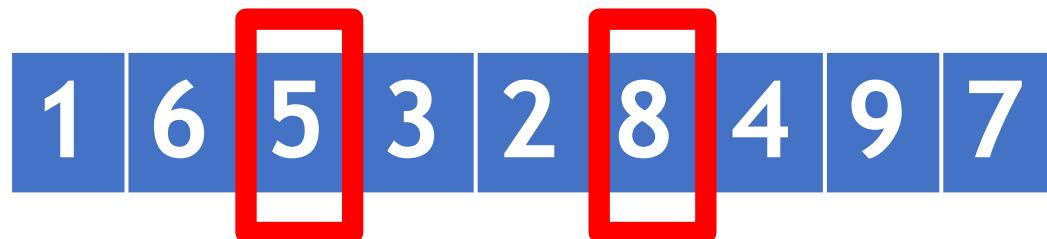


Metaheuristic Optimization

Genetic Algorithms

Dr. Diarmuid Grimes

Recap: How does the mutation operator work?



Reciprocal Exchange

1 6 8 3 2 5 4 9 7

Recap: How does the mutation operator work?



Inversion Operator

1 6 8 2 3 5 4 9 7

Mutation for permutation problems

- Performed per **chromosome** rather than per gene
- Mutation rate = 0.1 => 1 in 10 children will be mutated on average

```
def mutation(self, ind):
    """
    Mutate an individual by swaping two cities with certain probability
    """
    if random.random() > self.mutationRate:
        return
    indexA = random.randint(0, self.genSize-1)
    indexB = random.randint(0, self.genSize-1)

    tmp = ind.genes[indexA]
    ind.genes[indexA] = ind.genes[indexB]
    ind.genes[indexB] = tmp

    ind.computeFitness()
    self.updateBest(ind)
```

Recap: Selection



- Main idea is to make copies of solutions that perform better at the expense of solutions that perform worse
- Two major classes of methods
 - Proportionate
 - Ordinal

Recap: Proportionate



- Sum fitness values across chromosomes to get total fitness
- Probability of selection for a chromosome is its fitness / total fitness

$$p_{select}(X) = \frac{f(X)}{\sum_{Y \in P(t)} f(Y)}$$

- For minimization?
 - Transform fitness first, e.g. 1/fitness or Max-fitness

Minimisation Example



Number of Individual	1	2	3	4	Total
Fitness	5	12	5	3	25
1/Fitness	0.2	0.08	0.2	0.33	0.81
Max_fit - Fitness + 1	8	1	8	10	27
Selection Prob (1/F)	0.25	0.1	0.25	0.4	1
Selection Prob (F-Max)	0.3	0.04	0.3	0.36	1

C1

$0 < x \leq 0.25$

$0 < x \leq 0.3$

C2

$0.25 < x \leq 0.35$

$0.3 < x \leq 0.34$

C3

$0.35 < x \leq 0.6$

$0.34 < x \leq 0.64$

C4

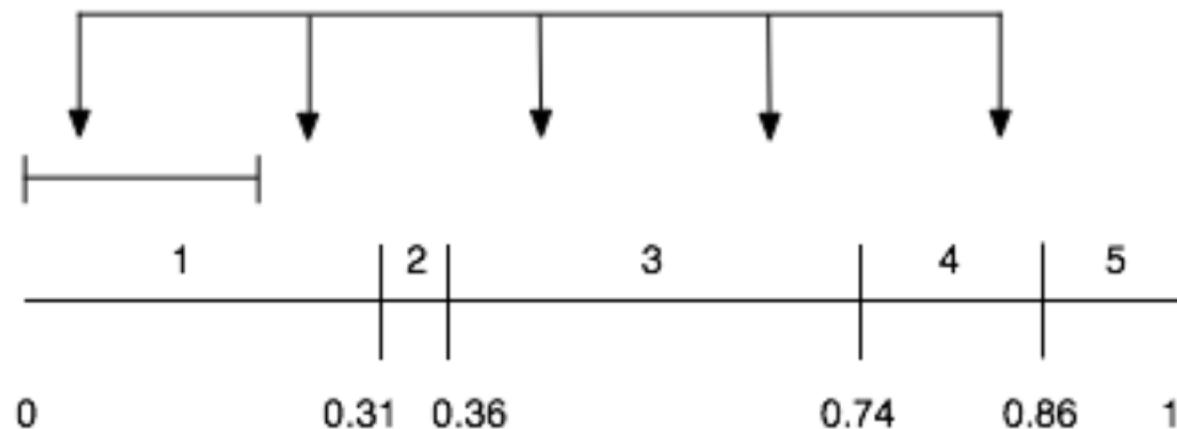
$0.6 < x \leq 1$

$0.64 < x \leq 1$

Recap: Stochastic Universal Sampling (SUS)



- Make N equally spaced marks
- Generate a uniform random number in $[0, 1/N]$. That number is the position of the 1st mark.
- (Equivalent to spinning the equally spaced marks on top of the roulette. Just need to spin it once)



Recap: Ordinal-base methods



- Probability of selecting an individual depends on its relative order (or ranking) compared with other members
- Two commonly used methods:
 - Tournament selection
 - Truncation selection

Recap: Proportionate vs. ordinal-based methods



- Fitness proportionate-based methods have 2 major drawbacks:
 - Super individual can take over the population too quickly
 - Search tends to stall when all the individuals have about the same fitness
- Ordinal-based methods are often preferred
 - Enable a sustained pressure towards better solutions
 - Can control selection pressure easily



Components of a GA

- Initialization
 - Usually at random, but can use prior knowledge
- Selection
 - Give preference to better solutions
- Variation
 - Create new solutions through crossover and mutation
- Replacement
 - Combine original population with newly created solutions

- Newly created solutions some (or all) of the current solutions
- Delete-all or full replacement
 - All previous solutions are deleted
- Steady-state replacement
 - Some of the old solutions become part of the new population
 - Extreme - only one child added per generation, e.g. replacing worst solution

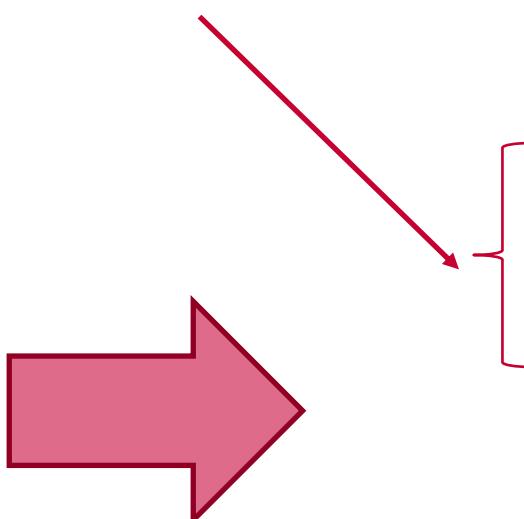
- Different strategies can be used to choose solutions to be replaced.
 - E.g. Replace worst x% of new generation with best x% of previous if fitness of old better than new
 - Need to be careful of premature convergence to good solution, “bad” solutions can be useful for diversification
- Whenever there is the guarantee that the best solution never dies, GAs are said to be ***elitist***
- Percentage of replaced individuals is called the ***generation gap***.

Top M candidates survive → No crossover or mutation for those candidates

Best Candidates
Or Elite Candidates

Generation at time t	
1	
2	
P-1	
P	

Current Generation



Generation at time t+1	
1	
2	
P-1	
P	

New Generation

Sorting required?

