IART Report

Streaming Videos - Grupo 50

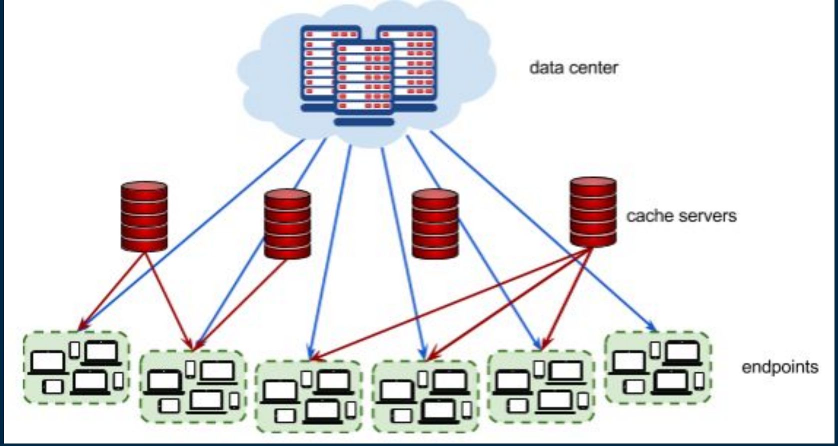
Ana Clara Gadelho - 201806309
Flávia Carvalhido - 201806857
Leonor Gomes - 201806567

Definition of the Optimization Problem: Streaming Videos

Whenever a YouTube User wants to watch a video, a request is performed. The videos are all stored in the data server but in order to minimize the waiting time, some popular videos are stored in cache servers. In order to do so, there's a need to figure out the best cache servers to store each video, while minimizing the average waiting time for all requests.

Each video has a size given in MB. Each video can be put in 0, 1, or more cache servers. Each cache server has a maximum capacity. Every video is stored in the data center.

An endpoint represents a group of users connecting to the Internet in the same geographical area. Every endpoint is connected to the data center and each endpoint may be connected to 1 or more cache servers. Endpoints are characterized by the latency of their 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 an endpoint is connected to (how long it takes to serve a video stored in the given cache server to a user in this endpoint).



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

Videos, endpoints and cache servers are referenced by integer IDs. There are V videos numbered from 0 to V – 1 , E endpoints numbered from 0 to E – 1 and C cache servers numbered from 0 to C – 1.

Formulation of the Problem

Rigid constraints:
Cache size: a cache server cannot store more than its max capacity.
Latency: the latency between a server and an endpoint is constant.

Solution representation:

3	We are using all 3 cache servers.
0 2	Cache server 0 contains only video 2.
1 3 1	Cache server 1 contains videos 3 and 1.
2 0 1	Cache server 2 contains videos 0 and 1.

Evaluation functions:
We wish to maximize the saved time for each request, so each state will be evaluated with: L_D-L (time if the video was transmitted directly from the data center minus the time it takes to transmit the video from the cache/data center with the smallest latency that is connected to the endpoint). We shall then multiply this by the times this video is requested and sum all the saved times in order to get a global evaluation of the solution (the more saved time, the better the solution).

$$\max\left(\sum_{r=0}^{N} Num_r \left(L_{D_r} - L_r\right)\right)$$

Formulation of the Problem

Neighborhood/mutation and crossover functions:

Neighbour1: Substitute a video in a cache for another

```
def subVideo(data,sol):
    count = 0
    while True:
        (randVideo, randCacheid) = sol.getRandomVideoFromCache()
        count += 1
        if count>=30:
            newSol = data.generateRandomSol()
            return newSol
        randCache=sol.caches[randCacheid]
        if randVideo == 0:
            if sol.caches[randCacheid].currentCapacity == sol.caches[randCacheid].maxCapacity:
                continue
            randVideo = data.getRandomVideo()
            if sol.caches[randCacheid].addVideo(randVideo):
                break
        else:
            otherRandVideo = data.getRandomVideo()
            if not randCache.checkVideo(otherRandVideo) and randCache.canSwapVideos(randVideo, otherRandVideo):
                randCache.takeVideo(randVideo)
                randCache.addVideo(otherRandVideo)
                break
    newSol = deepcopy(sol)
    newSol.subCache(randCache)
    return newSol
```

Formulation of the Problem

Neighborhood/mutation and crossover functions:

