

## Project: Hash Code – Streaming Videos

### Introduction:

This project was generated by Google for their Google Hash Code Contest. It is quite a relevant topic as the number of people watching videos online is continuously increasing, as well as the size of these videos! For these reasons, it is important to be able to handle these video requests reliably and efficiently.

### Requirements

The program will assign different video files to the cache servers in order to minimize the average waiting time for all requests from the different endpoints.

At the end of the program the best overall score will be generated. The score is calculated as the average time saved per request, in milliseconds.

All codes are written in python language.

### Architecture/design:

I have detailed comments in my code which goes step by step to describe what I'm doing in each line of code.

#### *Adding videos*

The design was to assign the videos that were requested by the endpoints to their corresponding (linked) caches, one at a time. This was based on a first come first served system. It was expected that there will be lots replicated and duplicated videos in the different caches linked to the endpoints. To solve the duplicate videos problem in a given cache, a Set Array data structure was used for the caches, meaning that if the video was already in the cache, the same video cannot be added again.

The addition of videos to the caches will start from endpoint 0 until the last endpoint in order and the videos that are being put in will follow that order. For example, at the first endpoint, all the caches linked to them will be iterated through and add the video requested (as long as it doesn't exceed the cache size). The program will iterate through the list, following the endpoints and will put the videos in the caches linked to them as long as there is space for their videos, after the other endpoints had added in their requested videos in their cache. This means that, most likely, as we reach the end of the list there will be little to no space left to add videos for the remaining endpoints. Not a problem though as I'm are not trying to optimize my algorithm right now!

The video order is random (because using dictionary) so not the same videos will go into the caches every time the program runs, so the videos that go into each cache depends on the first video that is listed in the dictionary and its corresponding video size. For example, if the size of cache1 is 100 and the first video grabbed, v0, is of size 80 then the following videos of v1 and v2 with respective sizes of 30 and 50 cannot be put into that cache as that will exceed the cache size. However, if the program grabs the v1 first then v0 will not be able to be put into cache because it will exceed the cache size, however, v2 will be able to go into that cache this time around.

I have also written my codes in such a way that in the case of cache server failure, the user will not be affect. This is because in my program, I pick the most optimal cache (out of all the caches that contains the requested video!) that is linked to each of the endpoints to get each of their requested

video, this means that if one of the cache servers are down, it can still pick the next best cache that is linked to that endpoint for that particular video.

### *Random Search*

As well as just adding videos in an ordered way as mentioned above, I implemented a random search algorithm to add videos completely at random into each cache in the list of caches. This way I can compare both methods.

I decided to run all my algorithms on the cache list that had videos added in randomly. This is because there was more variation in them. I could see more drastic changes in their scores and the results were also much clearer, whereas it appears that my orderly addition of videos to the cache was already fairly good, therefore, using them to compute the algorithms showed very little change and only the tiniest increase in score is every seen and this could take a very long time to happen (I tested this out first hand a few times).

### *Hill Climb*

The hill climb was pretty straight forward, my function iterates through each cache with each video and adds one video (if there is space in that cache) at a time. Each time it runs only one video is added (not an accumulation of videos). I do this by reverting the previous video value from true to false when I'm adding a new neighbour to the matrix.

### *Genetic Algorithm*

Throughout the program I have stored a number of different cache lists inside the parents list. First I would send the lists for mutation, and after than I would choose a random few to evolve (cross-over), creating children, which I would then further evolve, trying to get a better score.

### *Simulated Annealing*

I implemented the simulated annealing in both my hill climb algorithm and genetic algorithm. In the hill climb algorithm I store a list of the random 4 "best cache list" which I will add to my list of best cache list as these would be from the "local best" and then from the genetic algorithm (mutation\_algorithm) I store all the matrices generated, regardless of whether they are the best or not and add them to the list. Then for the evolution, instead of choosing the matrices that gave the best solution, I would choose a random number from the list which I would evolve. And then again, from the children's list, I would choose a random 4 (regardless of which was best or not) and mutate them in hope of finding a better score.

### *Fitness function (calculating the score):*

The score is calculated using the functions from the object REEndpoint: `score_per_EP_per_videoRequest()` and `score()`:

How do they work? E.g. video 1 has request of 200 and the best cache storing that video for endpoint 0 is cache 1 with time saved of 900ms. The score per EP per video request then calculates one part of the total score which  $900 \times 200 = 1800\text{ms}$  (this is the time that the endpoint saves by getting this video from this cache). This is accumulated into a variable in the Endpoint object so at the end of the `best_time()`, all the endpoints will have their own total time saved by using their linked caches and then the `score()` function sums everything up from all the endpoints.