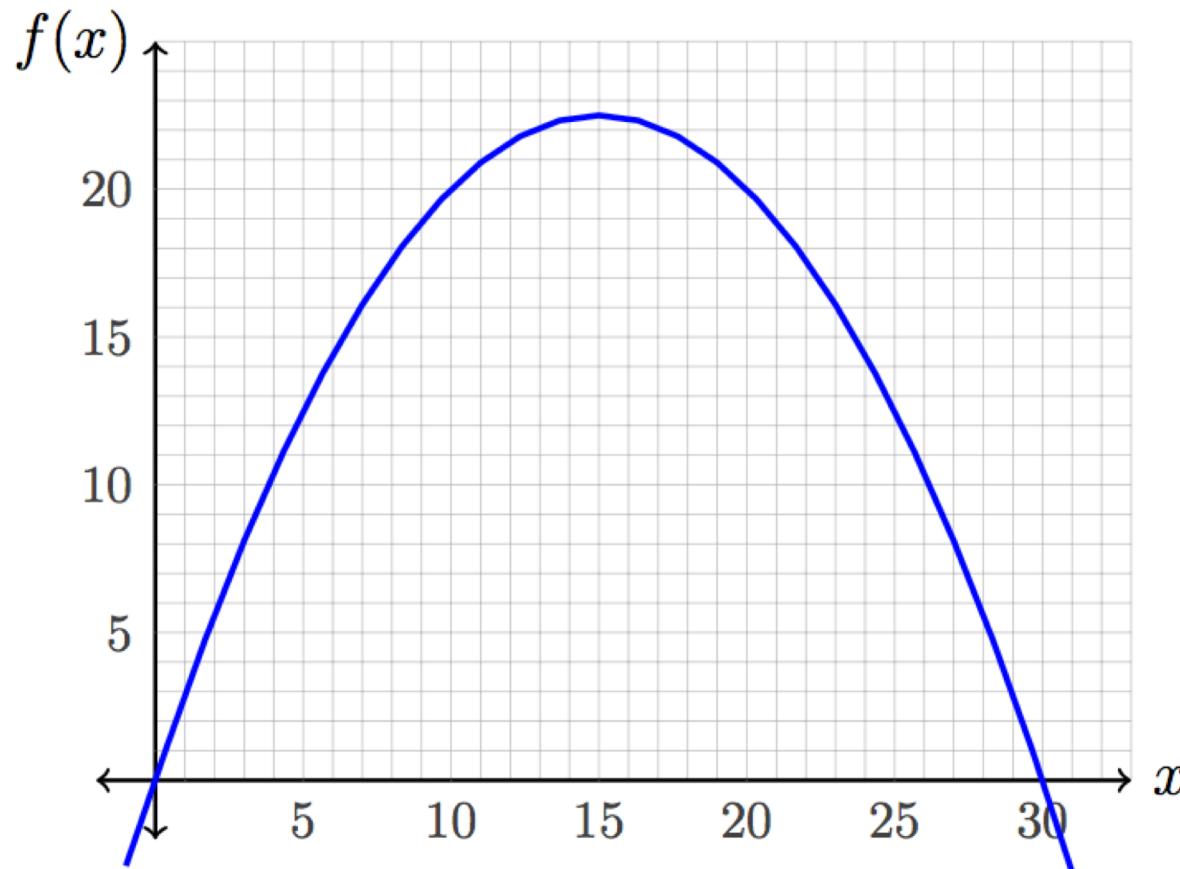
Possible method:

- Store best x solutions from previous generation
- We compute the fitness of each child after generation
- We can keep top_x and bottom_x lists, update if required when computing fitness
 - And single fitness values. E.g., for $x=5$, could also maintain fitness value of 5th best and 5th worst
 - Check if fitness of new child is better than 5th best, or worse than 5th worst, or neither. Replace where necessary.
 - Compare best x solutions from previous with worst x from new generation, keep x best chromosome in the generation
 - Update top x where appropriate with previous top x .

Example: Optimize a math function



x is allowed to vary between 0 and 31



00000

11111

$$f(x) = \frac{-x^2}{10} + 3x$$

Representation



Example

- Initial population size: 10 chromosomes (at random)

$$f(x) = \frac{-x^2}{10} + 3x$$

Chromosome Number	Initial Population	x Value	Fitness Value $f(x)$	Selection Probability
1	01011	11	20.9	0.1416
2	11010	26	10.4	0.0705
3	00010	2	5.6	0.0379
4	01110	14	22.4	0.1518
5	01100	12	21.6	0.1463
6	11110	30	0	0
7	10110	22	17.6	0.1192
8	01001	9	18.9	0.1280
9	00011	3	8.1	0.0549
10	10001	17	22.1	0.1497

Initial
Population

Sum 147.6
Average 14.76
Max 22.4

Best chromosome so far

Selection



- Random selection

Selection

- Since our population has 10 chromosomes and each ‘mating’ produces 2 offspring
- We need 5 mating to produce a new generation of 10 chromosomes

Chromosome Number	Mating Pairs	New Population	x Value	Fitness Value $f(x)$
5	01 100			
2	11 010			
4	0111 0			
8	0100 1			
9	0001 1			
2	1101 0			
7	10110			
4	01110			
10	100 01			
8	010 01			

Crossover and Mutation

Mutation

Chromosome Number	Mating Pairs	New Population	x Value	Fitness Value $f(x)$
5	01 100	01010	10	20
2	11 010	11100	28	5.6
4	0111 0	01111	15	22.5
8	0100 1	01000	8	17.6
9	0001 1	01010	10	20
2	1101 0	11011	27	8.1
7	10110	10110	22	17.6
4	01110	01110	14	22.4
10	100 01	10001	17	22.1
8	010 01	01001	9	18.9

Sum: 174.9

Average: 17.48

Max: 22.5

Mutation



- Let's say that each bit of the new chromosomes mutates with a low probability, i.e., 0.001
- With 50 total transferred bit position, we expect $50 \times 0.001 = 0.05$ bits to mutate
- It is unlikely that no bits mutate in the second generation

```
class Individual:
    def __init__(self, _size):
        self.fitness = 0
        self.genes = []
        self.genSize = _size
        for i in range(0, self.genSize):
            self.genes.append(random.randint(0, 1))
```

```
def computeFitness(self):
    self.fitness = 0
    x = self.binaryToInt()
    self.fitness = -(x**2)/10.0 + 3.0*x
```

```
def copy(self):
    ind = Individual(self.genSize)
    for i in range(0, self.genSize):
        ind.genes[i] = self.genes[i]
    return ind
```

Chromosome Or Individual

Genes
Boolean variables 1/0

$$f(x) = \frac{-x^2}{10} + 3x$$

Random Initial values

Objective function

class GA:

```
def __init__(self, _popSize, _genSize, _mutationRate, _maxIterations):
    self.population = []
    self.matingPool = []
    self.best = None
    self.popSize = _popSize
    self.genSize = _genSize
    self.mutationRate = _mutationRate
    self.maxIterations = _maxIterations
    self.iteration = 0
    self.initPopulation()
```

```
def initPopulation(self):
    for i in range(0, self.popSize):
        individual = Individual(self.genSize)
        individual.computeFitness()
        self.population.append(individual)
    self.best = self.population[0]
```

Parameters
&
Initial values

```
def randomSelection(self):
```

```
    indA = self.matingPool[ random.randint(0, self.popSize-1) ]  
    indB = self.matingPool[ random.randint(0, self.popSize-1) ]  
    return [indA, indB]
```

Selection

```
def crossover(self, indA, indB):
```

```
    child      = Individual(self.genSize)  
    midPoint   = random.randint(0, self.genSize)  
    for i in range(0, self.genSize):  
        if( i > midPoint ):  
            child.genes[i] = indA.genes[i]  
        else:  
            child.genes[i] = indB.genes[i]  
    child.computeFitness()  
    self.updateBest(child)  
    return child
```

Mid-Point crossover

```
def mutation(self, candidate):
```

```
    for gene_index in range(0, self.genSize):  
        if(random.random() < self.mutationRate):  
            candidate.flipGene(gene_index)  
    candidate.computeFitness()  
    self.updateBest(candidate)
```

Mutation

```
def GAStep(self):  
    for ind_i in self.population:  
        ind_i.computeFitness()  
    self.updateMatingPool()  
    self.newGeneration()
```

```
def newGeneration(self):  
    for i in range(0, len(self.population)):  
        [ind1, ind2] = self.randomSelection()  
        child = self.crossover(ind1, ind2)  
        self.mutation(child)  
        self.population[i] = child
```

```
def search(self):  
    self.iteration = 0  
    while self.iteration < self.maxIterations:  
        self.GAStep()  
        self.iteration += 1  
        print "Total iterations: ", self.iteration  
        print "Best solution: ", self.best.getFitness()
```

