Lecture 5: Evolutionary Computation

Artificial Intelligence
CS-UY-4613-A / CS-GY-6613-I

Julian Togelius
julian.togelius@nyu.edu

Table of contents

- Recap: hill-climbing, search as optimization
- Biological metaphor
- Recombination, mutation, selection
- Representations
- Fitness functions
- Constraints
- Multiple objectives
- Coevolution

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms: keep a single "current" state, try to improve it

Hill-climbing

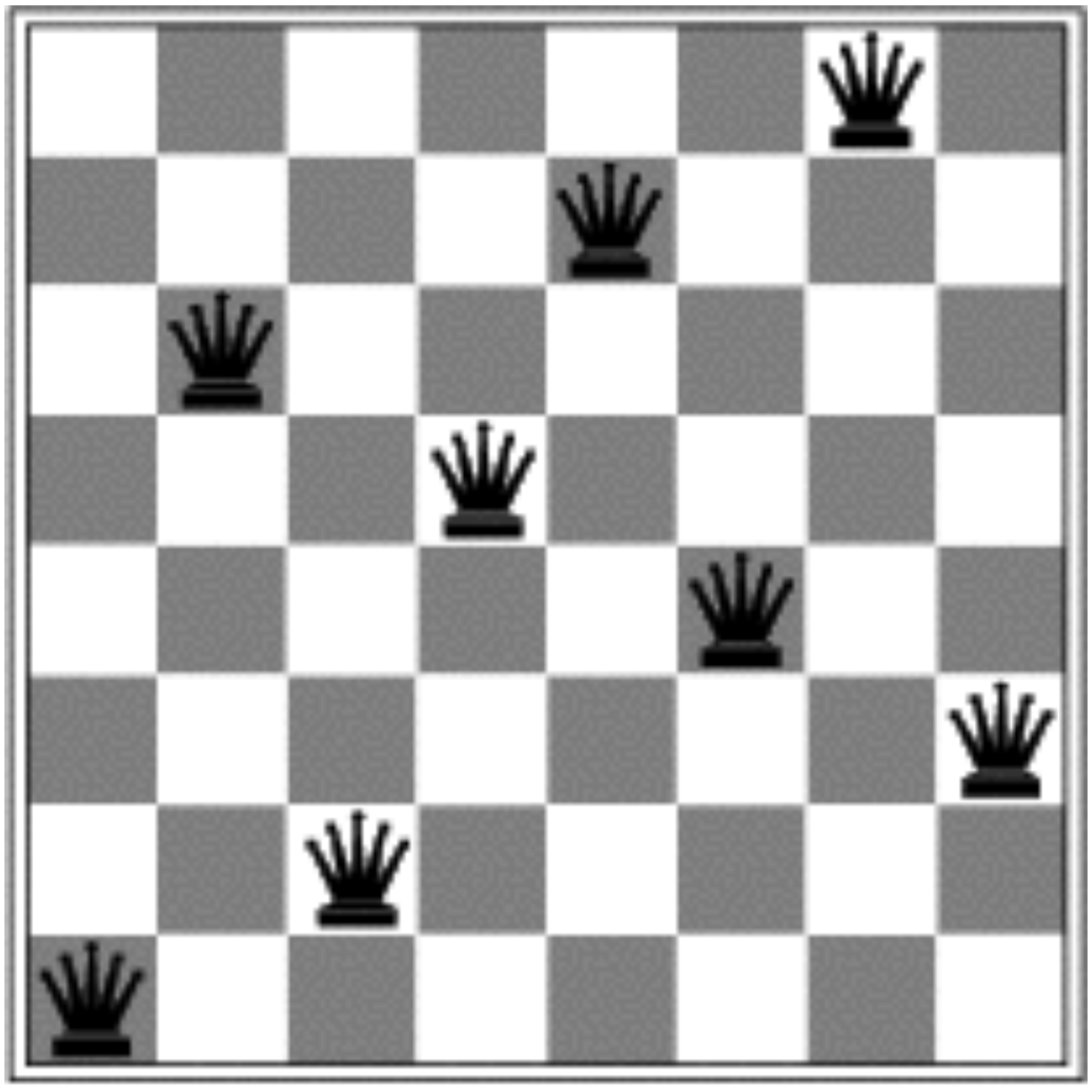
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

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
```

n-queens

- Put n queens on an n × n board with no two queens on the same row, column, or diagonal

Local minimum



Other ideas…

- Exhaustive search: try all configurations, one after another

- Random search: generate random configurations

Simulated annealing

- Do bad moves with decreasing probability

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *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 $T = 0$ **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 $e^{\Delta\ E/T}$

Local beam search

- Keep track of k states rather than just one
- Start with k randomly generated states
- At each iteration, all the successors of all k states are generated
- If any one is a goal state, stop; else select the k best successors from the complete list and repeat.

