An Evolutionary Approach to Timetabling

Dieter Brehm & Elias Gabriel

4 May 2020

An Evolutionary Approach to Timetabling

Introduction

We constructed an easy to use study scheduling tool to assist students in managing their workloads thorughout the week. It makes use a genetic algorithm, which is a subclass of evolutionary algorithm, a series of compiled heuistics, and a list of know constraints to attempt and evolve an optimal agenda.

Background

Evolutionary algorithms are different from raw implementations of solutions, and are a subset of optimization problems that derive strategies from real world speciation and evolution. As opposed to building globally-optimal strategies or combining locally-optimal ones, evolutionary algorithms adjust and adapt their current solutions over many generations to best fit some pre-defined goal. The process is intended to follow evolutionary theory, where a species adapts its genepool over time to best match environmental conditions and maximize liklihood of survival.

Genetic algorithms phrase solution-finding in terms of those natural adaptive strategies, like gene mutation and natural selection. The solutions, which are all progressive generations on some defined common ancestor, are ranked based on their position in some fitness landscape. In nature, the function that defines that landscape most often takes the form of "likelihood to survive and procreate". In the case of homebrew algorithms, that fitness function can be defined as any numerical representation of success, like "how far a robot moves before falling."

These differ from typical gradient descent or A* path finding algorithms (See (Patel 2020b)), in that we are not evaluating a gradient in order to determine a direction to move towards. Rather, we are trusting that the evolutionary process will result in an improvement of fitness.

In *similarity* to path finding algorithms, however, this process is done through heuristic methods to find approximations of the optimality of given solutions (See (Patel 2020a) and (Gour 2018) for more.)

Problem Space

We chose to study the details and implementation surrounding homework scheduling. This scheduler is a continuation of a previous project of ours, a meeting scheduling application called Let's Sched It.[1] We implemented a simple genetic homework scheduling system to assist students in optimizing when to study and complete their assignments. It can be difficult to balance NINJA hours, class time, and lengthy assignments; using a heuristic-based schedule in the context of time, we tried to ease that process by generating suggestions for how one could approach their work. Parameters to our weighting system include the due dates for specific homework, completion time estimations from the user, class hours, NINJA (TA) hours, and desired sleeping hours.

We see parallels to the knapsack and other pathfinding problems, and also in concepts used in CPU scheduling. Timetabling is also a very defined homework space in regards to university class scheduling methods, where improving the efficiency and optimality of solutions is important and increasingly complex.

Design Process

We first thought to use a path finding heuristic setup. We used Amit Patel's fabulous pathfinding website (Patel 2020b) to help build our foundational knowledge on heuristic-based path finding. We used another page on the site, found at (Patel 2020a), to learn more about heuristics and figure out how we could adapt them for our own algorithm. After a while researching, we discovered genetic algorithms, which seemed both interesting and very promising.

We decided that segmenting our problem into discrete blocks would be the first step to solving it. Each day is divided into half-hour blocks, which we figured would be sufficient as most classes (at Olin) fit into roughly half-hour increments.

To facilitate our testing, we created functions to allow our program to load data from three CSV files (course assistance hours, course schedules, and homework assignments). Also, we wanted the output of our algorithm to be a list of times during which we should complete those loaded assignments. To reflect that we created a class to store information about specific assignments as well as the times to complete them.

Knowing we would be dealing with a list of TODOs, we implemented several established crossover and mutation techniques found through research. Before breeding two soultions, we had to establish a robust way of selecting viable parents. We opted for an approach known as tournament selection (TS). In essence, TS selects the fittest solution from a subsample of size `k` from the parent population. We decided on a value of `k=3`, making our selection process an implementation of Ternary Tournament Selection (TTS). There is a significant amount of literature devoted to selecting appropriate values of `k`, all of which helped inform our decision (Miller and Goldberg 1995).

For our crossover (breeding) function, we implemented single-point crossover (SPX). SPX simply picks a random spot between both parents, splits them at it, and then recombines them in an alternating order to create a unique child. As before, there are many options for crossover algorithms, but our research seemed to indicate that SPX adequately and efficiently provided viable solutions.

