Search in Complex Environments

Presented by Yasin Ceran

September 30, 2024

Iterative Improvement Algorithms

Module4 focused on problems in:

- Fully observable
- Deterministic
- Static
- Known environments

where the solution is a **sequence of actions**.

In Module 5 we relax these constraints:

- **Finding good states** without considering the path (discrete or continuous paths)
- **Relaxing determinism**:
  - Agent needs a **conditional plan**, e.g., stop if red, go if green.
- **Partial observability**:
  - Agent tracks **possible states** it might be in.
- **Online search in unknown spaces**:
  - Agent must **learn as it goes**.

Local Search Algorithms

Key Characteristics of Local Search Algorithms:

- Search from a **start state** to neighboring states.
- **Do not track** the path or previously reached states.
- Not systematic—might miss parts of the search space where a solution exists.

Advantages of Local Search:

- Use very **little memory**.
- Can often find **reasonable solutions** in large or infinite state spaces.

Local search algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function.

Hill-climbing (or gradient ascent/descent)

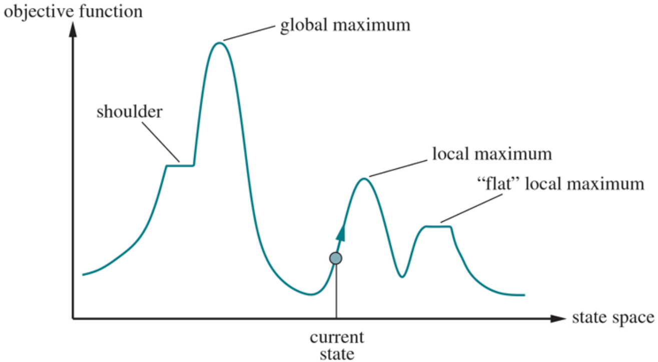
"Like climbing Everest in thick fog with amnesia"

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

    current ← Make-Node(Initial-State[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if Value[neighbor] ≤ Value[current] then return State[current]
        current ← neighbor
    end
```

Hill-climbing contd.

Useful to consider state space landscape



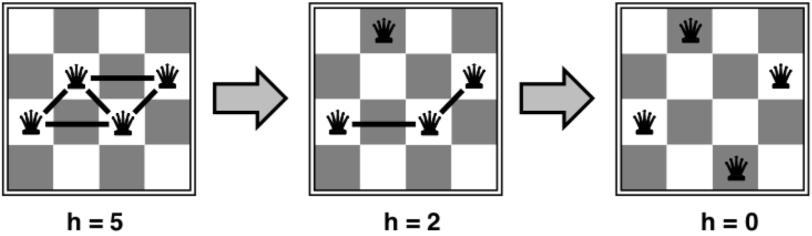
Random-restart hill climbing overcomes local maxima—trivially complete

Random sideways moves 😊 escape from shoulders ☹ loop on flat maxima

Example: n-queens

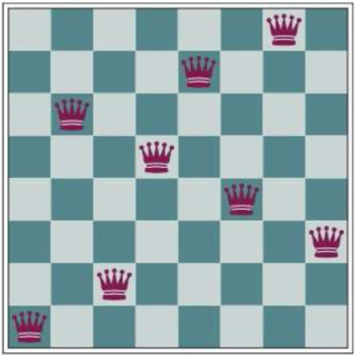
Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



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

Example: 8-queens



(a)

(b)

The board shows the value of h for each possible successor obtained by moving a queen within its column.

Simulated Annealing

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

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 ← Make-Node(Initial-State[*problem*])
for $t \leftarrow 1$ **to** ∞ **do**
$T \leftarrow$ *schedule*[t]
if $T = 0$ **then return** *current*
next ← a randomly selected successor of *current*
$\Delta E \leftarrow$ Value[*next*] – Value[*current*]
if $\Delta E > 0$ **then** *current* ← *next*
else *current* ← *next* only with probability $e^{-\Delta E/T}$

Properties of Simulated Annealing

At fixed "temperature" T, state occupation probability reaches Boltzman distribution

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

T decreased slowly enough $\Longrightarrow$ always reach best state x^* because $e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*)-E(x)}{kT}} \gg 1$ for small T

Is this necessarily an interesting guarantee??

Devised by Metropolis et al., 1953, for physical process modelling

Widely used in VLSI layout, airline scheduling, etc.

Local Beam Search

Idea: keep k states instead of 1; choose top k of all their successors

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

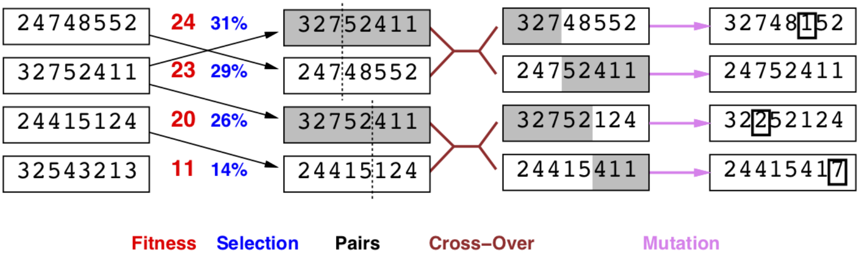
Problem: quite often, all k states end up on same local hill

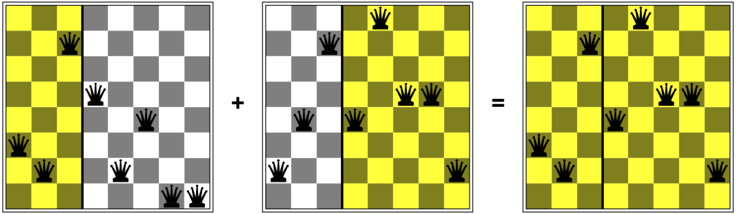
Idea: choose k successors randomly (Stochastic Beam Search), biased towards good ones

Observe the close analogy to natural selection!

Genetic Algorithms

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





Genetic Algorithms-II

```
function Genetic Algorithm( population, Fitness-Fn) returns an individ-
ual
   inputs: population, a set of individuals
           Fitness-Fn, a function that measures the fitness of an indi-
vidual

   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
   until some individual is fit enough, or enough time has elapsed
   return the best individual in population, according to Fitness-Fn
```

Genetic Algorithms-III

function REPRODUCE(*x,y*) **returns** returns an individual
inputs: *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))

Continuous State Spaces

Suppose we want to site three airports in Romania:
- 6-D state space defined by (x_1, y_2), (x_2, y_2), (x_3, y_3)
- objective function $f(x_1, y_2, x_2, y_2, x_3, y_3) =$
sum of squared distances from each city to nearest airport

Discretization methods turn continuous space into discrete space, e.g., empirical gradient considers $\pm\delta$ change in each coordinate

Gradient methods compute

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

to increase/reduce f, e.g., by $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city).

Summary

- Hill-climbing is simply a loop that continually moves in the direction of increasing value
- With simulated annealing, we can escape local maxima by allowing some "bad" moves
- Local beam search allows us to keep track of multiple states instead of 1, thus choose k successors that are biased towards the good ones
- Continuous state spaces can offer us search algorithms that can be used for real life continuous environments.