its children (by the heuristic evaluation function)
  - choose the BEST node with the best value

Heuristic for *n*-queens problem

- Goal: n queens on board with no ***conflicts***, i.e., no queen attacking another
- States: n queens on board, one per column
- Actions: move a queen in its column
- Heuristic value function: number of conflicts

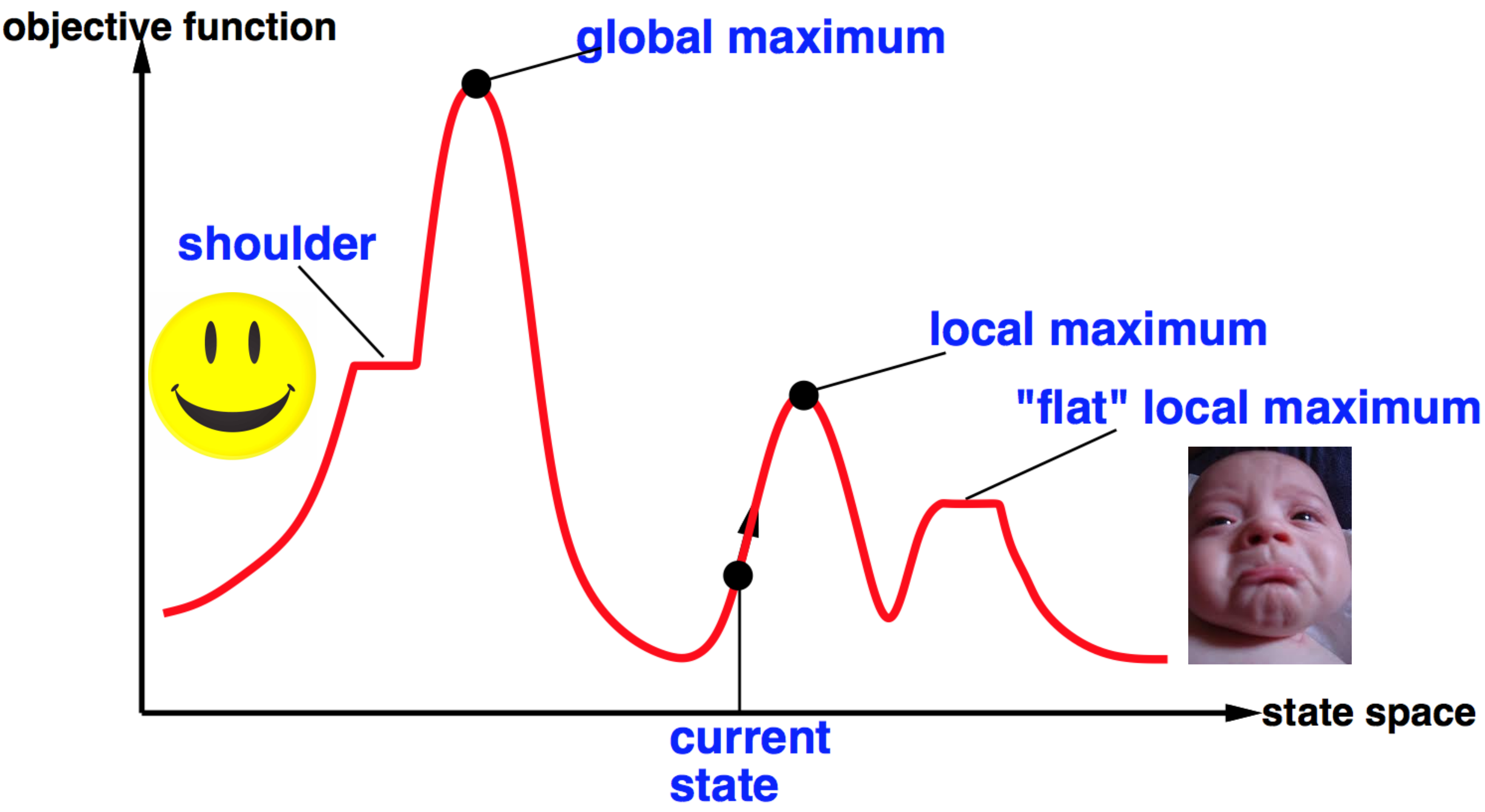


Hill-climbing algorithm

```
function HILL-CLIMBING(problem) returns a state
    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
```

"Like climbing Everest in thick fog with amnesia"

