CS:4420 Artificial Intelligence

Spring 2018

Beyond Classical Search

Cesare Tinelli

The University of Iowa

Readings

- Chap. 4 of [Russell and Norvig, 2012]

Beyond Classical Search

We have seen methods that systematically explore the search space, possibly using principled pruning (e.g., A*)

The best of these methods can currently handle search spaces of up to 10^{100} states / ~1,000 binary variables (ballpark figure)

What if we have much larger search spaces?

Search spaces for some real-world problems may be much larger e.g., $10^{30,000}$ states as in certain reasoning and planning tasks

Some of these problems can be solved by Iterative Improvement Methods

Iterative Improvement Methods

In many optimization problems the goal state itself is the solution

The state space is a set of *complete* configurations

Search is about finding the optimal configuration (as in TSP) or just a feasible configuration (as in scheduling problems)

In such cases, one can use iterative improvement, or local search, methods

An evaluation, or *objective*, function h must be available that measures the quality of each state

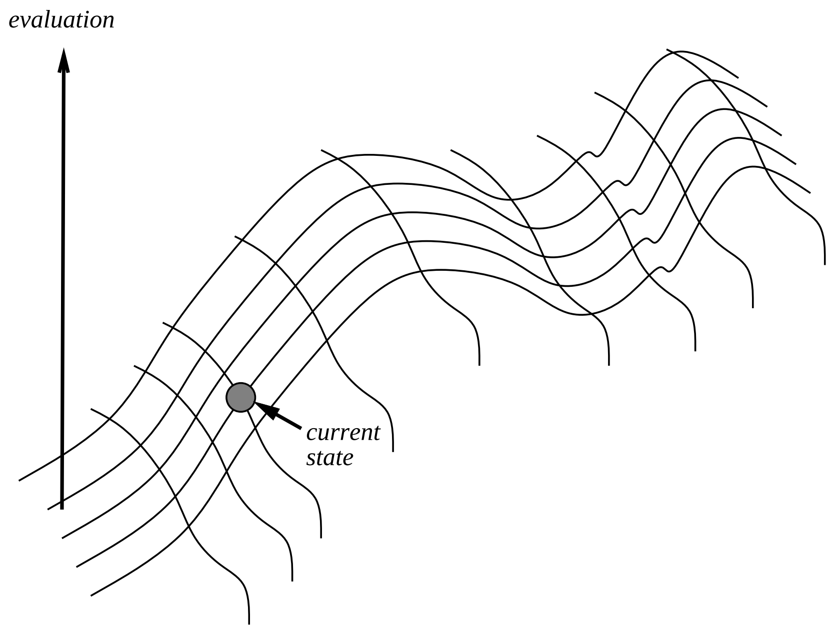
Main Idea: Start with a random initial configuration and make small, local changes to it that improve its quality

Local Search: The Landscape Metaphor

Ideally, the evaluation function h should be *monotonic*: the closer a state to an optimal goal state the better its h-value.

Each state can be seen as a point on a surface.

The search consists in moving on the surface, looking for its highest peaks (or, lowest valleys): the optimal solutions.

