EVOLUTIONARY PC

by

Thuong Hoang

Submitted in partial fulfillment of the

requirements for Departmental Honors in

the Department of Computer Science

Texas Christian University

Fort Worth, Texas

May 8, 2023

EVOLUTIONARY PC

Project Approved:

Supervising Professor: Michael Scherger, Ph.D.

Department of Computer Science

Bo Mei, Ph.D.

Department of Computer Science

Wendi Sierra, Ph.D.

Department of Honors College

ABSTRACT

Building a computer from common off-the-shelf components is a perplexing task. There are supply chain issues, compatibility issues, and budgetary constraints. This research investigates the use of an evolutionary algorithm to find the best possible components for a computer system within a designated budget. The algorithm starts with a set of parent combinations of builds and then creates a set of offspring. From the offspring set, they are mutated periodically and only the most compatible builds are kept for the next generation. This technique generates the best possible combination of components within the budget constraint in a finite number of generations. The application uses data from current component manufacturers, and a web interface has been created for ease of use.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 BACKGROUND

Technology has had a rapid pace of advancements over the last century with computers rapidly becoming a pivotal part of modern-day society. As technology advances with the growing needs of consumers, the research field of computer system optimization begins to form and grow. The term computer system optimization refers to the improvements that can be made to a computer system for utmost performance and efficiency. These improvements can range from fixing issues and system bugs, reducing the consumption of resources, and increasing the overall speed and response time of the system.

Computer optimization became a popular field amongst computer scientists or even hardware engineers because consumers need high performing computer systems. Optimizing a computer system has advantages such as minimizing energy usage, better productivity, and a small chance of system crashes. This becomes advantageous for businesses or users who need high performing computers for video editing, personal tasks, or gaming. While optimizing computer performance is an important task to achieve, it becomes difficult with the several ways to improve the performance of a computer system.

Changing the hardware of a computer system is the easiest way to improve the performance and efficiency of the system. However, selecting hardware components can be a challenging task, especially for a non-specialist. Hence, improving a computer system becomes a tedious task with all the compatibility issues and budget restrictions. Therefore, most consumers typically go for selecting computer components from the components market to build a computer rather than purchasing components to optimize a manufacturer’s computer system or a current computer system.

1.2 RESEARCH PROPOSAL

This research proposes an algorithm to aid users in selecting the most crucial components with significant impact on a computer system's performance, which are the CPU[1], GPU[2], and RAM[3]. This type of problem is known as an optimization problem. With the current market for components available to consumers, there are billions of combinations of computer components that can be used to build a functioning computer system. The limitations of budgets and compatibility between components become an issue for non-specialists selecting components for the first time.

While there are guides available to be used as resources for selecting components, the efforts of remaining in budget while retaining the utmost performance specifications for the budget limit is a crucial task. This research will look at an optimization algorithm that will incorporate a user's wants or needs within a computer system to generate a list of the three major components of a computer system.

2 ANALYSIS

2.1 OPTIMIZATION PROBLEM

Optimization is a problem that occurs when given a set of data with a function that finds a maximum or minimum. This becomes a perplexing task with the variety of algorithms that have the capability to provide a solution. There are two distinct types of optimization problems. The first type involves a function where a derivative can be found. This type of problem can be solved by the following types of algorithms: Bracketing, Local Descent, First-Order, and Second-Order (Brownlee, 2019). The second type involves a function where a derivative cannot be calculated at any given point. This can be solved by the following types of algorithms: Direct, Stochastic, and Population (Brownlee, 2019).

2.2 ALGORITHM

Generating components for a computer system with the utmost performance does not involve a function with a derivative. Each component has a performance benchmark, which the overall performance can be found by summing the performance benchmark of all components in the system. Note that the CPU, GPU, and RAM will each possess a collection of available and compatible items that can be used within a computer system. Hence, there is a set population that can be used to find the global maxima. In this circumstance with the given constraints, the population kind of algorithm would be best fit.

2.1.1 Population

Computer components have different purposes and can be used for several types of devices. In this case, a desktop computer is the only type of device with customizable components. Hence, a desktop computer can be built by the end-user with all components selected by the end-user. Within the initial population, the components that can be used within a desktop are the only components within the initial population.

2.1.2 Fitness

The fitness of a computer system consists of the performance of each individual component that composes that system. The benchmark of a computer specifies the performance of the system, which is defined through a series of tests that stress the computer system. Typically, a higher benchmark indicates a better performing computer system. To determine the best possible fit computer with the initial population, the benchmark can be taken of each individual component and summed together to get an overall benchmark. The highest benchmark will be considered the best optimized computer system.

3 **RESEARCH OBJECTIVES**

The research in this study intends to find the optimal CPU, GPU, and RAM of a computer system. To do this, the study evaluates an evolutionary algorithm. This algorithm is developed to find the optimal combination of the CPU, GPU, and RAM. The algorithm will be measured in effectiveness in selecting the optimal combination with the initial population.

A custom computer may not have an infinite budget. Hence, there are various constraints that must be applied to the algorithm. There are constraints such as limiting the spending of the CPU, GPU, and RAM and the branding preference of the CPU and GPU. This research will further analyze how the algorithm can select components that fit the constraints along with having the optimal performance within those constraints.

