UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Artificial Intelligence

Search in Complex Environments

LESSON 6

prof. Antonino Staiano

M.Sc. In ''Machine Learning e Big Data'' - University Parthenope of Naples

Informed Search

- Uniformed and informed search concern with finding a solution as a sequence of actions
  - The environments are fully observable, deterministic, static, and known
- Now, we want to relax some of those constraints
  - Finding a good state without considering the path to get there
    - Discrete and continuous states
  - Nondeterministic environments

What we are going to learn today

- Local Search and Optimization Problems
  - Hill-climbing
  - Simulated annealing
- Genetic algorithms
- Local search in continuous spaces
- Belief states and conditional plans

Optimization Problems

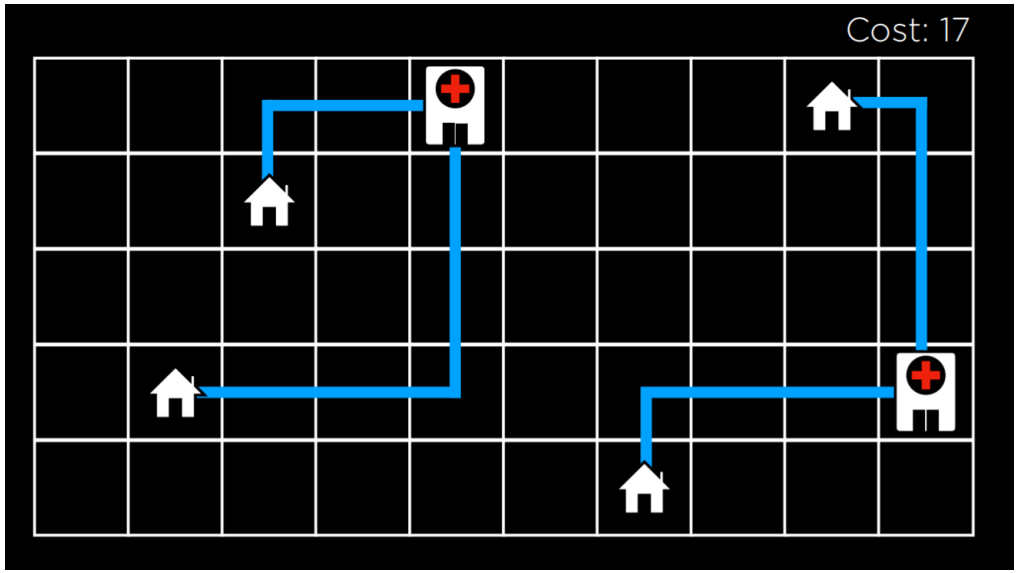
- In search problems examined so far, the agent needed to find a path from a source to a destination
  - E.g., a path from Arad to Bucharest
- In many optimization problems, the path is irrelevant
  - The goal state itself is the solution, that is, the objective is to choose the best option from a set of possible options
    - We care only about finding a valid final configuration
      - 8-queens
      - Integrated-circuits design
      - Job shop scheduling
      - Telecommunications network optimization
      - Crop planning

Local Search and Optimization Problems

- Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths nor the set of states that have been reached
  - local search is interested in finding the best answer to a question
- The state space is a set of "complete" configurations
  - find the optimal configuration, e.g., TSP
  - find configuration satisfying constraints, e.g., timetable
- We can use iterative improvement algorithms
  - keep a single "current" state and try to improve it
- Often, local search will bring to an answer that is not optimal but "good enough"
  - They are not systematic; they might never explore a portion of the search space where a solution actually resides

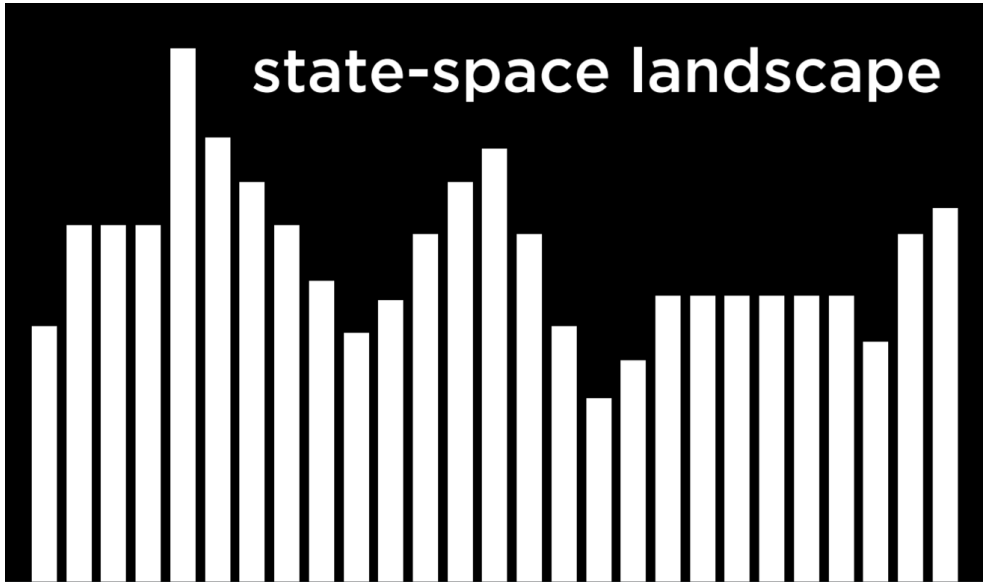
Example

- We have four houses in set locations
  - Goal: build two hospitals such that the distance from each house to a hospital is minimized
    - A state is any configuration of houses and hospitals



State-space Landscape

- We can represent each configuration of houses and hospitals as a state-space landscape
  - Each of the bars represents a value of a state, e.g., the cost of a certain configuration of houses and hospitals

