Building an Evolutionary Algorithm

Prof. Emma Hart
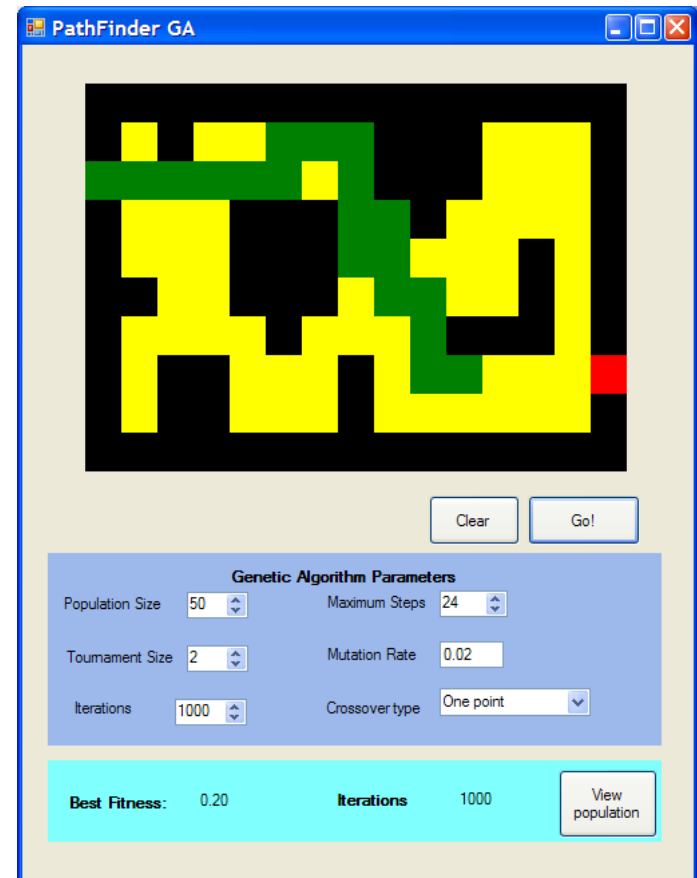
(Updated by Neil Urquhart 2013)

Today

- Detailed examples of using an EA
  - A Lunar Lander
  - Finding a path through a maze
- How to build an EA
  - Designing operators
  - Coding operators

Pathfinding in a maze

- Finding a way from a given start point to the exit in a maze
- (Could be done with A*)
- For big mazes with lots of states, EA might be faster

A Lunar Lander

- Example from AI Techniques for Games Programming
  – Mat Buckland
- Land a rocket using 4 controls:
  – Thrust
  – Rotate-left
 – Rotate-right
  – Do nothing (drift)

Evolution as search

- Evolution is searching through the space of possible solutions to find the best possible solution to a problem

- To search effectively, evolution needs to balance exploration and exploitation:

  – Exploration: look in unexplored regions of the search space

  – Exploitation: focus the search in a particular region of the search space

Search Spaces and Optima…
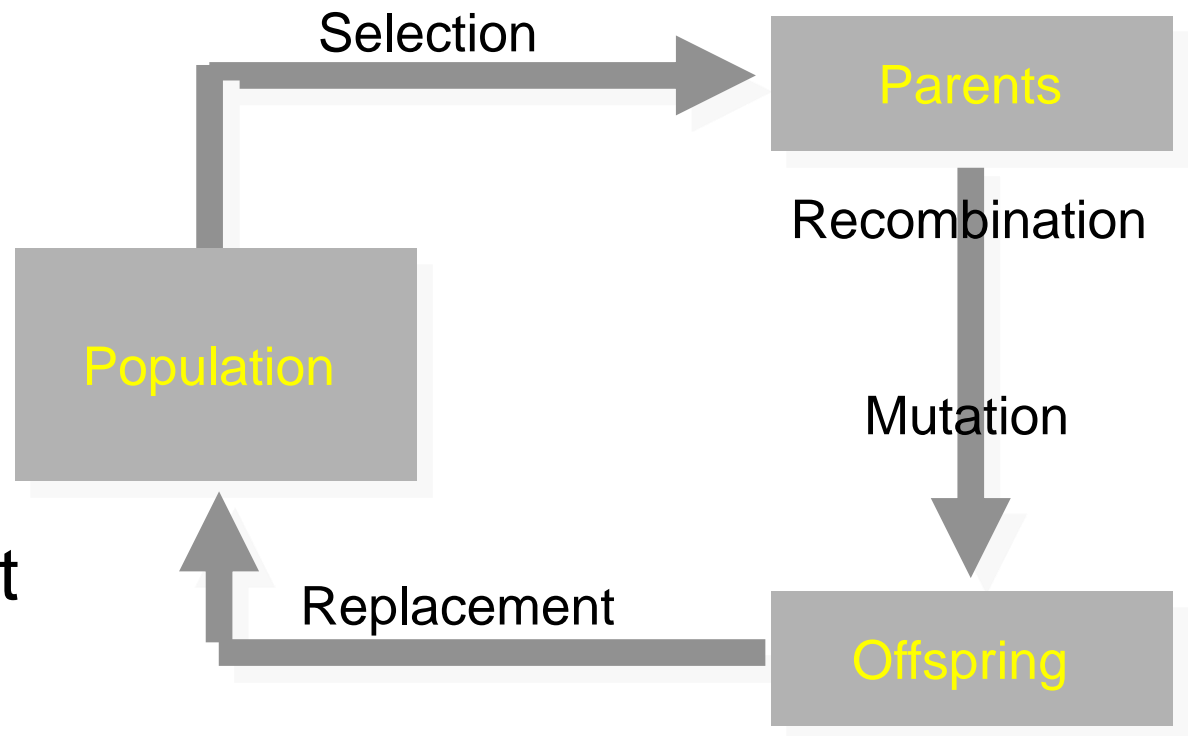

fitness

genotype space
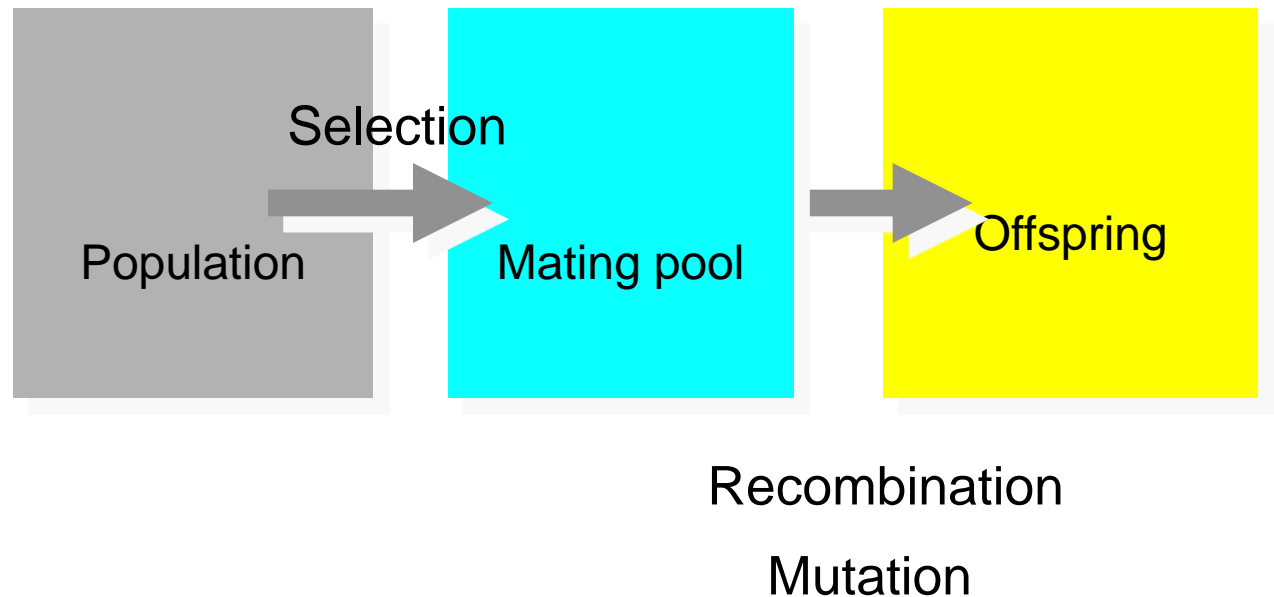
Fitness

Search Space
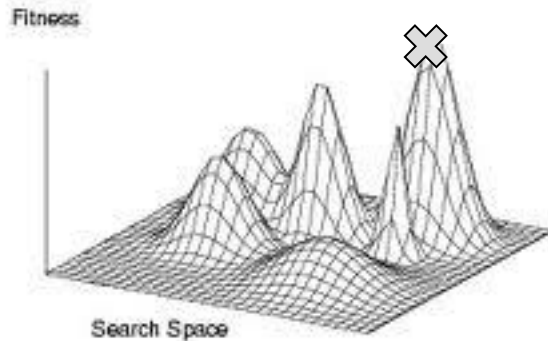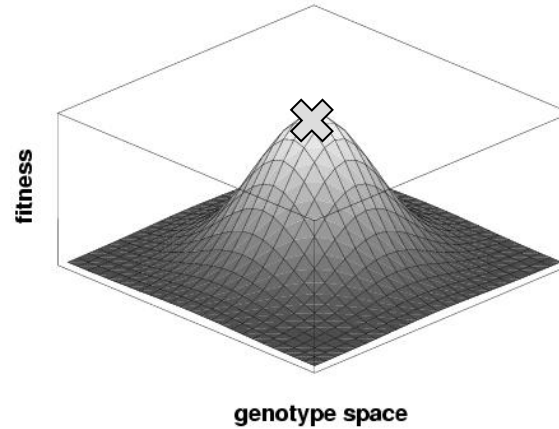
credit: S. Verel
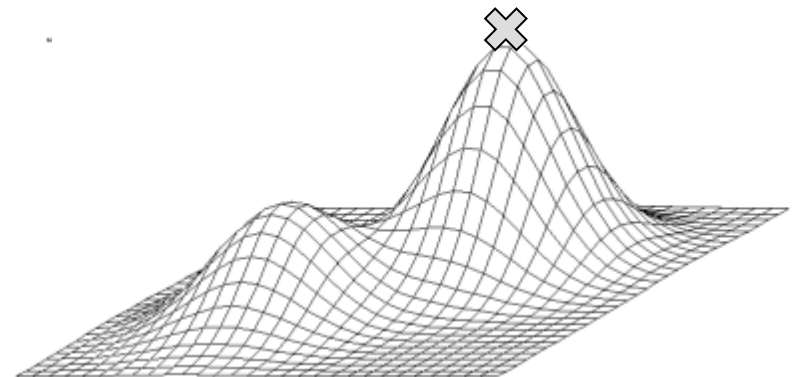
The Steady-State Evolutionary Cycle

