# CS 188 Introduction to Artificial Intelligence
# Spring 2022 Note 2

These lecture notes are based on notes originally written by Nikhil Sharma and the textbook Artificial Intelligence: A Modern Approach.

## Local Search

In the previous note, we wanted to find the goal state, along with the optimal path to get there. But in some problems, we only care about finding the goal state — reconstructing the path can be trivial. For example, in Sudoku, the optimal configuration is the goal. Once you know it, you know to get there by filling in the squares one by one.

Local search algorithms allow us to find goal states without worrying about the path to get there. In local search problems, the state space are sets of "complete" solutions. We use these algorithms to try to find a configuration that satisfies some constraints or optimizes some objective function.

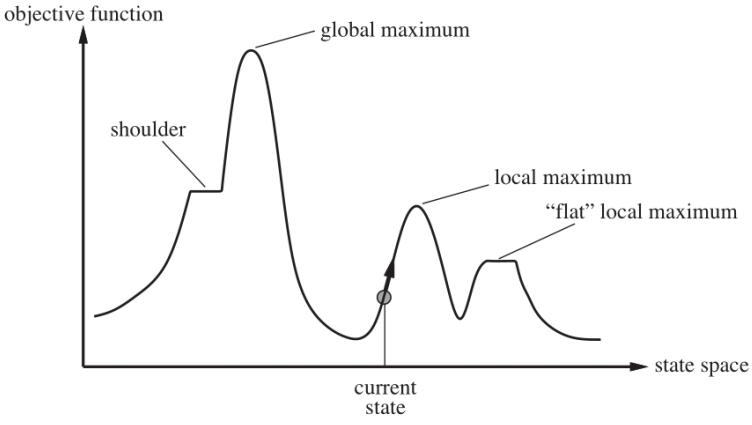



**Figure 4.1** A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

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. We will be covering four such algorithms, **hill-climbing**, **simulated annealing**, **local beam search** 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 the neighboring state that increases the objective value the most. 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 **plateaus** (see figure 4.1). Plateaus 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 possible uphill moves, have been proposed. Stochastic hill-climbing has been shown in practice to converge to higher maxima at the cost of more iterations. Another variant, **random sideways moves**, allows moves that don't strictly increase the objective, allowing the algorithm to escape "shoulders".

```
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, which conducts a number of hill-climbing searches from randomly chosen initial states, is trivially complete as at some point a randomly chosen initial state can converge to the global maximum.

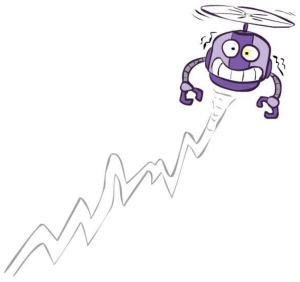
As a note, later in this course you will encounter the term "gradient descent". It is the exact same idea as hill-climbing, except instead of maximizing an objective function we will want to minimize a cost function.

# 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.

The algorithm chooses a random move at each timestep. If the move leads to higher objective value, it is always accepted. If it leads to a smaller objective value, 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". Theoretically, if temperature is decreased slowly enough, 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)
```



## Local Beam Search

Local beam search is another variant of the hill-climbing search algorithm. The key difference between the two is that local beam search keeps track of $k$ states (threads) at each iteration. The algorithm starts with a random initialization of $k$ states and at each iteration it takes on $k$ new states as done in hill-climbing. These aren't just $k$ copies of the regular hill-climbing algorithm. Crucially, the algorithm selects the $k$ best successor states from the complete list of successor states from all the threads. If any of the threads finds the optimal value the algorithm stops.

The $k$ threads can share information between them, allowing "good" threads (for which objectives are high) to "attract" the other threads in that region as well.

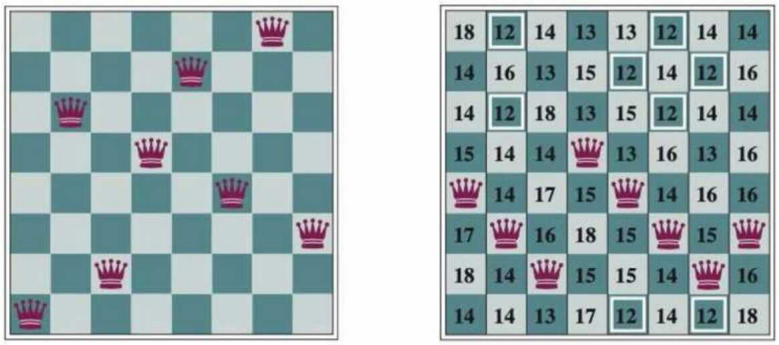
Local beam search is also susceptible in getting stuck in "flat" regions like hill-climbing does. **Stochastic beam search**, analogous to stochastic hill-climbing, can alleviate this issue.