Demo

Best: 22.4 iterations: 0

Best: 17.6 iterations: 0

Best: 21.6 iterations: 0

Best: 10.4 iterations: 0

Best: 17.6 iterations: 0

Best: 21.6 iterations: 1

Best: 22.4 iterations: 2

Best: 22.5 iterations: 5

Total iterations: 10000

Best solution: 22.5

Number of 1's in a string



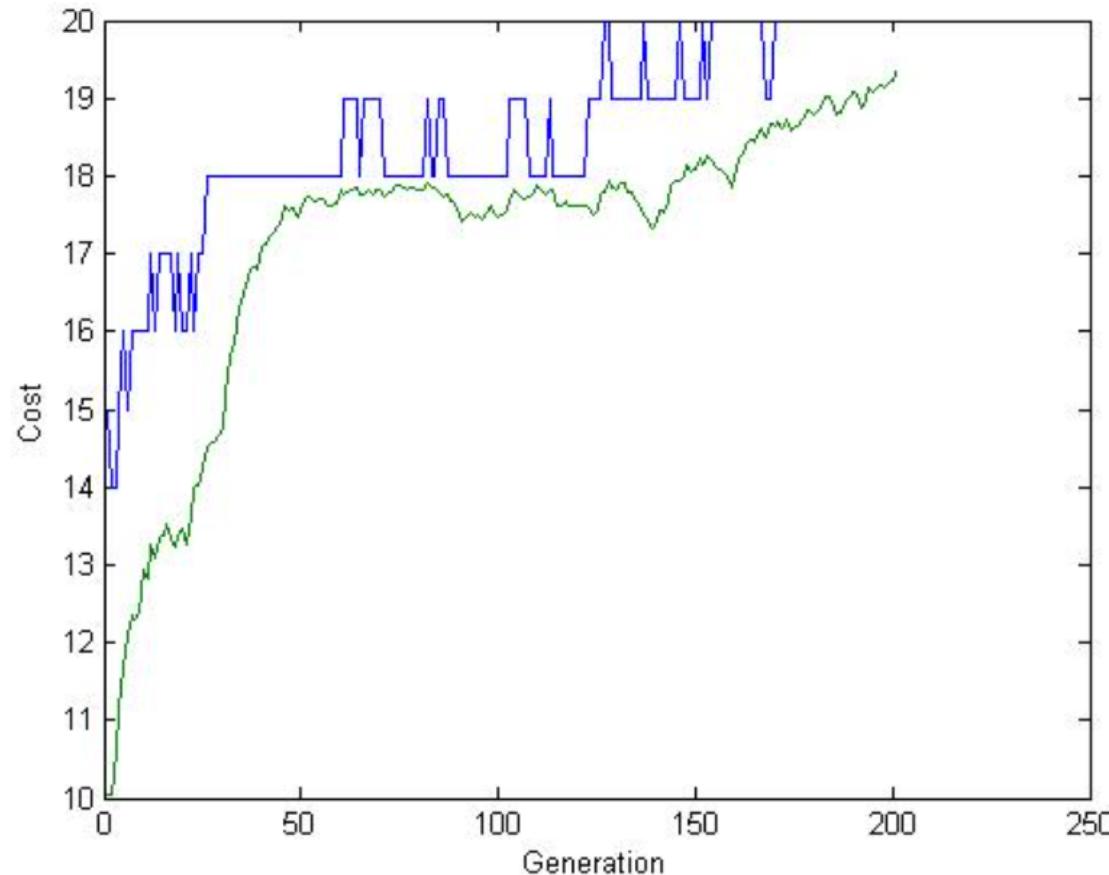
- Maximize the number of 1's in a bit string of length 20

OneMax Problem

[11111 11111 11111 11111]

- Fitness function: Sum of bits in each chromosome
- Population size → 100
- Mutation rate → 0.001
- Number of iterations (generations) → 200

Number of 1's in a string – First run



First run

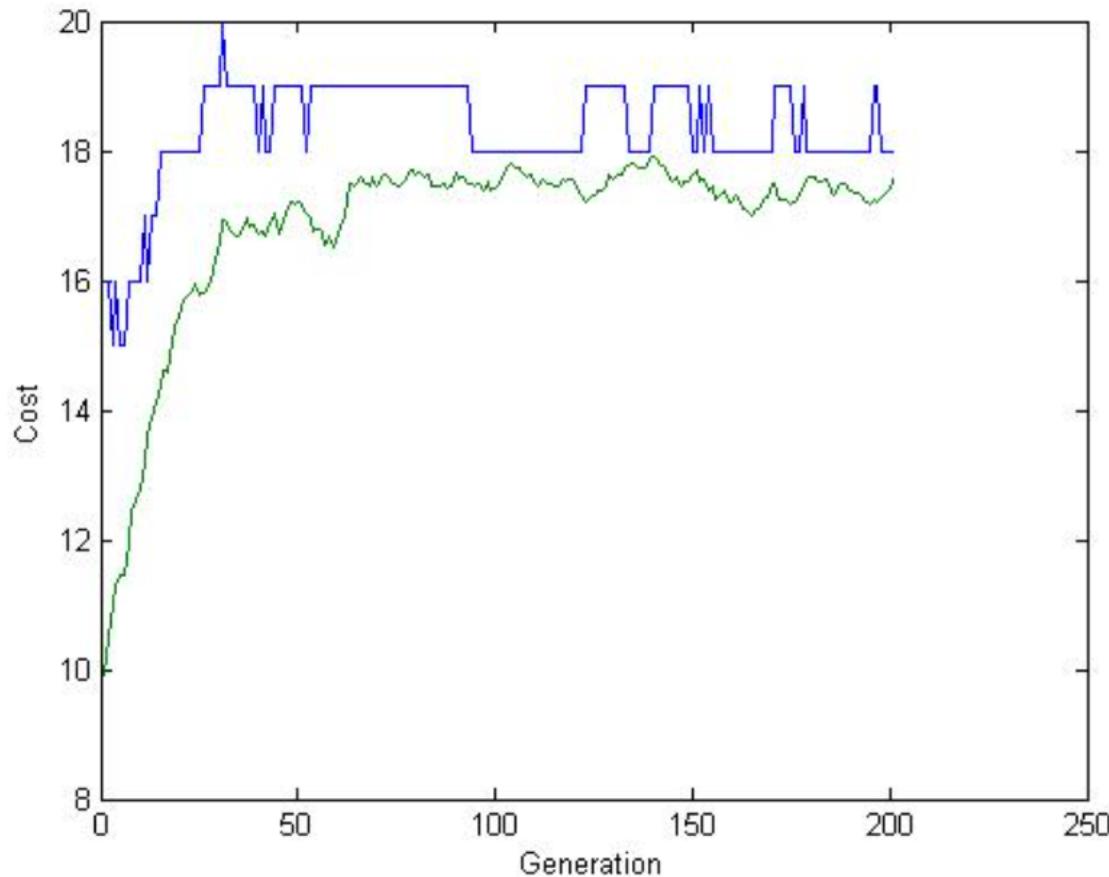
Blue curve is the highest fitness

Green curve is average fitness

Best Solution:

[11111 11111 11111 11111]

