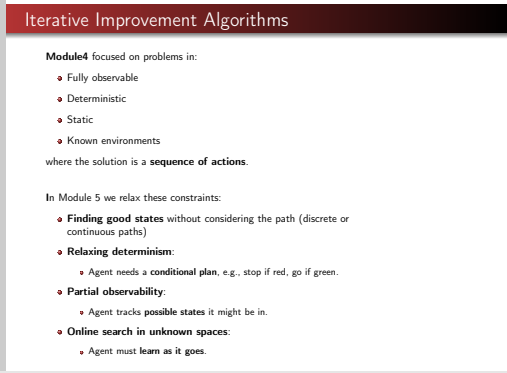
Chapter 3 addressed problems in fully observable, deterministic, static, known environments where the solution is a sequence of actions. In this chapter, we relax those constraints. We begin with the problem of finding a good state without worrying about the path to get there, covering both discrete (Section 4.1) and continuous (Section 4.2) states. Then we relax the assumptions of determinism (Section 4.3) and observability (Section 4.4). In a nondeterministic world, the agent will need a conditional plan and carry out different actions depending on what it observes—for example, stopping if the light is red and going if it is green. With partial observability, the agent will also need to keep track of the possible states it might be in. Finally, Section 4.5 guides the agent through an unknown space that it must learn as it goes, using online search.


Iterative Improvement Algorithms

Module4 focused on problems in:

- Fully observable
- Deterministic
- Static
- Known environments

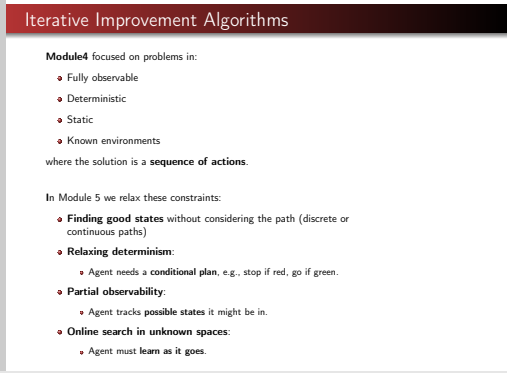
where the solution is a **sequence of actions**.

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**.

In order to systematically search the spaces, we need to keep one or more paths in memory and record which alternatives have been explored at each point along a path. When a goal is found, the path leading to the goal is considered as the solution to the problem. In many problems, however, the path to goal is irrelevant. For example, in the n-queens problem that we will see in the next slide, what matters is the final configuration of the queens, not the order in which the queens are placed. There are numerous real life problems such as telecom network optimization, vehicle routing, portfolio management, that the order in which the solution is searched does not matter.

Local search algorithms operate using a single current node (rather than multiple paths)and generally move only to neighbors of that node.



Although local search algorithms are not systematic, they two advantages: (1) they sue very little memory and (2) they can find reasonably well solutions in large search spaces.

In addition to finding goals, local search algorithms are useful to solve optimization problems where the aim is to find the best state according to some objective function.