- Representation
- Initialisation
- Fitness
- Selection
- Crossover
- Mutation
- Replacement

Generational Cycle:

- Representation
- Initialisation
- Fitness
- Selection
- Crossover
- Mutation
- Replacement

Selection

Population → Mating pool → Offspring

Recombination

Mutation

Reminder of terms

- A **chromosome** or **genotype** is a representation of a single potential solution to a problem
- Each chromosome consists of a number of **genes**
  - Each gene has a **value** (sometimes called an **allele)**
- The set of all possible chromosomes is called the **genotype space**

chromosome

| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

gene

gene value
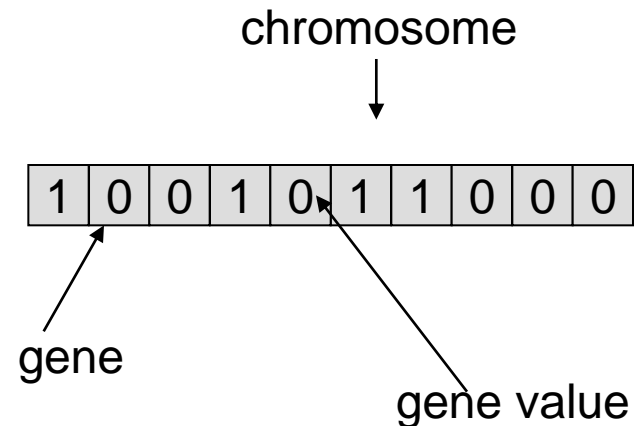
Representation

- To use an EA, you need to represent possible solutions as a chromosome:
- The general rule is to use a representation that is natural for the problem:
- Genes can be of any **type** (or a **mixture**)

– Binary strings

1	0	0	1	0	1	1	0	0	0

– Integer strings

1	4	2	1	7	6	3	8	1	2

– Real values

0.1	0.4	0.2	1.8	0.9	4.3	2.7	0.6	0.8	5.0

– Symbols

A	C	A	D	E	A	C	C	C	B

– Permutations

1	3	4	9	2	5	8	6	10	7

Representation: Pathfinding

- A solution is a possible path across the maze
  - E E S S E E E N W N N E E
- We can easily create a population of random solutions for a path of a given length:
  - S S W E S S W N S N N E E
  - W W W S N N E N W W N E
  - N E S N W E W N W N W W
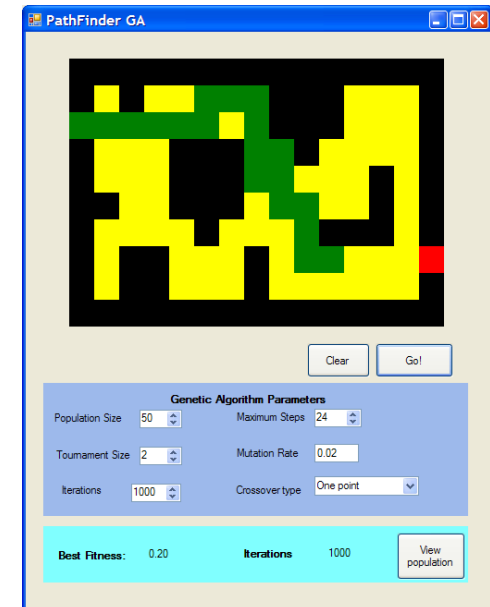- (most of them won't be very good!)
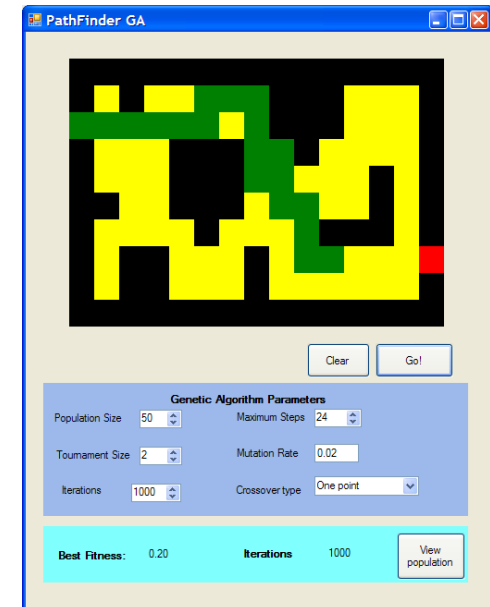
Representation: Pathfinding

- Often it's easier to work with numbers in code:

- We can use a **binary representation**: 2-bits to represent each possible direction:
  - 00 NORTH
  - 01 SOUTH
  - 10 EAST
  - 11 WEST

- If there are S steps, we need 2*S genes:

| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

East North East South East

Representation: Lunar Lander

- Four different control actions
  - 0   Thrust
  - 1   Rotate Left
  - 2   Rotate Right
  - 3   Drift (do nothing)
- Each one can be applied for a chosen length of time
- A **solution** is a sequence of actions and durations

A	D	A	D	A	D	A	D

Representation: Lunar Lander

- Assume a sequence of actions length A
- To create a solution, we randomly choose an action and a duration for each action in the sequence:
  - Actions: random number 0,1,2,3 (as there are 4 actions)
  - Duration: random number between 0-30 (max duration)
  - There will be 2* A genes in total in the chromosome (a potential solution)

