COMP2002 - Artificial Intelligence

Week 5 - Evolutionary Computation Exercises - Model Answer

Introduction

This set of model answers is intended to show you possible ways of solving the evolutionary computation exercises in week 5 – there are others. Please attempt the exercises before looking at the answers.

Activities

Your task is to go through the following tasks. Please note, that you are expected to complete some work on this outside of the timetabled sessions.

Exercise 1

The full code for this exercise was provided within the workshop sheet, but I have replicated it here so that you can see hot it all fits together. Please see the exercise sheet for a description of what each part does.

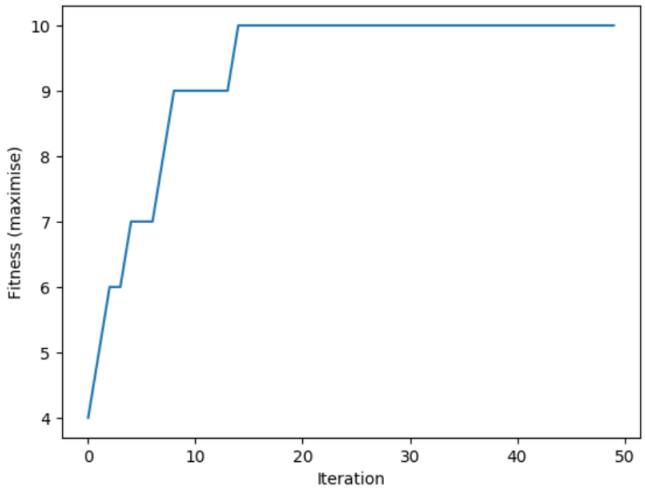
```
import numpy as np
import matplotlib.pyplot as plt

def onemax(x):
  return x.sum()

class BitFlipMutation:

  def mutate(self, x):
        idx = np.random.randint(x.shape[0])
        xp = x.copy()
        xp[idx]=abs(1-x[idx])
        return xp

def evolve(x, y, func, mutation, compare, A):
       xp = mutation.mutate(x)
       yp = func(xp)

       if not compare (y, yp):
          x=xp
          y=yp

        A.append(y)
        return x, y, A

def optimise(D, func, mutation, ngens, compare):
       x = np.random.randint(0, 2, D)
       y = func(x)

       archive = []

       for gen in range(ngens):
          x, y, archive = evolve(x, y, func, mutation, compare, archive)

       return x, y, archive

def greaterThanOrEqual(u, v):
```

```
  return u>=v
```

Having written the machinery of the optimiser you can run it as follows:

```
x, y, ylist=optimise(10, onemax, BitFlipMutation(), 50, greaterThanOrEqual)
print(x, y)
plt.plot(ylist)
plt.show()
```

Your plot should look something like this (remember you won't get the same answer every time as there is randomness in the process):



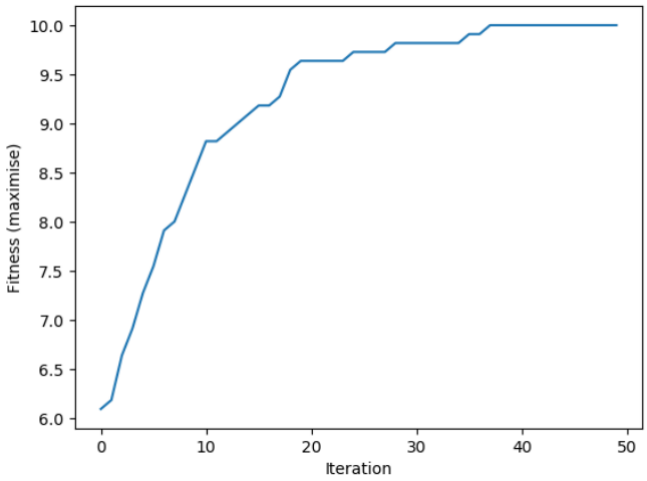
Exercise 2

Your task here is to embed the optimiser in a loop so that it runs multiple times. You need to record the fitnesses from each, and keep them so that you can compute the mean fitness at each iteration. Produce a plot of that mean fitness, and you will see the average performance of the optimiser over time – this is useful, for example, for working out how long you need to run the algorithm for to solve the problem. Once the average fitness levels out you can usually stop.

