**Combinatorial Optimization project: Google Hashcode - Libraries**

**Artificial Intelligence, semester 3**

**Kajetan Kubik** id: **145451**

**Karol Roszak** id: **145452**

**INSTRUCTION:**

To run the program simply run the "main.py" with python 3.8.3 (used plugins: threading, math, time, sys (exit function to check if specified by user parameters are correct), random. You can do it either from some ide (like spyder), or directly from console. We decided to implement an easy text interface so you can specify what exactly you want to do.

When you run the program it will ask you what method you want to use, type:

- '1' for genetic algorithm.
- '2' for greedy approach.

Then it will ask you for the result format you prefer:

- 'p' for printing out the results in a Google HashCode format to stdout.
- 's' for writing (saving) the results in a Google HashCode format to file results.txt (this type is particularly useful as sometimes stdout is not able to show the whole result at once)
- 'os' for returning only the score of all supplied books.

Next it will ask for specifying what input file to use, type 'a', 'b', 'c', 'd', 'e', or 'f' for file you want to test given algorithm against, if you want to test it against new file first of all place your file in "data" folder, then write its full name ('example.txt') during this stage, if you want to exit type 'exit'. Program will be looping through this stage so you can test all files against a given algorithm if you want.

When type of presenting results will be specified as 'p' or 'os', results will be shown between:

"##########SOLUTION FOR {given_data} BEGINNING##########"

{Time used to find the solution}

...{solution}...

"##########SOLUTION FOR {given data} END##########"

If the chosen format is 's', it will be saved to the file and only necessary time will be printed out.

There are also timers implemented, time for algorithm to run is 5 minutes for any file (including reading file, and printing/saving solution):

- **For Genetic Algorithm:**

  There is an upper bound limit of epochoes equal to 400, which for most problem instances is a way too much. Bounds can be changed in code, in function calculate(). There is also a timer, which after 5 minutes ends at the current epoch and returns the best solution.

- **For Greedy:**

  If the specified time pass it will return the solution found so far and finish its execution, (greedy is very fast so it will rather not happen, the worst time for it was 1 minute 30 seconds)

**Theoretical part, covering the method with a short explanation why it was chosen,**

**what are the advantages:**

- **Genetic Algorithm:**

  It was chosen mostly because at first it seemed that it might be effective enough for the work involved. With a bit of randomness and ability to find the best solution we thought it would be best fitted for this task. During the creation of the program it turned out that we cannot make this program to work fast enough to produce very good results. Second one is,

- **Greedy:**

  It was chosen as we thought it would be a good move to implement some "simple-idea" algorithm to test against genetic one and compare. His biggest advantage is simplicity and

determinism (in opposition to genetic one), so if efficient heuristic will be found then results will always be good. We anticipated the algorithm itself would not be hard to code and certainly it wasn't (only at the beginning), but its first versions did not even manage to finish example c or d on time, and results for all of them together was terrible (less than 8 million). Later on when we were changing its performance and heuristic it suddenly turned out it can do better than genetics, so we decided to spend more time developing it for the best possible score. Greedy works in a way that each step it takes the best (according to its heuristics) option, that of course does not imply that the overall solution will be the best possible, still it is only heuristic, so approximation. For more explanation of how it works see in Implementation part -> Greedy.

**Implementation part – a description how the algorithm was implemented:**

- **Genetic Algorithm(GA):**

In the first approach, while we had not yet understood the nature of the problem correctly, we decided to first define some order of libraries using a greedy algorithm. Then, using the genetic algorithm for each library, we change the order of the books(our population) in which they will be scanned (library order was needed to fitness function). At the end, again by using greedy algorithm we try to find a new order of libraries. This approach has 2 major issues: a) second changing of order was almost never effective, b) upper-bound limit of score was near to first approach of greedy algorithm(although once for file b_read_on.txt somehow genetic algorithm have much better score).

In the second approach, after realising that order of libraries is critical, we decided to separately change order of libraries and book order (our two sub-populations). Next, we combine them in one, big main-population of size equal to product of sizes of two sub-population. We calculate fitness values for each combination, sort this population by this value and create a new sub-population for the next iteration. Thanks to messing with the library's order, the starting value we got was larger, but due to the complexity of this solution, only a few iterations of this algorithm there was, so there was no significant score upgrade.

The last, but not the worst, even the best approach was using Genetic Algorithm to establish some order of libraries, and add book greedily, it mean, for each librier taking n best, not already taken books, where n is number per day that ibrary can scan times amount of days that libriery going to be scanned. This solution has good enough results, but in problem d), where there were 31 000 libraries, results do not significantly improve over time.

