CT5141 Lab Week 6

James McDermott

2D Cars

Operators on car genotypes

A 2D car is defined by the following 15 parameters (notice types and ranges):

- Shape (8 floats in [0, 1], 1 per chassis vertex)
- Wheel size (2 floats in [0, 1], 1 per wheel)
- Wheel position (2 ints in [0, 7], 1 per wheel specifying which chassis vertex it attaches to)
- Wheel density (2 floats in [0, 1], 1 per wheel)
- Chassis density (1 float in [0, 1])

Here are two example genotypes:

- `[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]`
- `-----------------------------------------------`
- `[.3 .4 .6 .1 .3 .4 .6 .1 .6 .7  3  6 .3 .8 .4]`
- `[.2 .5 .7 .1 .2 .2 .2 .5 .1 .1  2  6 .2 .2 .3]`

1. How could we define the following operators for 2D cars? It's ok to define them in Python, in pseudo-code, or in unambiguous English.

- Initialisation
- Mutation
- Crossover.

Remember that the operators have to produce *valid* genotypes, and the output of initialisation has to be *uniform* in the space, and the output of mutation should be *near* the input, and the output of crossover should be *intermediate* to the two inputs.

2. Some of the genes *interact*. Which ones? Are they close together? What is the effect of this?

Genetic Algorithm

Implement a GA on bitstrings!

1. Implement a function `init()` which returns a random genome (a `list` of bits of length 30).
2. Implement a function `mut(x)` which makes a copy of the input genome `x`, mutates it, and returns it.
3. Implement a function `xover(x, y)` which returns a genome created by crossover of the input bitstrings `x` and `y`.
4. Implement `tournament_select(pop, tsize)` which accepts the population and the tournament size, and returns one genome.

5. Define the simple test function `f = sum`, where the objective is to maximise the number of 1s in the genome.
6. Put it all together in a function `GA(f, init, mut, xover, select, popsize, ngens, tsize)` which creates a population of size 100, then runs 50 generations. It should print out the best fitness after every generation. The fitness function, the four operators, and the three hyperparameters are passed-in.

A solution to all of the above is provided in `ga.py` in Bb. You also need the knapsack files for that problem.

Hint / spoiler: one main decision you have to make is the data structures you'll use for the population and the fitness values. In my implementation, I represent the `population` as a list of tuples. Each tuple is `(fitness, genome)`, where `fitness` is a `float` and `genome` is a `list` of 0s and 1s. But there are other possibilities, e.g. you could have one list of genomes and one list of fitness values.