Coláiste na Tríonóide, Baile Átha Cliath
Trinity College Dublin
Ollscoil Átha Cliath | The University of Dublin

Report

Module : Algorithms And Data Structure II

Module Code : CSU22012

Module Lecturer : ANTHONY VENTRESQUE

Name : Inam Ul Haq Syed

Student Id : 21364500

Date : 01/04/2024

Introduction

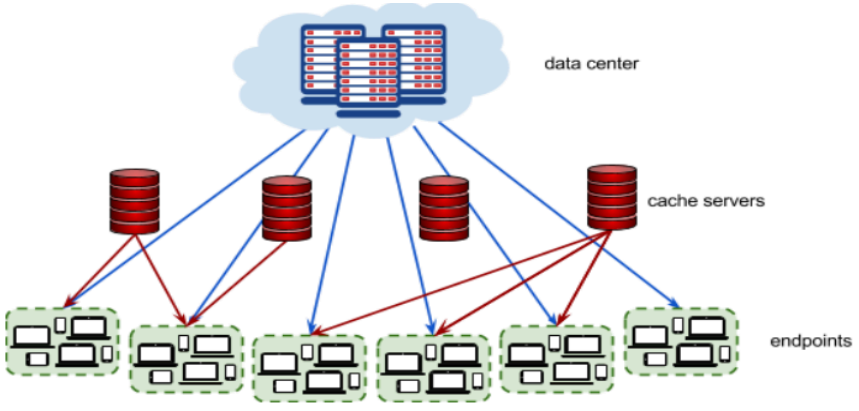
In this project, our primary objective was to implement and optimise an algorithm tailored to assist a real-life video serving infrastructure. Video serving infrastructure typically utilises cache servers to efficiently handle user requests. These caches store copies of popular videos, allowing them to fulfil user requests locally instead of fetching the videos from the central data centre, which could come with a very high cost.

Each video in the infrastructure has its own size, and each cache has a specific capacity. Additionally, the transmission time from the user's endpoint to both the data centre and the caches plays a crucial role and is a major factor.

Our primary task revolved around **devising an algorithm that intelligently allocates videos to caches in the most optimal manner possible.**

When I first encountered this project, I felt that the problem bears resemblance to a well-known algorithmic challenge known as the knapsack problem. Much like organising items into a bag without exceeding its weight limit while maximising the total value, our goal was to allocate videos to caches without surpassing their capacity constraints, aiming to achieve the highest possible efficiency.

Once I recognized the similarities between the given problem and the knapsack problem, I knew that devising a solution would likely involve employing techniques similar to dynamic programming or possibly utilising a greedy algorithm.



Functionality and Architecture/Design

What does the System do ?

The system's primary objective is to tackle the given problem by intelligently allocating each video to cache servers to maximise efficiency. It employs two main solutions:

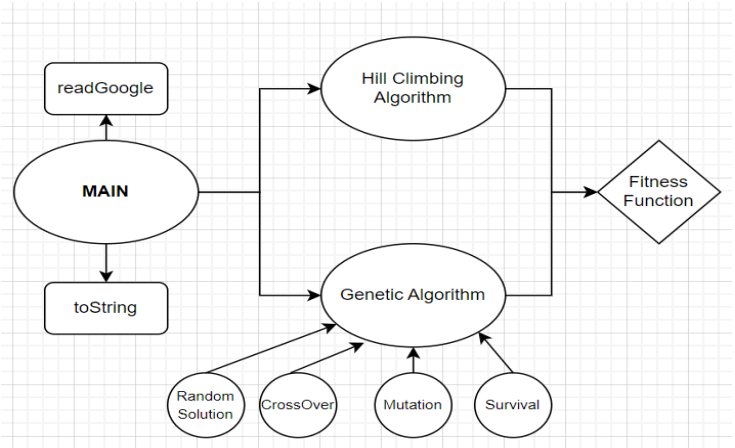
1) Hill climbing Algorithm
2) Genetic Algorithm

The system implements both algorithms to assess their effectiveness in solving the problem at hand.