Limitations of local search

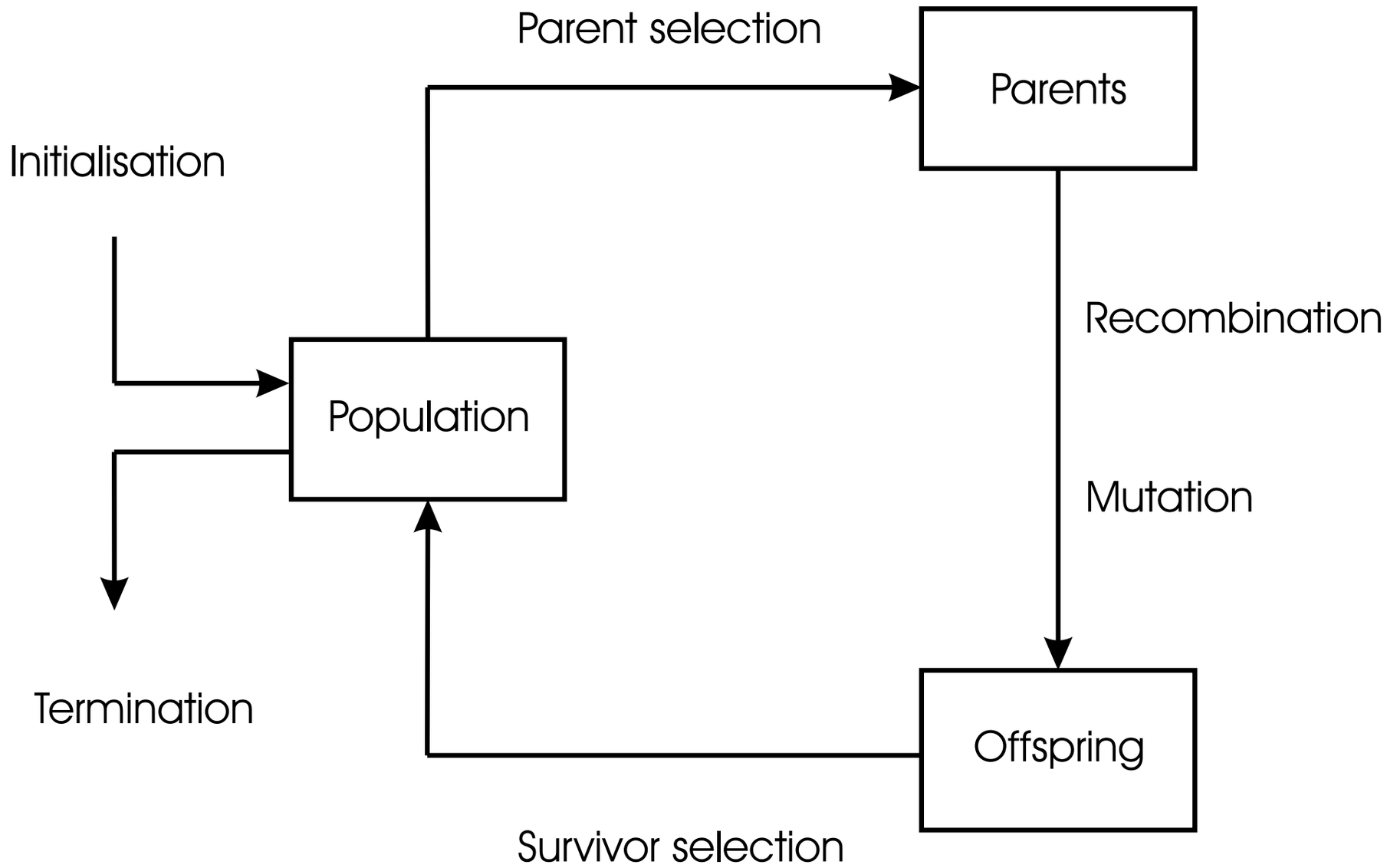
- Susceptible to local optima
- No sharing of information between parallel hill-climbing runs
- No possibility to learn from previous search experience

Evolutionary algorithms

- Stochastic global optimisation algorithms
