Streaming Videos

GitHub Repository: https://github.com/BrunoM6/Y3S2-IA

Beatriz Ferreira | Bruno Moreira | Miguel Cabral
3LEIC12

Problem Specification

Given a description of **cache servers**, **network endpoints** and **videos**, along with **predicted requests** for individual videos, decide **which videos to put in which cache server** in order to **minimize average waiting time** for all requests. The network endpoints have a specified number of requests for certain videos, with associated latencies for both the data center and the cache servers. This problem was presented during the 2017 Qualification Round for Google's Hash Code competition.

Each "problem formulation file" contains information on:

- the number of videos
- the number of endpoints (and which cache servers are connected to them)
- the number of request descriptions (times a specific video is requested by a certain endpoint)
- cache servers and their capacity in MB

Streaming Videos as an Optimization Problem

Solution State: a Python dictionary data structure, where each key is a cache identifier and the value is a list of the videos stored in the respective cache.

Solution Constraints: the sum of the size of the videos stored in a cache server can't surpass the cache server capacity

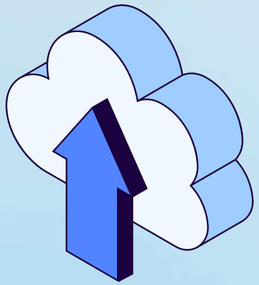
We created a **get_neighbors** method in order to compute the neighboring solutions for **Tabu Search** and **Simulated Annealing.** It was also used as a **mutation method** for the **Genetic Algorithm.**

Furthermore, the **Genetic Algorithm** also performs crossover between two parent solutions, but this will be further explained later on.

Data Parsing and Result Storage

The parsed data gets divided into the following data structures:

- **problem_description**: list of integers, always with the first five values that describe the problem at hand;
- **video_size**: list of integers, the index of the video is the index at the array, and the associated value is the list size, so size of video 0 is video_size[0];
- **endpoint_data_description**: list of integers, the index of the list is the endpoint being described and the value is the latency to the datacenter;
- **endpoint_cache_description**: dictionary of tuple (int, int) mapped to int, key is a tuple (endpoint_index, cache_index) that maps an endpoint-cache pair to a latency, not every combination of endpoint and cache is described;
- **request_description**: dictionary of tuple (int, int) mapped to int, key is a tuple (endpoint_index, video_index) that maps an endpoint-video pair to a request number;



Due to some performance issues we experienced when we first started running meta heuristic algorithms to solve this problem, we decided to start storing the intermediate solutions. This is used so that everytime we run an algorithm it can pick up from where it left off without having to retrace all the steps it took to get to that specific point.

In the genetic algorithm, this result storage is also used to generate some of the initial population, which leads to more optimized results over time.

Greedy, Random Approach and Score Method

Greedy Approach

To provide our solution space with a decent starting position, we apply a **greedy** heuristic to our solution before any other algorithms. It is defined in the **greedy.py** file. It works by **computing and comparing the score** (total time saved) for every potential placement of videos in every cache based on the description; In order to save time, this **result is stored.**

Random Approach

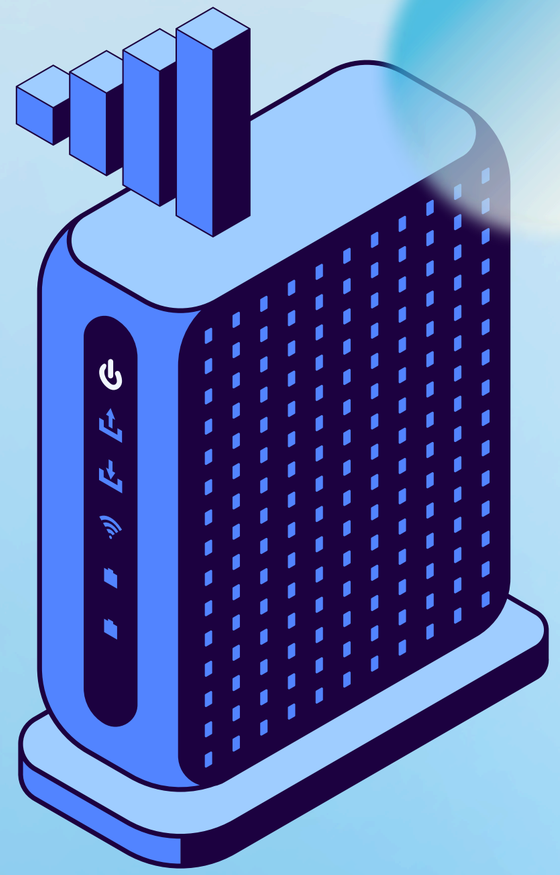
A random starting point from a function defined in our **random.py** file serves as a **control group** for our algorithms, in which we fill every single cache with a maximal random amount of videos, respecting the constraints of size. This is used to test how quickly our algorithms get to a satisfactory score given an arbitrary starting point