[1] https://github.com/thearchitector/LetsSchedIt/tree/gh-pages

Mutations are perhaps the most critical step in the entire process of genetic evolution, as they allow children to evolve properties that they might not get from either of their parents. That genetic variation provided a much larger search space over which our fitness function could evaluate solutions, and therefore significantly increased the success of an evolved solution. When designing our mutations, we had to come up with possible ways in which a given solution(s) could gain or lose traits. After thinking, we settled on a list of several possible mutations:

- swapping the completion day of two distinct solutions
- swapping the completion time of two distinct solutions
- randomly altering the start/end time of a random solution

Results

We successfully implemented a homework scheduler using the functions above, which yieled fairly promising results. Our entire program was able to generate a solution with no conflicts with existing classes and no overdue assignments. We walk through the results and our program in a recorded video[2] and the code is open source.[3] The output of the program is shown in 1.

[2]https://www.youtube.com/watch?v=hLF6YvrsABU
[3]https://github.com/Inkering/Letshwit

	S	U	M	T	W	R	F
0	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
1	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
2	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
3	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
4	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
5	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
6	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
7	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
8	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
9	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
10	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
11	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
12	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
13	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
14	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
15	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep	Sleep
16							
17							
18			DesNat	ISIM		DesNat	ISIM
19			DesNat	ISIM		DesNat	ISIM
20			DesNat	ISIM		DesNat	ISIM
21			DesNat	ISIM		DesNat	ISIM
22			DesNat	ISIM		DesNat	ISIM
23	Meals	Meals	Meals	Meals	Meals	Meals	Meals
24	Meals	Meals	Meals	Meals	Meals	Meals	Meals
25	Meals	Meals	Meals	Meals	Meals	Meals	Meals
26							
27		Elias	ModSim	SoftDes	Elias	ModSim	SoftDes
28		Elias	ModSim	SoftDes	Elias	ModSim	SoftDes
29		Elias	ModSim	SoftDes	Elias	ModSim	SoftDes
30		Elias	ModSim	SoftDes	Toolbox #3	ModSim	SoftDes
31		Elias			Toolbox #3		
32		Elias	Dieter		Elias	Dieter	
33			Do your report			Dieter	
34			Do your report			Dieter	
35			Dieter			Dieter	
36			Dieter			Dieter	
37							
38							
39			Build the hopper				
40			Build the hopper				
41			Build the hopper				
42			Build the hopper				
43			Build the hopper				
44			Build the hopper		Get lab working		
45					Get lab working		
46	NightSleep	NightSleep	NightSleep	NightSleep	NightSleep	NightSleep	NightSleep
47	NightSleep	NightSleep	NightSleep	NightSleep	NightSleep	NightSleep	NightSleep

Figure 1: Example result from the scheduling program

Experiment

We also ran an experiment to determine the real runtime performance of the application through a range of population and generation sizes. Interestingly, the runtime of our algorithm remained constant with a given problem no matter the population size or number of generations. Figure 2 shows a plot of time versus population size, and Figure 3 shows time versus generation number.