- The Hill Climbing Algorithm is a type of greedy algorithm that consistently makes changes it believes will lead to the best solution. However, it may encounter some limitations, which we'll discuss later.
- The Genetic Algorithm is an evolutionary algorithm which uses biological evolution principles. By iteratively evolving populations of solutions over multiple generations, genetic algorithms efficiently explore complex search spaces and converge towards high-quality optimization solutions.

Architecture and Design of the System

The design of my solution revolves around a single main function, along with two distinct algorithmic functions that implement different approaches. The genetic algorithm function incorporates several helper functions, enhancing code readability and navigation. Additionally, a crucial function central to both algorithms is the fitness function. It plays a pivotal role in evaluating the efficiency and outcome of the solution, aiding in the assessment of the algorithms' performance.



Section 1 : Reading a problem File

Task

The task was relatively straightforward. Our objective was to read the files and verify the input to ensure everything was functioning correctly.

Action

My approach involved thoroughly understanding the data structures and functions involved, especially the “readGoogle” function. I familiarised myself with the structure of each component, including arrays, lists, and hashmaps, understanding their stored data and use. This approach proved particularly beneficial as it laid a solid foundation for future tasks. I also documented the functionality of each code block within the "readGoogle" function using comments, enabling me to recall the purpose of each data structure and its associated information.

Section 2 : Representation of Solution

Task

This task involved devising a representation for the solution. As the ultimate goal was to illustrate the placement of videos in caches, we needed to establish a suitable approach for representing this.

Action

My approach used a 2D array, where the rows denoted caches and the columns represented individual files where the index was the identifier for the caches and files. I determined the row and column sizes based on the variables obtained from the readGoogle function. I opted for a 2D array over a list of lists due to its robustness. The potential issue with lists is their mutable size, which could lead to data corruption.

Problems

One significant challenge in this procedure was ensuring that the total size of files within each cache did not exceed the cache's capacity. This required some calculation to address the issue effectively.

Testing

I validated my 2D array representation by creating a dummy array and passing it into the function, observing the output on the terminal.

Section 3 : Fitness Function

Task

This task/function was crucial. The reason for this is because it was used for evaluating our solution and determining the most efficient algorithm. However, I found this function to be tedious, as it required understanding the precise storage mechanism of all data related to the video files. Essentially, this function assessed the feasibility of a solution and, if feasible, evaluated its efficiency by providing a fitness score. A higher fitness score indicated a better solution.
The fitness score = Average time saved per request.

Action

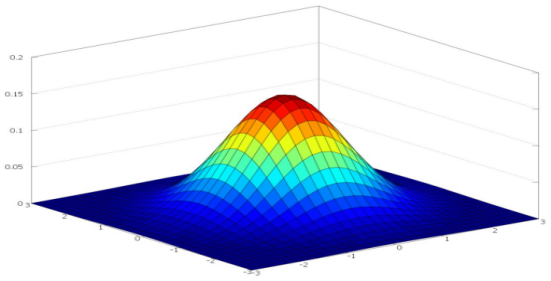
I followed the document's instructions to calculate the fitness score. This meant extracting data mainly from a HashMap named "population." I then did calculations to figure out the cost from the endpoint to the data centre, compared it to the cost from the endpoint to the cache, and found the time saved. The steps were all outlined in the document.

Problems

In this function, it was crucial to ensure that all data used to calculate the fitness score came from the correct source. This required a solid understanding of the different data structures used to store the video data. This included “video_ed_request” HashMap of Strings, "ep_to_cache_latency" list of lists, etc.

Testing

After creating the function, I tested it using the provided 2D array from the Google Doc. The expected result was also provided in the same document. By passing the 2D array to the fitness function and comparing the result with the expected output, I verified the function's correctness. I obtained a result of 462500 ms, which matched the expected output from the Google Doc, it confirmed that the function was operating as intended.



Section 4 : Hill Climbing Algorithm

Task

The objective of this task was to implement a greedy algorithm known as the hill climbing algorithm to automatically populate the array. Once populated, the array was passed to the fitness function to assess its feasibility and calculate its fitness score.