Global and local maxima



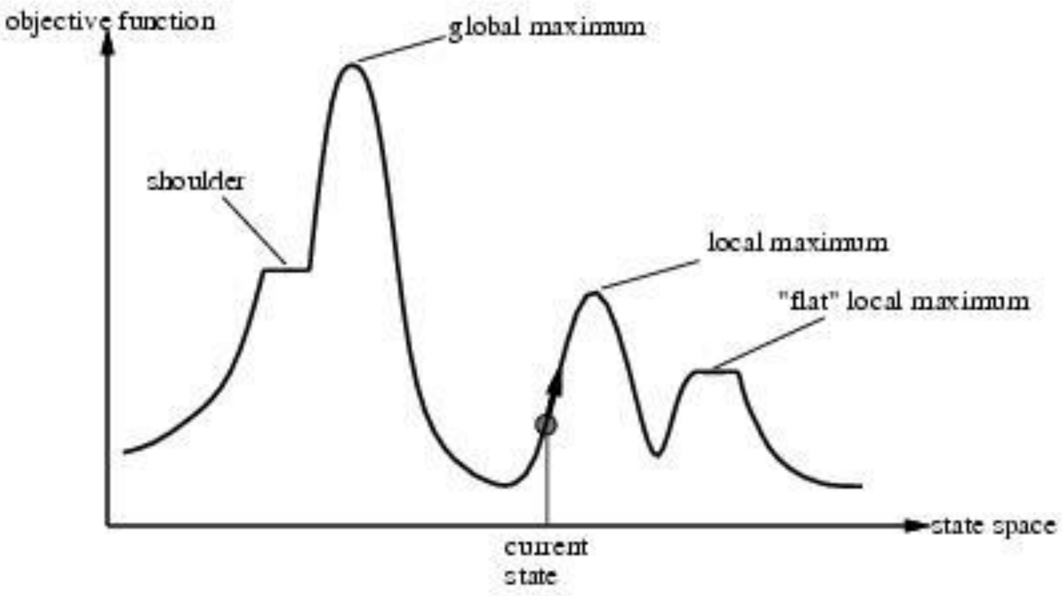
Random restarts

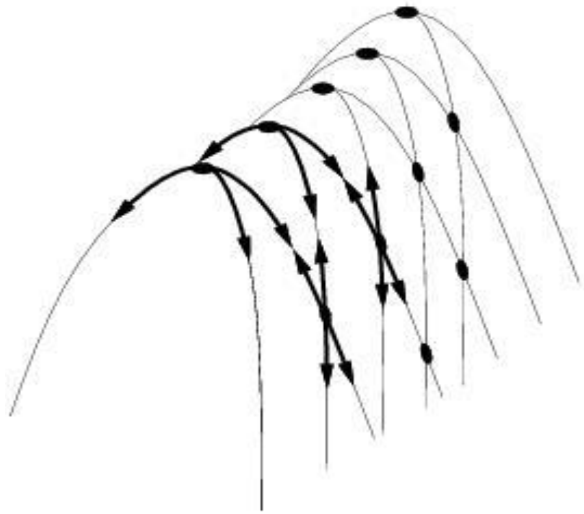
- find global optimum
- duh

Random sideways moves

- Escape from shoulders
- Loop forever on flat local maxima

Drawbacks





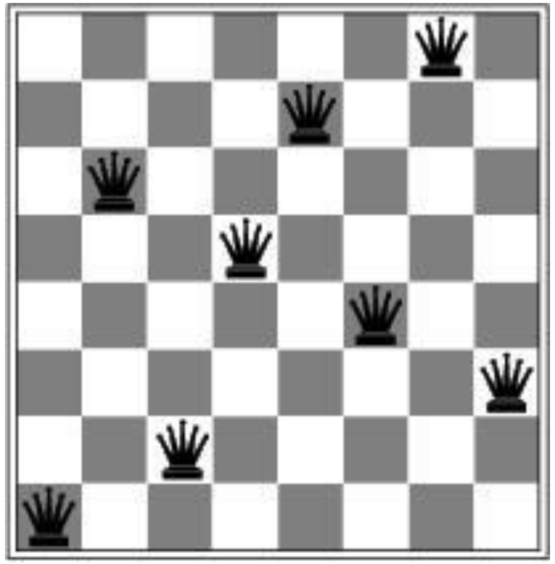
- Ridge = sequence of local maxima difficult for greedy algorithms to navigate

- Plateau = an area of the state space where the evaluation function is flat.