Core of the algorithm is a rather typical GA approach. Main parameters of the algorithm are: population size, mutation probability, number of new children per epoch and tournament

size. Firstly, we create a random initial population, calculate their fitness, and sort. Then, each “epoch” look like this:

- Repeat for number of new child/2:
    - Using tournament selection with parameter k - tournament size(tournament relies on choosing k candidates at random, and then choosing the best one from them ) choose 2 parents among the starting population.
    - Perform operation of crossover on chosen parents, which create two new childs, which gonna be added to population
- For each member of the new population perform mutation operation with mutation probability rate.
- Calculate new fitnesses and sort new population.
- Cut the new population to parameter size of population.

Crossover population looks like this:

For both parents find common elements. Then, the first child is going to have these elements in the same place as the first parent, and the second child is going to have these elements in the same place as the second parent. Next, fill in empty spaces in the first child with non-common elements from the second parent, in order they are and do the same for the second child and first parent. If some spaces in either child are empty, put over there random elements.

Normal mutation operators add some random element at the beginning of the individual, and sometimes shuffle all. We also added Special mutation for large examples, in which we are going to instead of adding one element, we will add a bunch of.

- **Greedy:**

First approach was implemented in a way that while iterating through all available days it was either signing a library and supplying books per day from each library that has already been signed, or just deciding which library to sign and supplying books per day, already supplied books were deleted from each not signed library. It was also using lists for books so that they might be sorted (but deleting a given book consumed too much time, O(N) to determine where it is and delete). Used heuristic was counting average value of 1 book in library and multiplying it by (books_per_day * time_left)

This approach was completely inefficient as written just for comparison and in rush, but after discussion we decided to develop it and change the way of storing books (to sets) and thus to definitely change heuristic (instead of approximation, we could have counted the actual score that given library is able to provide). We also decided to optimize the way of supplying books (the old way was time consuming and not really necessary to use - from supplying some books day after day to supplying all new books from a given library at once in available time.

Current, and best approach we developed works like this:

We have number of days_left to supply books, we then determine the order of libraries to sign by counting the actual value they can supply and dividing by sign_up_time (taking into consideration books_per_day, sign_up_time and days_left), also books that have already been supplied are deleted from libraries. If a particular library has no enough time to sign up, or it has no books that haven't been supplied yet we delete it. The libraries are then sorted in descending order according to the determined fitness. Next, one library is chosen to supply books. We substract days_left -= sign_up_time for a given library (instead of iterating day after day). Then we determine number of books to supply from chosen library num_of_books = days_left * books_per_day and supply those books. The process is repeated until there are no more libraries to sign up, there is no more time to sign up the next library (days_left), or timer_ finishes the algorithm running and returns an already determined solution.

Also algorithm wasn't able to determine new ordering of libraries after each new library was signed up, thus we invented simple trick to update ordering of libraries maximally a thousand times depending on how much time do we have, update_count = whole_available_time / 1000, so if there is 3000 days, new order of libraries will be determined each 3 libraries are sign up.

**Conclusion and sources (literature and webpages used):**

- **Genetic Algorithm:**

Knowledge learnt from Artificial Life Lectures with dr hab. inż.

Maciej Komosiński(first semester, Genetic Algorithm was a big part of this lecture).

For sure, we have observed a high coefficient of the dependence of the result on the selection of parameters. For example, parameters for problem c), on which we can receive a reasonable solution, for problem e), give nothing but salty sticks. Another conclusion we can have is that Genetic Algorithms, despite their simple nature, can be very complex in calculation, and can give insufficient results.

Crossover idea from: “Crossover Operators in Genetic Algorithms: a Review'' by A.J. Umbarkar and P.D. Sheth(2015).

- **Greedy:**

Knowledge learnt from Data Structures with dr. inż. Grzegorz Pawlak (second semester, we spent some time discussing greedy approach)

We used  to read others' ideas, but to be honest most time we came up with similar or the same solution to some problems simply by testing different approaches and ending on the most efficient what usually was also the one others developed. (for example using sets instead of lists, or dividing library fitness by sign_up_time that was a complete "accident" in our case but turned out to improve score, etc.)

https://towardsdatascience.com/google-hash-code-2020-a-greedy-approach-2dd4587b6033

https://medium.com/better-programming/google-hash-code-2020-how-we-took-98-5-of-the-best-score-e5b6fa4abc1b