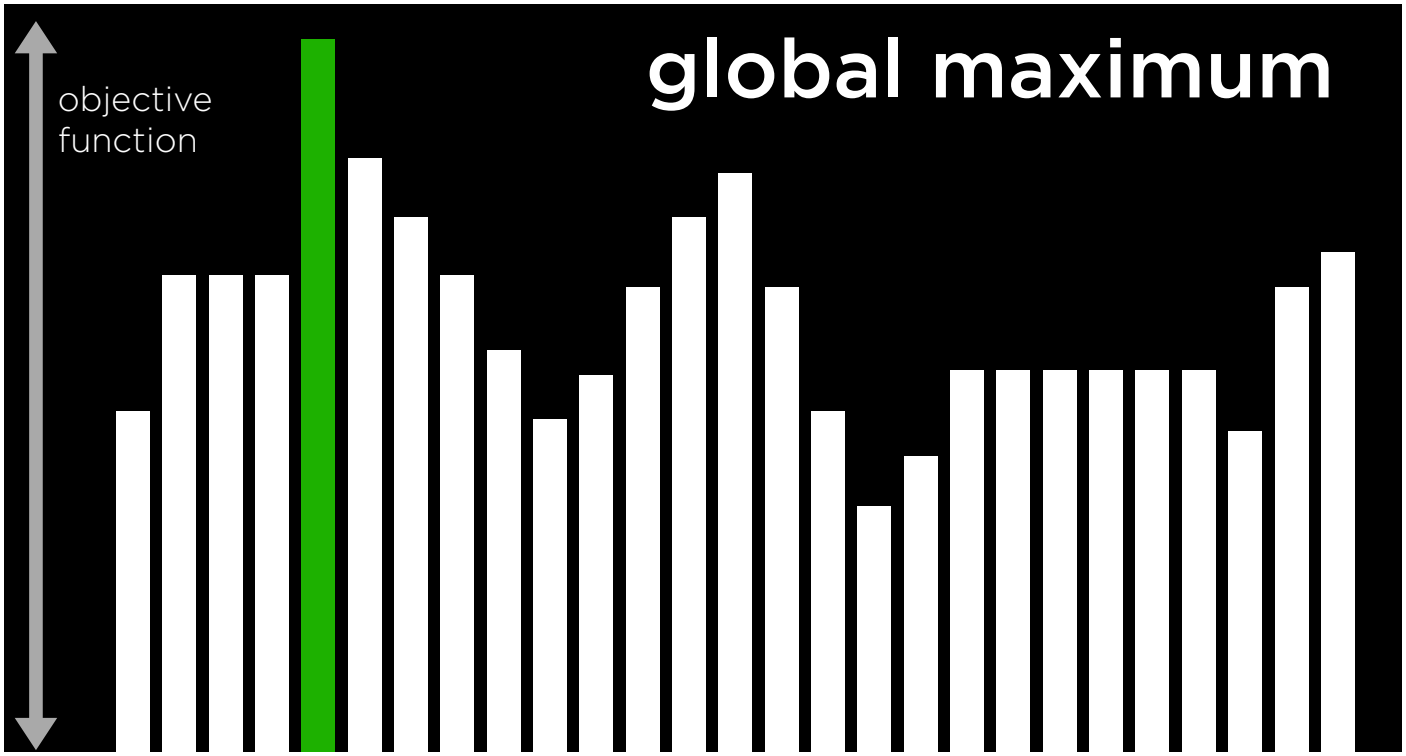


Local Search and optimization ingredients

- Objective function or Cost function
  - A function that we use to maximize (minimize) the value (the cost) of the solution
- Current state
  - The state that is currently being considered by the function
- Neighbor state
  - A state that the current state can transition to
    - Usually similar to the current state, so its value is close to the current state's value
- Local search algorithms work by considering one node in a current state and then moving to a node of the current state's neighbors

