Local search

Outline

- ♢ Optimization problems
- ♢ Local search
- ♢ Hill-climbing
- ♢ Simulated annealing
- ♢ Genetic algorithms
- ♢ Local search in continuous spaces

Types of problems

We have considered search techniques for problems where we want a **path** from initial state to goal.

- Route-finding.
- Theorem proving.
- Planning (e.g., vacuum world).

In many problems we just want a **state** which meets some requirements (“cheapest” or “best”)

- Timetabling.
- Product design and configuration.
- Puzzles such as 8-queens.

Optimization problems

Optimization problems are specified by
- a (usually very large) state space
- a goal test and/or
  an objective function $f(n)$ that measures the "goodness" of state n

Problem: find a goal state n_g that maximizes (minimizes) $f(n_g)$ (if present)

Examples:
- the knapsack problem: pack as many objects of weight $w_1, \ldots, w_n$ and utility $u_1, \ldots, u_n$ into a knapsack that can handle weight at most W while maximizing the overall utility
- n-queens: place n queens on a chess board so that no two queens share a row, column, or diagonal
- find the cheapest computer configuration satisfying certain conditions

Only the goal state (= solution) is important; paths are irrelevant
- e.g., in 8-queens, only the final board configuration is interesting

Local search

Keep a bounded (typically constant) number of "current" states; try to improve them iteratively by looking at neighboring states.

They work with **complete-state formulation** – e.g., in 8 queens, each state has 8 queen on the borard and successor function returns all states obtained by moving a single queen to another square in same column.

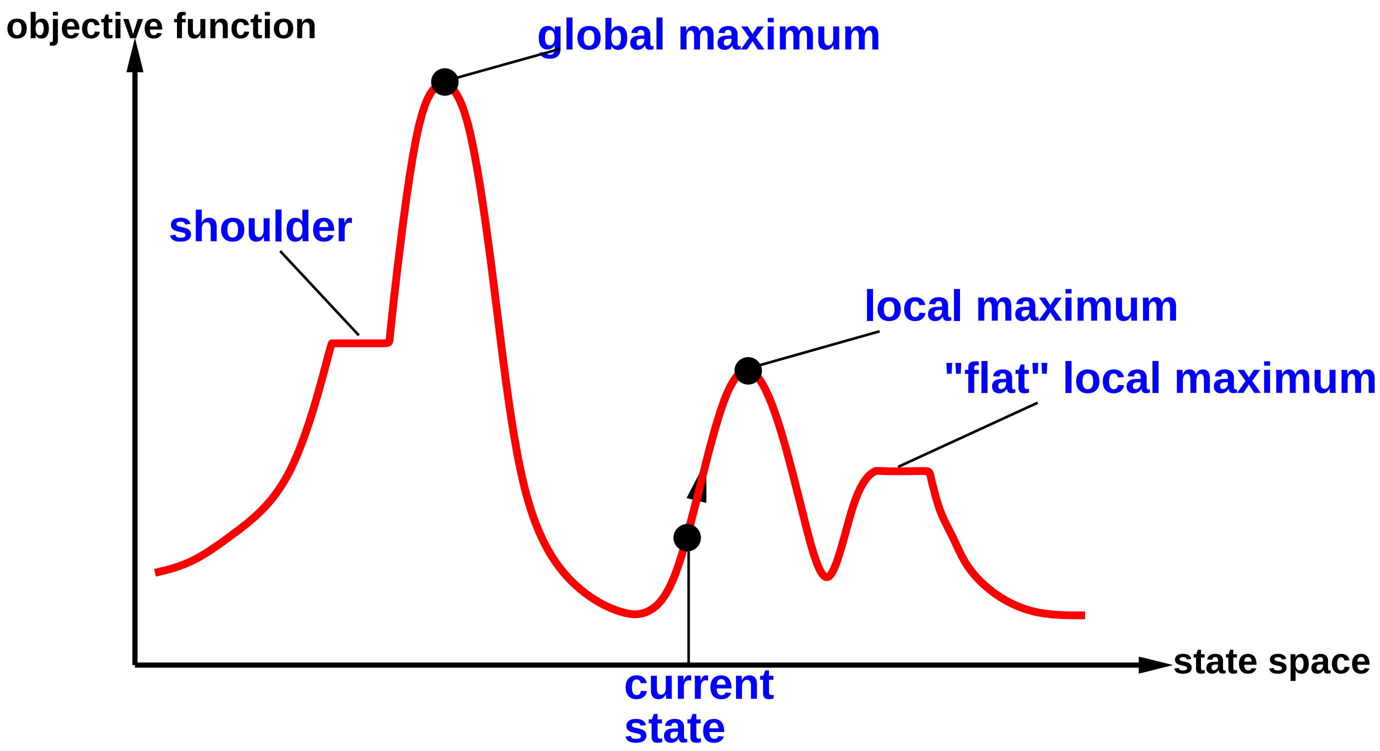
Objective function guides the search (pick a neighbor with best value of the function)

– if a problem does not have an objective function, then invent a heuristic function that estimates closeness to a goal
– e.g., for n-queens, use the number of queens not under attack

Local search algorithms are not systematic, but

– use little memory (constant number of states, no need to record paths)
– can offer good solutions to problems with huge search space.

