Scheduling

Genetic Algorithm

Jaden Alesi, Logan Haberman, Grant Harsch, Daniel Hough

Problem Description

This problem revolves around making a sixteen-hour long schedule, filled with approximately sixteen different assignments with different priority levels. It is the genetic algorithm's job to create and find an optimal schedule that gets the most work done, with the priority of the assignments taken into account.

The objective of the algorithm is to produce an optimized schedule, in which the student completes the most amount of work, with priority of assignments taken into account. The metrics that need to be optimized are completion of high priority assignments and not wasting time on assignments that are already going to be complete.

College classes lead to busy schedules, with each class assigning work due each week. Prioritizing different classes' work can be difficult, as every assignment will take different amounts of time, be worth different points, and have different levels of priority. By using a genetic algorithm to optimize a schedule, the student can find how to most effectively use their time.

Parameters

Population size (popSize)	250
Number of generations (numGens)	80
Number of children per gen (genSize)	100
Mutation rate	0.1
Immigration rate	0.08

The above parameters may seem somewhat arbitrary, however these settings made sure that the algorithm would converge as consistently as possible to the maximum found fitness of ~169.45. Occasionally the algorithm gets stuck at a fitness of ~122, however increasing immigration rate and mutation rate to their respective values minimized the amount this happened. After analyzing a few runs of the algorithm, most would converge around generation 70, but the extra few generations would ensure that more of the runs converge fully with 100 children per generation.

Population Initialization

The initial population in this genetic algorithm was generated randomly. First, an empty array is created, which then gets filled with sixteen randomized real numbers. The chosen chromosome size was based on the scenario of being a busy college student trying to get work done, with time for meals and sleep.

Each gene in a chromosome represents a task, and this task is assigned a random number in the range of one to nine. Tasks and priorities were represented as real numbers rather than binary, as real numbers are a straightforward representation of the data. Representing the tasks and priorities as real numbers rather than binary was chosen for intuitiveness and simplicity.

```
Function populateChromosome():

i = 0

While i < 16 DO

array[i] = random_int_between(1,9)

i = i + 1
```

Fitness Evaluation

The goal of the fitness function was to maximize a score that depended on the priority of tasks done, as well as how much progress was made on the tasks done, while punishing overcommitting time to a task. To begin, a function that would vary between zero and one, while rising to a maximum when the time committed to a task was equal to the total time required by the task, and then quickly falling down the more time was committed.

To find a good function, different functions were plotted in Desmos. The base function chosen was $e^{\frac{-1}{x}\left|1-e^{(x-t)}\right|}$, however this would cause all fitnesses to be relatively close to one another, so to make the better fitnesses stand out better, the function was put in the exponent of four. $4^{e^{\frac{-1}{x}\left|1-e^{(x-t)}\right|}}$-1 was thus our final equation, with the minus one added to flatten the function back to zero as the exponent approaches zero.