To get the TOTAL score, the function in REmain is to sum up all the scores from each endpoint.

### Extras

To try optimise my score, I tried to add conditions in my cache function "mutate\_solution" where I would only mutate a video from True to False if the video number being passed in was divisible by 3, to randomize it up a bit, but the results didn't seem to benefit from this. It wasn't a low score, but it was never better than the original score, an example is shown in the figure below (using the me\_at\_the\_zoo data set) with a red circle labelled "A". I guess this is because when I do this, there is less variation compared to when I change True to False and False to True every time.

I also evaluated the genetic algorithm to try optimise the results from that. If there were no better solutions generated by the evolution of the parents, then I would mutate the children generated from the evolution list. Sometimes I got a higher score, sometimes not. Again, this is due to random selection.

```
Finished adding video.
Adding video randomly...
Finished adding video randomly...
Original score before using any algorithms: 4517.407464517431
Random search (i.e. completely random addition of videos) before using any algorithms: 1827.0

Starting mutation algorithm...
Finished mutation algorithm...
best mutation algorithm score: 3092.1738821533118
Time to evolve!
Evolution complete
Nope... no good children this time round! Lets make more!
The best overall solution from the genetic algorithm: 3137.0795281542555

Let's try mutating the children...
After trying mutating the children, the best overall score from genetic algorithm: 3530.882262810885
time taken: 3.939631223678589
```

**A**

```
Finished adding video.
Adding video randomly...
Finished adding video randomly...
Original score before using any algorithms: 3694.403246187083
Random search (i.e. completely random addition of videos) before using any algorithms: 1827.0

Starting mutation algorithm...
Finished mutation algorithm...
best mutation algorithm score: 2892.7971958303892
Time to evolve!
Evolution complete
Nope... no good children this time round! Lets make more!
Genetic algorithm for 5 generations didn't provide a better solution compared to the solution produced by the parent.

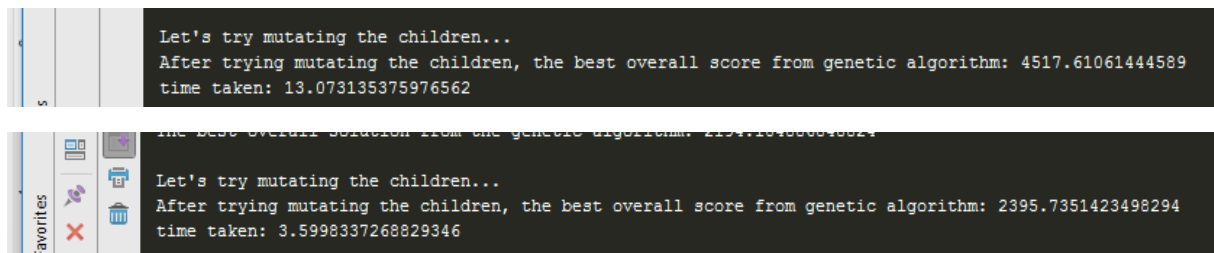
Let's try mutating the children...
After trying mutating the children, the best overall score from genetic algorithm: 4682.158479692174
time taken: 8.454989910125732
```

**B**

As you can see the algorithm that changes every time (circled in red labelled B) gives a much higher score on a good day, but on a bad day it is not far off from the original value. Often times than not, the score from mutation gave a better score than the original score.

I noticed that I had already been optimising my algorithms unconsciously by running time each one through many generations. So to see if there is a difference when I don't run them through as many

generations I got rid of some of my while and for loops. I saved the optimised one in the file called maximising\_score\_generation and the one with less generations generated is called sample.py



The first screenshot shows a terminal window with the following output:

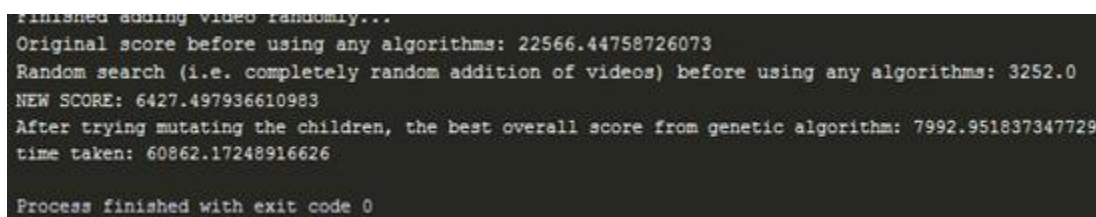
```
Let's try mutating the children...
After trying mutating the children, the best overall score from genetic algorithm: 4517.61061444589
time taken: 13.073135375976562
```

The second screenshot shows a terminal window with the following output:

```
Let's try mutating the children...
After trying mutating the children, the best overall score from genetic algorithm: 2395.7351423498294
time taken: 3.5998337268829346
```

As you can see, getting rid of my loops and changing up some code gave me a much lower score, however, it did increase the performance of my program considerably (which is not surprising seeing as I reduced the complexity of my program by removing loops).

