Evolutionary Computing
Lecture 2
Genetic Algorithms for Optimization

Danylo Tavrov

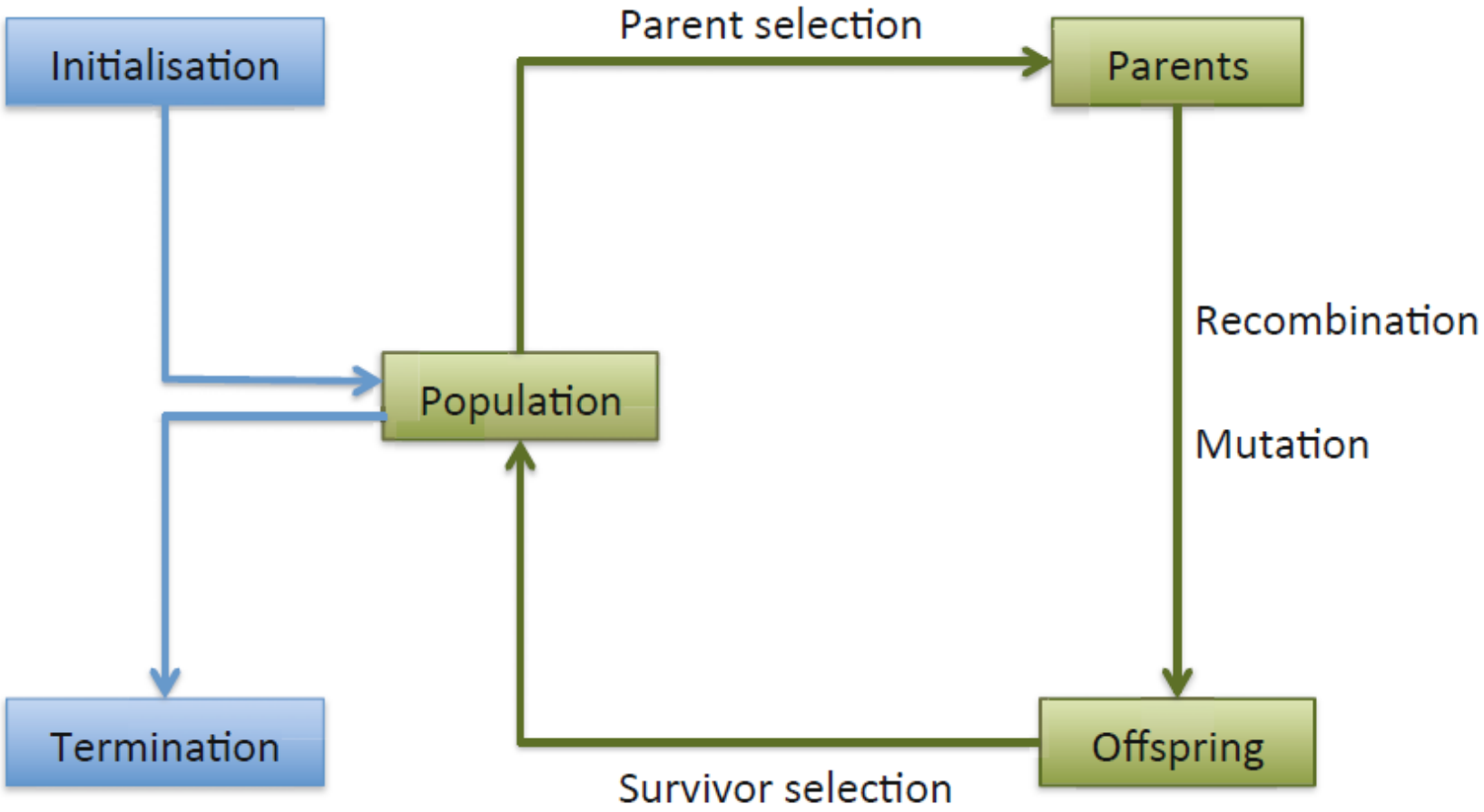
Outline

1. Applying Genetic Algorithms to Real-Valued Optimization
2. Beyond Fitness-Proportionate Selection
3. Ideas for Multimodal Optimization
4. Beyond Binary Representations
5. Parameter Control in Evolutionary Algorithms
6. Constraint Handling

Outline

1. Applying Genetic Algorithms to Real-Valued Optimization
2. Beyond Fitness-Proportionate Selection
3. Ideas for Multimodal Optimization
4. Beyond Binary Representations
5. Parameter Control in Evolutionary Algorithms
6. Constraint Handling

General Scheme of an Evolutionary Algorithm



Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 3.2

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Choices to Be Made

- To implement an evolutionary algorithm for a given problem, one needs to make a number of choices
- Representation and fitness function
- Type of crossover *applicable to selected representation*
- Type of mutation *applicable to selected representation*
- Population model
- Type of selection
- Initialization procedure
- Termination condition(s)
- Numerical parameters of variation operators (population size, offspring number, crossover probability, mutation probability, etc.)

Representation for Real-Valued Optimization

- In this lecture, we will consider the problem of **parameter optimization**
- The task is to find a vector of parameters $\mathbf{x}$ such that $g(\mathbf{x})$ is **optimal** (minimal or maximal, depending on the problem)
- To even begin solving such problems, one needs to answer two questions
- How to **represent** real numbers with binary chromosomes?
- How to connect a **fitness function** f and **objective function** g to each other?

Representation for Real-Valued Optimization

- In this lecture, we will consider the problem of **parameter optimization**
- The task is to find a vector of parameters $\mathbf{x}$ such that $g(\mathbf{x})$ is **optimal** (minimal or maximal, depending on the problem)
- To even begin solving such problems, one needs to answer two questions
- How to **represent** real numbers with binary chromosomes?
- How to connect a **fitness function** f and **objective function** g to each other?

Representation for Real-Valued Optimization

- In this lecture, we will consider the problem of **parameter optimization**
- The task is to find a vector of parameters $\mathbf{x}$ such that $g(\mathbf{x})$ is **optimal** (minimal or maximal, depending on the problem)
- To even begin solving such problems, one needs to answer two questions
- How to **represent** real numbers with binary chromosomes?
- How to connect a **fitness function** f and **objective function** g to each other?
  - Recall that fitness needs to be positive and should be maximized
  - On the other hands, objective functions can be negative and hopefully need to be minimized

Representation for Real-Valued Optimization

- In this lecture, we will consider the problem of **parameter optimization**
- The task is to find a vector of parameters $\mathbf{x}$ such that $g(\mathbf{x})$ is **optimal** (minimal or maximal, depending on the problem)
- To even begin solving such problems, one needs to answer two questions
- How to **represent** real numbers with binary chromosomes?
- How to connect a **fitness function** f and **objective function** g to each other?
  - Recall that fitness needs to be positive and should be maximized
  - On the other hands, objective functions can be negative and typically need to be minimized

Representation for Real-Valued Optimization

- In this lecture, we will consider the problem of **parameter optimization**
- The task is to find a vector of parameters $\mathbf{x}$ such that $g(\mathbf{x})$ is **optimal** (minimal or maximal, depending on the problem)
- To even begin solving such problems, one needs to answer two questions
- How to **represent** real numbers with binary chromosomes?
- How to connect a **fitness function** f and **objective function** g to each other?
  - Recall that fitness needs to be positive and should be maximized
  - On the other hands, objective functions can be negative and typically need to be minimized

Representation for Real-Valued Optimization

- In this lecture, we will consider the problem of **parameter optimization**
- The task is to find a vector of parameters $\mathbf{x}$ such that $g(\mathbf{x})$ is **optimal** (minimal or maximal, depending on the problem)
- To even begin solving such problems, one needs to answer two questions
- How to **represent** real numbers with binary chromosomes?
- How to connect a **fitness function** f and **objective function** g to each other?
  - Recall that fitness needs to be positive and should be maximized
  - On the other hands, objective functions can be negative and typically need to be minimized

Representation for Real-Valued Optimization

- In this lecture, we will consider the problem of **parameter optimization**
- The task is to find a vector of parameters $\mathbf{x}$ such that $g(\mathbf{x})$ is **optimal** (minimal or maximal, depending on the problem)
- To even begin solving such problems, one needs to answer two questions
- How to **represent** real numbers with binary chromosomes?
- How to connect a **fitness function** f and **objective function** g to each other?
  - Recall that fitness needs to be positive and should be maximized
  - On the other hands, objective functions can be negative and typically need to be minimized

Representation of Real Numbers

- **One way** to represent real numbers, which is native to GA, is using binary strings
- In a general case, we need to code n real parameters limited to interval $[a; b]$
- For each parameter we use l bits
- Then the binary string of all 0's is mapped onto a
- The binary string of all 1's is mapped onto b
- And all other numbers are linearly mapped onto values in between
- Clearly, l determines precision:

$$\text{precision} = \frac{b-a}{2^l - 1}$$

Representation of Real Numbers

- **One way** to represent real numbers, which is native to GA, is using binary strings
- In a general case, we need to code n real parameters limited to interval $[a; b]$
- For each parameter we use l bits
- Then the binary string of all 0's is mapped onto a
- The binary string of all 1's is mapped onto b
- And all other numbers are linearly mapped onto values in between
- Clearly, l determines precision:

$$\text{precision} = \frac{b-a}{2^l - 1}$$

Representation of Real Numbers

- **One way** to represent real numbers, which is native to GA, is using binary strings
- In a general case, we need to code n real parameters limited to interval $[a; b]$
- For each parameter we use l bits
- Then the binary string of all 0's is mapped onto a
- The binary string of all 1's is mapped onto b
- And all other numbers are linearly mapped onto values in between
- Clearly, l determines precision:

$$\text{precision} = \frac{b-a}{2^l - 1}$$

Representation of Real Numbers

- **One way** to represent real numbers, which is native to GA, is using binary strings
- In a general case, we need to code n real parameters limited to interval $[a; b]$
- For each parameter we use l bits
- Then the binary string of all 0's is mapped onto a
- The binary string of all 1's is mapped onto b
- And all other numbers are linearly mapped onto values in between
- Clearly, l determines precision:

$$\text{precision} = \frac{b-a}{2^l - 1}$$

Representation of Real Numbers

- **One way** to represent real numbers, which is native to GA, is using binary strings
- In a general case, we need to code n real parameters limited to interval $[a; b]$
- For each parameter we use l bits
- Then the binary string of all 0's is mapped onto a
- The binary string of all 1's is mapped onto b
- And all other numbers are linearly mapped onto values in between
- Clearly, l determines precision:

$$\text{precision} = \frac{b-a}{2^l - 1}$$

Representation of Real Numbers

- **One way** to represent real numbers, which is native to GA, is using binary strings
- In a general case, we need to code n real parameters limited to interval $[a; b]$
- For each parameter we use l bits
- Then the binary string of all 0's is mapped onto a
- The binary string of all 1's is mapped onto b
- And all other numbers are linearly mapped onto values in between
- Clearly, l determines precision:

$$\text{precision} = \frac{b-a}{2^l - 1}$$

Representation of Real Numbers

- **One way** to represent real numbers, which is native to GA, is using binary strings
- In a general case, we need to code n real parameters limited to interval $[a; b]$
- For each parameter we use l bits
- Then the binary string of all 0's is mapped onto a
- The binary string of all 1's is mapped onto b
- And all other numbers are linearly mapped onto values in between
- Clearly, l determines precision:

$$\text{precision} = \frac{b - a}{2^l - 1}$$

Representation of Real Numbers

- For example, suppose we want to encode 3 parameters in $[-1; 2]$ and use $l = 5$ bits for each of them
- Then we can have a chromosome

$$10011 \mid 01100 \mid 01000$$

- Since $00000 \to -1$ and $11111 \to 2$, we compute

$$10011 \to \frac{19}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.839$$

$$01100 \to \frac{12}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.161$$

$$01000 \to \frac{8}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx -0.516$$

Representation of Real Numbers

- For example, suppose we want to encode 3 parameters in $[-1; 2]$ and use $l = 5$ bits for each of them
- Then we can have a chromosome

$$10011 \mid 01100 \mid 01000$$

- Since $00000 \to -1$ and $11111 \to 2$, we compute

$$10011 \to \frac{19}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.839$$

$$01100 \to \frac{12}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.161$$

$$01000 \to \frac{8}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx -0.226$$

Representation of Real Numbers

- For example, suppose we want to encode 3 parameters in $[-1; 2]$ and use $l = 5$ bits for each of them
- Then we can have a chromosome

$$10011 \mid 01100 \mid 01000$$

- Since $00000 \to -1$ and $11111 \to 2$, we compute

$$10011 \to \frac{19}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.839$$

$$01100 \to \frac{12}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.161$$

$$01000 \to \frac{8}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx -0.226$$

Representation of Real Numbers

- For example, suppose we want to encode 3 parameters in $[-1; 2]$ and use $l = 5$ bits for each of them
- Then we can have a chromosome

$$10011 \mid 01100 \mid 01000$$

- Since $00000 \to -1$ and $11111 \to 2$, we compute

$$10011 \to \frac{19}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.839$$

$$01100 \to \frac{12}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.161$$

$$01000 \to \frac{8}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx -0.226$$

Representation of Real Numbers

- For example, suppose we want to encode 3 parameters in $[-1; 2]$ and use $l = 5$ bits for each of them
- Then we can have a chromosome

$$10011 \mid 01100 \mid 01000$$

- Since $00000 \to -1$ and $11111 \to 2$, we compute

$$10011 \to \frac{19}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.839$$
$$01100 \to \frac{12}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx 0.161$$
$$01000 \to \frac{8}{2^5 - 1} \cdot (2 - (-1)) + (-1) \approx -0.226$$

Mapping Objective Function onto Fitness Function

- For minimization problems, we can use

$$f(\mathbf{x}) = \begin{cases} C_{\max} - g(\mathbf{x}) \ , & g(\mathbf{x}) < C_{\max} \\ 0 \ , & \text{otherwise} \end{cases}$$

- For maximization problems, we can use

$$f(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - C_{\min} \ , & g(\mathbf{x}) > C_{\min} \\ 0 \ , & \text{otherwise} \end{cases}$$

- Here, $C_{\max}$ ($C_{\min}$) can be taken as:
  - The largest (smallest) value possible on principle (may or may not be known in advance)
  - The largest (smallest) value of the objective function observed in the current population
  - The largest (smallest) value of the objective function observed in the last few

Mapping Objective Function onto Fitness Function

- For minimization problems, we can use

$$f(\mathbf{x}) = \begin{cases} C_{\max} - g(\mathbf{x}) \ , & g(\mathbf{x}) < C_{\max} \\ 0 \ , & \text{otherwise} \end{cases}$$

- For maximization problems, we can use

$$f(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - C_{\min} \ , & g(\mathbf{x}) > C_{\min} \\ 0 \ , & \text{otherwise} \end{cases}$$

- Here, $C_{\max}$ ($C_{\min}$) can be taken as:
  - The largest (smallest) value possible *in principle* (may or may not be known in advance)
  - The largest (smallest) value of the objective function observed in the current population
  - The largest (smallest) value of the objective function observed so far (i.e., during the

Mapping Objective Function onto Fitness Function

- For minimization problems, we can use

$$f(\mathbf{x}) = \begin{cases} C_{\max} - g(\mathbf{x}) \ , & g(\mathbf{x}) < C_{\max} \\ 0 \ , & \text{otherwise} \end{cases}$$

- For maximization problems, we can use

$$f(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - C_{\min} \ , & g(\mathbf{x}) > C_{\min} \\ 0 \ , & \text{otherwise} \end{cases}$$

- Here, $C_{\max}$ ($C_{\min}$) can be taken as:
  - The largest (smallest) value possible *in principle* (may or may not be known in advance)
  - The largest (smallest) value of the objective function observed in the current population
  - The largest (smallest) value of the objective function observed so far (**the best**)