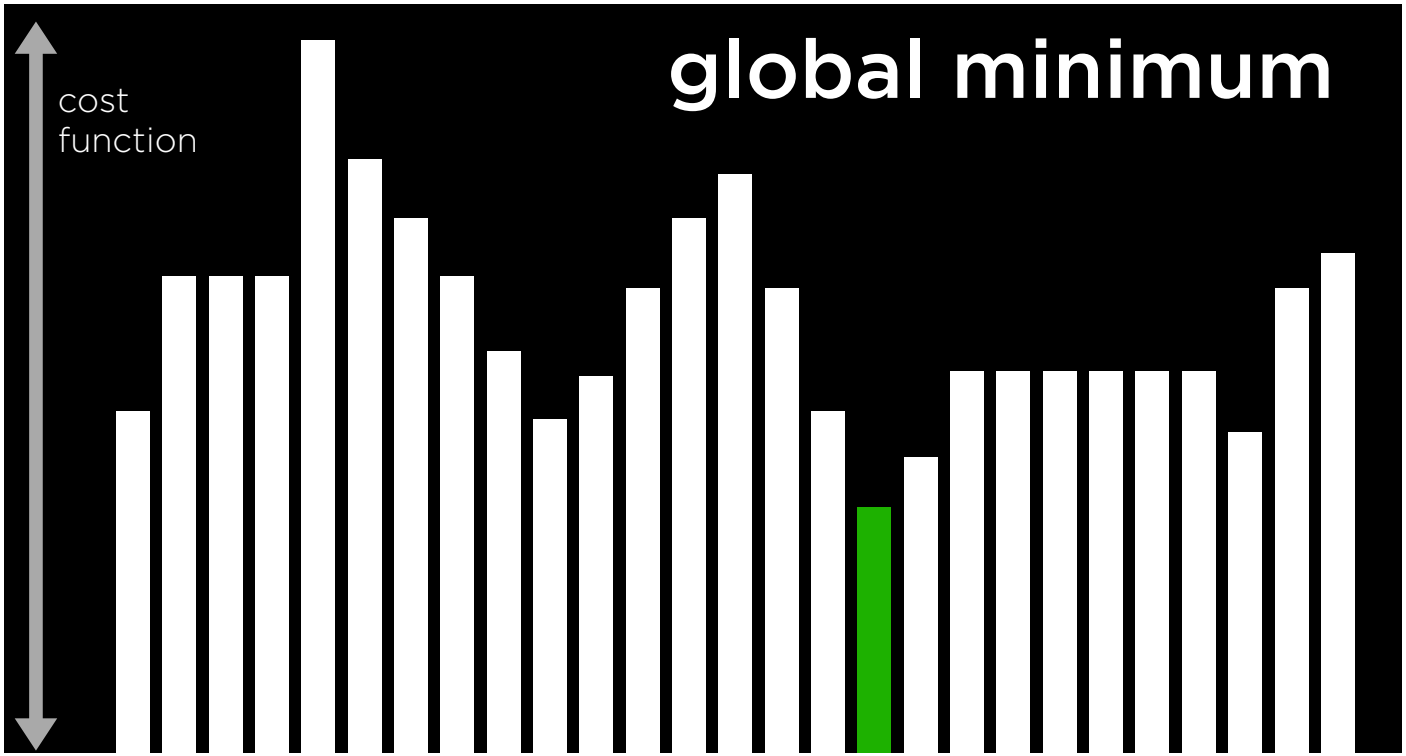
Maximization problem

- Optimal solution: global maximum

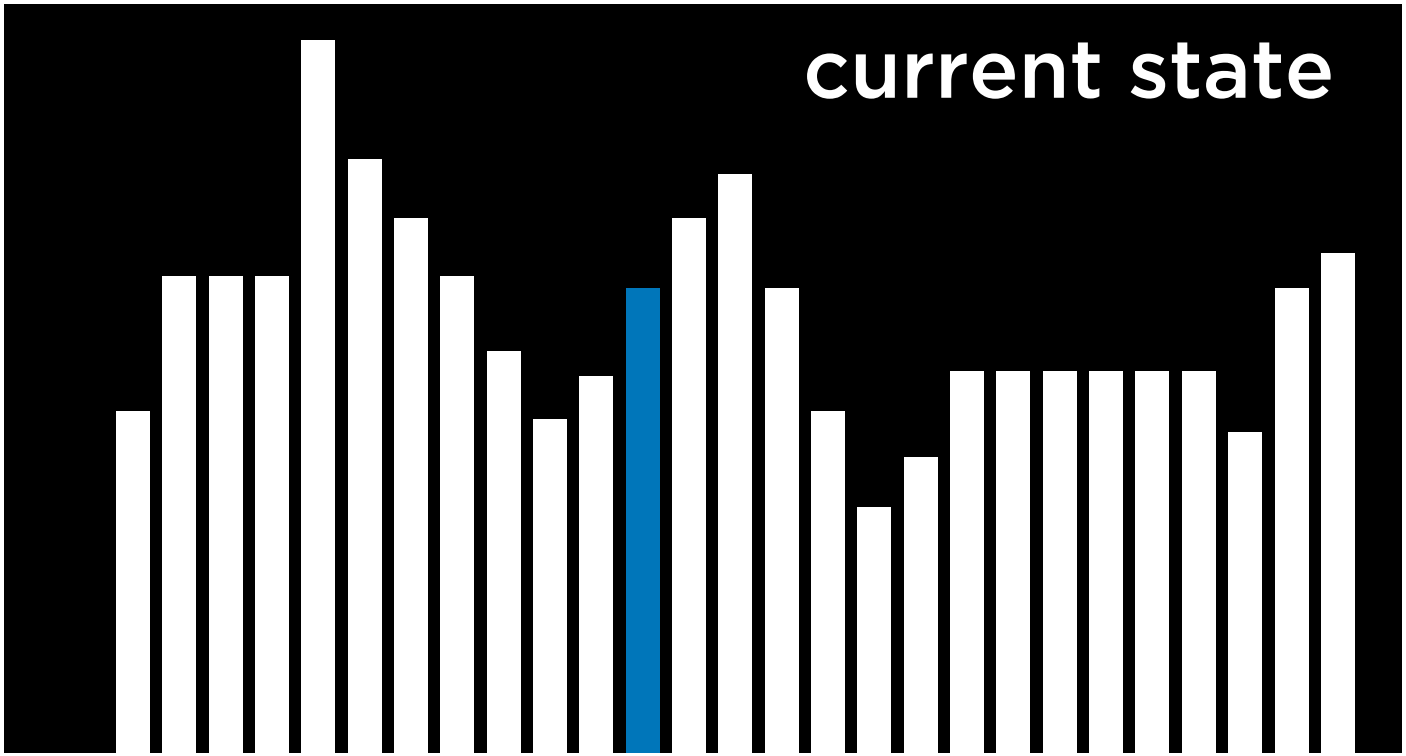


Minimization problem

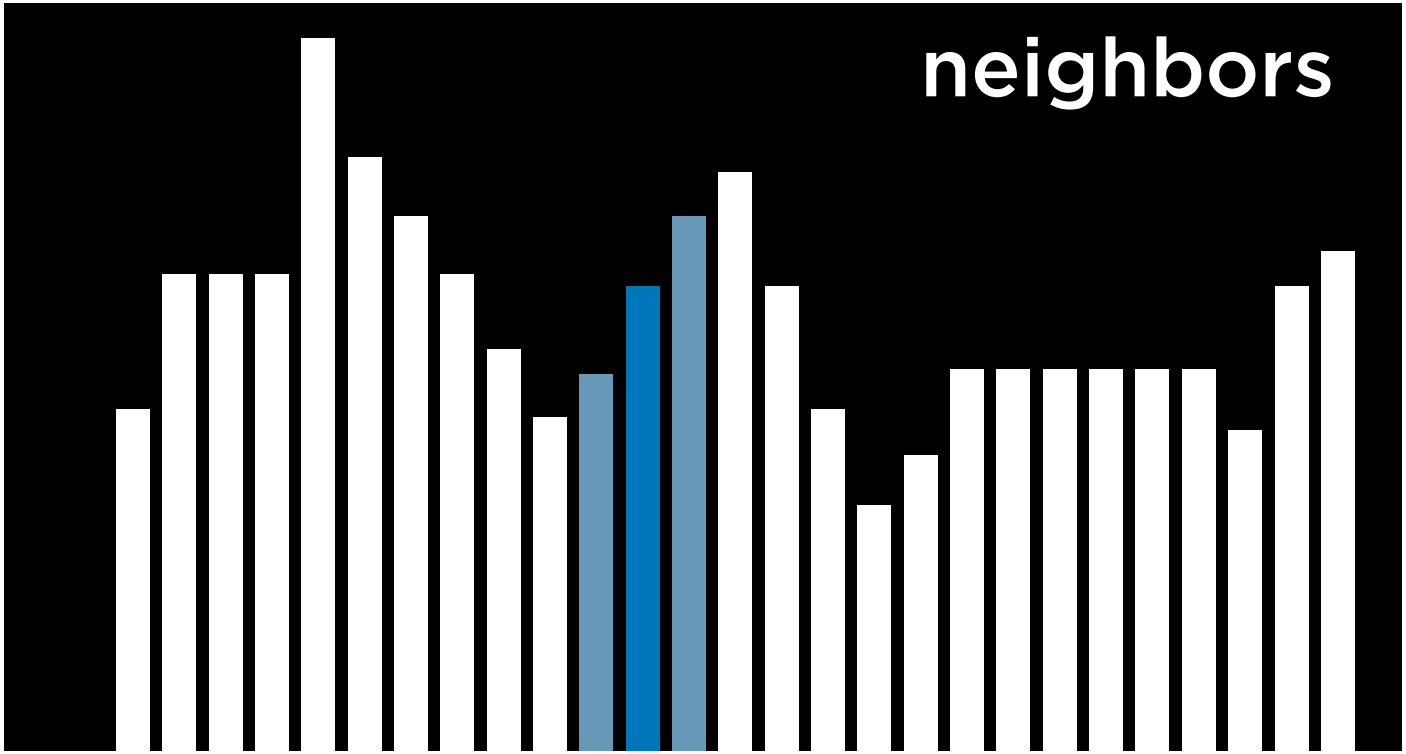
- Optimal solution: global minimum



Current state

