COMP2002 - Artificial Intelligence

Week 5 - Optimisation

Introduction

The aim of this sheet of exercises is to get you started with implementing machine learning software. You should complete the exercises ahead of the week 6 seminar session. The model answers will be published shortly after, giving you enough time to re-attempt the exercises after the demonstration in the seminar.

Activities

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

Exercise 1 - OneMax

In this week's lecture you saw a range of optimisation problems. In this exercise your task is to implement the **OneMax** problem – OneMax is a benchmark problem, and is used to test that an optimiser can optimise. In the case of a benchmark problem we know what the answer is, so can check that the optimiser arrives at the optimal point. If we developed against a realworld problem we wouldn't know how well the optimiser was working.

The goal of OneMax is to find the bit string in which all bits are "1"s. The optimiser doesn't know this, and so its task is to generate new potential solutions and find the one that best solves the problem according to the fitness function. The first thing to do is implement the fitness function, which you can do as follows:

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

The argument **x** is a Numpy array of 1s and 0s. This fitness function returns the sum of the bits – so in the case when all bits are set to 1, the optimal solution, the sum of the bits will be as high as it can be.

Having defined a fitness function, you'll need a mutation operator. For this we'll be using **bit flip** operator, which functions by selecting a random bit and flipping it. If it starts off as a 1 it becomes 0, and if it's a 0 to start with it becomes 1. We're going to be using different operators in this set of exercises, so to follow good software engineering practice we'll structure them as a class so that different operators can be initialized and slotted into the optimiser before it runs.

Here is the BitFlipMutation class:

```
class BitFlipMutation:

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

Following standard object-oriented principles, our mutation objects will always have a **mutate** method that takes a solution and returns the mutated solution.

Now that we have a fitness function and mutation operator we can construct the function that will perform a single iteration of the optimiser. The function follows these steps:

1. Mutate the current solution to create a new child.
2. Evaluate the child solution under the fitness function.
3. Compare the parent and child solution – if the selection criterion is met then keep the child. Otherwise revert to the parent for the next iteration.
4. Log the retained solution's fitness in an archive.

The code for the **evolve** function is as follows:

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

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

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

We're using a function called **compare** to do our solution selection. In this case it's a simple check to see if the child is greater than or equal to the parent. I've used a function here so that when we do multi-objective optimisation next week you can use the same code and add in a different comparison function. You could also swap in a less than or equal equivalent without changing the optimiser if you wanted to do a minimization problem.

Here is the **greaterThanEqualTo** function:

```
def greaterThanOrEqual(u, v):
  return u>= v
```

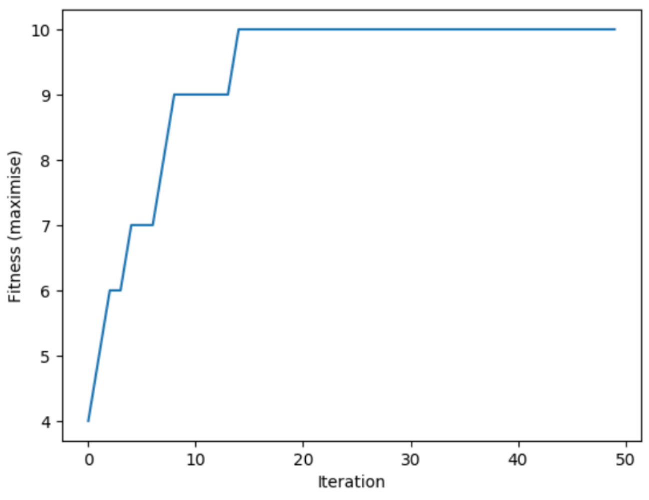
The last step in constructing your optimiser is to build the **optimise** function, which initiates the optimisation process and returns the result. It looks like this:

```
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
```

Having constructed the optimiser and its associated functions you can run it as follows (for a problem with 10 bits):

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

The plot it produces should look something like this:



Exercise 2

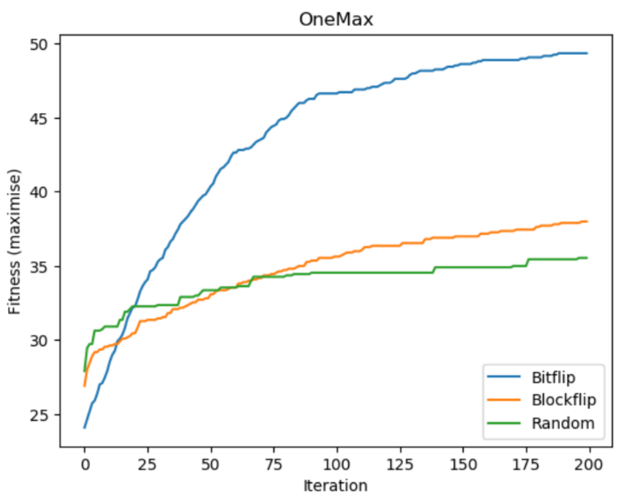
Having run the optimiser once, you should run it multiple times (at least ten) to see what the average fitness is. Produce a plot like the one from the end of Exercise 1, but show the average fitness at each iteration.

Exercise 3

Implement another two mutation operators:

- One should randomly create a new solution from scratch.
- The other should pick two bits, and flip each bit between the two.

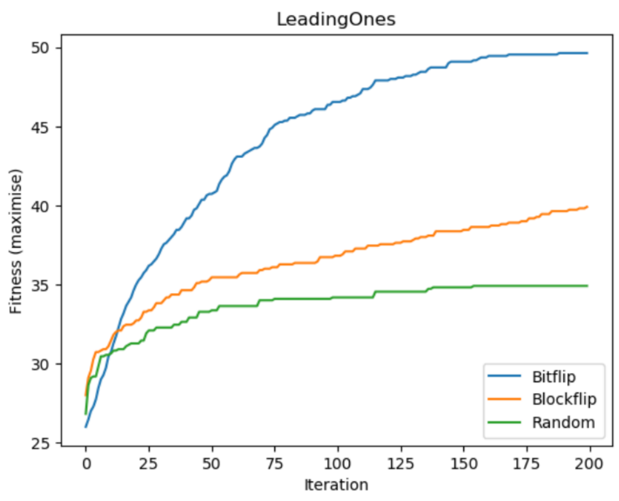
Run your optimiser for at least ten repeats, for each of the three mutation operators. You should make the problem harder than before – do this by optimising a 50-bit version of the problem instead of 10. Plot the average fitness for each mutation operator so that you can see the different performance they cause. Your plot should look something like this:



Exercise 4

Another similar benchmark problem is the **LeadingOnes** problem. This optimises the number of bits that are set to 1 before a 0 is encountered. The optimal solution is the same as for the OneMax problem, but again the optimiser doesn't know that. It simply follows its optimisation heuristics to try and learn it.

Implement a fitness function like the **onemax** function you were provided with and repeat Exercise 3 with it for the LeadingOnes problem. You should get something like this:


LeadingOnes
Fitness (maximise)
Iteration
Bitflip
Blockflip
Random