Number of 1's in a string – Second run



Second run

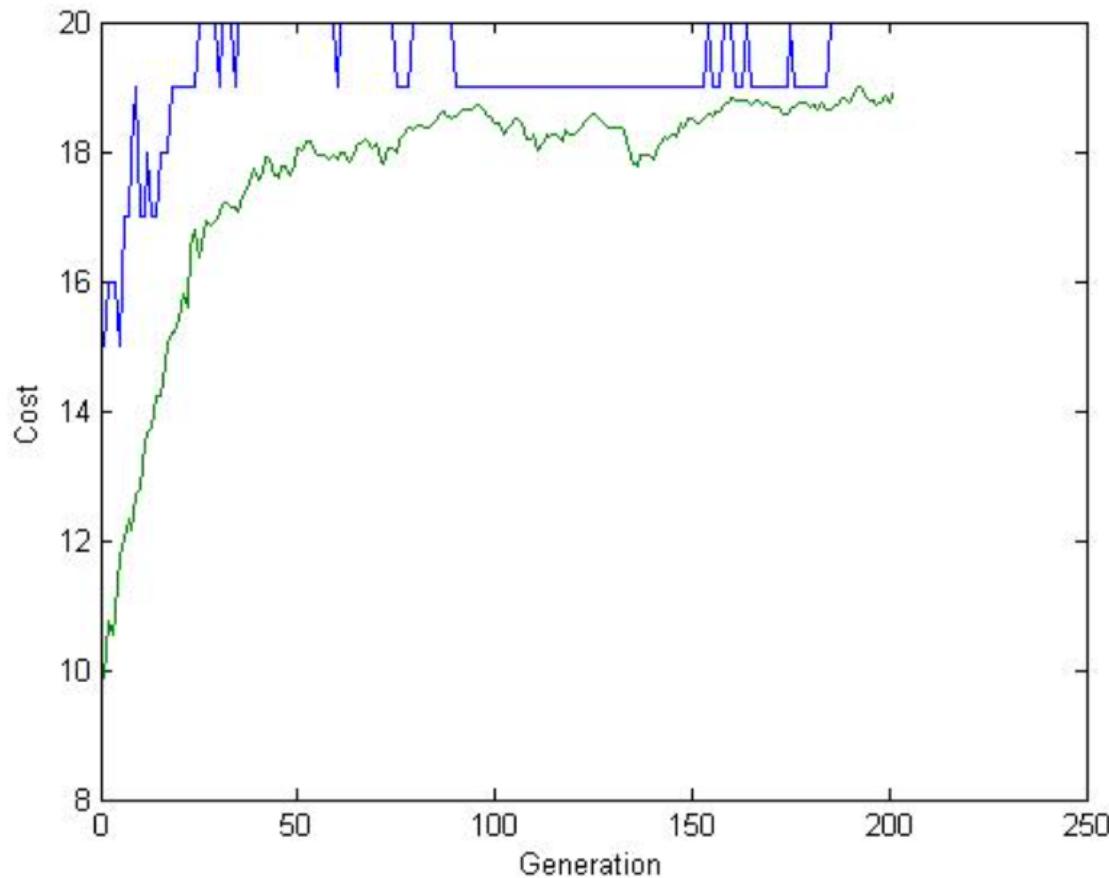
Blue curve is the highest fitness

Green curve is average fitness

Best Solution:

[11111 11111 11111 11111]

Number of 1's in a string – Third run



Third run

Blue curve is the highest fitness

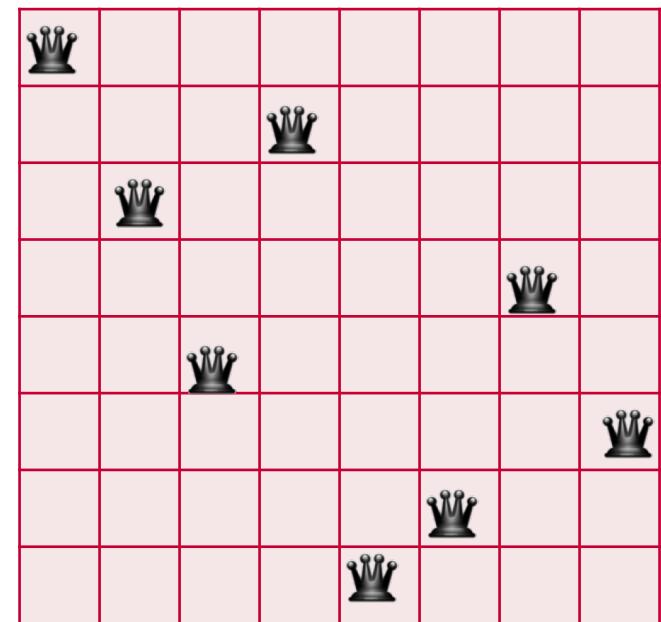
Green curve is average fitness

Best Solution:

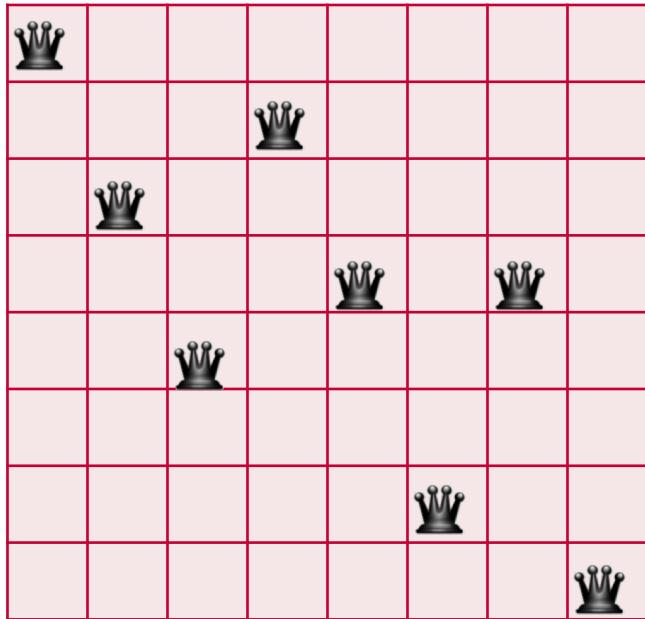
[11111 11111 11111 11111]

8-Queen Puzzle

- The objective of the 8-Queens puzzle is to place 8 chess queens on an 8×8 chessboard so that no two queens attack each other.
- The 8-Queens can be formulated as a nAI search problem. A **state is any placement of the 8 queens on the chess board**. The goal state is the arrangement of the 8 queens on the board in such a way that none can be attacked.
- The problem can be quite computationally expensive as there are **4,426,165,368** possible arrangements of eight queens on an 8×8 board, but only **92** solutions!
- **Representation** is an important issue. How would you suggest that we represent the individual in the figure?

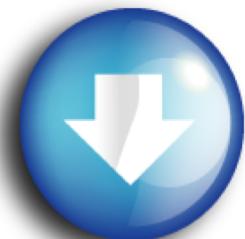
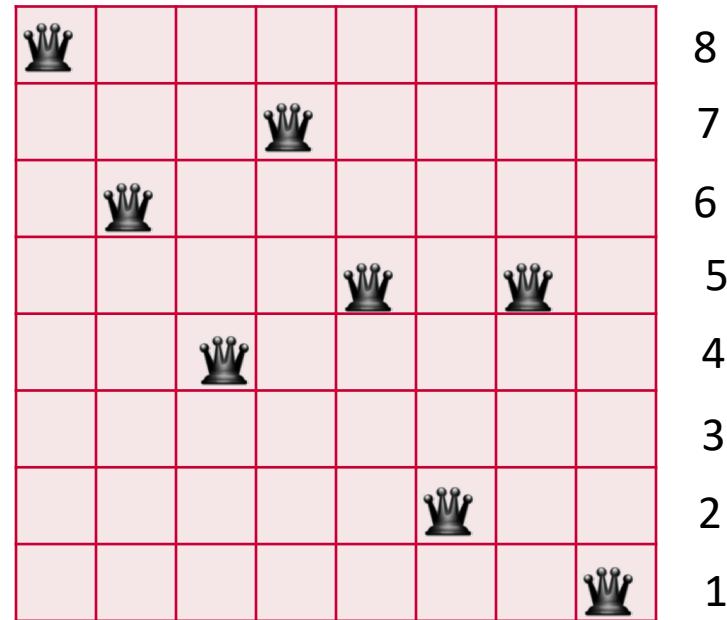


8-Queen Puzzle



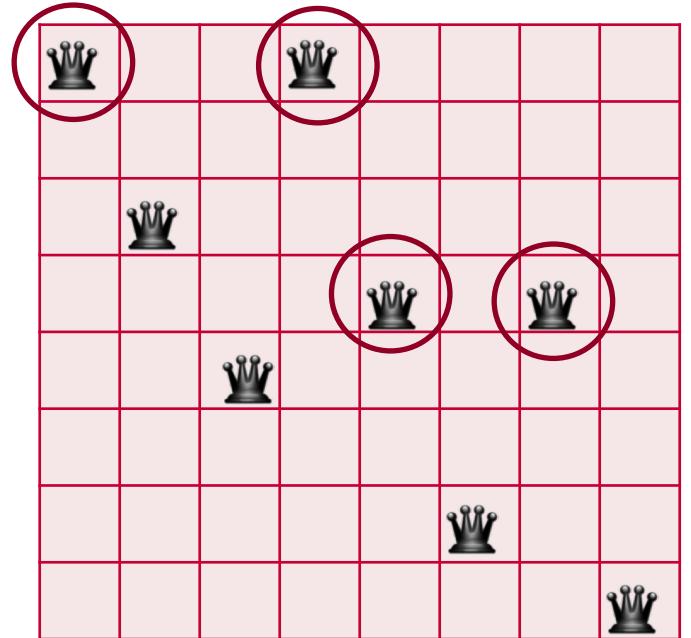
Just by applying a simple rule that constrains each queen to a single column (or row), it is possible to reduce the number of possibilities to just **16,777,216** (that is, 8^8) possible combinations.