3	10	**2**	5	**0**	27	**2**	14	**1**	19	**2**	21

Genotypes and Phenotypes

- A **genotype** is the chromosome used to represent the solution to a problem

    - *Or a method of constructing a solution to a problem*

- A **phenotype** is a solution to a problem;

- Sometimes genotype = phenotype

- Sometimes we need to map the *genotype* to the *phenotype*

Genotype

| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

Phenotype

East North East South East
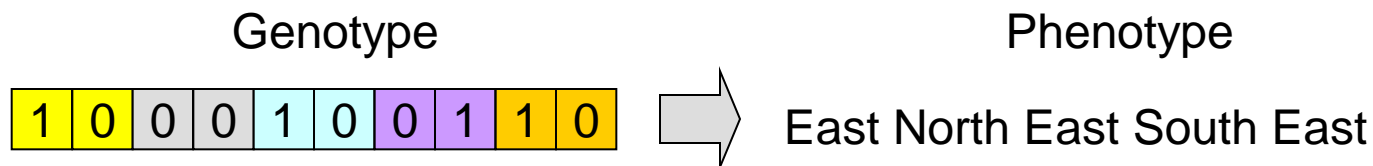
Writing an Evolutionary Algorithm

- We can represent a population of solutions with a 2d array:

int[][] = population[numChromo][lengthChromo]

Number of chromosomes in the population

Number of genes in each chromosome

Initialisation

- The initial population is usually initialised with a uniformly random distribution of genes:
    - Each gene value has a uniform chance of being selected for each gene

```
for (i=0;i<populationSize;i++)
  for (j=0;j<chromosomeLength;j++)
    population[i][j] = myRandom.Next(2);
```

Chromosome index   Gene index   For binary strings

Initialisation: Seeding the population

- Rather than create an initial random population we can seed the population with some good solutions
  - E.g obtained from heuristics/previous experience
- Advantage:
  - Population has higher starting fitness, could lead to faster evolution
- Disadvantage
  - Can lead to loss of diversity/premature convergence
  - Population might converge to a local optimum

Fitness

- The **fitness** of an individual is a measure of its quality:

  - hit points in a game

  - damage received or inflicted etc.

- How you define it depends on the game (or the problem you want to solve)

Example: Pathfinding in a maze

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)
(2,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)
(3,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)
(4,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)
(5,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)

EXIT

(10,3)

Fitness

- Minimise the distance from the exit
- Allow fitness to have values between 0 and 1:
    - The higher the value, the better the fitness
    - A perfect solution should have fitness = 1

$$Fitness = 1 / (DistX + DistY + 1)$$

$DistX = (exitX - endX)$
$DistY = (exitY - endY)$

Add 1 so that **FITNESS** is **maximum**
when distance is **minimum**

Fitness Example: Pathfinder

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)	
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)	
(2,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	
(3,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)
(4,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	
(5,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)	

DistX = (10-7) = 3   **Fitness** = 1/(3+0+1) = 1/4 = **0.25**

DistY = (3,3)=0

Example

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)	
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)	
(2,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	
(3,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)
(4,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	
(5,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)	

DistX = (10-9) = 1

DistY = (3,3)=0

Fitness = 1/(1+0+1) = 1/2 = **0.5**

Fitness Example: Pathfinder

(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)	(7,0)	(8,0)	(9,0)	
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)	(7,1)	(8,1)	(9,1)	
(2,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	
(3,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)
(4,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)	(7,4)	(8,4)	(9,4)	
(5,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)	(7,5)	(8,5)	(9,5)	

DistX = (10-10) = 0

DistY = (3,3)=0

Fitness = 1/(0+0+1) = 1/1 = **1.0**

Fitness: Lunar Lander

- Lander can only land if constraints are satisfied:
  - Distance from landing pad is within a certain limit
  - Velocity has to be below a certain limit
  - Rotation from vertical within limits

- We need a fitness function that encourages the EA to find values that tend towards these limits

- We can combine these objectives into a single fitness function

Fitness: Lunar Lander

- Calculate distance from pad: *fitDistance*
- Calculate speed: *fitSpeed*
- Calculate rotation from vertical: *fitRotate*

Fitness = fitDistance + fitSpeed + fitRotate

fitDistance = screenWidth – distanceFromPad

fitRotation = 1/(rotation + 1)

Fitness: Lunar Lander

- Calculate distance from pad: *fitDistance*

- Calculate speed: *fitSpeed*

- Calculate rotation from vertical: *fitRotate*

Fitness = fitDistance + fitSpeed + fitRotate

- BUT …we need to be careful:

  – We don't want the lander to be **too good**

Fitness: Lunar Lander

- Include **air time** to reward the lander for staying longer in the air:

$$fitAirTime = (time\ passed)/(speed+ 1)$$

Fitness = fitDistance + fitAirTime+ fitRotate

- All quantities increase as the solution get better

- But … there is still a problem….

