Diploma Programme
Programme du diplôme
Programa del Diploma

Computer science
Case study: Genetic algorithms

For use in November 2021, May 2022 and November 2022

Instructions to candidates

- Case study booklet required for higher level paper 3.

8 pages

Introduction

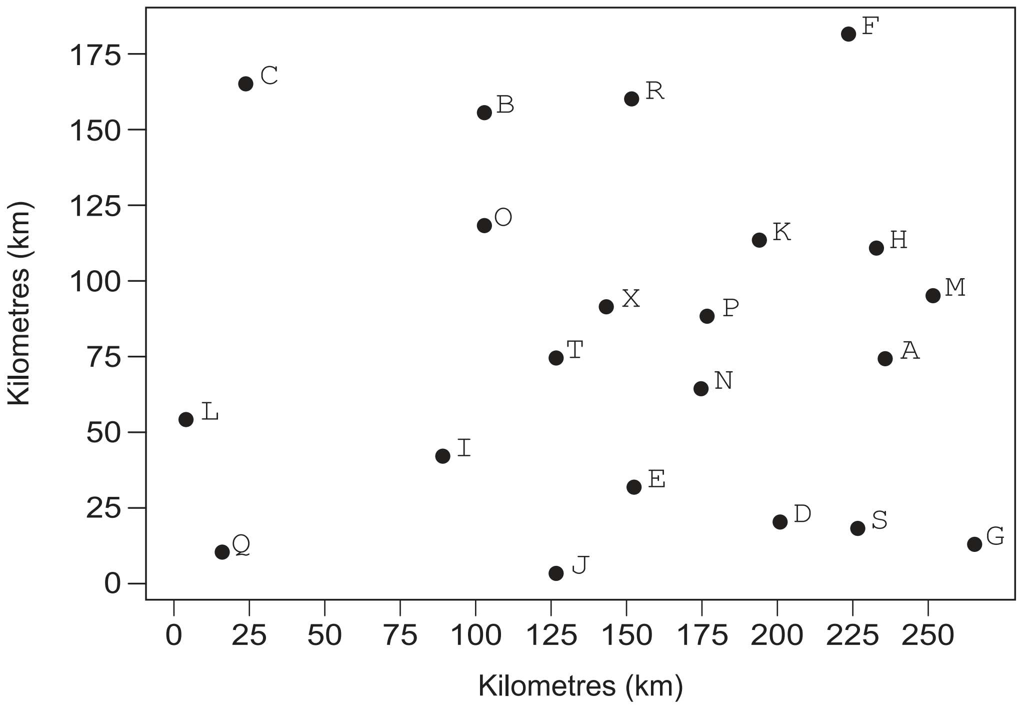
It is 19:00 on Saturday evening and Lotte is packing her things before she sets off for a motorcycle tour of Vlakland on Monday morning. She has identified 20 cities she wants to visit before returning home. Her friend, Fenna, a twelfth-grade computer science student, asks what route she plans to take. "Actually, I was hoping you could help me with that," Lotte replies. "I've found a table of the distances between the cities I want to visit," she explains (see **Figure 1**). "I was wondering if you could write a short computer program to test all the possible routes?"