Hill-climbing example

a)

b)



a) Shows a state of h=17 and the h-value for each possible successor.

b) A local minimum in the 8-queens state space (h=1).

Hill-climbing variations

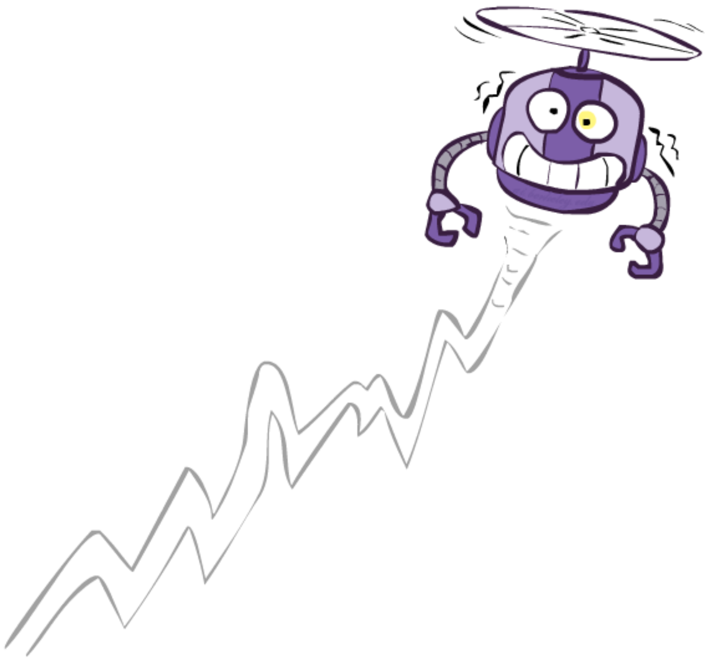
- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
  - A series of Hill Climbing searches from randomly generated initial states
- Simulated Annealing
  - Escape local maxima by allowing some "bad" moves but gradually decrease their frequency

Simulated annealing

- Resembles the annealing process used to cool metals slowly to reach an ordered (low-energy) state
- Basic idea:
  - Allow “bad” moves occasionally, depending on “temperature”
  - High temperature => more bad moves allowed, shake the system out of its local minimum
  - Gradually reduce temperature according to some schedule
  - Sounds pretty flaky, doesn’t it?

Simulated annealing algorithm

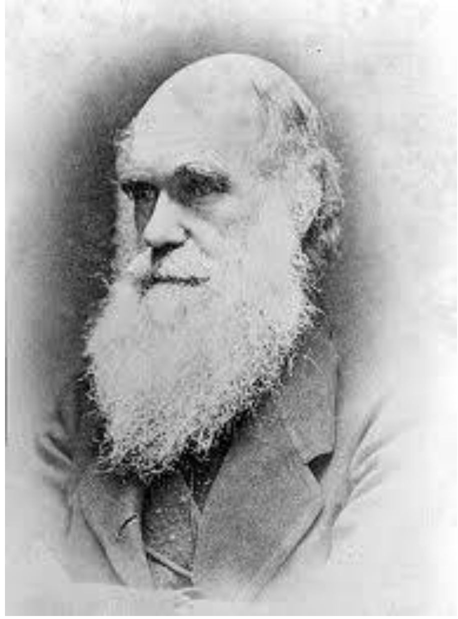
```
function SIMULATED-ANNEALING(problem,schedule) returns a  state
current ← problem.initial-state
for t = 1 to ∞ do
        T ←schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.value – current.value
        if ΔE > 0 then current ← next
                  else current ← next only with probability e^(ΔE/T)
```



Local beam search

- Keep track of *k* “current” states instead of one
  - Initially: *k* random initial states
  - Next: determine all successors of the *k* current states
  - If any of successors is goal $\rightarrow$ finished
  - Else select *k* best from the successors and repeat.
- Major difference with *k* random-restart search
  - Information is shared among *k* search threads.
- Can suffer from lack of diversity.
  - **Stochastic** variant: choose *k* successors at proportionally to the state success.