- Inspired by Darwinian natural evolution
- Extremely domain-general, widely used in practice


© Reuters

Generic EA


Parent selection
Parents
Initialisation
Recombination
Population
Mutation
Termination
Offspring
Survivor selection

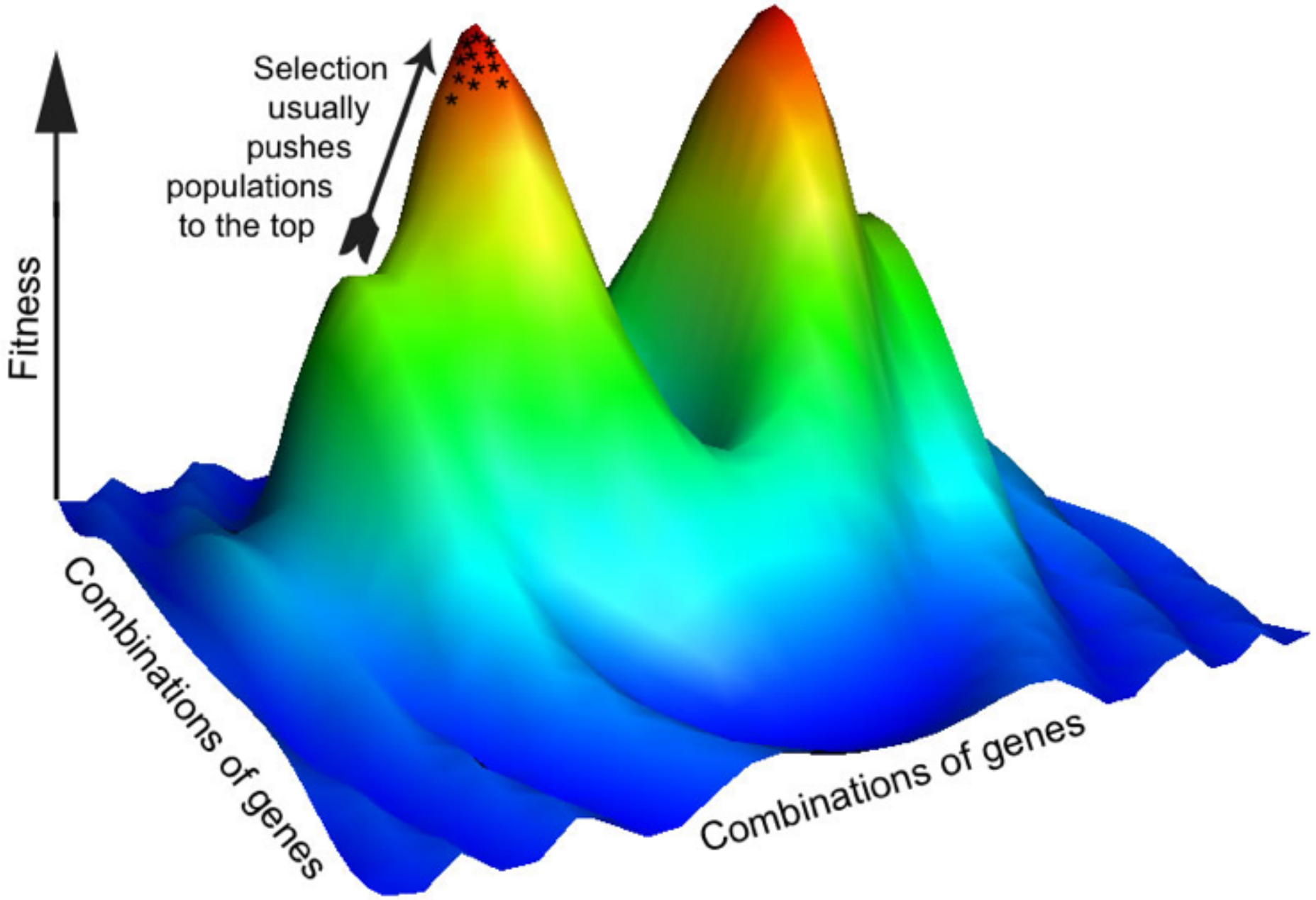
What's essential?