Figure 1: Distances in kilometres (km) between the cities Lotte wants to visit

	X	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**
X	0	94	76	141	91	60	120	145	91	74	90	55	145	108	41	49	33	151	69	111	24
A	94	0	156	231	64	93	108	68	37	150	130	57	233	26	62	140	61	229	120	57	109
B	76	156	0	80	167	133	124	216	137	114	154	100	141	161	116	37	100	169	49	185	84
C	141	231	80	0	229	185	201	286	216	139	192	178	113	239	182	92	171	155	128	251	137
D	91	64	167	229	0	49	163	65	96	114	76	93	200	91	51	139	72	185	148	26	92
E	60	93	133	185	49	0	165	115	112	65	39	91	151	117	39	99	61	139	128	75	49
F	120	108	124	201	163	165	0	173	71	194	203	74	254	90	127	136	104	269	75	163	144
G	145	68	216	286	65	115	173	0	103	179	139	123	265	83	104	194	116	250	186	39	152
H	91	37	137	216	96	112	71	103	0	160	151	39	236	25	75	130	61	239	95	93	112
I	74	150	114	139	114	65	194	179	160	0	54	127	86	171	89	77	99	80	134	140	50
J	90	130	154	192	76	39	203	139	151	54	0	129	133	155	78	117	99	111	159	101	71
K	55	57	100	178	93	91	74	123	39	127	129	0	199	61	53	91	30	206	63	101	78
L	145	233	141	113	200	151	254	265	236	86	133	199	0	251	171	118	176	46	182	226	125
M	108	26	161	239	91	117	90	83	25	171	155	61	251	0	83	151	75	251	119	81	127
N	41	62	116	182	51	39	127	104	75	89	78	53	171	83	0	90	24	168	99	69	49
O	49	140	37	92	139	99	136	194	130	77	117	91	118	151	90	0	80	139	65	159	50
P	33	61	100	171	72	61	104	116	61	99	99	30	176	75	24	80	0	179	76	86	52
Q	151	229	169	155	185	139	269	250	239	80	111	206	46	251	168	139	179	0	202	211	128
R	69	120	49	128	148	128	75	186	95	134	159	63	182	119	99	65	76	202	0	161	90
S	111	57	185	251	26	75	163	39	93	140	101	101	226	81	69	159	86	211	161	0	115
T	24	109	84	137	92	49	144	152	112	50	71	78	125	127	49	50	52	128	90	115	0

Note: X is Lotte's house

Figure 2: Computer-generated image using the data in Figure 1 to show the location of the cities Lotte wants to visit



Fenna frowns. She recognizes Lotte's situation as an example of a *combinatorial optimization* problem known as the *travelling salesman problem.* In this problem, the number of possible solutions grows extremely rapidly with input size so that even quite small problems, such as visiting 20 different cities by the shortest possible route, become *computationally intractable*.

"Let's say your starting point is `X` and the cities you want to visit are labelled `A`, `B`, `C`, *etc*," explains Fenna. "If you only visit one city, there is only one choice, which is `X` to `A` and back again; we can write this as `XAX`. If you have two cities, then you have `XABX` or `XBAX`. If there are three cities, then you have six choices:

`XABCX   XACBX   XBACX   XBCAX   XCABX   XCBAX`"

"But `XABX` and `XBAX` are the same route, just in different directions!" observes Lotte.

"That's correct," says Fenna. "The number of permutations is always halved, because each route occurs twice, once in each of the two directions. The problem is this: every time we add a new city, the new city can be inserted at any point in all of the current possible routes. Adding the Nth city multiplies the number of existing solutions by N."

"We don't have to do it by hand though!" laughs Lotte. "We have a computer!"

"Well, yes," Fenna agrees, "but with 20 locations we have $^{20!}/_{2}$ permutations." She opens the calculator app on her phone. "That's 1 216 451 004 088 320 000"—Fenna busily taps her phone and then finally looks up—"it would still take more than 30 000 years to test them all."

The travelling salesman problem

This involves starting at one city and visiting every other city before returning to the starting point. Each city is connected to every other by a direct route, all cities must be visited once, and no city can be visited more than once. Any sequence of cities that obeys these rules is a valid *tour*. The aim of the problem in its strictest version is to find the shortest possible tour.

Turn over

There are a variety of approaches to the travelling salesman problem, many of which require considerable mathematical training, but it is currently not known whether an algorithm exists that will find the optimal solution in a reasonable amount of time. There are, however, *heuristics* that have had some success in finding good solutions quickly enough to make them a practical approach, and it is one of these that is the subject of this case study.

Genetic algorithms

Genetic algorithms mimic the process of natural selection in an attempt to evolve solutions to otherwise computationally intractable problems.