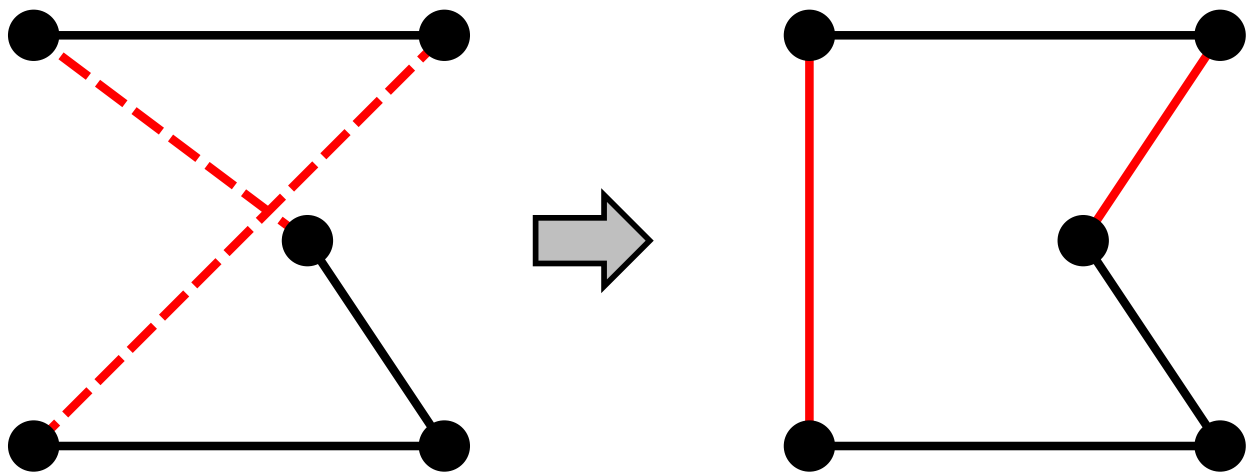


Local Search Example: TSP

TSP: Travelling Salesperson Problem

$h =$ length of the tour

Strategy: Start with any complete tour, perform pairwise exchanges

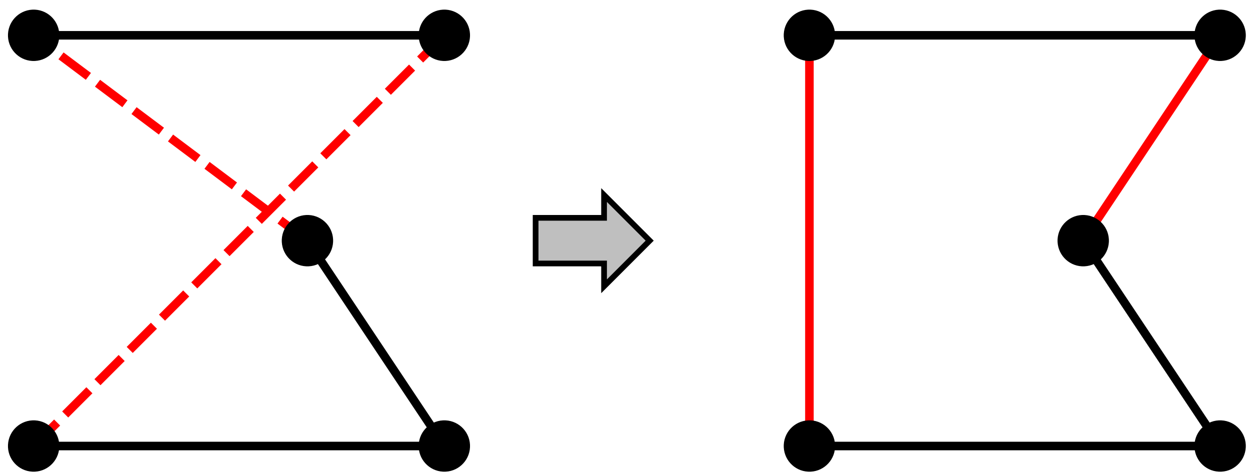


Local Search Example: TSP

TSP: Travelling Salesperson Problem

$h =$ length of the tour

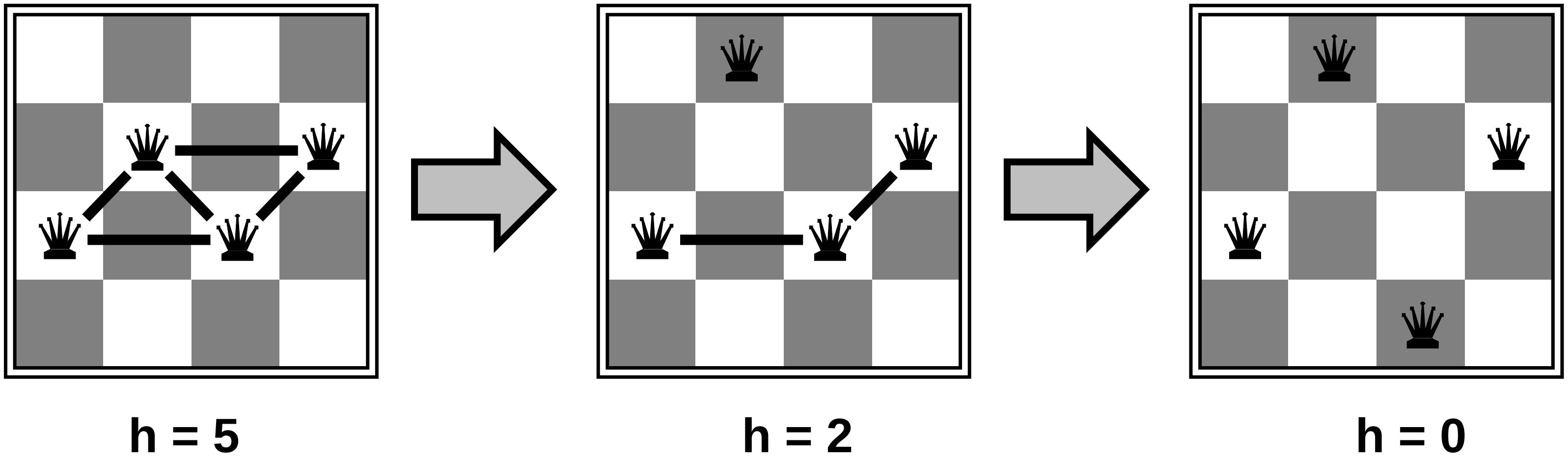
Strategy: Start with any complete tour, perform pairwise exchanges