Local beam search



- Basic idea:
  - *K* copies of a local search algorithm, initialized randomly
  - For each iteration
    - Generate ALL successors from *K* current states
    - Choose best *K* of these to be the new current states

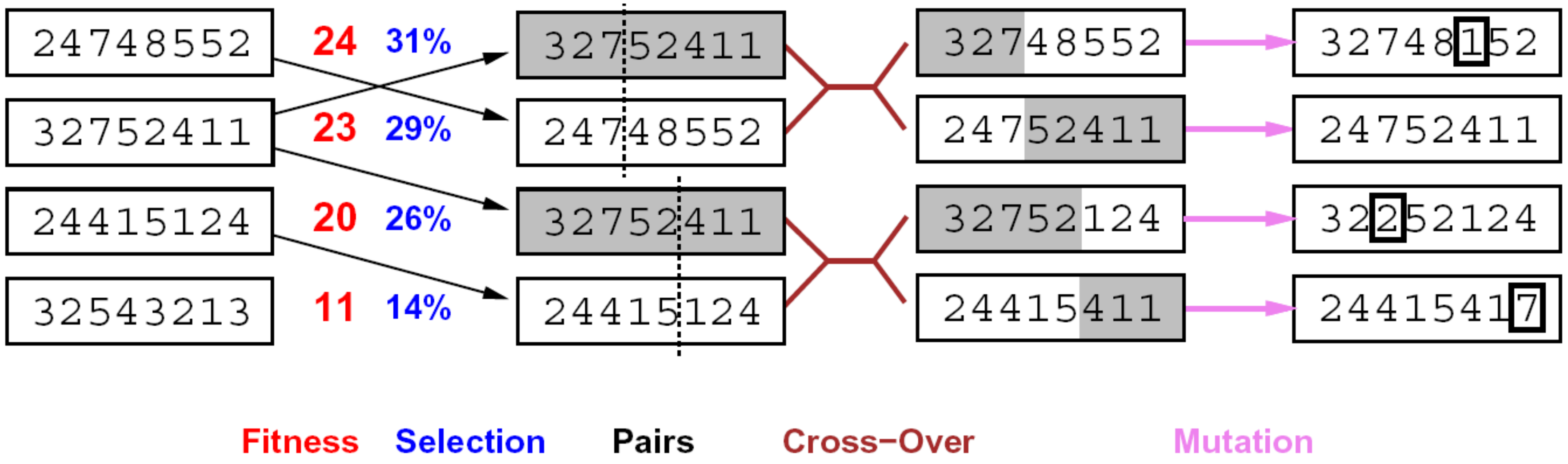
Or, K chosen randomly with a bias towards good ones

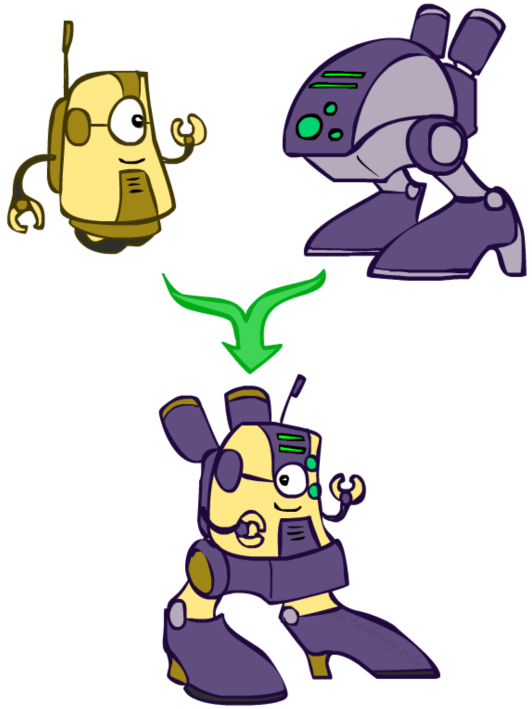
- Why is this different from *K* local searches in parallel?
  - The searches ***communicate***! "Come over here, the grass is greener!"
- What other well-known algorithm does this remind you of?
  - Evolution!