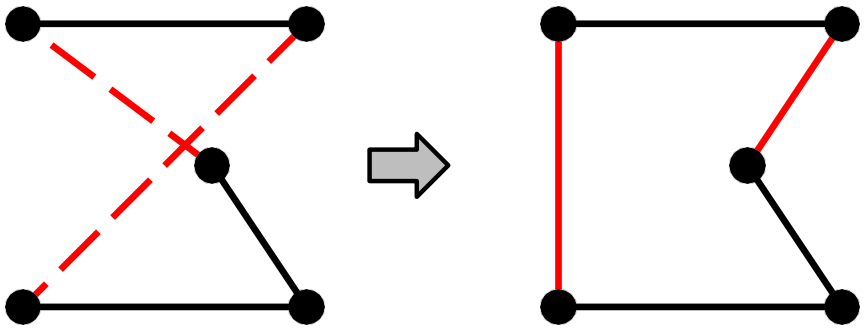


Current state's neighbors



Example: TSP

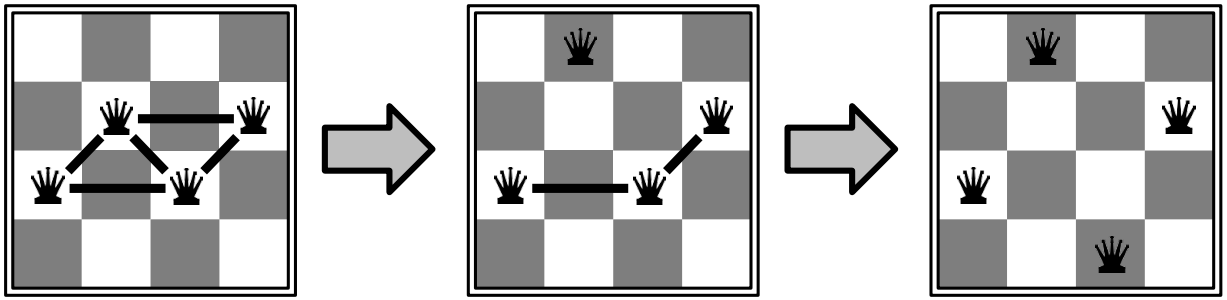
- Start with any complete tour, perform pairwise exchange



- Variants of this approach get within 1% of optimal very quickly with thousands of cities