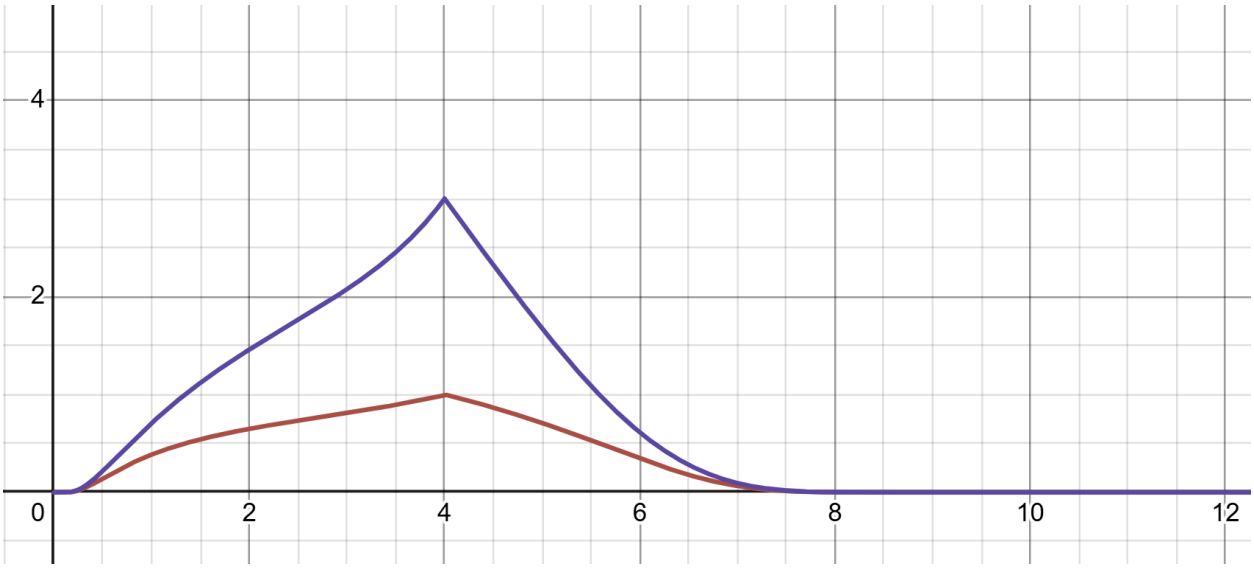

Figure 1: Plots of original function in red, and scaled function in purple, for a task that should take four hours to finish.

Parent Selection

Parent selection is performed via a Roulette Wheel strategy. Our reasoning for this is that we wanted to make sure that we explored as much of the solution space as possible.

Roulette wheel gives us the means to do this by permitting the lower-fitness members of the population a chance to reproduce. This does introduce a problem that undesirable children will also be produced, but the roulette wheel is designed to favor higher-fitness members of the population and only provide a small chance for the underperforming members of the population to be selected.

Crossover

We used the order crossover technique for our project. The two crossover points we used were at the 4^{th} and 14^{th} index. The children that were created by this crossover are given the middle slices of their respective parents. Then we used the orders of the opposite parents to fill out the rest of the children.

The order in which the children are filled is the end slice of the opposite parent, then first slice, then middle. When choosing an allele from the order we must make sure that there is not already that same allele in the child already. However, this method would not work entirely for us since we have multiple copies of alleles in our chromosomes to begin with, which creates problems in order crossover.

So we had to change the way we added alleles in the child chromosome. Instead of checking if a child had one of the alleles already, we checked how many copies of that allele the child had instead. If the child had less than the count of that allele in the parent, then we can use that allele in the child. If there was more of that allele in the child, then we cannot use it and move down through the order.

Parent 1: Parent 2

1,1,1,1,2,2,3,6,6,7,4,3,3,4,4,6 5,5,5,5,6,6,6,7,8,8,3,2,1,4,6,9

^ ^ ^ ^

Child 1: Child 2

0,0,0,0, 2,2,3,6,6,7,4,3,3,4,0,0 0,0,0,0,6,6,6,7,8,8,3,2,1,4,0,0

To fill in the zeroes use these orders of parents. Fill in from end slice -> front slice -> middle slice:

Order for Child 1:

6,9,5,5,5,5,6,6,6,7,8,8,3,2,1,4

Order for Child 2:

4,6,1,1,1,1,2,2,3,6,6,7,4,3,3,4

Child 1 is appended into new population:

5,5,5,5, 2,2,3,6,6,7,4,3,3,4,6,9

Child 2 is appended into new population:

1,1,2,3,6,6,6,7,8,8,3,2,1,4,4,1

Pseudo Code:

```
//Do these loops for each child
while ((i < length of child) and (j < length of child)):
        if (countOf(alleleInChild) < countOf(alleleInOrder)):
                child(i) = alleleInOrder
                i++
                j++
        else:
        j++
//set placement of index in child
i = 0
while ((i < firstCrossOverPoint) and (j < length of child)):
        if (countOf(alleleInChild) < countOf(alleleInOrder)):
                child(i) = alleleInOrder
                i++
                j++
        else:
        j++
```

Mutation

The mutation operator implemented in this genetic algorithm is purely random. A mutation will occur based on the mutation probability of 10%, so that it is rare enough to not affect every chromosome, but frequent enough to add diversity into the population. This allows for exploration in the solution space and diversity across the upcoming generations.