Genetic algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state (an individual) is represented as a string over a finite alphabet (often a string of 0s and 1s), just as DNA that is a string over the alphabet **ACGT.**
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

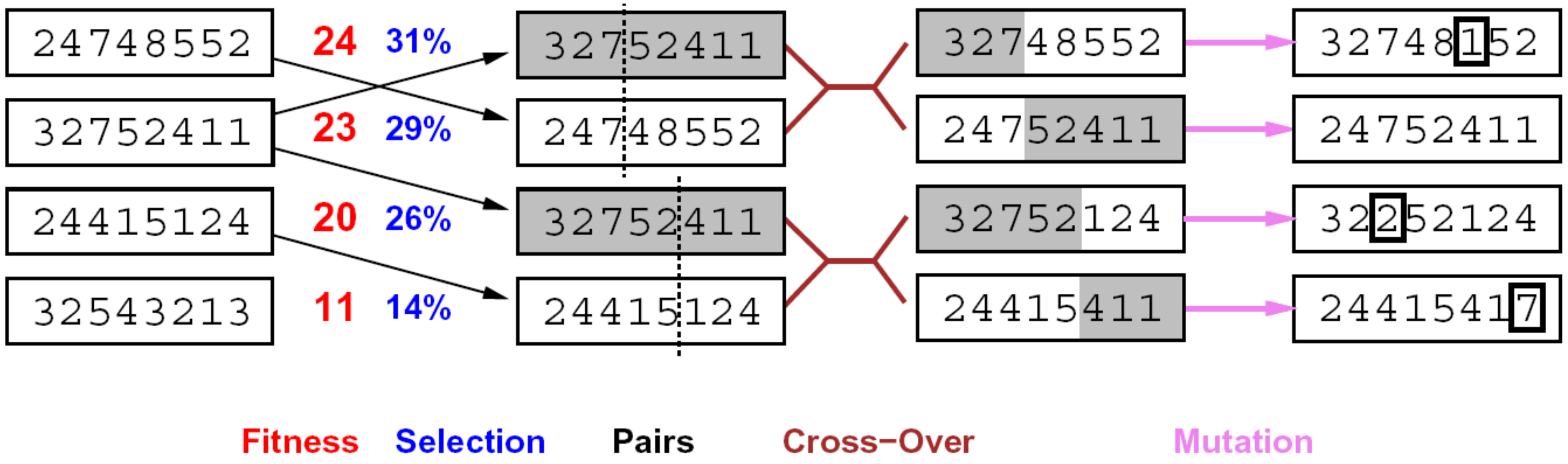
Genetic algorithms





- Genetic algorithms use a natural selection metaphor
  - Resample *K* individuals at each step (selection) weighted by fitness function
  - Combine by pairwise crossover operators, plus mutation to give variety

