Project 2
Optimisation Algorithms (Hill-climbing and Genetic Algorithm)

General Instructions

- From Hash Code 2017, Online Qualification Round: ”Have you ever wondered what happens behind the scenes when you watch a YouTube video? As more and more people watch online videos (and as the size of these videos increases), it is critical that video-serving infrastructure is optimized to handle requests reliably and quickly. This typically involves putting in place cache servers, which store copies of popular videos. When a user request for a particular video arrives, it can be handled by a cache server close to the user, rather than by a remote data center thousands of kilometers away. Given a description of cache servers, network endpoints and videos, along with predicted requests for individual videos, decide which videos to put in which cache server in order to minimize the average waiting time for all requests.”
- This project is not ”focusing” on the Google Hash Code contest as such – it is only a (good) excuse to make you design and implement optimisations algorithms.
- You are encouraged to collaborate with your peers on this project, but all written work must be your own. In particular we expect you to be able to explain every aspect of your solution if asked.
- We ask you to hand in an archive (zip or tar.gz) of your solution: code/scripts, README.txt file describing how to run your programs, a 5-10 page pdf report of your work (no need to include code in it).
- The report should include the following sections:

  1. a short introduction
  2. a requirement section that answers the question *what* is the system supposed to do
  3. an architecture/design section that answers the question *how* your solution has been designed to address the requirements described in the previous section
  4. a series of sections that describe the different challenges you faced and your solutions. For instance, take one of the script, describe the difficulty you faced and your solution. These sections can be short – the objective here is to show how you crafted the solutions with the tools you have learnt so far.
  5. a short conclusion

- The project is worth **15% of the total grade** for this module. The breakdown of marks for the project will be as follows:
  - reading the file and creating the solutions (2D array) and fitness function: 30%
  - hill-climbing: 20%
  - genetic algorithm: 30%
  - bonus (different ”greedy” or ”neighbouring” algorithm, evaluation of some parameters of the genetic algorithm, etc.): +15%
  - report: 20%
- **Due date: 18/04/2017**

Section 1. *Reading a Problem File*

You will find a Python file ”read input.py” that reads the files with the different instances of the problem (you’ll find them in the repository ”input”). Try to read the different input files and check ”empirically” that each of the fields of the data structure populated by the function ”read google” is correct.

Section 2. *Representation of a Solution*

The first step of any optimisation algorithm consists in representing a solution, i.e., a viable (or not) composition of the different simple elements of the problem. In our case, a solution is an assignment of the different files to the cache servers. For instance, a solution would be (see slide 5 of ”COMP20230 Project2 Presentation.pdf”) ”file 2 to cache 0; files 1 and 3 to cache 1; files 0 and 1 to cache 2”. We can represent that using a 2 dimensional array, showing whether a file (one dimension) is situated in any of the cache servers (the other dimension). Table 1 shows an example with 4 files and 3 cache servers. You can see that file 1 is only assigned to cache server 1 and file 2 is located in two cache servers (2 and 3), while file 3 is in none of the cache servers and file 4 is in all of them.

	File 1	File 2	File 3	File 4
Cache 1	1	0	0	1
Cache 2	0	1	0	1
Cache 3	0	1	0	1

Table 1: Example of a Solution (2 dimensional array)

Another way of representing that (without any conceptual difference is the following: ((1, 0, 0, 1), (0, 1, 0, 1), (0, 1, 0, 1))

The problem has only one constraint: each file has a size (MB) and each cache server has a maximum capacity. You have to make sure that the following formula is always satisfied:

$$\forall cache\ c_i, \forall file f_j f_j \in c_i, \sum f_j < capacity(c_i)$$

This means that for every solution as given above you need to check whether the condition holds or not – i.e., you just need, for each cache, to sum up the sizes of the files that are assigned to the cache and check whether it is less than the capacity of the cache.

The fields ”number of videos” and ”number of caches” of the data structure populated by the function ”read google” should give you all the information you need to create the 2D array (matrix) required to represent a solution. Start with a solution with only 0s (i.e., no file is in any of the cache servers).

For more detail, see the deck of slides ”COMP20230 Project2 Presentation.pdf”.

