Optimisation Algorithms – Lucia Brown

February-April 2024

Contents:

1. Introduction:

The aim of this project is to implement various optimisation algorithms on YouTube videos and caches to ensure handling of requests is efficient and timely. When a request occurs, it can be handled by a cache server which is closer to the user endpoint, rather than a data centre which is far from the desired endpoint of the video.

I have been provided with a sample file from GitHub classroom which contains the functionality of reading in some input files.

The structure of the input files is as follows:

5 2 4 3 100 = 5 videos, 2 endpoints, 4 request descriptions, 3 caches 100MB each.

50 50 80 30 110 = Videos 0, 1, 2, 3, 4 have sizes 50MB, 50MB, 80MB, 30MB, 110MB.

1000 3 = Endpoint 0 has 1000ms data center latency and is connected to 3 caches.

0 100 = The latency (of endpoint 0) to cache 0 is 100ms.

2 200 = The latency (of endpoint 0) to cache 2 is 200ms.

1 300 = The latency (of endpoint 0) to cache 1 is 300ms.

500 0 = Endpoint 1 has 500ms data center latency and is not connected to a cache.

3 0 1500 = 1500 requests for video 3 coming from endpoint 0.

0 1 1000 = 1000 requests for video 0 coming from endpoint 1.

4 0 500 = 500 requests for video 4 coming from endpoint 0.

1 0 1000 = 1000 requests for video 1 coming from endpoint 0.

As seen above, caches have certain size restrictions. This cannot be violated and still result in a valid optimisation.

I discuss my plans for obeying the cache condition in the Requirements section.

2. Requirements

The system is designed to take in various provided .in files:

“example.in” - a brief example,

“me_at_the_zoo.in” - a moderately sized input

“kittens.in” and “trending_today.in” - extremely large inputs

The system is designed to use various optimisation algorithms to improve the latency.

First, a 2D solution array must be developed. This 2D array will be based around the variables “number_of_videos” and “number_of_caches” which are to be read in by the provided main function.

The latency is first calculated by the designed fitness() function.

The optimisation algorithms are then run.

The fitness() function is run a subsequent time, to evaluate the change in latency before and after the optimisation step.

At no point should the system exceed the capacity of the various caches.

For this purpose, I will be implementing a method to evaluate the cache condition and make changes if it is violated. This method should be called within each of the optimisation algorithms.

The optimisation algorithms which are required for this project are as follows:

A hill-climbing algorithm:

This algorithm assesses the current latency of the video/cache relationship. It evaluates the “neighbouring” solutions, and if a “better” neighbour was found. If the neighbour’s solution is deemed a better solution, the neighbouring solution becomes the new current solution. This process continues with the current solution’s new neighbours. Hill-climbing algorithms are vulnerable to getting stuck in local maxima.

This occurs when a better solution is found, the current solution is updated. But no better neighbours are discovered. Even though there may be a better solution than the current solution in some other iteration which is not a neighbour.

A genetic algorithm:

A genetic algorithm mimics real-world genetics to find optimal solutions. First, an initial population is generated. The fitness of the “individuals” in the population are evaluated. From these individual solutions, “parent” solutions are chosen. A new “generation” of solutions can thus be created from the parent solutions through series of crossovers and mutations.

Once a final generation of solution is reached, the best solution from that generation can be chosen as the final solution to the genetic algorithm.

Genetic algorithms are much more complex than the other two required algorithms.

A greedy algorithm:

A greedy algorithm is a simplistic algorithm. It chooses the best choice at every step, regardless of if this will actually result in the overall best choice.

Given each video request, the greedy algorithm attempts to assign videos to caches in a “greedy” way such that it takes the best step at each iteration.

3. Architecture & Design;

Requirement 1 – Reading Input Files:

Fortunately, the requirement of reading in the various input files was already satisfied with the file I received from GitHub Classroom, so no architectural decisions were made regarding reading in the files.

I made no changes to the functionality of reading in the files.

Requirements 2 & 3 – Solution Array and Latency Calculation:

As discussed in the requirements section, the cache condition is essential for optimisation.

To ensure that my output returns meaningfully in the event that the cache constraint is violated, my fitness function contains the line:

if (contentsSize > cacheSize) return -1;

Which will return –1 from the fitness function immediately, without attempting to calculate latency with an invalid cache.

I made the architectural decision to put this line within the fitness function because each of the optimisation algorithms will make a call to the fitness function at various steps in the optimisation to ensure that the results are in fact improving.

For similar reasons, in the fitness function, I chose to pass the solution array as a parameter to fitness():

public int fitness(boolean[][] solution) {}

Because of how the fitness function is being used within my algorithms, I thought it wise to pass the current solution as a parameter, as opposed to attempting to do more calculations and processing within fitness().

Another key architectural decision made was the decision to format the 2D solution array as a boolean[][] array, instead of some other primitive data type.