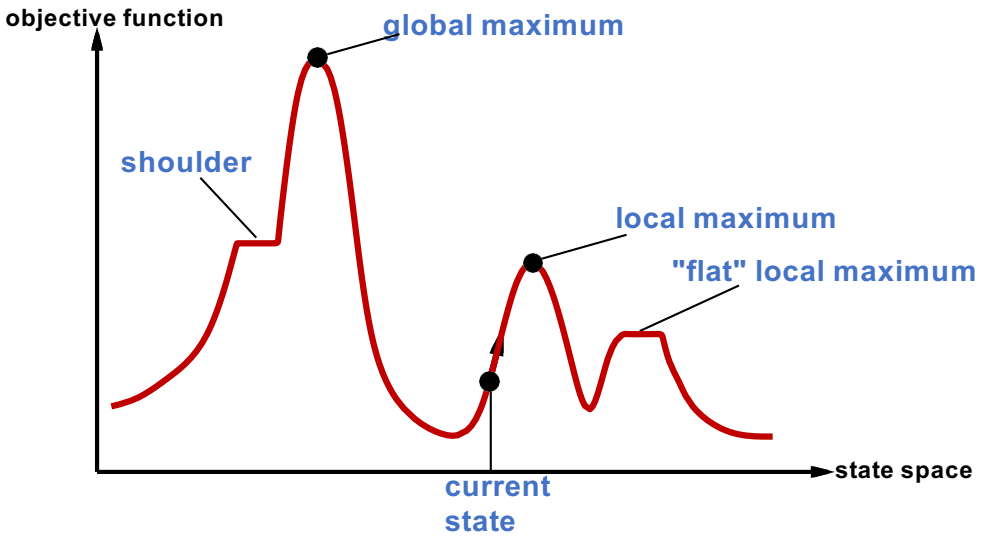
Example: n-Queens

- Put n-queens on a nxn board, with no queen on the same row, column or diagonal
- Move a queen to reduce the number of conflicts



Local Search

- Local search algorithms solve optimization problems
  - Find the best state according to an objective function

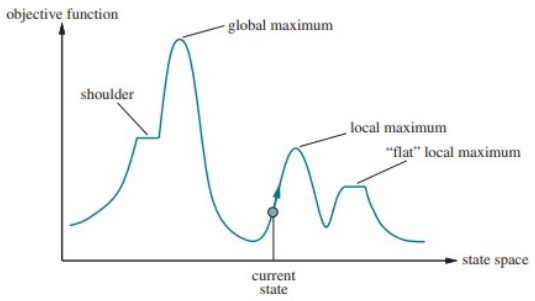


Hill-climbing Search

- It keeps track of one current state and on each iteration moves to the neighboring state with the highest value
  - It heads in the direction that provides the steepest ascent
  - Stops at a peak with no neighbor with a higher value
  - Does not look ahead beyond the immediate neighbors of the current state
- In a Hill-climbing search, the negative of a heuristic cost function can be used as the objective function
  - that will climb locally to the state with the smallest heuristic distance to the goal

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
    neighbor ← a highest-valued successor state of current
    if VALUE(neighbor) ≤ VALUE(current) then return current
    current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.



Hill-climbing Search and 8-Queens

- The initial state is chosen at random
- The successors of a state are all possible states generated by moving one queen to another square in the same column (56 successors)
- The heuristic cost function h is the number of pairs of queens that are attacking each other
  - Zero only for solutions
  - Count as an attack if two pieces in the same line with an intervening piece between them

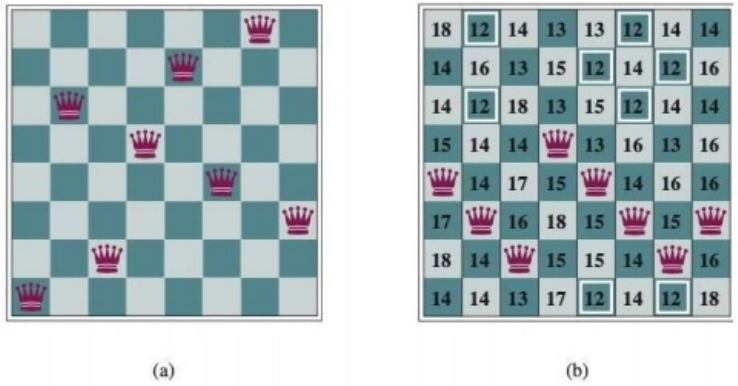
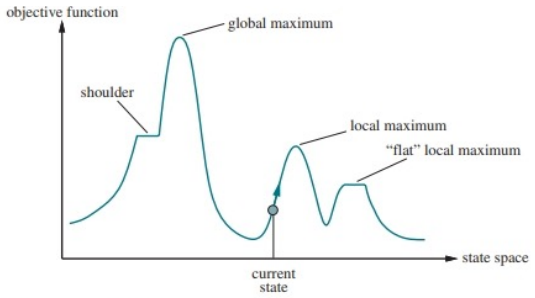
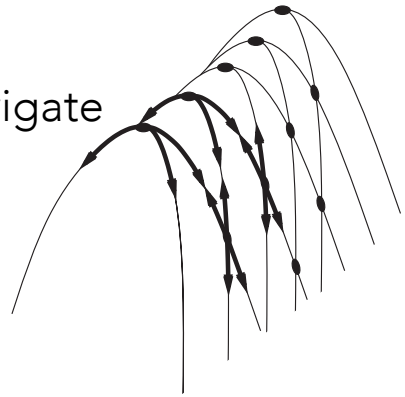


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (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.