When a mutation is triggered by the randomly generated number, a random index within the chromosome array is chosen. The gene at this index is replaced with a random number in the range of 1-9.

```
Function mutation (chromosome, mutation prob = 0.1):
    if random_value_between_0_and_1 < mutation prob THEN

        mutated_index = random_int_between(0, len(chrome.array) - 1)

        mutated_gene = random_int_between(1 and 9)

        chrome.array[mutated_index] = mutated_gene
```

Replacement Strategy

We employed an elitist replacement strategy. We did this because we wanted to favor higher fitness members of the population. It was very important to us that we try and constantly improve the population, so getting rid of less-desirable members meant we had a better selection of candidates to choose from in parent selection.

By employing elitism, we were able to make sure that better children were being produced, and that our population was improving as a whole.

When considering lower fitness individuals for random immigration, we used a rate of 8%. This allowed us to continue to explore the solution space.

Results and Analysis

Below is a graph of the fitness over time for one run of the genetic algorithm.

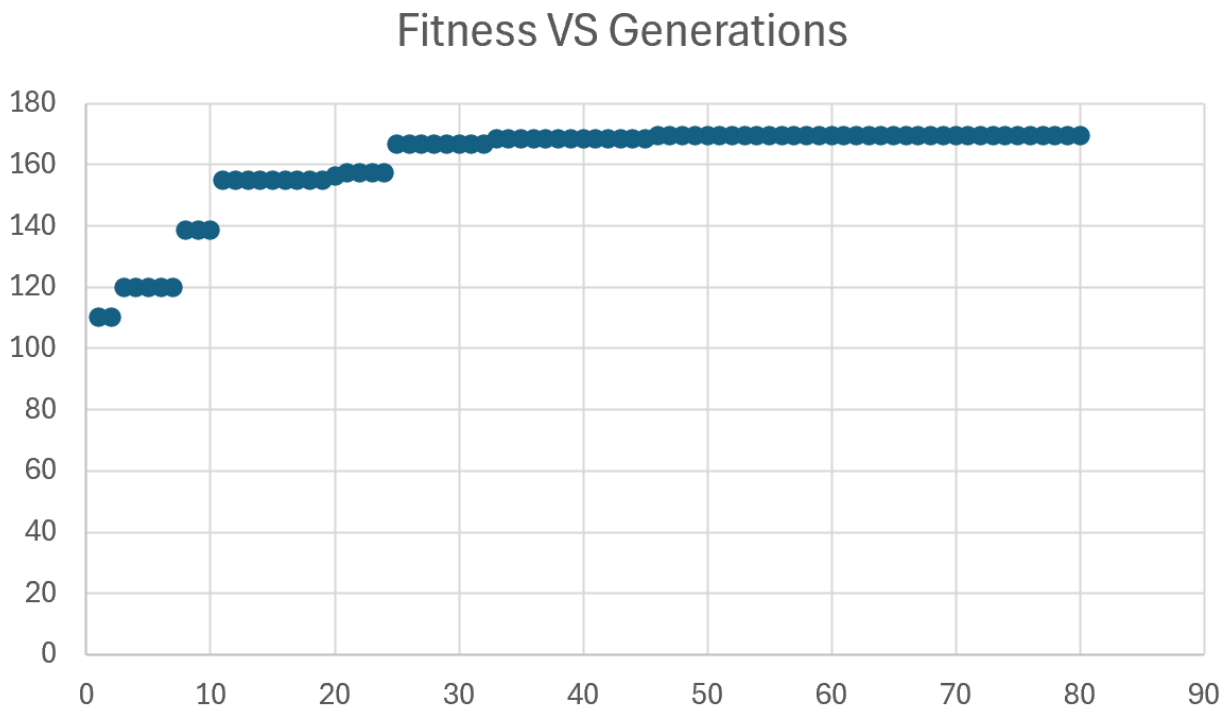

Figure 2: Fitness over generations for one run.

After trying other parameters, lowering mutation rate or immigration rate would cause the algorithm to get stuck more often. Increasing generations served little purpose as most runs would flatten out after around 70 generations and not increase any more no matter how long it runs. Decreasing the children per generation and population size would cause the diversity to be low and potentially get stuck in worse solutions.

Our solution seems to cap out at a fitness of 169.45, which may be a maximum value based on our situation or maybe a local maximum that keeps catching solutions based on our encoding and crossover. Our solution gets found quickly; however, it takes up a lot of space while running.

Links

https://github.com/CIS-department-SVC/ga-project-scheduler-group