4 IMPLEMENTATION

4.1 EVOLUTIONARY ALGORITHM

According to Eiben and Smith (2015), the theory behind this algorithm is to have a population of individuals that are within an environment where they would have to compete for limited resources. The competition between the individuals in the population gives the algorithm a survival of the fittest aspect, which is typically given by a function. With this function, a randomized set of candidate solutions is generated. Once the fitness values are assigned to the individuals, the most fit individuals will be selected to move on to the next generation. These fit individuals will be considered the parents of that generation. The parents will generate offspring, typically one or more new solution candidates. The offspring may go through mutation, which does not always happen, and a new candidate is formed. This process is repeated until an end condition, or a candidate has sufficient qualities is found. This flow is found in figure 1.

ALGORITHM 1 EVOLUTIONARY ALGORITHM

```
INITIALIZE population;
SELECT parents;
REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO
    GENERATE offspring;
    MUTATE offspring;
    EVALUATE new candidates;
    SELECT new parents for next generation;
END
```

Figure 1 Pseudocode of Evolutionary Algorithm

With the pseudocode as a reference, the algorithm can be implemented. The following figure shows the full implementation of the algorithm in this study.

ALGORITHM 2 FULL EVOLUTIONARY ALGORITHM IMPLEMENTATION

Input: N/A

Output: solution to the optimization problem

```
def EvolutionaryAlgorithm ():
    population = InitializePopulation()
    parents = SelectParents()
    for jj in range (1000):
        parents = GenerateOffspring(parents)
        parents = Mutate(parents)
    solution = parents[0]
    for parent in parents:
        if CalculateFitness(parent) >
        CalculateFitness(solution):
            solution = parent
    return solution
```

Figure 2 Implementation of evolutionary algorithm

In figure two, the implementation of the algorithm is very similar to the pseudocode provided in figure one. The methods being used within the algorithm will be explored further on in the next few sections. This implementation could be done through the examples provided by Kübler (2020) and Ippolito (2020).

4.1.1 Population Initialization

To begin solving this problem, the creation of an initial population must be done. In this case, the population should consist of the CPU, GPU, and RAM that are available for purchase. Once the data is retrieved for each component, a list of permutations will be created. The permutations are generated in a random manner. Each time the algorithm runs, the list will vary.

ALGORITHM 3 POPULATION INITIALIZATION

```
def InitializePopulation ():
    cpus = CPUSerializer (CPU.objects.all(),
    many=True).data
    gpus = GPUSerializer (GPU.objects.all(),
    many=True).data
    rams = RAMSerializer (RAM.objects.all(),
    many=True).data
    population = []
    max_list = max(len(cpus), len(gpus), len(rams))
    ii, jj, kk = 0
    if max_list == len(cpus):
        for ii in range(len(cpus)):
            if jj == len(gpus):
                jj = 0
            if kk == len(rams):
                kk = 0
            population.append([cpus[ii], gpus[jj],
            rams[kk]])
            jj, kk += 1
    elif max_list == len(gpus):
        for ii in range(len(gpus)):
            if jj == len(cpus):
                jj = 0
            if kk == len(rams):
                kk = 0
```

```
22              population.append([cpus[jj], gpus[ii],
                rams[kk]])
23              jj, kk += 1
24      else:
25          for ii in range(len(rams)):
26              if jj == len(cpus):
27                  jj = 0
28              if kk == len(gpus):
29                  kk = 0
30              population.append([cpus[jj], gpus[kk],
                rams[ii]])
31              jj, kk += 1
```

Figure 3 Population Initialization Implementation

Figure three shows the implementation of initializing the population used in this study. The first step is to get the data on all the components, which is done in lines one to three. The serializer is a function that gets all the data from the database. Next, permutations need to be created. One problem that may occur from trying to generate the permutations is that there is a varying number of CPU, GPU, and RAM. Hence, there may not be enough of one component in the component's list to be matched with another component. Line five finds the component list that has the most available parts. Next, lines seven to thirty check which list has the most components. A loop is created to go through the longest list while going through the shorter list and repeating and adding the combination of components to the population. The shorter list will loop back to the first element once it reaches the last element.

The population is completely random and consists of a wide variety of components from each component category. There are different brands, processing speeds, and component generations that are in the initial population pool. Having this variety allows the algorithm to explore many different possibilities that could have many different genes (Soni, 2018).

4.1.2 Parent Selection

This step presumes that the initial population has been created and has a wide variety of components. The members, elements in the permutation list, in the initial population need to be selected as the parents of the first generation. Since this algorithm is survival of the fittest, the parents should be the most fit individuals from the initial population. This set of parents will then be the nth generation of parents. To determine which individuals are selected, the members are sent through a fitness evaluation[4]. The number of parents being selected is a parameter that is customizable and may vary with each implementation.