Fitness: Lunar Lander

- The three quantities are measured on different scales:

  - Maximum distance fitness: 400

  - Maximum rotational fitness: 1

  - Maximum fitness from speed: 100

- We want to make them all contribute equally to the fitness:

Fitness = fitDistance + 4* fitAirTime+ 400*fitRotate

Selection

- Selection chooses individuals to become parents
  - Drives the evolution forward by biasing selection towards better individuals
- Selection is an **exploitation** operator
  - It exploits good solutions by choosing them to become parents
- **Selection pressure** is the pressure on the population to select good individuals for reproduction

Selection Pressure

- **High** selection pressure:

  – Good individuals selected for reproduction more often than weak

  – Can lead to faster evolution

  – But can also lead to **premature convergence**

- **Low** selection pressure:

  – Little pressure to select good individuals

  – Weak ones selected often

  – Evolution likely to progress very slowly/not at all in extreme

Selection: Tournament Selection

- Pick **k** individuals at random from the population

    - With or without replacement

- The best of the **k** individuals becomes the parent

- Varying k allows us to vary the **selection pressure**

Selection: Tournament Selection

```
//Pick k random individuals, with replacement
//Select the best of the k individuals to become a parent

for (i=0;i<tnSize;i++)
  potentialParent[i] = myRandom.Next(popSize);


bestFitness = fitness[potentialParent[0]];


for (i=1;i<tnSize;i++)
 if (fitness[potentialParent[i]] >bestFitness{
   bestFitness = fitness[potentialParent[i]] ;
   chosenParent = potentialParent[i];
}
```

Array stores index of potential parents

Array containing population fitness values

Index of potential parent

Tournament Selection

- The selection pressure can be controlled by varying the size of the tournament:

    - **High** tournament size: **increases** selection pressure

    - **Low** tournament size: **decreases** selection pressure

Selection: Fitness proportionate

- Choose parents with a probability proportional on their absolute fitness value compared to absolute fitness of population
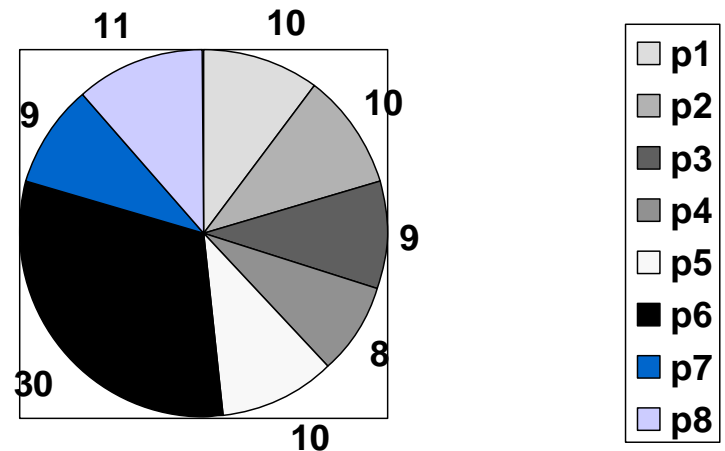
- Probability = $f_i/\Sigma f_j$

- Example:

  prob (p3) = 4/(10+5+4+1)

  $\qquad\qquad$ = 4/20

  $\qquad\qquad$ = 0.20
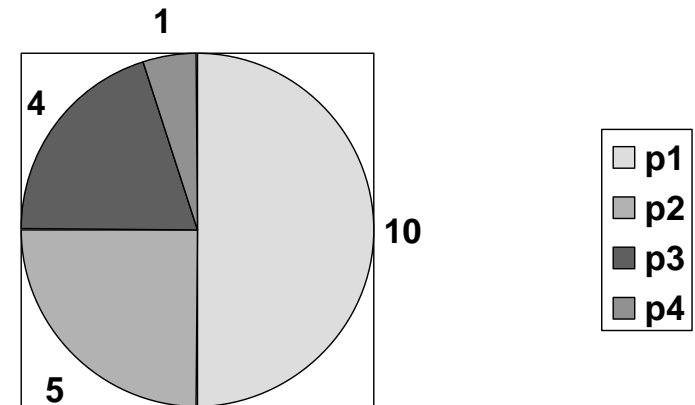
Selection: Fitness Proportionate

- There are some problems:
  - Favours outstanding individuals which can lead to premature convergence
 - If many chromosomes have similar values, there is very little selection pressure

Crossover

- Recombination operator:
  - Combines information from two parents together
- It is an **exploitation** operator
  - it utilises information already contained in the parents
- Biologically, crossover confined to 2 parents and unlimited children
- In artificial evolution, no reason to maintain this distinction
  - But generally restrict to 2 parents which give 1 or 2 children