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 maximum
    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
```

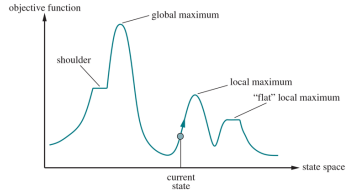
The hill climbing search algorithm is the most basic local search technique. It is simply a loop that continually moves in the direction of increasing value – that is , uphill. At each step, the current node is replaced by the best neighbor, as seen in the algorithm on the slide, the neighbor with the highest value replaces the current node, and sometimes, if we use a heruistic cost estimate h, we would find the neighbor with the lowest cost h.

This algorithm does not maintain a search tree, so the data structure only records the state and the value of the objective function for the current node.Hill climbing does not look ahead beyond the neighbor.

Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead where to go next. To understand local search, it might be useful to consider the state-space landscape. A landscape has both "location"(defined by the state) and the "elevation" (defined by the value of the heuristic cost function or objective function)

Hill-climbing contd.

Useful to consider state space landscape

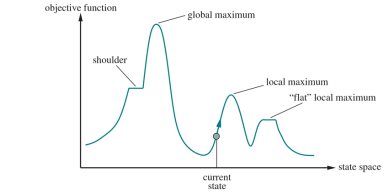


Random-restart hill climbing overcomes local maxima—trivially complete

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

Hill-climbing contd.

Useful to consider state space landscape

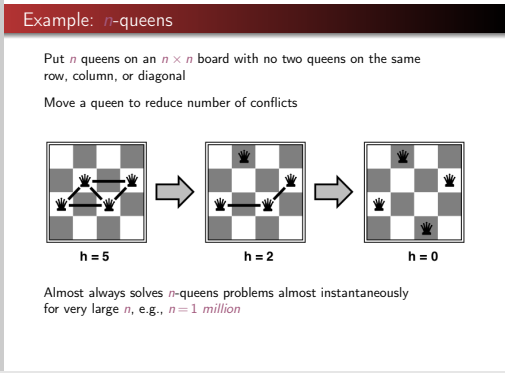


Random-restart hill climbing overcomes local maxima—trivially complete

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

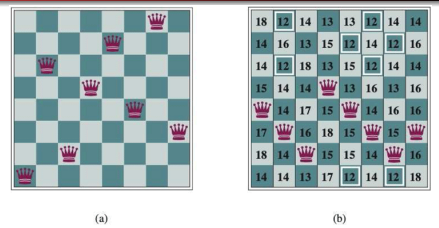
- If elevation corresponds to cost, then the aim is to find the lowest valley, that is a global minimum.
- If elevation corresponds to an objective function, the aim is then to find a global maximum.
- Local Maxima is a peak that is higher than each of its neighboring states but lower than the global maximum.
- A complete local search algorithm always finds a goal if one exists
- An optimal algorithm always finds a global minimum/maximum.
- A plateaux is a flat area of the state-space landscape; it can be a flat local maximum or a shoulder from which progress is possible.

The algorithm sometimes reaches a plateau where the best successor has the same value with the current state. In that case, it might be a good idea to take sideways moves,that is to jump somewhere else and restart the search. There might be a limit to the number of sideway moves allowed.



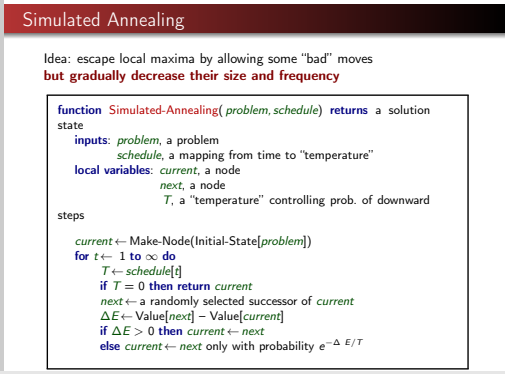
Local search algorithm in this case typically uses a complete-state formulation, where each state has 4 queens on the board, one per column. The successors of a state are all possible states generated by moving a single queen to another square in the same column, so each state has $4 \times 3 = 12$ successors. The heuristic cost function h is the number of pairs of queens that are attacking each other, directly or indirectly. The global minimum is zero, which occurs at a perfect solution. In the figure, at first $h = 5$, then reduces to $h = 2$ and then to $h = 0$. As we move from left to right, we improve the solution at each step.

Example: 8-queens

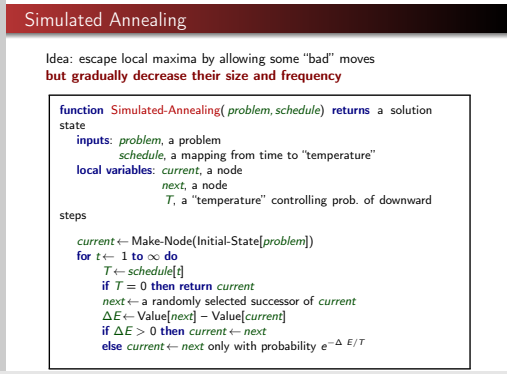


The board shows the value of h for each possible successor obtained by moving a queen within its column.

Local search algorithm in this case typically uses a complete-state formulation, where each state has 4 queens on the board, one per column. The successors of a state are all possible states generated by moving a single queen to another square in the same column, so each state has $4 \times 3 = 12$ successors. The heuristic cost function h is the number of pairs of queens that are attacking each other, directly or indirectly. The global minimum is zero, which occurs at a perfect solution. In the figure, at first $h = 5$, then reduces to $h = 2$ and then to $h = 0$. As we move from left to right, we improve the solution at each step.



A hill climbing algorithm tat **never** makes downhill moves towards lower value of the objective function is guaranteed to be incomplete because it can stuck on a local maxima. In contrast, a purely random walk, that is moving to a successor that is chosen randomly is complete but very inefficient. It seems reasonable to try to combine hill climbing with a random walk. Simulated annealing is such an algorithm. To explain this new algorithm, we switch our attention from hill climbing to gradient descent, i.e., minimizing cost. A good example to motivate this algorithm is getting a ping-pong ball into the deepest crevice in a bumpy surface: if we let the ball simply roll, it may come to rest at a local minima. if we can shake the ball just hard enough to bounce it from the local minima but not too much to dislodge it from the global minimum, we can get the ball to the target. As the ball approaches to the target, we can reduce the intensity of shaking.