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

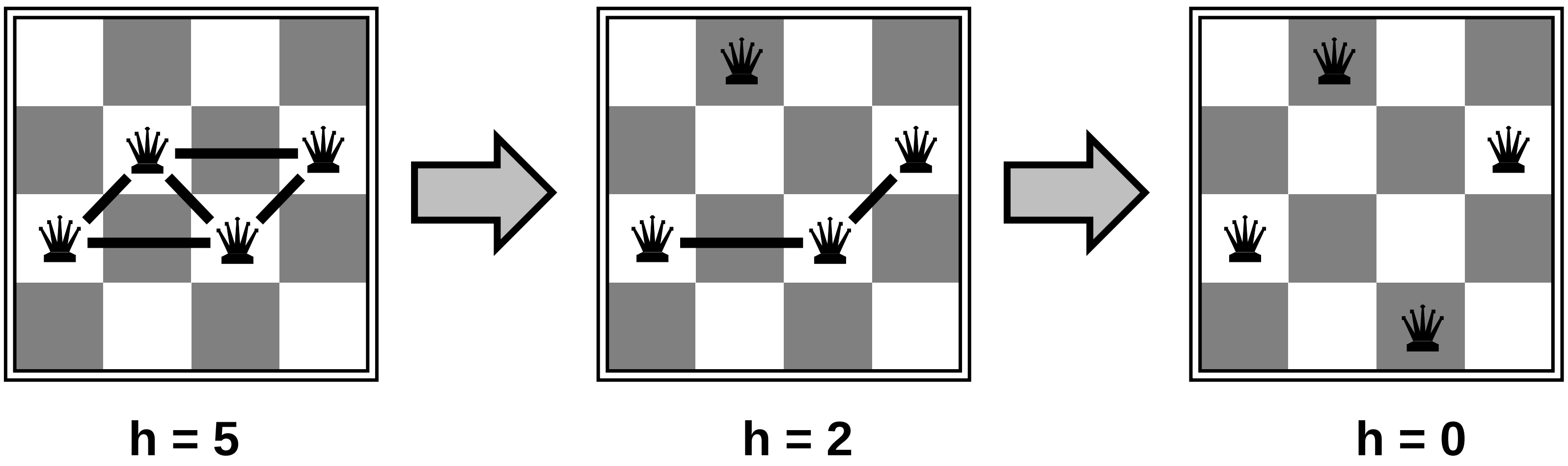
Local Search Example: n-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
- $h =$ number of conflicts
- Strategy: Move a queen to reduce number of conflicts



Local Search Example: n-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
- $h =$ number of conflicts
- Strategy: Move a queen to reduce number of conflicts



Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 10^6$

Hill-Climbing Search

Aka: gradient descent/ascent search

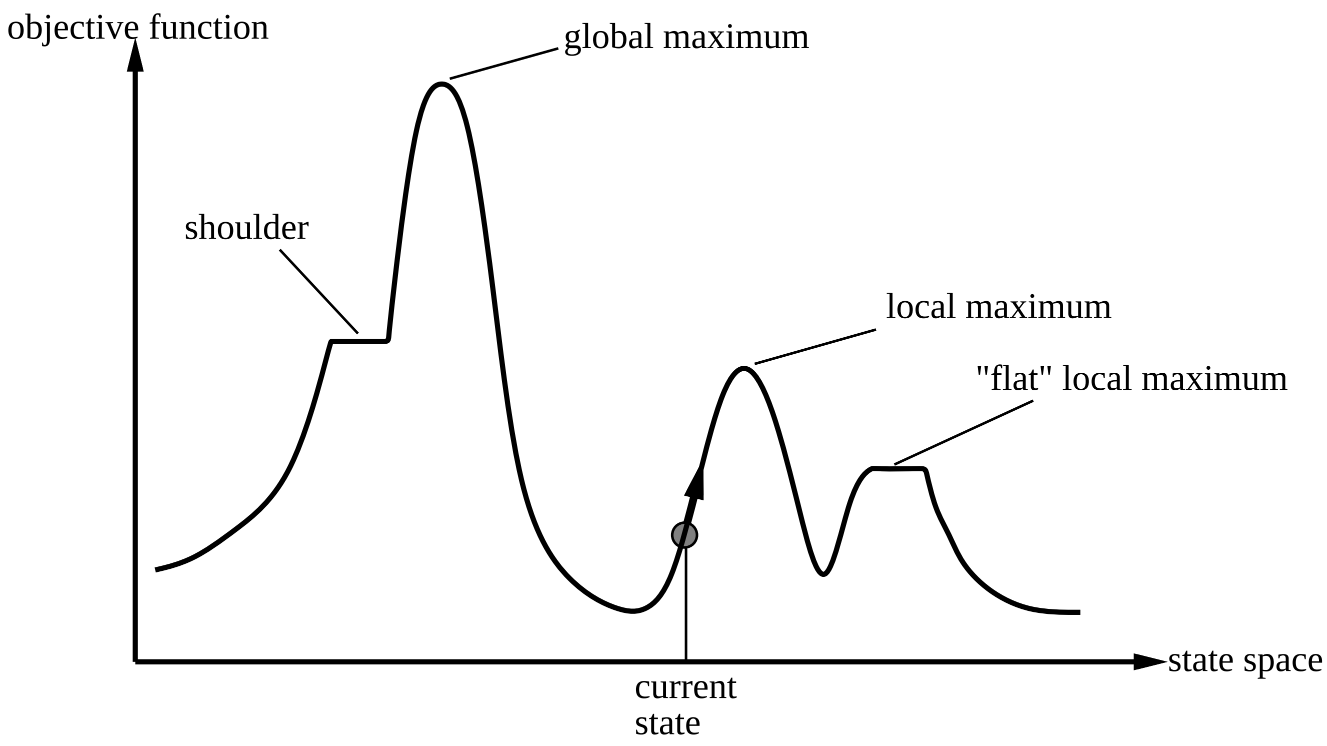
"Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local vars: 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
```

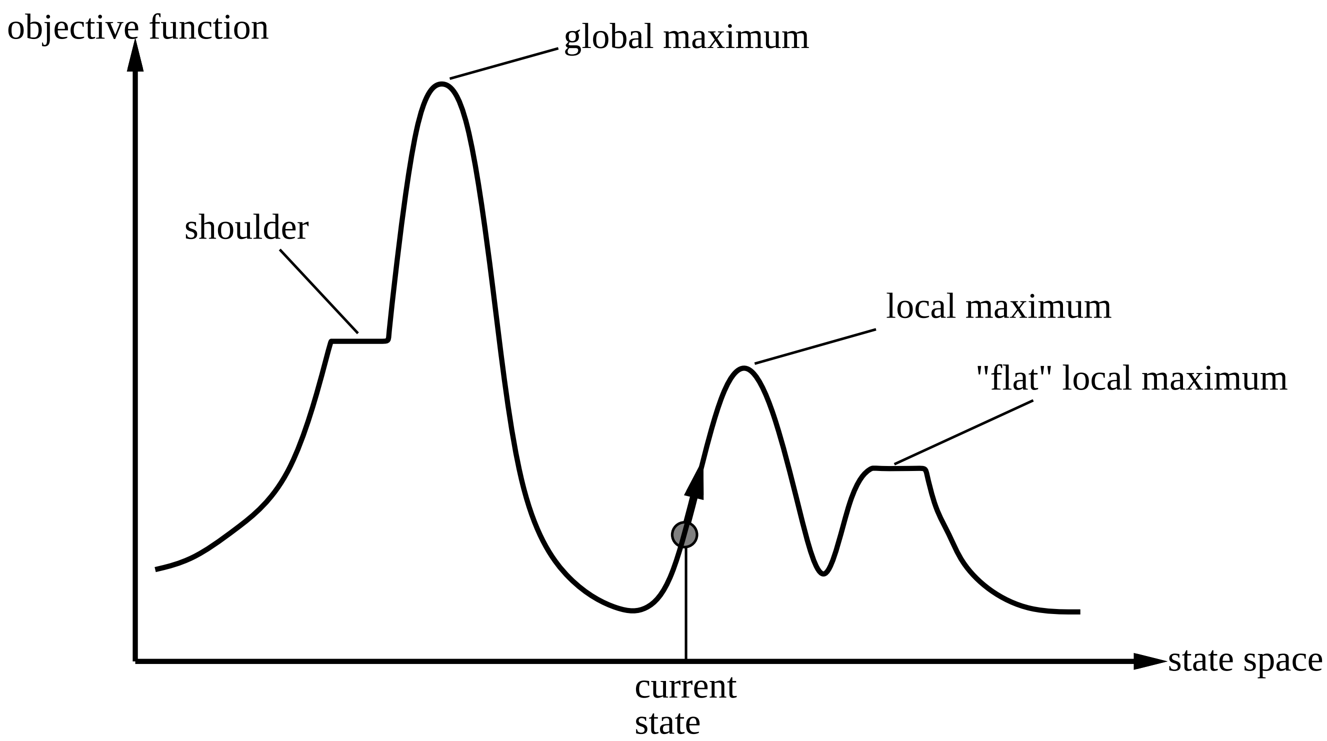
Hill-Climbing: Shortcomings

- Depending on the initial state, it can get stuck on local maxima
- It may converge very slowly
- In continuous spaces, choosing the step-size is non-obvious



Hill-Climbing: Shortcomings

- Depending on the initial state, it can get stuck on local maxima
- It may converge very slowly
- In continuous spaces, choosing the step-size is non-obvious



However, true local optima are surprisingly rare in high-dimensional spaces. There often is an escape to a better state

Hill Climbing: Improvements

Various possible alternatives:

- Restart from a random point in the space
- Look ahead: expand up to $m > 1$ generations of descendants before choosing best node
- Introduce *noise:* allow down-hill moves sometimes
- Keep $n > 1$ nodes in the fringe at each step

Simulated Annealing Search