ALGORITHM 4 PARENT SELECTION

Input: initial population

Output: top 10 most fit members

```
def SelectParents(population):
    parents = []
    for member in population:
        if len(parents) < 10:
            parents.append(group)
        else:
            for ii in range (10):
                if CalculateFitness(parents[ii]) <
                CalculateFitness(group):
                    parents[ii] = group
                    break
    return parents
```

Figure 4 Parent Selection Implementation

Figure four is the implementation used in the research to select the parents. In this implementation, population is passed as an input, which is a matrix that was created in the population initialization step. The first loop looks at each member in the initial population and selects the best fit members. In the loop within the else statement, it checks each parent and ensures that the best fit individual is within the parent list. In this research, there were ten parents that were selected. The number of parents parameter was selected as ten to help improve the algorithm's speed and accuracy. Although the increased number of parents causes the space complexity to increase, the perk of having an accurate algorithm is worth the space.

4.1.3 Generate Offspring

Once the parents have been selected, the generation needs to proceed after the offspring is created. This is typically done through combinations and having the offspring have genes of the parents. The offspring will also experience mutations[5], crossovers, or both.

ALGORITHM 5 GENERATE OFFSPRING

Input: Selected Parents

Output: Offspring

```
def GenerateOffspring(parents):
    cpus, gpus, rams = []
    for parent in parents:
        cpus.append(parent[0])
        gpus.append(parent[1])
        rams.append(parent[2])
    children = list(itertools.product(cpus, gpus, rams))
    return children
```

Figure 5 Generate Offspring Implementation

Figure five shows the implementation of the generating offspring used in this study. The input in this implementation is the parents, which is a matrix that contains that most fit individuals from the initial population. The loop retrieves all the possible components from the parents. Once all components have been retrieved, itertools[6] is used to create a combination of all the components for the offspring.

4.1.4 Mutation

Mutation is the step that alters the components in a way to get closer to what the solution would be. It assists in widening the search space from the selection of parents in the beginning. There are a variety of different mutation methods that can be implemented. These methods include fitness proportionate selection, rank base selection, and tournament selection (Ippolito, 2020).

According to Ippolito (2020), the fitness proportionate selection method is done by creating a wheel that is used to categorize each member's fitness based on its fitness relative to other individuals. This method is not ideal when an individual's fitness is significantly higher than the other individuals. Hence, the rank base selection method can help fix the problem. This method is similar to the fitness proportionate selection method except each individual is given a rank to help distribute the wheel more evenly. These methods of mutation would be difficult to implement with the current problem. There would be millions of possibilities that could be the solution, and the previous methods would not be ideal with a big data set. Hence, this research implements the tournament selection as the mutation method.

According to Ippolito (2020), tournament selection is the method where N individuals are selected from the population. From the N individuals, a chosen element with better performance is selected to replace one of the current genes in the solution pool. Once the offspring has been created, this method of mutation is applied to the offspring. The offspring will be considered the new set of population.

ALGORITHM 6 MUTATE

Input: Offspring population, population

Output: Offspring population with mutation

```
def Mutate (offspring, population):
    parent_1_ran = random.randint(0, len(offspring) - 1)
    cpu_ran = random.randint(0, len(CPUs) - 1)
    parent_2_ran = random.randint(0, len(offspring) - 1)
    while parent_2_ran == parent_1_ran:
        parent_2_ran == random.randint(0, len(offspring) - 1)
    while float(offspring[parent_1_ran][0][‘benchmark’]) >
    float(population[cpu_ran][0][‘benchmark’]):
        cpu_ran = random.randint(0, len(population) - 1)
    gpu_ran = random.randint(0, len(population) - 1)
    parent_3_ran = random.randint(0, len(offspring) - 1)
    while parent_3_ran == parent_2_ran or parent_3_ran == parent_1_ran:
        parent_3_ran = random.randint(0, len(offspring) - 1)
    while float(offspring[parent_2_ran][1][‘benchmark’]) >
    float(population[gpu_ran][1][‘benchmark’]):
        gpu_ran = random.randint(0, len(GPUs) - 1)
    ram_ran = random.randint(0, len(RAMs) - 1)
    while float(offspring[parent_3_ran][2][‘benchmark’]) >
    float(population[ram_ran][2][‘benchmark’]):
        ram_ran = random.randint(0, len(RAMs) - 1)
    list(offspring)[parent_1_ran][0] = population[cpu_ran][0]
    list(offspring)[parent_2_ran][1] = population[gpu_ran][1]
    list(offspring)[parent_3_ran][2] = population[ram_ran][2]
    return offspring
```

Figure 6 Mutation Implementation

Figure six shows the mutation implementation used in this study. The parameter offspring is an array obtained from the generate offspring function previously mentioned[7]. Population is the parameter that contains all the possible CPU, GPU, and RAM. It is a matrix that contains the names of each component. The python module random[8] is used to generate a number between a given inclusive range. The variables parent_1_ran, parent_2_ran, and parent_3_ran are integers that will be used to select a random offspring. To ensure this mutation is efficient, the while loops on lines five and eleven are used to make sure the random numbers generated for each variable do not match. This allows the algorithm to select three different offspring from the offspring pool. The variables cpu_ran, gpu_ran, and ram_ran is used to get a random component from their corresponding list. The while loop on lines seven, thirteen, and sixteen are used to ensure that the new component being added will be a component of higher fitness. At the end of the function, the offspring are replaced by the corresponding component.

This type of implementation of mutation allows the algorithm to not get stuck on a local extremum. The random library ensures that the mutation occurs at a random rate and the same offspring is not selected each time the algorithm runs. It truly ensures that the algorithm maintains its nature of survival of the fittest.

4.1.5 Fitness Evaluation

Fitness evaluation is typically done through a fitness function. A fitness function takes the characteristics of each member and gives a numerical representation of that member, which shows how viable of a solution it is (Soni, 2018). The fitness function aids in the selection process. When selecting the best fit members in the population or in the offspring population, it is difficult to decide with just the component names or just an array. Therefore, the fitness function is used to help select the best fit components because it returns a number which is simpler to rank amongst the other individuals.

ALGORITHM 7 FITNESS EVALUATION

Input: member

Output: fitness of the member

```
def CalculateFitness (member):
    cpu_fitness = float(member[0][‘benchmark’])
    gpu_fitness = float(member[1][‘benchmark’])
    ram_fitness = float(member[2][‘benchmark’])
    return cpu_fitness + gpu_fitness + ram_fitness
```

Figure 7 Fitness Evaluation Implementation

Figure seven shows the fitness evaluation function that was used in this study. The parameter member is an array of components consisting of the CPU, GPU, and RAM. Each component has data such as the name, benchmark, and price. The function converts the benchmark of each component into a float and sums up all the benchmarks to give the member a fitness value.

Benchmark is a score given to a computer based on the computer’s performance. This is usually done through a series of tests that a computer system must undergo. Since this score is fair amongst every computer system, the fitness value being based on the benchmark of each component is the best method in evaluating how well that specific member can perform in comparison to other members of the population.

4.1.6 Solution

Typically, when an algorithm ends, it returns a solution to the user. In this case, the evolutionary algorithm can end under two circumstances. The first circumstance is that there exists a solution in the search pool that fits the criteria of the kind of solution the developer is looking for. This can occur when the threshold of performance has been reached. The other circumstance that this algorithm can end is that the run time has reached its limit. For a solution to be considered a probable solution, it must be given in a reasonable time.

For this research, performance of a computer for a given set of constraints becomes difficult to determine. Hence, the solution reaching a performance threshold becomes a difficult circumstance for the algorithm to end. For this reason, the algorithm must end after N number of iterations. The algorithm must undergo numerous executions to determine the number of iterations that will give a solution. After these trials and execution of the algorithm, the algorithm seems to return a probable solution after one-thousand iterations.

4.2 USER INTERFACE

The algorithm, as implemented above, only has a command line. A user-friendly interface is needed to interact with the algorithm. The algorithm is used to process big data, which is why Python was the preferred programming language for this study. Python has built-in graphical user interface libraries that can be used to create the interface. The available libraries in Python are not modern, and the interface can only be used after the python code is executed through the command line. Hence, the interface solution used in this study is a web application.

This web application will be built using different frameworks and libraries in multiple programming languages. A web application allows users to click on a domain name and see the program run without having the source code on their personal computers. The interface will also have a more modern interface that is responsive to any screen. When creating a web application, it must have a frontend and backend.

4.2.1 Front-end Interface

Front-end is a term used to describe the part of the web application that is for the users to see and interact with. It is typically designed by a graphic designer, and it is visually appealing, functional, and easy to use. This part of the website is typically developed using technology such as HTML, CSS, JavaScript, or TypeScript. The interface is usually responsive and accessible to provide an easy and seamless experience for all users across multiple devices and browsers. There are many front-end libraries created for developers to have ease in developing websites. Vue, React, and Tailwind are some of the most popular libraries used. In this study, React is the library that will be used to implement the front-end interface of the web application. Since the algorithm is developed using Python, React is the best library to connect the front-end interface with the solution that the algorithm will provide.

React is a JavaScript library that is maintained by Facebook, and it allows developers to create user interface components that are reusable and can update efficiently in response to changes made to the application (React, n.d.). It has a component-based architecture, where components are organized into a structure that resembles a tree. Each component has its own states and methods. React is one of the best at rendering all its components in a timely manner (React, n.d.). It has a wide variety of libraries or frameworks that can be used to better the user experience. React has a modern look, is interactive, and responsive to any user interface. Hence, React was chosen as the front-end library.

To build a web application with React, there must be components that are created by the developer to start the web application. These components can be functioning components that are on specific pages or the pages themselves. For the interface created in this study, it will have typical pages of other web applications such as the home page, about page, and frequently asked questions page. These pages will be static meaning the contents will not change per user. Other pages that will be needed for this project will include a build customization page, registration page, log in page, and an account dashboard page.

Once the pages are completed, the website needs functionality. These functionalities include account creation, build creation, email verification, password change, forgot password, change username, and modifying current builds. For these functionalities to work, a back-end interface needs to be created along with a database. This will allow the front-end to finally have functional components.

4.2.2 Back-end Interface

Back-end is the interface that is not interacted directly by the users. Rather, it is a way for users to access their data or manage their data. Back-end is known to be the server-side part of the development of a web application. It is used to give the application functionality. The back-end includes a variety of APIs[8]. The APIs are created to handle the logic for managing data, user authentication, and other operations to fit the user's needs.