Score Method

Our scoring function, present in the file **score_functions.py,** has two components:

- **base scoring -** runs through all the request descriptions and computes the total time saved
- **rescoring -** reverse the score calculation to obtain the original total time, and re-calculate the score after adding optional parameters of changes, previous solution score and state

Yet again this was a decision made in order to **boost efficiency**, since calling the total scoring function every time would make running our implementations unfeasible for medium and large datasets.



Tabu Algorithm

- Chosen by user
  - can pick what neighbor generating function is used

- iteratively explores neighbor solutions
  - add
  - swap
  - remove

- keeps tabu list to prevent cycling back to recent solutions (solutions stay on tabu list based on defined tabu tenure)

- accepts move if
  - not tabu
  - tabu but beats global best

- returns best solution found only if it improves on initial solution

Simulated Annealing

- Chosen by user:
  - maximum number of iterations
  - capacity of iterations without improvement
  - initial temperature
  - cooling rate
  - minimum temperature
  - neighbors generated

- iteratively explores solution space using neighbor generating functions
  - add
  - swap
  - remove

- **annealing -** utilizes current temperature to define odds of acceptance of worse solutions

- incrementally improves solution quality

- doesn’t generate much improvement when used with greedy as a starting point, especially for smaller data sets

Hill Climb

- Chosen by user:
  - maximum number of iterations
  - number of neighbors
- Number of iterations can be fine tuned in order to handle large data sets
- Gets neighbors from specific state and chooses best improvement
- Stops when none of the generated neighbors are better and returns current best
- Finds local bests, works better on random start rather than greedy start

Genetic Algorithm

- Chosen by user:
  - population size
  - generation number
  - mutation rate
  - tournament size
  - elitism (yes or no)

- Runs for a fixed number of generations
- Initial population consists of greedy, mutated greedy and random solutions
- Chosen fitness function is the score method used in other heuristics
- Parents chosen in tournament between randomly picked X number of solutions
  - offspring generated by switching cache contents between parents, may be mutated based on mutation rate

Visualization Method

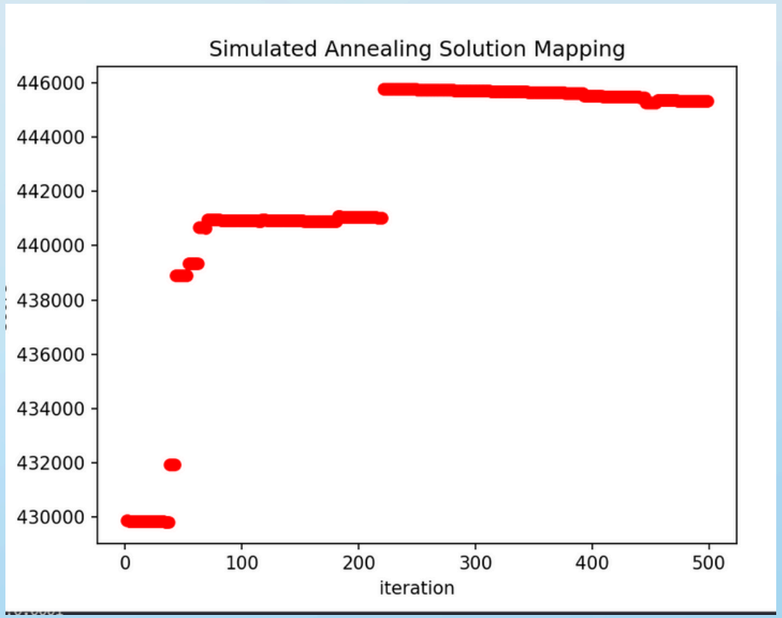
We used **matplotlib** in order to be able to view the algorithm progression for our heuristics.

