ASSIGNMENT 2: GOOGLE HASH CODE

COMP 20230 - Data Structures and Algorithms

Katherine Campbell
16201226

Contents

Introduction

The goal of this project is to create solutions which give the best fitness score. These solutions are presented as grid representations of caches by videos. If a video is in a cache, a one will be at the location [cache number][video number] in the grid. Solutions were found via four separate algorithms: hill-climbing, genetic, randomize, and simulated anneal. This report is broken down into corresponding sections outlining these differing approaches.

Requirements

The first requirement of note was the google hash code which created the dictionary that the following programs I wrote must access. This code was copied into each file because for some reason importing was not working properly.

The crux of the problem as can be gleaned by the goal of the project, is the find_fitness function – the function that we will run to determine the fitness score of each solution. This was the most challenging part of the entire project in my opinion. In writing this, I learned a lot about python dictionaries – I had to access these dictionaries to find the data necessary to calculate some essential components of this function. Luckily, the resources provided, as well as the google hash code slides really helped. The first step in checking the fitness of a solution is to find the values of the variables necessary for the fitness

formula. These, as mentioned before, are accessed via a dictionary. The fileID is key[0] (key referring to the key of the dictionary). This is necessary because it tells us which video we're working with, and thus which column in the grid is being examined. The endpoint is key[1] which is important because the endpoint determines latency (both cache and dc latency) – what is actually being measured when we talk about a solution's fitness.

The tricky part about the find fitness function was finding the individual cache latency because this process entailed identifying the cache that a video was in and then find the latency of that specific cache for that specific video. In order to do this, I started by looking at each key in the dictionary via a for loop. Each key references a specific video. I then made an empty array to store caches. Then I checked the entire length of the grid for ones (the representation of a video in the cache). If there was a 1 in that particular cache for a particular video, I would append the cache number to the empty cache array. I was getting the wrong answers with my test file initially but then I realized that it was only checking the cache once and then setting all other latencies to the DC latency (the latency if no cache is employed). In order to fix this, I put in place a cache_check, initially set to false but if something is appended to the cache array, then the cache_check is set to true and a later written if statement ensures that the cache array is checked if cache_ check is true. I then check that the lowest cache latency is being referenced for I in caches and then get the difference between DC latency and the cache latency. The gains are that difference multiplied by the number of requests for a video, a value shown by the integer value following the key in the dictionary. I then summed all gains for all videos and all requests for all videos and divided gains by requests, which gave a fitness score in milliseconds. Multiplying that product by 1000 gives the fitness score in seconds. I recognize that this was a wordy explanation of a function but again, this function is the keystone for the rest of the project and thus a detailed overview of its construction and makeup is necessary.

Hill-Climbing

Implementation

As the instructions dictated, the next step was to create a function that would make a grid of zeros. This grid would represent a list of all videos and caches but the caches are all empty, thus providing a starting point to finding solutions. The grid is a 2D array with the dimensions of number of caches by number of videos. This function is called make_grid.

Once I get a solution, I must ensure that said solution is feasible, which means that the sum of the size of all the videos in the cache does not exceed the cache size (otherwise, the solution won't work because the cache couldn't hold all the videos proposed). The function check_grid takes in a solution and iterates through, searching for 1s (the representation of a particular video present in a particular cache). Upon finding a 1, the respective video size is appended to an array called memory. Each time a number is appended to this array, the array is summed and compared to the size of that cache. Once the sum exceeds the cache size, the function returns a -1, indicating that the solution isn't feasible.

My approach to the hill-climbing solution was to first take in a solution grid (first all zeros) and then make a copy but then change a singular space in the grid – a [cache][video] to a 1 and find and save its fitness. I then change it back to zero and change the next slot, repeating the same steps. I do that for every slot in the grid. I check the feasibility of these grids. If they are feasible, I save the fitness scores in an array and the coordinate of the changed slot in a different array. If they are not, I do the same except save a 0 instead of the fitness_score. After doing this for the entire grid, I iterate over the solutions and find the

best one and then find the same index in the coordinate array because this is the location of the 1 that provided the best solution. I then change that slot to a 1, and return that newly changed grid. This function was called hill_climb_new_grid.

The next step was to repeat the previous step until finding an optimal solution. The catch here is that it is difficult to know what exactly is an optimal solution. You could spend forever looking for solutions because, depending on how many videos and caches you have, there are many different solutions. We need to balance the need for a good solution with the amount of time we are willing to wait. To address this, I decided to search for solutions to a new grid for a finite number of times, which I set to be 1/3 of the number of slots in the grid (number of caches * number of videos / 3). I do this using the previously detailed hill_climb_new_grid function. I then implemented that in a while loop with a caveat that, if the algorithm was continuously finding improved solutions, I would not stop searching. I set a variable dead_end to false. The loop continues while the number of loops is less than or equal to 1/3 the slots on the grid, or, if that number is exceeded, if dead_end is True. the variable dead_end becomes true only if the solutions being found are not at least +1000 better than the previous best solution. If a found solution's fitness exceeds the previous one, that fitness score and its corresponding grid are saved as variables 'best score" and "best grid". It returns these variables. This function is called climb_the_hills.

Finally, I wrote one function called hill_climb_find_solution which has the only purpose of tying together all previously outlined functions so in the end, only one function needs to be called. This function prints and returns the best fitness score found as well as the best solution grid.

Problems and Possible Improvements

One issue I had encountered was creating a fast way to run this code. The fastest way I could find was selecting a set number of times to hill climb. I recognize that this may limit the quality of the solution (say, for example my code stopped halfway up the hill due to reaching its limit of iterations), but I attempted to address this with the while loop allowing the code to continue should it be finding significantly better solutions.

If time had allowed, I would have somehow liked to work in code that checks if the hill-climbing was not caught in some sort of local maximum, possibly implementing a way to randomly shake up the answers, so to speak thus jumping to a close but not exactly neighboring solution and testing those neighboring solutions to see if it couldn't climb higher. I now think that perhaps a better approach would have been to somehow integrate the random_grid function I have used in the other approaches or perhaps the random neighbor function created in the simulated anneal function (detailed below). This might allow a user to start checking for other solutions once a "best" solution has been found, just to ensure that the solution found is actually the best. I also wish that it were a faster algorithm – I could test the test file and the smallest given input file, but everything else took far too long.

Genetic Algorithm

Implementation

I took a slightly different approach when implementing this algorithm. Here, I also began with a grid of zeros but instead of going through each slot, changing it to a 1, testing the fitness, and saving it if it were thus far the best solution, I opted to take a new approach. Instead I wrote a function that took in two grids (starting with two grids of zeros made with the previously outlined make_grid function) and then iterated over these two grids. I would go through each video in each cache in each grid and, via the

random function, have a 15% chance of turning the 0 into a 1 for each slot. This particular percentage chance was selected by guess and check – I ran the code first with 10%, then increased it to 15%, 20% and 30%. I ran the function 3 times each on the same two files and then opted for the percentage that across the board always yielded the best final fitness score. I am unsure as to the deeper mathematics at work, but empirically it has been proven to be the best percentage to use, at least in this algorithm for my specific code.

The function would then check if adding that particular video to that particular cache were feasible. If so, it goes to the next video in that same cache and repeats the same steps. If it is not feasible, the 1 is changed back into a zero before moving to the next video in that cache. If the video in question is the last video in the list, it will try the first item in the list. This function then returns the newly changed grids. This function is called random_grid. I opted to take this approach because it allows me to insert randomness into my grid making and check for feasibility simultaneously, as opposed to completing a random grid and then checking for feasibility. The latter approach took far too long and thus I abandoned it.

The point of the genetic algorithm is that it takes in two or more feasible solutions and mixes and matches parts of said solutions, gluing these parts together in a solution which is an amalgamation, or child solution, of the original solutions. In order to run this algorithm, I needed to have two or more grids. I opted to make two zero grids, send them through the random_grid function and then add that to an array called population. I do this ten times to make a population of ten grids. This population will serve as the first parent solutions in my genetic algorithm code. The function make_starter_pop completes this task, returning the array of grids called population.

I then want to do the same thing as previously detailed in the hill-climbing section – find the best fitness and the best grid solution that the genetic algorithm produces. I wrote a function that finds the fitness of the original random grids on the off chance that it is one of the random grids and not one of the children grids. This function was called find_best_fitness_orig and returns the population (list of parent solutions), the best score and the best grid of these solutions.

The most important function was the one that made the children grids, aptly called make_children. This function took in the population, best grid, and best score that the previously detailed find_best_fitness_orig function returned. I approached the genetic algorithm in that I wanted the parent grids to swap caches so for example, grid 1 would replace its cache 5 with its neighbor's cache 5, thus creating a new grid with its same caches except for cache 5 which was from a different grid. I wanted the cache and number of caches swapped to be random so I created a variable cache_swap which, using the random.randomint function was set to a number between 1 (because I wanted at least 1 cache swapped) and number of caches minus one (because if you swapped all caches it would just be the neighbor grid and not a new grid at all).

Nested for loops were made going through each cache in each grid of the population. The inner loop goes in range from zero to the random cache_swap number plus one. The function takes the caches in that range from each grid and gives them to the previous grid in the grid population list. With another try and except IndexError, I account for if the grid is first in the list, in which case I give its caches in the range to the last grid in the population. I then find the fitness of all these new grids and, if the fitness is better than the previously stored best score from the original population, the function stores this new fitness and its

corresponding grid. This function then returns the population of new grids, the best fitness score, and the best grid.

Finally, I wrote a function genetics_at_work which combines all previously outlined functions, establishes a clamp of sorts to ensure that children are not made infinitely, and incorporates the children grids into the population with every loop of the make_children function. The clamp was ensured via a while loop with a dead_end = False variable which switches to true if the new best score does not exceed the old best score by more than ten, much like what was used in the hill-climbing algorithm, but smaller because the solutions found in this algorithm had smaller differentiation. However, I established a base number of loops that it must go through, regardless of score yielded – I decided on five as a simple round number. Initially I wanted to set the minimum number of loops like I did in the hill-climbing algorithm to 1/3 of the slots in the solution grids. However, this took far too long so a single digit number was selected. This function prints and returns the best fitness score found.

Problems and Possible Improvements

The main issue I ran into when writing this program was with random_grid, when checking if adding that particular video to that particular cache were feasible. If so, it goes to the next video in that same cache and repeats the same steps. If it is not feasible, the 1 is changed back into a zero before moving to the next video in that cache. If the video in question is the last video in the list, it will try the first item in the list. I had some problems implementing this in the code because index errors were being thrown – the code recognized it was at the end of the list and I needed somehow to tell it to go back to the first item if it were out of index. After some research, I achieved this with a try….except IndexError. If the indexerror appeared, it would check the front of the list, not the following item in the list (which is None). This worked perfectly. I implemented this in the rest of my programs when I wanted to loop back to the front of a list.

The only problem with the way I approached the genetics algorithm with selecting and switching caches is that it always started the cache replacement from cache number zero so the final cache will never be swapped (because the number of caches swapped and thus the endpoint of the for loop was never equivalent to the total number of caches to ensure that grids did not take all caches from their neighbor, merely replicating an already existing grid). With more time, I could have tackled this problem, perhaps implementing a code something like how I included the last grid in the list by having it take caches from the first grid in the list.

Additionally, I think I could have gotten a better solution from my genetic algorithm if I were to not only swap caches between subsequent grids in rows but someone have the grids switch random caches with random grids in the list. I think this could be achieved with random.randint where the range would be 0 to number of grids in the population. However, I do not have time to run and test now, but I will definitely do so and see if it helps to find better solutions.

Randomization

Implementation

The approach I took to this code was exactly what the name implies – inject as much randomness as I could. This was achieved mostly by using the same random_grid function as described in the above section but instead of a 15% chance of a 0 becoming a one, I also made it a 15% chance that a one would become a zero. The rest of the function remained the same.

I then wanted to test the fitness of each of these grids, because I want to test all random grids created. This function iterated through a given list of grids given and would save and return the best score and the best grid.

I then wrote a randomize_grids function which tied the previously mentioned functions together. It started with an array of 50 random grids which were originally a list of 50 zero grids which were then put through the random_grid function. Then through a while loop it loops through these 50 random solutions, checks for fitness scores of each one and saves the best solution and score. Then on each loop it sends the now processed list of random solutions back through the random_grid function and then uses those new random solutions in its next loop. It repeats this process at a minimum of 5 times and may continue if the algorithm is finding significantly improved solutions (this time best_score + 100). The best score and best solution grid found are then printed and returned.

Problems and Possible Improvements

An interesting problem I came across. I am unsure as to why import function worked for some methods (specifically make_zero_grid and sometimes random_grid although because that function changed between algorithms, I didn't end up using the import) but wouldn't work for the find_fitness function. Copying and pasting the function worked perfectly but importing it would always result in an indexing error in the find_fitness function in the randomization file. Again, unsure as to why this would be, but I provided an easy work around.

Along those same lines, the randomize.py file would not import the random_grid from genetics, so I copied and pasted it. Upon running it, it threw an index out of range error for number of videos. Seeing that the code was exactly the same line for line as the functioning code in the genetics.py file, I just copied and pasted again, once again changed the variable names to suit the randomization file, and it worked. I've had to do this the last three times I have exited and then restarted Eclipse, but a copy and paste over fixes it every time as long as you don't exit again. I am baffled, but luckily it is an easy fix. Additionally, every time the randomize code runs, it also prints the genetic code solution. I do not know why this should be (no print statements were found). Other than these oddities, the code works, and I'm really happy with my work.