Implementation details vary considerably, but a standard genetic algorithm includes the following steps:

```
Initialize
While true
    Evaluate
    If (termination condition) break
    Select
    Crossover
    Mutate
Output best result
```

The rest of this section examines some implementation options when applying genetic algorithms to the travelling salesman problem.

Initialization

This involves generating a random *population* of individual tours, each of which is a possible solution to the problem. In the travelling salesman problem, with a starting point of X, one possible tour is:

F	J	G	I	L	C	M	E	S	Q	P	H	T	B	K	N	R	D	O	A

Evaluation

The algorithm determines if the *termination condition* has been met and, if so, outputs the best solution found so far and terminates. If not, it determines the *fitness* of each individual tour. In Lotte's problem, this involves calculating the total distance that Lotte would travel using that tour.

A *fitness function* is used to assign a fitness value to each tour. Individual tours are sorted according to their fitness. The highest fitness value is assigned to the shortest tour.

Selection

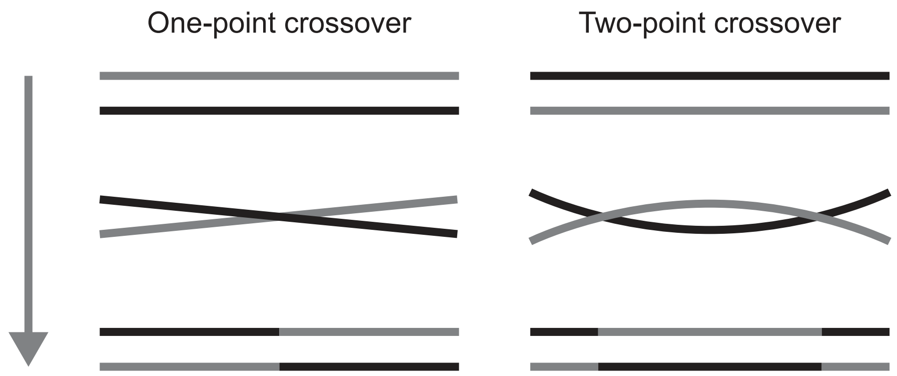
A sample of tours is taken from the population and placed in the *mating pool*. This can be done in a number of different ways but, in general, fitter tours have a higher probability of being selected to enter the mating pool. Four common *selection strategies* are:
- *roulette wheel selection*
- *stochastic universal sampling*
- *tournament selection*
- *truncation selection*.

A further design consideration in selection is to decide whether to ensure that the best solution(s) are carried to the next generation with certainty. This is known as *elitism*.

Crossover

In biology, *crossover* is the term given to the mechanism by which the chromosomes of a new individual tour are created from a combination of its parents' chromosomes.

Figure 3: Types of crossover



In the travelling salesman problem, simple one- or two-point crossover (**Figure 3**) presents a problem because cities can be repeated or omitted in the *offspring*, leading to an invalid tour that does not visit each city as required.

Consider the following crossover (**Figure 4**) between two valid tours of a ten-city travelling salesman problem, in which the parents, `P1` and `P2`, combine using one-point crossover to make a new individual tour, `F1`.

Figure 4: An example of crossover

P1	**B**	**F**	**C**	**A**	**D**	**H**	**G**	**I**	**E**	**J**
P2	G	J	C	D	I	A	E	B	F	H
F1	**B**	**F**	**C**	**A**	**D**	A	E	B	F	H

The resulting offspring, `F1`, is not a valid tour because it repeats cities `A`, `B` and `F` and omits cities `G`, `I` and `J`. The same problem occurs with a simple two-point crossover. A number of different crossover mechanisms exist, each trying to preserve the characteristics of the parents as much as possible.

