Introduction to Optimization techniques and the need for Evolutionary Computing Algorithms

Prof C Vasantha Lakshmi
DEI Dayalbagh
Content

- What is Optimization?
- Categorization of Optimization Problems
- Some Optimization Problems
What is Optimization?

- **Objective Function**
  - A function to be *minimized* or *maximized*

- **Unknowns or Variables**
  - *Affect* the value of the objective function

- **Constraints**
  - *Restrict* unknowns to take on certain values but exclude others
The optimization problem is then:
Find values of the variables that minimize or maximize the objective function while satisfying the constraints.

- **Objective Function**
  - A function to be minimized or maximized
- **Unknowns or Variables**
  - Affect the value of the objective function
- **Constraints**
  - Restrict unknowns to take on certain values but exclude others
Example: 0-1 Knapsack Problem

Which boxes should be chosen to maximize the amount of money while still keeping the overall weight under 15 kg?
Example: 0-1 Knapsack Problem

- Objective Function
  Maximize
  \[ 4x_1 + 2x_2 + 10x_3 + 2x_4 + 1x_5 \]

- Unknowns or Variables
  \( x_i \)'s, \( i = 1, \ldots, 5 \)

- Constraints
  Subject to
  \[ 12x_1 + 1x_2 + 4x_3 + 2x_4 + 1x_5 \leq 15 \]
  \( x_i \in \{0,1\}, \ i = 1, \ldots, 5 \)
Example: 
0-1 Knapsack Problem

Maximize \(4x_1 + 2x_2 + 10x_3 + 2x_4 + 1x_5\)

Subject to \(12x_1 + 1x_2 + 4x_3 + 2x_4 + 1x_5 \leq 15,\)
\[x_i \in \{0,1\}, i = 1, \ldots, 5\]
Minimum and maximum value of a function

This denotes the minimum value of the objective function, when choosing $x$ from the set of real numbers. The minimum value in this case is $1$, occurring at $x=0$. 
Similarly, the notation \( \max_{x \in \mathbb{R}} 2x \) asks for the maximum value of the objective function \( 2x \), where \( x \) may be any real number. In this case, there is no such maximum as the objective function is unbounded, so the answer is "infinity" or "undefined".
Ingredients of Optimization Problems

- Objective Function
- Unknowns or Variables
- Constraints

Are All these ingredients necessary?
Optimization Problems w/o Objective Function

- In some cases, the goal is to find a set of variables that satisfies some constraints only, e.g.,
  - circuit layouts
  - n-queen
  - Sudoku

- This type of problems is usually called a feasibility problem or constraint satisfaction problem.
Optimization Problems w/ Multiple Objective Functions

- Sometimes, we need to optimize a number of different objectives at once, e.g.,
  - In the panel design problem, it would be nice to minimize weight and maximize strength simultaneously.

- Usually, the different objectives are *not* compatible
  - The variables that optimize one objective may be far from optimal for the others.

- In practice, problems with multiple objectives are reformulated as single-objective problems by either forming a *weighted combination* of the different objectives or else replacing some of the objectives by constraints.
Variables

- Variables are essential.
  - Without variables, we cannot define the objective function and the problem constraints.

- Continuous Optimization
  - all the variables are allowed to take values from subintervals of the real line;

- Discrete Optimization
  - require some or all of the variables to have integer values.
Constraints

- Constraints are not essential.
- Unconstrained optimization
  - A large and important one for which a lot of algorithms and software are available.
- However, almost all problems really do have constraints, e.g.,
  - Any variable denoting the “number of objects” in a system can only be useful if it is less than the number of elementary particles in the known universe!
  - In practice, though, answers that make good sense in terms of the underlying physical or economic problem can often be obtained without putting constraints on the variables.
Shortest Path Problems

Straight-line distance to Bucharest

- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobrota: 242
- Eforie: 161
- Fagaras: 178
- Giurgiu: 77
- Hirsova: 151
- Iasi: 226
- Lugoj: 244
- Mehedia: 241
- Neamt: 234
- Oradea: 380
- Pitesti: 98
- Rimnicu Vilcea: 193
- Sibiu: 253
- Timisoara: 329
- Urziceni: 80
- Vaslui: 199
- Zerind: 374
Traveling Salesman Problem (TSP)
Traveling Salesman Problem (TSP)

How many feasible paths?

\[
\frac{n!}{2n} = \frac{(n-1)!}{2}
\]

\(n\) cities
Example
Example
Example
Gradient descent (illustration)
Evolution/Survival of the Fittest

In particular, any new mutation that appears in a child (e.g. longer neck, longer legs, thicker skin, longer gestation period, bigger brain, light-sensitive patch on the skin, a harmless `loose’ bone, etc etc) and which helps it in its efforts to survive long enough to have children, will become more and more widespread in future generations.

