CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Metaheuristic Optimization

Genetic Algorithms

Dr. Diarmuid Grimes

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 previous population with newly created choromosomes

Replacement Variants

1. Newly created solutions replace some (or all) of the current solutions

2. Delete-all or full replacement
   - All previous solutions are deleted

Our current method

3. 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

Steady-state replacement

- 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***.

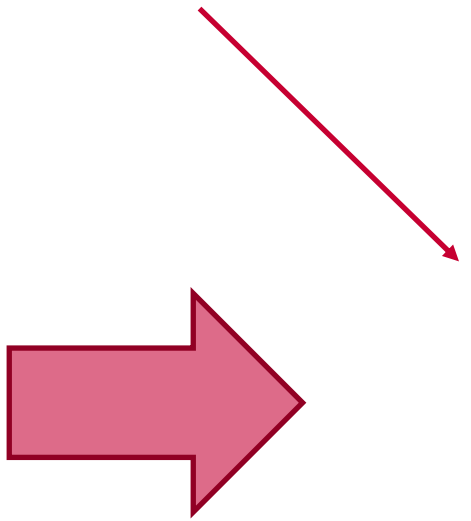
Top M candidates survive → No crossover or mutation for those candidates

Elitist

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

Elitist Implementation

Possible method (for maximization problem):

- Store best ***x*** chromosomes from previous generation

```
elite_sols = pop[0:x]
elite_fits = [i.fitness for i in elite_sols]
worst_fit = min(elite_fits)
worst_idx = elite_fits.index(worst_fit)
for i in range(x,len(pop)):
    if pop[i].fitness > worst_fit:
        elite_sols[worst_idx] = pop[i]
        elite_fits[worst_idx] = i.fitness
        worst_fit = min(elite_fits)
        worst_idx = elite_fits.index(worst_fit)
```

- Generate (P-x) children

Alternative Elitist Implementation

Possible method:

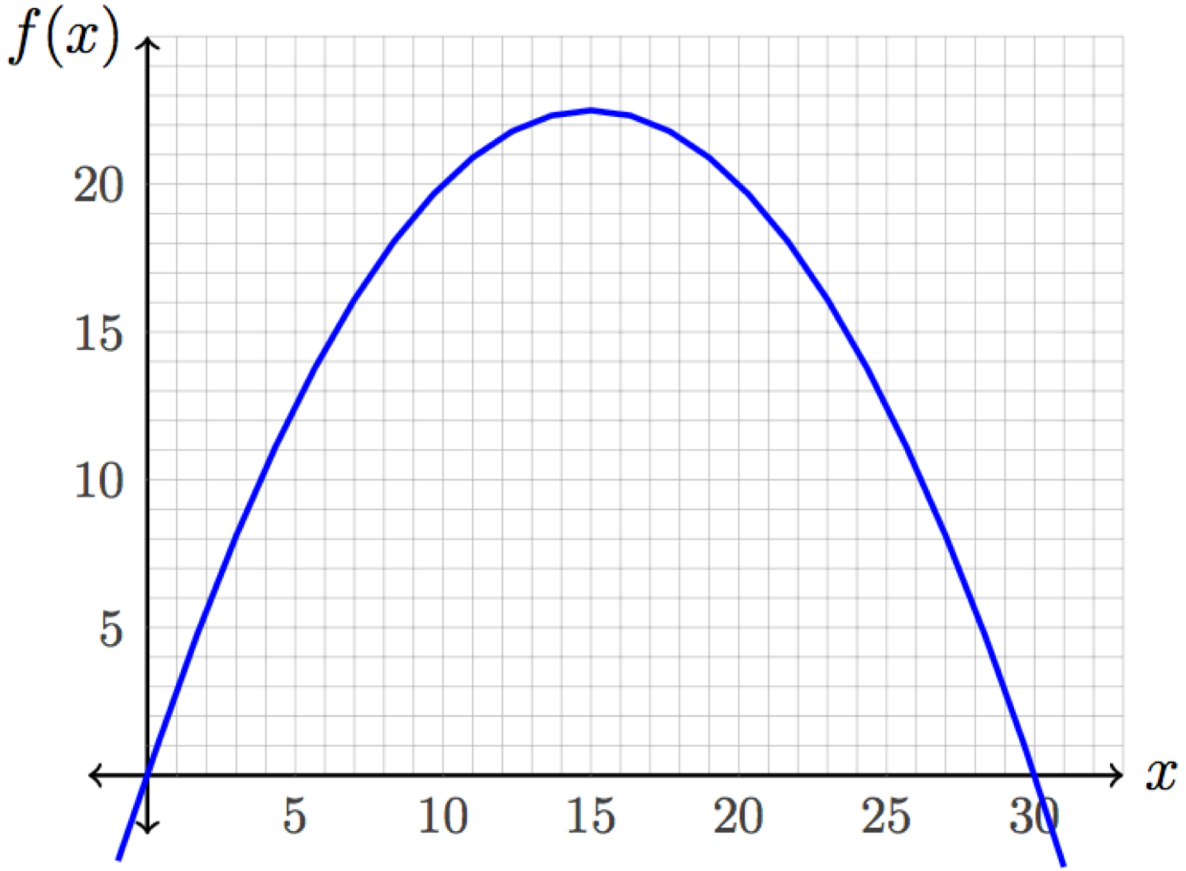
- Store best ***x*** chrosomes from previous generation
- We compute the fitness of each child after generation
- We can keep top_x and bottom_x lists, update if required when computing fitness
  - And single fitness values. E.g., for x=5, could also maintain fitness value of 5^{th} best and 5^{th} worst
  - Check if fitness of new child is better than 5^{th} best, or worse than 5^{th} 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

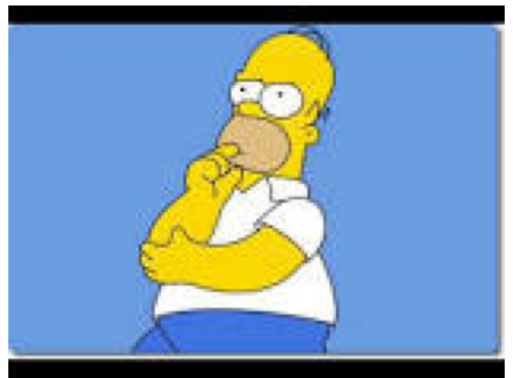
00000

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
		Sum	147.6	
		Average	14.76	
		Max	22.4	

Initial Population

Best chromosome so far

Selection

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

Chromosome Number	Mating Pairs	New Population	x Value	Fitness Value $f(x)$
5	0 1 | 1 0 0			
2	1 1 | 0 1 0			
4	0 1 1 1 | 0			
8	0 1 0 0 | 1			
9	0 0 0 1 | 1			
2	1 1 0 1 | 0			
7	1 0 1 1 0			
4	0 1 1 1 0			
10	1 0 0 | 0 1			
8	0 1 0 | 0 1			

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 x 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):
        """
        Current fitness value.
        Objective function: -(x^2/10) + 3*x (x --> current individual)
        """
        self.fitness = 0
        x = self.binaryToInt()
        self.fitness = -(x**2)/10.0 + 3.0*x

    def copy(self):
        # Cloning this object
        ind = Individual(self.genSize)
        ind.genes = []+self.genes
        return ind
```

Genes
Boolean variables 1/0

Random Initial values

Objective function

CIT

Chromosome Or Individual

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

```
class GA:
    def __init__(self, _popSize, _genSize, _mutationRate, _maxIterations, _elites, _trunk):
        self.population     = []
        self.matingPool     = []
        self.best           = None
        self.bestVal        = -1
        self.popSize        = _popSize
        self.genSize        = _genSize
        self.mutationRate   = _mutationRate
        self.maxIterations  = _maxIterations
        self.elites         = _elites
        self.trunk          = _trunk
        self.iteration      = 0
        self.initPopulation()
```

Parameters
&
Initial values

```
    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].genes
        self.bestVal = self.population[0].getFitness()
```

```
def updateMatingPool(self):

    self.matingPool = []
    x = round(self.popSize*self.trunk)
    mybest = self.population[0:x]
    best_fits = [i.getFitness() for i in mybest]
    worst_fit=min(best_fits)
    worst_idx = best_fits.index(worst_fit)

    for i in range(x,self.popSize):
        if self.population[i].getFitness() > worst_fit:
            mybest[worst_idx] = self.population[i]
            best_fits[worst_idx] = self.population[i].getFitness()
            worst_fit = min(best_fits)
            worst_idx = best_fits.index(worst_fit)

    for ind_i in mybest:
        for j in range(int(1/self.trunk)):
            self.matingPool.append( ind_i.copy() )
```

Find top x chromosomes

Fill mating pool with (popSize/x) copies of each of these chromosomes

Truncation Selection

