# Artificial Intelligence

## Lecture 4, Chapter 4
## Local Search Algorithm and Optimizations

Supta Richard Philip

Department of CS
AIUB

AIUB, January 2025

## Table of Contents

## Table of Contents

## Beyond Classical Search

- A single category of problems: observable, deterministic, known environments where the solution is a sequence of actions.
- when these assumptions are relaxed.
- algorithms that perform purely local search in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state.
- These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it.
- The family of local search algorithms includes methods inspired by statistical physics (simulated annealing) and evolutionary biology (genetic algorithms).

## Table of Contents

## Local Search Algorithms and Optimization Problems

- The search algorithms that we have seen so far are designed to explore search spaces systematically.
- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path.
- When a goal is found, the path to that goal also constitutes a solution to the problem.
- In many problems, however, the path to the goal is irrelevant.

## Local Search Algorithms and Optimization Problems

- For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.
- The same general property holds for many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.
- If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all.
- Example: **Local search**

## Local search

- Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node.
- Typically, the paths followed by the search are not retained.
- Although local search algorithms are not systematic, they have two key advantages:
  - they use very little memory—usually a constant amount; and
  - they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

## Local search solves optimization problems

- **Local search algorithms** are useful for solving **pure optimization problems**, in which the aim is to find the best state according to an **objective function**.
- Many optimization problems do not fit the "standard" search model but provides an objective function as attempting to optimize, but **there is no "goal test" and no "path cost"** for this problem.
- To understand local search, considering the **state-space landscape**. A landscape has both **"location" (defined by the state)** and **"elevation" (defined by the value of the heuristic cost function or objective function)**.

## Optimization problems

- If elevation corresponds to **cost function**, then the aim is to find the **lowest valley—a global minimum**;
- if elevation corresponds to an **objective function**, then the aim is to find the **highest peak—a global maximum**. (You can convert from one to the other just by inserting a minus sign.)
- Local search algorithms explore this landscape.
- A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.

# optimization problems: Hill-climbing search

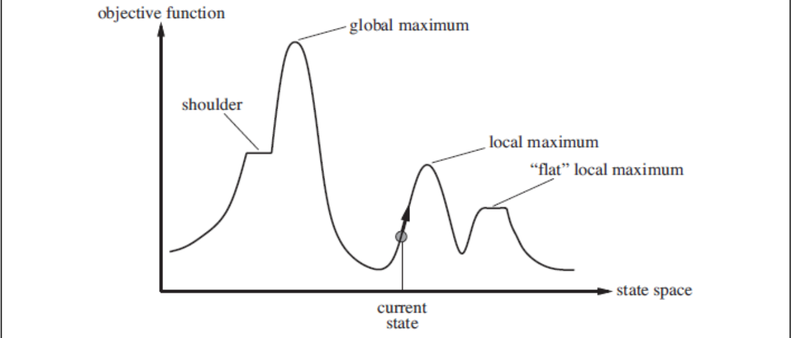
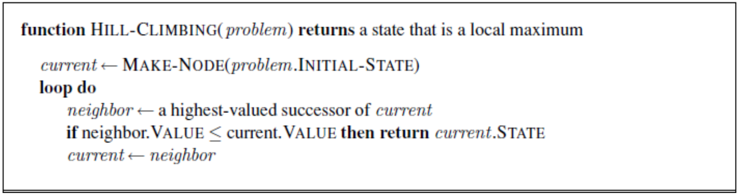


**Figure 4.1** A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

## Hill-climbing Algorithm

- The hill-climbing search algorithm (steepest-ascent version) is shown in Figure. It is simply a loop that continually moves in the direction of increasing value—that is, uphill.
- It terminates when it reaches a “peak” where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Unfortunately hill climbing often gets stuck for the following reasons: Local maxima, Ridges, and Plateaux.



```
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 $\leq$ current.VALUE then return current.STATE
     current ← neighbor
```

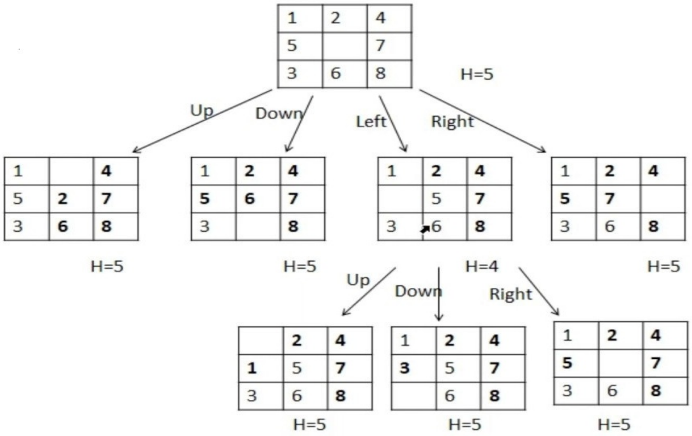
## Hill-climbing Cont.