The presence of a video being in a cache is denoted with zeros and ones.

Again, to avoid needless processing, I decided that my solution array should be created as a boolean.

```
boolean [][] solution = new boolean
[(int)ri.data.get("number_of_videos")][(int)ri.data.get("number_of_caches")];
```

Requirement 3 – Latency Calculation:

The fitness function works as follows:

- It obtains the sizes of the videos and caches
- It then obtains the request information and the latency between the endpoint and the cache
- It enters a for loop which iterates around each video request and calculates latency to the data center
- It updates the current latency, if a worse one is found, i.e. currentLatency > latencyToCache
- It adds the found latency for that video request to a running total variable called totalLatency. And repeats for all requests

Requirement 4 – Hill-Climbing Algorithm:

The hill-climbing algorithm works as follows:

- The initial currentLatency is set to the value obtained by the fitness function
- A variable is set to define how many iterations are allowed without finding an improvement (to avoid wasted time and resources), i.e. int maxConsecutiveNoImprovment=50;
- If an iteration does not improve the latency, the consecutiveNoImprovement variable will increment
- While the maxConsecutiveNoImprovment condition has not been met, the algorithm iterates around the 2D solution array and evaluates the neighbours to the current solution by making a call to generateNeighbourSolution(solution, i, j); where i and j are the indexes of the 2D array
- The algorithm then evaluates the solution obtained from the helper method to see if the generated solution is better. If it is, it becomes the new solution, and currentLatency is updated, consecutiveNoImprovement is set back to zero
- Once the algorithm terminates, it returns the optimised solution

The private helper method generateNeighborSolution works as follows:

- A new 2D array called neighbourSolution is initialised
- The current solution is copied to the neighbourSolution
- The method inverts the Boolean values of the neighbourSolution, i.e. if a video is contained in a cache, it is changed to not. If a video if not contained in a cache, it is changed to true
- This inverted neighbour is returned to the rest of the algorithm to be assessed

Requirement 5 – Genetic Algorithm:

The genetic algorithm works as follows:

- The algorithm creates its initial population by making a call to generateRandomSolution() and adding the given solution to an arrayList of generated solutions. These are the “individuals” of the initial population

- The algorithm then iterates through the individuals of the population and evaluates their fitness by making a call to the fitness function and adds these fitness values to another arrayList called fitnessValues
- The algorithm then chooses two “parents” by making a call to parentSelection(population, fitnessValues, selectionSize, random);
- The parents chosen are added to a parents arrayList
- The new generation of solutions is created by making a call to crossover() and mutation()
- Offspring are created from crossover: boolean[][] offspring1 = crossover(parents.get(i), parents.get(i + 1), random);
- The population is increased with mutation: population.add(mutate(offspring1, mutationRate, random));
- This repeats for the specified number of generations
- Once this is completed, the algorithm returns the best solution by evaluating fitness again

The private helper method generateRandomSolution works as follows:

- It creates a 2D array called randomSolution
- It populates the boolean values of the 2D array at random for all indexes of the array andt hen returns

The private helper method parentSelection works as follows:

- It sets a variable bestIndividual to null and sets bestFitness to Integer.MAX_VALUE
- It iterates randokmly through the population and evaluates the fitness of the individuals
- It returns the bestIndividual
- This function is called twice in the main algorithm to choose two parents

The private helper method crossover works as follows:

- It takes in two parents as parameters and creates an empty 2D boolean array called offspring
- It selects a crossover point using Random
- It iterates over the offspring 2D array, inserting the content of parent1 up until the crossover point and inserting the conent of parent2 from the crossover point to the number of videos given from the input data
- This method calls the enforceCacheSizeConstraint method on offspring that has been created
- It returns the offspring after ensuring that the cache constraint hasn’t been violated

The private helper method mutate works as follows:

- As ints, it stores the numberOfVideos and numberOfCaches. It creates a 2D boolean array called mutatedSolution.
- It copies the values of the current solution into the mutatedSolution array.

- It takes in a value called mutationRate. It iterates through the 2D array and checks if the value at that index is less than the mutation rate.
- If the value being assessed is less than the mutation rate, the method flips the value, introducing a random change to just some of the values.
- After mutations, it makes a call to the enforceCacheSizeConstraint method to ensure that the mutated solution is valid.
- It then returns the mutated solution and allows the rest of the genetic algorithm to continue.

Requirement 6 – Greedy Algorithm:

The greedy algorithm works as follows:

- The algorithm takes in all of the parameters from the input data and initialises a 2D solution array
- In a for loop, it iterates over all the video requests from the endpoints for (String videoEndpointKey : videoEndpointRequest.keySet()) {}
- The algorithm identifies the cache server with the least latency for that endpoint by iterating over the caches
- Once the cache with minimum latency is found, the video is assigned to that cache server, making this a greedy algorithm. Choosing the minimum cache latency at each individual iteration without consideration for what the global best choice may be
- The algorithm also makes a call to the enforceCacheSizeConstraint() method discussed above after exiting the outer for loop, to ensure that the solution that will return from the greedy() method is in fact valid.