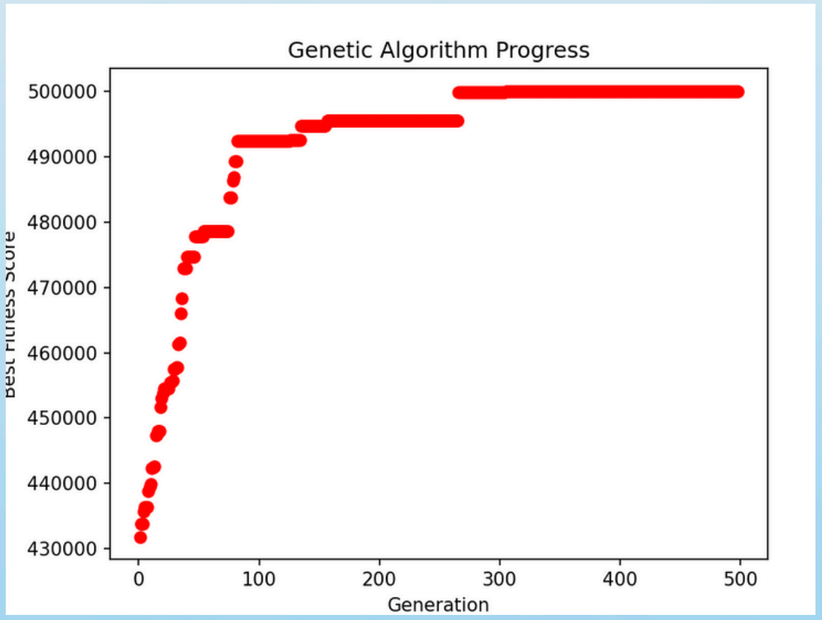
The user is able to toggle visualization on and off since it slows down execution time.

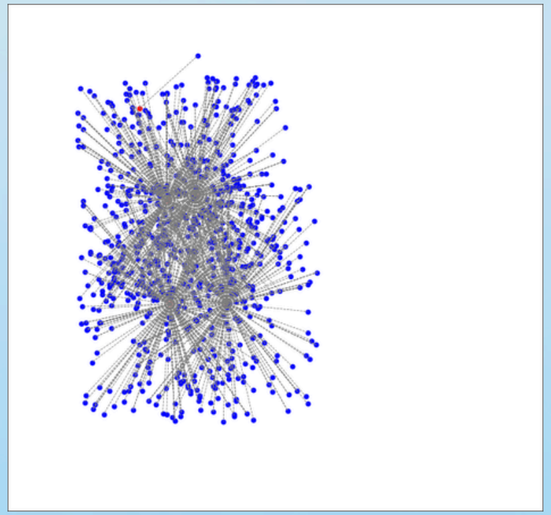
Our base approach is implemented in the **visual.py** file and the data collection is controlled within each relevant method.

We have two different types of graphs: one that plots the **evolution of the best score** over iterations/generations (used in genetic algorithm and simulated annealing) and another evolution tree visualization graph that tracks the exploration of different solutions and their neighbors in Tabu Search and Hill Climb.

Results







Tabu Search: Search tree result

Bibliography

In order to solve the Streaming Videos Optimization problem, we used as reference the materials provided in the original prompt, such as the problem description and the data files built by Google for the competition. When it came to picking out which algorithms to use, we decided on trying to implement the ones that were mentioned during the theory classes **Lecture 3b: Meta-Heuristics – Simulated Annealing and Tabu Search** and **Lecture 3c: Optimization and Genetic Algorithms.**

We also used generative AI tools, such as **ChatGPT,** in order to guide our approach, but always made sure to double check the information with what we learnt in theory class and what made sense based on what was already implemented in other methods.

Thank You!