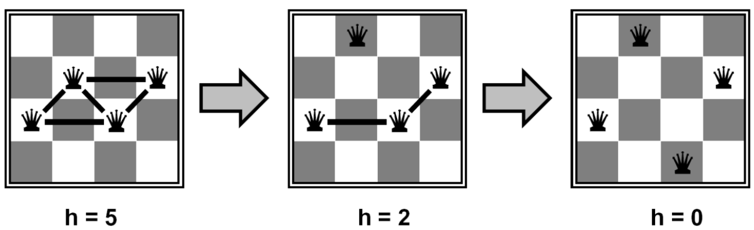
CS 188 Introduction to Artificial Intelligence
Summer 2023 Note 4

These lecture notes are heavily based on notes originally written by Nikhil Sharma.

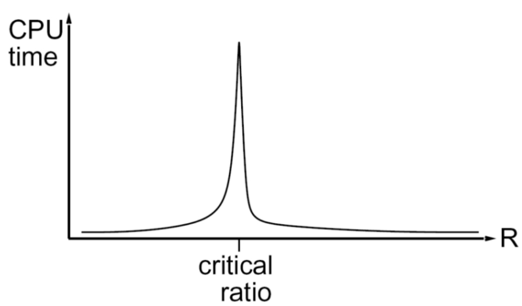
Local Search

As a final topic of interest, backtracking search is not the only algorithm that exists for solving constraint satisfaction problems. Another widely used algorithm is **local search**, for which the idea is childishly simple but remarkably useful. Local search works by iterative improvement - start with some random assignment to values then iteratively select a random conflicted variable and reassign its value to the one that violates the fewest constraints until no more constraint violations exist (a policy known as the **min-conflicts heuristic**). Under such a policy, constraint satisfaction problems like N-queens becomes both very time efficient and space efficient to solve. For example, in following example with 4 queens, we arrive at a solution after only 2 iterations:



In fact, local search appears to run in almost constant time and have a high probability of success not only for N-queens with arbitrarily large N, but also for any randomly generated CSP! However, despite these advantages, local search is both incomplete and suboptimal and so won't necessarily converge to an optimal solution. Additionally, there is a critical ratio around which using local search becomes extremely expensive:

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



The figure above shows the one dimensional plot of an objective function on the state space. For that function we wish to find the state that corresponds to the highest objective value. The basic idea of local search algorithms is that from each state they locally move towards states that have a higher objective value until a maximum (hopefully the global) is reached.

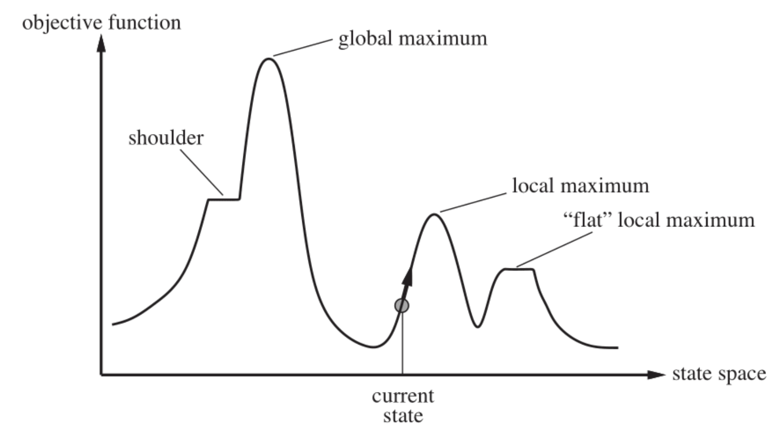


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

We will be covering three such algorithms, **hill-climbing**, **simulated annealing** and **genetic algorithms**. All these algorithms are also used in optimization tasks to either maximize or minimize an objective function.

Hill-Climbing Search

The hill-climbing search algorithm (or **steepest-ascent**) moves from the current state towards a neighboring state that increases the objective value. The algorithm does not maintain a search tree but only the states and the corresponding values of the objective. The "greediness" of hill-climbing makes it vulnerable to being trapped in **local maxima** (see figure 4.1), as locally those points appear as global maxima to the algorithm, and **plateaux** (see figure 4.1). Plateaux can be categorized into "flat" areas at which no direction leads to improvement ("flat local maxima") or flat areas from which progress can be slow ("shoulders"). Variants of hill-climbing, like **stochastic hill-climbing** which selects an action randomly among the uphill moves, have been proposed . This version of hill-climbing has been shown in practice to converge to higher maxima at the cost of more iterations.

```
function HILL-CLIMBING(problem) returns a state
  current ← make-node(problem.initial-state)
  loop do
      neighbor ← a highest-valued successor of current
      if neighbor.value ≤ current.value then
          return current.state
      current ← neighbor
```

The pseudocode of hill-climbing can be seen above. As the name suggests the algorithm iteratively moves to a state with higher objective value until no such progress is possible. Hill-climbing is incomplete. **Random-Restart hill-climbing** on the other hand, that conducts a number of hill-climbing searches each time from a randomly chosen initial state, is trivially complete as at some point the randomly chosen initial state will coincide with the global maximum.

