Local search algorithms

Chapter 4, Sections 1–2

Outline

- ♢ Hill-climbing
- ♢ Simulated annealing
- ♢ Genetic algorithms

Iterative improvement algorithms

In many optimization problems, **path** is irrelevant;
the goal state itself is the solution; e.g. N-queen problem.

In such cases, we can use iterative improvement algorithms;
keep a single "current" state, try to improve it by making small changes it in each iteration, until no further improvement is possible.

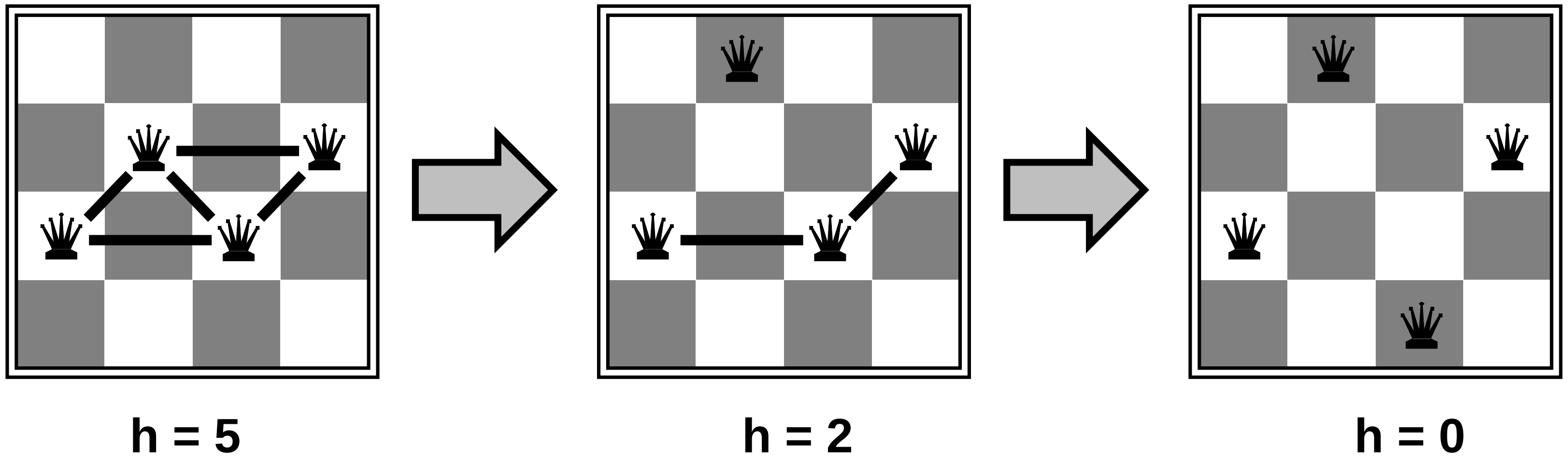
Local search in the state space which is the set of "complete" configurations.
find optimal configuration, e.g., TSP or,
find configuration satisfying constraints, e.g., , N-queen, timetablee.

Constant space, suitable for online as well as offline search

Example: n-queens

Problem: Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts (number of attacking queens).

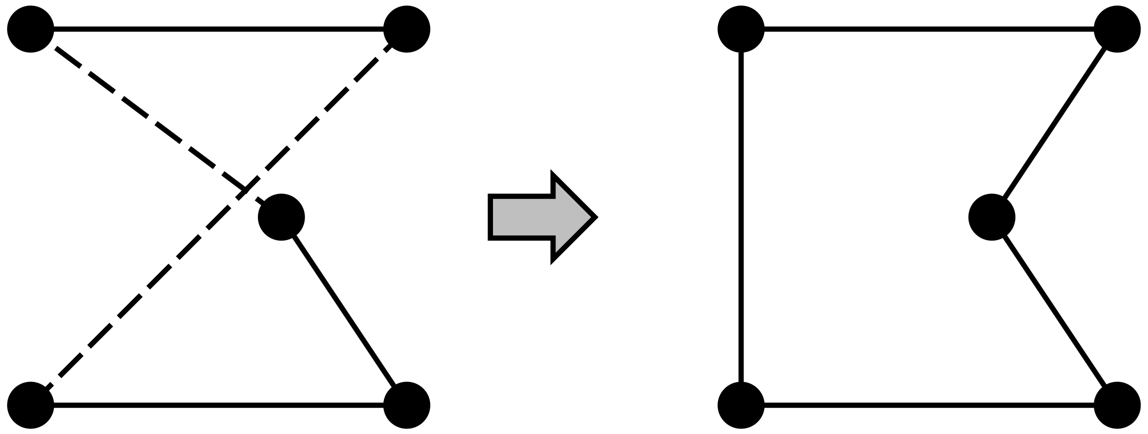


Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n=1$ million

Example: Travelling Salesperson Problem

Short definition: Find the shortest path connecting N given cities.

Start with any complete tour, perform pairwise exchanges:



Variants of this approach get within 1% of optimal very quickly with thousands of cities

Iterative Improvement Algorithms

Iterative improvement algorithms can be applied to minimize a cost (e.g. TSP or reducing number of conflicts) or to maximize a benefit.