Simulated Annealing

Implementation

Here we take a random grid (found by using a slightly modified version of the random_grid function previously outlined which takes in and changes a single grid as opposed to a list of grids). This function was called random_solution. We then find a random neighbor of that grid. A neighbor in this case refers to a grid that is identical to the current grid apart from a single slot (where it will have a one where the original will have a zero or vice versa). This was done via a new function, the make_neighbor function which takes in the grid created and returned by random_solution and selects and random cache and random video number using random.randint in the range zero to number of caches or number of videos minus 1. This function plus those numbers as indexes in the grid, finds the slot at that spot and switches whatever value is there. This is now a neighboring grid. This grid is returned.

The last component before being able to put together the full simulated_annealing function is the acceptance probability. This is found by taking the difference between the new score (the fitness of the neighboring solution) and the first score (the fitness of the original solution) and dividing that difference

by the temperature. The temperature is found by taking 1 times the alpha (a number between .8 and 1). This quotient is then multiplied by the constant 2.71828. This yields the acceptance_probability and occurs in a function called acceptance_prob.

These parts must be glued together through a function called anneal. This function takes in the original random solution. It finds the fitness of that solution, and searches for a random neighbor to see if it has a better solution. The neighbor is not rejected or accepted outright. Instead it finds the acceptance probability and as long as that value is greater than a random number between 0 and 1, THEN the neighbor is accepted. This value will always be greater than a random number between 0 and 1 if the neighbor solution is automatically better. But if it is worse, this algorithm ensures that the worse solution isn't rejected outright, it only MIGHT be rejected (depending on the random number selected). This algorithm is an attempt to prevent getting stuck in a local maximum (if we sometimes move to a worse neighbor solution, there is a chance that the next time the code is implemented, the solution may improve – when climbing hills, you sometimes must go down to go up). The best solution and score is returned.

For simplicity's sake, I created a function called find_solution which calls all appropriate functions in order, starting with making the zero grids and ending with the anneal(random_grid). This function prints and returns a solution.

Problems and Possible Improvements

For this one, I heavily based my code upon the code in the blog post that was linked to us, although I did do my own research to ensure I understood what was happening beneath the hood. This ended up being important because the example code was searching for the smallest score whereas we were looking for the largest score. I discovered this when implementing the acceptance_probability function because when I followed the code in the blog exactly (old score – new score instead of vice versa), and running many print statements, I realized that the new_neighbor was being rejected much more often than it should have been. It was being rejected even when the scores were better. After breaking down the mathematics and printing each variable at each loop, I realized it was because the code was looking to accept the smaller score, not the larger one. This becomes important when whether to accept or reject the new solution hinges on an acceptance probability being larger than a random integer between 0 and 1. If the score is negative (which it may be if the old score is less than the new score) then the new score would be rejected. Switching the acceptance probability function so that it was the new score subtracted by the old score made all the difference. Understanding the mathematics behind the algorithm was essential to correctly writing this code because I needed to know where in the formula to account for this difference in project goals. I hope I correctly figured it out.

Comments about the Project

I found the simulated anneal part of the project to be the most difficult because it was a totally new concept (had never heard of it before) and also because I had to do a lot of my own research to get my head around it. I enjoyed learning about it though, because I think it is an interesting way to approach the problem of local maxima. I appreciate the mathematics behind such a solution.

As one can see from the findings illustrated below that the simulated anneal program seems to yield the best results and it does so very quickly.

I really enjoyed working on this project because it reflects a real-world problem. Additionally, I feel a lot more comfortable approaching a coding problem and a lot more confident in developing solutions. My

skills in utilizing dictionaries in python has vastly improved, I am comfortable and now better understand the use of try and except, and I have no hesitation in implementing and using arrays – I didn't have to look up how to work with them even once – a vast improvement from the beginning of this semesters. I also have definitely improved in taking a solution I have created and optimizing it. Finally, testing and debugging my code is much easier and goes much faster. I hope to improve this even more with experience.

Findings

The below findings are form the two smallest files. This is because the larger files took too long to run and my antique laptop was looking like it was going to up and die. It would be interesting to see what these programs might find, should the computing power be available.

Test input

Best score found with hill-climbing was: `562500.0`

Best score found with genetics was: `562500.0`

Best score found with randomization was: `650000.0`

Best score found with simulated annealing was: 675000.0

me_at_the_zoo

The best score found by hill-climbing was: 516740.5483269181

The best score found by genetics was: 467571.6343542636

The best score found by randomization was: 532661.4636219803

The best score found by simulated annealing was: 557191.7727347864