```
def updateMatingPool(self):
```

```
    if self.elites < x:
        x = self.elites
    elite_sols = mybest[0:x]
    if x:
        elite_fits = [i.getFitness() for i in elite_sols]
        worst_fit = min(elite_fits)
        worst_idx = elite_fits.index(worst_fit)
        for i in range(x,len(mybest)):
            if mybest[i].getFitness() > worst_fit:
                elite_sols[worst_idx] = mybest[i]
                elite_fits[worst_idx] = mybest[i].getFitness()
                worst_fit = min(elite_fits)
                worst_idx = elite_fits.index(worst_fit)
    return elite_sols
```

CIT

Elite Selection

```
def trunkSelection(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)
    onePoint    = random.randint(0, self.genSize)
    child.genes = []+indA.genes
    child.genes[onePoint:] = indB.genes[onePoint:]
    child.computeFitness()
    self.updateBest(child)
    return child
```

1-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()
    elite_sols = self.updateMatingPool()
    self.population[:self.elites] = elite_sols
    self.newGeneration()
```

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

Initialising first ***num_elite*** chromosomes of new population to best of old

Fill remaining through crossover and mutation

```
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.bestVal, self.best,"\n\n")
```

Demo

Best: 21.6	Sol [0, 0, 1, 1, 0]	iterations: 0
Best: 22.1	Sol [1, 0, 1, 1, 0]	iterations: 3
Best: 22.4	Sol [0, 0, 0, 0, 1]	iterations: 7
Best: 22.5	Sol [1, 1, 1, 1, 0]	iterations: 9

Total iterations: 10000

Best solution: 22.5 [1, 1, 1, 1, 0]

Number of 1’s in a string

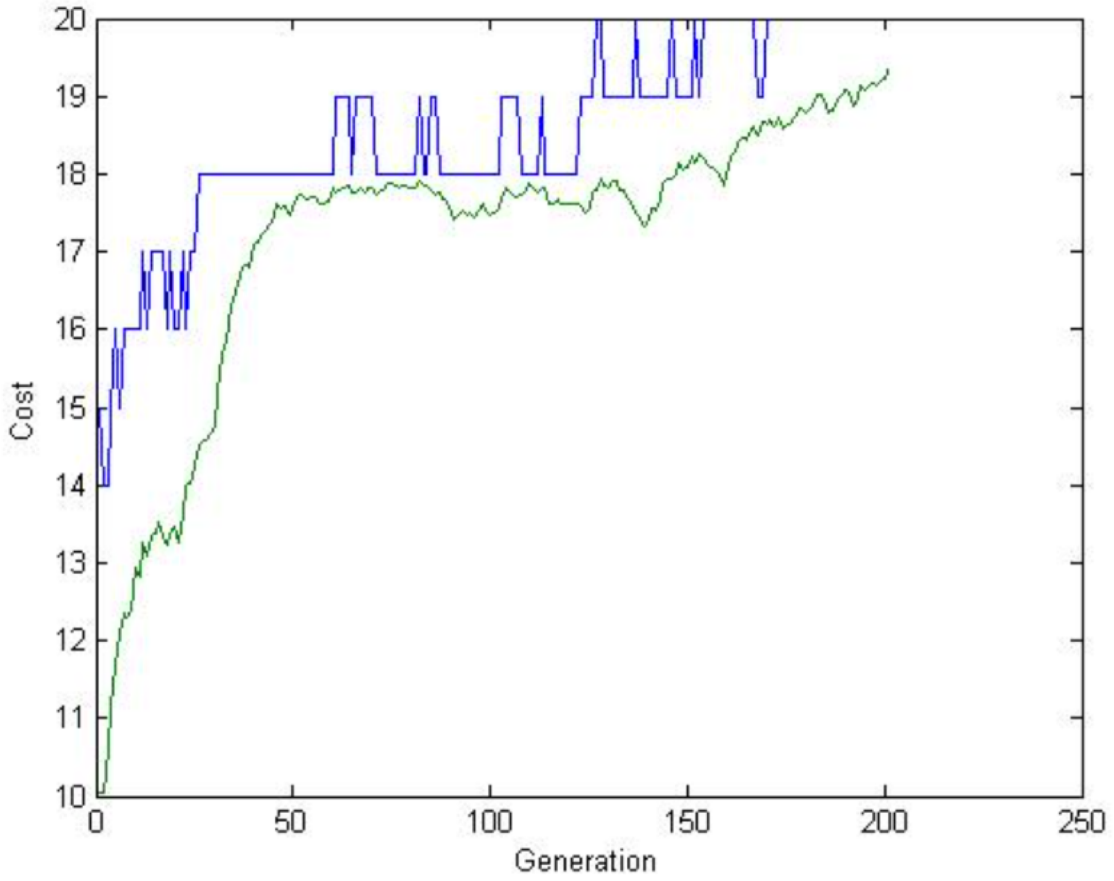
- Maximize the number of 1’s in a bit string of length 20

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

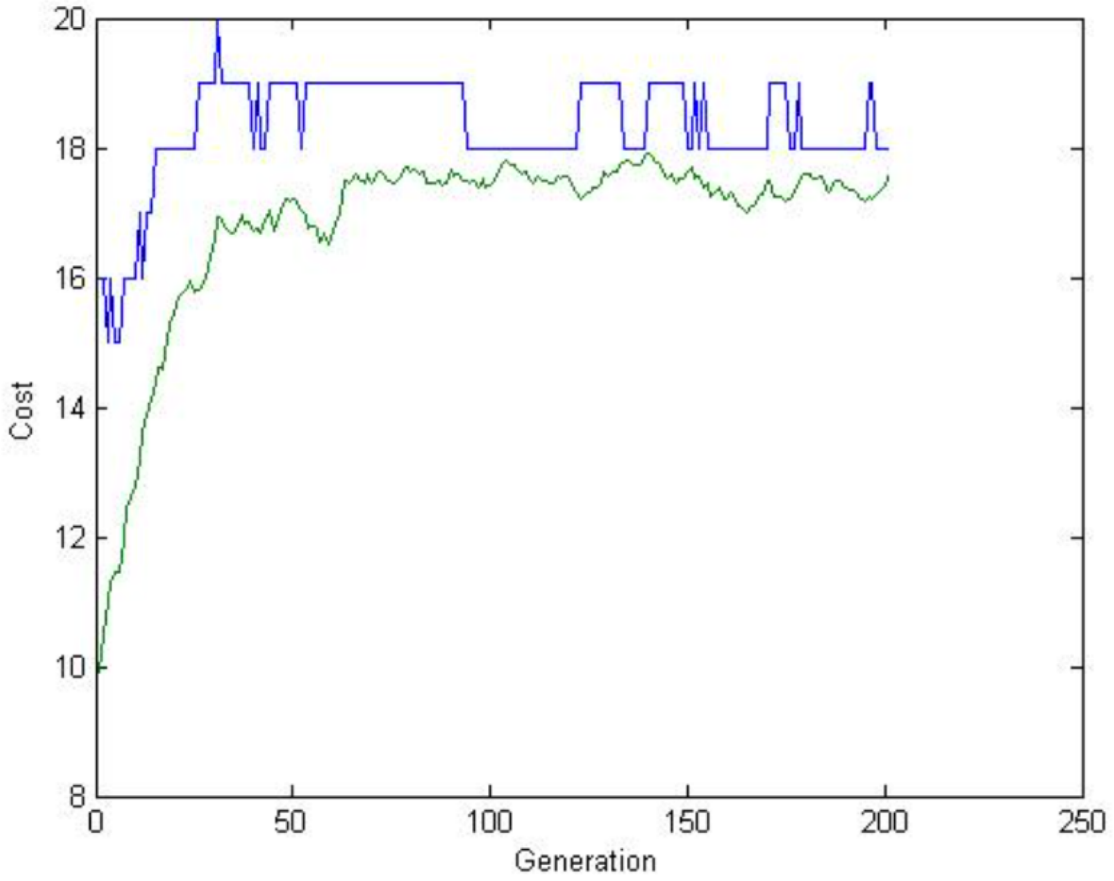
Blue curve is the highest fitness

Green curve is average fitness

Best Solution:

[11111 11111 11111 11111]