- Fitness evaluation
- Selection
    - Fitness influences how many offspring an individual has
- Variation
    - Could be mutation *and/or* crossover

Generic EA

```
BEGIN
   INITIALISE population with random candidate solutions;
   EVALUATE each candidate;
   REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
      1 SELECT parents;
      2 RECOMBINE pairs of parents;
      3 MUTATE the resulting offspring;
      4 EVALUATE new candidates;
      5 SELECT individuals for the next generation;
   OD
END
```

The fitness landscape



Simple μ+λ Evolution Strategy

- Create a population of μ+λ individuals
- Each generation
  - Evaluate all individuals in the population
  - Sort by fitness
  - Remove the worst λ individuals
  - Replace with mutated copies of the μ best

Example: OneMax

- Representation: binary strings of length 5
- Examples: 00000, 01001, 00100, 11101, 11111
- Fitness function: count the number of ones
- Mutation: flip a random bit

Example: OneMax

- Initial population: 00100, 11001, 10010, 01010
- Gen 1: 11001 : 3, 10010 : 2, 01010 : 2, 00100 : 1
  Selection: 11001 : 3, 10010 : 2
- Gen 2: 11101 : 4, 11001 : 3, 10010 : 2, 00010 : 1
  Selection: 11101 : 4, 11001 : 3
- Gen 3: 11111 : 5, 11101 : 4, 11101 : 4, 11001 : 3

Generally, you need

- A solution representation
- Variation operators (mutation and/or crossover)
- A fitness (evaluation) function

Representation

- Individuals in the population are referred to as genotypes (e.g. arrays of doubles)
    - All variation and replication mechanisms are performed on the genotype
- What is evaluated by the fitness function is the phenotype (e.g. neural networks)
- Genotype-to-phenotype mapping should be fast, deterministic, preserve locality

Representations should be...

- Compact
- Expressive (represent all “interesting” parts of phenotype space)
- Human-understandable (?)
- Local (?)

Evaluation function

- Also called fitness function
- Returns a number (or ranking) of the evaluated individual (phenotype)
- You could have one or many
- Evaluation functions should be smooth, fast to evaluate, noise-free, accurate and relevant

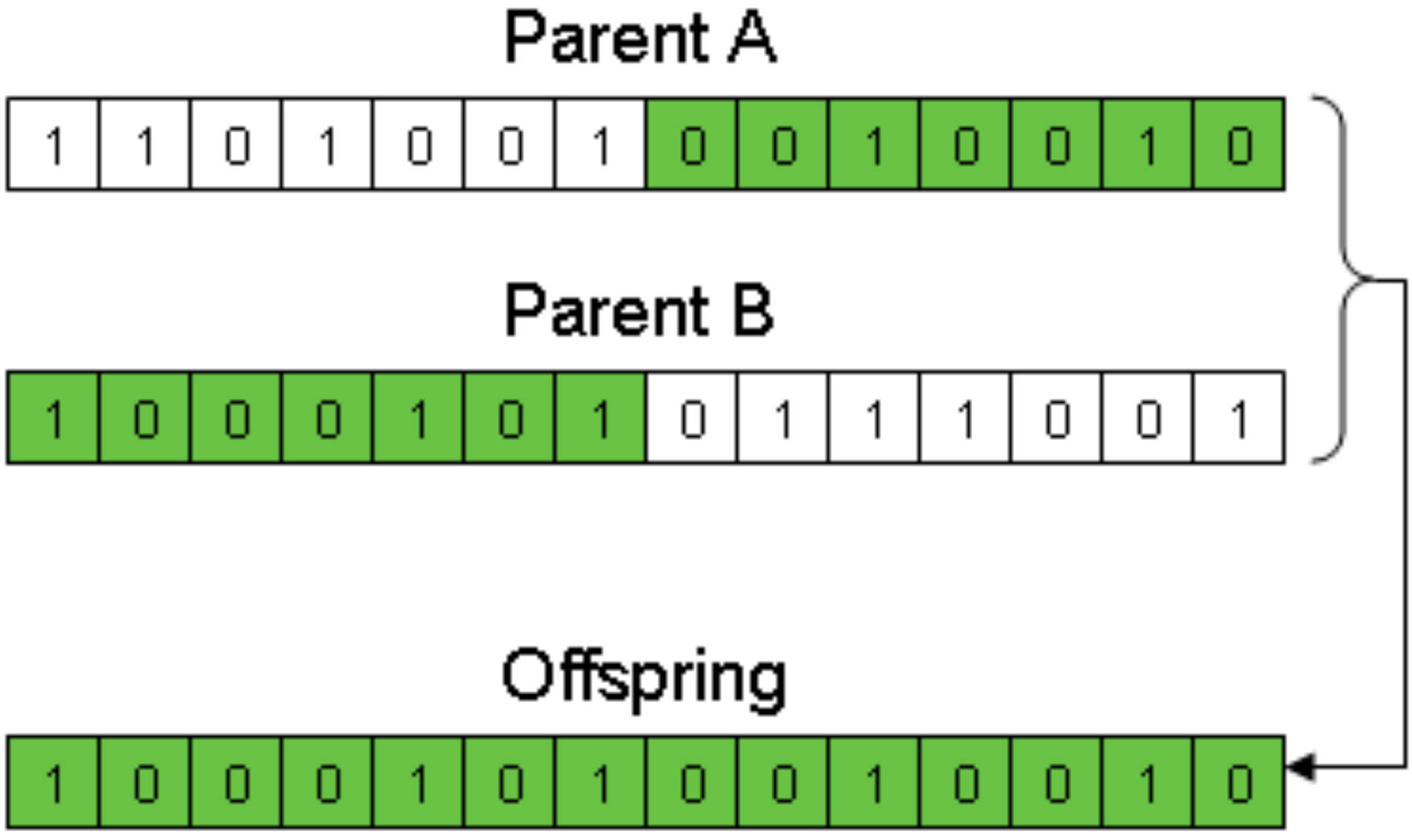