Action

The hill climbing algorithm first initialised an array filled with 0s. Then there was a for loop which iterated through each element of the 2D array, converting that specific location into a 1 to evaluate if it improved the fitness.

If the fitness score improved, the coordinates of that element (location) were stored. We iterated through the whole array, to find the element that yielded the highest fitness score and changed it to a 1, while the remaining elements retained their original values. We did multiple passes of this until we reached a point where altering any element from 0 to 1 did not result in any fitness improvement. At this point, the newly created array was returned.

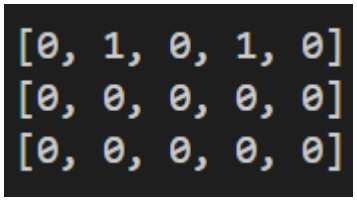
Problems

The main issue of the hill climbing algorithm is its susceptibility to getting stuck at a local optimum. This situation arises when the algorithm becomes trapped in a local maximum or minimum and cannot find a better solution because it only evaluates immediate neighbouring solutions. Since hill climbing is a greedy algorithm that only considers the next steps, it may overlook superior solutions that require non-greedy moves.

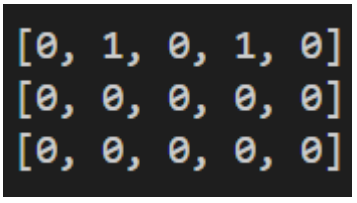
Testing

I tested my hill climbing function by initialising an array of all 0s. Then the procedure of populating the array took place. After obtaining the final array, I calculated its fitness score, which was higher than the example in the Google document. The resultant array also differed.

Resultant Array (example.in)



Google Doc Array (example.in)



Section 5 : Genetic Algorithm

Task

In this task, we utilised the Genetic Algorithm to address our main problem. Genetic algorithms are a subset of evolutionary algorithms. These algorithms are inspired by natural selection and evolution, reflecting the stages and processes observed in nature. Here are the different stages and sub-functions involved in constructing this algorithm.

1) Creating a population of random solutions
2) Crossover between the individuals within the population
3) Mutation of individuals in the population
4) Survival of the fittest individuals

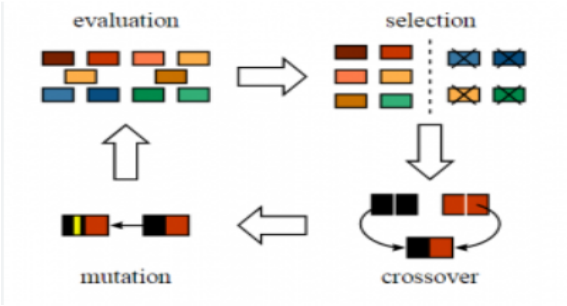
1) Creating a population of random solutions

Parameter involved : Num_Of_0s_Bias

Task : This task involved generating a random population of 2D arrays initialised with 0s and 1s as the starting point.

Action and Analysis of parameter : A simple call to the "randomSolution" function, created and returned, a new 2D array. However, a crucial aspect of this task/function was the Num_Of_0s_Bias parameter. This bias determined the probability of each element being 0 when randomly distributing 1s and 0s within the new array.

I noticed that a high value of this bias was necessary to generate feasible/valid 2D arrays. This became particularly evident when I attempted to run the algorithm on larger datasets like kittens.in. In such cases, if the bias wasn't very high, there would be enough 1s in the 2d array (Files per cache) to exceed the capacity of ALL * our caches hence making all our solutions (the whole population) infeasible.



2) Crossover between individuals within the population

Parameter involved : Crossover_Bias

Task : After obtaining our population of 2D arrays, we randomly selected individuals and produced two children from these two parents. Each child inherited half of its data (genetic material) from one parent and the other half from the other parent.

Action and Analysis of parameter : My crossover function takes two 2D arrays as input and produces a new array containing data from both parent arrays. Initially, I considered a strategy of alternating rows between the two parents in the new array. However, I used a simpler approach where each child inherits the 1st half from parent 1 and the other half from parent 2. The occurrence of this was dictated by the Crossover_Bias.