Idea: improve hill-climbing by allowing occasional down-hill steps, to minimize the probability of getting stuck in a local maximum

Down-hill steps taken randomly but with probability that decreases with time

Probability controlled by a given *annealing* schedule for a *temperature* parameter T

If schedule lowers T slowly enough, search is guaranteed to end in a global maximum

Catch: it may take several tries with test problems to devise a good annealing schedule

Simulated Annealing Algorithm

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local vars: *current*, a node
    next, a node
    T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow$ **1 to** ∞ **do**
    $T \leftarrow schedule[t]$
    if $T = 0$ **or** IS-GOAL-STATE(*current*) **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $\left(\frac{1}{e}\right)^{\frac{|\Delta E|}{T}}$

Properties of Simulated Annealing

Note: $\left(\frac{1}{\mathrm{e}}\right)^{\frac{|\Delta E|}{T}}$ is directly proportional to T and inversely proportional to $|\Delta E|$

It can be proven that if T is decreased slowly enough, the search converges to the best state

This is not necessarily an interesting guarantee

Devised by Metropolis et al., 1953, for physical process modeling

Widely used in VLSI layout, airline scheduling, etc.

Local Beam Search

Idea: improve hill-climbing by keeping 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 maximum

Solution: choose k successors randomly, biased towards good ones

Genetic Algorithms

Inspired by Darwin's theory of natural selection

Each state is seen as an individual in a population

A genetic algorithm applies selection and reproduction operators to an initial population

The aim is to generate individuals that are most successful, according to a given fitness function

It is basically a stochastic local beam search, but with successors generated from pairs of states

Local maxima are avoided by giving a nonzero chance of reproduction to low-scoring individuals

The Basic Genetic Algorithm

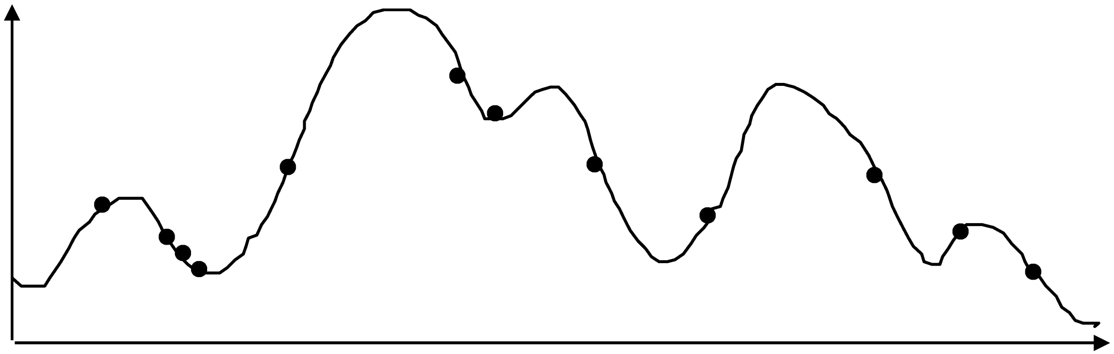
```
function GENETIC-ALGORITHM( population, FITNESS-FN) returns an individual
   inputs: population, a set of individuals
           FITNESS-FN, a function that measures the fitness of an individual