Despite already using some libraries, there were probably more libraries I could have used to replace some of my codes to increase performance and score at the same time. However, my knowledge in the libraries available are unfortunately limited.

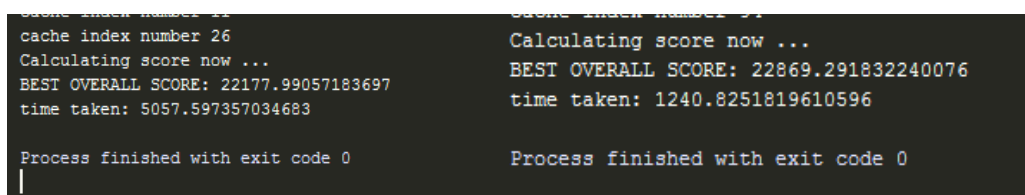


The screenshot shows a terminal window with the following output:

```
finished adding videos randomly...
Original score before using any algorithms: 22566.44758726073
Random search (i.e. completely random addition of videos) before using any algorithms: 3252.0
NEW SCORE: 6427.497936610983
After trying mutating the children, the best overall score from genetic algorithm: 7992.951837347729
time taken: 60862.17248916626

Process finished with exit code 0
```

This score about is from the run for videos\_worth\_spreading file before I removed the loops. As we can see the score from mutating does not give a very good score compared to an orderly addition of videos, and it's definitely not a good time performance, especially because this is EXCLUDING the hill climb algorithm.



The left terminal window shows the following output:

```
cache index number 26
Calculating score now ...
BEST OVERALL SCORE: 22177.99057183697
time taken: 5057.597357034683

Process finished with exit code 0
```

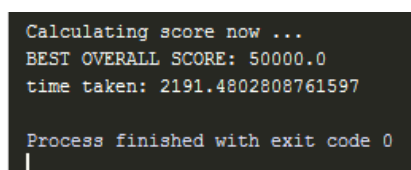
The right terminal window shows the following output:

```
Calculating score now ...
BEST OVERALL SCORE: 22869.291832240076
time taken: 1240.8251819610596

Process finished with exit code 0
```

The run on the left is done with the data file videos\_worth\_spreading using the "faster" performance algorithm... taking 84 minutes to run (and this is excluding the Hill Climb Algorithm!)

On the right, I try to increase the performance (reducing the parent number that is used for evolution so that it runs faster) and it appears that the overall score is not too bad. In fact, it appears that the program now performs at an alright speed as well as giving a better score!



The screenshot shows a terminal window with the following output:

```
Calculating score now ...
BEST OVERALL SCORE: 50000.0
time taken: 2191.4802808761597

Process finished with exit code 0
```

The above pictures shows the score for trending\_today without running the hill climb algorithm, taking roughly 36 minutes which is not bad compared to previous runs!

**Challenges faced:**

## ***(Version 1)***

### **First attempt at writing the program**

When I first started writing the program I had a lot of difficulty using the parsed file given as it was in the format of a dictionary (which we hadn't come across before) so I was having a lot of errors accessing the right part of the code initially. I, for some illogical reason, regrouped all the data that was provided into new variables and lists to try understand and use the data (we will see why it was not a very smart move in just a moment).

#### *Challenges faced:*

I was having a problem with the cache printing out a matrix of size 100, but only 1 out of the total changed at any one time. For example. (using same example of videos and their size from above) Cache list is [100, 100, 100, 100], add v0 to cache 1 results in [100, 20, 100, 100], change remaining size and add v0 to cache 2, [100, 100, 20, 100], it seems that it always reset back to full size and subtract the video size from the total cache 2 size. I realised after many trial and errors that it was because I had the matrix inside the cache object function, which it shouldn't be there. It wasn't that each time I checked a cache it reset to full size, it was that each cache had an array of 4 slots of "video sizes". So each time I thought I was accessing a particular cache, I was accessing that cache BUT I was accessing a particular one of the four elements in the "size array" inside the cache. To fix it, I just removed the matrix variable from the function and made it a global variable inside the cache object. I created a temporary array "list\_remaining\_cache\_size" (remaining cache size x the number of caches) to keep track of the remaining cache size and also it will automatically modify it whenever I add a video to it (this is done in the add\_video\_to\_cache function).