## Genetic Algorithms

Finally, we present **genetic algorithms** which are a variant of local beam search and are also extensively used in many optimization tasks. As indicated by the name, genetic algorithms take inspiration from evolution. Genetic algorithms begin as beam search with $k$ randomly initialized states called the **population**. States (called **individuals**) are represented as a string over a finite alphabet.

Let's revisit the 8**-Queens problem** presented in lecture. As a recap, 8-Queens is a constraint-satisfaction problem where we hope to situate 8 queens on an 8-by-8 board. The constraint-satisfying solution will not have any **attacking pairs** of queens, which are queens that are in the same row, column, or diagonal. All of the previously covered algorithms are possible ways to approach the 8-Queens problem.

Figure 4.3




(a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h = 17$. The board shows the value of $h$ for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h = 12$. The hill-climbing algorithm will pick one of these.

For a genetic algorithm, we represent each of the eight queens with a number from $1-8$ representing the location of each queen in its 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 (non-conflicted) pairs of queens.

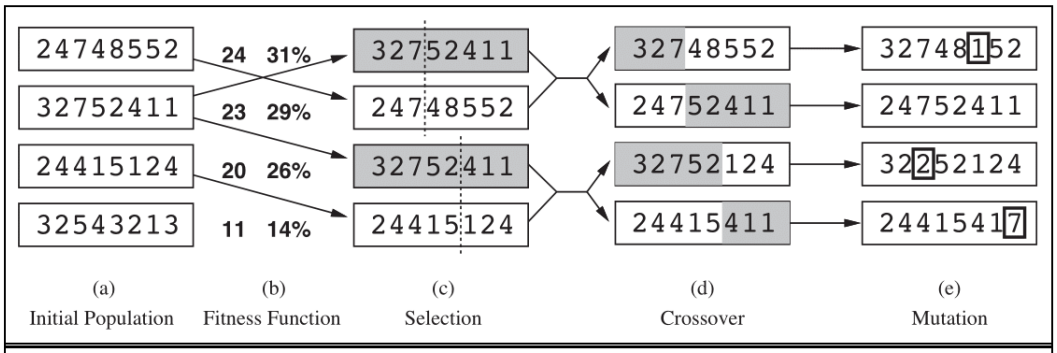



**Figure 4.6** The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

The probability of choosing a state to “reproduce" is proportional to the value of that state. We proceed to select pairs of states to reproduce by sampling from these probabilities (column (c) in Fig. 4.6). Offspring are generated by crossing over parent strings at the crossover point. The 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 below.

```
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
```

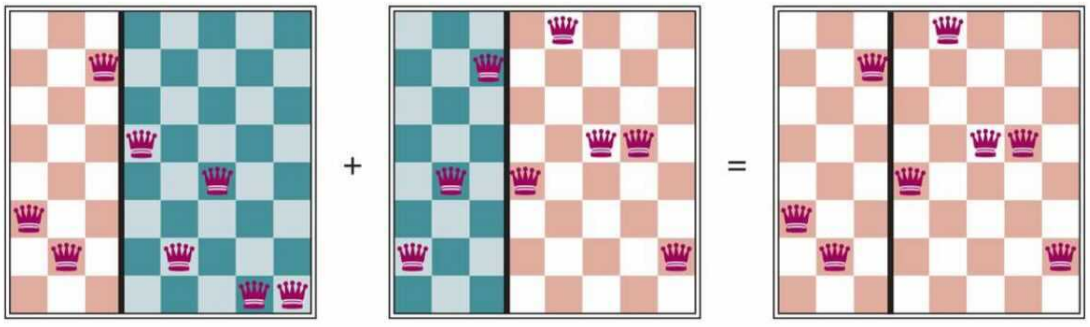
```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

**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.

Similar to stochastic beam search, genetic algorithms try to move uphill while exploring the state space and exchanging information between threads. Their main advantage is the use of crossovers — large blocks of letters that have evolved and lead to high valuations can be combined with other such blocks and produce a solution with high total score.

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 green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.6: row 1 is the bottom row, and 8 is the top row.)

# Summary

In this note, we discussed *local search algorithms* and their motivation. We can use these approaches when we don't care about the path to some goal state, and want to satisfy constraints or optimize some objective. Local search approaches allow us to save space and find adequate solutions when working in large state spaces!

We went over a few foundational local search approaches, which build upon each other:

- *Hill-Climbing*
- *Simulated Annealing*
- *Local Beam Search*
- *Genetic Algorithms*

The idea of optimizing a function will reappear later in this course, especially when we cover neural networks.