Properties of Hill-climbing



- Sometimes called greedy local search
- Hill-climbing can get stuck for several reasons
  - Local maxima
    - A peak higher than each of its neighbors but lower than the global maximum
  - Ridges
    - A sequence of local maxima that is difficult for greedy algorithms to navigate
  - Plateau
    - Flat area of the state-space landscape
      - Local maximum from which no uphill exists
      - Shoulder from which progress is possible



- In each case, the algorithm reaches a point at which no progress is being made

Hill-Climbing Improvements

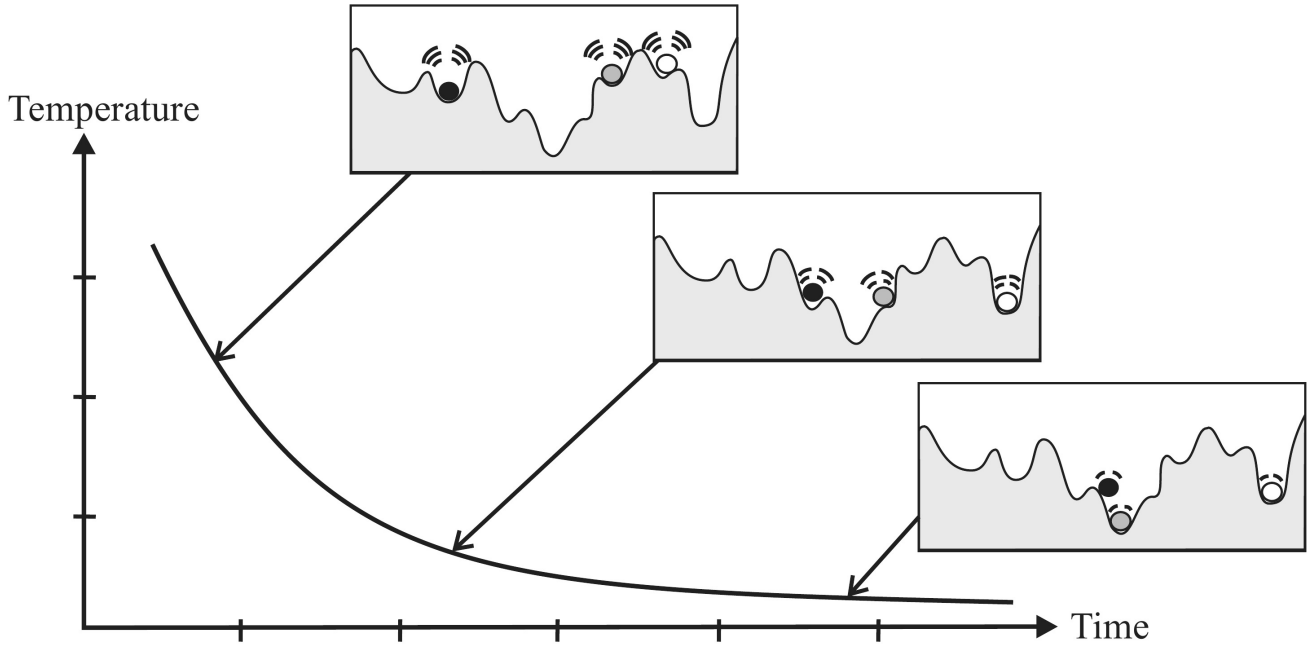
- Sideways move
  - When a plateau is reached keep going (works only for a shoulder)
    - We can limit the number of consecutive sideways moves
      - It raises the percentage of problem instances solved (8-queens) from 14% to 94%
- Stochastic hill-climbing
  - Randomly chooses among uphill moves with a probability of selection varying with their steepness
    - Slower convergence, but better solutions (sometimes)
- First-choice hill climbing
  - A stochastic hill climbing that generates successors randomly until one is better than the current state
    - Good strategy when state as many (e.g., thousands) of successors
- Random-restart hill climbing
  - A series of hill-climbing searches from randomly generated initial state, until a goal is found
- Hill-climbing performance depends upon the shape of the landscape

Simulated Annealing

- Hill-climbing is always vulnerable to getting stuck in a local maximum
  - At the other extreme, a pure random walk will eventually reach the global maximum
    - However, extremely inefficient
- Simulated annealing combines both worlds for yielding both efficiency and completeness
  - Annealing is the process to harden metals and glass by heating them to a high temperature
    - Then, gradually cooling the material allows it to reach a low-energy crystalline state

Simulated Annealing

- To understand simulated annealing let's view the problem as a gradient descent (that is, minimizing the cost)



Simulated Annealing Algorithm

- Pick a random move
  - If the move leads to an improvement, accept it
  - Otherwise, the move is accepted with some probability p < 1
    - The probability decreases exponentially according to the *badness* of a move

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
  current ← problem.INITIAL
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE(current) – VALUE(next)
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

The probability decreases exponentially with the amount ΔE by which the evaluation is worsened.
The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the "temperature" T as a function of time.

Local Beam Search

- The local beam search algorithm keeps track of k states rather than just one
  - Randomly generates k states
  - At each step, all the successors of all k are generated
    - If anyone is a goal, stop
    - Otherwise, select the k best successors from the complete list and repeat
- A local beam search with k states might seem as running parallel k random restarts instead of in sequence
  - However, in a random-restart search, each search process runs independently of the others, whereas, in a local beam search, useful information is passed among the parallel search threads
  - The algorithm quickly leaves unfruitful searches and moves its resources to where the most progress is being made
- A variant called stochastic beam search chooses successors with probability proportional to the successor's value to increase diversity