Initially I wanted to be able to remove the videos requested from an endpoint that have been added to a cache so that the remaining videos in the list are videos that haven't been placed inside a cache (those videos are the ones I would need to get from the data centre). I could also use this list to solve the replication of many of the same videos in the different caches by simply seeing which videos have already found a place in a cache.

To do this, I tried using the pop() function, however, this was used in a loop and it gave me an index error: list out of range. This is potentially due to the fact that after I pop the sub-array, it adjusts the index in a way that the loop doesn't work anymore. Instead of trying to pop the video that couldn't be stored in the cache, I decided to add the video's request into a new list so I can use it later when calculating the score.

There were problems with trying to work out the score, i.e. trying to match the video to the corresponding video requests in the videos\_requested\_list which I had created. Reason was because in the video list the videos were numbered accordingly, however in the video variable, video 1 would be assigned the value 0 instead of one. To fix this problem I just added 1 to the video variable.

There were also problems with the order I was calling each condition in the nested loops. In fact there were so many problems with my monster of nested loops.

I used the array data structure for my Endpoint and Cache objects, however I was thinking of changing the cache to a linked list as there are a lot of adding and removing of video files. Endpoint does not change therefore will keep it as an array. Then I realised that it was better to keep it as an array because despite there being a lot of adding and removing of videos, the size of the array was actually set from the beginning. Therefore, adding videos just meant slotting them into the given

index location in the matrix. Linked list would not have benefited this algorithm in any way, so that idea was scratched.

Towards the end of this whole messy algorithm, I was getting list index errors and unfortunately I couldn't locate the error, so I tried implementing the try and except error handlers and tested the index individually. No luck. I was getting a lot of errors in my nested loops despite using the try and except to try locate the root of my problem!

```
try:
    if listCache[cache].get_videoMatrix()[video] == True:
        # if the current cache is linked to current endpoint AND the video is in
        # the list video num req from current endpoint
        if cache in ep_cache_list[endpoint]:
            if vid + 1 in vidNumReq[endpoint]:
                # get_request(video_ep_request, video_size_desc, endpoint, video)
                # list_video[endpoint][video][2]*
                # print("vid req * time saved in endpoint", endpoint, "from cache",
                cache, "= video", vid, ":",
                # listVid[endpoint][vid][2], "x",
                EP_cache_lat[endpoint][cache])
                s += listVid[endpoint][vid][2] * EP_cache_lat[endpoint][cache]
except:
    print("dog listVid", len(listVid[endpoint]))
    # print("dog video index", video)

    print("dog EP_cache", len(EP_cache_lat[endpoint]))
    print("dog endpoint index", endpoint)
```

But this failed to work. I soon came to a realisation that my code was just too messy to fix, especially with the use of an unnecessary amount of nested loops because I didn't understand how to properly use the information that was provided for us. The biggest nested loop I had was with a complexity of  $O(n^7)$ . Yes, I had, in one function, 7 nested loops. Crazy, I know. So, despite having spent many days writing this fully working program (before my errors started!) I decided it would be a better investment to scratch this program and write a new one that is shorter and cleaner (and easier to debug!) rather than to waste more time on this. So, I learned how to use the parsed information properly and wrote out a MUCH shorter and cleaner code!! A miracle, one might say! If you want to have a laugh at my crazy code it's saved under "optimize.py" (the irony of that name!)

## (Version 2)

### Second attempt at writing the program

#### *The design and challenges faced:*

I kept the Endpoint and Cache objects I created in the last program. These objects made it easier to debug the program and also gave the code a cleaner feel. The logic behind the addition is pretty much the same as before only this time it's much cleaner because I'm using the information directly from the dictionary instead of passing each value into another variable (what was I thinking?!).

There were many issues originally with writing the different algorithms, and the hill climb algorithm took a long time to get working because score wouldn't change nor were the requests for videos getting changed. I was also getting the error <built-in function max> because nothing was going into the loop to compute the hill climb algorithm.

So with the overwhelming amount of problems with the first hill climb algorithm, I rewrote the algorithm from scratch again... third time lucky (let's hope!). The difference was that this time I

wrote out the pseudo code in detail in hope of getting a better idea of how to write a functioning one for any size input file (because majority of the time it worked for the smaller file but the problems really only came to light once I tried reading the bigger files).

I eventually realised the root of my problem to <built-in function max>. Turns out I was using the keyword, max, as a variable, so once I changed that to maximum it was fine. Variable names were a bit of a trouble maker for me, because a downside to my solution is that I use a lot of variables, therefore many times I was getting errors because I was using same variable name for different things without realising! For example an error I got often was “TypeError: float is not callable”. Well at least I started to recognise the root of these errors which meant that debugging became a little faster.

An important thing I learnt was that the size of the call for random number IMPORTANT to get a better random search. If the size is low then there is a much lower chance of getting a better random search score, however this depends on the function which I will mention a bit further on in the report in regards to the genetics algorithm.

The design for my code is written with many iterative conditions to try to optimize selection and increase my score. For the hill climb algorithm, whenever a new matrix is generated and gives a better score than the original one, I would store that matrix. However, in the mutation algorithm, I would store the matrix whenever the video number being added in is divisible by 2 with a score that is not better than the original high score. This is my way of implementing the idea of simulated annealing, by taking matrices that are not the best (perhaps the worst). To further that idea, I add up all the matrices from the hill climb and the mutation, and randomly pick 2 “parents” out of that list for each generation of children.

Because the random addition of videos is random and runs for a certain number of times, I had to be careful to not over run this function because then for the genetic algorithm, we convert those that are true to false and false to true, so if there are more true than false, there will be a higher chance of converting from true to false than the other way around, which in this case, lowers the score continuously until there are enough False values in the list to convert back to True.

In this project I went through a lot of bugs and most of them were related to variables, some which I previously mentioned. Another one I encountered was that I had a lot of variables which I didn’t reset when I went on to the next algorithm, so when I tried to calculate a new score the variables in those functions still had values from the previous algorithm which gave me unexpected results.

Some errors were caused by conditions where they were based on Boolean expressions (i.e. do this if true) and I simply forgot to return True/False, resulting in the condition defaulting at false.

I did some online searching about getting unique random numbers so I tried applying that in my functions. As expected with new codes, there was an issue where some of my cache list were just random numbers (not a cache list which I was expecting). I found that my problem was here:

```
parents = sample(range(len(parents)//2, len(parents)), 4)
return mutate_Max, parents
```

I didn’t realise that “parents” were just index numbers which could be used to select from the parents list, so once I realised that problem, I changed it to:

```
parent_index = sample(range(len(parents)//2, len(parents)), 4)
for i in parent_index:
    randomSelection.append(parents[i])
return mutate_Max, randomSelection
```

I kept getting this error and spent a long time trying to fix it as the error message wasn't very descriptive about what was going wrong. It turned out that I needed a condition when I was sampling in case the argument (parents) I was passing into the function was an empty array, so now I have added it in.

```
Starting mutation algorithm...
Traceback (most recent call last):
  File "C:/Users/Nikki/Desktop/DataStructure/input/REmain.py", line 254, in
<module>
    video_size_desc)
  File "C:/Users/Nikki/Anaconda3/lib/random.py", line 315, in sample
    raise ValueError("Sample larger than population")
ValueError: Sample larger than population
if len(parents)>8:
    parents_index = sample(range(len(parents) // 2, len(parents)), 4)
    for i in parents_index:
        randomSelection.append(parents[i])
    return mutate_Max, randomSelection
else:
    return mutate_Max, parents
```

It was a bit tricky setting the limit for how many generations to mutate the cache list for because if it was too high, I ended up in what seems like an infinity loop, but in fact it's just because it takes so long for a tiny change to occur because it's already at an optimal value so to surpass that value (in the condition in the function) it will take a very long time, so with trial and error (using the smallest data set) I managed to find a number that worked.

### Conclusion:

The score using the orderly addition of videos always started off much higher than the randomly added videos. However after hill climbing, carrying out a number of genetic algorithms (mutation and evolution), the list of randomly added videos increased tremendously and usually betters the score of the orderly added videos list!

The removal of many loops and reducing the number of generations definitely increased the performance of my program immensely. It was a matter of trial and error to find the right amount of randomness and the right amount of generations needed to give the better score (not the best because it's hard to tell what the best score would be!)

The file containing all the loops through the generations can be found in `maximising_score_generation.py` and the faster, more efficient file is saved under `sample.py`. After many tests, I came to a conclusion that despite running a number of generations, the results did not appear to improve by much (again this is based on random selection, so the solution/ score varies with every run!).

I tried increasing the performance of my code by using a `@jit` from a library called Numba and with multiprocessing, but I just couldn't seem to get it working. If I had more time I would have definitely liked to explore optimizing the performance of my program as well as the score as processing speed is essential when developing programs.