The theory of evolution is the statement that all species on Earth have arisen in this way by evolution from one or more very simple self-reproducing molecules in the primeval soup. I.e. we have evolved via the accumulation of countless advantageous (in context) mutations over countless generations, and species have diversified to occupy niches, as a result of different environments favouring different mutations.
Evolution as a Problem Solving Method

Can view evolution as a way of solving the problem: How can I survive in this environment?

The basic method of it is trial and error. I.e. evolution is in the family of methods that do something like this:

1. Come up with a new solution by randomly changing an old one. Does it work better than previous solutions? If yes, keep it and throw away the old ones. Otherwise, discard it.
2. Go to 1.

But this appears to be a recipe for problem solving algorithms which take forever, with little or no eventual success!
The Magic Ingredients

Not so – since there are two vital things (and one other sometimes useful thing) we learn from natural evolution, which, with a sprinkling of our own commonsense added, lead to generally superb problem solving methods called evolutionary algorithms:

Lesson1:
Keep a population/collection of different things on the go.

Lesson2:
Select `parents’ with a relatively weak bias towards the fittest. It’s not really plain survival of the fittest, what works is the fitter you are, the more chance you have to reproduce, and it works best if even the least fit still have some chance.

Lesson3:
It can sometimes help to use recombination of two or more `parents’ – I.e. generate new candidate solutions by combining bits and pieces from different previous solutions.
A Generic Evolutionary Algorithm

Suppose you have to find a solution to some problem or other, and suppose, given any candidate solution \( s \) you have a function \( f(s) \) which measures how good \( s \) is as a solution to your problem.

Generate an initial population \( P \) of randomly generated solutions (this is typically 100 or 500 or so). Evaluate the fitness of each. Then:

Repeat until a termination condition is reached:

1. **Selection**: Choose some of \( P \) to be parents

2. **Variation**: Apply genetic operators to the parents to produce some children, and then evaluate the fitness of the children.

3. **Population update**: Update the population \( P \) by retaining some of the children and removing some of the
Basic Varieties of Evolutionary Algorithm

1. Selection: Choose some of $P$ to be parents

There are many different ways to select – e.g. choose top 10% of the population; choose with probability proportionate to fitness; choose randomly from top 20%, etc …

2. Variation: Apply genetic operators …

There are many different ways to do this, and it depends much on the encoding (see next slide). We will learn certain standard ways.

3. Population update: Update the population $P$ by …

There are many several ways to do this, e.g. replace entire population with the new children; choose best $|P|$ from $P$ and the new ones, etc.
Some of what EA-ists (theorists and practitioners) are concerned with:

**How to select?**
Always select the best? Bad results, quickly
Select almost randomly? Great results, too slowly

**How to encode?**
Can make all the difference, and is intricately tied up with:

**How to vary?**
(mutation, recombination, etc…)
small-step mutation preferred, recombination seems to be a principled way to do large steps, but large steps are usually abysmal.

**What parameters? How to adapt with time?**
What are they good for?

Suppose we want the best possible schedule for a university lecture timetable.

- Or the best possible pipe network design for a ship’s engine room
- Or the best possible design for an antenna with given requirements
- Or a formula that fits a curve better than any others
- Or the best design for a comms network in terms of reliability for
- Or the best strategy for flying a fighter aircraft
- Or the best factory production schedule we can get,
- Or the most accurate neural network for a control problem,
- Or the best treatment plan (beam shapes and angles) for radiotherapy cancer treatment
- And so on and so on ....!
- The applications cover all of optimisation and machine learning.
Every Evolutionary Algorithm

Given a problem to solve, a way to generate candidate solutions, and a way to assign fitness values:

1. Generate and evaluate a population of candidate solutions
2. Select a few of them
3. Breed the selected ones to obtain some new candidate solutions, and evaluate them
4. Throw out some of the population to make way for some of the new children.
5. Go back to step 2 until finished.
Initial population
Select
Crossover
Another Crossover
A mutation
Another Mutation
Old population + children
New Population: Generation 2
Generation 3
Generation 4, etc …
The basic ingredients

A running EA can be seen as a sequence of *generations*. A generation is simply characterised by a *population*.

An EA goes from generation $t$ to generation $t+1$ in 3 steps:

1. **Select parents**
2. **Produce children from the parents**
3. **Merge parent and child populations**

And remove some so that popsize is maintained.
The basic ingredients

A running EA can be seen as a sequence of *generations*. A generation is simply characterised by a *population*. An EA goes from generation $t$ to generation $t+1$ in 3 steps:

1. **Select parents**

2. ETC ... as on last slide, leading to Gen $t + 2$, *and so on.*
Steady State vs Generational

There are two extremes:

Steady state: population only changes slightly in each generation. (e.g. select 1 parent, produce 1 child, add that child to pop and remove the worst)

Generational: population changes completely in each generation. (select some [could still be 1] parent(s), produce popsize children, they become the next generation.

Although, in practice, if we are using a generational EA, we usually use elitism, which means the new generation always contains the best solution from the previous generation, the remaining popsize-1 individuals being new.
Selection and Variation

A selection method is a way to choose a parent from the population, in such a way that the fitter an individual is, the more likely it is to be selected.

A variation operator or genetic operator is any method that takes in a (set of) parent(s), and produces a new individual (called child). If the input is a single parent, it is called a mutation operator. If two parents, it is called crossover or recombination.
Replacement

A replacement method is a way to decide how to produce the next generation from the merged previous generation and children.

E.g. we might simply sort them in order of fitness, and take the best *popsize* of them. What else might we do instead?
Encodings

We want to evolve schedules, networks, routes, coffee percolators, drug designs – how do we **encode** or **represent** solutions?

How you **encode** dictates what your **operators** can be, and certain constraints that the operators must meet.
The Travelling Salesperson Problem

An example (hard) problem, for illustration

The Travelling Salesperson Problem
Find the shortest tour through the cities.

The one below is length: 33

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Simplest possible EA: Hillclimbing

0. Initialise: Generate a random solution \( c \); evaluate its fitness, \( f(c) \). Call \( c \) the current solution.

1. Mutate a copy of the current solution – call the mutant \( m \)
   Evaluate fitness of \( m \), \( f(m) \).

2. If \( f(m) \) is no worse than \( f(c) \), then replace \( c \) with \( m \),
   otherwise do nothing (effectively discarding \( m \)).

3. If a termination condition has been reached, stop.
   Otherwise, go to 1.

Note. No population (well, population of 1). This is a very simple version of an EA, although it has been around for much longer.
Why “Hillclimbing”?

Suppose that solutions are lined up along the $x$ axis, and that mutation always gives you a nearby solution. Fitness is on the $y$ axis; this is a **landscape**.

1. Initial solution; 2. rejected mutant; 3. new current solution, 4. New current solution; 5. new current solution; 6. new current soln 7. Rejected mutant; 8. rejected mutant; 9. new current solution, 10. Rejected mutant, …
Example: HC on the TSP

We can encode a candidate solution to the TSP as a permutation.

Here is our initial random solution ACEDB with fitness 32.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Example: HC on the TSP

We can encode a candidate solution to the TSP as a permutation.

Here is our initial random solution ACEDB with fitness 32. We are going to mutate it – first make Mutant a copy of Current.
We randomly mutate it (swap randomly chosen adjacent nodes) from ACEDB to ACDEB which has fitness 33 -- so current stays the same. Because we reject this mutant.
We now try another mutation of Current (swap randomly chosen adjacent nodes) from ACEDB to CAEDDB. Fitness is 38, so reject that too.
Our next mutant of Current is from ACEDB to AECDB. Fitness 33, reject this too.
Our next mutant of Current is from ACEDB to ACDEB. Fitness 33, reject this too.
Our next mutant of Current is from ACEDB to ACEBD. Fitness is 32. Equal to Current, so this becomes the new Current.
HC on the TSP

ACEBD is our Current solution, with fitness 32

Current solution

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
ACEBD is our Current solution, with fitness 32. We mutate it to DCEBA (note that first and last are adjacent nodes); fitness is 28. So this becomes our new current solution.
HC on the TSP

Our new Current, DCEBA, with fitness 28

Current solution

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
HC on the TSP

Our new Current, DCEBA, with fitness 28. We mutate it, this time getting DCEAB, with fitness 33 – so we reject that and DCEBA is still our Current solution.
SGA operators: 1-point crossover

- Choose a random point on the two parents
- Split parents at this crossover point
- Create children by exchanging tails
- $P_c$ typically in range (0.6, 0.9)
SGA operators: mutation

- Alter each gene independently with a probability $p_m$
- $p_m$ is called the mutation rate
  - Typically between 1/pop_size and 1/chromosome_length

| parent | 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 |
| child  | 0 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1 |
Main idea: better individuals get higher chance
- Chances proportional to fitness
- Implementation: roulette wheel technique
  - Assign to each individual a part of the roulette wheel
  - Spin the wheel $n$ times to select $n$ individuals