Neighbour2: Switch videos between caches

```
def swapVideos(data,sol):
    count = 0
    while True:
        (randVideo1, randCacheid1) = sol.getRandomVideoFromCache()
        (randVideo2, randCacheid2) = sol.getRandomVideoFromCache()
        count += 1
        if count == 30:
            newSol = data.generateRandomSol()
            return newSol
        if(randVideo1==0 or randVideo2==0): continue
        randCache1=sol.caches[randCacheid1]
        randCache2=sol.caches[randCacheid2]

        if randVideo1.id != randVideo2.id and randCache1.id != randCache2.id and randCache1.canSwapVideos(randVideo1, randVideo2)
            randCache1.takeVideo(randVideo1)
            randCache2.takeVideo(randVideo2)
            randCache1.addVideo(randVideo2)
            randCache2.addVideo(randVideo1)
            break

    newSol = deepcopy(sol)

    newSol.subCache(randCache1)
    newSol.subCache(randCache2)
    return newSol
```

```
def neighbourFunc(data,sol):
    if random.randrange(2) == 0:
        return swapVideos(data,sol)
    else: return subVideo(data,sol)
```

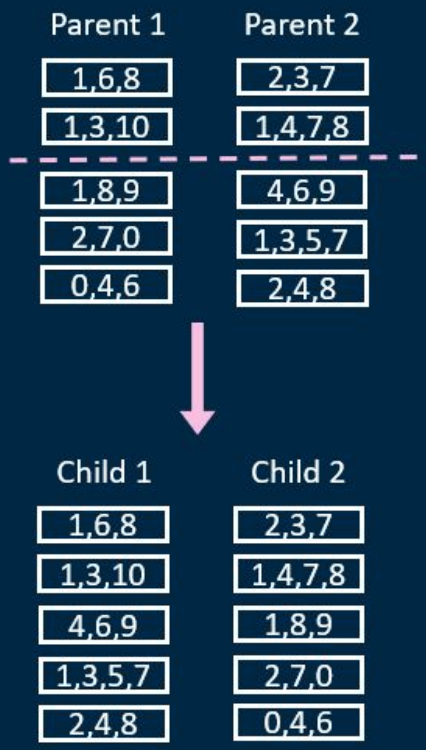
Neighbour3: Randomly choose between the two functions

Formulation of the Problem

Neighborhood/mutation and crossover functions:

Crossover function: Choose a random crossover point (index in the caches list). Divide parents' caches accordingly to the crossover point and each children gets a part of each parents configuration.
Child with est fitness is returned

```
def classicalCrossover(population,data):
    parents=tournament(population)
    x=parents[0]
    y=parents[1]
    #choose the crosspoint
    crossPoint=random.randrange(len(x[1].caches))
    cachesX=(x[1].caches)
    cachesY=(y[1].caches)
    # build new cache groups
    c1=cachesX[0:crossPoint]+cachesY[crossPoint:len(cachesY)]
    c2=cachesY[0:crossPoint]+cachesX[crossPoint:len(cachesY)]
    child1=(x[1])
    child2=(y[1])
    for i in range(crossPoint):
        swapCachesContent(child1.caches[i],y[1].caches[i])
        swapCachesContent(child2.caches[i],x[1].caches[i])
    ev1=data.evaluation(child1)
    ev2=data.evaluation(child2)
    # choose the best child
    if(ev1>ev2):
        return [ev1,child1]
    else:
        return [ev2,child2]
```



Implementation

Python 3 was be used to develop the project with Visual Studio Code.

As for data structures, the problem was represented by classes (next slide) to represent the problem such as Video, CacheServer, Endpoint and Request.

The structure of the file directory is:

- **readme**
- **src:** a folder for the main file and sub-folders
  - **structure:** a folder for files that define structures used in the project
  - **input:** a folder with input files
  - **algorithm files**

The Approach

Classes

```python
# class to represent videos
class Video:
    def __init__(self, size, id):
        self.size = size
        self.id=id
```

```python
# class to represent caches
class CacheServer:
    def __init__(self, maxCapacity,id):
        self.id=id
        self.maxCapacity = maxCapacity
        self.videos = set()
        self.currentCapacity = 0
```

```python
#class to represent endpoints
class Endpoint:
    def __init__(self, latency,id):
        self.id=id
        self.latency = latency
        self.dic = {}
```

```python
#class to store all the information of the problem
class Data:

    def __init__(self, videos, numCaches,sizeCaches, endpoints, requests):
        self.videos=videos
        self.numCaches=numCaches
        self.sizeCaches=sizeCaches
        self.endpoints=endpoints
        self.requests=requests
```

```python
#class to represent a request
class Request:
    def __init__(self, video, endpoint,ammount, id):
        self.video = video
        self.endpoint = endpoint
        self.ammount = ammount
        self.id=id
```

```python
#class to represent a possible solution
class Solution:
    def __init__(self, numCaches,size):
        self.caches=[]
        for i in range(numCaches):
            self.caches.append(CacheServer(size,i))
```