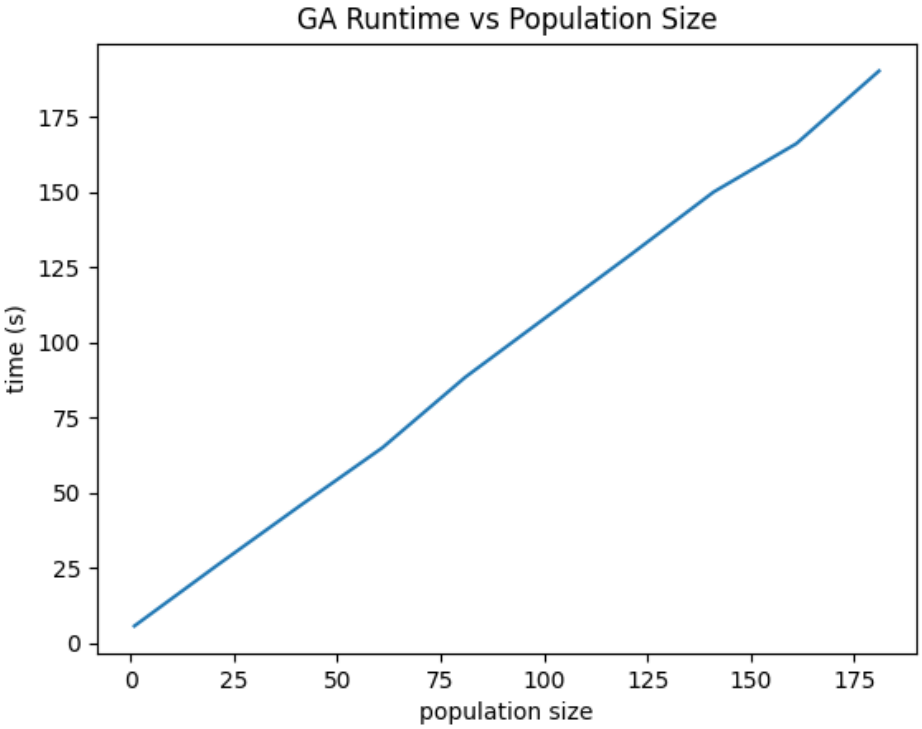

Figure 2: time versus population size for a 100 iteration run

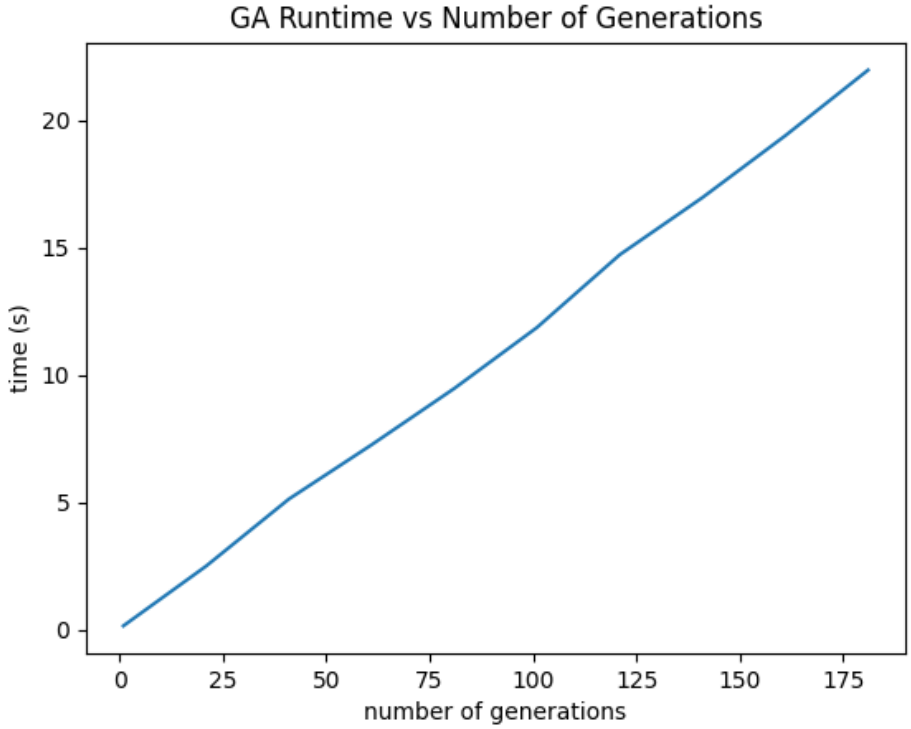

Figure 3: time versus generation number for a 12 population run

Conclusions

We're really happy with how our project turned out. It finds optimal-looking solutions regularly, and appears to be quite robust and adaptive to a wide range of input constraints. As a set of next steps, we might improve its runtime by selecting and utilizing more efficient data structures, as well as integrate it into a into a more usable user interface (like a separate web application or Let's Sched It.)

References

Gour, Rinu. 2018. "Heuristic Search in Artificial Intelligence — Python." https://medium.com/@rinu.gour123/heuristic-search-in-artificial-intelligence-python-3087ecfece4d.

Miller, Brad L., and David E. Goldberg. 1995. "Genetic Algorithms, Tournament Selection, and the Effects of Noise."

Patel, Amit. 2020a. "A*'s Use of the Heuristic." https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

———. 2020b. "Introduction to a*." https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.