Number of 1's in a string – Second run



Second run

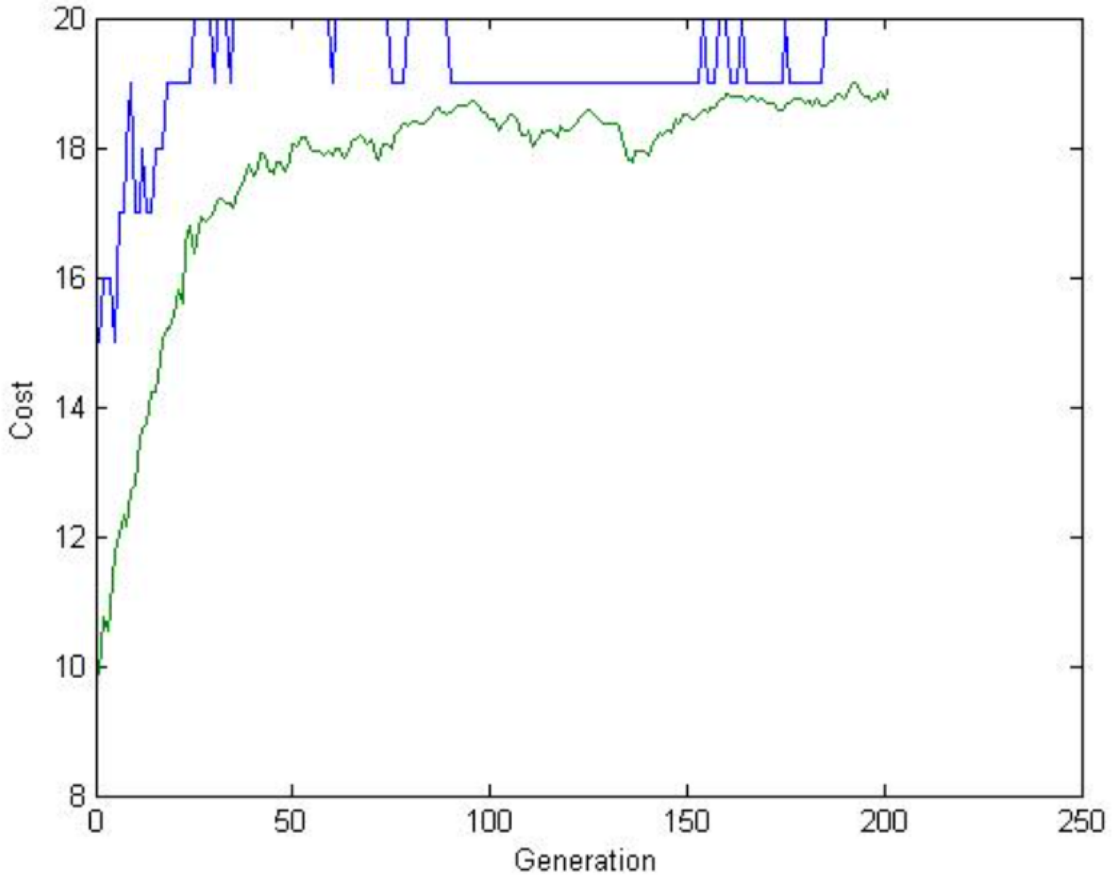
Blue curve is the highest fitness

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

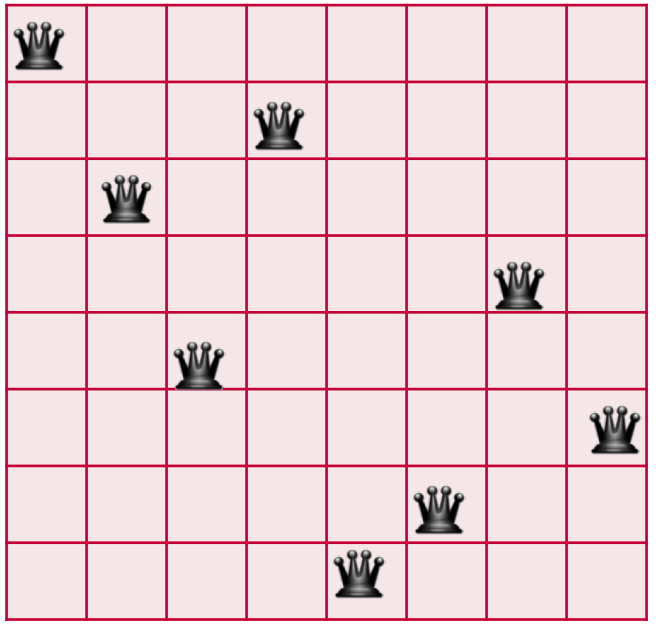
Green curve is average fitness

Best Solution:

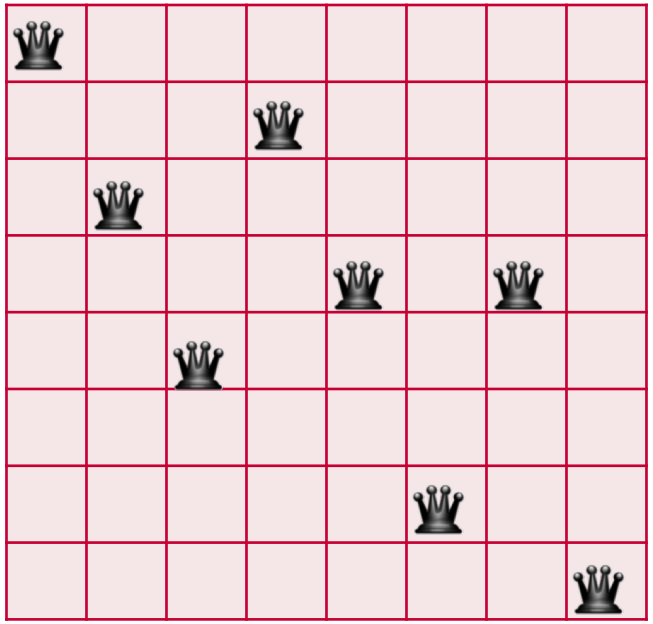
[11111 11111 11111 11111]

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** in an important issue. How would you suggest that we represent the individual in the figure?

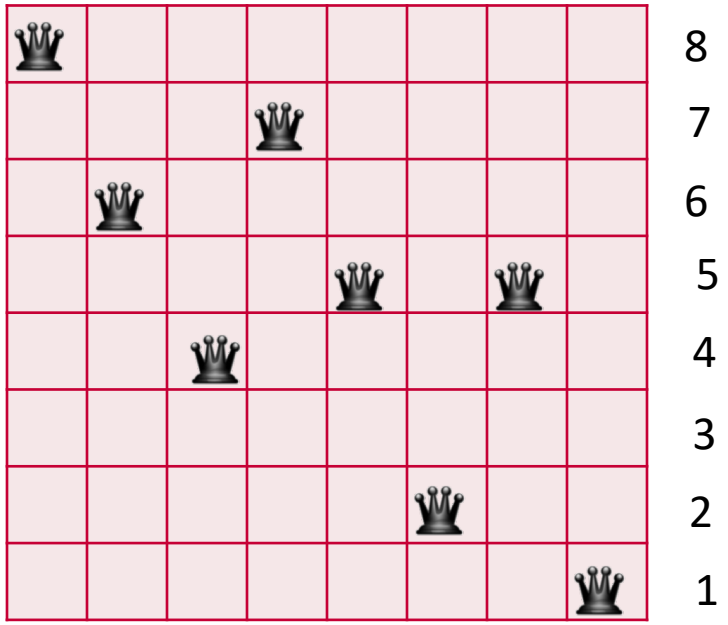


8-Queen Puzzle



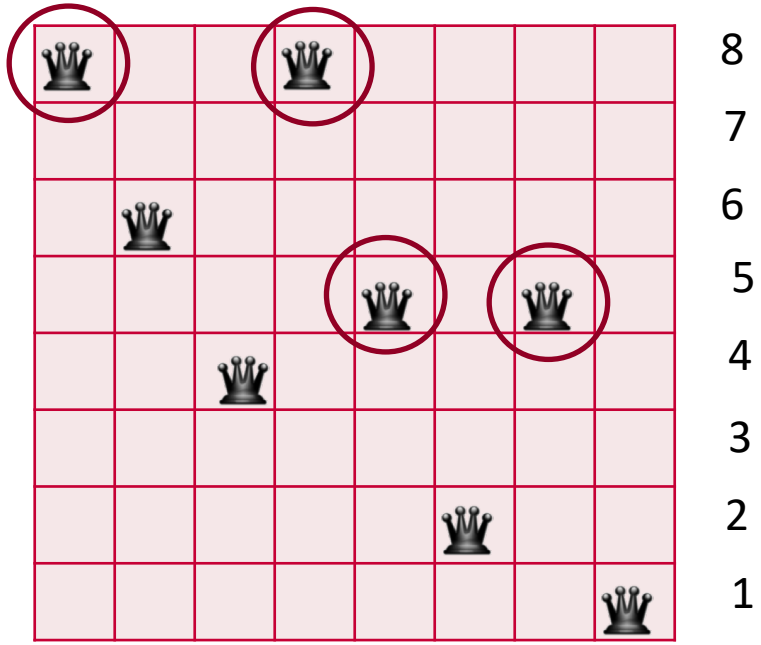
Just by applying a simple rule that constrains each queen to a single column (or row), it is possible to reduce the number of possibilities to just **16,777,216** (that is, 8^8) possible combinations.