Annealing means the process of bringing liquid metal down from a high temperature to crystallize into the desired structure. We start from a high energy state and we try to reach a low energy state (low cost). It's a minimizing instead of maximizing algorithm. The algorithm works as follows:

- Start with a random initial state of energy E and temperature T.
- Perturb the state randomly to obtain a neighbor. Let ΔE be the change in energy between the states.
- The energy is a measure of how disturbed a state is from the optimal solution. The temperature starts high and it slowly goes down with time. This means that "bad moves" are allowed more easily at the beginning than later on.
- If $\Delta E > 0$, move to the neighbor. The move improves the situation, it is always accepted.
- If If $\Delta E < 0$, move to the neighbor with a probability.This probability decreases with the badness of the move $-\Delta E$ is the amount by which the evaluation is worsened.
- example schedule: $T = 1 - \frac{k+1}{k_{max}}$

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

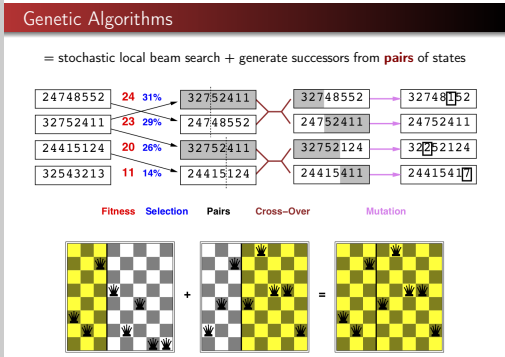
Not the same as k searches run in parallel!
Searches that find good states recruit other searches to join them

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

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

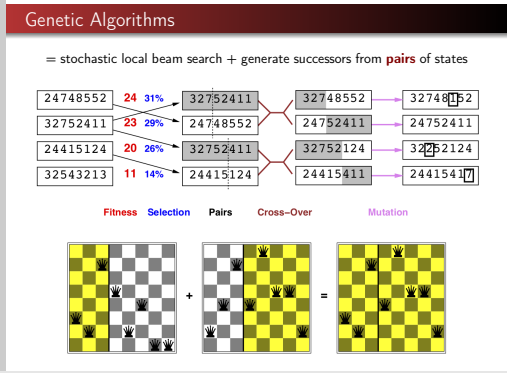
Observe the close analogy to natural selection!

In the hill climbing method, we needed to keep one node in memory, but we may want to keep track of k number of states as opposed to one. It begins with k randomly generated states and at each step all the successors of all k states are generated. If any of the new states is a goal, it halts, otherwise it selects the best k successors from the list and repeats. As the search goes on, useful information is passed among the parallel threads and the algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made. In a variant of local beam search, called stochastic beam search, in order to prevent the search concentrating in a small region of the state space, instead of choosing the best k best successors, the algorithm chooses k successors at random.



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. Like beam searches, GAs begin with a set of k randomly generated states, called the population.Each state, or individual is represented as a string of finite alphabet, most commonly from 0s and 1s. In this figure, a population of four 8-digit strings represent 8-queens state (the problem of n-queens), each digit representing the location of a queen.

In the next step in our figure, each state is rated by the objective function, in GA terminology, by the fitness function. A fitness function should return a higher values for better states when producing the next generation of states. In our example, we use the number of nonattacking pairs of queens, which is 28 for a solution (8*7/2). In the initial state, the scores are 24,23,20, and 11. In this example, the probability of being chosen for reproducing is proportional with the fitness score (24–>31%, 23–>29%, 20–>26%, and 11–>%14).



In the next step, two pairs are selected randomly in accordance with their probabilities in the previous step. That is why you see 3275.. appearing twice in the pairs. For each pair to be mated, a crossover point is chosen randomly from the positions in the string. And next, the offsprings are created by crossing over the parent string at the crossover point. And finally, each location is exposed to random mutation with a small independent probability.

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 $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

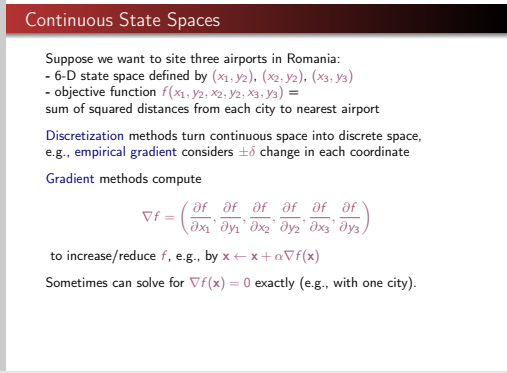
Sometimes can solve for $\nabla f(\mathbf{x}) = 0$ exactly (e.g., with one city).

Most real world environments are continuous. Suppose you want to place three new airports anywhere in Romania such that the sum of squared distances from each city on the map to the nearest airport is minimized. Our state space is defined by the coordinates of the airports: $(x_1, y_1), (x_2, y_2), (x_3, y_3)$. This is a six dimensional space and we say that states are defined by six variables. The objective function for this problem would be:

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

One way to avoid a continuous problem is simply to discretize the neighborhood of each site. What we can do is we can move the location of each airport in six different directions with small amounts, and this gives us 12 possible successors for each state. We can also apply hill climbing or simulated annealing directly without discretisizing.

Continuous State Spaces



What we can do is we can use the gradient method and take the gradient of the objective function with respect to each variable. The gradient ∇f is a vector that gives the magnitude and direction of the slopes. After obtaining the gradient for each variable, we can implement a form of hill climbing that is called steepest-ascent hill climbing by updating the current state by perturbing the current state with amplified or deamplified version of the gradient.