There are various back-end frameworks that can be used to create a fully functional web application such as Express.js, Django, and Ruby on Rails. Since the algorithm was written in Python, Django would be the only probable back-end library that can be used with a React front-end.

Django is a framework written in Python that is typically used as a back-end for React web interfaces. It has a model-view-controller architecture with various tool sets and libraries for a dynamic database-driven website (Django, n.d.). Django, by default, uses a version of SQL as its database engine. This allows for relational database management. Django also has other features such as built-in admin interface, templates for web access, and user authentication and authorization. This framework has many security features such as protection against cross-site scripting attacks, cross-site request forgery protection, and secure password handling (Django, n.d.).

For Django to function, models need to be created. Models are the entities that lie in the database. Typically, they include data fields such as ID. Once the database is created, views need to be created. Views are the python functions that handle all API requests, which includes logic and database management. These views will return a status of whether the specific action was successful or not. Once all views are finished, user management is implemented. This includes user authentication, user authorization, user creation, and user deletion. Django is compatible with many different user management libraries that can be used to aid in the process of implementing this feature of the website.

4.3 DATABASE SCHEMA

With each web application, there needs to be a database that is created to manage data. Databases can have a variety of engines, with some engines having better performance than others. The database engine used in this study is SQLite which is the default engine used by the Django framework.

SQLite is an engine that mimics SQL; however, their difference is that SQLite can run serverless. This engine is ideal for small-scale projects, and projects with the need of a local database. It is created from a C library, which makes the engine fast, reliable, and full of features of a database server. This database engine is one of the most used engines worldwide where it is built into many popular applications. According to the SQLlite homepage, there are over one trillion active SQLite databases.

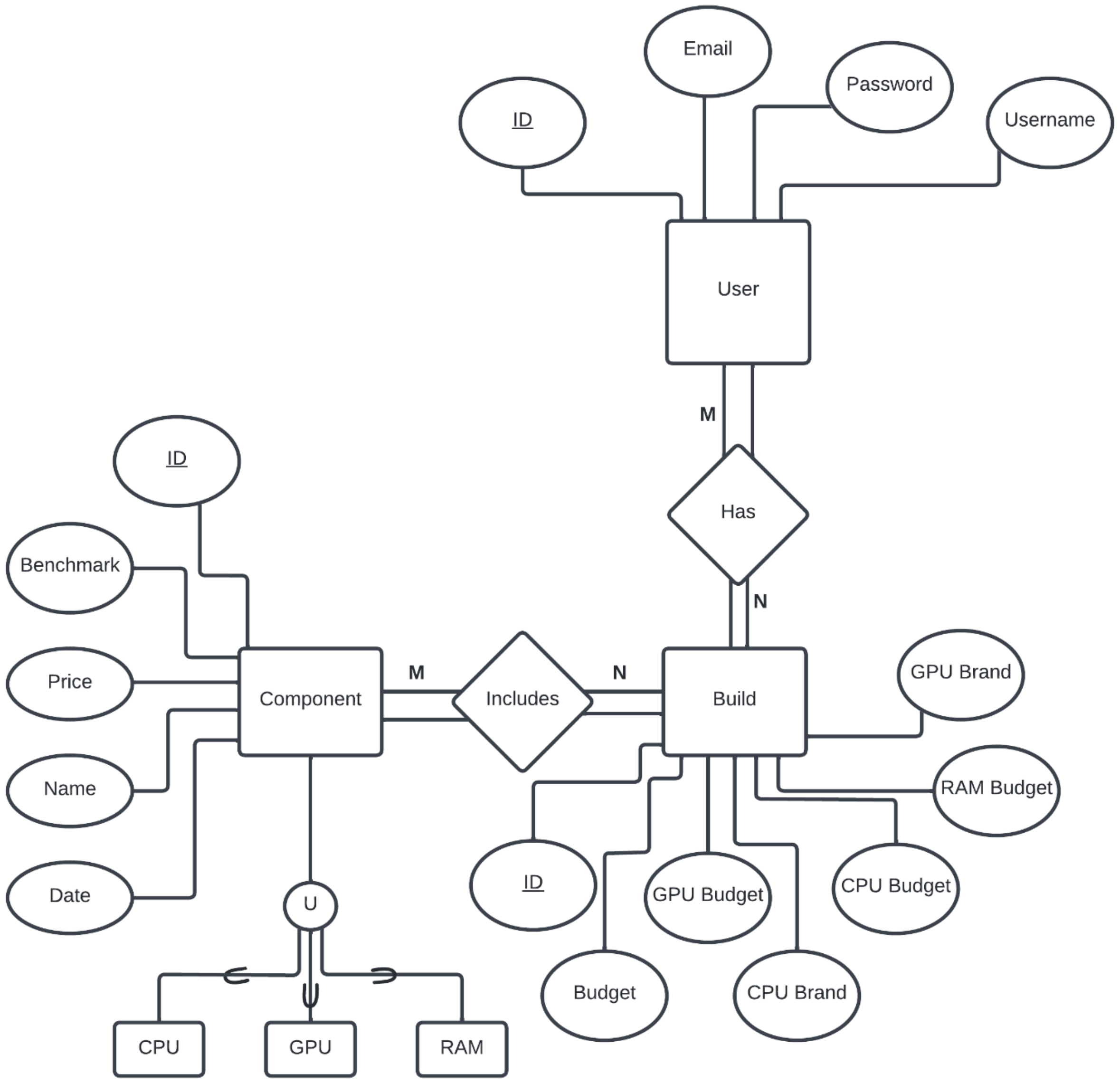


Figure 8 Database Schema

Figure eight shows the database schema created to manage users and their builds along with the components that are used in each build. The schema shows the basic components of the database that was created in this research. There are other parts of the database not shown because it was created by the Django framework to help with authentication and authorization. In this schema, there are three major entities. The component, build, and user entities each have one relation. User has a relation with build, and Build has a relation with component. Users can have multiple builds, and Builds can only include one of each type of component.

The user_build relation to connect a build with a user. This relation contains the user ID and build ID, where both are foreign keys that reference their respective table entries. This relation is created to allow users to have multiple builds, and builds can have multiple users. This relation aids in the idea that users may have the same build constraints, which there is no need to create duplicate builds in the database. This indicates a many-to-many relation. With each build, there is a many-to-many relation between the build to the components. This relation is necessary because there are builds that may reuse the components that belong to another build. To allow this relation there are three tables created to connect the build with the component that belongs to that build. There are three relations that show this connection. The build_ram, build_cpu, and build_gpu connect the component to its build. Each relation has a build ID and the component ID.

Each component has its own table in the database. However, they are all formatted the same way. Each component will have information about its name, price, benchmark, and date that it was found. The name and price are used to display to users when the build is selected. The benchmark and price are used with the algorithm to help find the best fit components for that specific build criteria. The date field is used to determine if the database needs to be updated with new components that may have entered the market.

This database is used to help with the functionality of the web application. It keeps track of all the user accounts created along with each user's build. The database also keeps information about each component that is available in the market for users to purchase. The relationships between each entity in the database are necessary due to the nature of many users having many different builds and each build can have similar components to one another.

5 RESULTS

5.1 OPTIMIZATION PROCESS

Optimization for an algorithm like this becomes quite difficult to do. There are many parameters such as the number of generations and the number of parents that can be modified to help improve the algorithm's performance. To fully ensure that the algorithm would give a correct solution each time it ran, the numbers had to be changed and tested, and the solution had to be verified. The following figure shows the performance of the solution versus the number of parents it took to generate the solution.

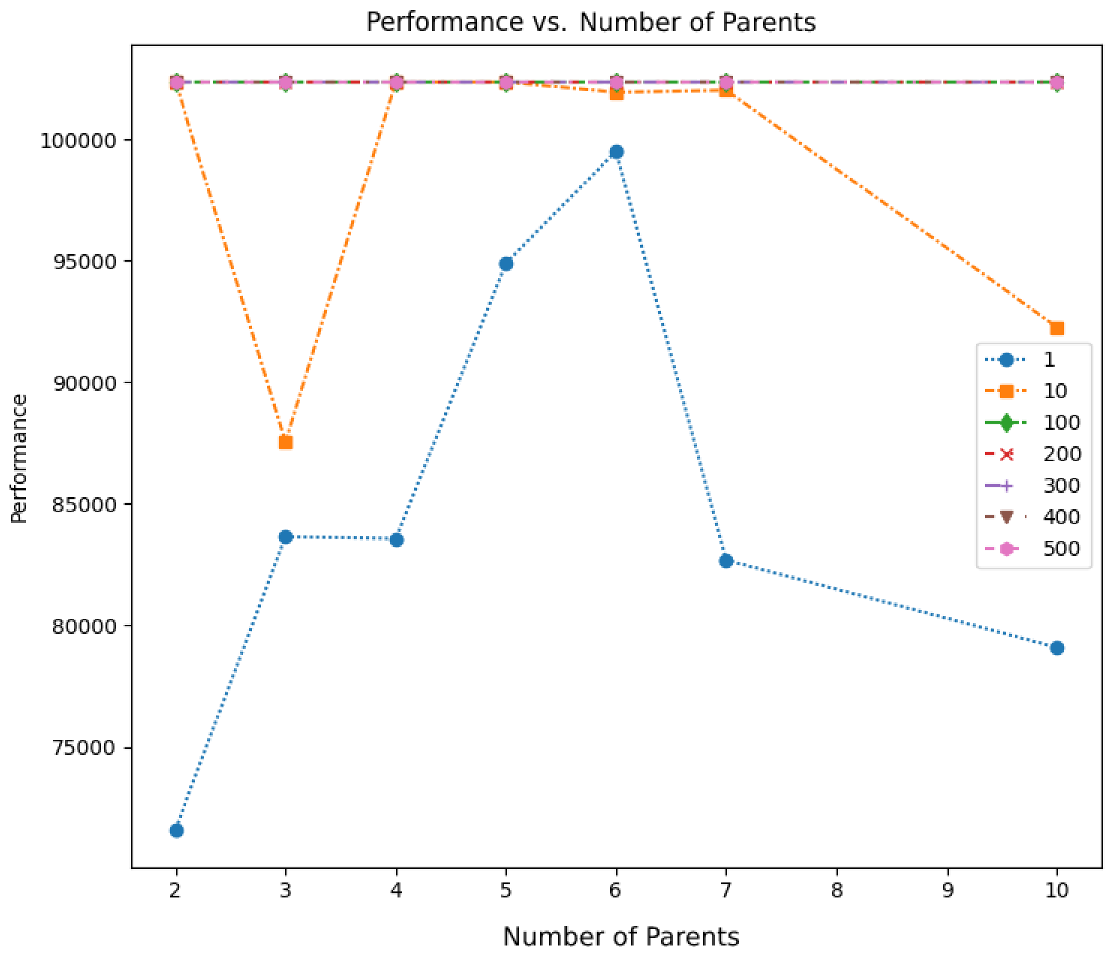
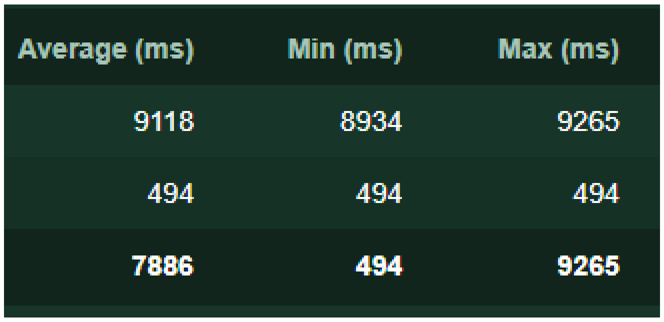