Example of Representation

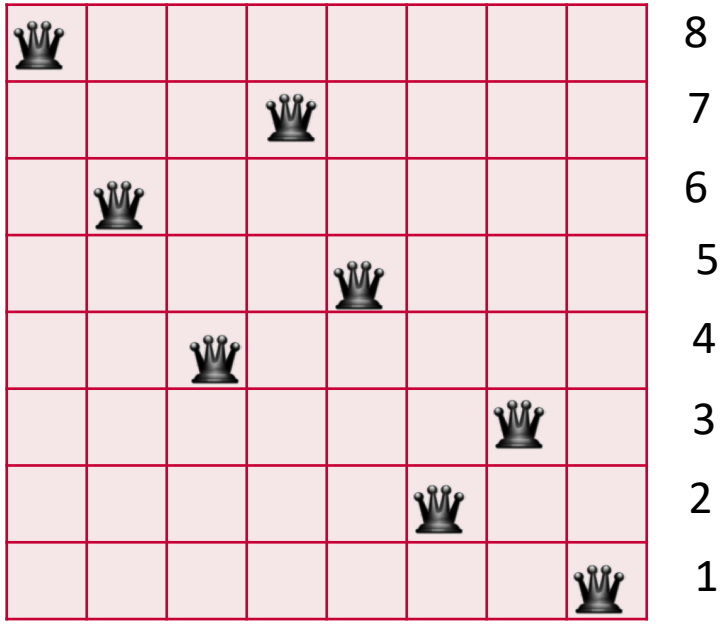


8	6	4	7	5	2	5	1

With Repetition



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

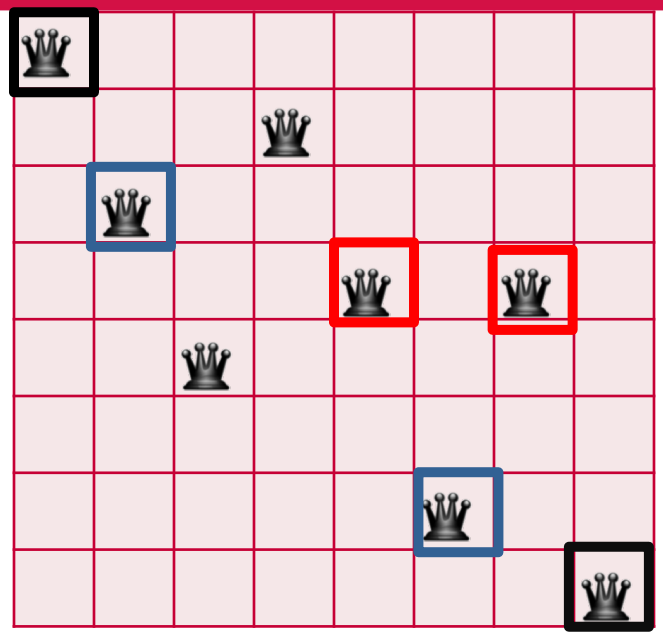
Population Generation and Fitness Evaluation

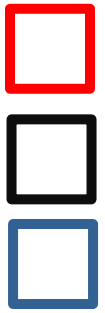
- Initially a GA will generate a **random population**. Each individual represents a particular board configuration.

- 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)

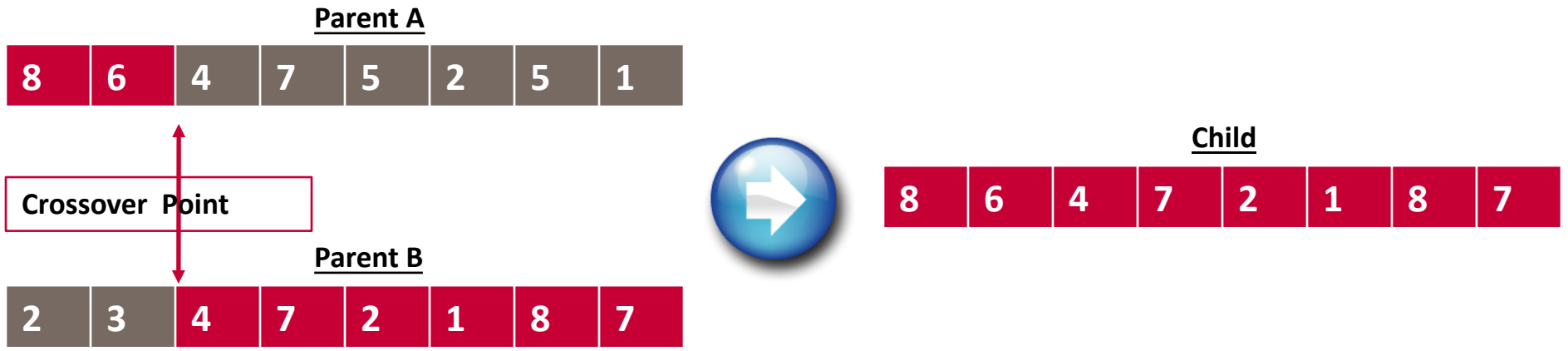




3 Conflicts

Fitness Value = 3

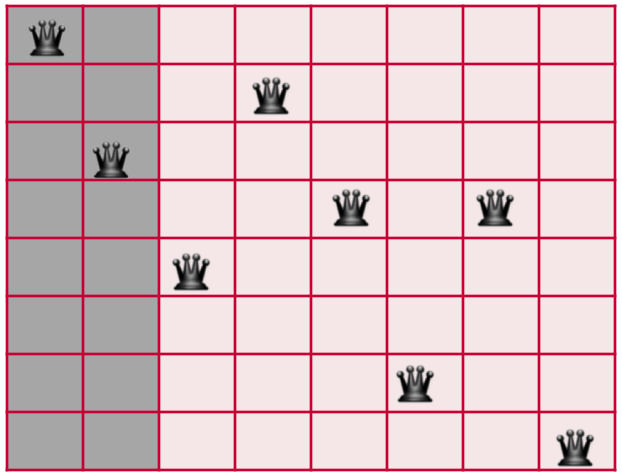
Reproduction Example (N-queens)



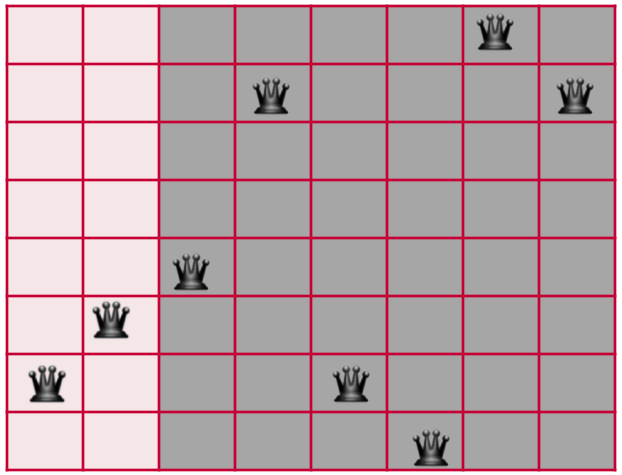
- In the example a crossover point of 2 is selected
- Child individual is constructed by combining the first two digits of parents A with the last six last digits of parent B
- Note you can also have multi-point crossover