Example of Representation

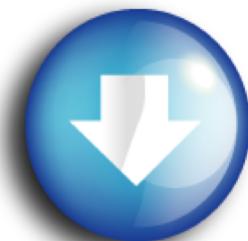


8	6	4	7	5	2	5	1
---	---	---	---	---	---	---	---

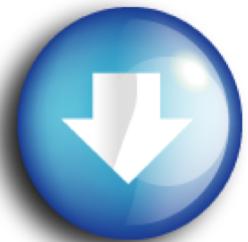
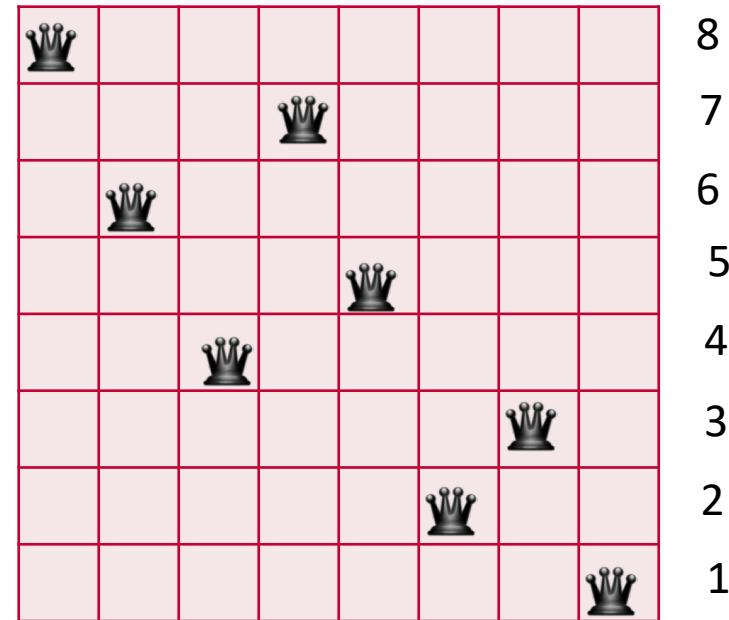
With Repetition



8
7
6
5
4
3
2
1



Without Repetition



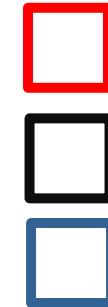
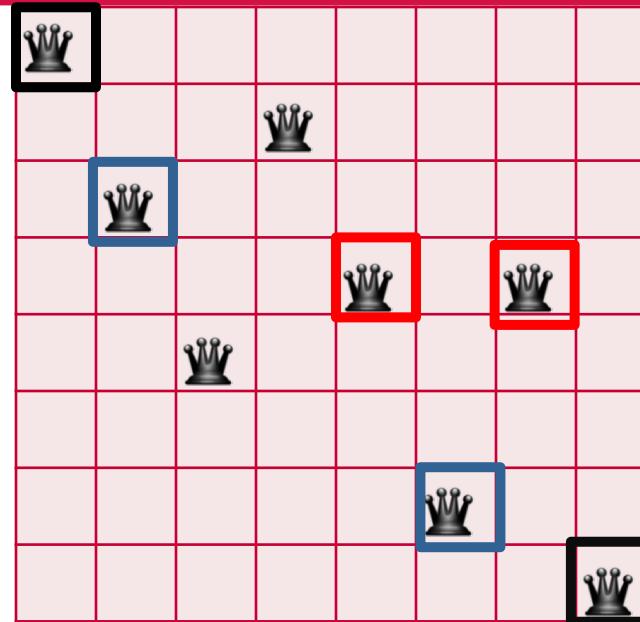
With vs. Without Repetition



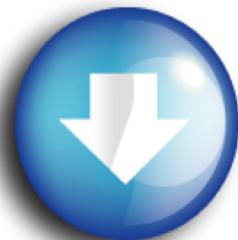
- Without repetition reduces the search space
 - Reduces from 16,777,216 candidate solutions to 40,320 (number of permutations of 8 values, so $8!$)
- No need to verify queens in the same row/column
- Typically *without* works better

- Initially a GA will generate a **random population**. Each individual represents a particular board configuration.
- A **heuristic** function is used to assess the fitness of all individuals
 - What kind of heuristic might be appropriate for the 8-Queens problem?
- Traditionally for the 8-Queen puzzle we use the number of attacking pairs or the number of non-attacking pairs of queens on the board

Fitness Evaluation Example (Attacking Pairs)

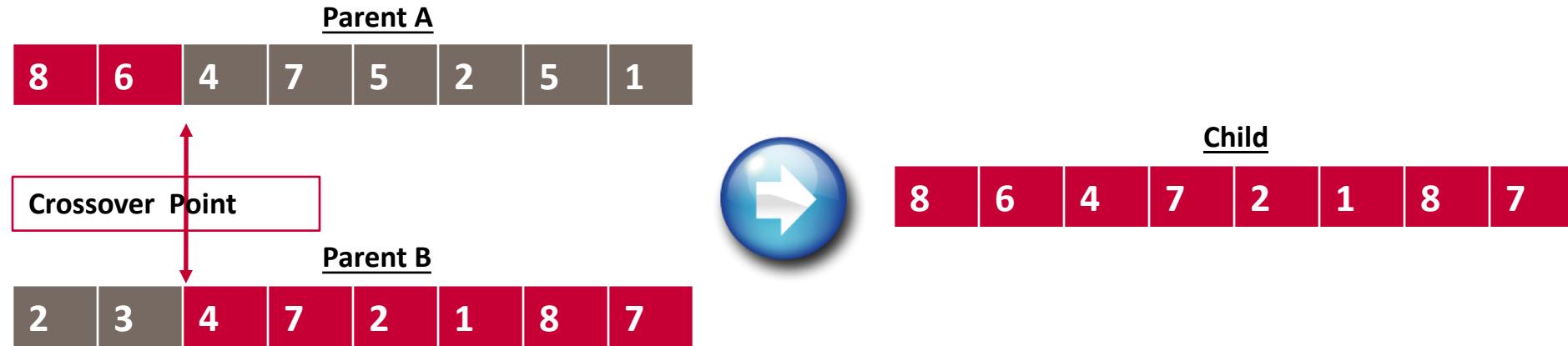


3 Conflicts



Fitness Value = 3

Reproduction Example (N-queens)

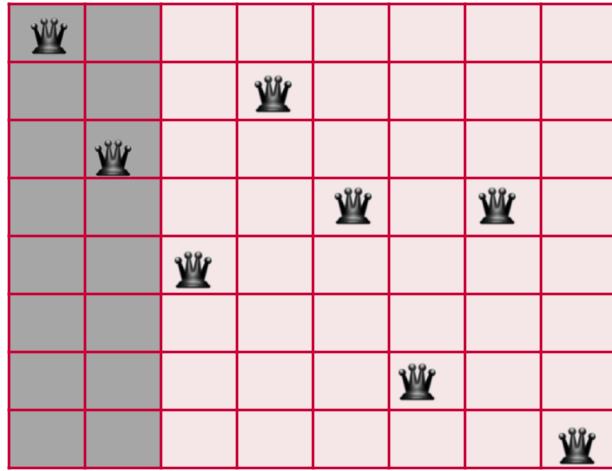


- In the example a crossover point of 2 is selected
- Child individual is constructed by combining the first two digits of parents A with the last six last digits of parent B
- Note you can also have multi-point crossover