At each iteration, whatever measure we use, we will select the configuration that improves that measure.

Hill-climbing (or gradient ascent/descent)

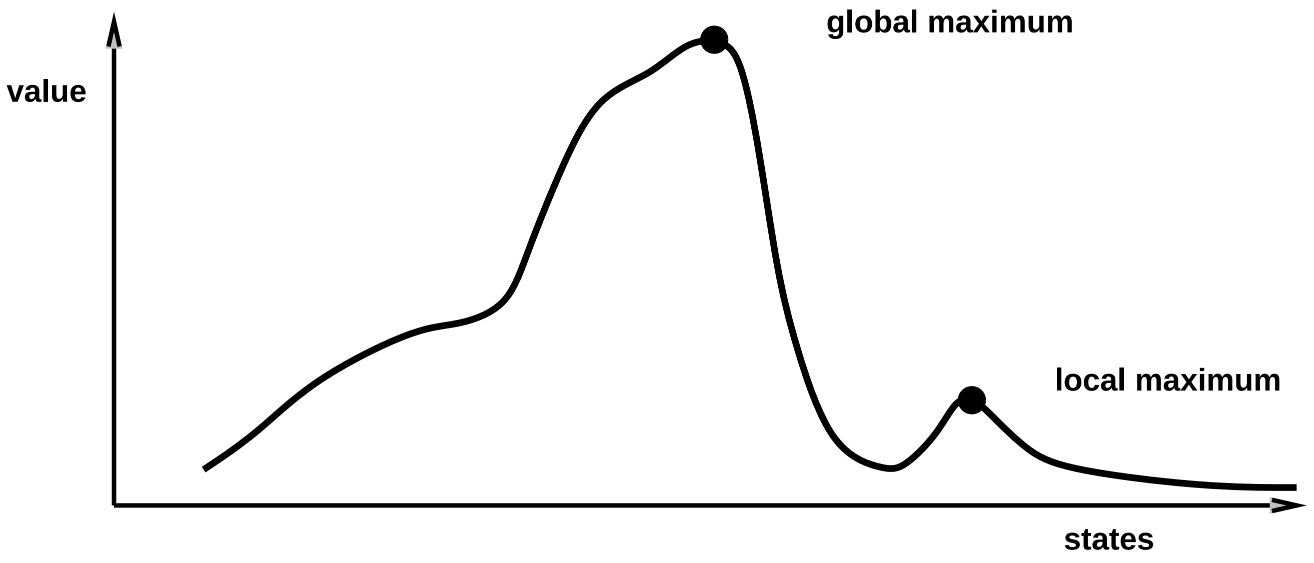
```
function HILL-CLIMBING(problem) returns a solution state
    inputs: problem, a problem
    local variables: current, a node
                     next, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        next ← a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current ← next
    end
```

Note that this is a maximization problem, but Hill Climbing can be used for minimizing a cost, equally well (use negative of the cost as a benefit).

Hill-climbing contd.

Problem: depending on initial state, can get stuck on local maxima



Hill-climbing variations

♢ Stochastic hill-climbing

Choose at random from among the uphill moves. The probability of selection can vary with the steepness of the uphill move.

Convergence: usually slower than hill climbing Solutions: Sometimes better

Hill-climbing variations

♢ First choice hill-climbing

A variation of stochastic hill-climbing. Instead of finding all the possible moves (successor states) and picking one at random, it randomly generates a next state until it finds one which is better.

Useful when there are many successor states (thousands).

Hill-climbing variations

All hill-climbing algorithms so far are incomplete.

♢ Random-start-hill-climbing

Search a goal state, starting from randomly generated initial states.

This variation is complete with probability approaching to 1 (as the number of searches increase).

In fact, it can solve a 3-million queen problem in under a minute (for 8-queen, the probability of success is roughly 0.14, hence 7 random starts is expected to find the solution ($n = 1/p$))

Simulated annealing

Hill-climbing algorithms never makes ”downhill” moves, hence they are guaranteed to be incomplete since they can get stuck in local maxima.

Solution: simulated annealing (again, complete only probabilistically)

Idea: escape local maxima by allowing some “bad” moves
but gradually decrease their size and frequency

E.g. shaking the surface where a ping pong ball is rolling to get it to the global minimum.

♢ Devised by Metropolis et al., 1953, for physical process modelling.
♢ Widely used in VLSI layout, airline scheduling, etc.

Simulated annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a maximization problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling the probability of downward steps

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

Local beam search

Idea: Keeping only one node in memory is an extreme reaction to memory problems.