Crossover/Reproduction Example

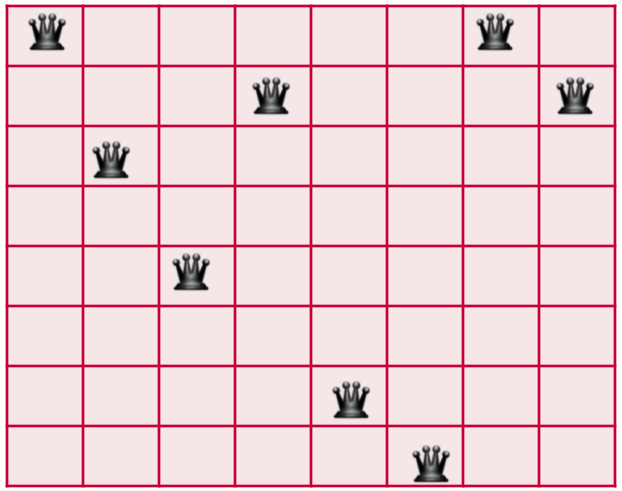
Parent A



Parent B



Child

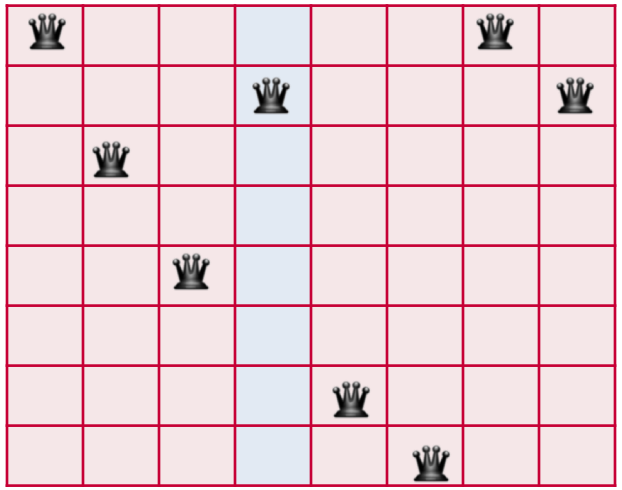


Mutation Example -- Queens

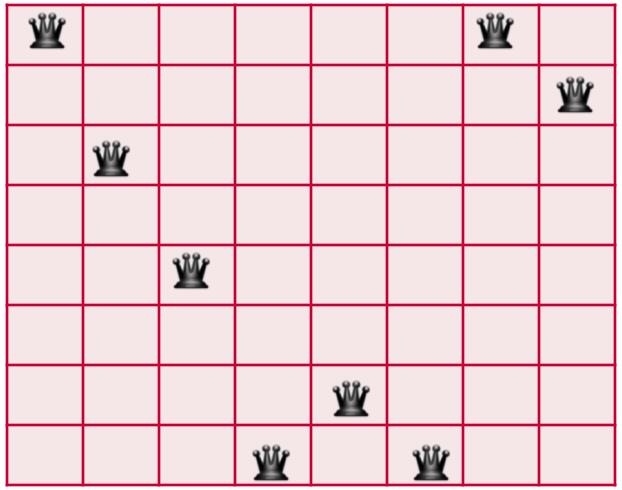
8	6	4	7	2	1	8	7

8	6	4	1	2	1	8	7

Parent



Child



Crossover for Real Valued GAs

- 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: $\langle x_1,\ldots, x_n \rangle$ and $\langle y_1,\ldots, y_n \rangle$
- Pick a single gene (k) at random

- $child_1$ is:

$$\langle x_1, \ldots, x_k, \alpha y_{k+1} + (1-\alpha)x_{k+1}, \ldots, x_n \rangle$$

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

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

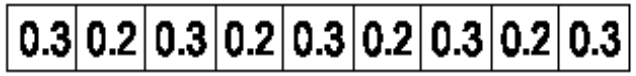
$$0.5 \times 0.8 + (1-0.5) \times 0.5 = 0.5$$

$X_i = 0.8$

$Y_i = 0.2$

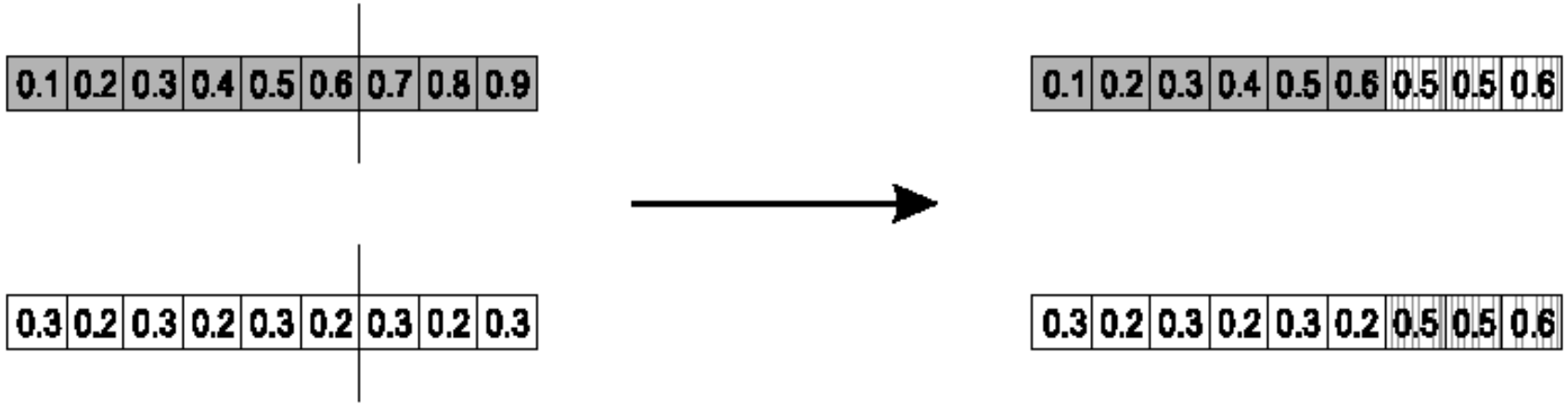
$$\alpha y_i + (1-\alpha)x_i$$

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



Simple Arithmetic Crossover

- Parents: $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$
- Pick random gene (k) after this point mix values
- child$_1$ is: $\langle x_1, \ldots, x_k, \alpha y_{k+1} + (1-\alpha)x_{k+1}, \ldots, \alpha y_n + (1-\alpha)x_n \rangle$
- reverse for other child. e.g. with $\alpha = 0.5$

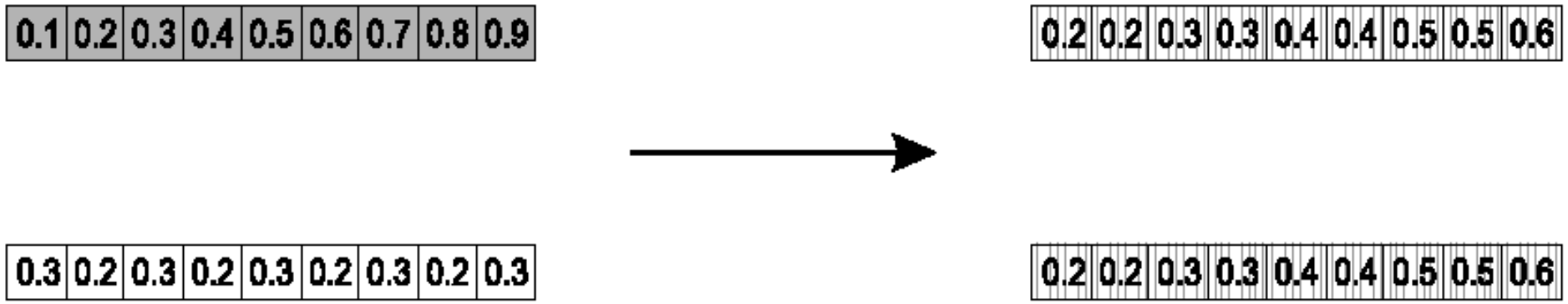


Whole Arithmetic Crossover

- Parents: $\langle x_1,\ldots, x_n \rangle$ and $\langle y_1,\ldots, y_n \rangle$
- $child_1$ is:

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

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



Swarm Intelligence

"The emergent collective intelligence of groups of simple agents"

[Bonabeu et at, 1999]