Mapping Objective Function onto Fitness Function

- For minimization problems, we can use

$$f(\mathbf{x}) = \begin{cases} C_{\max} - g(\mathbf{x}) \ , & g(\mathbf{x}) < C_{\max} \\ 0 \ , & \text{otherwise} \end{cases}$$

- For maximization problems, we can use

$$f(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - C_{\min} \ , & g(\mathbf{x}) > C_{\min} \\ 0 \ , & \text{otherwise} \end{cases}$$

- Here, $C_{\max}$ ($C_{\min}$) can be taken as:
  - The largest (smallest) value possible *in principle* (may or may not be known in advance)
  - The largest (smallest) value of the objective function observed in the current population
  - The largest (smallest) value of the objective function observed so far (**the best**)

Mapping Objective Function onto Fitness Function

- For minimization problems, we can use

$$f(\mathbf{x}) = \begin{cases} C_{\max} - g(\mathbf{x}) \ , & g(\mathbf{x}) < C_{\max} \\ 0 \ , & \text{otherwise} \end{cases}$$

- For maximization problems, we can use

$$f(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - C_{\min} \ , & g(\mathbf{x}) > C_{\min} \\ 0 \ , & \text{otherwise} \end{cases}$$

- Here, $C_{\max}$ ($C_{\min}$) can be taken as:
  - The largest (smallest) value possible *in principle* (may or may not be known in advance)
  - The largest (smallest) value of the objective function observed in the current population
  - The largest (smallest) value of the objective function observed so far (**the best**)

Mapping Objective Function onto Fitness Function

- For minimization problems, we can use

$$f(\mathbf{x}) = \begin{cases} C_{\max} - g(\mathbf{x}) \ , & g(\mathbf{x}) < C_{\max} \\ 0 \ , & \text{otherwise} \end{cases}$$

- For maximization problems, we can use

$$f(\mathbf{x}) = \begin{cases} g(\mathbf{x}) - C_{\min} \ , & g(\mathbf{x}) > C_{\min} \\ 0 \ , & \text{otherwise} \end{cases}$$

- Here, $C_{\max}$ ($C_{\min}$) can be taken as:
  - The largest (smallest) value possible *in principle* (may or may not be known in advance)
  - The largest (smallest) value of the objective function observed in the current population
  - The largest (smallest) value of the objective function observed so far (**the best**)

Other Crossovers for GA

- **n-point crossover**:
  1) Choose n random integers in $[1; l-1]$
  2) Split the parents along these points
  3) Glue parts, *alternating* between parents
- Has **positional bias**: genes located close to each other have tendency to be kept together

Other Crossovers for GA

- **n-point crossover**:
  1) Choose n random integers in $[1; l-1]$
  2) Split the parents along these points
  3) Glue parts, *alternating* between parents
- Has **positional bias**: genes located close to each other have tendency to be kept together

Other Crossovers for GA

- **n-point crossover**:
  1) Choose n random integers in $[1; l-1]$
  2) Split the parents along these points
  3) Glue parts, *alternating* between parents
- Has **positional bias**: genes located close to each other have tendency to be kept together

Other Crossovers for GA

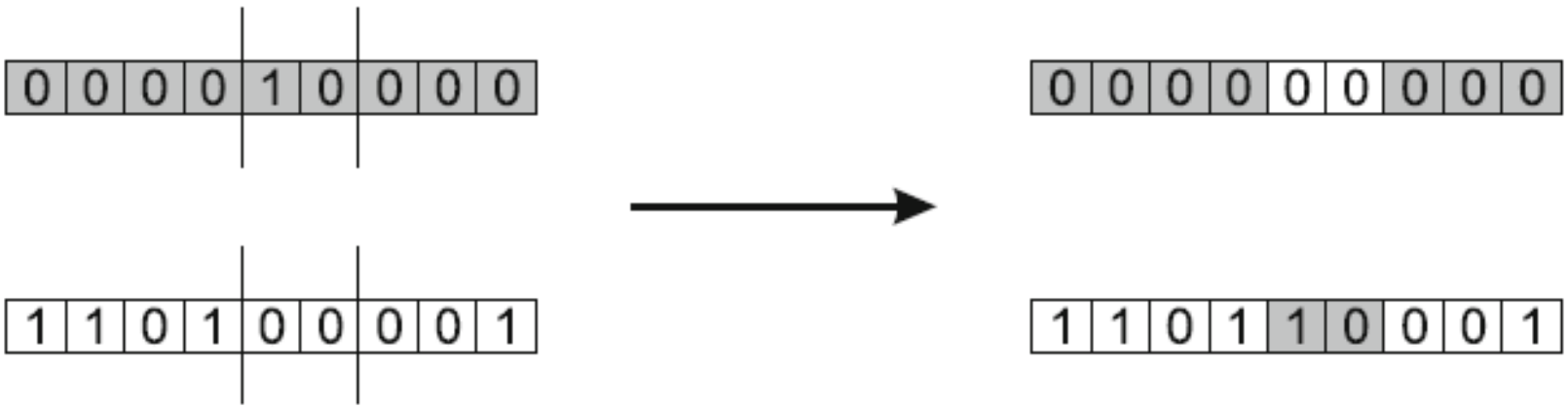
- **n-point crossover**:
  1) Choose n random integers in $[1; l-1]$
  2) Split the parents along these points
  3) Glue parts, *alternating* between parents
- Has **positional bias**: genes located close to each other have tendency to be kept together

Other Crossovers for GA

- **n-point crossover**:
  1) Choose n random integers in $[1; l-1]$
  2) Split the parents along these points
  3) Glue parts, *alternating* between parents
- Has **positional bias**: genes located close to each other have tendency to be kept together

Other Crossovers for GA

- **n-point crossover**:
  1) Choose n random integers in $[1; l-1]$
  2) Split the parents along these points
  3) Glue parts, *alternating* between parents
- Has **positional bias**: genes located close to each other have tendency to be kept together



Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.2

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

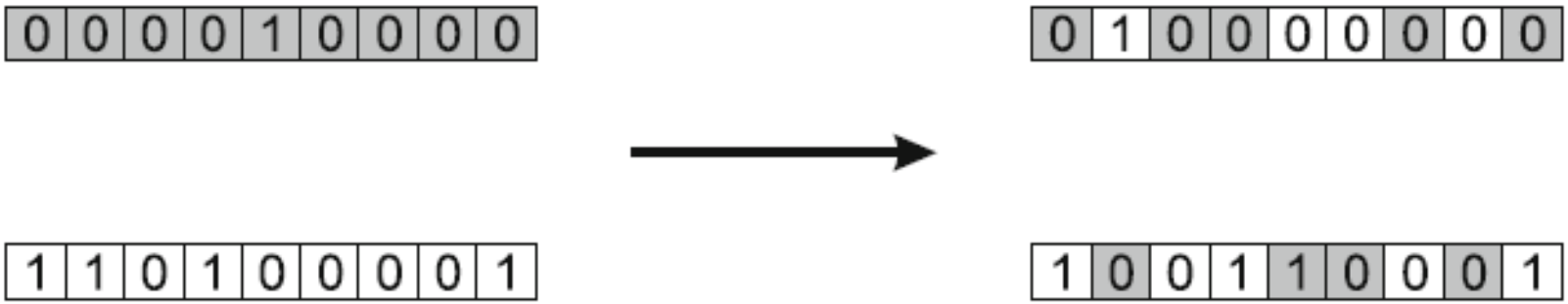
Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Other Crossovers for GA

- **Uniform crossover**[1]:
  1) Generate l random numbers in $[0; 1]$ (where l is the total length of the chromosome)
  2) For each gene i, decide whether ith number is below 0.5
  3) If yes, copy the gene from parent 1 to child 1, and from parent 2 to child 2
  4) Otherwise, copy the gene from parent 1 to child 2, and from parent 2 to child 1
- Does *not* have positional bias
- Has **distributional bias**: has a tendency against transmitting a large number of coadapted genes from one parent to an offspring
- Choice of a crossover depends on a problem to be solved (and representation)



Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.3

1. Syswerda, G.: Uniform crossover in genetic algorithms. In: Schaffer, J.D. (ed.) Proceedings of the 3rd International Conference on Genetic Algorithms, pp. 1–6. Morgan Kaufmann, San Francisco (1989)

Computer Simulation of GA

Genetic Algorithm Jupyter Notebook, sections 1–2

Outline

1. Applying Genetic Algorithms to Real-Valued Optimization
2. Beyond Fitness-Proportionate Selection
3. Ideas for Multimodal Optimization
4. Beyond Binary Representations
5. Parameter Control in Evolutionary Algorithms
6. Constraint Handling

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Stochastic Remainder Selection

- When we select parents to be mated using fitness-proportionate selection, we select them probabilistically given their relative fitness
- The main issue in algorithms of this kind is that once the probability distribution is computed, the expected number of copies of each parent is

$$e_i = p_i \cdot \mu \ ,$$

  where p_i is the fitness-proportionate probability
- Clearly, in practice, e_i has to be an integer, whereas $p_i \cdot \mu$ might not necessarily be so
- Therefore, different implementations can lead to different *estimators* of e_i, having different biases and variances
- An interesting approach[2] that is claimed to have lower mean-squared error that the regular fitness-proportionate selection is called **stochastic remainder selection**
- We represent each e_i as a sum of its integer part $[e_i]$ and its fractional part (remainder) $r_i = \{e_i\}$
- We select each individual *exactly* $[e_i]$ times
- Then we create *another* probability distribution as follows: $q_i = r_i / \sum_{i=1}^{\mu} r_i$
- We select remaining parents using these probabilities q_i

2. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Ranking Selection

- A totally different approach[3] suggests **sorting** the individuals based on their fitness values
- Then selection probabilities are allocated according to individuals' ranks
- Most commonly, a linear mapping is used
- One particular formula suggested in (Eiben and Smith, 2015) is as follows:

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \quad 1 < s \leq 2$$

- If stronger selection is required for fitter individuals, exponential mapping is used:

$$p_i = \frac{1-e^{-i}}{c},$$

  where c is chosen so that all probabilities sum to 1
- Then, stochastic remainder selection can be used with these probabilities

3. Baker, J.E.: Adaptive Selection Methods for Genetic Algorithms. In: Proceedings of an International Conferences on Genetic Algorithms and Their Applications, pp. 101–111 (1985)

Ranking Selection

- A totally different approach[3] suggests **sorting** the individuals based on their fitness values
- Then selection probabilities are allocated according to individuals' ranks
- Most commonly, a linear mapping is used
- One particular formula suggested in (Eiben and Smith, 2015) is as follows:

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \quad 1 < s \leq 2$$

- If stronger selection is required for fitter individuals, exponential mapping is used:

$$p_i = \frac{1-e^{-i}}{c},$$

  where c is chosen so that all probabilities sum to 1
- Then, stochastic remainder selection can be used with these probabilities

3. Baker, J.E.: Adaptive Selection Methods for Genetic Algorithms. In: Proceedings of an International Conferences on Genetic Algorithms and Their Applications, pp. 101–111 (1985)

Ranking Selection

- A totally different approach[3] suggests **sorting** the individuals based on their fitness values
- Then selection probabilities are allocated according to individuals' ranks
- Most commonly, a linear mapping is used
- One particular formula suggested in (Eiben and Smith, 2015) is as follows:

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \quad 1 < s \leq 2$$

- If stronger selection is required for fitter individuals, exponential mapping is used:

$$p_i = \frac{1-e^{-i}}{c},$$

  where c is chosen so that all probabilities sum to 1
- Then, stochastic remainder selection can be used with these probabilities

3. Baker, J.E.: Adaptive Selection Methods for Genetic Algorithms. In: Proceedings of an International Conferences on Genetic Algorithms and Their Applications, pp. 101–111 (1985)

Ranking Selection

- A totally different approach[3] suggests **sorting** the individuals based on their fitness values
- Then selection probabilities are allocated according to individuals' ranks
- Most commonly, a linear mapping is used
- One particular formula suggested in (Eiben and Smith, 2015) is as follows:

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \quad 1 < s \leq 2$$

- If stronger selection is required for fitter individuals, exponential mapping is used:

$$p_i = \frac{1-e^{-i}}{c},$$

  where c is chosen so that all probabilities sum to 1
- Then, stochastic remainder selection can be used with these probabilities

3. Baker, J.E.: Adaptive Selection Methods for Genetic Algorithms. In: Proceedings of an International Conferences on Genetic Algorithms and Their Applications, pp. 101–111 (1985)

Ranking Selection

- A totally different approach[3] suggests **sorting** the individuals based on their fitness values
- Then selection probabilities are allocated according to individuals' ranks
- Most commonly, a linear mapping is used
- One particular formula suggested in (Eiben and Smith, 2015) is as follows:

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \quad 1 < s \leq 2$$

- If stronger selection is required for fitter individuals, exponential mapping is used:

$$p_i = \frac{1-e^{-i}}{c},$$

  where c is chosen so that all probabilities sum to 1
- Then, stochastic remainder selection can be used with these probabilities

3. Baker, J.E.: Adaptive Selection Methods for Genetic Algorithms. In: Proceedings of an International Conferences on Genetic Algorithms and Their Applications, pp. 101–111 (1985)

Ranking Selection

- A totally different approach[3] suggests **sorting** the individuals based on their fitness values
- Then selection probabilities are allocated according to individuals' ranks
- Most commonly, a linear mapping is used
- One particular formula suggested in (Eiben and Smith, 2015) is as follows:

$$p_i = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \quad 1 < s \leq 2$$

- If stronger selection is required for fitter individuals, exponential mapping is used:

$$p_i = \frac{1-e^{-i}}{c},$$

  where c is chosen so that all probabilities sum to 1
- Then, stochastic remainder selection can be used with these probabilities

3. Baker, J.E.: Adaptive Selection Methods for Genetic Algorithms. In: Proceedings of an International Conferences on Genetic Algorithms and Their Applications, pp. 101–111 (1985)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Tournament Selection

- This approach[4] looks at relative rather than absolute fitness
- To select an individual, we pick q members at random, and select the best one among them
- q is called **tournament size**
- This method relies only on ordering relation that can rank any two individuals
- The larger the tournament, the greater the chance that it will contain members of above-average fitness
- Thus, as q is increased, the probability of selecting a high-fitness member increases, and that of selecting a low fitness member decreases
- In other words, increasing q increases the selection pressure
- Even though this method exhibits some variance, it is perhaps the most widely used method, at least in some evolutionary algorithms

4. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Elitist Selection

- This type of selection[5] can be used with any other selection
- The best k individuals are always selected deterministically
- In this fashion, the best individual so far always survives
- On the other hand, if it is only locally optimal, its preservation can lead to slower overall convergence

5. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Elitist Selection

- This type of selection[5] can be used with any other selection
- The best k individuals are always selected deterministically
- In this fashion, the best individual so far always survives
- On the other hand, if it is only locally optimal, its preservation can lead to slower overall convergence

5. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Elitist Selection

- This type of selection[5] can be used with any other selection
- The best k individuals are always selected deterministically
- In this fashion, the best individual so far always survives
- On the other hand, if it is only locally optimal, its preservation can lead to slower overall convergence

5. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Elitist Selection

- This type of selection[5] can be used with any other selection
- The best k individuals are always selected deterministically
- In this fashion, the best individual so far always survives
- On the other hand, if it is only locally optimal, its preservation can lead to slower overall convergence

5. Brindle, A.: Genetic Algorithms for Function Optimization. PhD thesis, University of Alberta (1981)

Computer Simulation of GA

Genetic Algorithm Jupyter Notebook, section 3

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
  - Completely at random (very nice approach)
  - Based on their fitness values
  - Deterministically (the worst ones are replaced)
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
  - Completely at random (very nice approach)
  - Based on their fitness values
  - Deterministically (the worst ones are replaced)
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
  - Completely at random (very nice approach)
  - Based on their fitness values
  - Deterministically (the worst ones are replaced)
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
  - Completely at random (very nice approach)
  - Based on their fitness values
  - Deterministically (the worst ones are replaced)
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
  - Completely at random (very nice approach)
  - Based on their fitness values
  - Deterministically (the worst ones are replaced)
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
  - Completely at random (very nice approach)
  - Based on their fitness values
  - Deterministically (the worst ones are replaced)
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Generation Gap

- Up to this point, we discussed only generational population models, where **all μ individuals** are replaced with the new offspring
- In the **steady-state** population model, only $G \cdot \mu$ individuals are replaced
- G is called **generation gap**
- Individuals to be replaced can be chosen:
  - Completely at random (very nice approach)
  - Based on their fitness values
  - Deterministically (the worst ones are replaced)
- Arguments in favor of each approach are mixed, the final choice needs to be made according to each task being solved

Computer Simulation of GA

Genetic Algorithm Jupyter Notebook, section 4

Outline

1. Applying Genetic Algorithms to Real-Valued Optimization
2. Beyond Fitness-Proportionate Selection
3. Ideas for Multimodal Optimization
4. Beyond Binary Representations
5. Parameter Control in Evolutionary Algorithms
6. Constraint Handling

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Motivation

- Many real-life problems have **multimodal** fitness landscapes
- One approach is to pick the right parameters to be able to escape local optima (including parameter control techniques discussed next)
- Another approach is to try to obtain several **sets of solutions** that correspond to different **local optima**, especially when they are very close to each other
- In practice, the finite population size usually leads to **genetic drift**, i.e. convergence to one optimum
- In nature, there are many cases that can be used as inspirations for developing advanced techniques in evolutionary computing
- E.g., **speciation**: different species adapt to occupy different environmental niches
- Or **punctuated equilibria and local adaptation**: periods of evolutionary stasis are interrupted by rapid growth when the main population is "invaded" by individuals from isolated groups from the same species
- In the latter case, the population is split into isolated subpopulations (**demes**), with only occasional migrations

Implicit and Explicit Techniques

- Evolutionary approaches to solving multimodal problems can be classified into[6]:
  - **Implicit**: a framework is used that permits, *but does not guarantee*, the preservation of diverse solutions
  - **Explicit**: specific operators are used to preserve diversity in the population

6. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

Implicit and Explicit Techniques

- Evolutionary approaches to solving multimodal problems can be classified into[6]:
  - **Implicit**: a framework is used that permits, *but does not guarantee*, the preservation of diverse solutions
  - **Explicit**: specific operators are used to preserve diversity in the population

6. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

Implicit and Explicit Techniques

- Evolutionary approaches to solving multimodal problems can be classified into[6]:
  - **Implicit**: a framework is used that permits, *but does not guarantee*, the preservation of diverse solutions
  - **Explicit**: specific operators are used to preserve diversity in the population

6. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Island Model

- One implicit approach is called **island model (coarse-grain parallel) evolutionary algorithm**
- Multiple populations are run in parallel in some kind of communication structure (usually a ring or a torus)
- After a (usually fixed) number of generations (**epoch**), a number of individuals are selected from each population to be exchanged with others from neighboring populations (**migration**)
- The idea is that each subpopulation explores the search space around the fitter solutions that they contain (exploitation), and migration facilitates exploration
- In practice, however, there is no guarantee that different subpopulations are actually exploring different regions of the search space
- Some questions need to be addressed
- How often to migrate? Too frequently will lead to all subpopulations converging on the same solution, too infrequently will waste computational effort
- Which individuals to select for migration? First of all, it is generally better to migrate a small number (2–5) of individuals to prevent too rapid convergence. The individuals can be selected at random or based on fitness
- How to divide the population into subpopulations? Usually, more subpopulations give better results

Cellular Evolutionary Algorithms

- In **cellular (fine-grain parallel, diffusion model, distributed) evolutionary algorithms**, one population is divided into demes[7]
- Each individual has its own neighborhood, which acts as a separate deme
- Evolution proceeds as follows
- For each individual, we select two parents from its deme
- We recombine and mutate the parents to obtain one offspring
- We replace the individual with the offspring

7. Pettey, C.C.: Diffusion (Cellular) Models. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 125–133. Institute of Physics Publishing, Bristol (2000)

Cellular Evolutionary Algorithms

- In **cellular (fine-grain parallel, diffusion model, distributed) evolutionary algorithms**, one population is divided into demes[7]
- Each individual has its own neighborhood, which acts as a separate deme
- Evolution proceeds as follows
- For each individual, we select two parents from its deme
- We recombine and mutate the parents to obtain one offspring
- We replace the individual with the offspring

7. Pettey, C.C.: Diffusion (Cellular) Models. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 125–133. Institute of Physics Publishing, Bristol (2000)

Cellular Evolutionary Algorithms

- In **cellular (fine-grain parallel, diffusion model, distributed) evolutionary algorithms**, one population is divided into demes[7]
- Each individual has its own neighborhood, which acts as a separate deme
- Evolution proceeds as follows
- For each individual, we select two parents from its deme
- We recombine and mutate the parents to obtain one offspring
- We replace the individual with the offspring

7. Pettey, C.C.: Diffusion (Cellular) Models. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 125–133. Institute of Physics Publishing, Bristol (2000)

Cellular Evolutionary Algorithms

- In **cellular (fine-grain parallel, diffusion model, distributed) evolutionary algorithms**, one population is divided into demes[7]
- Each individual has its own neighborhood, which acts as a separate deme
- Evolution proceeds as follows
- For each individual, we select two parents from its deme
- We recombine and mutate the parents to obtain one offspring
- We replace the individual with the offspring

7. Pettey, C.C.: Diffusion (Cellular) Models. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 125–133. Institute of Physics Publishing, Bristol (2000)

Cellular Evolutionary Algorithms

- In **cellular (fine-grain parallel, diffusion model, distributed) evolutionary algorithms**, one population is divided into demes[7]
- Each individual has its own neighborhood, which acts as a separate deme
- Evolution proceeds as follows
- For each individual, we select two parents from its deme
- We recombine and mutate the parents to obtain one offspring
- We replace the individual with the offspring

7. Pettey, C.C.: Diffusion (Cellular) Models. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 125–133. Institute of Physics Publishing, Bristol (2000)

Cellular Evolutionary Algorithms

- In **cellular (fine-grain parallel, diffusion model, distributed) evolutionary algorithms**, one population is divided into demes[7]
- Each individual has its own neighborhood, which acts as a separate deme
- Evolution proceeds as follows
- For each individual, we select two parents from its deme
- We recombine and mutate the parents to obtain one offspring
- We replace the individual with the offspring

7. Pettey, C.C.: Diffusion (Cellular) Models. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 125–133. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols 0, 1, and #)
  - A **tag** (a small string only of 0 and 1 of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (0 matches 0, 1 matches 1, and # matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lceil \log_2 q \rceil + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lceil \log_2 q \rceil + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lfloor \log_2 q \rfloor + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lceil \log_2 q \rceil + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lceil \log_2 q \rceil + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lfloor \log_2 q \rfloor + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lceil \log_2 q \rceil + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lceil \log_2 q \rceil + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $\lceil \log_2 q \rceil + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Speciation

- Let us consider an explicit approach, in which the population contains multiple **species**[8], and during parent selection individuals only mate with others of the same species
- E.g., in the **tag-template scheme**, each individual consists of three strings:
  - A **functional** string (chromosome)
  - A **template** (a small string consisting of symbols `0`, `1`, and `#`)
  - A **tag** (a small string only of `0` and `1` of the same length as the template)
- The tag and template are initialized randomly
- They don't participate in fitness calculation, but they are subject to crossover and mutation as the rest of the string
- Before recombining two parents, their tag and template strings are **matched**: the number of matches between bits of the template of one parent and the tag of the other parent is calculated (`0` matches `0`, `1` matches `1`, and `#` matches both)
- If the number of matches exceeds the specified threshold, individuals are recombined; otherwise, another pair is selected
- If q different optimal solutions are desired, the minimum template string length is $[\log_2 q] + 1$

8. Deb, K., Spears, W.M.: Speciation Methods. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 93–100. Institute of Physics Publishing, Bristol (2000)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Sharing

- Another explicit approach is called **sharing**[9]
- For each individual, we obtain a **degree of sharing** as the sum of sharing function values contributed by all other individuals, and then the fitness is derated:

$$f'(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{j=1}^{\mu} \mathrm{sh}(d(\mathbf{x}_i, \mathbf{x}_j))}$$

- Here d is the distance between two individuals (Hamming distance for binary strings, Euclidean distance for real vectors, etc.)
- The closer the individuals are to each other, the higher should be their sharing function values
- Thus, when many individuals are **close** to each other, they **derate** each other's fitness, and so the uncontrolled growth of particular species within population is limited
- A common sharing function is

$$\mathrm{sh}(x) = \begin{cases} 1 - \left(\frac{x}{\sigma}\right)^{\alpha}, & x < \sigma \\ 0, & \text{otherwise} \end{cases}$$

- Here σ is user-specified, α regulates the shape
- In general, it is not easy to pick σ, the default is recommended to be 5–10

9. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Computer Simulation of GA

Genetic Algorithm Jupyter Notebook, section 5

Outline

1. Applying Genetic Algorithms to Real-Valued Optimization
2. Beyond Fitness-Proportionate Selection
3. Ideas for Multimodal Optimization
4. **Beyond Binary Representations**
5. Parameter Control in Evolutionary Algorithms
6. Constraint Handling

Other Representations in GAs

- Aside from classical binary strings, other representations can be used in genetic algorithms
- We will discuss them only superficially, if needed, you can implement them on your own
- Representations we will touch upon are as follows:
  - Gray coded binary representation
  - Integer representation
  - Permutation representation

Other Representations in GAs

- Aside from classical binary strings, other representations can be used in genetic algorithms
- We will discuss them only superficially, if needed, you can implement them on your own
- Representations we will touch upon are as follows:
  - Gray coded binary representation
  - Integer representation
  - Permutation representation

Other Representations in GAs

- Aside from classical binary strings, other representations can be used in genetic algorithms
- We will discuss them only superficially, if needed, you can implement them on your own
- Representations we will touch upon are as follows:
  - Gray coded binary representation
  - Integer representation
  - Permutation representation

Other Representations in GAs

- Aside from classical binary strings, other representations can be used in genetic algorithms
- We will discuss them only superficially, if needed, you can implement them on your own
- Representations we will touch upon are as follows:
  - Gray coded binary representation
  - Integer representation
  - Permutation representation

Other Representations in GAs

- Aside from classical binary strings, other representations can be used in genetic algorithms
- We will discuss them only superficially, if needed, you can implement them on your own
- Representations we will touch upon are as follows:
  - Gray coded binary representation
  - Integer representation
  - Permutation representation

Other Representations in GAs

- Aside from classical binary strings, other representations can be used in genetic algorithms
- We will discuss them only superficially, if needed, you can implement them on your own
- Representations we will touch upon are as follows:
  - Gray coded binary representation
  - Integer representation
  - Permutation representation

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Gray Coded Binary Representation

- Each chromosome is still a binary string
- But unlike conventional binary coding, we use **Gray coding**
- It is thought to be better in the sense that *Hamming distance* between consecutive integers is always one, hence mutations are more uniform
- We can convert a binary number $\mathbf{a} = (a_1, \ldots, a_n)$ to a Gray coded number $\mathbf{b} = (b_1, \ldots, b_n)$ as follows:

$$b_i = \begin{cases} a_i \ , & i = 1 \\ a_{i-1} \oplus a_i \ , & i > 1 \end{cases}$$

- Here $\oplus$ stands for sum modulo 2
- Backward conversion is achieved by

$$a_i = \oplus_{j=1}^{i} b_j$$

- For example, 4 is coded as 100 in binary code and as 110 in Gray code
- Other than a different decoding of a phenotype, nothing else changes

Integer Representation

- Each chromosome is a string of integers
- Such encoding can be useful when variables are categorical values from a fixed set
- I.e., there is no order, so usual crossover and mutation can be meaningless
- Crossover operators are the same as for the binary representation
- But instead of bit-flipping mutation, we typically use the **random-resetting mutation**
- Each gene (with probability p_m) gets a new value from the set of permissible values

Integer Representation

- Each chromosome is a string of integers
- Such encoding can be useful when variables are categorical values from a fixed set
- I.e., there is no order, so usual crossover and mutation can be meaningless
- Crossover operators are the same as for the binary representation
- But instead of bit-flipping mutation, we typically use the **random-resetting mutation**
- Each gene (with probability p_m) gets a new value from the set of permissible values

Integer Representation

- Each chromosome is a string of integers
- Such encoding can be useful when variables are categorical values from a fixed set
- I.e., there is no order, so usual crossover and mutation can be meaningless
- Crossover operators are the same as for the binary representation
- But instead of bit-flipping mutation, we typically use the **random-resetting mutation**
- Each gene (with probability p_m) gets a new value from the set of permissible values

Integer Representation

- Each chromosome is a string of integers
- Such encoding can be useful when variables are categorical values from a fixed set
- I.e., there is no order, so usual crossover and mutation can be meaningless
- Crossover operators are the same as for the binary representation
- But instead of bit-flipping mutation, we typically use the **random-resetting mutation**
- Each gene (with probability p_m) gets a new value from the set of permissible values

Integer Representation

- Each chromosome is a string of integers
- Such encoding can be useful when variables are categorical values from a fixed set
- I.e., there is no order, so usual crossover and mutation can be meaningless
- Crossover operators are the same as for the binary representation
- But instead of bit-flipping mutation, we typically use the **random-resetting mutation**
- Each gene (with probability p_m) gets a new value from the set of permissible values

Integer Representation

- Each chromosome is a string of integers
- Such encoding can be useful when variables are categorical values from a fixed set
- I.e., there is no order, so usual crossover and mutation can be meaningless
- Crossover operators are the same as for the binary representation
- But instead of bit-flipping mutation, we typically use the **random-resetting mutation**
- Each gene (with probability p_m) gets a new value from the set of permissible values

Permutation Representation

- Each chromosome is a permutation of a set of integers
- Such encoding is most useful for problems of deciding on the **order**, in which a sequence of events should occur
- It is clear that in this case variation operators should preserve the permutation property
- There **cannot be** two equal numbers in a permutation
- There are two classes of permutation problems:

Permutation Representation

- Each chromosome is a permutation of a set of integers
- Such encoding is most useful for problems of deciding on the **order**, in which a sequence of events should occur
- It is clear that in this case variation operators should preserve the permutation property
- There **cannot be** two equal numbers in a permutation
- There are two classes of permutation problems:

Permutation Representation

- Each chromosome is a permutation of a set of integers
- Such encoding is most useful for problems of deciding on the **order**, in which a sequence of events should occur
- It is clear that in this case variation operators should preserve the permutation property
- There **cannot be** two equal numbers in a permutation
- There are two classes of permutation problems:
  - Those where the order of events is relevant (e.g., job scheduling problems)
  - Those where only adjacency matters (e.g., traveling salesman problem)

Permutation Representation

- Each chromosome is a permutation of a set of integers
- Such encoding is most useful for problems of deciding on the **order**, in which a sequence of events should occur
- It is clear that in this case variation operators should preserve the permutation property
- There **cannot be** two equal numbers in a permutation
- There are two classes of permutation problems:
  - Those where the **order of events** is relevant (e.g., job scheduling problems)
  - Those where only **adjacency** matters (e.g., traveling salesman problem)

Permutation Representation

- Each chromosome is a permutation of a set of integers
- Such encoding is most useful for problems of deciding on the **order**, in which a sequence of events should occur
- It is clear that in this case variation operators should preserve the permutation property
- There **cannot be** two equal numbers in a permutation
- There are two classes of permutation problems:
  - Those where the **order of events** is relevant (e.g., job scheduling problems)
  - Those where only **adjacency** matters (e.g., traveling salesman problem)

Permutation Representation

- Each chromosome is a permutation of a set of integers
- Such encoding is most useful for problems of deciding on the **order**, in which a sequence of events should occur
- It is clear that in this case variation operators should preserve the permutation property
- There **cannot be** two equal numbers in a permutation
- There are two classes of permutation problems:
  - Those where the **order of events** is relevant (e.g., job scheduling problems)
  - Those where only **adjacency** matters (e.g., traveling salesman problem)

Permutation Representation

- Each chromosome is a permutation of a set of integers
- Such encoding is most useful for problems of deciding on the **order**, in which a sequence of events should occur
- It is clear that in this case variation operators should preserve the permutation property
- There **cannot be** two equal numbers in a permutation
- There are two classes of permutation problems:
  - Those where the **order of events** is relevant (e.g., job scheduling problems)
  - Those where only **adjacency** matters (e.g., traveling salesman problem)

Mutation for Permutation Representation

- Mutations are not gene-wise, but string-wise
- In **swap mutation**, we choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size) and **swap** appropriate alleles
- This mutation is best suited for order-based problems

Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.10

Mutation for Permutation Representation

- Mutations are not gene-wise, but string-wise
- In **swap mutation**, we choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size) and **swap** appropriate alleles
- This mutation is best suited for order-based problems

Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.10

Mutation for Permutation Representation

- Mutations are not gene-wise, but string-wise
- In **swap mutation**, we choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size) and **swap** appropriate alleles
- This mutation is best suited for order-based problems

Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.10

Mutation for Permutation Representation

- In **inversion mutation**, we choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size) and **invert** the substring between selected alleles
- This mutation is best suited for adjacency-based problems

Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.11

Mutation for Permutation Representation

- In **inversion mutation**, we choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size) and **invert** the substring between selected alleles
- This mutation is best suited for adjacency-based problems

Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.11

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 into the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already taken by an element k, put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 in the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already taken by an element k, put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 in the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already filled (by an element k), put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 in the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already filled (by an element k), put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 in the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already filled (by an element k), put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 in the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already filled (by an element k), put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 in the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already filled (by an element k), put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

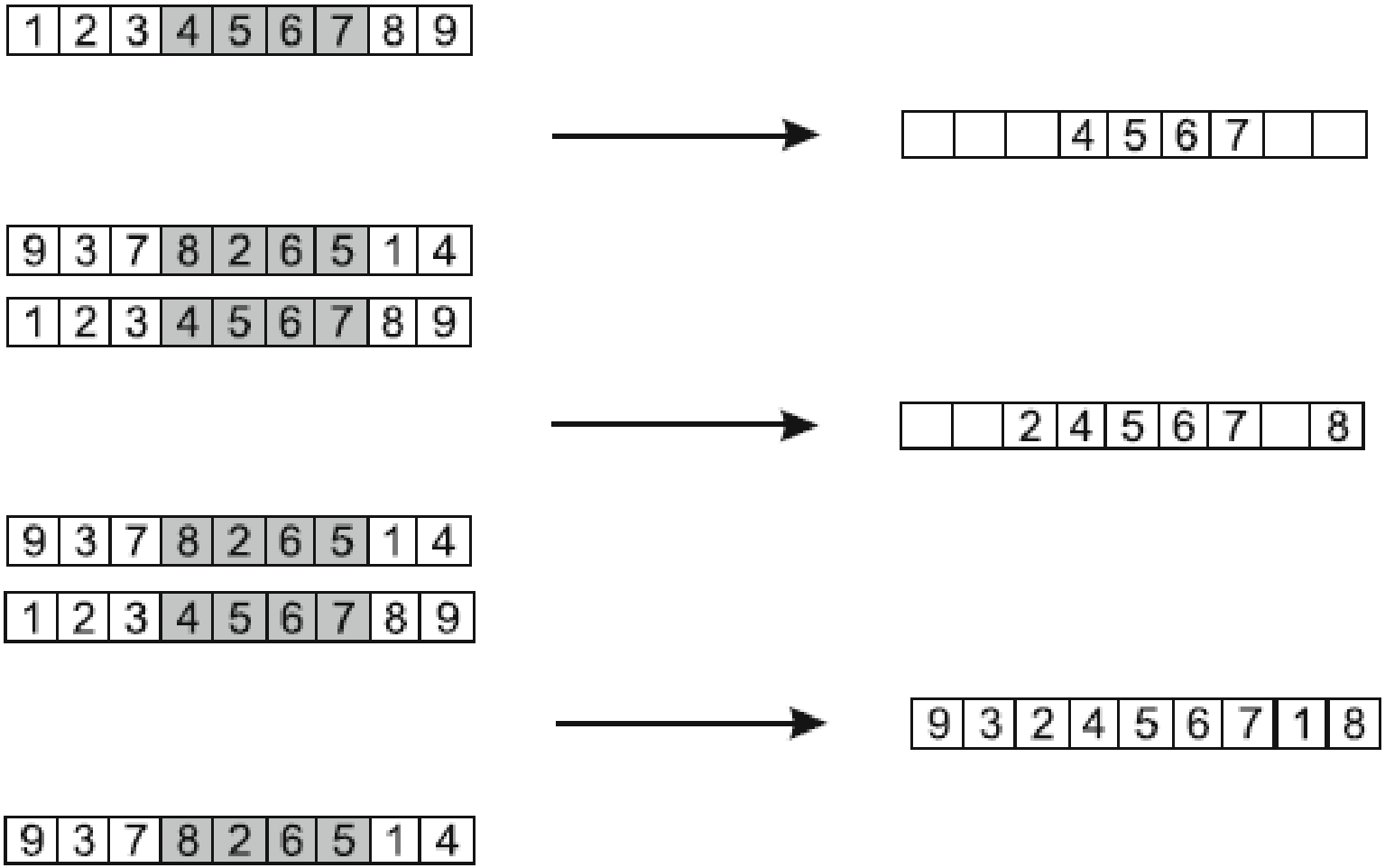
10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation

- Let us discuss the **partially mapped crossover** (PMX)[10]
- We choose two random integers $k_1, k_2 \in [1; l]$ (where l is the chromosome size)
- We copy the segment between appropriate alleles (**matching section**) from parent 1 to child 1
- Starting from k_1, for each element i in the matching section of parent 2, we do the following steps:
  - Find an element j that has been copied from parent 1 to child 1 in the place of i
  - Put i in the position occupied by j in parent 2
  - If the place occupied by j in parent 2 is already filled (by an element k), put i in the position occupied by k in parent 2
- Fill the rest of child 1 using parent 2
- The whole process is then repeated, replacing parent 2 (1) with parent 1 (2) and child 1 with child 2

10. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional (1989)

Crossover for Permutation Representation



Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015), Fig. 4.12–4.14

Crossover for Permutation Representation

- PMX is best suited for order-based problems
- Other similar algorithms are order crossover[11] and cycle crossover[12]
- For adjacency-based problems, a commonly used crossover is **edge crossover**[13]
- We will not discuss these crossovers here but you can always look them up in appropriate sources

11. Davis, L. (ed.): Handbook of Genetic Algorithms. Van Nostrand Reinhold (1991)

12. Oliver, I.M., Smith, D.J., Holland, J.: A Study of Permutation Crossover Operators on the Travelling Salesman Problem. In: Grefenstette, J.J. (ed.) Proceedings of the 2nd International Conference on Genetic Algorithms and Their Applications, pp. 224–230. Lawrence Erlbaum, Hillsdale, New Jersey (1987)

13. Whitley, D.: Recombination. Permutations. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.): Evolutionary Computation 1. Basic Algorithms and Operators, pp. 274–284. Taylor & Francis Group LLC (2000)

Crossover for Permutation Representation

- PMX is best suited for order-based problems
- Other similar algorithms are order crossover[11] and cycle crossover[12]
- For adjacency-based problems, a commonly used crossover is **edge crossover**[13]
- We will not discuss these crossovers here but you can always look them up in appropriate sources

11. Davis, L. (ed.): Handbook of Genetic Algorithms. Van Nostrand Reinhold (1991)

12. Oliver, I.M., Smith, D.J., Holland, J.: A Study of Permutation Crossover Operators on the Travelling Salesman Problem. In: Grefenstette, J.J. (ed.) Proceedings of the 2nd International Conference on Genetic Algorithms and Their Applications, pp. 224–230. Lawrence Erlbaum, Hillsdale, New Jersey (1987)

13. Whitley, D.: Recombination. Permutations. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.): Evolutionary Computation 1. Basic Algorithms and Operators, pp. 274–284. Taylor & Francis Group LLC (2000)

Crossover for Permutation Representation

- PMX is best suited for order-based problems
- Other similar algorithms are order crossover[11] and cycle crossover[12]
- For adjacency-based problems, a commonly used crossover is **edge crossover**[13]
- We will not discuss these crossovers here but you can always look them up in appropriate sources

11. Davis, L. (ed.): Handbook of Genetic Algorithms. Van Nostrand Reinhold (1991)
12. Oliver, I.M., Smith, D.J., Holland, J.: A Study of Permutation Crossover Operators on the Travelling Salesman Problem. In: Grefenstette, J.J. (ed.) Proceedings of the 2nd International Conference on Genetic Algorithms and Their Applications, pp. 224–230. Lawrence Erlbaum, Hillsdale, New Jersey (1987)
13. Whitley, D.: Recombination. Permutations. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.): Evolutionary Computation 1. Basic Algorithms and Operators, pp. 274–284. Taylor & Francis Group LLC (2000)

Crossover for Permutation Representation

- PMX is best suited for order-based problems
- Other similar algorithms are order crossover[11] and cycle crossover[12]
- For adjacency-based problems, a commonly used crossover is **edge crossover**[13]
- We will not discuss these crossovers here but you can always look them up in appropriate sources

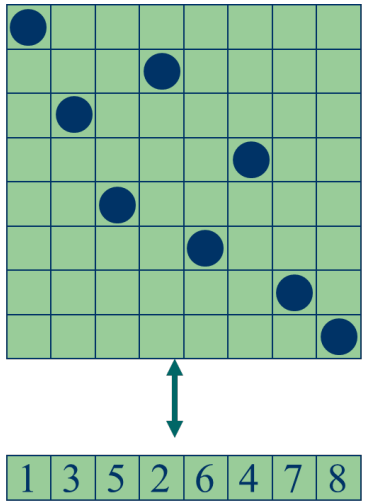
11. Davis, L. (ed.): Handbook of Genetic Algorithms. Van Nostrand Reinhold (1991)

12. Oliver, I.M., Smith, D.J., Holland, J.: A Study of Permutation Crossover Operators on the Travelling Salesman Problem. In: Grefenstette, J.J. (ed.) Proceedings of the 2nd International Conference on Genetic Algorithms and Their Applications, pp. 224–230. Lawrence Erlbaum, Hillsdale, New Jersey (1987)

13. Whitley, D.: Recombination. Permutations. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.): Evolutionary Computation 1. Basic Algorithms and Operators, pp. 274–284. Taylor & Francis Group LLC (2000)

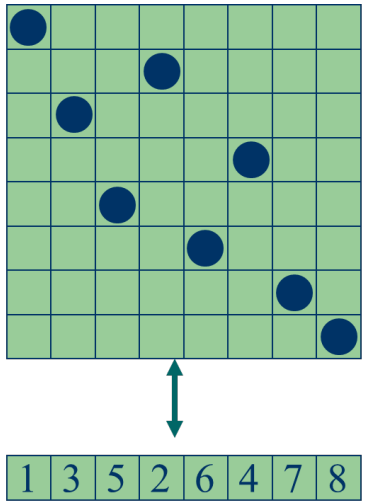
Examples of Problem Specification

- Let us discuss a couple of real optimization problems and ways to solve them using GA
- Consider the so-called Eight queens puzzle
- The goal is to place n queens on an $n \times n$ chessboard so that they cannot check each other
- **Phenotype**: a board configuration
- **Genotype**: a **permutation** of integers from 1 to n



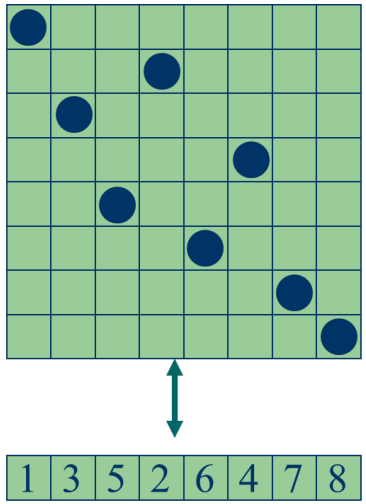
Examples of Problem Specification

- Let us discuss a couple of real optimization problems and ways to solve them using GA
- Consider the so-called Eight queens puzzle
- The goal is to place n queens on an $n \times n$ chessboard so that they cannot check each other
- **Phenotype**: a board configuration
- **Genotype**: a **permutation** of integers from 1 to n



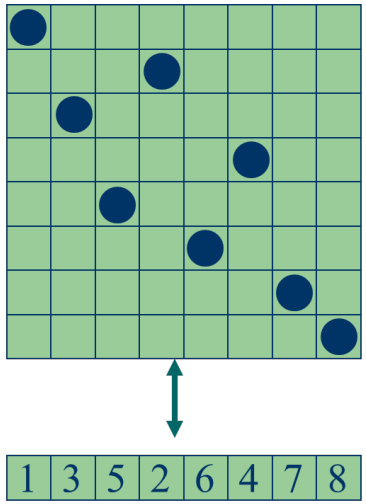
Examples of Problem Specification

- Let us discuss a couple of real optimization problems and ways to solve them using GA
- Consider the so-called Eight queens puzzle
- The goal is to place n queens on an $n \times n$ chessboard so that they cannot check each other
- **Phenotype**: a board configuration
- **Genotype**: a **permutation** of integers from 1 to n



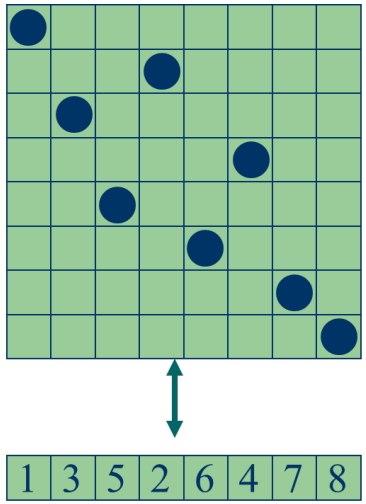
Examples of Problem Specification

- Let us discuss a couple of real optimization problems and ways to solve them using GA
- Consider the so-called Eight queens puzzle
- The goal is to place n queens on an $n \times n$ chessboard so that they cannot check each other
- **Phenotype**: a board configuration
- **Genotype**: a **permutation** of integers from 1 to n



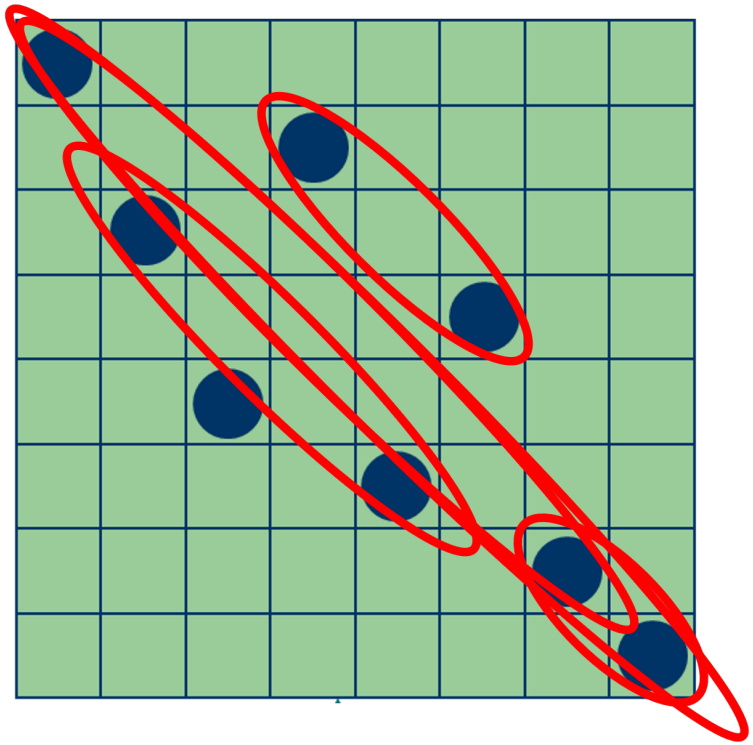
Examples of Problem Specification

- Let us discuss a couple of real optimization problems and ways to solve them using GA
- Consider the so-called Eight queens puzzle
- The goal is to place n queens on an $n \times n$ chessboard so that they cannot check each other
- **Phenotype**: a board configuration
- **Genotype**: a **permutation** of integers from 1 to n



Examples of Problem Specification

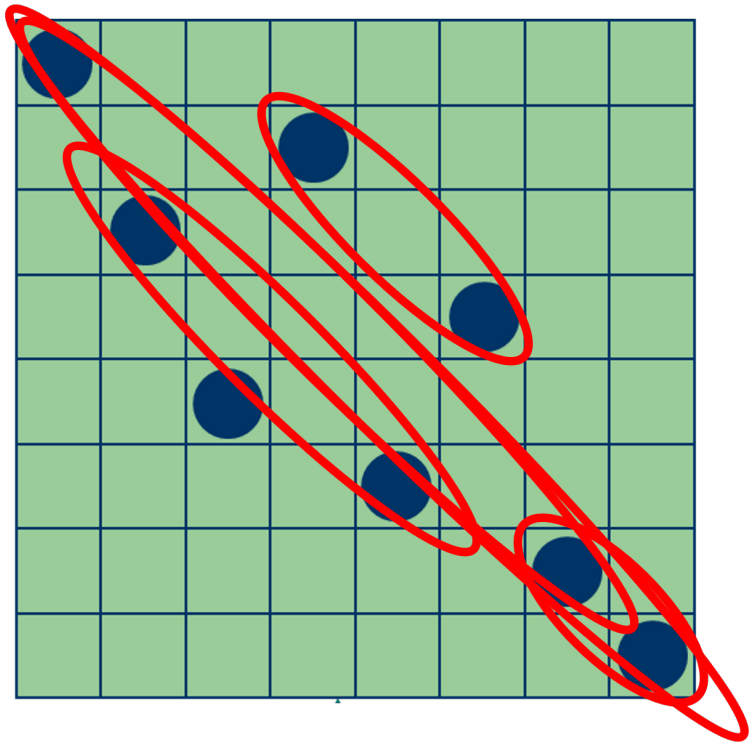
- How to specify a fitness function?
- One option is to define *quality* $q(P)$ of a phenotype P as the number of checking queen pairs
- The lower $q(P)$, the better
- In particular, when $q(P) = 0$, we have found the perfect solution
- Then, fitness function can be defined as $f(P) = 28 - q(P)$



Here, $q(P) = 5$

Examples of Problem Specification

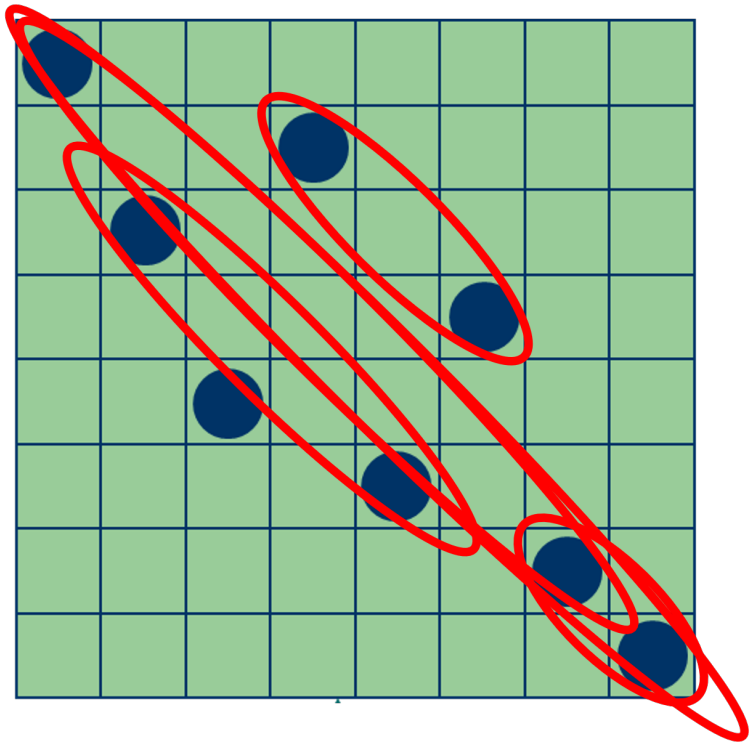
- How to specify a fitness function?
- One option is to define *quality* $q(P)$ of a phenotype P as the number of checking queen pairs
- The lower $q(P)$, the better
- In particular, when $q(P) = 0$, we have found the perfect solution
- Then, fitness function can be defined as $f(P) = 28 - q(P)$



Here, $q(P) = 5$

Examples of Problem Specification

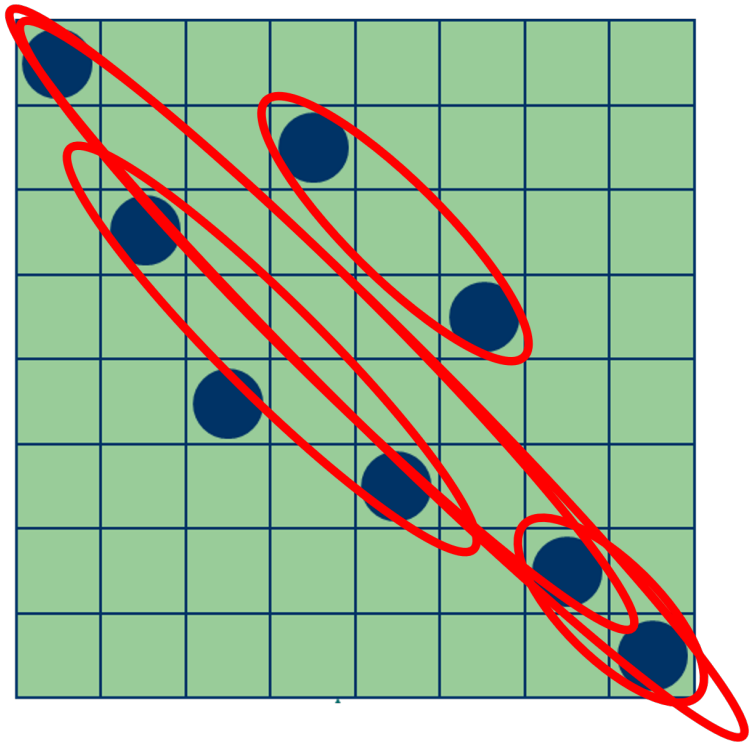
- How to specify a fitness function?
- One option is to define *quality* $q(P)$ of a phenotype P as the number of checking queen pairs
- The lower $q(P)$, the better
- In particular, when $q(P) = 0$, we have found the perfect solution
- Then, fitness function can be defined as $f(P) = 28 - q(P)$



Here, $q(P) = 5$

Examples of Problem Specification

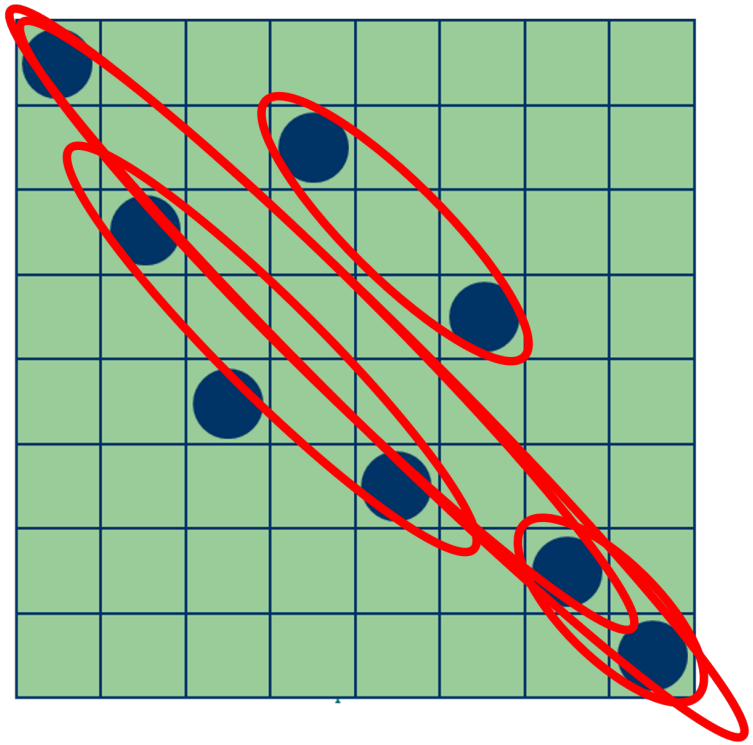
- How to specify a fitness function?
- One option is to define *quality* $q(P)$ of a phenotype P as the number of checking queen pairs
- The lower $q(P)$, the better
- In particular, when $q(P) = 0$, we have found the perfect solution
- Then, fitness function can be defined as $f(P) = 28 - q(P)$



Here, $q(P) = 5$

Examples of Problem Specification

- How to specify a fitness function?
- One option is to define *quality* $q(P)$ of a phenotype P as the number of checking queen pairs
- The lower $q(P)$, the better
- In particular, when $q(P) = 0$, we have found the perfect solution
- Then, fitness function can be defined as $f(P) = 28 - q(P)$



Here, $q(P) = 5$

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider the Traveling Salesman Problem
- Given a set of cities with known distances between then, we need to find the order of visiting them so that the total distance covered in minimal
- Is known to be NP-hard
- Many heuristics specifically tailored for this problem exist
- Genetic algorithms cannot solve the problem effectively
- But incorporated heuristics can sufficiently enhance their performance
- Representation in this case is simply a permutation of city numbers
- Fitness can be naturally defined in terms of the total distance traveled

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A schedule is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A schedule is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- Let us now consider a more complicated job shop scheduling problem[14]
- Let us have the following parameters:
  - J — the set of jobs
  - O — the set of operations
  - M — the set of machines
- Let $\text{Able} : O \to M$ be the mapping defining which machine can perform which operation (for simplicity, each operation can be carried out only on one machine)
- Let $\text{Pre} \subseteq O \times O$ be the relation defining which operation should *precede* which operations
- Let $d : O \times M \to \mathbb{R}$ be the function defining the *duration* of a particular operation on a particular machine
- *Scheduling an operation* means to assign a starting time to it on the machine that can perform it
- A **schedule** is a collection of such assignments containing each operation at most once

14. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2003)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Examples of Problem Specification

- The goal is to find a schedule that is:
  - *Complete*: all jobs are scheduled
  - *Correct*: all conditions defined by Pre are satisfied
  - *Optimal*: the total duration of the schedule is minimal
- The **representation** is the **permutation** of the set of possible operations (for each machine)
- The **phenotype** is a schedule created the following way:
  - Take the next operation from the permutation
  - Look up its machine
  - Assign the earliest possible starting time on this machine (subject to the occupation of the machine and to the precedence relation)
- This mapping guarantees both the completeness and correctness of the schedule
- The optimality will be sought for by GA, if the **fitness function** is the total duration (to be minimized)

Outline

1. Applying Genetic Algorithms to Real-Valued Optimization
2. Beyond Fitness-Proportionate Selection
3. Ideas for Multimodal Optimization
4. Beyond Binary Representations
5. Parameter Control in Evolutionary Algorithms
6. Constraint Handling

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - Parameter tuning: finding good values for the parameters before the run of the algorithm (hard but widely used in practice)
  - Parameter control: starting a run with initial parameter values and changing them during the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - **Parameter tuning**: finding good values for the parameters before the run of the algorithm (hard but widely used in practice!)
  - **Parameter control**: starting a run with initial parameter values and changing them during the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - **Parameter tuning**: finding good values for the parameters **before** the run of the algorithm (hard but widely used in practice!)
  - **Parameter control**: starting a run with initial parameter values and changing them **during** the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - **Parameter tuning**: finding good values for the parameters **before** the run of the algorithm (hard but widely used in practice!)
  - **Parameter control**: starting a run with initial parameter values and changing them **during** the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - **Parameter tuning**: finding good values for the parameters **before** the run of the algorithm (hard but widely used in practice!)
  - **Parameter control**: starting a run with initial parameter values and changing them **during** the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - **Parameter tuning**: finding good values for the parameters **before** the run of the algorithm (hard but widely used in practice!)
  - **Parameter control**: starting a run with initial parameter values and changing them **during** the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - **Parameter tuning**: finding good values for the parameters **before** the run of the algorithm (hard but widely used in practice!)
  - **Parameter control**: starting a run with initial parameter values and changing them **during** the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Motivation

- To run any evolutionary algorithm, it is important to fully specify all of its parameters
- In particular, one has to choose population size μ, crossover probability p_c, mutation probability p_m, and selection parameters (e.g., tournament size q)
- Values of these parameters greatly influence the algorithm's ability to find a solution and the efficiency of the search
- There can be distinguished two major forms of setting parameter values:
  - **Parameter tuning**: finding good values for the parameters **before** the run of the algorithm (hard but widely used in practice!)
  - **Parameter control**: starting a run with initial parameter values and changing them **during** the run
- Conceptually, parameter control should provide better results
- Indeed, the run of a GA is a dynamic and adaptive process, so different values of parameters might be better at different stages
- E.g., mutation probability can be large in the early generations to provide better exploration of the search space, and small in the late generations to facilitate exploitation

Classification

- Parameter control techniques can be classified according to several criteria[15]:
  - **What** is changed (population size, crossover probability, mutation probability, selection parameters)
  - **How** the change is made (deterministic, adaptive, self-adaptive)
  - The **evidence** upon which the change is carried out (e.g., diversity of the population)
  - The **scope** of change (e.g., population-level, individual-level etc.)

15. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

Classification

- Parameter control techniques can be classified according to several criteria[15]:
  - **What** is changed (population size, crossover probability, mutation probability, selection parameters)
  - **How** the change is made (deterministic, adaptive, self-adaptive)
  - The **evidence** upon which the change is carried out (e.g., diversity of the population)
  - The **scope** of change (e.g., population-level, individual-level etc.)

15. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

Classification

- Parameter control techniques can be classified according to several criteria[15]:
  - **What** is changed (population size, crossover probability, mutation probability, selection parameters)
  - **How** the change is made (deterministic, adaptive, self-adaptive)
  - The **evidence** upon which the change is carried out (e.g., diversity of the population)
  - The **scope** of change (e.g., population-level, individual-level etc.)

15. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

Classification

- Parameter control techniques can be classified according to several criteria[15]:
  - **What** is changed (population size, crossover probability, mutation probability, selection parameters)
  - **How** the change is made (deterministic, adaptive, self-adaptive)
  - The **evidence** upon which the change is carried out (e.g., diversity of the population)
  - The **scope** of change (e.g., population-level, individual-level etc.)

15. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

Classification

- Parameter control techniques can be classified according to several criteria[15]:
  - **What** is changed (population size, crossover probability, mutation probability, selection parameters)
  - **How** the change is made (deterministic, adaptive, self-adaptive)
  - The **evidence** upon which the change is carried out (e.g., diversity of the population)
  - The **scope** of change (e.g., population-level, individual-level etc.)

15. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. Springer-Verlag, Berlin, Heidelberg (2015)

How the Change is Made

- With **deterministic** parameter control, the parameter values are altered by a predetermined deterministic rule without using any feedback from the search (usually, a time-varying schedule is used)
- E.g., mutation probability can be time-varying[16]:

$$p_m(t) = \sqrt{\frac{c_1}{c_2}} \cdot \frac{e^{-c_3 \frac{t}{2}}}{\mu\sqrt{l}}$$

where c_1, c_2, c_3 are user-defined constants, l is chromosome length, t is the generation number

- In general, deterministic schedules are of limited utility

16. Hesser, J., Maenner, R.: Investigation of the m-Heuristic for Optimal Mutation Probabilities. In: Maenner, R., Manderick, B. (eds.) Parallel Problem Solving from Nature 2, pp. 115–124. North-Holland, Amsterdam (1992)

How the Change is Made

- With **deterministic** parameter control, the parameter values are altered by a predetermined deterministic rule without using any feedback from the search (usually, a time-varying schedule is used)
- E.g., mutation probability can be time-varying[16]:

$$p_m(t) = \sqrt{\frac{c_1}{c_2}} \cdot \frac{e^{-c_3 \frac{t}{2}}}{\mu\sqrt{l}}$$

  where c_1, c_2, c_3 are user-defined constants, l is chromosome length, t is the generation number
- In general, deterministic schedules are of limited utility

16. Hesser, J., Maenner, R.: Investigation of the m-Heuristic for Optimal Mutation Probabilities. In: Maenner, R., Manderick, B. (eds.) Parallel Problem Solving from Nature 2, pp. 115–124. North-Holland, Amsterdam (1992)

How the Change is Made

- With **deterministic** parameter control, the parameter values are altered by a predetermined deterministic rule without using any feedback from the search (usually, a time-varying schedule is used)
- E.g., mutation probability can be time-varying[16]:

$$p_m(t) = \sqrt{\frac{c_1}{c_2}} \cdot \frac{e^{-c_3 \frac{t}{2}}}{\mu\sqrt{l}}$$

  where c_1, c_2, c_3 are user-defined constants, l is chromosome length, t is the generation number
- In general, deterministic schedules are of limited utility

16. Hesser, J., Maenner, R.: Investigation of the m-Heuristic for Optimal Mutation Probabilities. In: Maenner, R., Manderick, B. (eds.) Parallel Problem Solving from Nature 2, pp. 115–124. North-Holland, Amsterdam (1992)

How the Change is Made

- With **adaptive** parameter control, feedback from the search serves as input to a mechanism that determines the change of the parameter values
- E.g., we can sort fitnesses in descending order and calculate **dispersion rate**[17]:

$$\rho(t) = \frac{\sqrt{\frac{1}{\mu/2}\sum_{j=1}^{\mu/2}\left(f_j(t) - \overline{f(t)}\right)^2}}{\max_j f_j(t)}$$

  - $f_j(t)$ is the jth fitness value in the *sorted* array at generation t
  - $\overline{f(t)}$ is the *average* over the *first half* of the sorted array at generation t
- Values of $\rho \in [0.10; 0.15]$ are considered to be optimal
- To maintain ρ in this interval, we update mutation probability in each 5th generation by multiplying p_m by the following coefficient:

$$c(\overline{\rho}) = \begin{cases} 0.5\ , & \overline{\rho} \in [0.30; \infty) \\ 0.8\ , & \overline{\rho} \in [0.15; 0.30) \\ 1.0\ , & \overline{\rho} \in [0.10; 0.15) \\ 1.2\ , & \overline{\rho} \in [0.05; 0.10) \\ 2.0\ , & \overline{\rho} \in (-\infty; 0.05) \end{cases}$$

  where $\overline{\rho}$ is the value of $\rho(t)$ averaged over last 20 generations

17. Lis, J.: Genetic Algorithm with the Dynamic Probability of Mutation in the Classification Problem. Pattern Recognition Letters 16(12), 1311–1320 (1995)

How the Change is Made

- With **adaptive** parameter control, feedback from the search serves as input to a mechanism that determines the change of the parameter values
- E.g., we can sort fitnesses in descending order and calculate **dispersion rate**[17]:

$$\rho(t) = \frac{\sqrt{\frac{1}{\mu/2}\sum_{j=1}^{\mu/2}\left(f_j(t) - \overline{f(t)}\right)^2}}{\max_j f_j(t)}$$

  - $f_j(t)$ is the jth fitness value in the *sorted* array at generation t
  - $\overline{f(t)}$ is the *average* over the *first half* of the sorted array at generation t
- Values of $\rho \in [0.10; 0.15]$ are considered to be optimal
- To maintain ρ in this interval, we update mutation probability in each 5th generation by multiplying p_m by the following coefficient:

$$c(\overline{\rho}) = \begin{cases} 0.5\ , & \overline{\rho} \in [0.30; \infty) \\ 0.8\ , & \overline{\rho} \in [0.15; 0.30) \\ 1.0\ , & \overline{\rho} \in [0.10; 0.15) \\ 1.2\ , & \overline{\rho} \in [0.05; 0.10) \\ 2.0\ , & \overline{\rho} \in (-\infty; 0.05) \end{cases}$$

  where $\overline{\rho}$ is the value of $\rho(t)$ averaged over last 20 generations

17. Lis, J.: Genetic Algorithm with the Dynamic Probability of Mutation in the Classification Problem. Pattern Recognition Letters 16(12), 1311–1320 (1995)

How the Change is Made

- With **adaptive** parameter control, feedback from the search serves as input to a mechanism that determines the change of the parameter values
- E.g., we can sort fitnesses in descending order and calculate **dispersion rate**[17]:

$$\rho(t) = \frac{\sqrt{\frac{1}{\mu/2}\sum_{j=1}^{\mu/2}\left(f_j(t) - \overline{f(t)}\right)^2}}{\max_j f_j(t)}$$

  - $f_j(t)$ is the jth fitness value in the *sorted* array at generation t
  - $\overline{f(t)}$ is the *average* over the *first half* of the sorted array at generation t
- Values of $\rho \in [0.10; 0.15]$ are considered to be optimal
- To maintain ρ in this interval, we update mutation probability in each 5th generation by multiplying p_m by the following coefficient:

$$c(\overline{\rho}) = \begin{cases} 0.5\,, & \overline{\rho} \in [0.30; \infty) \\ 0.8\,, & \overline{\rho} \in [0.15; 0.30) \\ 1.0\,, & \overline{\rho} \in [0.10; 0.15) \\ 1.2\,, & \overline{\rho} \in [0.05; 0.10) \\ 2.0\,, & \overline{\rho} \in (-\infty; 0.05) \end{cases}$$

  where $\overline{\rho}$ is the value of $\rho(t)$ averaged over last 20 generations

17. Lis, J.: Genetic Algorithm with the Dynamic Probability of Mutation in the Classification Problem. Pattern Recognition Letters 16(12), 1311–1320 (1995)

How the Change is Made

- With **adaptive** parameter control, feedback from the search serves as input to a mechanism that determines the change of the parameter values
- E.g., we can sort fitnesses in descending order and calculate **dispersion rate**[17]:

$$\rho(t) = \frac{\sqrt{\frac{1}{\mu/2}\sum_{j=1}^{\mu/2}\left(f_j(t) - \overline{f(t)}\right)^2}}{\max_j f_j(t)}$$

  - $f_j(t)$ is the jth fitness value in the *sorted* array at generation t
  - $\overline{f(t)}$ is the *average* over the *first half* of the sorted array at generation t
- Values of $\rho \in [0.10; 0.15]$ are considered to be optimal
- To maintain ρ in this interval, we update mutation probability in each 5th generation by multiplying p_m by the following coefficient:

$$c(\overline{\rho}) = \begin{cases} 0.5\,, & \overline{\rho} \in [0.30; \infty) \\ 0.8\,, & \overline{\rho} \in [0.15; 0.30) \\ 1.0\,, & \overline{\rho} \in [0.10; 0.15) \\ 1.2\,, & \overline{\rho} \in [0.05; 0.10) \\ 2.0\,, & \overline{\rho} \in (-\infty; 0.05) \end{cases}$$

  where $\overline{\rho}$ is the value of $\rho(t)$ averaged over last 20 generations

17. Lis, J.: Genetic Algorithm with the Dynamic Probability of Mutation in the Classification Problem. Pattern Recognition Letters 16(12), 1311–1320 (1995)

How the Change is Made

- With **adaptive** parameter control, feedback from the search serves as input to a mechanism that determines the change of the parameter values
- E.g., we can sort fitnesses in descending order and calculate **dispersion rate**[17]:

$$\rho(t) = \frac{\sqrt{\frac{1}{\mu/2}\sum_{j=1}^{\mu/2}\left(f_j(t) - \overline{f(t)}\right)^2}}{\max_j f_j(t)}$$

  - $f_j(t)$ is the jth fitness value in the *sorted* array at generation t
  - $\overline{f(t)}$ is the *average* over the *first half* of the sorted array at generation t
- Values of $\rho \in [0.10; 0.15]$ are considered to be optimal
- To maintain ρ in this interval, we update mutation probability in each 5th generation by multiplying p_m by the following coefficient:

$$c(\overline{\rho}) = \begin{cases} 0.5\ , & \overline{\rho} \in [0.30; \infty) \\ 0.8\ , & \overline{\rho} \in [0.15; 0.30) \\ 1.0\ , & \overline{\rho} \in [0.10; 0.15) \\ 1.2\ , & \overline{\rho} \in [0.05; 0.10) \\ 2.0\ , & \overline{\rho} \in (-\infty; 0.05) \end{cases}$$

  where $\overline{\rho}$ is the value of $\rho(t)$ averaged over last 20 generations

17. Lis, J.: Genetic Algorithm with the Dynamic Probability of Mutation in the Classification Problem. Pattern Recognition Letters 16(12), 1311–1320 (1995)

How the Change is Made

- With **adaptive** parameter control, feedback from the search serves as input to a mechanism that determines the change of the parameter values
- E.g., we can sort fitnesses in descending order and calculate **dispersion rate**[17]:

$$\rho(t) = \frac{\sqrt{\frac{1}{\mu/2}\sum_{j=1}^{\mu/2}\left(f_j(t) - \overline{f(t)}\right)^2}}{\max_j f_j(t)}$$

  - $f_j(t)$ is the jth fitness value in the *sorted* array at generation t
  - $\overline{f(t)}$ is the *average* over the *first half* of the sorted array at generation t
- Values of $\rho \in [0.10; 0.15]$ are considered to be optimal
- To maintain ρ in this interval, we update mutation probability in each 5th generation by multiplying p_m by the following coefficient:

$$c(\overline{\rho}) = \begin{cases} 0.5\ , & \overline{\rho} \in [0.30; \infty) \\ 0.8\ , & \overline{\rho} \in [0.15; 0.30) \\ 1.0\ , & \overline{\rho} \in [0.10; 0.15) \\ 1.2\ , & \overline{\rho} \in [0.05; 0.10) \\ 2.0\ , & \overline{\rho} \in (-\infty; 0.05) \end{cases}$$

where $\overline{\rho}$ is the value of $\rho(t)$ averaged over last 20 generations

17. Lis, J.: Genetic Algorithm with the Dynamic Probability of Mutation in the Classification Problem. Pattern Recognition Letters 16(12), 1311–1320 (1995)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f}, & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}}, & f_i > \overline{f} \end{cases}$$

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f}, & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}}, & f_i > \overline{f} \end{cases}$$

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f}, & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}}, & f_i > \overline{f} \end{cases}$$

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f}, & f_i \leq \overline{f} \\ 0.5(m+M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}}, & f_i > \overline{f} \end{cases}$$

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f} , & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}} , & f_i > \overline{f} \end{cases}$$

  - m and M and the smallest and biggest lifetimes that can possibly be allowed (specified by the user)
  - $\eta = 0.5(M - m)$
  - $\overline{f}$ is the average over all fitnesses

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f} , & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}} , & f_i > \overline{f} \end{cases}$$

- m and M and the smallest and biggest lifetimes that can possibly be allowed (specified by the user)
- $\eta = 0.5(M - m)$
- $\overline{f}$ is the average over all fitnesses

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f} , & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}} , & f_i > \overline{f} \end{cases}$$

  - m and M and the smallest and biggest lifetimes that can possibly be allowed (specified by the user)
  - $\eta = 0.5(M - m)$
  - $\overline{f}$ is the average over all fitnesses

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f} , & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}} , & f_i > \overline{f} \end{cases}$$

  - m and M and the smallest and biggest lifetimes that can possibly be allowed (specified by the user)
  - $\eta = 0.5(M - m)$
  - $\overline{f}$ is the average over all fitnesses

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- Another example is the Genetic Algorithm with Varying Population Size (GAVAPS)[18]
- Each individual is assigned some **lifetime**
- The idea is that on each new generation every individual becomes *older*: its age is increased by 1
- When an individual reaches maximum lifetime, it is removed from the population
- Authors of this approach proposed to perform parent selection absolutely at random
- Authors suggest allocating lifetime T_i for individual i as follows:

$$T_i = \begin{cases} m + \eta \frac{f_i - \min f}{\overline{f} - \min f} , & f_i \leq \overline{f} \\ 0.5(m + M) + \eta \frac{f_i - \overline{f}}{\max f - \overline{f}} , & f_i > \overline{f} \end{cases}$$

  - m and M and the smallest and biggest lifetimes that can possibly be allowed (specified by the user)
  - $\eta = 0.5(M - m)$
  - $\overline{f}$ is the average over all fitnesses

18. Arabas, J., Michalewicz, Z., Mulawka, J.: GAVaPS — a Genetic Algorithm with Varying Population Size. In: Michalewicz, Z., Schaffer, D., Schwefel, H.-P., Fogel, D., Kitano, H. (eds.) Proceedings of the First IEEE International Conference on Evolutionary Computation, pp. 73–78 (1994)

How the Change is Made

- With **self-adaptive** parameter control, the *evolution of evolution* is used to change the values
- Parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination
- When we discuss *evolution strategies* next time, we will see how self-adaptation is implemented there
- In the context of GA, we can briefly mention genetic algorithms "without parameters"[19]
- It has variable population size just as in GAVAPS, with $m = 1$ and $M = 11$
- However, the age of the best individual in every generation is **not** changed
- Selection is always tournament selection, crossover is always uniform crossover

19. Baeck, T., Eiben, A.E., van der Vaart, N.A.L.: An Empirical Study on GAs "Without Parameters." In: Schoenauer, M. et al. (eds.) Proceedings of the 6th Conference on Parallel Problem Solving from Nature, pp. 315–324. Springer-Verlag, Berlin, Heidelberg (2000)

How the Change is Made

- With **self-adaptive** parameter control, the *evolution of evolution* is used to change the values
- Parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination
- When we discuss *evolution strategies* next time, we will see how self-adaptation is implemented there
- In the context of GA, we can briefly mention genetic algorithms "without parameters"[19]
- It has variable population size just as in GAVAPS, with $m = 1$ and $M = 11$
- However, the age of the best individual in every generation is **not** changed
- Selection is always tournament selection, crossover is always uniform crossover

19. Baeck, T., Eiben, A.E., van der Vaart, N.A.L.: An Empirical Study on GAs "Without Parameters." In: Schoenauer, M. et al. (eds.) Proceedings of the 6th Conference on Parallel Problem Solving from Nature, pp. 315–324. Springer-Verlag, Berlin, Heidelberg (2000)

How the Change is Made

- With **self-adaptive** parameter control, the *evolution of evolution* is used to change the values
- Parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination
- When we discuss *evolution strategies* next time, we will see how self-adaptation is implemented there
- In the context of GA, we can briefly mention genetic algorithms "without parameters"[19]
- It has variable population size just as in GAVAPS, with $m = 1$ and $M = 11$
- However, the age of **the best** individual in every generation is **not** changed
- Selection is always tournament selection, crossover is always uniform crossover

19. Baeck, T., Eiben, A.E., van der Vaart, N.A.L.: An Empirical Study on GAs "Without Parameters." In: Schoenauer, M. et al. (eds.) Proceedings of the 6th Conference on Parallel Problem Solving from Nature, pp. 315–324. Springer-Verlag, Berlin, Heidelberg (2000)

How the Change is Made

- With **self-adaptive** parameter control, the *evolution of evolution* is used to change the values
- Parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination
- When we discuss *evolution strategies* next time, we will see how self-adaptation is implemented there
- In the context of GA, we can briefly mention genetic algorithms "without parameters"[19]
- It has variable population size just as in GAVAPS, with $m = 1$ and $M = 11$
- However, the age of **the best** individual in every generation is **not** changed
- Selection is always tournament selection, crossover is always uniform crossover

19. Baeck, T., Eiben, A.E., van der Vaart, N.A.L.: An Empirical Study on GAs "Without Parameters." In: Schoenauer, M. et al. (eds.) Proceedings of the 6th Conference on Parallel Problem Solving from Nature, pp. 315–324. Springer-Verlag, Berlin, Heidelberg (2000)

How the Change is Made

- With **self-adaptive** parameter control, the *evolution of evolution* is used to change the values
- Parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination
- When we discuss *evolution strategies* next time, we will see how self-adaptation is implemented there
- In the context of GA, we can briefly mention genetic algorithms "without parameters"[19]
- It has variable population size just as in GAVAPS, with $m = 1$ and $M = 11$
- However, the age of **the best** individual in every generation is **not** changed
- Selection is always tournament selection, crossover is always uniform crossover

19. Baeck, T., Eiben, A.E., van der Vaart, N.A.L.: An Empirical Study on GAs "Without Parameters." In: Schoenauer, M. et al. (eds.) Proceedings of the 6th Conference on Parallel Problem Solving from Nature, pp. 315–324. Springer-Verlag, Berlin, Heidelberg (2000)

How the Change is Made

- With **self-adaptive** parameter control, the *evolution of evolution* is used to change the values
- Parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination
- When we discuss *evolution strategies* next time, we will see how self-adaptation is implemented there
- In the context of GA, we can briefly mention genetic algorithms "without parameters"[19]
- It has variable population size just as in GAVAPS, with $m = 1$ and $M = 11$
- However, the age of **the best** individual in every generation is **not** changed
- Selection is always tournament selection, crossover is always uniform crossover

19. Baeck, T., Eiben, A.E., van der Vaart, N.A.L.: An Empirical Study on GAs "Without Parameters." In: Schoenauer, M. et al. (eds.) Proceedings of the 6th Conference on Parallel Problem Solving from Nature, pp. 315–324. Springer-Verlag, Berlin, Heidelberg (2000)

How the Change is Made

- With **self-adaptive** parameter control, the *evolution of evolution* is used to change the values
- Parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination
- When we discuss *evolution strategies* next time, we will see how self-adaptation is implemented there
- In the context of GA, we can briefly mention genetic algorithms "without parameters"[19]
- It has variable population size just as in GAVAPS, with $m = 1$ and $M = 11$
- However, the age of **the best** individual in every generation is **not** changed
- Selection is always tournament selection, crossover is always uniform crossover

19. Baeck, T., Eiben, A.E., van der Vaart, N.A.L.: An Empirical Study on GAs "Without Parameters." In: Schoenauer, M. et al. (eds.) Proceedings of the 6th Conference on Parallel Problem Solving from Nature, pp. 315–324. Springer-Verlag, Berlin, Heidelberg (2000)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**.

- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**.

- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:

- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:
  - If both parents are ready to mate, they are crossed over
  - If both parents are ready to mate, they are not crossed over, but they undergo mutation
  - If one parent is ready to mate, and the other one is not, then the latter one is ready to be mutated, and the one that is ready is put on hold and undergoes the next generation one parent closer to reproduction
- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:
  - If both parents are ready to mate, they are crossed over
  - If both parents are not ready to mate, they are not crossed over, but only undergo mutation
  - If one parent is ready to mate, and the other one is not, then the latter (not ready to mate) is mutated, and the one that is ready to mate waits until the next generation, one parent here is selected
- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:
  - If both parents are ready to mate, they are crossed over
  - If both parents are not ready to mate, they are not crossed over, but only undergo mutation
  - If one parent is ready to mate, and the other one is not, than the one that is not ready is mutated, and the one that is ready is put on hold, and on the next generation one parent less is selected
- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:
  - If both parents are ready to mate, they are crossed over
  - If both parents are not ready to mate, they are not crossed over, but only undergo mutation
  - If one parent is ready to mate, and the other one is not, than the one that is not ready is mutated, and the one that is ready is put on hold, and on the next generation one parent less is selected
- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:
  - If both parents are ready to mate, they are crossed over
  - If both parents are not ready to mate, they are not crossed over, but only undergo mutation
  - If one parent is ready to mate, and the other one is not, than the one that is not ready is mutated, and the one that is ready is put on hold, and on the next generation one parent less is selected
- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:
  - If both parents are ready to mate, they are crossed over
  - If both parents are not ready to mate, they are not crossed over, but only undergo mutation
  - If one parent is ready to mate, and the other one is not, than the one that is not ready is mutated, and the one that is ready is put on hold, and on the next generation one parent less is selected
- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

How the Change is Made

- Mutation and crossover probabilities are also self-adapted: at the tail of every individual, 10 bits are added to code crossover probability, and then 10 bits are added to code mutation probability
- Crossover probability can range between 0 and 1, mutation probability can range between 0.001 and 0.250
- When performing mutation, first the 10 bits that encode mutation probability are mutated with the constant metamutation probability (provided by the user)
- Then, the mutated 10 bits are decoded to obtain mutation probability for the given individual. The individual is mutated using these new probabilities (except for 20 last bits)
- Likewise, when performing crossover, the 10 bits that encode crossover probability are decoded to obtain the probability
- A random number is drawn uniformly between 0 and 1. If it is lower than the crossover probability, the parent is said to be **ready to mate**:
  - If both parents are ready to mate, they are crossed over
  - If both parents are not ready to mate, they are not crossed over, but only undergo mutation
  - If one parent is ready to mate, and the other one is not, than the one that is not ready is mutated, and the one that is ready is put on hold, and on the next generation one parent less is selected
- We get rid of specifying some parameters at the cost of specifying metaparameters (which are easier to choose and more robust, though)

Outline

1. Applying Genetic Algorithms to Real-Valued Optimization
2. Beyond Fitness-Proportionate Selection
3. Ideas for Multimodal Optimization
4. Beyond Binary Representations
5. Parameter Control in Evolutionary Algorithms
6. Constraint Handling

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \ , \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \ , \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \ , \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \ , \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0\,, \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0\,, \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:
  - Penalty functions
  - Decoders
  - Repair functions
  - Constraint-preserving operators

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \,, \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \,, \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:
  - Penalty functions
  - Decoders
  - Repair functions
  - Constraint-preserving operators

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \,, \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \,, \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:
  - Penalty functions
  - Decoders
  - Repair functions
  - Constraint-preserving operators

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \,, \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \,, \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:
  - Penalty functions
  - Decoders
  - Repair functions
  - Constraint-preserving operators

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \,, \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \,, \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:
  - Penalty functions
  - Decoders
  - Repair functions
  - Constraint-preserving operators

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \,, \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \,, \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:
  - Penalty functions
  - Decoders
  - Repair functions
  - Constraint-preserving operators

Motivation

- Up to this point, we considered only unconstrained optimization problems
- In reality, many problems are in fact constrained
- We need to optimize some function $f(\mathbf{x})$ of many arguments subject to certain constraints:

$$g_j(\mathbf{x}) \leq 0 \,, \quad j = 1, \ldots, q$$
$$h_j(\mathbf{x}) = 0 \,, \quad j = q+1, \ldots, m$$

- Genetic algorithms need to be modified to take into account the constraints
- Constraint handling techniques can be classified into four broad categories:
  - Penalty functions
  - Decoders
  - Repair functions
  - Constraint-preserving operators

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
  - Number of constraints that are violated by $\mathbf{x}$
  - Euclidean distance between $\mathbf{x}$ and B
  - Sum of individual constraint violations, possibly raised to a power k
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
  - Number of constraints that are violated by $\mathbf{x}$
  - Euclidean distance between $\mathbf{x}$ and B
  - Sum of individual constraint violations, possibly raised to a power k
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
  - Number of constraints that are violated by $\mathbf{x}$
  - Euclidean distance between $\mathbf{x}$ and B
  - Sum of individual constraint violations, possibly raised to power k
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
  - Number of constraints that are violated by $\mathbf{x}$
  - Euclidean distance between $\mathbf{x}$ and B
  - Sum of individual constraint violations, possibly raised to power k
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
  - Number of constraints that are violated by $\mathbf{x}$
  - Euclidean distance between $\mathbf{x}$ and B
  - Sum of individual constraint violations, possibly raised to power k
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
  - Number of constraints that are violated by $\mathbf{x}$
  - Euclidean distance between $\mathbf{x}$ and B
  - Sum of individual constraint violations, possibly raised to power k
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Penalty Functions

- Let us consider minimization problems
- The idea is to change[20] the objective function from $f(\mathbf{x})$ to

$$f_p(\mathbf{x}) = f(\mathbf{x}) + p(d(\mathbf{x}, B))$$

- Here B is the **feasible region**
- d is some distance metric:
  - Number of constraints that are violated by $\mathbf{x}$
  - Euclidean distance between $\mathbf{x}$ and B
  - Sum of individual constraint violations, possibly raised to power k
- p is the penalty function, which can be static, dynamic, or adaptive

20. Smith, A.E., Coit, D.W.: Penalty Functions. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 41–48. Institute of Physics Publishing, Bristol (2000)

Static Penalty Functions

- Static penalty functions can look like

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} C_i \left(d_i(\mathbf{x})\right)^k$$

- Here

$$d_i(\mathbf{x}) = \begin{cases} \delta_i g_i(\mathbf{x}) \,, & i = 1, \ldots, q \\ |h_i(\mathbf{x})| \,, & i = q+1, \ldots, m \end{cases}$$

- Here δ_i is an indicator of whether ith constraint is violated, C_i are some weights
- **Disadvantage**: necessary to choose weights, which is not easy in most cases

Static Penalty Functions

- Static penalty functions can look like

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} C_i \left(d_i(\mathbf{x})\right)^k$$

- Here

$$d_i(\mathbf{x}) = \begin{cases} \delta_i g_i(\mathbf{x}) \,, & i = 1, \ldots, q \\ |h_i(\mathbf{x})| \,, & i = q+1, \ldots, m \end{cases}$$

- Here δ_i is an indicator of whether ith constraint is violated, C_i are some weights
- **Disadvantage**: necessary to choose weights, which is not easy in most cases

Static Penalty Functions

- Static penalty functions can look like

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} C_i \left(d_i(\mathbf{x})\right)^k$$

- Here

$$d_i(\mathbf{x}) = \begin{cases} \delta_i g_i(\mathbf{x}) \,, & i = 1, \ldots, q \\ |h_i(\mathbf{x})| \,, & i = q+1, \ldots, m \end{cases}$$

- Here δ_i is an indicator of whether ith constraint is violated, C_i are some weights
- **Disadvantage**: necessary to choose weights, which is not easy in most cases

Static Penalty Functions

- Static penalty functions can look like

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} C_i \left(d_i(\mathbf{x})\right)^k$$

- Here

$$d_i(\mathbf{x}) = \begin{cases} \delta_i g_i(\mathbf{x}) \,, & i = 1, \ldots, q \\ |h_i(\mathbf{x})| \,, & i = q+1, \ldots, m \end{cases}$$

- Here δ_i is an indicator of whether ith constraint is violated, C_i are some weights
- **Disadvantage**: necessary to choose weights, which is not easy in most cases

Dynamic Penalty Functions

- Dynamic penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} s_i(t) \left(d_i(\mathbf{x})\right)^k$$

- Here the only difference is that s_i is a monotonically non-decreasing function, and t is either the generation number or the number of objective function evaluations
- The severity of the penalty should be increased with time
- Then, highly infeasible solutions can survive early during the exploration phase, but eventually the solution will be moved to the feasible region
- If s_i is too weak, we can get infeasible solutions; otherwise, the search may converge to non-optimal feasible solutions
- Therefore, tuning is needed for each problem
- One example is as follows:

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} (C_i t)^\alpha \left(d_i(\mathbf{x})\right)^k, \quad \alpha \in \{1, 2\}$$

Dynamic Penalty Functions

- Dynamic penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} s_i(t) \left(d_i(\mathbf{x})\right)^k$$

- Here the only difference is that s_i is a monotonically non-decreasing function, and t is either the generation number or the number of objective function evaluations
- The severity of the penalty should be increased with time
- Then, highly infeasible solutions can survive early during the exploration phase, but eventually the solution will be moved to the feasible region
- If s_i is too weak, we can get infeasible solutions; otherwise, the search may converge to non-optimal feasible solutions
- Therefore, tuning is needed for each problem
- One example is as follows:

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} (C_i t)^\alpha \left(d_i(\mathbf{x})\right)^k \quad , \quad \alpha \in \{1, 2\}$$

Dynamic Penalty Functions

- Dynamic penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} s_i(t) \left(d_i(\mathbf{x})\right)^k$$

- Here the only difference is that s_i is a monotonically non-decreasing function, and t is either the generation number or the number of objective function evaluations
- The severity of the penalty should be increased with time
- Then, highly infeasible solutions can survive early during the exploration phase, but eventually the solution will be moved to the feasible region
- If s_i is too weak, we can get infeasible solutions; otherwise, the search may converge to non-optimal feasible solutions
- Therefore, tuning is needed for each problem
- One example is as follows:

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} (C_i t)^\alpha \left(d_i(\mathbf{x})\right)^k, \quad \alpha \in \{1, 2\}$$

Dynamic Penalty Functions

- Dynamic penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} s_i(t) \left(d_i(\mathbf{x})\right)^k$$

- Here the only difference is that s_i is a monotonically non-decreasing function, and t is either the generation number or the number of objective function evaluations
- The severity of the penalty should be increased with time
- Then, highly infeasible solutions can survive early during the exploration phase, but eventually the solution will be moved to the feasible region
- If s_i is too weak, we can get infeasible solutions; otherwise, the search may converge to non-optimal feasible solutions
- Therefore, tuning is needed for each problem
- One example is as follows:

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} (C_i t)^{\alpha} \left(d_i(\mathbf{x})\right)^k \ , \quad \alpha \in \{1, 2\}$$

Dynamic Penalty Functions

- Dynamic penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} s_i(t) \left(d_i(\mathbf{x})\right)^k$$

- Here the only difference is that s_i is a monotonically non-decreasing function, and t is either the generation number or the number of objective function evaluations
- The severity of the penalty should be increased with time
- Then, highly infeasible solutions can survive early during the exploration phase, but eventually the solution will be moved to the feasible region
- If s_i is too weak, we can get infeasible solutions; otherwise, the search may converge to non-optimal feasible solutions
- Therefore, tuning is needed for each problem
- One example is as follows:

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} (C_i t)^{\alpha} \left(d_i(\mathbf{x})\right)^k \ , \quad \alpha \in \{1, 2\}$$

Dynamic Penalty Functions

- Dynamic penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} s_i(t) \left(d_i(\mathbf{x})\right)^k$$

- Here the only difference is that s_i is a monotonically non-decreasing function, and t is either the generation number or the number of objective function evaluations
- The severity of the penalty should be increased with time
- Then, highly infeasible solutions can survive early during the exploration phase, but eventually the solution will be moved to the feasible region
- If s_i is too weak, we can get infeasible solutions; otherwise, the search may converge to non-optimal feasible solutions
- Therefore, tuning is needed for each problem
- One example is as follows:

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} (C_i t)^{\alpha} \left(d_i(\mathbf{x})\right)^k \ , \quad \alpha \in \{1, 2\}$$

Dynamic Penalty Functions

- Dynamic penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} s_i(t) \left(d_i(\mathbf{x})\right)^k$$

- Here the only difference is that s_i is a monotonically non-decreasing function, and t is either the generation number or the number of objective function evaluations
- The severity of the penalty should be increased with time
- Then, highly infeasible solutions can survive early during the exploration phase, but eventually the solution will be moved to the feasible region
- If s_i is too weak, we can get infeasible solutions; otherwise, the search may converge to non-optimal feasible solutions
- Therefore, tuning is needed for each problem
- One example is as follows:

$$f_p(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{m} (C_i t)^{\alpha} \left(d_i(\mathbf{x})\right)^k \ , \quad \alpha \in \{1, 2\}$$

Adaptive Penalty Functions

- Adaptive penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda(t) \left(d_i(\mathbf{x})\right)^k$$

- Here,

$$\lambda(t+1) = \begin{cases} \lambda(t) \cdot \beta_1 \ , & \text{previous } N_f \text{ generations have only infeasible best solution} \\ \lambda(t)/\beta_2 \ , & \text{previous } N_f \text{ generations have only feasible best solution} \\ \lambda(t) \ , & \text{otherwise} \end{cases}$$

- Here $\beta_1 > \beta_2 > 1$ are user-defined constants, N_f is the user-defined number of generations
- In this case, information about the search is taken into account to update the penalty weights

Adaptive Penalty Functions

- Adaptive penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda(t) \left(d_i(\mathbf{x})\right)^k$$

- Here,

$$\lambda(t+1) = \begin{cases} \lambda(t) \cdot \beta_1 \ , & \text{previous } N_f \text{ generations have only infeasible best solution} \\ \lambda(t)/\beta_2 \ , & \text{previous } N_f \text{ generations have only feasible best solution} \\ \lambda(t) \ , & \text{otherwise} \end{cases}$$

- Here $\beta_1 > \beta_2 > 1$ are user-defined constants, N_f is the user-defined number of generations
- In this case, information about the search is taken into account to update the penalty weights

Adaptive Penalty Functions

- Adaptive penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda(t) \left(d_i(\mathbf{x})\right)^k$$

- Here,

$$\lambda(t+1) = \begin{cases} \lambda(t) \cdot \beta_1 \ , & \text{previous } N_f \text{ generations have only infeasible best solution} \\ \lambda(t)/\beta_2 \ , & \text{previous } N_f \text{ generations have only feasible best solution} \\ \lambda(t) \ , & \text{otherwise} \end{cases}$$

- Here $\beta_1 > \beta_2 > 1$ are user-defined constants, N_f is the user-defined number of generations
- In this case, information about the search is taken into account to update the penalty weights

Adaptive Penalty Functions

- Adaptive penalty functions can look like

$$f_p(\mathbf{x}, t) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda(t) \left(d_i(\mathbf{x})\right)^k$$

- Here,

$$\lambda(t+1) = \begin{cases} \lambda(t) \cdot \beta_1 \ , & \text{previous } N_f \text{ generations have only infeasible best solution} \\ \lambda(t)/\beta_2 \ , & \text{previous } N_f \text{ generations have only feasible best solution} \\ \lambda(t) \ , & \text{otherwise} \end{cases}$$

- Here $\beta_1 > \beta_2 > 1$ are user-defined constants, N_f is the user-defined number of generations
- In this case, information about the search is taken into account to update the penalty weights

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector x such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector x such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Then, any bit string will translate to a feasible solution
  - Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Then, any bit string will translate to a feasible solution
  - Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Sort all the items according to their profit/weight ratio
  - Analyze the chromosome from left to right and select items for the knapsack if they fit
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Sort all the items according to their profit/weight ration p_i/w_i in descending order
  - Analyze the chromosome from left to right, and select item i if ith bit is 1, and if the item fits
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Sort all the items according to their profit/weight ration p_i/w_i in descending order
  - Analyze the chromosome from left to right, and select item i if ith bit is 1, and **if the item fits**
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Sort all the items according to their profit/weight ration p_i/w_i in descending order
  - Analyze the chromosome from left to right, and select item i if ith bit is 1, and **if the item fits**
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Sort all the items according to their profit/weight ration p_i/w_i in descending order
  - Analyze the chromosome from left to right, and select item i if ith bit is 1, and **if the item fits**
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Sort all the items according to their profit/weight ration p_i/w_i in descending order
  - Analyze the chromosome from left to right, and select item i if ith bit is 1, and **if the item fits**
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Decoders

- Let us discuss a different way to handle constraints[21]
- The chromosome "gives instructions" to a **decoder** on how to build a feasible solution
- Stated formally, a decoder is a mapping from a representation space to the solution space
- It is important that several conditions are satisfied:
  - Each solution has a representation
  - Each representation corresponds to a feasible solution
  - All solutions should be represented by the same number of representations
- E.g., consider the **knapsack problem**: for a given set of weights w_i, profits p_i, and capacity C, find a binary vector $\mathbf{x}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Obviously, if we represent the solutions as binary strings where ith bit is 1 only when we select ith item to place in the knapsack, and 0 otherwise, many solutions will be infeasible
- We can apply the following decoder:
  - Sort all the items according to their profit/weight ration p_i/w_i in descending order
  - Analyze the chromosome from left to right, and select item i if ith bit is 1, and **if the item fits**
- Then, any bit string will translate to a feasible solution
- Of course large parts of chromosomes would be "junk"

21. Michalewicz, Z.: Decoders. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 49–55. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene z is selected and set to 0
- Genes for removal from the knapsack can be selected:

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene z is selected and set to 0
- Genes for removal from the knapsack can be selected:

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene z is selected and set to 0
- Genes for removal from the knapsack can be selected:

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
- Genes for removal from the knapsack can be selected:

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:
  - Uniformly at random
  - Deterministically from the left
  - Deterministically according to the value of i
  - Stochastically, based on a probability

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:
  - Uniformly at random
  - Deterministically from the left
  - Deterministically according to $\frac{p_i}{w_i}$
  - Stochastically, based on $\frac{p_i}{w_i}$

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:
  - Uniformly at random
  - Deterministically from the left
  - Deterministically according to p_i/w_i
  - Stochastically based on p_i/w_i

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:
  - Uniformly at random
  - Deterministically from the left
  - Deterministically according to p_i/w_i
  - Stochastically based on p_i/w_i

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:
  - Uniformly at random
  - Deterministically from the left
  - Deterministically according to p_i/w_i
  - Stochastically based on p_i/w_i

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:
  - Uniformly at random
  - Deterministically from the left
  - Deterministically according to p_i/w_i
  - Stochastically based on p_i/w_i

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Repair Algorithms

- For many combinatorial optimization problems (knapsack, traveling salesman, set covering, etc.), it is relatively easy to repair the solution[22]
- The repaired solution can be:
  - Used for fitness calculation only but not inserted in the population (Baldwin effect)
  - Used for fitness calculation and inserted in the population (Lamarckism)
- Usually, some mix is used, where the second option is applied with a certain probability
- In fact, this is equivalent to the local search, which will be discussed later in the course in the context of memetic algorithms
- E.g., for the knapsack problem, if a solution is infeasible, a certain gene i is selected and set to 0
  - This goes on until the solution becomes feasible again
- Genes for removal from the knapsack can be selected:
  - Uniformly at random
  - Deterministically from the left
  - Deterministically according to p_i/w_i
  - Stochastically based on p_i/w_i

22. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 56–61. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal their corresponding
  - For instance, let us have the following constraints:
    - Sources: (8, 4, 12, 6)
    - Destinations: (3, 5, 10, 12)
    - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
    - Sources: (8, 4, 12, 6)
    - Destinations: (6, 6, 10, 8)
    - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
  - Sources: (8, 4, 12, 6)
  - Destinations: (3, 5, 10, 7, 5)
  - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
  - Sources: (8, 4, 12, 6)
  - Destinations: (3, 5, 10, 7, 5)
  - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
  - Sources: (8, 4, 12, 6)
  - Destinations: (3, 5, 10, 7, 5)
  - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
  - Sources: (8, 4, 12, 6)
  - Destinations: (3, 5, 10, 7, 5)
  - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
  - Sources: (8, 4, 12, 6)
  - Destinations: (3, 5, 10, 7, 5)
  - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
  - Sources: (8, 4, 12, 6)
  - Destinations: (3, 5, 10, 7, 5)
  - Example of a feasible solution:

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- Certain specialized operators can be used that incorporate problem-specific knowledge[23]
- This approach is widely used in practice
- We already saw this idea when we were talking about crossovers and mutations for permutation representations
- E.g., consider the transportation problem:
  - A solution matrix is feasible if all the marginal sums equal the constraints
  - For instance, let us have the following constraints
  - Sources: (8, 4, 12, 6)
  - Destinations: (3, 5, 10, 7, 5)
  - Example of a feasible solution:

0.0	0.0	**5.0**	0.0	**3.0**
0.0	4.0	0.0	0.0	0.0
0.0	0.0	**5.0**	7.0	**0.0**
3.0	1.0	0.0	0.0	2.0

23. Michalewicz, Z.: Repair Algorithms. In: Baeck, T., Fogel, D.B., Michalewicz, Z. (eds.) Evolutionary Computation 2. Advanced Algorithms and Operators, pp. 62–68. Institute of Physics Publishing, Bristol (2000)

Constraint-Preserving Operators

- We can devise at least two different mutation operators that would preserve the feasibility
- In the **first one**, we select a random submatrix, e.g. rows 1 and 3 and columns 1, 3, and 5 (**bold** on the previous slide)
- The values in this submatrix are changed to introduce *as many zeros as possible*, leaving marginal totals unchanged
- We can get something like:

- In the **second mutation**, we do the same thing but introduce *as few zeros as possible*
- A **crossover** that can be used in this case is to create children (C_1, C_2) by *averaging out* the two parents (P_1, P_2) as follows:

$$C_1 = aP_1 + bP_2$$
$$C_2 = aP_2 + bP_1$$

where $c_1 + c_2 = 1$, $c_1, c_2 > 0$

Constraint-Preserving Operators

- We can devise at least two different mutation operators that would preserve the feasibility
- In the **first one**, we select a random submatrix, e.g. rows 1 and 3 and columns 1, 3, and 5 (**bold** on the previous slide)
- The values in this submatrix are changed to introduce *as many zeros as possible*, leaving marginal totals unchanged
- We can get something like:

- In the **second mutation**, we do the same thing but introduce *as few zeros as possible*
- A **crossover** that can be used in this case is to create children (C_1, C_2) by *averaging out* the two parents (P_1, P_2) as follows:

$$C_1 = aP_1 + bP_2$$
$$C_2 = aP_2 + bP_1$$

where $c_1 + c_2 = 1$, $c_1, c_2 > 0$

Constraint-Preserving Operators

- We can devise at least two different mutation operators that would preserve the feasibility
- In the **first one**, we select a random submatrix, e.g. rows 1 and 3 and columns 1, 3, and 5 (**bold** on the previous slide)
- The values in this submatrix are changed to introduce *as many zeros as possible*, leaving marginal totals unchanged
- We can get something like:

- In the **second mutation**, we do the same thing but introduce *as few zeros as possible*
- A **crossover** that can be used in this case is to create children (C_1, C_2) by *averaging out* the two parents (P_1, P_2) as follows:

$$C_1 = aP_1 + bP_2$$
$$C_2 = aP_2 + bP_1$$

where $c_1 + c_2 = 1$, $c_1, c_2 > 0$

Constraint-Preserving Operators

- We can devise at least two different mutation operators that would preserve the feasibility
- In the **first one**, we select a random submatrix, e.g. rows 1 and 3 and columns 1, 3, and 5 (**bold** on the previous slide)
- The values in this submatrix are changed to introduce *as many zeros as possible*, leaving marginal totals unchanged
- We can get something like:

- In the **second mutation**, we do the same thing but introduce *as few zeros as possible*
- A **crossover** that can be used in this case is to create children (C_1, C_2) by *averaging out* the two parents (P_1, P_2) as follows:

$$C_1 = aP_1 + bP_2$$

$$C_2 = aP_2 + bP_1$$

where $c_1 + c_2 = 1$, $c_1, c_2 > 0$

Constraint-Preserving Operators

- We can devise at least two different mutation operators that would preserve the feasibility
- In the **first one**, we select a random submatrix, e.g. rows 1 and 3 and columns 1, 3, and 5 (**bold** on the previous slide)
- The values in this submatrix are changed to introduce *as many zeros as possible*, leaving marginal totals unchanged
- We can get something like:

0.0	0.0	**8.0**	0.0	**0.0**
0.0	4.0	0.0	0.0	0.0
0.0	0.0	**2.0**	7.0	**3.0**
3.0	1.0	0.0	0.0	2.0

- In the **second mutation**, we do the same thing but introduce *as few zeros as possible*
- A **crossover** that can be used in this case is to create children (C_1, C_2) by *averaging out* the two parents (P_1, P_2) as follows:

$$C_1 = aP_1 + bP_2$$

$$C_2 = aP_2 + bP_1$$

where $c_1 + c_2 = 1$, $c_1, c_2 > 0$

Constraint-Preserving Operators

- We can devise at least two different mutation operators that would preserve the feasibility
- In the **first one**, we select a random submatrix, e.g. rows 1 and 3 and columns 1, 3, and 5 (**bold** on the previous slide)
- The values in this submatrix are changed to introduce *as many zeros as possible*, leaving marginal totals unchanged
- We can get something like:

0.0	0.0	**8.0**	0.0	**0.0**
0.0	4.0	0.0	0.0	0.0
0.0	0.0	**2.0**	7.0	**3.0**
3.0	1.0	0.0	0.0	2.0

- In the **second mutation**, we do the same thing but introduce *as few zeros as possible*
- A **crossover** that can be used in this case is to create children (C_1, C_2) by *averaging out* the two parents (P_1, P_2) as follows:

$$C_1 = aP_1 + bP_2$$

$$C_2 = aP_2 + bP_1$$

where $c_1 + c_2 = 1$, $c_1, c_2 > 0$

Constraint-Preserving Operators

- We can devise at least two different mutation operators that would preserve the feasibility
- In the **first one**, we select a random submatrix, e.g. rows 1 and 3 and columns 1, 3, and 5 (**bold** on the previous slide)
- The values in this submatrix are changed to introduce *as many zeros as possible*, leaving marginal totals unchanged
- We can get something like:

0.0	0.0	**8.0**	0.0	**0.0**
0.0	4.0	0.0	0.0	0.0
0.0	0.0	**2.0**	7.0	**3.0**
3.0	1.0	0.0	0.0	2.0

- In the **second mutation**, we do the same thing but introduce *as few zeros as possible*
- A **crossover** that can be used in this case is to create children (C_1, C_2) by *averaging out* the two parents (P_1, P_2) as follows:

$$C_1 = aP_1 + bP_2$$
$$C_2 = aP_2 + bP_1$$

where $c_1 + c_2 = 1$, $c_1, c_2 > 0$