<table>
<thead>
<tr>
<th>Individual</th>
<th>Fitness</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>1/6 = 17%</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>2/6 = 33%</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>3/6 = 50%</td>
</tr>
</tbody>
</table>

fitness(A) = 3
fitness(B) = 1
fitness(C) = 2
An example after Goldberg ‘89 (1)

- Simple problem: max $x^2$ over \{0,1,...,31\}
- GA approach:
  - Representation: binary code, e.g. 01101 $\leftrightarrow$ 13
  - Population size: 4
  - 1-point xover, bitwise mutation
  - Roulette wheel selection
  - Random initialisation
- We show one generational cycle done by hand
### $x^2$ Example: Selection

<table>
<thead>
<tr>
<th>String no.</th>
<th>Initial population</th>
<th>$x$ Value</th>
<th>Fitness $f(x) = x^2$</th>
<th>$Prob_i$</th>
<th>Expected count</th>
<th>Actual count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01101</td>
<td>13</td>
<td>169</td>
<td>0.14</td>
<td>0.58</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11000</td>
<td>24</td>
<td>576</td>
<td>0.49</td>
<td>1.97</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>01000</td>
<td>8</td>
<td>64</td>
<td>0.06</td>
<td>0.22</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10011</td>
<td>19</td>
<td>361</td>
<td>0.31</td>
<td>1.23</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td></td>
<td>1170</td>
<td>1.00</td>
<td>4.00</td>
<td>4</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>293</td>
<td>0.25</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td></td>
<td>576</td>
<td>0.49</td>
<td>1.97</td>
<td>2</td>
</tr>
</tbody>
</table>
### $X^2$ example: crossover

<table>
<thead>
<tr>
<th>String no.</th>
<th>Mating pool</th>
<th>Crossover point</th>
<th>Offspring after xover</th>
<th>$x$ Value</th>
<th>Fitness $f(x) = x^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 1 1 0</td>
<td>4</td>
<td>0 1 1 0 0</td>
<td>12</td>
<td>144</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 0</td>
<td>4</td>
<td>1 1 0 0 1</td>
<td>25</td>
<td>625</td>
</tr>
<tr>
<td>2</td>
<td>1 1</td>
<td>2</td>
<td>1 1 0 1 1</td>
<td>27</td>
<td>729</td>
</tr>
<tr>
<td>4</td>
<td>1 0</td>
<td>2</td>
<td>1 0 0 0 0</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>1754</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>439</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>729</strong></td>
<td></td>
</tr>
</tbody>
</table>
**X² example: mutation**

<table>
<thead>
<tr>
<th>String no.</th>
<th>Offspring after xover</th>
<th>Offspring after mutation</th>
<th>x Value</th>
<th>Fitness $f(x) = x^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 1 1 0 0</td>
<td>1 1 1 0 0</td>
<td>26</td>
<td>676</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 0 1</td>
<td>1 1 0 0 1</td>
<td>25</td>
<td>625</td>
</tr>
<tr>
<td>2</td>
<td>1 1 0 1 1</td>
<td>1 1 0 1 1</td>
<td>27</td>
<td>729</td>
</tr>
<tr>
<td>4</td>
<td>1 0 0 0 0</td>
<td>1 0 1 0 0</td>
<td>18</td>
<td>324</td>
</tr>
<tr>
<td>Sum Average Max</td>
<td></td>
<td></td>
<td></td>
<td>2354 588.5 729</td>
</tr>
</tbody>
</table>
The simple GA

- Has been subject of many (early) studies
  - still often used as benchmark for novel GAs
- Shows many shortcomings, e.g.
  - Representation is too restrictive
  - Mutation & crossovers only applicable for bit-string & integer representations
  - Selection mechanism sensitive for converging populations with close fitness values
  - Generational population model (step 5 in SGA repr. cycle) can be improved with explicit survivor selection
Alternative Crossover Operators

- Performance with 1 Point Crossover depends on the order that variables occur in the representation
  - more likely to keep together genes that are near each other
  - Can never keep together genes from opposite ends of string
  - This is known as *Positional Bias*
  - Can be exploited if we know about the structure of our problem, but this is not usually the case
n-point crossover

- Choose n random crossover points
- Split along those points
- Glue parts, alternating between parents
- Generalisation of 1 point (still some positional bias)

```
parents
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1
children
0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 0 0
```
Uniform crossover