Swarm Intelligence

- 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

ACO metaheuristic

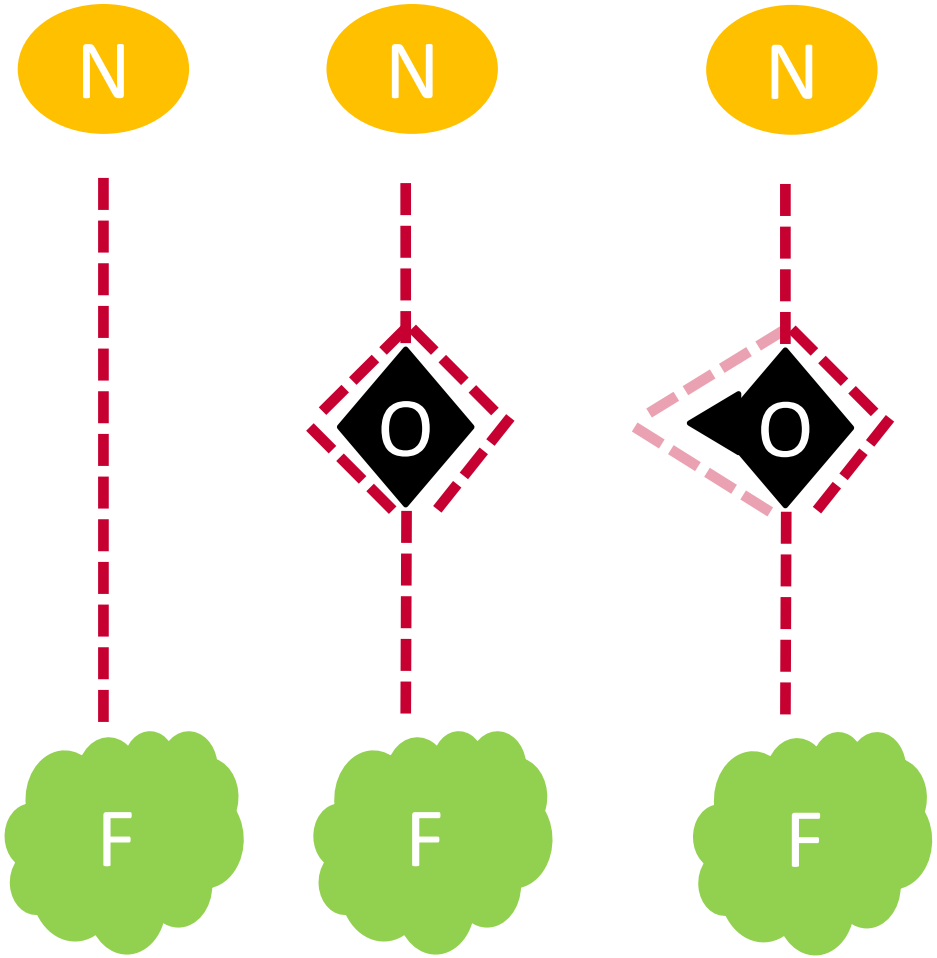
- Ants cooperate to find shortest paths to food sources
- A colony of *artificial* ants cooperate in finding good solutions
- Each ant – simple agent
- Ants communicate indirectly using chemicals ***(stigmergy)***
- Artificial ants have some local information and memory of where they've been

Denebourg's double bridge experiments

- Studied Argentine ants *I. humilis*
- Ants first had clear path from nest to food source

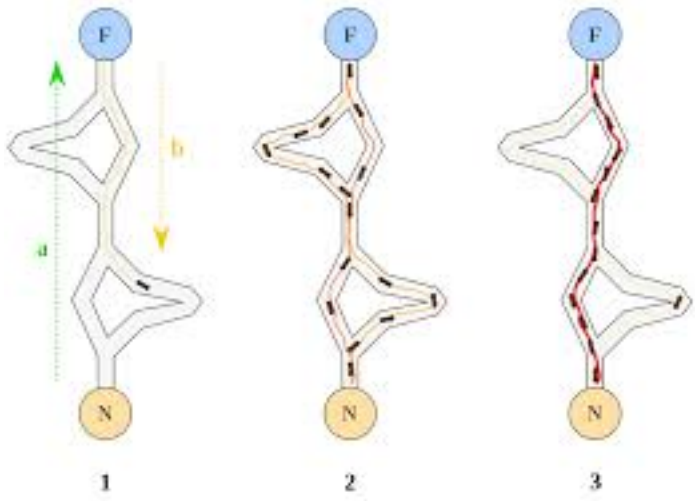
- Obstacle was subsequently place in the path, s.t. ants would have an equal distance to food if they went left or right around obstacle
  - Found roughly 50% of ants went one way, 50% the other

- 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



ACO System

- 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
  - Trail ***evaporates*** over time
  - 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)

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:

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

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} .

ACO: Global Pheromone update

- 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)

ACO: Local Pheromone update

- 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

ACO for TSP

Each ant *k*

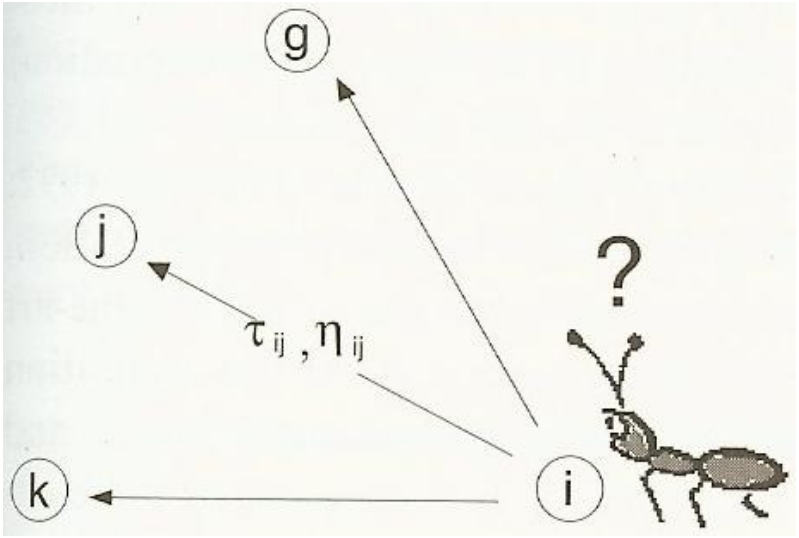
- Randomly selects a start node
- Chooses the next city on route with probability p^k_{ij} based on function of the distance from current city *i* to that city *j*, and the amount of pheromone on the edge between the two cities.
- Has memory, so only chooses cities that hasn't visited yet (i.e. always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

ACO for TSP

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

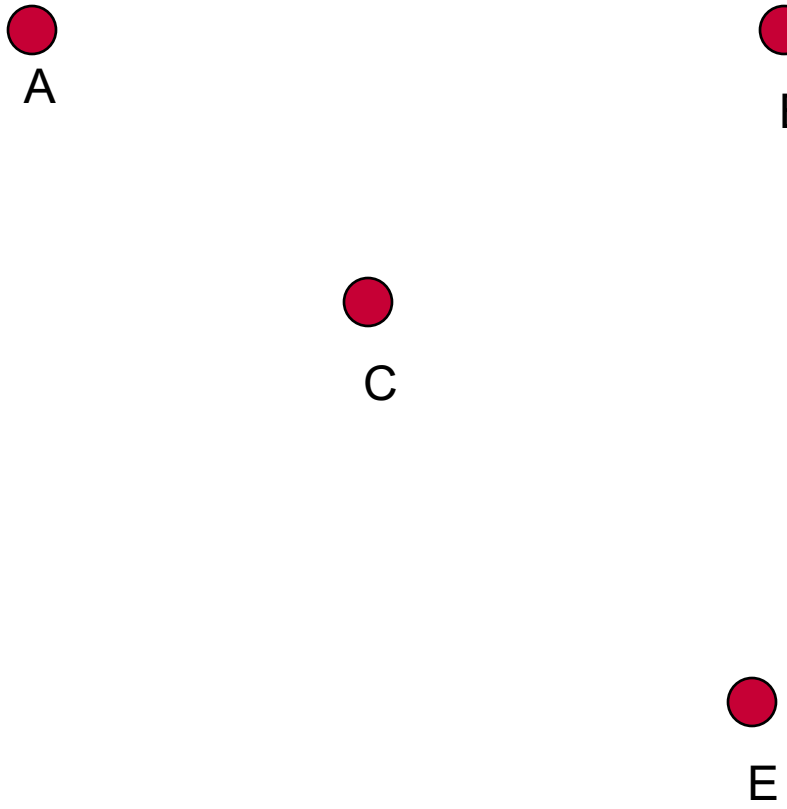
Tour Construction



1) Choose a start city.
2) Use pheromone and heuristic values to add cites until all have been visited.
3) Go back to the initial city.

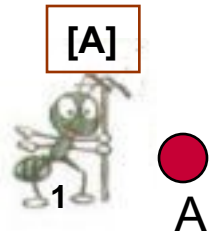
Simple TSP Example

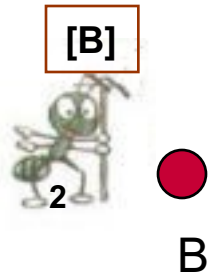
Nodes are not to scale at all!!!!

