Optimisation Algorithms

Dylan Gallagher

April 1, 2024

Introduction

This project focuses on the real-world problem of effectively storing videos on cache servers to minimize the average waiting time for video requests. To achieve this objective, I have implemented advanced optimisation algorithms such as genetic algorithms and hill climbing. Hill climbing is a heuristic-based optimisation algorithm which searches for solution by checking neighbouring solutions. On the other hand, genetic algorithms attempt to find optimal solutions by mimicking the process of evolution. This report will give a quick overview of the system, I will also go into detail on the implementation and refinement of these algorithms. I will also cover the process of implementing these algorithms and document any issues I had during the development process.

Requirements

This section will describe the requirements of my system. The task of my system is to assign videos to cache servers to minimise average waiting time for all requests, given a description of the cache servers, network endpoints and videos.

Videos

Each video has a size given in megabytes (MB). The data center stores **all videos**. Additionally, each video can be put in 0, 1, or more **cache servers**. Each cache server has a maximum capacity given in megabytes.

Endpoints

Each **endpoint** represents a group of users connecting to the Internet in the same geographical area (for example, a neighborhood in a city). Every endpoint is connected to the data center. Additionally, each endpoint may (but doesn't have to) be connected to 1 or more cache servers.

Each endpoint is characterized by the latency of its connection to the data center (how long it takes to serve a video from the data center to a user in this endpoint), and by the latencies to each cache server that the endpoint is connected to (how long it takes to serve a video stored in the given cache server to a user in this endpoint).

Requests

The predicted **requests** provide data on how many times a particular video is requested from a particular endpoint.

Implementation

This section will cover my implementation of the system.

Reading the file

I used the provided `ReadInput` Java class and `readGoogle` method to parse the input file into a Java `HashMap` in the following format (sample data).

Solution Representation

I chose a Java 2D array to represent my solution. A big advantage of using Java 2D arrays, is that the outer array is composed of references to inner arrays. The rows represent different caches and the columns represent which files are stored in those caches in a one-hot encoding.

`solution[i][j] = 1`, if cache i has file j, and 0 otherwise.

Fitness Function

In order to asses the fit of a particular solution, I implemented a fitness function that calculates the score of a solution. It takes in a solution in the format specified above and outputs an integer representing the score. I followed the algorithm specified in the project description, and confirmed it works on the sample inputs.

The fitness function first check whether a solution is valid or not, and returns -1 if it's invalid.

Hill Climbing

Hill climbing is one approach for optimisation problems. It works by iteratively generating neighbouring solutions and choosing the best one, repeating the process until it reaches a local optimum.

Hill climbing algorithm can either start from an empty solution (no videos stored on cache servers) or start from a randomised solution. I implemented both ways and ran tests on them.

Genetic Algorithm

The genetic algorithm attempts to mimic evolution and natural selection. Each iteration, or in genetic algorithm terminology, generation, the algorithm goes through a process of selection, crossover, mutation, and insertion back into the population.

Population

The population is a group of "individuals" or "chromosomes", each representing a solution. Each individual in the population can be evaluated using the fitness function.

Population size is one of the most important parameters to the genetic algorithm. If it's set too low, there's not enough possibilities for mating, and the fitness of the population as a whole does not improve drastically. If the population is too large, the algorithm becomes extremely slow. Striking a balance in the population size parameter is critical.

Population Initialisation

There are many ways of initialising a population for the genetic algorithm. Most implementations rely on random initialisation, but in our case, each individual in the population has conditions, i.e., they must be valid solutions. I implemented random initialisation for my population with different levels of "sensitivity". Sensitivity is a parameter that determines the sparcity of the solution. If the sensitivity is too high, too many 1s are filled in and most solutions are invalid, but if sensitivity is too low, very few 1s are filled in and this hinders the performance of the genetic algorithm.

Selection

Each generation, a number of individuals from the population are selected to be "parents". The process of selecting parents can drastically affect the performance of the genetic algorithm. I implemented multiple processes and will discuss them.

Rank Selection is a method of choosing parents in which we simply rank the individuals in a population and pick the top k individuals to be parents, where k is the parameter controlling the number of parents. This method works well when the fitness values of the individuals in the population are very different, i.e. we have a lot of variety in the population.

Roulette Wheel Selection is a method of choosing parents in which we have an imaginary roulette wheel, which we split into sections, where each individual has a section with a size proportional to it's fitness. The fraction of the roulette wheel an individual x_i has is given by the formula:

$$\frac{f(x_i)}{\sum_{i=0}^{n}} \tag{1}$$

Tournament Selection is a method of choosing parents in which we pick two individuals at random from the population and select the better of them to be selected as parents. This process can be repeated as many times as needed to meet the "number of parents" parameter of the genetic algorithm. This process uses the random selection process to its advantage, because this introduces more variety into the next generation by not always picking the best individuals each generation.

Crossover

Crossover is a process that mimics biological reproduction, in which two individuals / parents combine to produce offspring. A splitting index is chosen, and the genes / solution of both parents are split at that index, and swapped with the other section of the other parents. This process produces two offspring. If the offspring are invalid, they are removed.