Crossover: 1-point crossover

- Randomly choose a cut-point along the chromosome length
- Exchange the parts either side of the split
- Used in binary & integer representations
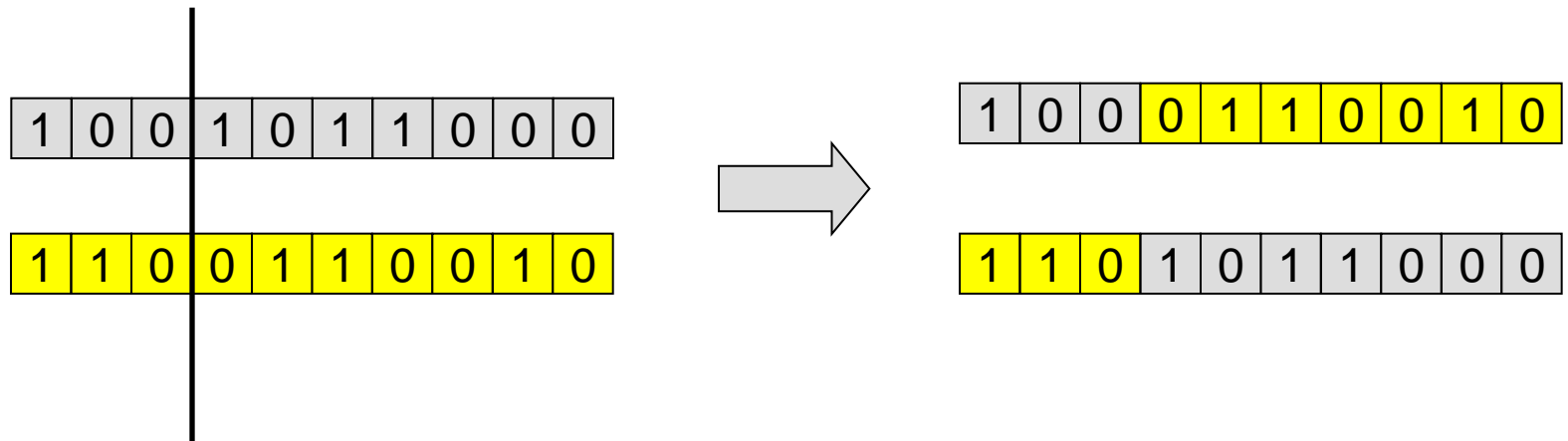
Crossover: code for 1pt crossover

```
// pick cut point
cutPoint  = myRandom.Next(chromosomeLength);

//create empty child array
child  =  new int[chromosomeLength];

// genes from parent1
for (i=0; i<cutPoint; i++)
  child[i] = population[parent1][i];

// and genes from parent2
for (i=cutPoint; i<chromosomeLength; i++)
  child[i] = population[parent2][i];
```
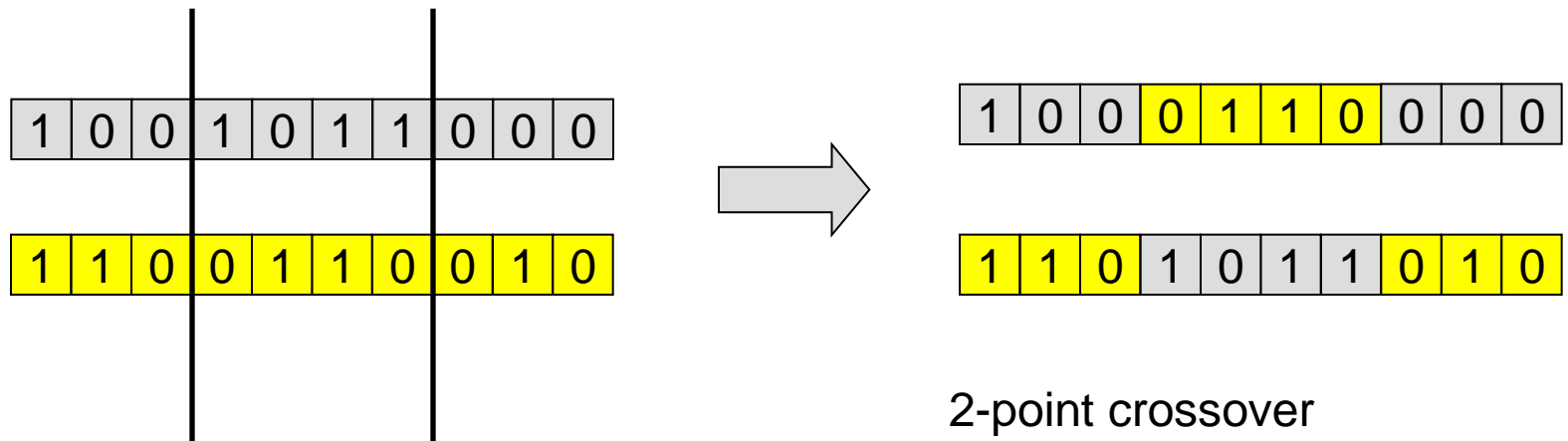
Crossover: n-point crossover

- Randomly choose **_n_** cut-points along the chromosome length
- Exchange the parts either sides of the split
- Used in binary & integer representations

2-point crossover

Crossover: uniform crossover

- Treat each gene independently
- Make a random choice which parent to inherit the gene from
- Used in binary & integer representations

Crossover: Arithmetic Crossover

- Useful for floating-point representations
- Take arithmetic average of gene-values at each position:

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9

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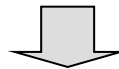
0.2	0.2	0.3	0.3	0.4	0.4	0.5	0.5	0.6

Recombination: Permutations

- We can't use any of these methods for permutation representation as they would result in an infeasible solution:

A	C	J	H	F	I	B

A	B	C	D	E	F	G		I	J

C	J	H	E	F	G	H	I	J

| B	C	D	F	I	B	G	E	D
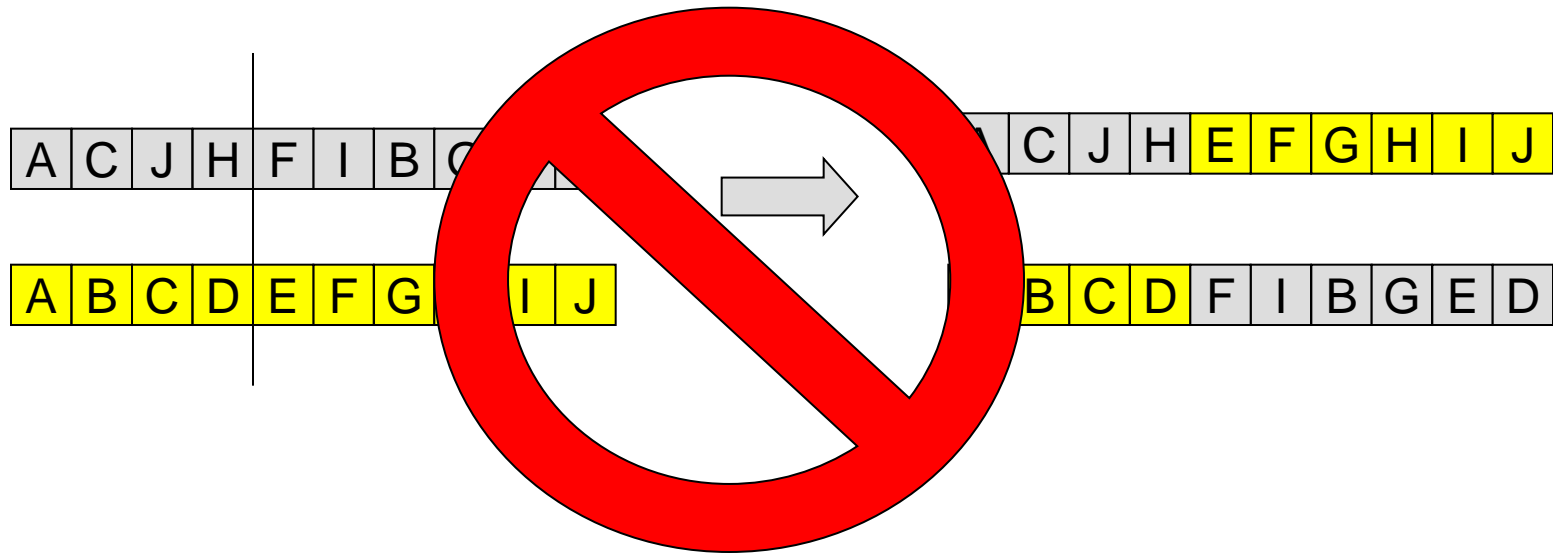
Recombination: Permutations

- Need to design special crossover operators:
  - Must result in feasible children
  - But also must inherit something from parents
  - Could inherit :
    - Ordering
    - Position
    - Edges
  - (Look up if you are interested)

Mutation

- Mutation is a variation operator

- Its role is to produce (slight) variants of existing solutions

- It often introduces new genes into the population

  - Therefore can increase diversity in the population
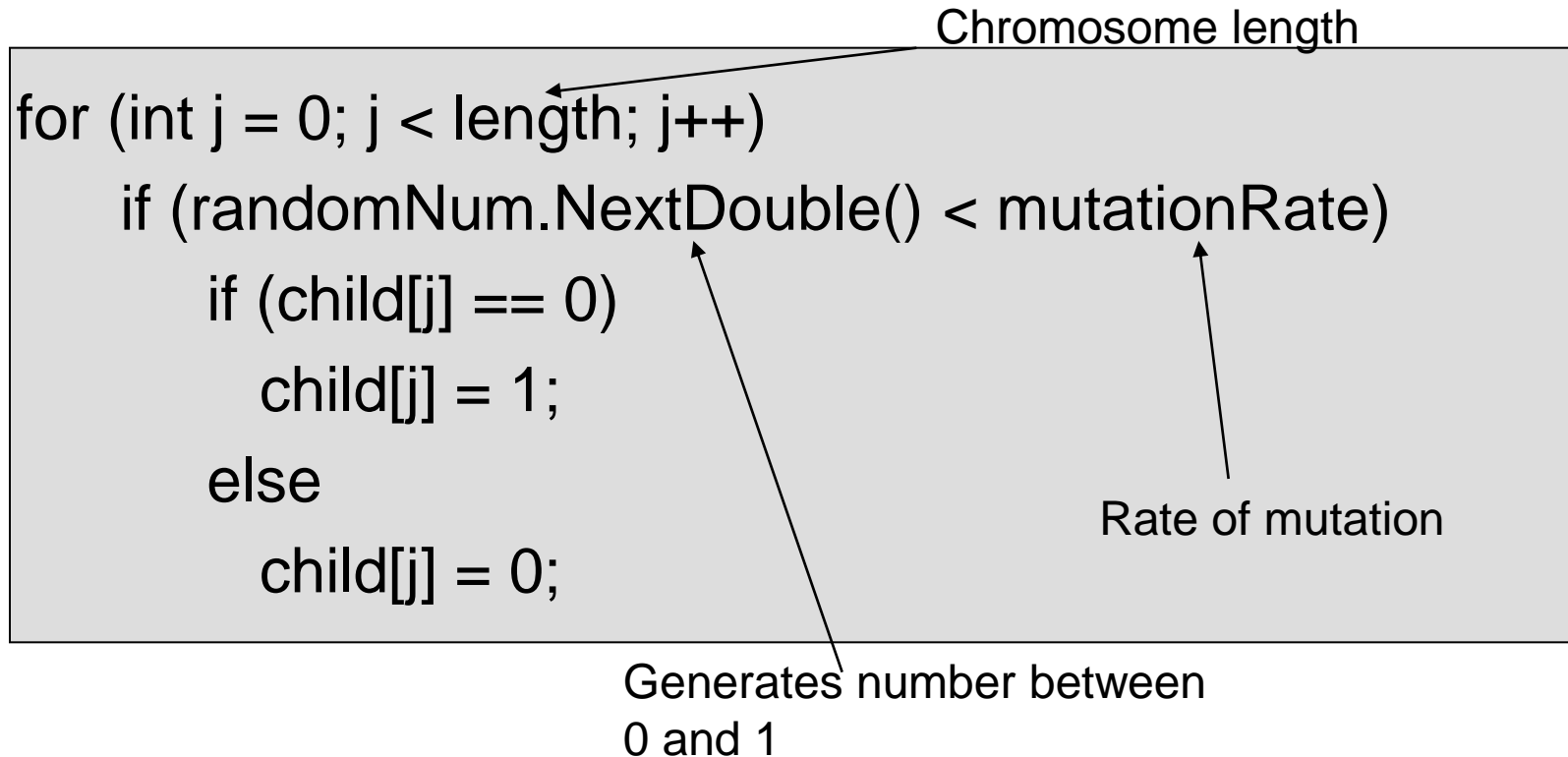
- It is an **exploration** operator