- **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
- This usually converges more slowly than the steepest ascent, but in some state landscapes, it finds better solutions.
- **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- This is a good strategy when a state has many (e.g., thousands) of successors.
- The hill-climbing algorithms described so far are incomplete; they often fail to find a goal when one exists because they can get stuck on local maxima.

## Hill-climbing Cont.

- **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, until a goal is found.
- For 8-queens, then, random-restart hill climbing is very effective indeed.
- if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.
- NP-hard problems typically have an exponential number of local maxima to get stuck on.

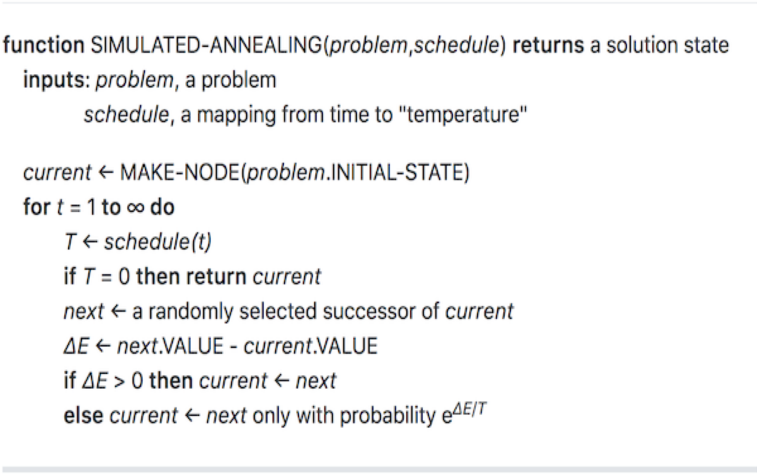
# Hill-climbing in Local Minima



## Simulated annealing

- **A hill-climbing algorithm** that never makes “downhill” moves toward states with lower value.
- **A purely random walk** that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.
- **Simulated annealing** is such an algorithm 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 ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      $\Delta E$ ← next.VALUE - current.VALUE
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $e^{\Delta E/T}$
```

## Simulated Annealing