The crossover parameter is vital to the performance of the genetic algorithm. It determines the probability that crossover occurs. Since it's a probability, it can take on values from 0 to 1. If crossover = 1, then we perform crossover with all parents, and all offspring are made from a crossover. If crossover = 0, then we never perform crossover. Obviously, the optimal value is somewhere in between, and I discuss this further in the exploration of parameters section.

Another parameter that can either be set or randomised is the splitting index. It is normal to split the genes 50:50, but you can also determine it with a uniform random distribution, in which each index is equally likely to be chosen as the splitting index.

Mutation

The next stage in the genetic algorithm is mutation. This process involves randomly selecting a single piece of the chromosome / solution and inverting it. This generates randomness and variety in the population, which improves the performance of the genetic algorithm.

The mutation parameter controls the probability that mutation occurs. It is critical that this parameter isn't set too high. Normal values of this parameter are around 0.01. If this parameter is set too high, too much randomness gets injected into the population, which normally leads to a less fit population, but if used in moderation, it can increase the performance of the genetic algorithm.

Replacement Strategies

After the parents have been chosen, and crossover and mutation have occurred, we have to decide whether to put the offspring back into the population. Since we normally keep the size of the population constant, if we add the offspring back into the population, we must also remove some individuals from the population.

Elitism is a replacement method in which the top k "elite" individuals always move forward to the next generation and the remaining individuals are at risk of being replaced by new offspring.

Above-Average Replacement is a replacement method in which offspring are only inserted into the population if they have an above-average fitness. They will replace the individuals with the lowest fitness function in the population.

Stopping Conditions

Deciding when to stop the genetic algorithm can save a lot of time and wasted computation. Generally, during the execution of the genetic algorithm, the fitness of the population steadily increases until it reaches a plateau. Multiple early-stopping techniques can drastically reduce the runtime of genetic algorithms while also maintaining a good solution.

Maximum number of generations

Perhaps the simplest stopping condition that is present in most genetic algorithm implementations is the maximum number of generations. This stops the evolution process after a pre-defined number of generations.

Diversity Threshold

Generally, with a random initialisation of the population, the individuals of the population are quite diverse (they are far away from each other in the solution space). This is a good thing in the early and middle stages of the genetic algorithm as it avoids getting stuck in local optima, but in the late stages of the algorithm, the individuals of the population tend to cluster together in the solution space.

A measure of similarity between individuals can give a diversity metric to a population and when diversity drops below a certain level, we stop the algorithm, as this signals that all of the individuals are nearby in the solution space and algorithm will not explore drastically different solutions.

Fitness Threshold

If we have a fitness goal beforehand, we can set a fitness threshold, and once we get an individual with a fitness score exceeding that, we stop the algorithm and take that solution. Similar to stopping the algorithm after a maximum number of generations, setting a fitness threshold can limit the potential of the algorithm to generate good solutions.

Stagnant Population

Another potential early-stopping method is trying to detect plateaus in the fitness of the population. The genetic algorithm tends to increase steadily and then plateau, so a simple condition checking if the fitness has improved in the last 20 or so generations is a good indicator of a plateauing population. This can save a lot of time, but if you are too aggressive with this early stopping method, it can limit good solutions.

Greedy Algorithms

I implemented a greedy algorithm that greedily adds videos to a cache, with the ones with the most requests from that cache first. This served as a decent algorithm in its own right, generating quite competitive solutions, but passing these solutions into the hill climbing algorithm helped them converge to the local optimum, increasing it's fitness slightly.

Difficulties Faced

Performance

While implementing this project, I mainly faced difficulties with the performance of the genetic algorithm. It worked fine for the small inputs, but was extremely slow for anything bigger. Initially, my implementation was quite computationally intensive, calculating the fitness function many times per generation. I did tournament selection, which would calculate the fitness function for each contender, I would sort the population based on their fitness function each generation to remove the weakest individuals. This drastically slowed down my algorithm, so I changed it to be less computationally intensive.

Fitness Function

I experiences a mysterious bug in my fitness function that only appeared in inputs larger than example.in, The bug was related to overflow when calculating the total time saved and resulted in large negative values being returned for the fitness of some valid solutions. The fix was trivial once the bug was found.

Exploration of the Performance of Algorithms

In this section, I will dive into the performance of the algorithms as the size of the inputs scale.

isValidSolution()

This method checks whether a given solution is valid, i.e. it makes sure that the caches aren't assigned more videos than they can store.

It uses two nested for loops, which loop over the solution 2d array. The solution 2d array is of size (num caches) * (num videos), so the runtime of this method is $O(CV)$, where C is the number of caches, and V is the number of videos.

Fitness Function

Initially, the function calls isValidSolution(), which we know runs in $O(CV)$ time.

It then has 2 nested for loops, the outer loop iterating over requestsMap, which has R requests. The inner loop loops over the cache servers, which has C caches, so the runtime of this nested for loop is $O(RC)$.

All other operations are constant time, so the total runtime of this method is $O(CV + RC)$.