The Approach

Evaluation function

```python
# returns the saved time of each atended request
def getSavedTime(self, request,sol):
    dataCenterTime=request.endpoint.latency

    time=dataCenterTime

    for cacheId in request.endpoint.dic.keys():
        cache=sol.caches[cacheId]
        if cache.checkVideo(request.video):
            tenp=request.endpoint.dic[cacheId]
            if time> tenp :
                time = tenp     # searches for lower streaming time for each request
        else:
            continue

    return (dataCenterTime-time)*request.ammount    # multiplies saved time by the ammount
```

```python
# evaluation function
def evaluation(self,sol):

    t0=time.perf_counter()
    t=sum([self.getSavedTime(r,sol) for r in self.requests])

    t1=time.perf_counter()
    # print(t1-t0)
    return t
```

To evaluate the solutions we calculate the saved time for every request: the difference between streaming a videos directly from the data center and sending it from the nearest cache server.

Algorithms Implemented

Hill Climbing

This implementation is a basic “Hill Climbing” Random, which means, for each iteration, it generates a random successor of the current state and if it’s better evaluated than the current state, selects it and applies it. The stopping criteria used was a select number of iterations where the current state didn’t improve.

Simulated Annealing

In this algorithm, there’s an initial random solution and an initial temperature. When generating a random successor of the current state, if it’s better evaluated than the current state, selects it and applies it. If not, there’s a probability of selecting it as well. At each cycle, the temperature is decreased at a 0.9 rate. The algorithm stops when the temperature reaches the minimum defined temperature.

Algorithms Implemented

Genetic - Steady State

Consisted in creating a random initial population and then in each generation substitute a percentage of the ancestors by new individuals that resulted from the reproduction of members of the previous generation selected by tournament. Certain individuals were subjects of mutations.

Genetic - Generational

Consisted in creating a random initial population and then in each generation substitute the entire population by new individuals that resulted from the reproduction of members of the previous generation selected by tournament. Certain individuals were subjects of mutations. This version of the genetic algorithm obtained better results than the steady state version.

Iterative Local Search

Consisted in starting from a random solution and using local search to find local maximums, then causing a perturbation to those local maximums to try to escape them and find the global one.

Algorithms Implemented

Guided Local Search

Consisted in starting from a random solution and using feature penalties to penalize solutions in local maximums, making the algorithm prefer its neighbours and keep searching the neighbourhood. The penalties are increased according to the maximum utility of a given feature present in the current solution. The features used were the presence of empty caches and requested videos that were not cached. After a number of iterations without evolution, the algorithm returns.

Tabu Search

Consisted in starting from a random solution, searching the neighbourhood and choosing the best candidate solution. Adding that same candidate solution to the tabu list and keep searching its neighbourhood. This prevents the algorithm of revisiting local maximum solutions for a limited amount of iterations. The tabu list size used was static with a size of sqrt(neighbourhoodSize). After a number of iterations without evolution, the algorithm returns.

Experimental Results

	Hill Climbing		
Time taken	0.230232732	0.20704229	0.461781044
Evaluation	5307362	10437053	5042149

	Simulated Annealing		
Time taken	21.456115	21.22019088	21.04697
Evaluation	4782323	9931622	7480122

	Genetic Steady State		
Time taken	3.23255567	2.71971127	2.3912235
Evaluation	10638234	11681866	8136848

	Genetic Generational		
Time taken	5.9884054	5.33854	4.610865
Evaluation	19634400	15910074	17854022

	Iterative Local Search		
Time taken	4.604844728	4.04151	4.437561204
Evaluation	14139288	14321050	10974540