Evolutionary Algorithms

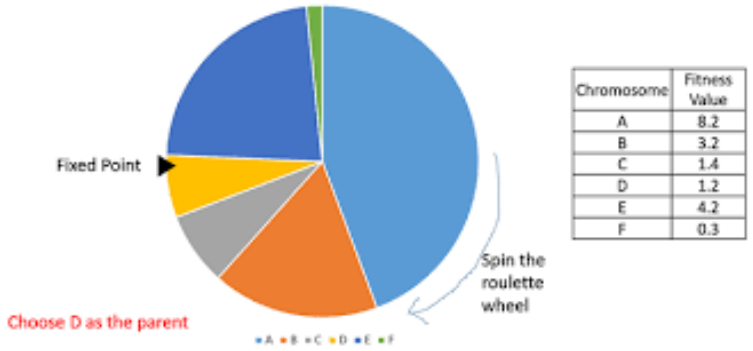
- Motivated by the metaphor of natural selection in biology
  - It is created a population of individuals (the state, that is, the solutions)
  - The fittest individuals (highest value) produce offspring (successor states)
    - This process is called recombination
    - The offspring after recombination form the next generation population
- Several variants of this evolutionary scheme exist
  - Genetic algorithms
  - Evolutionary strategy
  - Genetic programming

Genetic Algorithms (GAs)

- The population has a fixed size (number of individuals)
- Each individual, called a chromosome, is represented by a string over a finite alphabet (usually, a binary string)
- Each chromosome has a fitness value determining its goodness
- The population evolves through several generations
  - In each generation, the chromosomes are applied to three genetic operators
    - Selection
    - Crossover
    - Mutation

GA Operators: Selection

- Selects the chromosomes who will become parents of the next generation
  - The individuals are chosen with a probability proportional to their fitness value
    - This basic scheme is called roulette-wheel selection



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

GA Operators: Crossover

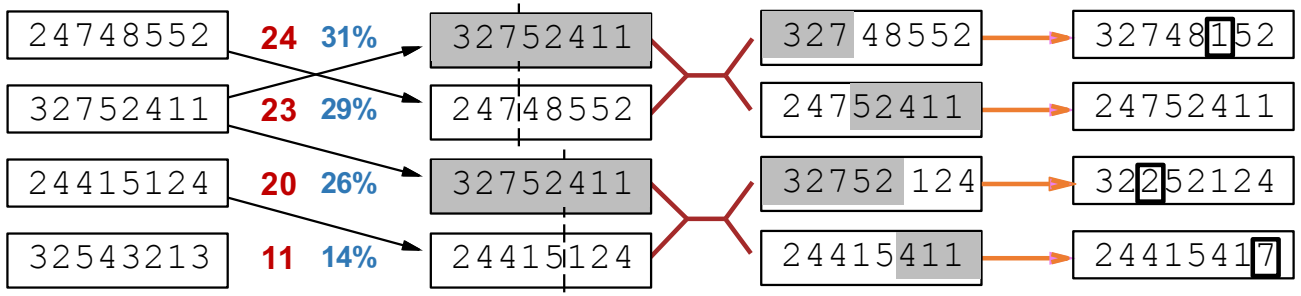
- Crossover is the operator for recombination
  - Once selected a pair of parents, it is randomly selected a crossover point where each parent string is split
  - The split substrings of one parent are recombined with the ones of the other parent to recombine and form the children (that, is the chromosome of the next generation)
    - one with the first part of parent 1 and the second part of parent 2
    - the other with the second part of parent 1 and the first part of parent 2

GA Operators: Mutation

- Mutation randomly changes a symbol in the chromosome representation with a given probability
  - Once the offspring are generated, every bit in its representation is flipped with probability equal to a mutation rate
- Eventually, the next generation is formed
  - It could be just the newly formed offspring or
  - A few top-scoring parents from the previous generation are hold
    - Elitism
      - Guarantees that the overall fitness will never decrease over time

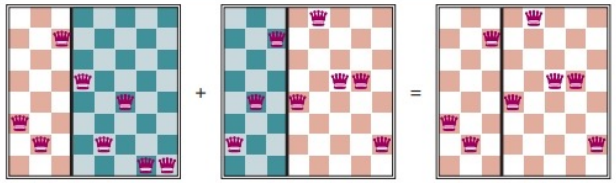
The Genetic Algorithm

- A GA with chromosomes representing 8-queens states
  - The initial population is ranked by a fitness function
  - The parents are then chosen for reproduction
  - The offspring are generated
  - Mutation possibly arises



Each state is rated by the fitness function: Higher fitness values are better, so we use the number of nonattacking pairs of queens, which has a value of 8×7/2 = 28 for a solution

Fitness **Selection** **Pairs** **Cross-Over** **Mutation**



How Gas Works

- Schema
  - a substring in which some of the positions can be left unspecified
    - the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6, respectively (they don't attack each other)
    - Strings that match the schema (such as 24613578) are called instances of the schema
  - It can be shown that if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema will grow over time

GA Implementation

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Figure 4.8 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Local Search in Continuous Spaces

- When the environment is continuous the branching factor is infinite, so the algorithms described so far are unsuitable
- Suppose we want to site three airports in Romania:
  - 6-D state space defined by (x1, y2), (x2, y2), (x3, y3)
  - objective function f (x1, y2, x2, y2, x3, y3) = sum of squared distances from each city to the nearest airport
  - This equation is correct for the state x and states in the local neighborhood of x
  - However, it is not correct globally; if we stray too far from x then the set of closest cities for that airport changes, and we need to recompute the nearest airports
- Discretization methods turn continuous space into discrete space and consider $\pm\delta$ changes in each coordinate. Alternatively, empirical gradient methods
- Gradient methods compute (f is expressed in analytical mathematical form)

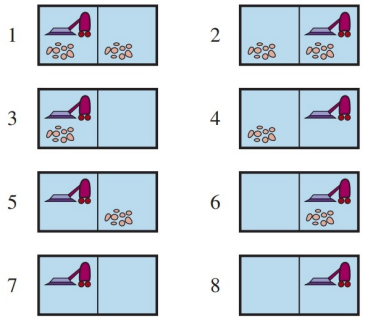
$$\nabla f = \left(\frac{\partial f}{\partial x^1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y^3} \right)$$

to increase/reduce f by $\mathrm{x} \leftarrow \mathrm{x} + a\nabla f(\mathrm{x})$

Sometimes can solve for $\nabla f(\mathrm{x}) = 0$ exactly (e.g., with one city). Newton–Raphson (1664, 1690) iterates $\mathrm{x} \leftarrow \mathrm{x} - \mathrm{H}_f^{-1}(\mathrm{x})\nabla f(\mathrm{x})$ to solve $\nabla f(\mathrm{x}) = 0$, where $\mathrm{H}_{ij} = \partial^2 f / \partial x_i \partial x_j$

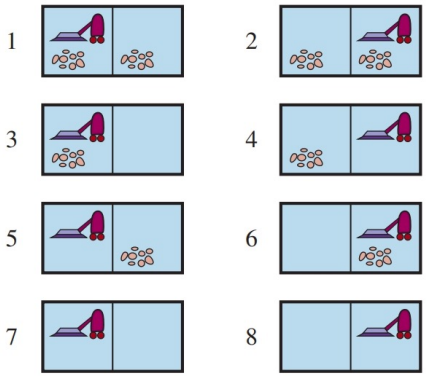
Nondeterministic Actions

- When the environment is partially observable and nondeterministic
  - the agent doesn't know for sure what state it is in
  - the agent doesn't know what state it transitions to after taking an action
    - Rather, it will know the possible states it will be in, which is called "**belief state**"
- The solution to a problem is, rather than a longer sequence, a conditional plan
  - It specifies what to do based on what it perceives while executing the plan
- Let's consider the erratic vacuum world
  - The suck action is
    - In a dirty square, clean it, and sometimes clean up dirt in an adjacent square
    - In a clean square the action sometimes dirt on the carpet
  - To precisely define the problem, a generalized transition model should be accounted



The Erratic Vacuum World

- The Result function returns a set of possible outcome states
  - Results(1,Suck) = {5,7}
  - Starting in state 1, no single sequence solves the problem, instead, a conditional plan should be followed
    - [Suck, if State=5 then [Right, Suck] else []]
- The plan contains if-then-else statement, so the solutions are trees
  - The if statement tests to see what the current state is
    - Observe it at a runtime
  - It doesn't know it at planning time
  - Alternatively, we could have a formulation where the agent tests the percepts rather than the state



AND-OR Search Trees

- How do we find these solutions to nondeterministic problems?
  - We build search trees
    - **AND-OR** search trees. Two possible actions: agent choice, **OR nodes,** and branching that happens from a choice (environment's choice of outcome for each action), **AND nodes**

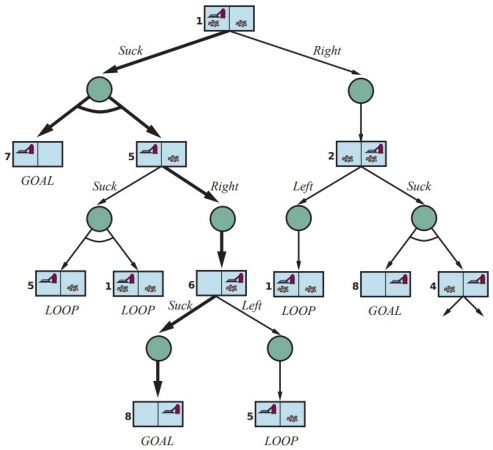


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

function AND-OR-SEARCH(*problem*) **returns** a conditional plan, or *failure*
  return OR-SEARCH(*problem*, *problem*.INITIAL, [])

function OR-SEARCH(*problem*, *state*, *path*) **returns** *a conditional plan, or failure*
  if *problem*.IS-GOAL(*state*) **then return** the empty plan
  if IS-CYCLE(*state*, *path*) **then return** *failure*
  for each *action* **in** *problem*.ACTIONS(*state*) **do**
    plan ← AND-SEARCH(*problem*, RESULTS(*state*, *action*), [*state*] + [*path*])
    if *plan* ≠ *failure* **then return** [*action*] + [*plan*]
  return *failure*

function AND-SEARCH(*problem*, *states*, *path*) **returns** *a conditional plan, or failure*
  for each s_i **in** *states* **do**
    $plan_i$ ← OR-SEARCH(*problem*, s_i, *path*)
    if $plan_i$ = *failure* **then return** *failure*
  return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** ... **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. A solution is a conditional plan that considers every nondeterministic outcome and makes a plan for each one.

Try, try again

- Slippery vacuum world
  - Movement actions sometimes fail
    - The agent remains in the same location
  - There are no longer any acyclic solutions

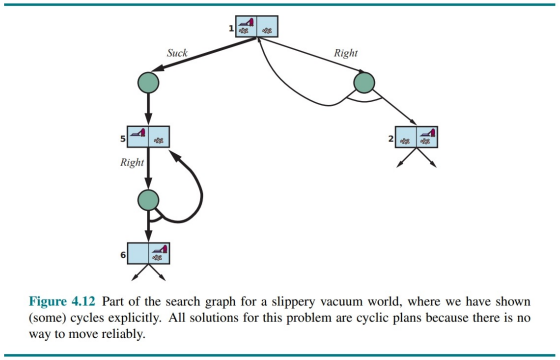


Figure 4.12 Part of the search graph for a slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

- A cyclic plan solution where a minimum condition to hold is that every leaf is a goal state and that a leaf is reachable from every point in the plan
  - E.g., from state 1, the possible actions are {1,2}
    - Keep trying Right until it works
    - [Suck, **while** State = 5 **do** Right, Suck]
- A cyclic plan is a solution if (minimum condition)
  - Every leaf is a goal state, and that leaf is reachable from every point in the plan