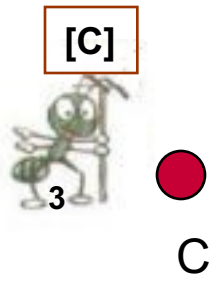


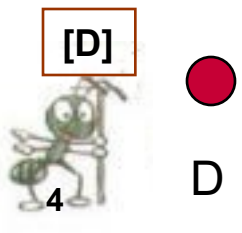
d_{AB} =100;d_{BC} = 60…;d_{DE} =150

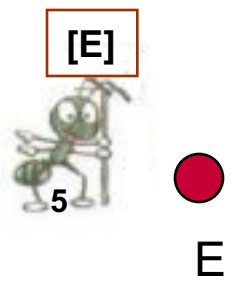
Node 1



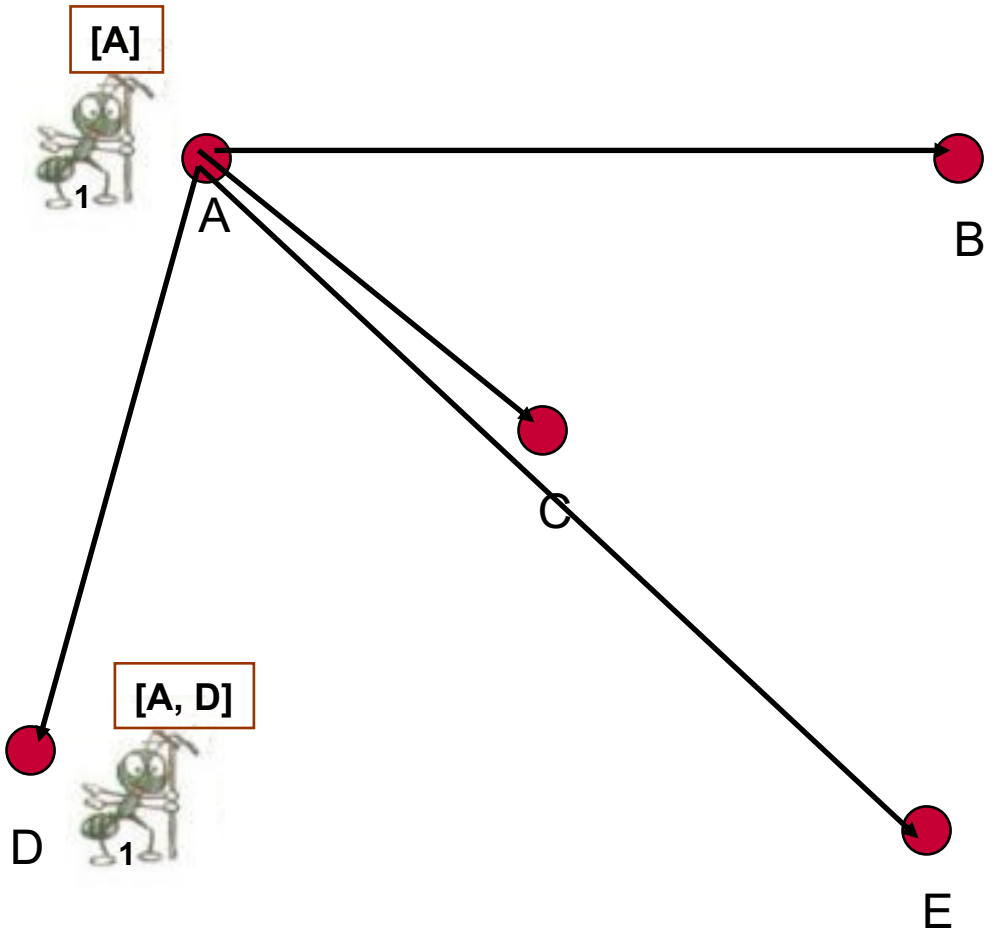








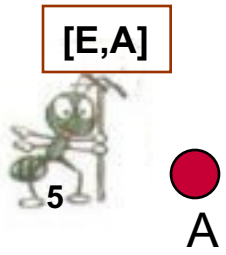
How to build next sub-solution?

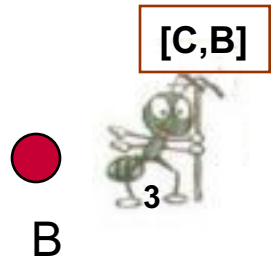


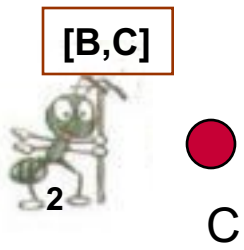
i/distance

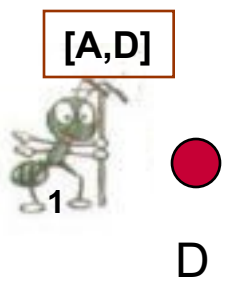
$$p_{ij}^{k}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{k \in allowed_k}[\tau_{ik}(t)]^{\alpha}[\eta_{ik}]^{\beta}} & \text{if } j \in \text{allowed}_k \\ 0 & \text{otherwise} \end{cases}$$