State space landscape



A complete algorithm always finds a goal

An optimal algorithm always finds a **global maximum** (or **minimum**)

Hill-climbing (or gradient ascent/descent)

"Like climbing Everest in thick fog with amnesia"

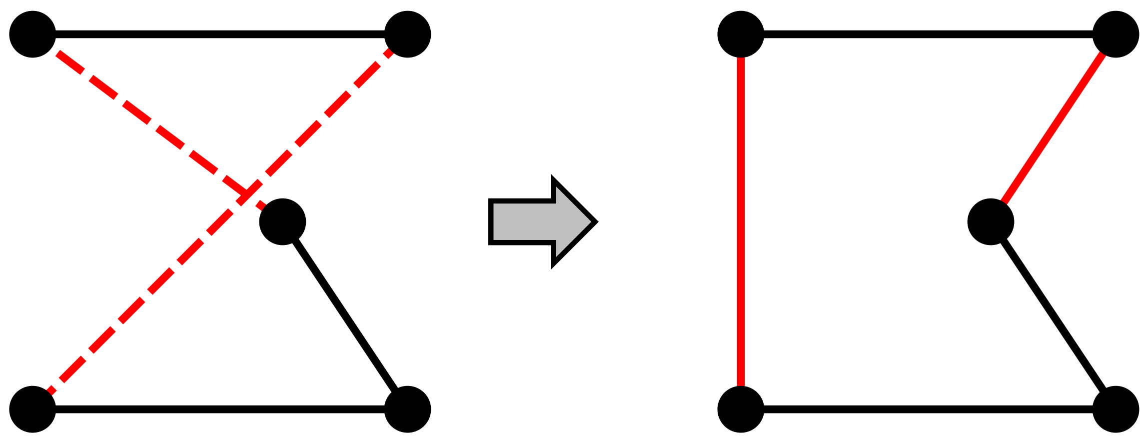
```
function Hill-Climbing(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← problem.Initial-State
   loop do
       neighbor ← a highest-valued successor of current
       if neighbor.Value < current.Value then return current
       current ← neighbor
   end
```

Example: Traveling salesman problem

Start with an arbitrary complete tour

Successor function: a successor n' is obtained from n by reconnecting end-points of two edges

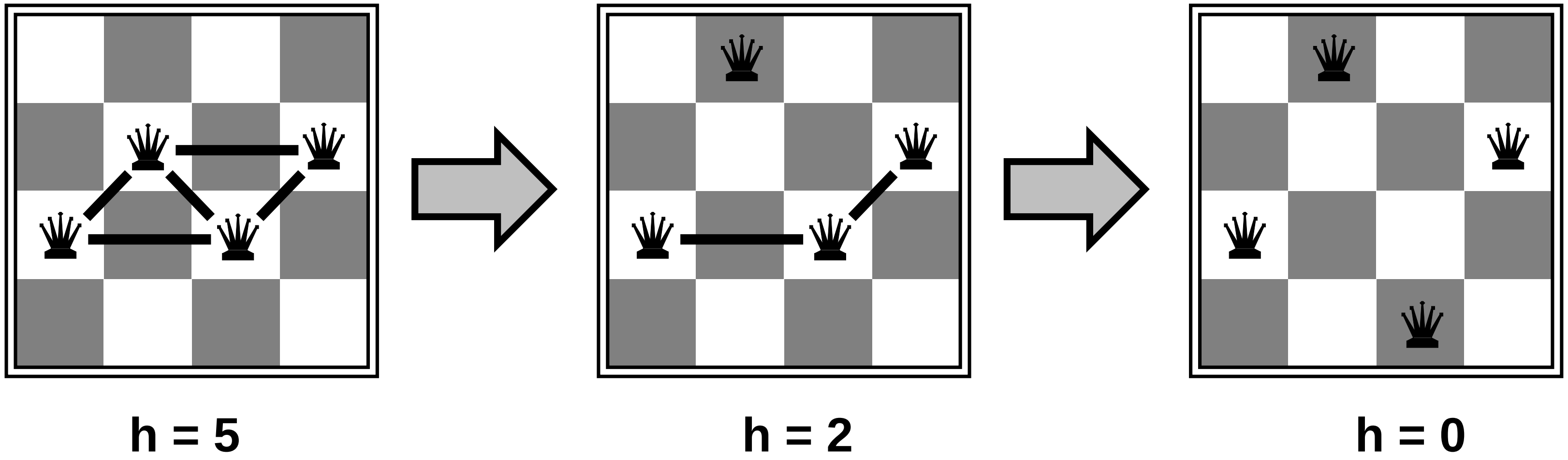


Variants of this approach get within 1% of the optimal solution very quickly even with thousands of cities