Parent selection

- Which individuals get to reproduce?
- Can choose this based on fitness...
- ...or fitness ranking
- Deterministic or stochastic
- How many to select?

Variation operators

- Should be appropriate to the representation
- Mutation
    - Should have the appropriate magnitude
        - e.g. Gaussian mutation: add values drawn from Gaussian distribution to an array of real numbers. Gaussian distribution is a random distribution with a mean and a standard deviation. e.g. in Java: random.nextGaussian ()
        - Or: add a value to all parts of the genome drawn from uniform distribution. e.g. Math.random () * some small value
- Recombination (crossover)
    - Only do this if you know what you are doing

Example: one-point crossover



Survivor selection

- Elitism?
  - Usually a good thing
- Who to replace?
  - Always kill the worst?
  - Diversity preservation?

Initialization

- Population should be diverse enough

- Should you start from results of previous evolutionary runs?

Termination condition

- When you're done?
- When you've waited long enough?
- Relevant factor to count: number of evaluations

Simple μ+λ ES (concrete)

- Create a population of μ+λ individuals (e.g. 50+50) represented as e.g. vectors of real numbers in [0..1]

- Each generation (until 100 generations)
  - Evaluate all individuals in the population
  - Sort by fitness (e.g. win rate or score for the game; higher is better)
  - Remove the worst λ individuals
  - Replace with mutated copies of the μ best (mutate through Gaussian mutation with mean 0 and s.d. 0.1)

Types of evolutionary algorithms

- Genetic Algorithms
  Representation: bitstring
  Mostly driven by crossover

- Evolution Strategies
  Representation: real values
  Mostly driven by (self-adaptive) mutation

- Genetic Programming
  Representation: expression trees

- Estimation of Distribution Algorithms, e.g. CMA-ES

- Others: Particle Swarm Optimization, Differential Evolution…

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 **??**, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

Taking a step back…
How would you actually use an evolutionary algorithm?