```
1: Generate initial solution $x^{c}$, initialise $R_{max}$ and $T$
2: for $r = 1$ to $R_{max}$ do
3:   while stopping criteria not met do
4:     Compute $x^{n} \in \mathcal{N}(x^{c})$    (neighbour to current solution)
5:     Compute $\Delta = f(x^{n}) - f(x^{c})$ and generate $u$ (uniform
       random variable)
6:     if $(\Delta < 0)$ or $(e^{-\Delta/T} > u)$ then $x^{c} = x^{n}$
7:   end while
8:   Reduce $T$
9: end for
```

## Local beam search

- **The local beam search algorithm** keeps track of k states rather than just one.
- Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.
- It begins with k randomly generated states. At each step, all the successors of all k states are generated.
- If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.
- A variant called **stochastic beam search**, instead of choosing the **best k** from the pool of candidate successors, stochastic beam search chooses **k successors at random**.

## Genetic algorithms

- A **genetic algorithm (or GA)**is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.

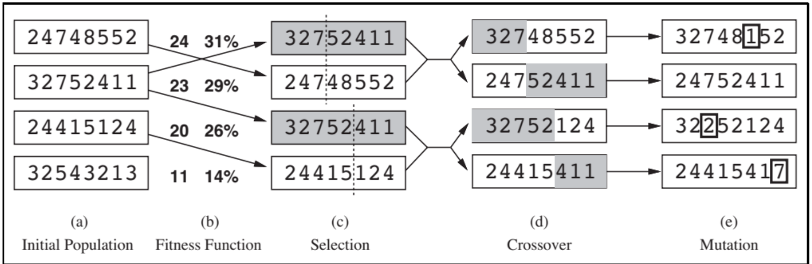


**Figure 4.6** The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

## Genetic algorithms cont.

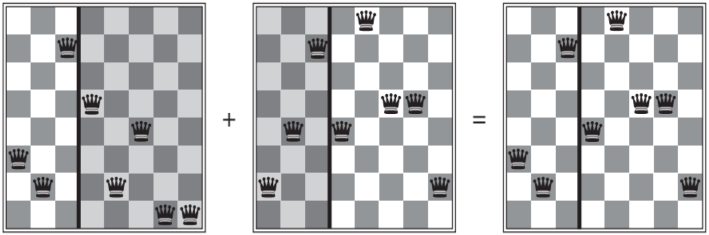
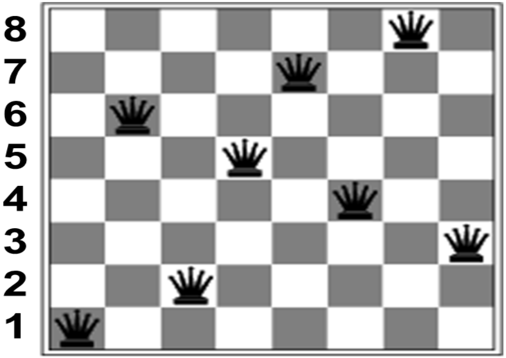


**Figure 4.7** The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

## Genetic algorithms

- Check every column and find the position of the queen.



String representation
16257483

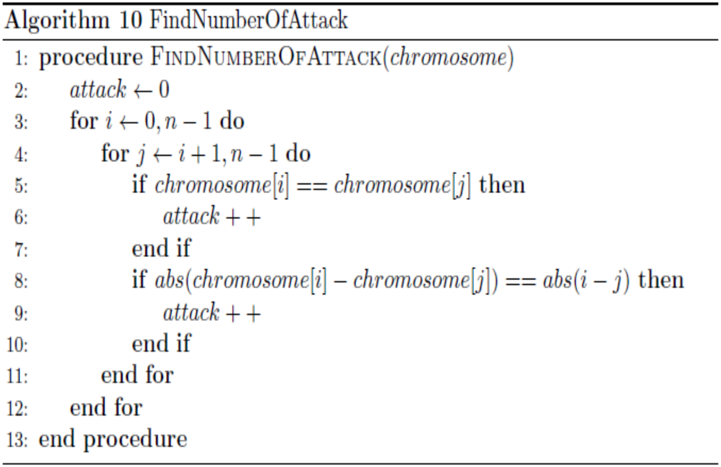
## Genetic algorithms

- Fitness function: number of non-attacking pairs of queens for 8-queen (min = 0, max = 8 * 7/2 = 28) .
- for n-queen max non attacking pairs = n*(n-1)/2

$$\binom{n}{2} = \frac{n!}{2!(n-2)!}$$

- First state value, non attacking (total non attacking(28) - attacking(4)) = 24
- The values (non-attacking) of the four states are 24, 23, 20, and 11.
- Probability of non-attacking for the first state: 24/(24+23+20+11) = 31%
- In (c), two pairs are selected at random for reproduction.

## Find the number of attacks.



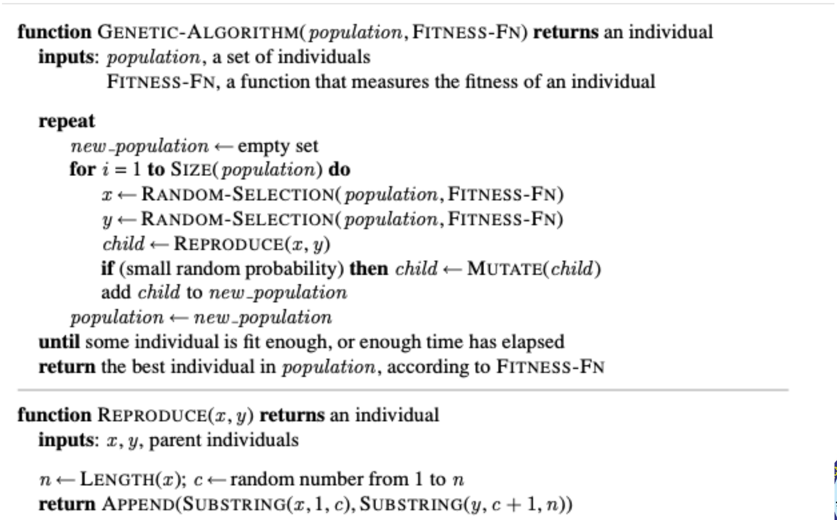
**Algorithm 10** FindNumberOfAttack