Example: n-queens

Start with an arbitrary configuration

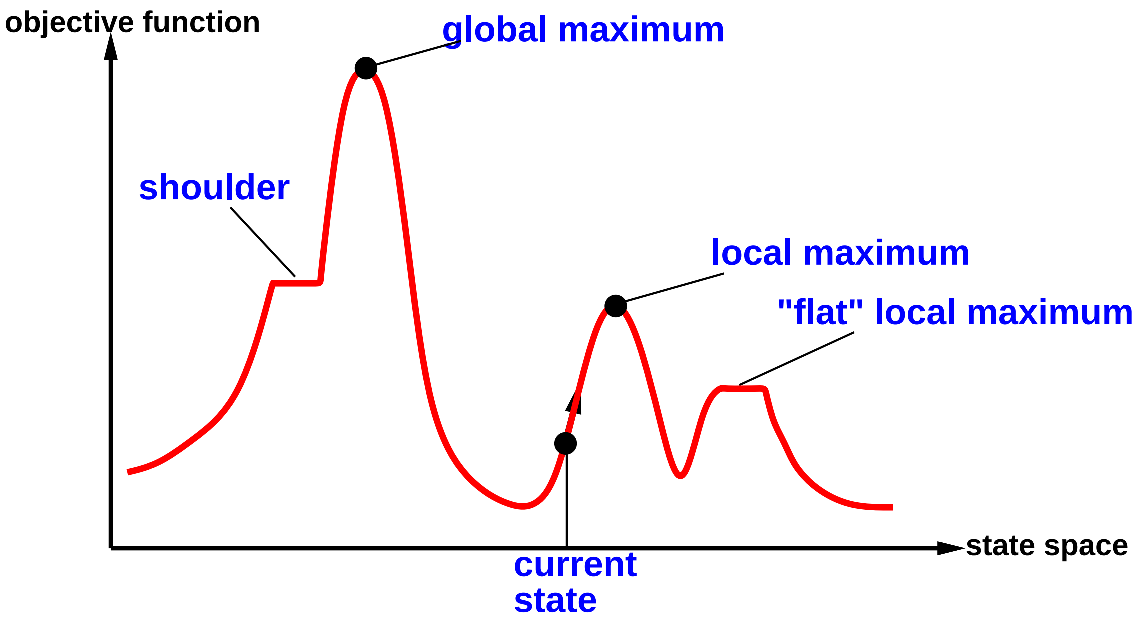
Successor function: a successor n' is obtained from n by moving one queen



Almost always solves the problem almost instantaneously for very large numbers of queens

– e.g., 10^6

Problems of hill-climbing



Easily gets stuck due to:

– local maxima: nowhere else to go!

– ridges (sequences of increasing local maxima): difficult to navigate

– plateaux (includes shoulders): difficult to get off

$\Rightarrow$ Incomplete and not optimal

Improving hill-climbing

Allow sideways moves: keep going on a plateaux
– we hope that plateaux is a shoulder
Can get into an endless loop
$\Rightarrow$ limit the number of consecutive sideways moves

Stochastic hill-climbing: choose at random from uphill moves
– the probability of choosing a move depends on the gradient of ascent

First choice hill-climbing: randomly generate successors until one is found that is better than the current state
– useful when a state has many (e.g., thousands) successors

Improving hill-climbing: Random restart

"If at first you don't succeed, try again!"

Conduct a series of searches starting from randomly generated states;
Stop when a goal is found

Complete with probability approaching 1
– eventually we generate the goal state

If each search has probability of success p, we need $1/p$ searches
– for 8-queens, $p \approx 0.14$, so we expect 7 searches on average
$\Rightarrow$ very effective: it works in under a minute even for 10^6 queens!

Simulated annealing

◇ Hill climbing: incomplete as it can get stuck in local maxima.

◇ Random walk (choose successor at random): complete but inefficient.

◇ Simulated annealing: combines hill climbing with random walk.

Inspired by metallurgy, where metals and glass are heated first and then allowed to cool. There is higher randomness at large temperatures.

◇ Instead of picking a best move, pick a random move
– if the random move improves the situation, accept it
– otherwise, accept the move with some probability.
– the probability decreases exponentially with badness of move and with the "temperature"

Simulated annealing

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

  current ← *problem*.INITIAL-STATE
  for $t \leftarrow 1$ **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 $e^{\Delta E/T}$

Local beam search

Keep k states instead of 1; choose top k of all their successors

Begin with k randomly generated states
- At each step, all successors of all k states are generated
- If any one is the goal, halt
- Otherwise, select best k successors amongst whole list and repeat.

Not the same as running k searches 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
$\Rightarrow$ Choose k successors randomly, biased towards good ones

Observe a close analogy to natural selection!

Summary

Optimisation problems:
- ignore paths
- goal states are solutions

Local search starts from a state and improves it iteratively

Hill-climbing: basic local search algorithm
- easily gets stuck in the search space
- various optimizations

Probabilistic local search techniques offer further improvement