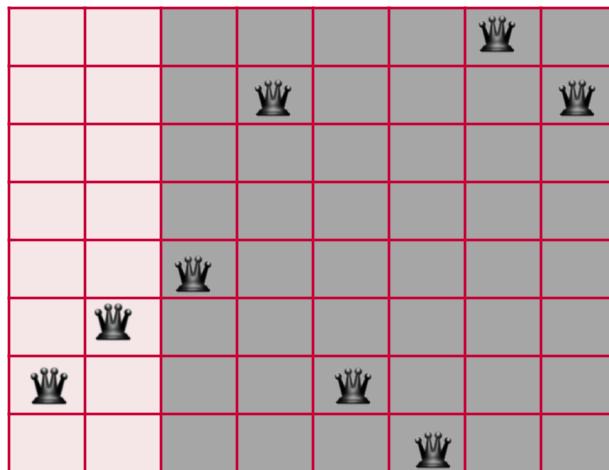
Crossover/Reproduction Example



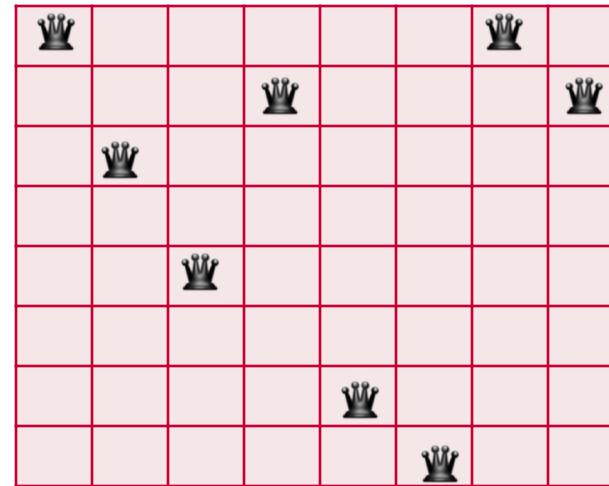
Parent A



Parent B



Child



Mutation Example -- Queens

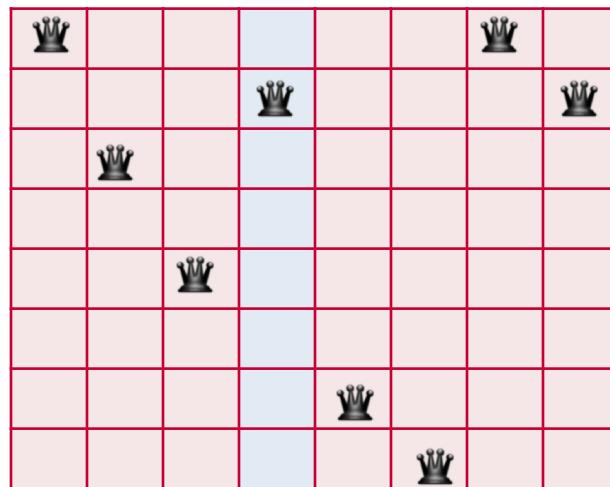


8	6	4	7	2	1	8	7
---	---	---	---	---	---	---	---

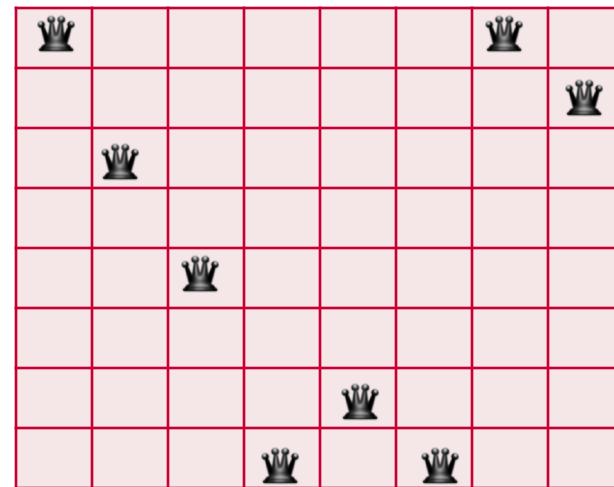


8	6	4	1	2	1	8	7
---	---	---	---	---	---	---	---

Parent



Child



- Discrete:
 - Each allele value in offspring (child) z comes from one of its parents (x, y) with equal probability: $z_i = x_i$ or y_i
 - Could use **n-point** or **uniform**
- Intermediate
 - exploits idea of creating children “between” parents (hence a.k.a. *arithmetic recombination*)
 - $z_i = \alpha x_i + (1 - \alpha) y_i$ where $\alpha: 0 \leq \alpha \leq 1$.
 - The parameter α can be:
 - **constant**: uniform arithmetical crossover
 - **variable** (e.g. depend on the age of the population)
 - picked at **random** every time

Single Arithmetic Crossover

- Parents: $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$
- Pick a single gene (k) at random
- child₁ is:
$$\langle x_1, \dots, x_k, \alpha y_{k+1} + (1 - \alpha)x_{k+1}, \dots, x_n \rangle$$

$$\alpha x_i + (1 - \alpha)y_i$$
- reverse for other child. e.g. with $\alpha = 0.5$
$$0.5 \times 0.8 + (1 - 0.5) \times 0.5 = 0.5$$

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
-----	-----	-----	-----	-----	-----	-----	-----	-----

$$X_i = 0.8$$

$$Y_i = 0.2$$

0.3	0.2	0.3	0.2	0.3	0.2	0.3	0.2	0.3
-----	-----	-----	-----	-----	-----	-----	-----	-----



0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.5	0.9
-----	-----	-----	-----	-----	-----	-----	-----	-----

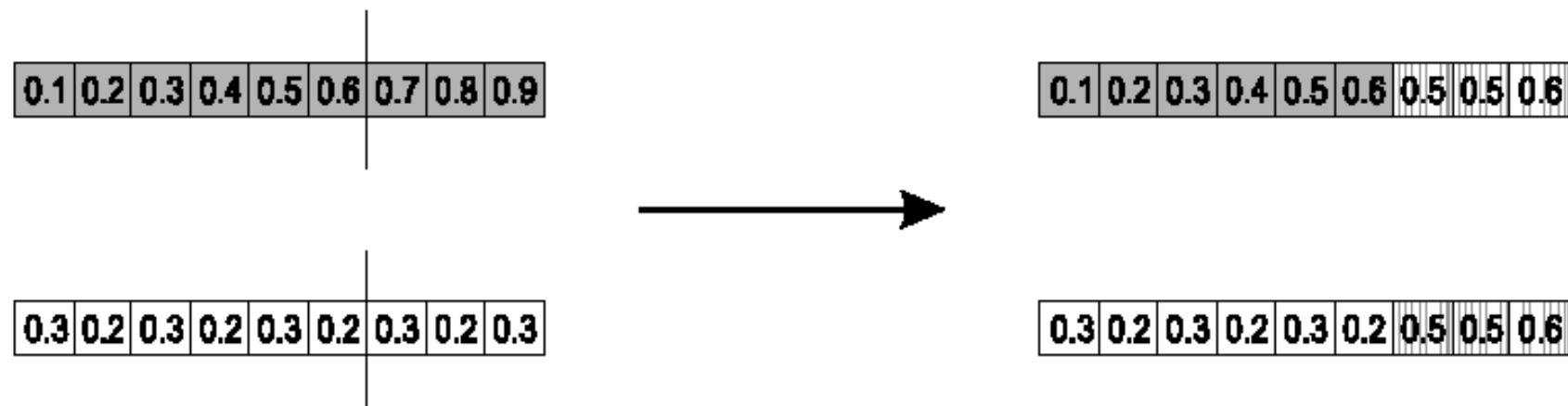
$$\alpha y_i + (1 - \alpha)x_i$$

$$0.5 \times 0.2 + (1 - 0.5) \times 0.8 = 0.5$$

0.3	0.2	0.3	0.2	0.3	0.2	0.3	0.5	0.3
-----	-----	-----	-----	-----	-----	-----	-----	-----

Simple Arithmetic Crossover

- Parents: $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$
- Pick random gene (k) after this point mix values
- child₁ is: $\langle x_1, \dots, x_k, \alpha y_{k+1} + (1-\alpha)x_{k+1}, \dots, \alpha y_n + (1-\alpha)x_n \rangle$
- reverse for other child. e.g. with $\alpha = 0.5$



Whole Arithmetic Crossover

- Parents: $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$

- child₁ is:

$$\alpha \mathbf{x} + (1 - \alpha) \mathbf{y}$$

- reverse for other child. e.g. with $\alpha = 0.5$

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
-----	-----	-----	-----	-----	-----	-----	-----	-----



0.2	0.2	0.3	0.3	0.4	0.4	0.5	0.5	0.6
-----	-----	-----	-----	-----	-----	-----	-----	-----

0.3	0.2	0.3	0.2	0.3	0.2	0.3	0.2	0.3
-----	-----	-----	-----	-----	-----	-----	-----	-----

0.2	0.2	0.3	0.3	0.4	0.4	0.5	0.5	0.6
-----	-----	-----	-----	-----	-----	-----	-----	-----