Figure 9 Graph of Performance vs. Number of Parents

To generate this graph, matplotlib[9] was used to plot the subgraphs and create the overall graph. The figure assumes an overall budget of ten thousand, which allows the algorithm to include all possible components from within the database. This gives the algorithm a chance at running at the worst run-time. From within the algorithm, a list of the number of generations was provided as shown in the legend. Within each generation, the number of parents varied. Each generation ran until a solution was provided with the number of parents it was given. Once the solution is provided, the next number of parents is used to run the algorithm again with the same number of generations.

In figure nine, the different color lines show the number of generations that were present to generate that specific solution. From the graph it seems that any number of generations greater than ten would give an optimal solution. However, this was not the case. The algorithm is completely random. This means that the components being used are generated randomly in a random order. Hence, the solution provided does not always guarantee that the optimal one was provided. Once the algorithm runs in real-time, the solution would vary by a component or two. This would show that the solution was stuck within a local maxima. After many trials and runs, the optimal number of generations that would provide a consistent solution was about one-thousand generations, which is not shown on the graph. The number of parents was kept at ten to help the algorithm not get stuck within a local maxima.

With the algorithm running with one-thousand generations, the execution time of the algorithm took a hit. However, it is still quick, accounting for the time it takes the database query to complete. Shown in the figure below. The average execution time was about nine seconds. The fastest time was about 8.9 seconds, and the slowest time was about 9.3 seconds. Considering the large database and large number of components being used, this run-time is exceptional. The time shown below is the time from when the user makes a request to run the algorithm to when the algorithm finishes and gives a response.



Average (ms)	Min (ms)	Max (ms)
9118	8934	9265
494	494	494
7886	**494**	**9265**

Figure 10 Algorithm Run Time

5.2 PERFORMANCE

An algorithm's efficiency and solution play an important role in determining its performance. Evolutionary algorithm's performance is mainly based upon the quality of the solution and how quickly it can come up with a solution. Typically, the run-time for any constraint given in the project would have the same run-time. This is due to the algorithm's end condition of reaching the maximum threshold of run-time.

Space complexity is important when determining performance. Space complexity for the evolutionary algorithm is mainly based on the parameters that are provided, which can include the population size. In this implementation, the population size dominates the space complexity. The format of the population is a matrix. Hence, the overall space complexity of this implementation of the algorithm is going to be O(CGR), with C being the number of CPUs, G being the number of GPUs, and R being the number of RAM. There would be no way of improving this value because every component can be a part of the solution. Therefore, space complexity cannot be compromised.

Run-time is an important factor that has to be accounted for since users need fast responses. When analyzing this algorithm, it becomes quite difficult to determine what the run-time would be. The first step in the algorithm is to initialize the population. In this function the first three lines retrieve the data on the components from the database. Since the query is simple, the run-time of each data query runs at O(n) time with n being the number of elements in the specified table. The next function that does not take constant time is the built-in python max function. This function takes O(p) time with p being the number of parameters passed to the function. Since the comparisons are constant, the next line that would take significant time would be the loop. The loop would take O(n) time with n being the number of elements in the largest dataset. The overall runtime of initializing the population is O(np).

The next step in the algorithm is to select the parent from the initial population. The run-time of the outside loop would be O(n) with n being the number of elements in the population matrix. The inside loop has a runtime of O(p) where p would be the number of parents. Assuming that the worst case does occur, the outer loop will run at O(n) and the inner loop will be executed inside of the if statement with a run-time of O(p). Overall, the function would have a runtime of O(np).