Higher Crossover_Bias = Higher Mating = Greater population = More calls to fitness function = Slower run time.

Problem : I encountered an issue while selecting two parents for mating. Initially, I used a for loop to sequentially pick parent 1 and randomly pick parent 2. However, I realised this approach had a flaw: there was a possibility that parent 1 could randomly select parent 9, and vice versa*, leading to identical children.

Solution : I modified the selection process by choosing parent 2 as the next consecutive element in the population eliminating the possibility of the randomness error occurring.

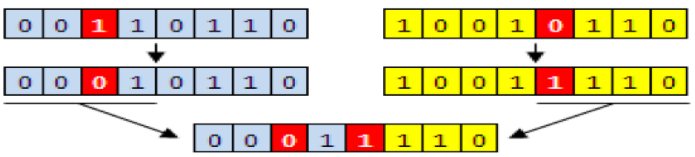
3) Mutation of individuals in the population

Parameter involved : Mutation_bias

Task : Introduced random variations to individuals in the population, simulating genetic mutations.

Action and Analysis of parameters : In this step, we iterated through all members of the population. Based on the mutation bias, we randomly selected an individual (2D array). Within that individual, we randomly chose a location to flip.

Increase in Mutation = Greater variation = Greater possibility of finding a better solution



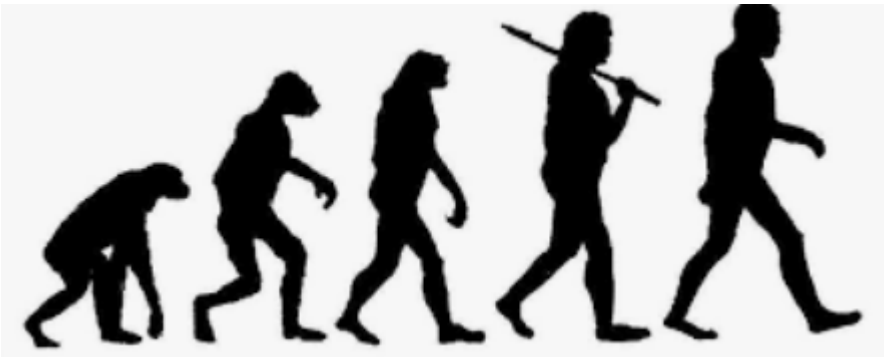
4) Survival of the fittest individuals

Parameter involved : New Population size
Task : Here we determine the survival of the fittest individuals based on their fitness scores. This is crucial as it means that only the top fittest individuals are allowed to survive and produce offspring.
Action : To achieve this task, I utilised both a HashMap and a PriorityQueue.

Initially, I computed the fitness score of each 2D array in the population. I then created a HashMap where the keys represented the index of each element, and the values represented their corresponding fitness scores. This mapping allowed me to associate each fitness score with its respective 2D array in the population.

Next, I added all key-value pairs from the HashMap to a PriorityQueue, which automatically sorted them based on their fitness scores. By extracting the top 50 key-value pairs from the PriorityQueue, I obtained the indexes of the 50 fittest elements in the population.
With these indexes, I constructed a new population consisting of the top 50 fittest individuals.

I opted for PriorityQueue over traditional sorting methods because it provided greater flexibility in selecting the desired number of elements and facilitated easier navigation to obtain the indexes of the top individuals. Additionally, PriorityQueue's inherent sorting capability saved time and streamlined the process.



Section 6 : Conclusion

This project provided a valuable learning experience, offering insights into the practical applications of algorithms in real-world scenarios. It highlighted the effectiveness of breaking down complex problems into manageable steps for systematic resolution. I was particularly fascinated by the utilisation of genetic algorithms, which effectively translated real-life phenomena (evolution) into algorithmic processes, showcasing the power of computational methods in problem-solving.

References

https://www.hindawi.com/journals/mpe/2015/906305/

https://www.geeksforgeeks.org/genetic-algorithms/

https://www.geeksforgeeks.org/genetic-algorithms/

https://www.javatpoint.com/hill-climbing-algorithm-in-ai