Estimation of Distribution Algorithms (EDAs)

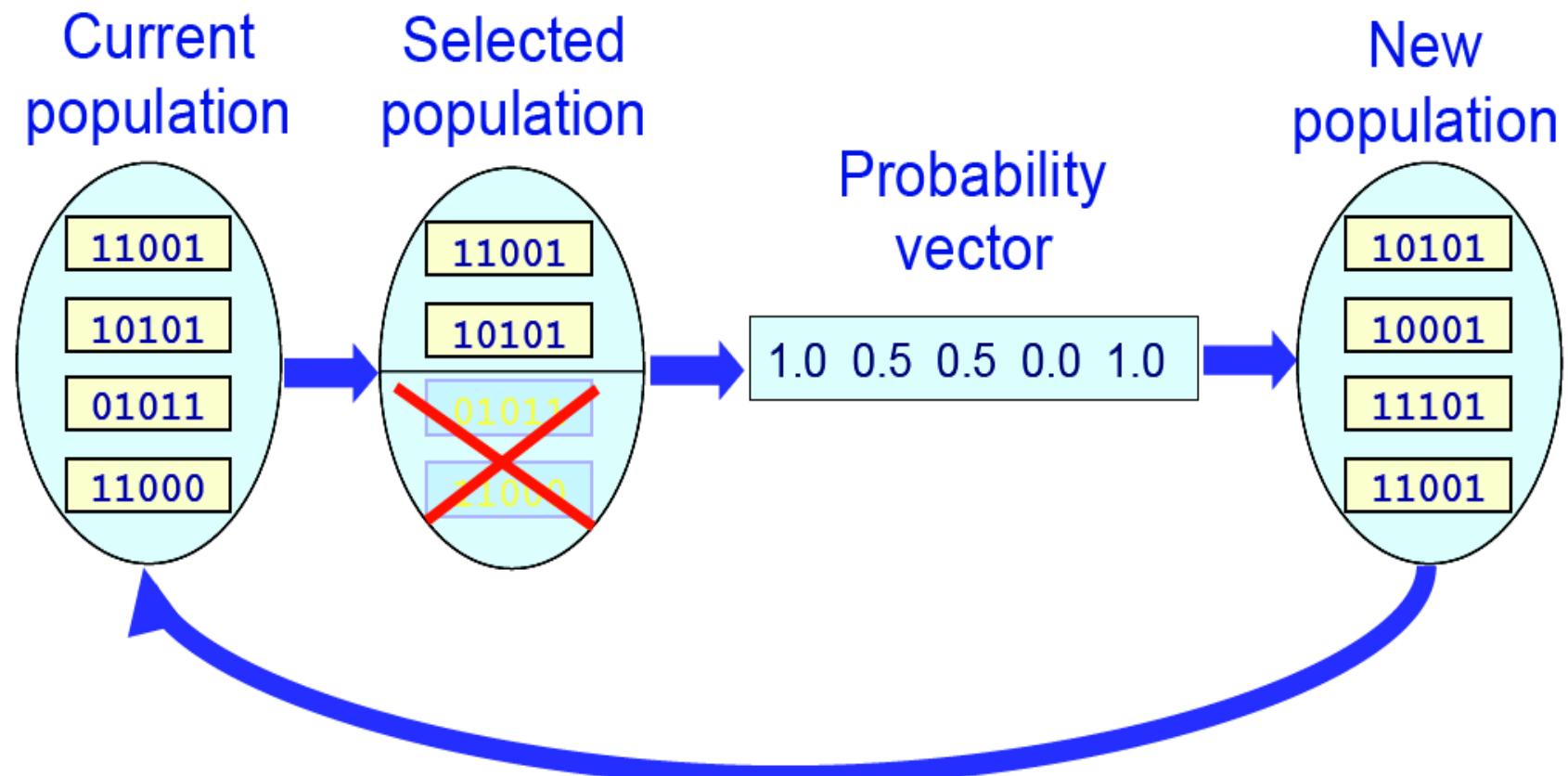
What is EDA



- EDA is a relatively new branch of **evolutionary algorithms**
M Pelikan, D.E Goldberg & E Cantu-paz (2000): “**Linkage Problem, Distribution Estimation and Bayesian Networks**”. Evolutionary Computation 8(3): 311-340.
- Replaces GA operators with the estimation of the distribution of selected individuals + sampling from this distribution
- The aim is to avoid the use of arbitrary operators (mutation, crossover) in favour of explicitly modelling and exploiting the distribution of promising individuals

The EDA Algorithm

- Step 1: Generate an initial population P_0 of M individuals uniformly at random in the search space
- Step 2: Select $N \leq M$ individuals from P_{l-1} according to a selection method
- **Step 3: Estimate the probability distribution $p_l(x)$ of an individual being among the selected individuals**
- **Step 4: Sample M individuals (the new population) from $p_l(x)$**
- Step 5: Repeat steps 2-4 for generations $l = 2, \dots$ until some stopping criteria met



Note: The variables (bits, genes) are treated independently here.
'Univariate Marginal Distribution Algorithm' (UMDA)



Swarm Intelligence



“The emergent collective intelligence of groups of simple agents”

[Bonabeu et al, 1999]



- Swarm intelligence (SI) is the property of a system whereby
 - The collective behaviors of simple agents interacting locally with their environment cause coherent functional global patterns to emerge
- Inspired by collective behavior of ant/bee colonies, flocks of birds, shoals of fish, etc.
- SI provides a basis with which it is possible to explore collective (or distributed) problem solving without centralized control or the provision of a global model
- Leverage the power of complex adaptive systems to solve difficult non-learn stochastic problems

Characteristics of Swarms



- Composed of many simple individuals or agents
 - In our case, these will represent solutions in the search space
- Individuals are homogeneous
- Local interactions based on simple rules
- Cooperation via indirect communication

Stochastic or Random Search



- Swarm Intelligence
- Genetic Algorithms
- Differential Evolution
- Local Search
 - Simulated Annealing
 - Tabu search
 - Dynamic Local Search
 - And more...

Swarm Intelligence Algorithms



- Ant Colony Optimization
- Particle Swarm Optimization (PSO)
- Artificial Bee Colony Algorithm
- And more...



Ant Colony Optimization

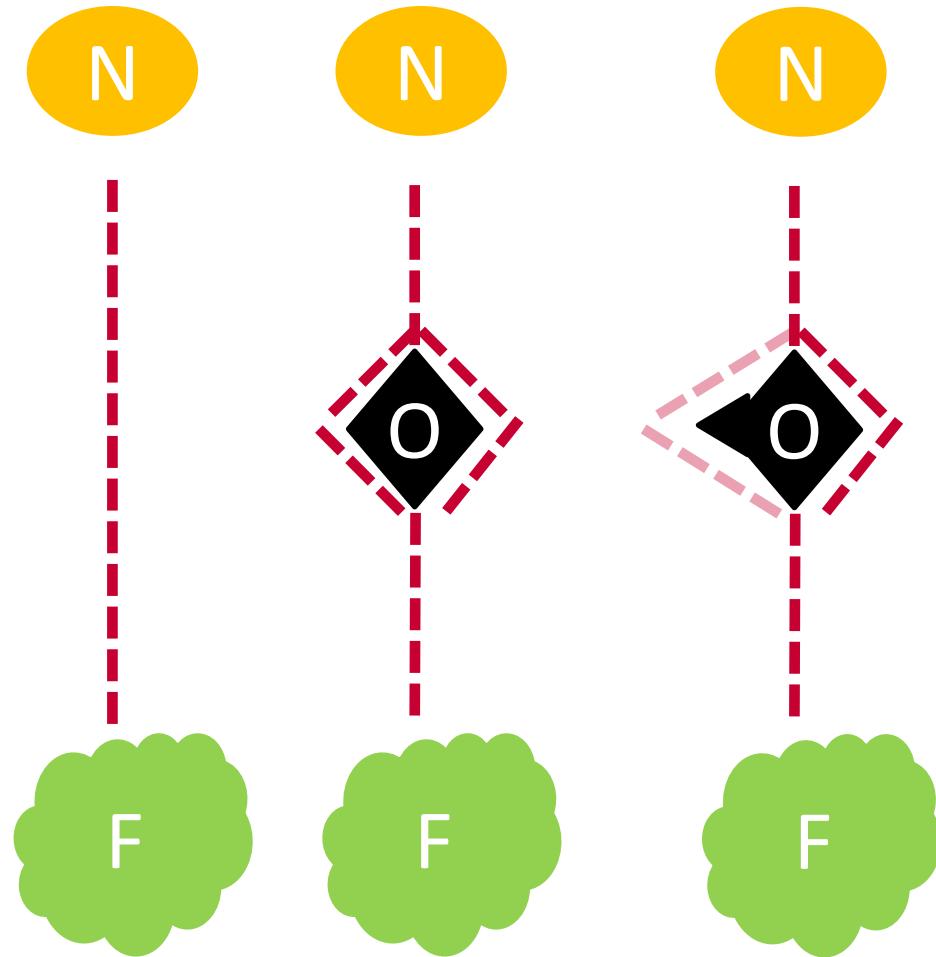
- Ants cooperate to find shortest paths to food sources
- A colony of *artificial* ants cooperate in finding good solutions
- Each ant – simple agent
- Ants communicate indirectly using chemicals (***stigmergy***)
- Artificial ants have some local information and memory of where its been