Hill Climbing

The hill climbing algorithm is harder to judge the runtime of, because it depends on the number of iterations, which is dependent on both the initial solution passed to it as well as the solution space. We can call the number of iterations I. There is an outer for loop looping over once per iteration, so this runs I times.

Inside the for loop, there are two nested for loops, looping over both numCaches and numVideos. Inside the inner loop, there is a call to the fitness function, which we know has a runtime of $O(CV + RC)$. The runtime of these nested for loops is $O(C^2V^2 + C^2RV)$. The rest of the computations run in constant time, so this is the runtime per iteration.

Since we let I be the number of iterations until stopping, the runtime of Hill Climbing is $O(IC^2V^2 + IC^2RV)$.

This can be made much faster if we use variants of Hill Climbing such as first choice hill climbing, when we choose the first neighbour that is fitter.

Exploration of Parameters for Genetic Algorithm

Exploring parameters for the genetic algorithm is a critical step in insuring that it works properly and efficiently. There are a few parameters fundamental to the genetic algorithm, and others are optional depending on the implementation.

Population Size

Finding the balance in population size is critical, as larger populations can lead to greater diversity and the opportunity to explore the solution space better, but they also bring computational complexity to the algorithm. The optimal value of the population size is dependent on constraints such as hardware and maximum preferred running time. On my computer, I found that a population around 250 strikes a nice balance.

Mutation Rate

Mutations should be used extremely sparingly in the genetic algorithm. While they offer tremendous potential to introduce diversity into the population, this can also be their weakness. Too much mutation tends to make the population too diverse, and I found it generally took longer to converge to a decent solution. After experimentation, I found that a mutation rate of 0.001 tends to work fine.

Crossover Rate

Crossover is the bread and butter of the genetic algorithm. I started with a crossover rate of 0.5, and experimented with extreme values such as 0.05 and 0.95. I immediately found a sharp increase in the fitness of the population when I increased crossover. A crossover rate of 1.0 also worked fine for this particular problem. I settled on a crossover rate of 0.9, as I found it to give a good balance.

Number of Parents

My implementation of the genetic algorithm takes a specified number of parents that are selected for mating each generation. This parameter is also quite important to set right for the performance of the algorithm. If set too low, e.g. 2 parents per generation, evolution of the population occurs extremely slowly, while if it's set too high, this both increases the computational complexity of the algorithm, but also tends to destabilise the population. After experimenting with different number of parents in different sized populations, I found that choosing approximately 10% of the population to be parents tends to work well.

Elitism Count

The elitism count, in certain implementations of the genetic algorithm, controls the top k individuals in the population that automatically get through to the next generation, while the rest are at risk of getting replaced. This parameter fundamentally controls the algorithms balance of exploration vs exploitation. Setting it higher tends to exploit solutions known to be good, while setting it lower allows for greater diversity. This parameter is relatively important, but setting it benefits from knowledge of the solution space. In simpler solution spaces, such as example.in, you can set this as high as you like, as the solution space makes it relatively easy to reach the global maximum. In more complicated solution spaces, the algorithm greatly benefits from a lower elitism count, as it allows the algorithm to explore more diverse solutions.

Number of Generations

This parameter is relatively straightforward to generate. My implementation of the genetic algorithm keeps the best performing individuals, so running for extra generations does not decrease the fitness of the best individual solution. If there is no other early-stopping implementation, this parameter determines how long the algorithm runs. Deciding on your maximum preferred runtime, doing a quick test to see how many iterations per second your algorithm can do and working backwards is a decent strategy for setting the number of generations. I found that there was no significant improvement after about 1000 generations.

Parallel / Distributed Computing

While researching the genetic algorithm, I found out that it can be very easily parallelised. I implemented an ”Islands” technique, where I have multiple different populations evolving on different islands. The islands are separate of each other and at the end of the evolution process, each island submits their best individual to face off against the individuals of the other islands. The best of these individuals is the overall best individual and this is the solution I chose.

This is implemented in code using parallelism, each island is running on a separate core, which means overall runtime is not effected much, but we effectively have multiplied our population by n by using n cores. Initially I tried this with 6 islands (on my 6 core desktop computer), but also experiemented running it on 96 virtual cores on an AWS virtual machine (I used free credits that were about to expire). It scaled up gracefully, with the runtime being in the same ballpark as my 6 islands on my local computer.

This can easily be scaled up to be distributed across different machines, as the logic for initialising the islands and combining the best individuals is already done. I briefly tested a really hacky distributed computing setup, where I had my PC and my MacBook both running the code, they would then export their n best individuals from the islands to a csv. I manually copied the solutions to my Desktop, combined the solutions, and chose the best one. This served as a proof of concept, and of course a more robust solution would be more streamlined, but I can explore this in the future.

Conclusion

This project gave me a lot of freedom to explore vastly different methods to accomplish the same task. A surprising observation I made was that fundamentally different algorithms, such as hill climbing and the genetic algorithm, tend to give very similar results.

To wrap things up, the topic of optimisation algorithms is extremely interesting and diverse. Mimicking nature (genetic algorithm, ant colony optimisation, particle swarm optimisation) or intuition (hill climbing) all tend to work well, and all have a clear interpretation in the real world.