   repeat
       parents ← SELECTION( population, FITNESS-FN)
       population ← REPRODUCTION( parents)
   until some individual is fit enough
   return the best individual in population, according to FITNESS-FN
```

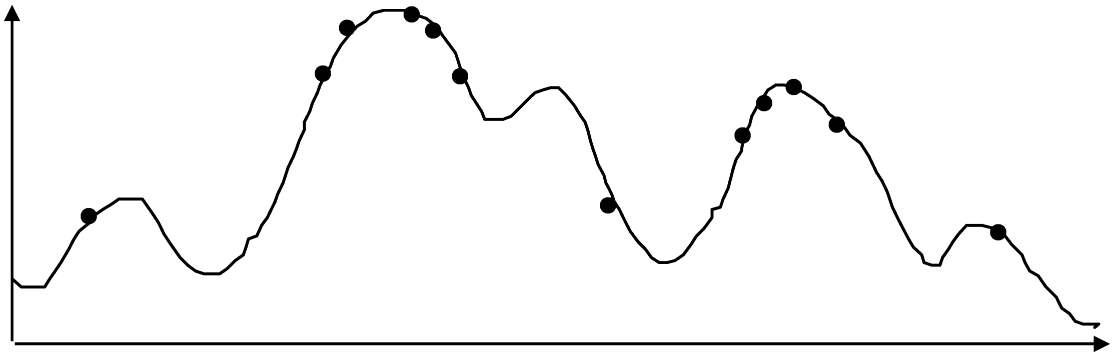
- Variation operators used in REPRODUCTION create the necessary diversity, facilitating novelty
- SELECTION reduces diversity but pushes quality by increasing fitness

Improving fitness

Abstract example: changes of values of fitness function



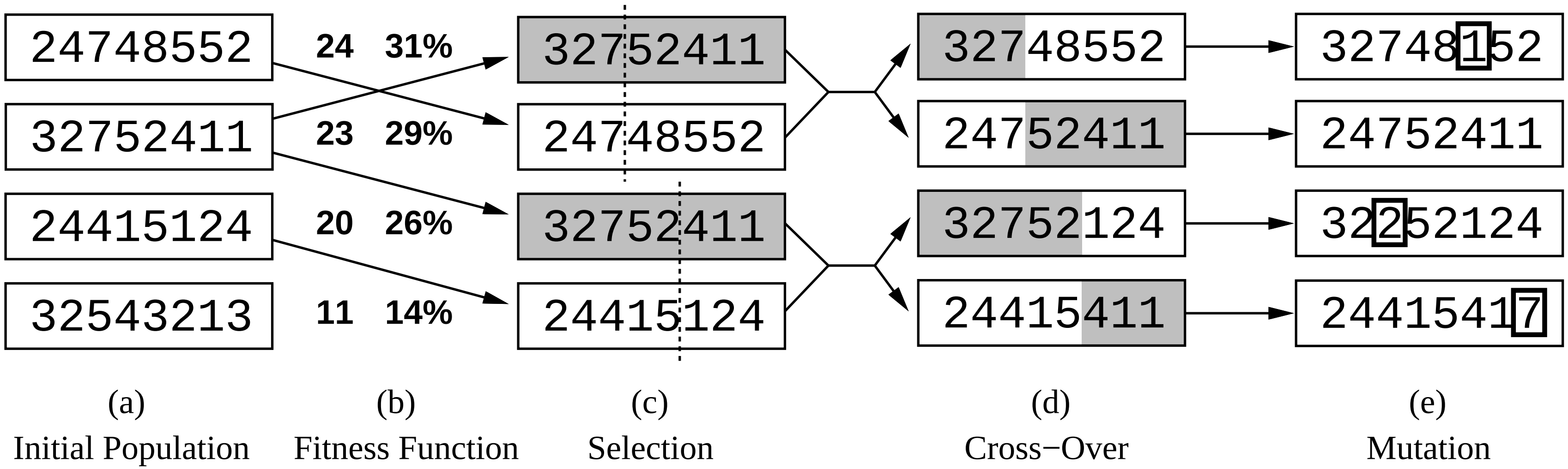
Distribution of Individuals in Generation 0



Distribution of Individuals in Generation N

Genetic Algorithms: Classic Approach

- Each state is represented by a finite string over a finite alphabet.
- Each character in the string is a gene.
- The selection mechanism is randomized, with the probability of selection proportional to the fitness measure.
- Reproduction is accomplished by crossover and mutation.



Possible Encodings

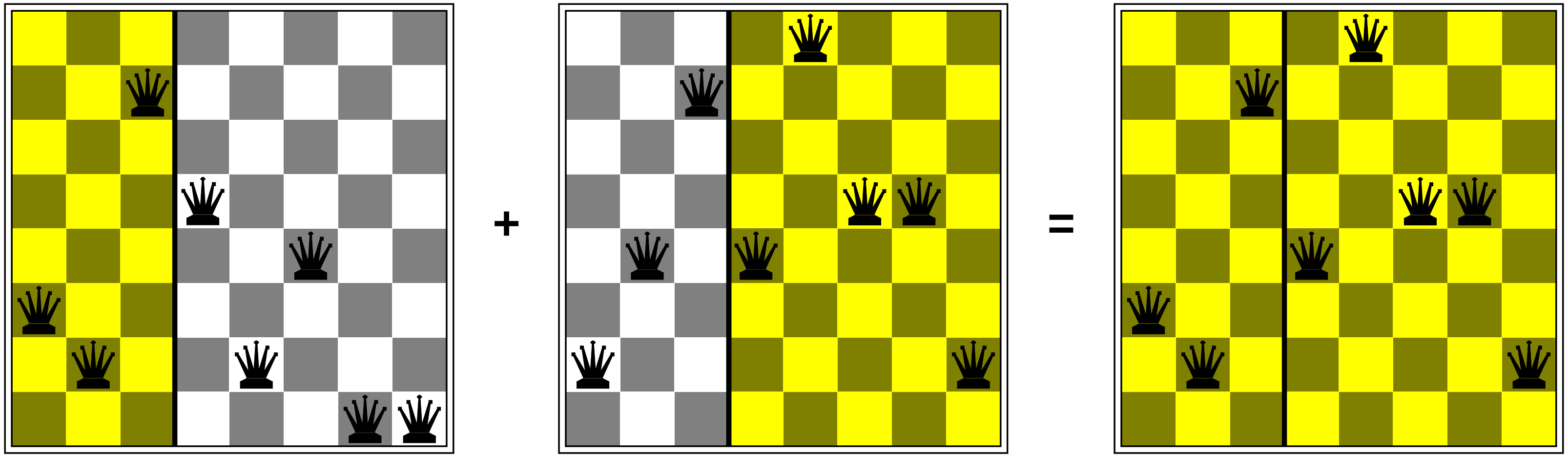
Character strings	0101 $\cdots$ 1100
Sequences of real numbers	(43.2 -33.1 $\cdots$ 0.0 89.2)
Tuples of elements	(E11 E3 E7 $\cdots$ E1 E15)
Lists of rules	(R1 R2 R3 $\cdots$ R22 R23)
Program elements	(genetic programming)

...

Encodings for Genetic Algorithms

Choosing the right encoding of state configurations to strings is crucial.

Crossover helps only if substrings are meaningful components that can be reassembled into a new meaningful configuration.



Note: GAs are not evolution: e.g., real genes encode replication machinery!

Problem Encoding

Problem: Find location that is closest to several given cities

Population encoding: Express location l as a 16-bit string

$$l = 1001010101011100$$

with the first 8 bits representing the location's X-coordinate and the second 8 bits representing the Y-coordinate