```
procedure FindNumberOfAttack($chromosome$)
    $attack \leftarrow 0$
    for $i \leftarrow 0, n-1$ do
        for $j \leftarrow i+1, n-1$ do
            if $chromosome[i] == chromosome[j]$ then
                $attack++$
            end if
            if $abs(chromosome[i] - chromosome[j]) == abs(i-j)$ then
                $attack++$
            end if
        end for
    end for
end procedure
```

## Genetic algorithms

- Find the number of attacks on the chromosomes (String) chromosome 1: 76012345 chromosome 2: 66743210

## Genetic algorithms cont.



```
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($y$, $c + 1$, $n$))
```

## Table of Contents

## Local Search in Continuous Spaces

- we explained the distinction between discrete and continuous environments, pointing out that **most real-world environments are continuous**.
- Yet none of the algorithms we have described (except for **first-choice hill climbing and simulated annealing**) can handle continuous state and action spaces, because they have infinite branching factors.
- Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map to its nearest airport is minimized.

## Local Search in Continuous Spaces

- The state space is then defined by the coordinates of the airports: (x1, y1), (x2, y2), and (x3, y3). This is a six-dimensional space; we also say that states are defined by six variables.
- The objective function

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

- This expression is correct locally, but not globally because the sets $C_i$ are (discontinuous) functions of the state.
- To avoid continuous problems is simply to discretize the neighborhood of each state.

## Local Search in Continuous Spaces: steepest-ascent hill climbing(Gradient descent)

- Many methods attempt to use the gradient of the landscape to find a maximum. The gradient of the objective function is a vector Δ f that gives the magnitude and direction of the steepest slope.

$$\Delta f = (\frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta y_1}, \frac{\delta f}{\delta x_2}, \frac{\delta f}{\delta y_2}, \frac{\delta f}{\delta x_3}, \frac{\delta f}{\delta x_3})$$

- we can find a maximum by solving the equation $\Delta f = 0$ .
- This means we can compute the gradient locally (but not globally); for example,

$$\frac{\delta f}{\delta x_1} = 2 \sum_{c \in C_i} (x_i - x_c)$$

## Local Search in Continuous Spaces: steepest-ascent hill climbing(Gradient descent)

- Given a locally correct expression for the gradient, we can perform the steepest-ascent hill climbing(Gradient descent) by updating the current state according to the formula:

  $$x \leftarrow x + \alpha \Delta f(x)$$

  where $\alpha$ is a small constant often called the step size.
- The basic problem is that, if $\alpha$ is too small, too many steps are needed; if $\alpha$ is too large, the search could overshoot the maximum.

## Local Search in Continuous Spaces: Newton–Raphson

- For many problems, the most effective algorithm is the Newton–Raphson method. This is a general technique for finding roots of functions—that is, solving equations of the form g(x) = 0. It works by computing a new estimate for the root x according to Newton's formula $x \leftarrow x - \frac{g(x)}{g'(x)}$
- To find a maximum or minimum of f, we need to find x such that the gradient is zero (i.e.,$\Delta$ f(x) = 0). Thus, g(x) in Newton's formula becomes $\Delta$f(x), and the update equation can be written in matrix-vector form as

$$x \leftarrow x - \mathbf{H}_f^{-1}(x)\Delta f(x)$$

where $\mathbf{H}_f(x)$ is the **Hessian** matrix of second derivatives, whose elements $H_{ij}$ are given by $\frac{\delta^2 f}{\delta x_i \delta x_j}$

## Table of Contents

## Find fitness value

- Calculate the fitness value (i.e., the number of non-attacking pairs) for each of the following populations in the 6-Queens problem. Assume column indices start from 0. The chromosomes are: 211345 403234 511023

## GA numerical Example

Maximize $f(x) = x^2/2 - 3x$ , when x in [0 to 31]
Binary Encoding – 5 digits

- Initialization of Population
- Selection
- Crossover/Recombination
- Mutation
- Survival/Accept
- Update Population

# Table of Contents

## References

- Stuart Russell and Peter Norvig. 2009. Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.
- https://www.cs.cmu.edu/ 15281-s25
- https://inst.eecs.berkeley.edu/ cs188/su25/