- Assign 'heads' to one parent, 'tails' to the other
- Flip a coin for each gene of the first child
- Make an inverse copy of the gene for the second child
- Inheritance is independent of position
Decade long debate: which one is better / necessary / main-background

Answer (at least, rather wide agreement):
- it depends on the problem, but
- in general, it is good to have both
- both have another role
- mutation-only-EA is possible, xover-only-EA would not work
Crossover OR mutation? (cont’d)

- Only crossover can combine information from two parents
- Only mutation can introduce new information (alleles)
- Crossover does not change the allele frequencies of the population (thought experiment: 50% 0’s on first bit in the population, ?% after performing \( n \) crossovers)
- To hit the optimum you often need a ‘lucky’ mutation
Integer representations

- Some problems naturally have integer variables, e.g. image processing parameters
- Others take *categorical* values from a fixed set e.g. {blue, green, yellow, pink}
- N-point / uniform crossover operators work
- Extend bit-flipping mutation to make
  - “creep” i.e. more likely to move to similar value
  - Random choice (esp. categorical variables)
  - For ordinal problems, it is hard to know correct range for creep, so often use two mutation operators in tandem
Crossover operators for real valued GAs

- **Discrete:**
  - each allele value in offspring $z$ comes from one of its parents $(x, y)$ with equal probability: $z_i = x_i$ or $y_i$
  - Could use n-point or uniform

- **Intermediate**
  - exploits idea of creating children “between” parents (hence a.k.a. *arithmetic* recombination)
  - $z_i = \alpha x_i + (1 - \alpha) y_i$ where $\alpha : 0 \leq \alpha \leq 1$.
  - The parameter $\alpha$ can be:
    - constant: uniform arithmetical crossover
    - variable (e.g. depend on the age of the population)
    - picked at random every time
Permutation Representations

- Ordering/sequencing problems form a special type
- Task is (or can be solved by) arranging some objects in a certain order
  - Example: sort algorithm: important thing is which elements occur before others (order)
  - Example: Travelling Salesman Problem (TSP): important thing is which elements occur next to each other (adjacency)
- These problems are generally expressed as a permutation:
  - if there are \( n \) variables then the representation is as a list of \( n \) integers, each of which occurs exactly once
Permutation representation: TSP example

- Problem:
  - Given n cities
  - Find a complete tour with minimal length

- Encoding:
  - Label the cities 1, 2, … , n
  - One complete tour is one permutation (e.g. for n =4 [1,2,3,4], [3,4,2,1] are OK)

- Search space is BIG:
  for 30 cities there are 30! ≈ 10^{32} possible tours
Mutation operators for permutations

- Normal mutation operators lead to inadmissible solutions
  - e.g. bit-wise mutation: let gene $i$ have value $j$
  - changing to some other value $k$ would mean that $k$ occurred twice and $j$ no longer occurred
- Therefore must change at least two values
- Mutation parameter now reflects the probability that some operator is applied once to the whole string, rather than individually in each position
Insert Mutation for permutations

- Pick two allele values at random
- Move the second to follow the first, shifting the rest along to accommodate
- Note that this preserves most of the order and the adjacency information
Swap mutation for permutations

- Pick two alleles at random and swap their positions
- Preserves most of adjacency information (4 links broken), disrupts order more
Inversion mutation for permutations

- Pick two alleles at random and then invert the substring between them.
- Preserves most adjacency information (only breaks two links) but disruptive of order information
Scramble mutation for permutations

- Pick a subset of genes at random
- Randomly rearrange the alleles in those positions

(1 2 3 4 5 6 7 8 9) → (1 3 5 4 2 6 7 8 9)

(note subset does not have to be contiguous)
Crossover operators for permutations

- “Normal” crossover operators will often lead to inadmissible solutions

- Many specialised operators have been devised which focus on combining order or adjacency information from the two parents
Order 1 crossover

- Idea is to preserve relative order that elements occur
- Informal procedure:
  1. Choose an arbitrary part from the first parent
  2. Copy this part to the first child
  3. Copy the numbers that are not in the first part, to the first child:
     - starting right from cut point of the copied part,
     - using the order of the second parent
     - and wrapping around at the end
  4. Analogous for the second child, with parent roles reversed
Order 1 crossover example

- Copy randomly selected set from first parent
  
  1 2 3 4 5 6 7 8 9

  \[\rightarrow\]

  4 5 6 7

  9 3 7 8 2 6 5 1 4

- Copy rest from second parent in order 1,9,3,8,2
  
  1 2 3 4 5 6 7 8 9

  \[\rightarrow\]

  3 8 2 4 5 6 7 1 9

  9 3 7 8 2 6 5 1 4