Local beam search: keep k states instead of just one

Loop:

Start from k randomly generated states
Generate all successors of all k states
If goal is found, stop
Otherwise, keep the top k of the successor states

Local beam search

Not the same as k random-start searches run in parallel!
Searches that find good states recruit other searches to join them

Problem: quite often, all k states end up on same local hill

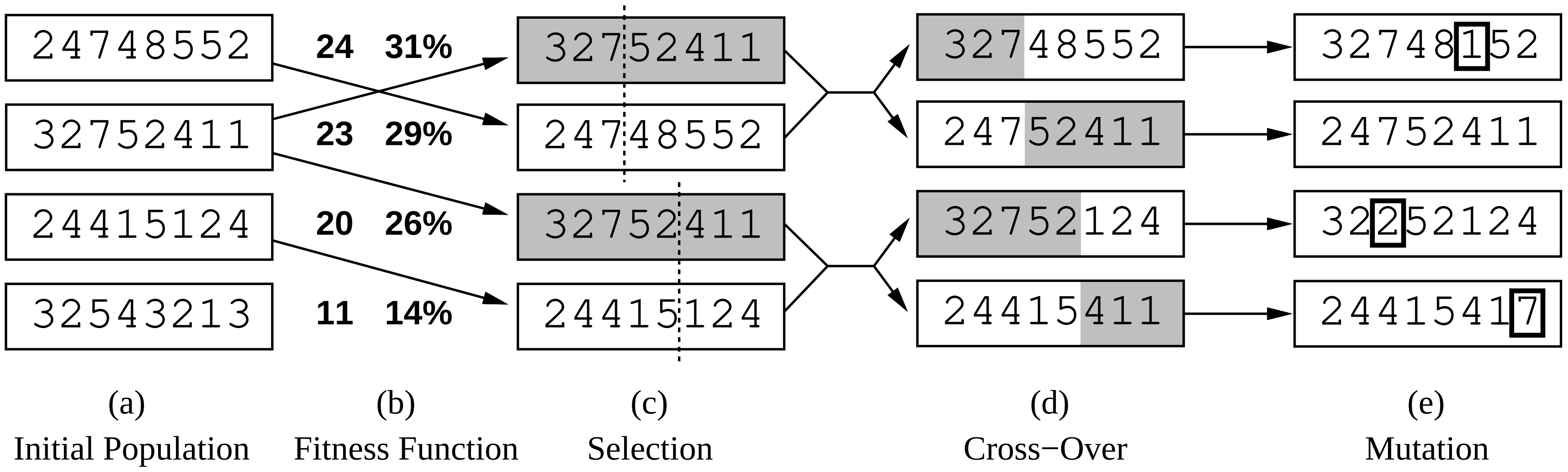
Idea: Stochastic beam search

Choose k successors randomly, biased towards good ones

Observe the close analogy to natural selection!

Genetic algorithms

= stochastic local beam search + generate successors from **pairs** of states



Genetic algorithms

◇ Population

◇ Fitness function

◇ Crossover

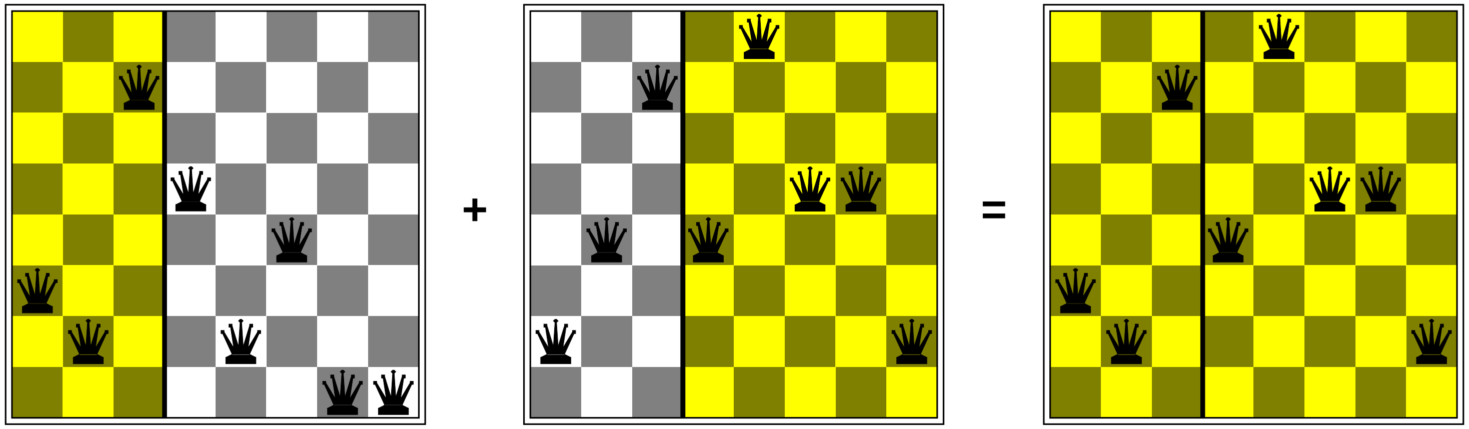
◇ Mutation

◇ Selection

Genetic algorithms contd.

GAs require states encoded as strings (GPs use programs)

Crossover helps **iff substrings are meaningful components**



Genetic algorithms

♢ Start with a population of k randomly selected states

♢ selection w.r.t the fitness function

♢ crossover (mating)

♢ mutation

Idea: Try to take solutions to different **subproblems** from different near-solutions.