Mutation: Binary Representation

- Generally allow each bit to flip with a small probability p_m

- Often p_m set to be equal to 1/n where *n* is the chromosome length

| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

$\Longrightarrow$

| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
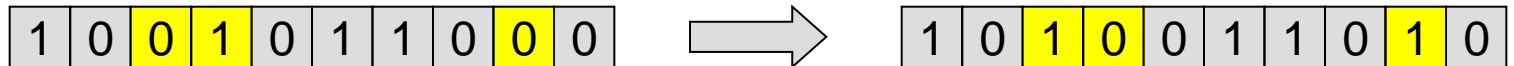
Mutation: Binary Representation

// mutate each gene with a small probability

Chromosome length

```
for (int j = 0; j < length; j++)
    if (randomNum.NextDouble() < mutationRate)
        if (child[j] == 0)
            child[j] = 1;
        else
            child[j] = 0;
```
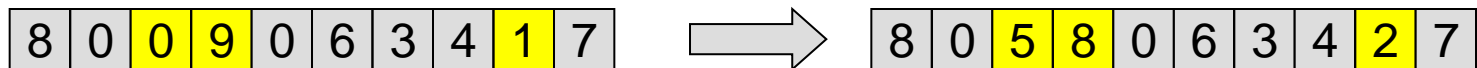
Rate of mutation

Generates number between
0 and 1

Mutation: Integer Representation

- With probability p_m mutate each gene to a value randomly chosen from the set of permissible values:

| 8 | 0 | 0 | 9 | 0 | 6 | 3 | 4 | 1 | 7 |

⟹

| 8 | 0 | 5 | 8 | 0 | 6 | 3 | 4 | 2 | 7 |

```
for (int j = 0; j < length; j++)
    if (randomNum.NextDouble()<mutationRate)
        child[j] = randNum.Next(10);
```
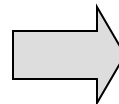
Mutation: Lunar Lander

```
for (j = 0; j < length-1; j+=2)
    // mutate the action
    if (randomNum.NextDouble()<mutationRate)
        child[j] = randNum.Next(MAX_ACTIONS);


for (j = 1; j < length; j+=2)
    // mutate the action
    if (randomNum.NextDouble()<mutationRate)
        child[j] = randNum.Next(MAX_DURATION);
```
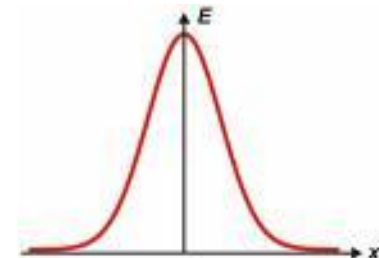
Could also use two different mutation rates

| 3 | 10 | 2 | 5 | 0 | 27 |

⟹

| 3 | 19 | 2 | 5 | 1 | 27 |

Mutation: Floating-point representations

- Assume values in the range [L,U]
- Uniform Mutation
  - With probability p_m mutate gene to a random value drawn from the range [L-U]
- Non-uniform
  - With probability p_m add a random number drawn from a Gaussian distribution with mean 1.0 and St.Dev SD to the current value

Mutation: Permutations

- Can't consider each gene independently as we would get an invalid solution

- The **mutation probability** now refers to whether or not the chromosome undergoes mutation

  - Not an individual gene as with integer & binary representations

```
if (myRandom.nextDouble() < mutRate){
        // perform a mutation to the chromosome
}
```

Replacement

- There are a number of ways of determining how to replace something in the population to make way for a new child:
  - Age-based replacement
  - Fitness-based replacement
    - Replace-worst
    - Tournament-replacement

  - Random-replacement

Replacement: Age-Based & Random

- Replace the oldest member of the population
  - i.e. the one that was created the longest time ago
- Replace a random **parent**:
  - But could randomly replace the best member!
- Replace a **random member** of the population
  - But same disadvantage as above

Replacement: Fitness based mechanisms

- Replace WORST
  - Can lead to rapid population improvement
  - But can also lead to premature convergence
  - Good in big populations
- Replace using tournament
  - As with selection, hold a tournament
  - Winner of the tournament is the chromosome with lowest fitness
  - As tournament size increases, increase the pressure to replace the worst individual

Summary

- We have described the components of an evolutionary algorithm:

  - Representations

  - Variation operators

  - Fitness measurements

- There is no "one" evolutionary algorithm

  - The right operators and representation must be determined for each application