Section 3. *Fitness Function*

Now we want to evaluate the solutions. For that, create a function that takes a solution (2D array) and computes the fitness function:

- checks, for each cache server, that the sum of file sizes is not overflowing the cache server. If this is the case (this solution is not possible) then return -1
- now we know that the solution is feasible – we need to find its fitness/cost/utility value.
  - For each element in the field "video ed request" (the structure of each element is "(file ID, endpoint ID): number of requests") compute the cost of downloading it from the data centre (check the field "ep to dc latency"'s element corresponding to the endpoint).
  - Find the cost of downloading the same file from a cache server connected to the endpoint (you have to check every cache server connected to this endpoint).
  - multiply the difference by the number of requests (remember: "(file ID, endpoint ID): number of requests"). This gives the gain obtained when downloading from an available cache server for 1 request
  - sum all gains and divide by the number of individual requests (not the field "number of requests"). Look at the example given at the end of the file "read input.py"
  - This should give you the fitness (score) of your solution.

Check that your function works well on some simple example (for instance the one given by Google in the problem document).

Section 4. *Hill Climbing*

Hill-climbing algorithms work in two steps:

- generate all the neighbouring solutions, by making simple "moves"
- keep the best (fittest) solution.

And iterate.

Check the deck of slides "COMP20230 Project2 Presentation.pdf" for more details.

Implement the hill-climbing algorithm and check that it generates better and better solutions at every iteration. Note that it may take a long time to end up with a (local) maximum solution. Is it the best possible solution? How do you know that better solutions exist?

Section 5. *Genetic Algorithm*

Genetic algorithms need an initial population (i.e. a set of starting solutions). For the moment we will consider a population size of 50 individuals. To generate your initial population, you have two options:

- use the same principle(s) you used with the hill-climbing algorithm to generate 50 different solutions (keep more that one, the best, solution at each round)
- This technique could be slow or generate very similar populations, so another options would be to use a greedy/random algorithms: generate random solutions giving ran- dom values (0 or 1) for every element in the 2D array. For instance solution[i][j] is

given (randomly) the value 1, solution[i][j+1] is given (randomly) the value 0, and so on. The problem is that you generate a lot of infeasible solutions (your fitness function should detect these) but it should work. If all the randomly generated solutions fail (they are not acceptable solutions) then try to find a couple of heuristics to "limit" the randomness of those solutions (for instance add a 1 only if this does not create an infeasible solution).

Then the genetic algorithm works as follows:

- For every couple of individuals in the population there is a (small) probability of a crossover. Think of a crossover as a partitioning of a solution in 2 halves (for instance, the first 50 endpoints and the next 40 endpoints if there are 90 endpoints in your problem) and the swapping of the 2nd half with another individual. This is a dramatic change in the individuals. The probability of a crossover cannot be too small: try with 10%. Crossover generates lot of infeasible individuals. Eliminate them. Try different values for the probability of crossover and evaluate the best value for your problem. The crossover operation generates 2 children from 2 parents. Keep all of them for the moment.
- For every individual there is a probability for each value (e.g., value of solution[i][j]) to be mutated - i.e., a 0 becomes 1 and a 1 becomes 0. These probability have to be very small. Try with $\frac{1}{\textit{number of elements in the solution}}$
-
- and increase or decrease depending on the effect on the algorithm. This mutation step generates slightly modified individuals.

At the end of the mutation and crossover steps, you end up with a bigger population: the parents (50 individuals) and the children (generated by crossover). Now you have to select the 50 fittest individuals that are going to stay in your population. Remove the other ones. And iterate – every iteration is also called a *generation*.

Again, check the deck of slides "COMP20230 Project2 Presentation.pdf" for more details.

There are many possible improvements to this basic description of genetic algorithms:

- you can vary crossover and mutation probabilities
- you can change the population size
- you can change the selection process: keeping some non fit individuals to allow for some variety in your population
- etc.

Section 6. *Conclusion*

This looks like a big project, but if you follow every step you should not have any major problems. the first lab after the break will be dedicated to the project - you should get a lot of help from the demonstrators. We will also answer your questions during the first tutorial after the break etc.

Good luck!