Denebourg's double bridge experiments

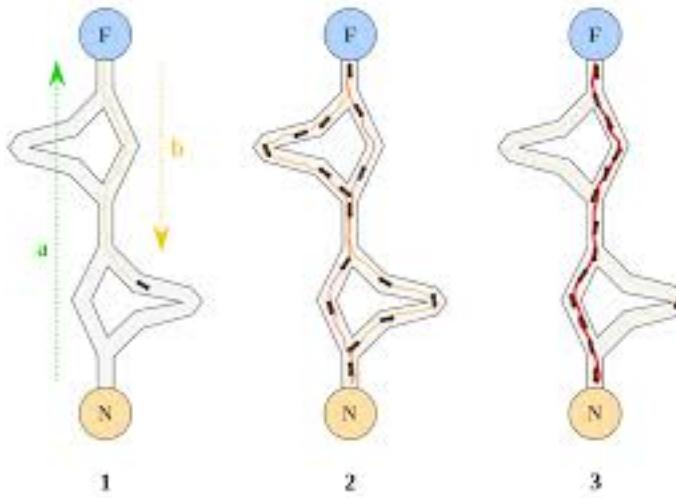


- Studied Argentine ants *I. humilis*
- Ants first had clear path from nest to food source
- Obstacle was subsequently placed in the path, s.t. ants would have an equal distance to food if they went left or right around obstacle
 - Found roughly 50% of ants went one way, 50% the other
- Obstacle was changed such that was quicker to go left and around obstacle than right
 - Found that after a short period of time most ants went shortest route around obstacle



ACO Concept

- Ants (blind) navigate from nest to food source
- Shortest path is discovered via pheromone trails
 - each ant moves at random
 - pheromone is deposited on path when **returning** to nest from food source
 - ants detect lead ant's path, *inclined* to follow
 - more pheromone on path increases **probability** of path being followed



- Virtual ***trail*** accumulated on path segments
- Starting node selected at random
- Path selected at random
 - based on amount of ***trail*** present on possible paths from starting node
 - higher probability for paths with more ***trail***
- Ant reaches next node, selects next path
- Continues until reaches starting node
- Finished ***tour*** is a solution

ACO System



- A completed tour is analyzed for optimality
- ***Trail*** amount adjusted to favor better solutions
 - Better solutions receive more trail
 - Worse solutions receive less trail
 - higher probability of ant selecting path that is part of a better-performing tour
- New cycle is performed
- Repeated until most ants select the same tour on every cycle (convergence to solution)

- Often applied to TSP (Travelling Salesman Problem): shortest path between n nodes
- Algorithm in Pseudocode:

Initialize Trail

Do While (Stopping Criteria Not Satisfied) – Cycle Loop

Do Until (Each Ant Completes a Tour) – Tour Loop

 Local Trail Update

End Do

 Analyze Tours

 Global Trail Update

End Do

ACO: Individual solution construction



- Ant chooses starting node randomly
- When ant is in node i and has constructed partial solution s , the probability of going to node j is:

$$p_{ij} = \begin{cases} \frac{(\tau_{ij})^\alpha (\eta_{ij})^\beta}{\sum_{k \in N_s} (\tau_{ik})^\alpha (\eta_{ik})^\beta}, & j \in N_s \\ 0, & \text{otherwise} \end{cases}$$

Where

τ_{ij} is pheromone on edge c_{ij} ; η_{ij} is the heuristic (local) information; and N_s is the neighbourhood of partial solution s , i.e. the set of feasible moves to extend solution s

α and β control the relative importance of pheromone τ_{ij} versus heuristic information η_{ij} .

- When all ants have constructed their solution, pheromones are updated on each edge
- If ant k used edge c_{ij} the pheromone deposited by that ant on that edge is :

$$\Delta\tau_{ij}^k = Q/L_k$$

where Q is a constant (e.g. 1) and L_k is the cost of the solution

- The pheromone update then for an edge c_{ij} is given by

$$\tau_{ij} \leftarrow (1 - p)\tau_{ij} + \sum_k \Delta\tau_{ij}^k$$

where p is the evaporation rate

- Improving variants (*Max-Min AS and ACS*): only best ant updates the pheromone, edges not traversed by best ant get an update of 0 for $\Delta\tau_{ij}^k$ (and still evaporate)

- Improving variant ACS added a diversification measure where each ant updates the pheromone on the last edge it traversed in each step:

$$\tau_{ij} \leftarrow (1 - \varphi)\tau_{ij} + \varphi\tau_0$$

where φ is the pheromone decay coefficient

- Reduces pheromone on arc it is choosing so encourages other ants to try

Each ant k

- Randomly selects a start node
- Chooses the next city on route with probability p_{ij}^k based on function of the distance from current city i to that city j , and the amount of pheromone on the edge between the two cities.
- Has memory, so only chooses cities that hasn't visited yet (i.e. always creates valid solution)
- Lays *pheromone* on each edge of route

Common Extensions



- Elitist Ant System
- Max-min Ant System (MMAS)
- Restart-based ACO
- ACO + Local search
- Rank-Based Ant System (ASrank)
- Continuous Orthogonal Ant Colony (COAC)
- Recursive Ant Colony optimization

Elitist Ant System (EAS)



- First improvement on ACO
- Provide strong additional reinforcement to the arcs belonging to the best tour found since the start of the algorithm

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k + \rho\Delta_{ij}^{best}$$

1/C^k If best ant k
travels on edge (i, j)

A diagram illustrating the elitist update rule for the pheromone trail τ_{ij} . The formula is shown as $\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k + \rho\Delta_{ij}^{best}$. The term $\rho\Delta_{ij}^{best}$ is circled in red, and a red arrow points from this circle to the text "1/C^k If best ant k travels on edge (i, j)".

Rank-Based Ant System (ASrank)



- Another improvement over AS
- Each ant deposits an amount of pheromone that decreases with its rank
- In each iteration, only the best ($w-1$) ranked ants and best-so-far ant are allowed to deposit pheromone

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{w-1} (w - k) \Delta \tau_{ij}^k + w \Delta_{ij}^{best}$$

Four main differences with respect to AS

- Strongly exploits the best tours found so far
 - This may lead to stagnation
- Limits the possible range of pheromone values (lower and upper bound)
- Pheromone values initialized to upper limit
- Pheromone values are reinitialized when the system approaches stagnation

Min-Max Ant System (MMAS)



- After all ants construct a solution, pheromone values are updated. Evaporation is the same as described before

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k + \rho\Delta_{ij}^{best}$$

- **Lower** and **upper** limits on pheromones limit the probability of selecting a city
- Initial pheromone values are set to the **upper limit**, resulting in initial exploration
- Occasionally pheromones are re-initialized

- Uses ideas not included in the original implementation
- Different in three main points:
 1. Exploits the accumulated search experience more strongly
 2. Pheromone evaporation and deposit take place only on the best-so-far tour
 3. Each time an ant uses an arc, some pheromone is removed from the arc

- Ants construct solutions
- Local search run on each solution to find nearby improvements
- Pheromone added after local search and process repeats
- Note ACO very suitable for parallelism

Restart-based ACO



- End of First Run
- Save Best Tour (route and cost)
- All ants die!
- New ants are born