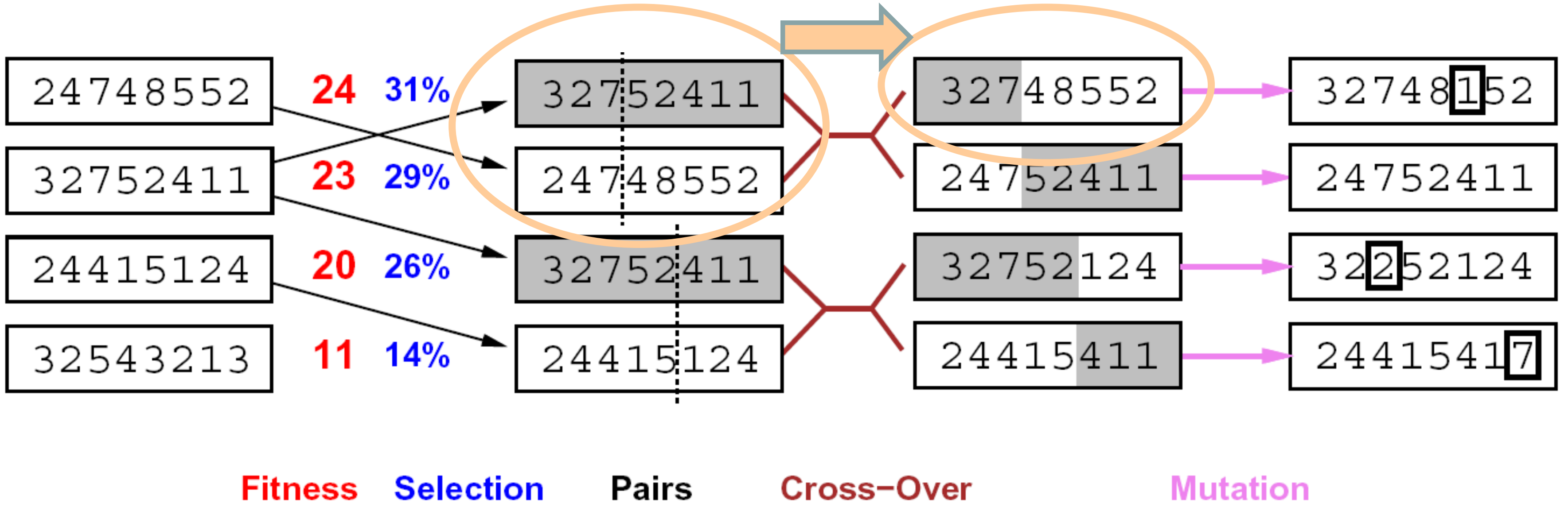
Genetic algorithms

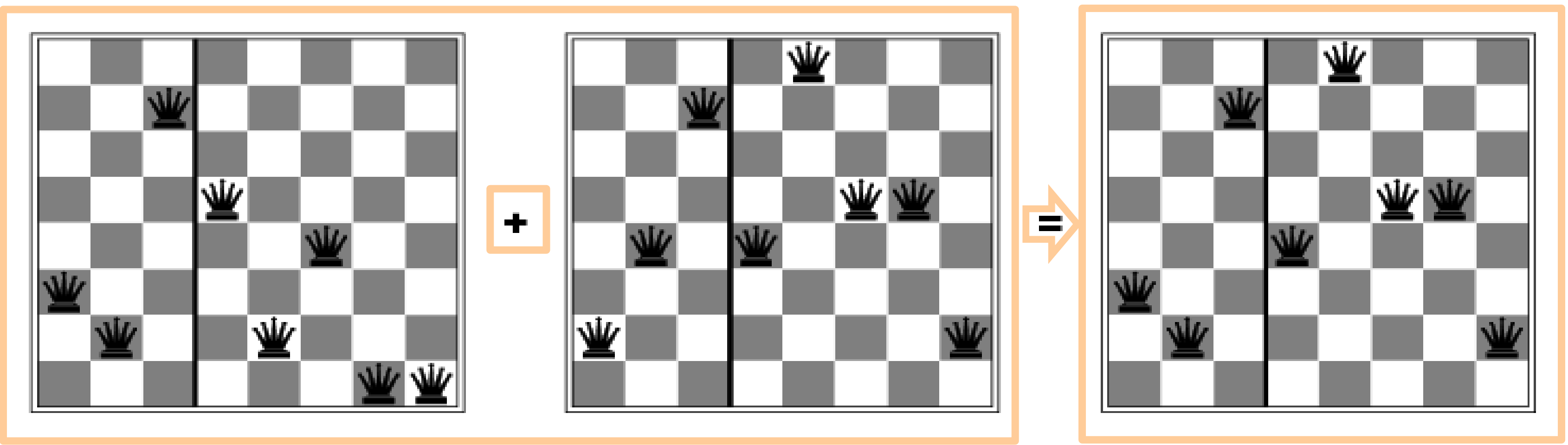


- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc.

[40-417, SUT]

Genetic algorithms





[40-417, SUT]

A Genetic algorithm

function GENETIC_ALGORITHM(*population,* FITNESS-FN) **return** an individual
  input: *population*, a set of individuals
    FITNESS-FN, a function which determines the quality of the individual
  repeat
    new_population $\leftarrow$ empty set
    loop for i **from** 1 **to** SIZE(*population*) **do**
      x $\leftarrow$ RANDOM_SELECTION(*population*, FITNESS_FN)
      y $\leftarrow$ RANDOM_SELECTION(*population*, FITNESS_FN)
      child $\leftarrow$ REPRODUCE(*x,y*)
      if (small random probability) **then** *child* $\leftarrow$ MUTATE(*child*)
      add *child* to *new_population*
    population $\leftarrow$ *new_population*
  until some individual is fit enough or enough time has elapsed
  return the best individual

A Genetic algorithm (Cont.)

function REPRODUCE(*x, y*) **return** an individual

input: *x, y*, parent individuals

$n \leftarrow$ LENGTH(*x*); $c \leftarrow$ random number from 1 to n

return APPEND(SUBSTRING(*x*, 1, *c*), SUBSTRING(y, $c + 1$, *n*))

In this more popular version of GA, from each two parents, only one offspring is produced, not two.

Summary

- Many configuration and optimization problems can be formulated as local search
- General families of algorithms:
  - Hill-climbing, continuous optimization
  - Simulated annealing (and other stochastic methods)
  - Local beam search: multiple interaction searches
  - Genetic algorithms: break and recombine states

Many machine learning algorithms are local searches