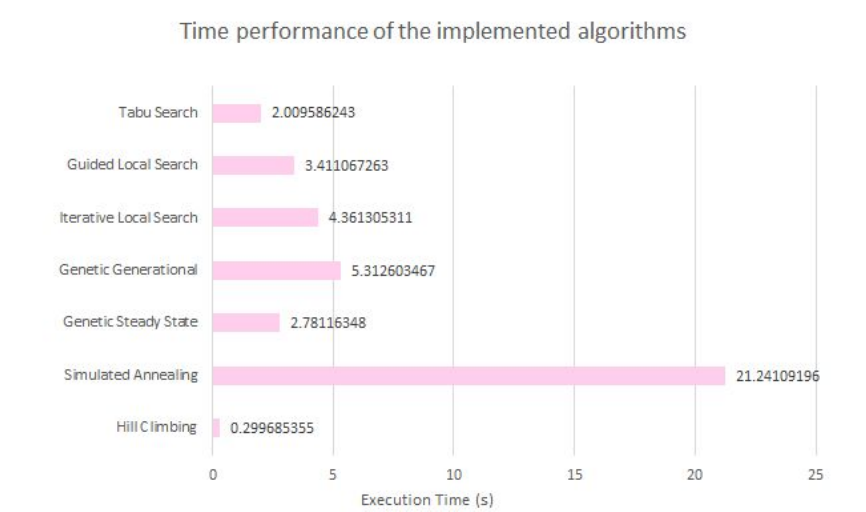
	Guided Local Search		
Time taken	3.83001579	1.876908	4.526278
Evaluation	12757204	9916220	11770786

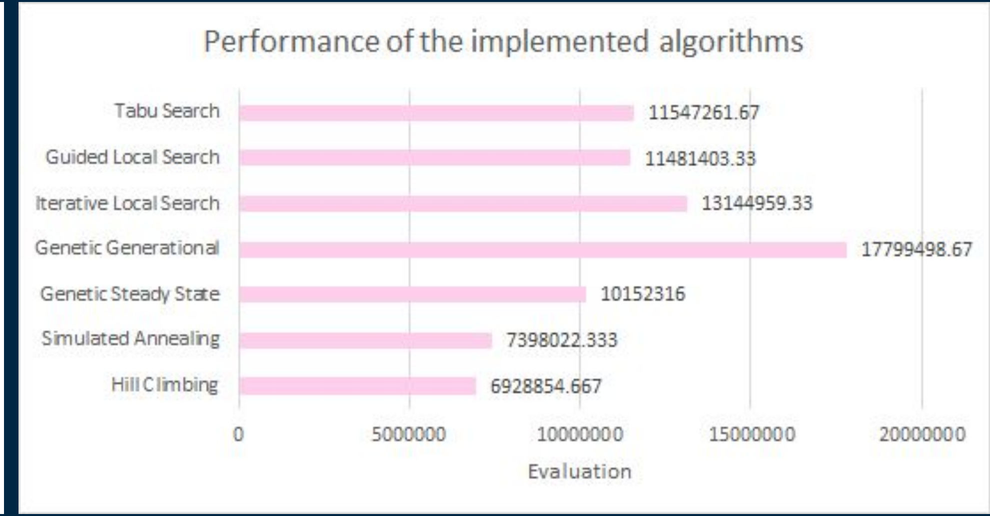
	Tabu Search		
Time taken	2.32651003	1.7068376	1.9954111
Evaluation	12044348	11348580	11248857

10 caches, 10 endpoints,
100 videos, 100 requests

Experimental Results

me_at_the _zoo file





10 caches, 10 endpoints,
100 videos, 100 requests

Conclusions

After developing all these different algorithms, it seems clear that some algorithms have worse performances than others. This is related to the problem characteristics and to the applicability of each algorithm to this sort of problem. During experimentation, the obtained results show a clear tendency for better performance in more complex algorithms, that take into account memory structures to escape local minimums, such as ILS, GLS and Tabu Search.

During implementation and developing of the general data structure, two options were considered: a tree-like structure and an object oriented structure. The latter was chosen, given its accessibility and related code readability.

While developing the evaluation function, the initial implementation was extremely slow and little to not optimized, taking several seconds to evaluate bigger datasets. The optimization measures taken have made it somewhat faster but it could still be further optimized, however it would also mean making major changes in the data structure and algorithms. Currently, as it isn't totally optimized, the algorithms still take several seconds in larger datasets. For example, Tabu Search presents very good and fast results, however, when running the dataset in "vws_small.in" , which contains 54182 requests, after just 21 iterations it provides results like the following:

```
Evaluation Result: 9841781814
Time taken: 6136.0259261 seconds
```

References and Materials

In order to implement these algorithms, there was the need to research about them. Besides the initial research, there was also a focus on searching for specific ways to implement more complex algorithms such as Tabu Search.

Articles:

Google Hashcode 2017 – How we managed to solve the proposed problem

Google Hash Code 2017 in a Knapsack

Google Hash Code 2017: streaming videos

Multiple Knapsacks

Local Search-based heuristics for the multiobjective multidimensional knapsack problem

Tabu Search

Implementations:

Hashcode-video

YouTubeCache

GoogleHashCode

HashCode