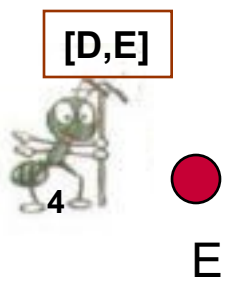
Node 2



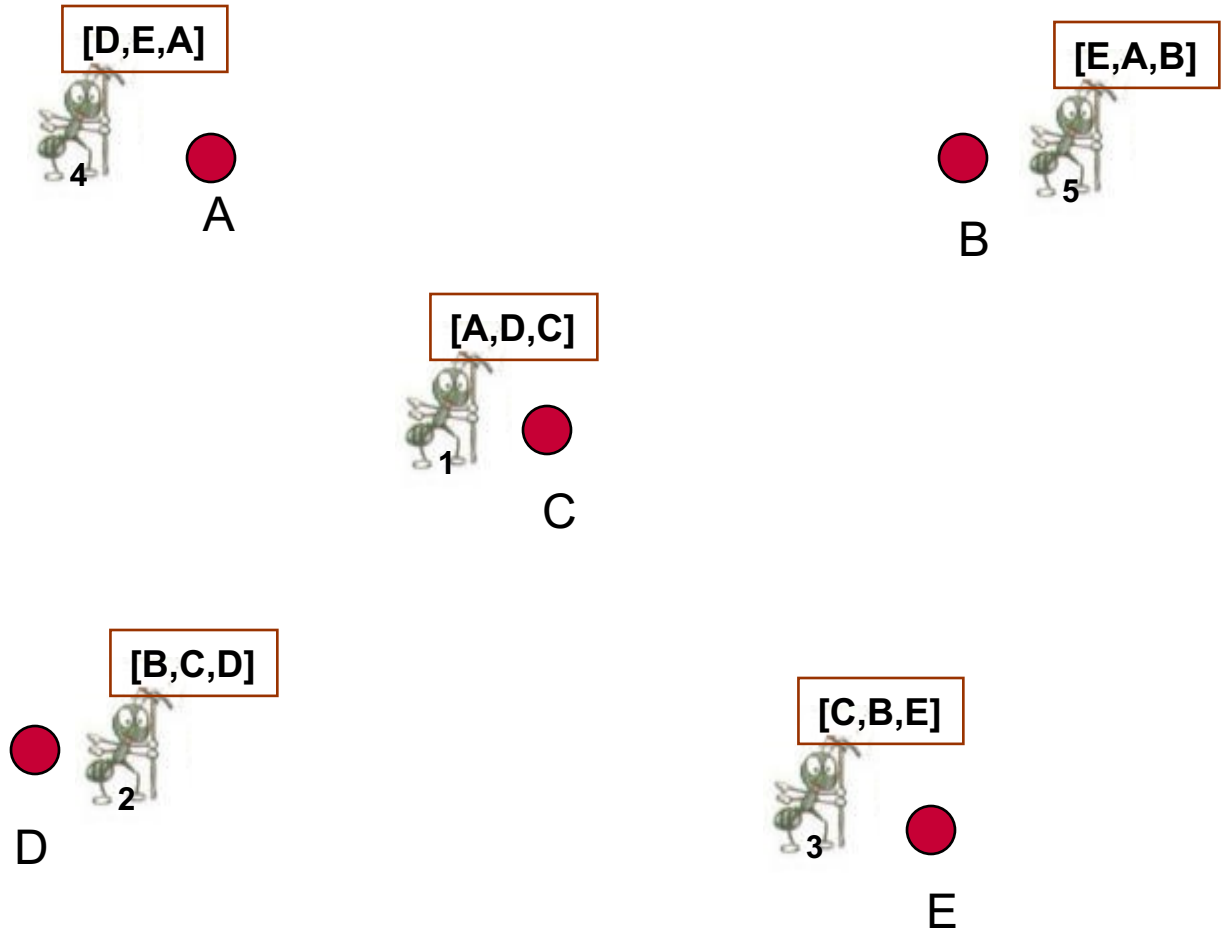




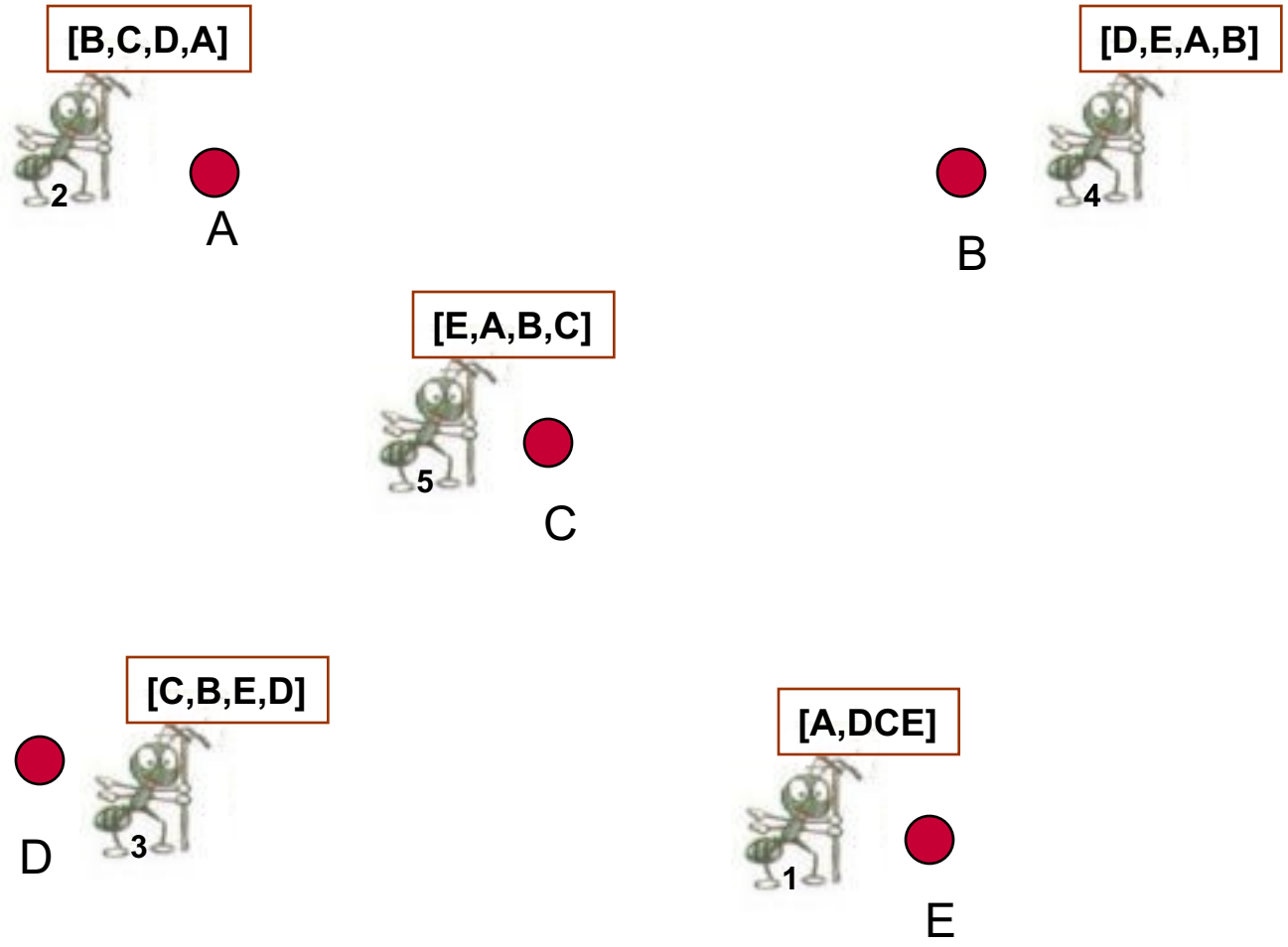




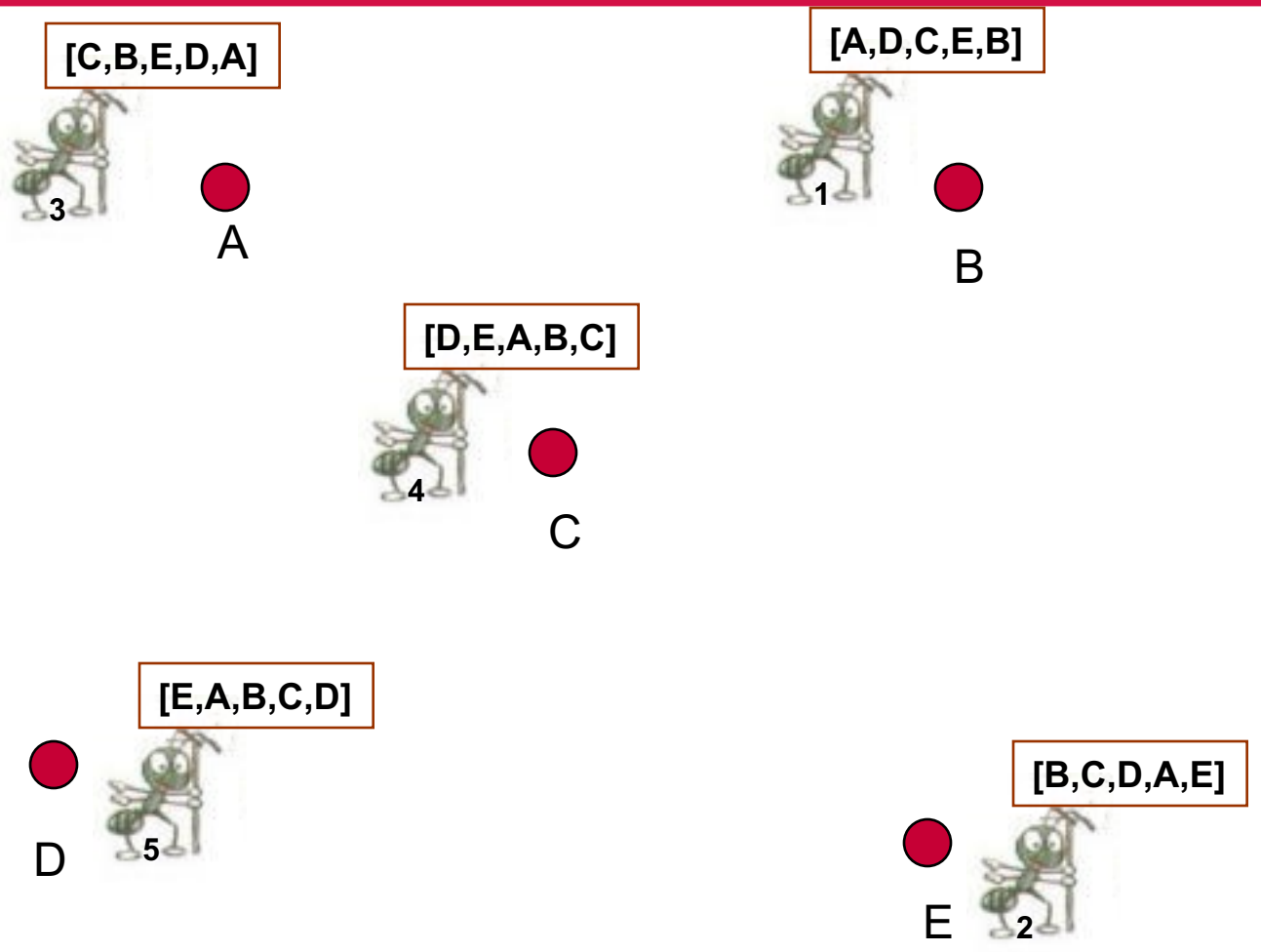
Node 3



Node 4



Node 5



Pheromones