Uses of evolution:

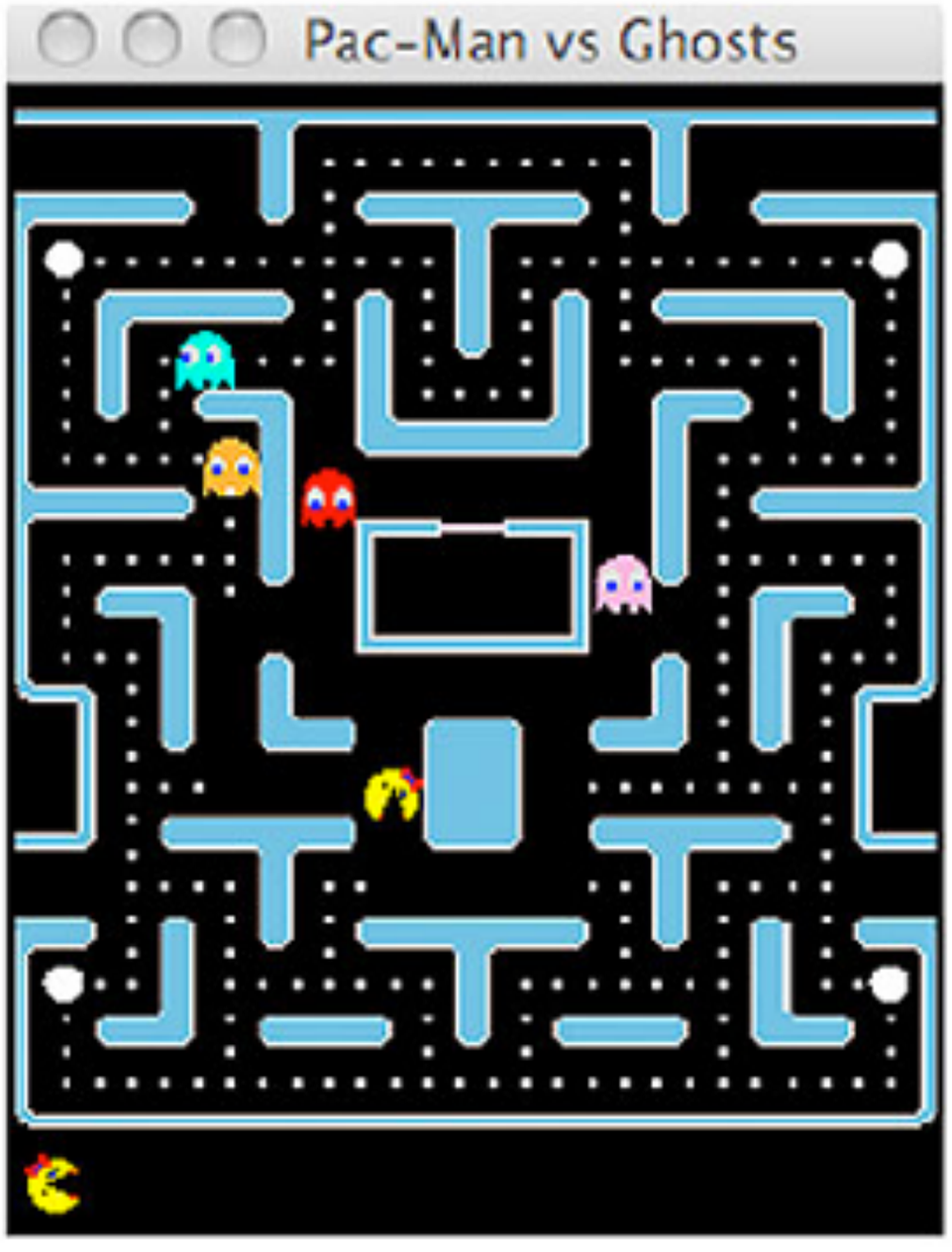
- Evolve a route
- Evolve the parameters of the car
- Evolve a controller that can drive the car


MARIO: 1024
COINS
18
DIFFICULTY
10
TIME
144
WorldPause
false

Uses of evolution:

- Evolve level, heuristic (for search algorithm), search algorithm, path, physics, physics prediction, enemies….