Here is the code to do this:

```
Ys = []
for repeat in range(11):
  x, y, ylist = optimise(10, onemax, BitFlipMutation(), 50, greaterThanOrEqual)
  Ys.append(ylist)
Ys = np.array(Ys)

plt.figure()
plt.plot(Ys.mean(axis=0))
plt.xlabel("Iteration")
plt.ylabel("Fitness (maximise)")
plt.savefig("onemax_repeat.png", bbox_inches="tight")
```

The plot should look something like this:



Exercise 3

The next exercise introduces some different mutation operators. These are constructed as classes, as with the one you were provided with. Recall from the exercise sheet, one of the operators randomly generates whole new solutions, and the other picks two bits and swaps every bit between them. Here are the two operators:

```
class BlockFlipMutation:

  def mutate(self, x):
    idx, jdx = sorted(np.random.randint(0, x.shape[0], 2))
    xp = x.copy()
    xp[idx:jdx] = abs(1-x[idx:jdx])
    return xp

class RandomMutation:

  def mutate(self, x):
    return np.random.randint(0, 2, x.shape[0])
```

The next part of the exercise requires you to show the average fitness for each mutation operator. Repeating the procedure from Exercise 2, I've produced a helper function (we'll be doing the same thing in Exercise 4, so it helps to avoid redundant code). The helper does the following:

```
def experiment(func, title):
  mutstrs = ["Bitflip", "Blockflip", "Random"]
  muts = [BitFlipMutation(), BlockFlipMutation(), RandomMutation()]

  plt.figure()

  for mut, mutstr in zip(muts, mutstrs):
    Ys = []
    for repeat in range(11):
      x, y, ylist = optimise(50, onemax, mut, 200, greaterThanOrEqual)
      Ys.append(ylist)
    Ys = np.array(Ys)

    plt.plot(Ys.mean(axis=0), label=mutstr)

  plt.title(title)
  plt.xlabel("Iteration")
  plt.ylabel("Fitness (maximise)")
  plt.legend(loc=4)
```

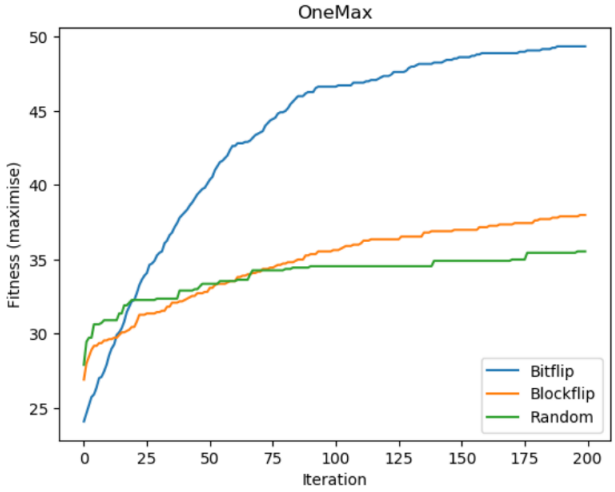
The zip function takes two lists and combines them, so given the two lists

```
L1 = ["A", "B", "C"]
L2 = [1, 2, 3]
```

the result of zip(L1, L2) is a list of tuples as follows:

```
[("A", 1), ("B", 2), ("C", 3)]
```

The resulting plot looks like this:

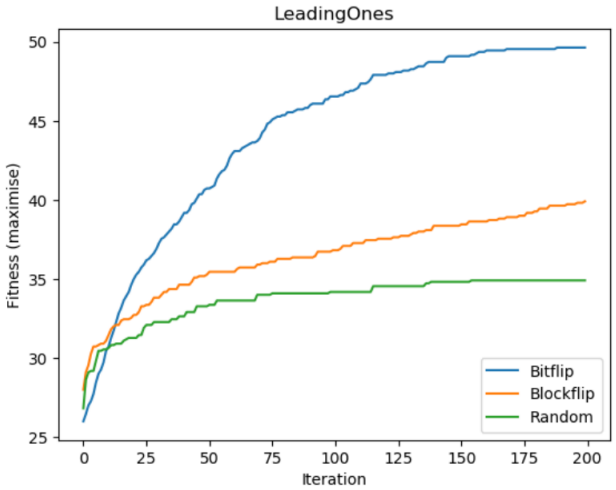


Exercise 4

The final exercise requires you to replace OneMax with the LeadingOnes problem. The code for the fitness function is as follows:

```
def leadingOnes(x):
  count = 0
  for i in range(x.shape[0]):
    if x[i] == 0:
      return count
    count += 1

  return count
```

Once you've implemented the fitness function you can use the **experiment** function from Exercise 3 to produce a similar plot for this problem. The plot should look like this:


LeadingOnes
Fitness (maximise)
Iteration
Bitflip
Blockflip
Random
50
45
40
35
30
25
0
25
50
75
100
125
150
175
200