Fitness function: Median distance of location from each city

Combination: Crossover of X-coordinate from one parent and Y-coordinate from the other

Mutation: one or more bit flips

Problem Encoding

Problem: Finding the max value of some function $f : [0, 1)^n \rightarrow \mathbb{R}$

Population encoding: Vectors of size n with elements from $[0, 1)$

Combination: Various options

Mutation: randomly replace a value in the vector with one from $[0, 1)$

Discrete Recombination

Similar to crossover

Equal probability of receiving each parameter from either parent

Example:

$$(8, 12, 31, \ldots, 5) \quad (2, 5, 23, \ldots, 14)$$

$$\Downarrow$$

$$(2, 12, 31, \ldots, 14)$$

Intermediate Recombination

Each child component is the average of the corresponding parent components

Example:

$$(8, 12, 31, \ldots, 5) \quad (2, 5, 23, \ldots, 14)$$

$$\Downarrow$$

$$(5, 8.5, 27, \ldots, 9.5)$$

GA for the Traveling Salesperson Probl.

Problem: Find a tour of a given set of cities so that
each city is visited only once and
total traveled is minimal

Representation: An ordered list of city numbers
(known as order-based GA)

1) London	3) Iowa City	5) Beijing	7) Tokyo
2) Venice	4) Singapore	6) Phoenix	8) Victoria

Ex.,

CityList1 (3 5 7 2 1 6 4 8)
CityList2 (2 5 7 6 8 1 3 4)

GA for the Traveling Salesperson Probl.

Combination: Order 1 crossover (combines inversion and recombination):

1. Copy a randomly selected portion of Parent 1 to Child
2. Fill the blanks in Child with those numbers in Parent 2 from left to right, avoiding duplicates in Child

Parent 1	(3 5 7 2 1 6 4 8)
Parent 2	(2 5 7 6 8 1 3 4)
Child	(5 8 7 2 1 6 3 4)

GA for the Traveling Salesperson Probl.

Mutation: swap two numbers in the list

Before: (5 8 7 2 1 6 3 4)

After: (5 8 6 2 1 7 3 4)