4. Challenges & Solutions:

Issue 1 – Fitness Function:

The first issue which I encountered when running my program was that sometimes the fitness function was returning negative latencies.

This does not make sense considering the task at hand.

This issue was partially caused by my early implementations of the optimisation algorithms, but fixable through just the fitness function.

Solution:

This issue was caused by the improper handling of caches. Violating the cache condition resulted in negative latencies.

This was the issue which inspired me to add the “return -1” functionality to my fitness function.

This solution actually proved to be extremely beneficial when I was debugging my optimisation algorithms.

For example, in one iteration of my greedy algorithm, the cache condition was violated.

So, the next time that the fitness function was called within the greedy algorithm, a "-1" was returned quickly and allowed me to assess what was wrong with the greedy algorithm.

Issue 2 – Hill-Climbing Algorithm:

In the early development of my hill-climbing algorithm, there never seemed to be any optimisation!

This was due to logic issues.

The way which my hill-climbing algorithm is set up, it uses Java's Integer.MAX_VALUE feature in order to start the search for better neighbours.

I am unsure how, but the algorithm always thought that this worst-case-scenario number was the correct answer.

Solution:

To amend this I added some more boolean logic for safety. I added a variable foundBetterNeighbor=false

This variable is only ever set to true when the neighbouring solution has a lower latency.

This new variable allowed me to add the condition

if(foundBetterNeighbour){}

This condition fixed the issue of the good neighbouring solutions being overwritten by Integer.MAX_VALUE at each iteration of the hill-climbing

Issue 3 – Genetic Algorithm:

The primary issue with the genetic algorithm is speed.

Even with the small inputs "example.in" and "me_at_the_zoo.in", the genetic algorithm is visibly slower than the other two algorithms.

This can easily be observed by eye.

Solution:

Due to the nature of genetic algorithms, there is no real way to bring it up to speed with the other two algorithms in this project.

The genetic algorithm makes calls to five different private methods to achieve the desired result (enforceCacheSizeConstraint, generateRandomSolution, parentSelection, crossover, mutate).

Whereas the hill-climbing algorithm only calls 2 (enforceCacheSizeConstraint & generateNeighbourSolution)

Issue 4 – Greedy Algorithm:

As discussed in the fitness function section, the greedy algorithm was the one most prone to returning a "-1" result

Solution:

The greedy algorithm is not allowed to return a value of –1 with the following condition making sure that the minimum latency cant be –1: if (bestCache != -1)
{solution[Integer.parseInt(videoId)][bestCache] = true;}

Overall Issues

Overall, in the beginning I had some issues with understanding the nature of the project and what exactly was required of me.

It took me a decent amount of research to understand how the different required algorithms worked.

As for my understanding of the project itself and how all the read-in parameters such as cache and videos work, I sought out clarification from classmates.

5. More

An evaluation of the running cost of my helper methods, frequently called in the program:

Fitness:

- Getting video and cache sizes from the data structure = O(1)
- Iterate through the cache and number of videoes = O(c*v) where c is the number of caches and v is the number of videos
- Calculate total latency for each request by iterating over the requests and over the caches = O(r*c) where r is the number of requests
- In total, this method runs in O(c*v + r*c)

Enforce Cache Size Constraint:

- Getting video and cache sizes from the data structure = O(1)
- Iterating over the files of the cache in a nested for loop = O(c*v)
- Removing files from the cache if the cache constraint was violated = O(1)
- In total, this method runs in O(c*v)

An example of the running cost of one of my algorithms:

Hill-Climbing Algorithm:

- The private helper method (generateNeighbourSolution) has initialisation statements O(1), a nested for loop O(c*v), and an operation that inverts the 0's and 1's in the cache. O(1).

Overall the helper method runs in O(c*v)

- Creating the initial solution array = O(1)
- Evaluate initial fitness = O(c*v + r*c)
- Nested for loop over number of videos and number of caches = O(c*v)
- Within the nested for loop, 3 methods are called:
  - GenerateNeighbourSolution = O(c*v)

- EnforceCacheSizeConstraint = O(c*v)
- Fitness = O(c*v + r*c)

- O(c*v) * (O(c*v) + O(c*v) + O(*v + r*c))
- The rest of the operations within this algorithm are of cost O(1), as they are assignment, comparison and return statements
- The total running cost of the hill-climbing algorithm is O(c*v) * (O(c*v) + O(c*v) + O(*v + r*c))

6. Conclusion

In conclusion, I am satisfied with the outcome of the project and what I learned from participating in the project.

I now feel confident in my understanding of the algorithms that were used throughout the project.

I am happy with the research that I undertook to gain understanding of these algorithms.