Simulated Annealing Search

The second local search algorithm we will cover is simulated annealing. Simulated annealing aims to combine random walk (randomly moves to nearby states) and hill-climbing to obtain a complete and efficient search algorithm. In simulated annealing we allow moves to states that can decrease the objective. More specifically, the algorithm at each state chooses a random move. If the move leads to higher objective it is always accepted. If on the other hand it leads to smaller objectives then the move is accepted with some probability. This probability is determined by the temperature parameter, which initially is high (more “bad" moves allowed) and gets decreased according to some schedule. If temperature is decreased slowly enough then the simulated annealing algorithm will reach the global maximum with probability approaching 1.

```
function SIMULATED-ANNEALING(problem,schedule) returns a state
current ← problem.initial-state
for t = 1 to ∞ do
    T ←schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← next.value – current.value
    if ΔE > 0 then current ← next
           else current ← next only with probability e^(ΔE/T)
```

Genetic Algorithms (Optional)

Finally, we present **genetic algorithms** which are a variant of local beam search and are also extensively used in many optimization tasks. Genetic algorithms begin as beam search with k randomly initialized states called the **population**. States (or **individuals**) are represented as a string over a finite alphabet. To understand the topic better let's revisit the 8 Queens problem presented in class. For the 8 Queens problem we can represent each of the eight individuals with a number that ranges from $1-8$ representing the location of each Queen in the column (column (a) in Fig. 4.6). Each individual is evaluated using an evaluation function (**fitness function**) and they are ranked according to the values of that function. For the 8 Queens problem this is the number of non-attacking pairs of queens.

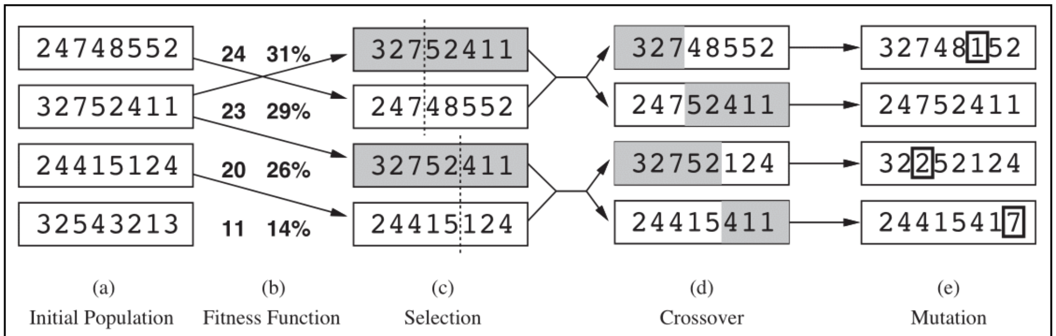


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

The probability of choosing a state to "reproduce" is propositional to the value of that state. We proceed to select pairs of states to reproduce according to these probabilities (column (c) in Fig. 4.6). Offsprings are generated by crossing over the parent strings at the crossover point. That crossover point is chosen randomly for each pair. Finally, each offspring is susceptible to some random mutation with independent probability. The pseudocode of the genetic algorithm can be seen in the following picture.

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
    $new\_population \leftarrow$ empty set
    for i = 1 **to** SIZE($population$) **do**
      $x \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
      $y \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
      $child \leftarrow$ REPRODUCE(x, y)
      if (small random probability) **then** $child \leftarrow$ MUTATE($child$)
      add $child$ to $new\_population$
    $population \leftarrow new\_population$
  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

  $n \leftarrow$ LENGTH(x); $c \leftarrow$ random number from 1 to n
  return APPEND(SUBSTRING($x, 1, c$), SUBSTRING($y, c + 1, n$))

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

Genetic algorithms try to move uphill while exploring the state space and exchanging information between threads. Their main advantage is the use of crossovers since this allows for large blocks of letters, that have evolved and lead to high valuations, to be combined with other such blocks and produce a solution with high total score.