Pac-Man vs Ghosts


Pac-Man vs Ghosts

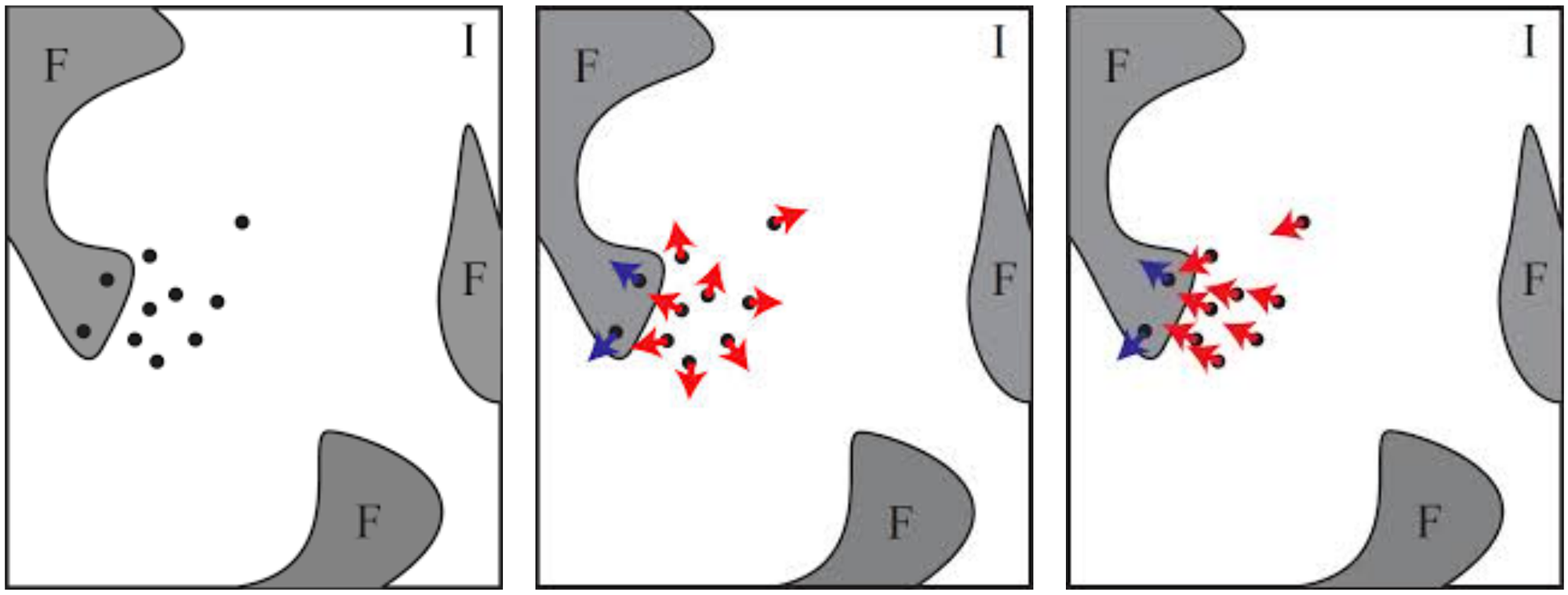
Uses of evolution:

Uses of evolution:

Evolution with constraints

- Sometimes there are certain features which *must* be part of the solution
  - e.g. car has four wheels, path passes through New Jersey, wing length less than 40m
- These could be encoded in the fitness function - but gets complicated
- Better, encode them as constraints

Feasible and Infeasible Areas



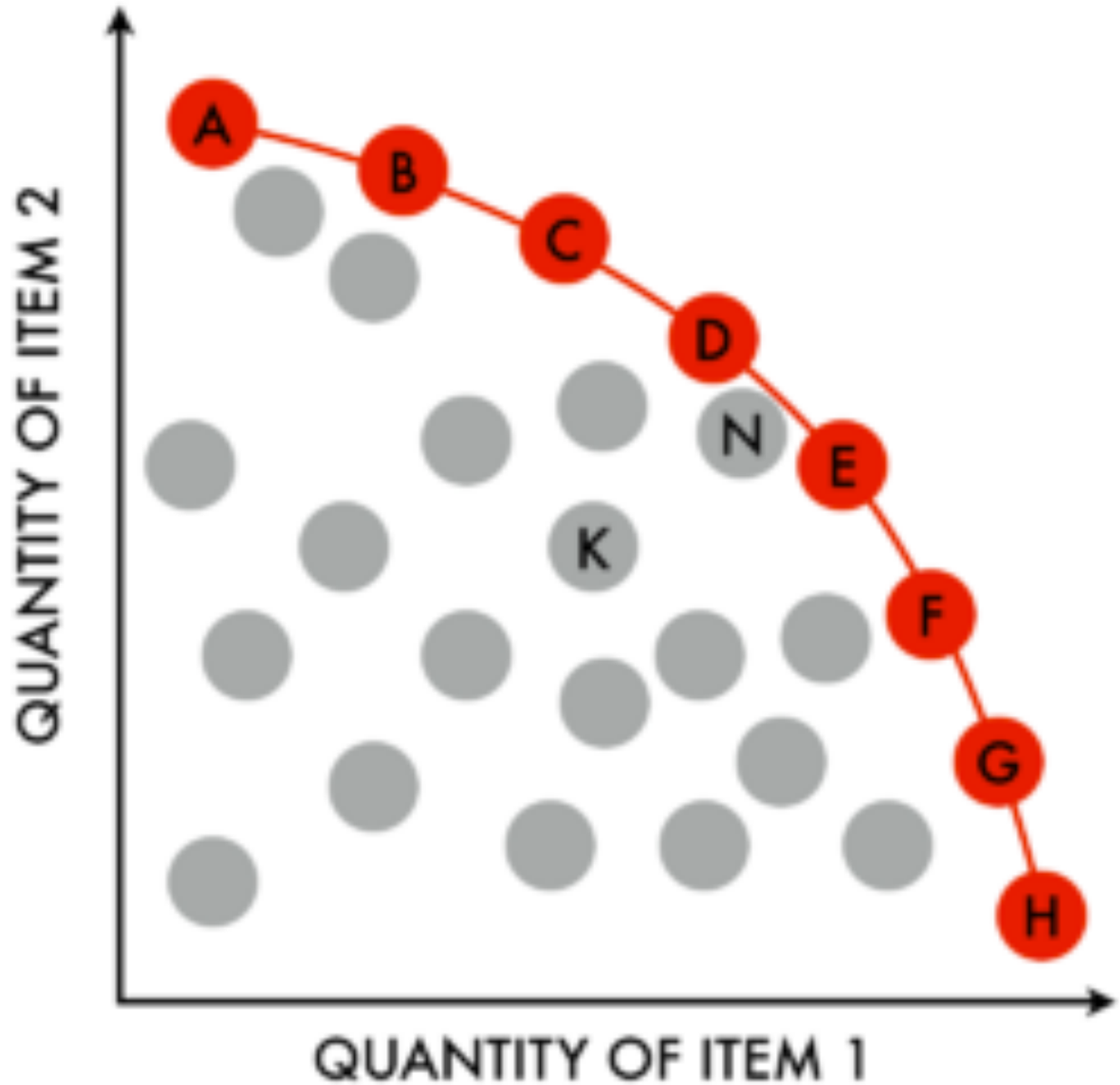
FI-2Pop algorithm

- Keep two populations: feasible and infeasible individuals
- Feasible individuals are selected for their fitness
- Infeasible individuals are selected for their proximity to feasibility
- Infeasible offspring of feasible parents are moved to infeasible population

Multiobjective evolution

- Often, you want to optimize for multiple fitness dimensions
    - For example, you want the car to be fuel-efficient, cheap and fast…
- These objectives are often *partially conflicting*
- A multiobjective evolutionary algorithm finds the *pareto front* of solutions given these objectives

Pareto fronts




© Reuters

Competitive coevolution

- Two (or more populations) with different fitness functions
  - e.g. predators and prey
- The fitness of one population depends on the other and vice versa
- Goal: start an *arms race*