Generating offspring is executed after the parents have been selected. This is the start of the first generation. The loop's run-time would be O(p) where p is the number of parents. The next step in the function is to create a list of possible combinations from the parents. This would have an overall run-time of O(CGR) because there are three components being used as part of the combinations. C is the number of CPUs from the parent, G is the number of GPUs from the parent, and R is the number of RAM from the parent. Essentially, C = G = R = p in this scenario. Overall, the run-time of this function would conclude to $O(p^3)$.

Mutation is the next major step to be executed. The initialization of variables all take constant time. With the next line being a while loop, it becomes difficult to determine what the run-time is. In this case, all the while loops ensure that all the numbers do not match and that there will be a new component selected with a better performance. Assuming that this would be the worst-case scenario, the run-time of the function would take O(n) where n is the population size. With all other lines in the function taking constant time, the run-time of this algorithm would be O(n).

The fitness evaluation function is the best performing function in this iteration. Its run-time is dependent on how many components are supported by the algorithm. In this case, the run-time would be O(n) where n is the number of components needed in the solution. In this case, n would be three because the CPU, GPU, and RAM are the only components being analyzed.

The analysis of these functions is not enough to determine the overall run-time of the algorithm. The algorithm is dominated by the number of generations that need to be ran. For each generation, the previous functions analyzed will be executed. Hence, the overall run-time of the implemented evolutionary algorithm in this study is $O(np^3g)$ where n is the number of components, p is the number of individuals in the population, and g is the number of generations.

6 FUTURE WORK

Evolutionary algorithms are a growing research field in computer science. The algorithm performs significantly well with problems such as optimization, scheduling, planning, design, and management (Slowik & Kwasnicka, 2020). Evolutionary algorithms have uses across many different fields that are currently in use. Electrical engineering, artificial intelligence, automation, and management services are some of the major fields that use evolutionary algorithm to solve their problems (Slowik & Kwasnicka, 2020). With the growing use of the evolutionary algorithm, there are research topics that can be done on this specific algorithm.

Improving the performance. With all algorithms, performance can always be improved. This can be finding the perfect number of parents or the ideal number of generations. The space complexity and intense run-time can be further improved.

Adding more components. This study only analyzes the CPU, GPU, and RAM. However, there are many other components that make up a computer system. This includes storage, power supply, and motherboard. Some of these components do not impact the performance of the computer; rather, it is more based on user preference and customization. The storage can impact the performance of a computer along with the power supply. It is common for users to purchase computer components and not have enough power rating in their power supply for the system. It is also common to experience a slow computer system with high-end components. This can usually occur from low performing storage.

7 CONCLUSION

With the evolutionary algorithm, it is possible to create an algorithmic way to solve the optimization problem that comes with purchasing computer components. The algorithm was able to take in constraints provided by the user to fully customize their system. The presented implementation of the algorithm provides a probable solution that accounts for compatibility and having utmost performance. With the runtime being high, in the real-world situation, the algorithm returns at a reasonable time. The algorithm yields a high accuracy rate of having a probable solution. However, the algorithm and implementation can be further improved with future work that can be done.

8 REFERENCES

Brownlee, J. (2019, July 15). A Tour of Optimization Algorithms. *Machine Learning Mastery. https://machinelearningmastery.com/tour-of-optimization-algorithms/*

Django. (n.d.) Django web framework. Retrieved from *https://www.djangoproject.com*

Eiben, A. E., & Smith, J. E. (2015). What is an evolutionary algorithm? In Introduction to Evolutionary Computing (pp. 25-48). Springer Berlin Heidelberg. *https://doi.org/10.1007/978-3-662-44874-8_3*

Ippolito, P. P. (2020, March 10). Introduction to Evolutionary Algorithms. Towards Data Science. https://towardsdatascience.com/introduction-to-evolutionary-algorithms-1278f335ead6

Kübler, R. (2020, January 15). An extensible Evolutionary Algorithm Example in Python. *Towards Data Science. https://towardsdatascience.com/an-extensible-evolutionary-algorithm-example-in-python-7372c56a557b*

Python Software Foundation. (n.d.). itertools – Functions creating iterators for efficient looping. Python 3.11.3 documentation. Retrieved from *https://docs.python.org/3/library/itertools.html*

Python Software Foundation. (n.d.) random – Generate pseudo-random numbers. Python 3.11.3 documentation. Retrieved from *https://docs.python.org/3/library/random.html*

React, (n.d.). React documentation. Retrieved from *https://react.dev*

Slowik, A., & Kwasnicka, H. (2020). Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, 32, 12363-12379. *https://doi.org/10.1007/s00521-020-04832-8*

Soni, D. (2018, February 18). Introduction to Evolutionary Algorithms. Towards Data Science. *https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac*

SQLite. (n.d.). SQLite: Embedded database software. Retrieved from

9 FOOTNOTES

[1]CPU: Stands for central processing unit; the brains of the computer.

[2]GPU: Stands for graphics processing unit; the unit that oversees heavy computations and graphics.

[3]RAM: Stands for random access memory; holds application data that may be used by the CPU.

[4]Refer to section 4.1.5

[5]This is a built-in python module that does math computations.

[6]Refer to figure 5.

[7]This is a built-in python module that can be used with the installation of Python.

[8]API: Stands for application programming interface. It is used to connect two pieces of software together.

[9]This is a third-party python module that is typically used by developers to aid in plotting graphs.