Having decided which individual tours will make up the mating pool, it is necessary to decide how they should combine to produce offspring. Three methods are presented:

- Partially mapped crossover (PMX)
- Order crossover (OX)
- Cycle crossover (CX)

Turn over

Partially mapped crossover (PMX)

Choose a random sub-sequence from P1 and copy it to F1:

P1: J B F C A D H G I E
P2: F A G D H C E B J I

F1: * * **F C A D H** * * *

Set up elementwise mappings between P1 and P2 for cities in P1 that are not already in F1. If the corresponding city C from P2 is already in F1 then resume mapping from the location of C in P1, repeating until a city not in F1 is found:

J ↔ G   B ↔ E   G ↔ B   I ↔ J   E ↔ I

Add the remaining cities from P1 to F1 and change them according to the mappings:

P1: J B F C A D H G I E
P2: F A G D H C E B J I

F1: * * **F C A D H** * * *
    ↓ ↓   ↓ ↓ ↓
F1: J B **F C A D H** G I E
    ↓ ↓   ↓ ↓ ↓
F1: G E **F C A D H** B J I

Order crossover (OX)

Choose a sub-sequence from one parent and preserve the relative order of the remaining cities from the other.

Take a random sub-sequence S from P1 and copy it to F1. Starting with the empty element just after S in F1, copy all cities that are not already in F1 from P2 in the order they appear in P2.

P1: J **B F C A D** H G I E
P2: F A G D H C E B J I

F1: * **B F C A D** * * * *
    H **B F C A D** E J I G

Cycle crossover (CX)

In F1, every city maintains the position it had in at least one of its parents.

P1: J B F C A D H G I E
P2: F A G D H C E B J I

Choose the first city from P1 and copy it to F1. Check the corresponding city in P2 (here, city F) and copy it to F1, in the same position it occurs in P1. Repeat.

F1: **J** B F * A * H G I E

When you encounter a city that is already in F1 the cycle is complete. Now fill in the remaining cities from P2.

F1: J B F **D** A **C** H G I E

Mutation

In biology, *mutation* refers to accidental errors in the copying of genetic information from one generation to the next. In a genetic algorithm, mutations are deliberately introduced in each new offspring according to the *mutation rate*.

Discussion

The advantages of genetic algorithms are thought to be their ability to simultaneously sample vast *fitness landscapes* while escaping the *local extrema* that might trap more traditional *hill-climbing* approaches. Successful implementations of genetic algorithms strike a natural balance between *exploration* and *exploitation*, and techniques such as *simulated annealing* can fine-tune that balance as the algorithm progresses towards *convergence*. More recent research has focused on specifically rewarding *novelty* as a means of encouraging algorithms to explore remote regions of the *problem space*. Other design choices, such as initial parameters, *selection strategy* and *crossover operator*, all influence the performance of the algorithm, although it is generally not possible to predict their effects, so a trial-and-error approach is usually adopted.

Challenges faced

There are a number of challenges associated with genetic algorithms. These include:

- understanding the role of convergence in genetic algorithms and the factors affecting convergence
- evaluating the use and implementation of roulette wheel selection, tournament selection and truncation selection strategies used within genetic algorithms
- discussing the different solutions to address the failure of simple crossover strategies for the travelling salesman problem. In particular:
  - why they are necessary
  - how they are applied
  - how they preserve the parental traits
  - what other possible methods are available
- understanding the advantages and disadvantages of genetic algorithms with respect to other approaches to the travelling salesman problem and combinatorial optimization problems in general.

Candidates are not required to know the implementation details of other approaches.

Additional terminology

Brute force approach
Combinatorial optimization
Computational intractability
Convergence
Crossover / crossover operator
Elitism
Exploration vs exploitation
Fitness / fitness function / fitness landscape
Heuristic
Hill climbing
Initialization parameters
Local extrema
Mating pool
Mutation / mutation rate
Novelty search
Offspring
Optimization
Population
Premature convergence
Problem space
Ranking
Roulette wheel selection
Selection strategy
Simulated annealing
Stochastic universal sampling
Termination condition
Tour
Tournament selection
Truncation selection

References: