Algorithms & Data Structures II Project Report

1. Introduction

This report will have three sections. The first section will describe what the system is supposed to do - ie, what is its function, why does each feature exist. The second section will describe how each feature was implemented, detailing design choices, and the third and final section will detail the various practical challenges I faced while implementing this system, and how I overcame these problems. I will detail why I made each decision to demonstrate my appreciation of the task presented.

2. What Is The System Supposed To Do?

All videos that are downloaded on YouTube are stored in a data centre. However, the latency (amount of time it takes a download just to travel between the user and the data centre) is often quite large, resulting in longer-than-possible wait times for a video to be downloaded.

The purpose of this system is to minimise the latency of downloading a video for the user. This is achieved by using cache servers to store the most 'important' videos, however, there are many different variables that effect how important a video is, and which cache server(s) it should be stored in. This system sorts through the request description of each video, and determines the optimal solution for which order the videos should be allocated into caches.

The system has five interdependent key components that link together to produce the optimal solution:

a. The first component is reading the input data. The system takes in the data, and organises it, neatly and concisely, into classes and objects with various methods to display and explain the data (ie, store the EndPoint and Cache objects in separate arrays, for ease of analysis)

b. The second component represents the solution (which videos were allocated to which caches) as a 3D array of integers. The row corresponds to the cache server, and the column corresponds to the video's ID. If a cell first element equals 0, this video was not stored in the cache server, and vice versa for 1. The second element represents the total latency savings of this video's decision ('decision' == whether the video was cached or not)

   However, this 3D array itself is not very helpful to the user, so a summary of the results was addressed in the next component.

c. The third component is the 'fitness' score. This evaluates how efficient the system is, ie, how much latency does it eliminate with its solution. I evaluated this with three main scores/metrics;

- How much latency was eliminated (fitness score, as requested)?
- How many individual requests, from users, were satisfied using cache servers (as a % of all individual requests)?
- How much of the caches' available memory, in MB, was utilised (as a % of all available memory)?

This uses a fitness function, which takes in a solution as an array and calculates its fitness score.

d. The fourth component of the system is the 'hill–climbing ' algorithm. That is, how efficiently can we implement this system, without exhaustively evaluating every possible outcome? How can we allocate certain requests to get the most optimal solution, in a reasonable amount of time? This component delivers a solution which strikes the optimal balance between fitness, individual requests cached and the cost of storing each video.

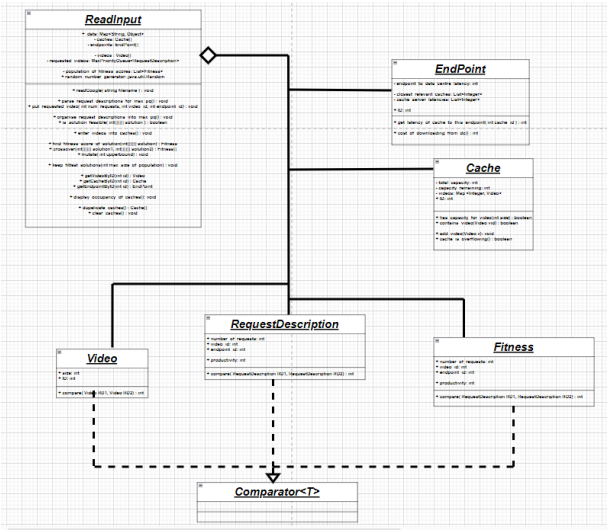
e. The fifth and final component is the genetic algorithm. This component generates 100 (or more/less, depending on programmer specifications) random solutions to the problem, with fitness scores. Then, it crosses-over different parts of random separate solutions to create new solutions, children.

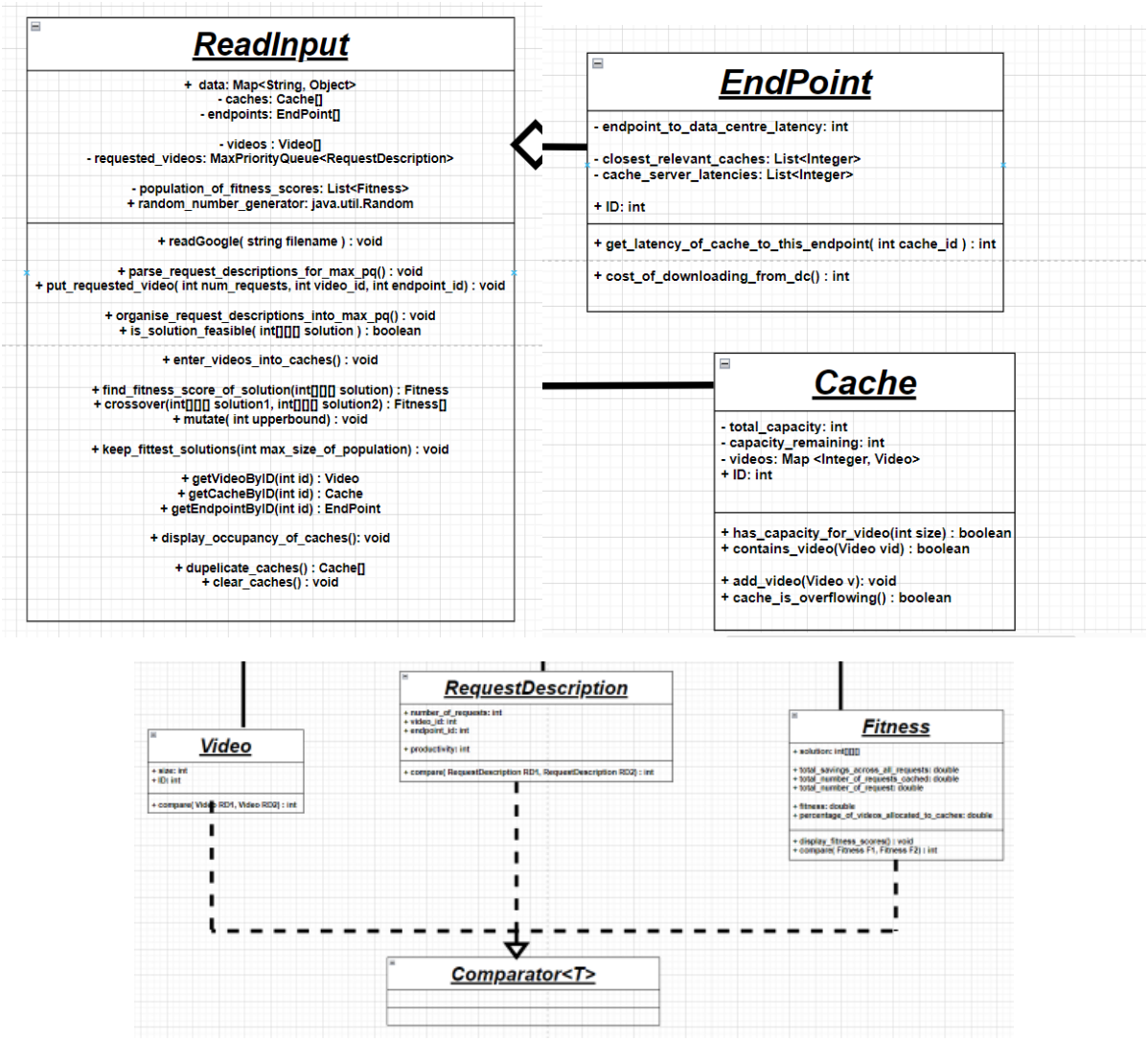
If the children are feasible solutions, they are admitted to the population of fitness scores, otherwise, they are deleted. After every new child has been generated, it mutates some of the values to see if this has a drastic affect on the solution's scores, and then discards any newly unfeasible solutions.

After the viable children have all been added to the population, it saves the 100 fittest solutions, and discards the others. We can now use these solutions to compare against our hill-climbing algorithm, to see if we got stuck in the trap of local maxima.

3. How The Solution Was Designed

Class diagram of implementation:





a. The system was designed to have all data easily accessible to use by the data-processing methods, so all cache, endpoint, video objects, etc, compose the ReadInput object for ease of storing and accessing data, rather than having many accessor and mutator methods for each class.

   For every EndPoint, I designed its list of caches ids to have the cache with the lowest latency first, in ascending order, to the cache with the largest latency last. This way, the system always prioritises the closest caches.

   I kept the RequestDescription objects as small as possible, as there will be O(V) RDs in memory at any given moment, so it is important to preserve as much memory as possible (V == number of videos)

   The Fitness, RequestDescription and Video classes all implement the Comparator interface for encapsulation, making cleaner code.

b. To represent a solution, I believed it wouldn't be helpful if there weren't methods to analyse its fitness. Therefore, I saved the solution as a 3D int array, inside a Fitness object. The first dimension represents caches (ie, each row is a cache), the second dimension represents videos (each column is a video id) and each element is an array of { video_is_in_cache, total_latency_savings_of_caching_this_video }.

Every cache has a HashMap of Video objects, to simulate how the cache would be able to access and manipulate all of the videos like in a real production scenario, not just have the video IDs and sizes. For example, the cache wouldn't just have the size of the data; it would have all of the data (visual, audio data, etc) itself. Thus, this design provides a foundation to scale up the system in future.

I recorded the savings because it was helpful to have the result stored in two places while crossing over two different solutions later on (for calculating fitness).

The memory complexity of a solution is O([(number of caches) * (number of videos)])

c. To save the Fitness data, after a solution has been created, the solution array and all of its relevant data (ie, amount of individual requests cached as a percentage, the absolute latency savings, the fitness score itself) is stored in a 'Fitness' object, which itself is stored in the ReadInput field *population_of_fitness_scores : List<Fitness>*, so it is always accessible to analyse (through the ReadInput object). These fields save a lot of calls to the fitness function as described in the project.

The memory complexity of a fitness object is O(S), where S is the space taken up by the solution field.

d. For the hill-climbing algorithm, I designed the project-specified hill-climbing algorithm as a ReadInput field. I also implemented a Max Priority Queue of Request Descriptions, using the Comparator interface for my alternative solution. This way, organising the requests by priority would take O(VlogV) time, and whichever RDs had the most 'important' attributes would always be accessible in O(logV) time (I will explain how I calculated importance later on).

The top of this Max PQ would have first access to its caches, and then second would have access after, and so on, ensuring the most important videos were always cached first.

A video could only be entered into a cache if there was enough space in the cache, to prevent overflow (in my implementation, it returns a boolean as opposed to an integer if it overflows, as I believe that to be more intuitive to the method's purpose).

Again, I placed this PQ inside the ReadInput object so it would easily accessible by any future functions that may need it.

e. For the genetic algorithm, I stored the results of each genetic algorithm result in the aforementioned *population_of_fitness_scores* in the ReadInput object, for ease of comparing one to another.

To generate random solutions, I used one global Random object to generate random scores for the importance of each request description.
For crossovers and mutations, after checking that the solution is feasible, the system creates new fitness objects for the children, and adds them to the list.

Each generation is not stored separately, but in one list, so the absolute fittest members across all populations are kept in one place, and also because we have no practical use for the unfit solutions. The population does not contain unfit members, as these solutions will never be used in a production application, and I did not see any practical application for unfit members (ie, I did not see the practical value knowing variance provides, if we only want the best solutions)

4. Different Challenges Faced & How They Were Solved

a. The first component presented was to read in the data efficiently, and my first challenge was to view the data itself. To do this, I added a 'SupressWarnings' command to the ReadInput.toString() function, so I could visualise the data, as it was raising warnings about potential casting errors.

My second challenge was storing the data in an accessible manner (ie, instead of indexing into one massive array of IDs, cache sizes, etc, parsing the data into classes with intuitive names). Thus, I took the Request Descriptions data from the global 'data' object provided (as a string), split it into individual strings (after removing irrelevant characters, like '} { =') and parsed those strings into integers, to be stored as RequestDescription object attributes (ie, video id, endpoint id, number of requests).

The next challenge I faced was how to determine which order to allocate videos into caches. Firstly, when initialising the EndPoint objects, I organised the closest relevant caches to the endpoint, in ascending order of latency, ie, the cache with the lowest latency is first, followed by the second-lowest, … all the way to the cache with the highest latency (that is still less than the data centre latency).
This way, the most important videos will always get priority, by getting cached in the closest cache server first.

b. One problem that was very important to address, was to make sure that no cache was overflowing. Therefore, every time a video requests to be added to a cache, we add it iff [(remaining_capacity_of_cache – video_size) >= 0]. This ensures that caches are never allocated more videos than they can store.
If a video is added, the solution array is updated accordingly

c. In the fitness scores section, one challenge I came across was that it would very compute- and time-expensive to get the total savings from just the solutions array, and even if it was done correctly, I believed it was not the most optimal way of obtaining a solution's fitness.

Therefore, while videos are being added, their cumulative statistics (ie latency savings, number of individual requests) are recorded, and when the solution array is placed in a Fitness object, its statistics are stored alongside it. This saves the cost of iterating through the entire array every time to find the solution's fitness, which will deliver large efficiency savings later on when comparing several fitness objects against each other in the genetic algorithm.

Of course, I still implemented a fitness function, which took in a solution, and returns the fitness of the solution. This was necessary when crossing over two solutions later on, in the genetic algorithm.

d. My objectives in this fourth component were to achieve the highest fitness score possible (in a reasonably short amount of time), to serve the highest number of individual requests and utilise as much of the caches' available memory without overflow.

My first approach to the hill climbing algorithm was to select a cell, separately change the 0s in all surrounding cells to 1s (individually) then calculate the fitness to find the change that had the best feasible score. To do this, I created an empty solutions array, and used the Random object to

choose a random cell. Then, I check the surrounding cells for which [video-> cache] gives the best latency savings (without overflowing), move to that cell, and repeat, saving the change with the best savings.

I did this iteratively, for many different values of 'number_of_iterations', but I was not satisfied with the results. I believed this approach of *randomly* selecting cells, then moving forward, could not be the best heuristic possible (greedy algorithms are not always optimal; https://softwareengineering.stackexchange.com/questions/342575/when-does-the-greedy-algorithm-fail) . You can find my approach in (commented out) in *project_specified_hill_climbing_algorithm(…)*)

I took several new approaches to the hill climbing algorithm before settling on one. I mainly used the aforementioned max priority queue to order videos by 'importance'. However, here, my first challenge was to define the 'importance' of a request – ie, which requests get prioritised over others.

In my first attempt, I thought to represent all the different choice-points (inspired from the backtracking component of context-free grammar parsers in natural language processing) as a weighted digraph, which I would run Dijkstra's Shortest Path algorithm to determine the most efficient set of choices. The backtracking component saves the current state of the program every time a choice (video entered into cache) is made, so the undoubtedly most efficient route can be discovered.

Each vertex would represent a video, and the different edges would represent a cache choice (ie, if on video 3, and we choose to enter it into cache 7, with a latency of 550), we would record the choice point as video 3 to video 4, with the edge a latency of 550. This would produce a perfect measure of 'importance' for each request.

However, after some idea expansion, I realised that this was not a hill climbing algorithm, as it was exhaustive, had backtracking (thus was not greedy), and to add to that, an exhaustive algorithm has an exponential order-of-growth anyway, so that was unacceptable.

At this point, I researched some of the most optimal caching algorithms already developed, such as the clairvoyant algorithm (https://www.researchgate.net/figure/Cache-hit-ratio-for-LRU-LFU-and-Clairvoyant-algorithms-The-Clairvoyant-algorithm-shows_fig3_270960529), which had excellent statistics. However, it availed of many types of data which I simply did not have (like the age of a video, or a continuous stream of data), so I tried to understand the nature of an LRU data structure, and incorporate its philosophy of 'most-relevant' videos being most 'important' into my next attempt.

Next, I decided that I would enter the videos into the Max Priority Queue by defining importance as 'number of individual requests' of a video (we could think of the video at the back of the LRU structure as the video with the least requests, and I implemented something similar) . I labelled this the 'Demand' approach. Thus, if we had:

	Num Requests	Video ID	EndPoint ID
Request Descriptions			
RD1	4000	57	..
RD2	5000	43	..
RD3	3000	23	..

The algorithm would sort videos by, and thus give access to caches, in the order; [Video 43, Video 57, Video 23].
This satisfied the requirements of a hill-climbing algorithm, as it cached the most productive videos immediately available. This completed in O(VlogV), V = number of videos. However, this was prone to getting stuck in local maxima, as demonstrated by the following:

	Num Requests	Video ID	Video Size	EndPoint ID
Request Descriptions				
RD1	4000	57	15MB	3
RD2	5000	43	90 MB	3
RD3	3000	23	15MB	3

In the above graphic, we can see that for a cache with 100MB of capacity, it would only cache 5000 requests altogether, with only 10MB of memory left and not enough room for the remaining two videos.

However, if we cache Videos 57 and 23, we would satisfy 7000 requests and have 70MB of memory left – serving more requests with less memory cost, and thus getting better fitness scores.

Therefore, I developed a new heuristic to prioritise videos by: '*Productivity*'.
This prioritises videos by (number of requests / video size). This promotes videos based on how much engagement they receive relative to the cost of storing them – for example; [video 57 productivity = (4000 / 15) = 266.66, video 43 productivity = (5000 / 90) = 55.5, video 23 productivity = (3000 / 15) = 200]. Therefore, it enters videos into caches in order of videos 57, 23, 43. This achieves our theorised improvements above, in O(VlogV) time.

Results of 'Demand' vs 'Productivity' Caching:

Number of videos = 100	"Me at the zoo"	
	Demand-Only	Producitivity
Requests Cached	80.2%	78.9%
Cache Memory Utilised	95.6%	82.0%
Fitness (in microseconds)	461 542	460 481

Number of videos = 10000	"Kittens"	
	Demand-Only	Producitivity
Requests Cached	7.023%	20.1%
Cache Memory Utilised	99.975%	99.923%
Fitness (in microseconds)	207	470

Number of videos = 10000	"Trending Today"	
	Demand-Only	Producitivity
Requests Cached	20.2%	36.5%
Cache Memory Utilised	99.935%	99.928%
Fitness (in microseconds)	1703	2306

We can see here that as the number of request descriptions scales, the 'Productivity' approach caches many more requests, has much better fitness scores and negligibly smaller memory

utilisation scores. Therefore, the 'Productivity' approach is far more efficient than the 'Demand-Only' approach, as the quantity of data scales.

Next, I believed that it would be even more efficient to enter videos by the average latency savings of each video being cached, as well as productivity.
However, given that this would change (as the previous savings would no longer be available as the cache is filled up), this would require dynamically recalculating the latency savings after every video-to-cache allocation. Also, this would see less productive videos bubbling up due to better fitness scores, but also block more productive videos from caches.

After some development, I concluded that the 'Productivity and Latency Savings' approach had O(V^2logV) time complexity and also prioritised the wrong videos sometimes, which is not scalable, so I chose the 'Productivity' method as my final method to compare against the hill-climbing algorithm.

I ran the 'Productivity' method against the project specified hill climbing algorithm (I ran the project's algorithm over 500 times, resulting in too much data to display here). While 'Productivity' was often beaten for smaller sets of videos, its performance improved as the quantity of videos scaled, and the available caches were limited. Thus, the benefits and drawbacks of the Productivity method were clear;

- The benefits of the Productivity method were that it performed much quicker than the project's hill climbing algorithm, and it was very efficient at caching the most important videos. It scaled very well, especially when there was a large number of videos and *relatively* small space available in the caches.
- The drawbacks of the Productivity method were that after creating ~500 random solutions (or more), it never outperformed the best solution from the project's hill climbing algorithm. Ultimately, the Productivity method never produced the best fitness score.

So, my final challenge in this component is answering "what is the optimal hill-climbing algorithm?" My answer - it depends.
If we have a supercomputer, that theoretically can run the project's hill climbing algorithm in a short-enough amount of time, then we should use that, for a better result.
However, the reason we use greedy algorithms is that we do not have that supercomputer, but we still need to come to a decision quickly.
If we are short on time, and can't afford to generate hundreds of solutions, the Productivity method will return a very practical, efficient solution, relative to the time the method took to generate it.

e. In the genetic algorithm section, my first challenge was to create a set of random solutions, which consisted of two sub-tasks; ensure randomness across different solutions, and storing the solutions.

To ensure randomness, I initialised a global Random object at the very start of the program. This was opposed to creating a new Random object at every new solution, because if the Random object used a deterministic RNG, it would give the same set of random numbers every time, and thus the same solution every time.

To store the random solutions, I placed them all in the aforementioned 'Fitness' objects, which implemented the Comparator interface. This made it straightforward to compare solutions,

provided an easy interface for any (hypothetical) future maintainers to understand the code and also enabled me to implement any built-in data structures / frameworks (like PQs).

The second challenge I came across was that even though the solutions were different from each other, the deterministic RNG produced the same set of random solutions every time I ran the program. Thus, I simply increased the number of iterations, which provided more variety.

My third challenge was sorting to obtain the fittest individuals. To do this, I used quicksort to sort the objects, running on average in (FlogF) time, F = number of solutions. This is where having the fitness scores stored saved a lot of time, as we did not have to recalculate the fitness score every time we want to compare two solutions (I reversed the Fitness comparator to place larger scores at the front).

When crossing over individuals, I made sure to check that the solution was feasible. However, I ran into a new problem; the parents' fitness scores were no longer valid, as they applied to the previous solutions, so I recalculated the fitness score every time a new solution was generated, with the project's fitness function.

In determining the probability of crossover, after much iteration, I finalised on a probability of 20% - this was to strike a balance between creating as many individuals as possible, but also not to take too long. To crossover two parents, I chose a random number in *number of videos* , and crossed-over all videos after that point.
For the population size, I found that as I increased the population size from 50, to 200, to 700, to 1000 (as advised by this post; https://cs.stackexchange.com/questions/13062/standard-parameters-for-genetic-algorithms), I found linear improvements in the fitness scores. However, the improvements began to level off after (population_size > 500).

For the probability of mutation, I experimented with several different probabilities, some as high as 0.05%. However, since the mutation was often applied to caches that were already very close to 100% capacity, it often resulted in overflow.
Thus, a challenge here was to firstly take a video out of cache (I prioritised the caches with the most capacity remaining), then enter a new video by mutation. However, this again made no noticeable difference, as I only applied mutations to the most fit solutions in the population (as they are the only solutions we care about), and this generally resulted in only marginally different fitness scores.

I then searched for advice online, and found this post: https://www.quora.com/What-happens-if-I-dont-use-mutation-within-my-genetic-algorithm-Would-I-get-the-same-population-at-each-iteration . While it is not a verifiable source on its own, it reflected my own growing opinion that mutation did not make a difference unless done excessively. After increasing the number of generations from 12 to 120, with a mutation probability of (2 / VC, V = number of videos, C = number of caches) I began to see less marginal improvements in my solutions. There was an even greater affect when I created less random solutions.
However, this excessive mutation did take a lot more time, and I came to the same final challenge as my hill climbing algorithm – when is it appropriate to use mutation?

I ultimately concluded that the value of using mutation, relative to the time it takes to apply it, is maximised when the probability of mutation has a negative relationship with the number of solutions generated. That is, if we have many random solutions generated, a lower probability of mutation is more efficient, while if we don't have many solutions, a higher value of mutation is more valuable, as there is more scope to optimise the solutions.

5. Conclusion

This system allocates videos into cache servers to lower the latency of waiting for a downloaded file. It prioritises the most 'productive' videos when filling up the caches, to satisfy three metrics; total latency saved, aggregate requests cached, and cache memory utilised. The system is designed to allow the easy access of information from one component to another.

The most pressing challenge I faced in this project was how to prioritise one video over another when caching videos. I believed that the use of the project description's fitness function repetitively was inefficient, and thus a challenge, so I developed a mechanism obtain a solution's fitness more efficiently. I also implemented a new hill-climbing algorithm based on my research of existing caching algorithms, and evaluated when it was appropriate to use, and when it was not. With the genetic algorithm, I determined that mutation is most efficient when the probability of mutation has a negative relationship with the number of random solutions generated.

Overall, this project was a valuable exercise in system design & architecture, the practical application of hill-climbing and genetic algorithms, and the implementation of data structures.