David Monteleone

Tomer Shemesh

Kelly Shiptoski

Genetic Scheduling

** Note the final project changed its approach significantly from the originally proposed idea.

** To run our program, please see the README.txt file in the included submission.

** CS385FinalPresentationGeneticScheduling.pdf and CS385FinalPresentationGeneticScheduling.pptx contain our presentation.

** RubyDFS.txt contains the Ruby code we wrote for DFS comparison. You can run it though due to it requiring the whole API.

Problem Statement

The group set out to create a program that would solve a problem for us. The problem we attempted to solve was to create a Drexel term schedule based on the user's required classes. The goals of this program were to be accurate and fast. People don't want to sit around for hours to wait for an optimal schedule to be built. Ultimately, the group wanted to design and implement a program that can solve this problem for us. ie: Evolutionary approaches!

Original Idea

The original proposed idea consisted of using neural networks to solve this problem. The group made several attempts to fit our data into the neural network library

that MATLAB provides. However, we could not come up with a reliable method for using neural networks. As such, the group switched to a genetic algorithm approach. Since genetic algorithms do not rely on training set data, the group did not utilize a training set as we suggested in the original proposal. The group also suggested using user preferences and input to further skew the direction that the program learns. We experimented with prompting the user on each generation, but that became too time consuming for the user. We built in the user preferences into our fitness function which will be discussed in further detail below. Apart from these changes, the overall scope of the project remains the same. Build a Drexel scheduling program that builds near optimal schedules for the user based on their desired preferences.

Problem Solution

The group decided to use our knowledge obtained from HW4, on genetic algorithms, to create our program. The solution would allow for a self learning scheduling application. The user would be able to set the classes they were wanting to take along with the preferences that they wanted to have in that schedule. We decided on having the following preferences: morning classes, a tightly compact schedule, having online classes, and having no classes on certain days. The program would ultimately create a random starting population of classes that the user wants. Then through genetic crossover, mutation, and ranking the program itself would determine the best schedule it could find within a set number of generations.

Our Approach

We agreed to use MATLAB to implement this program. MATLAB allows for excellent rapid prototyping and it seemed like the best tool for the job. Tomer had previously built a Ruby based Drexel WebTMS API. This gave the group easy access to a REST endpoint that can provide us with the needed class information. The overall approach would be as follows:

1. Have the user hardcode/define the classes which they want to have scheduled.
2. Have the user hardcode/define the preferred type of schedule they want to have.
3. Run the program
    a. Generate 10 random permutations of the desired classes to start the population.
    b. Rank the permutations based on how well they match the user's desired preferences via the fitness function. Higher scores are better.
    c. Choose the top 6 subjects in the population and have them breed 1 child per two parents. They breed as follows: 1 and 2, 3 and 4, and 5 and 6. This results in 3 new children. The breeding is done via a crossover method discussed later.
    d. Have a 33% chance that the child will mutate. Mutation is discussed later.
    e. Obtain the fitness values on the children and place them into the population in a sorted manner.
    f. Repeat steps c through e 100 times, or generations.
4. Display the final near optimal schedule in a MATLAB GUI.

5. Display a plot that shows the increase in fitness value over the generations.

Problem Representation

Since the group decided to use MATLAB, the structural representation relied heavily on MATLAB structures. The population is represented by a cell array. Each entry in the cell array contains a cell array of classes in the first position. The second position is the fitness score of that respective schedule or subject. A member in the population is represented as a pair of schedule and fitness score. This structure for the population can be seen below:

```
schedules =

    {7x7 cell}    [61.8000]
    {7x7 cell}    [59.5000]
    {7x7 cell}    [57.8000]
    {7x7 cell}    [54.8000]
    {7x7 cell}    [43.3000]
    {7x7 cell}    [41.5000]
    {7x7 cell}    [39.3000]
    {7x7 cell}    [38.3000]
    {7x7 cell}    [     34]
```

Within each subject of the population, is a cell array of structs. Each of these structs refers to a unique class that we desire in our final schedule. An example of this can be seen below:

```
K>> schedules{1}

ans =

    [1x1 struct]
    [1x1 struct]
    [1x1 struct]
    [1x1 struct]
    [1x1 struct]
    [1x1 struct]
    [1x1 struct]
```

Each of the structs that represent classes have the attributes in the figure below. These are the attributes that are kicked back to our program from our initial REST call to the Ruby server. The server returns class structs that can be seen below. These structs contain all of the needed information for us to perform our fitness evaluation and crossover methods.

```
K>> schedules{1}{1}

ans =

                  id: 3693
                 crn: '21354'
            class_id: 'ECON201'
    instruction_type: 'Lecture'
  instruction_method: 'Face To Face'
             section: '005'
          begin_time: '2000-01-01T12:00:00.000-05:00'
            end_time: '2000-01-01T13:50:00.000-05:00'
          max_enroll: 60
       current_enroll: 60
             credits: 4
              campus: 'University City'
    section_comments: 'Waitlist capabilities until Tuesday prior to class beginning…'
       textbook_link: 'http://drexel.bncollege.com/webapp/wcs/stores/servlet/TBListView?cm_mmc=RI-_-457-_-1-_…'
            building: 'GHALL'
                room: '033'
                term: 'Winter'
           term_type: 'Quarter'
           term_year: '15-16'
          created_at: '2015-12-19T05:02:12.601-05:00'
          updated_at: '2016-01-26T13:18:51.047-05:00'
    days_time_string: 'TR 12:00pm-1:50pm'
          professors: [1x1 struct]
```

REST Call Utilization

Without the utilization of Tomer's Drexel WebTMS Ruby REST API, this project would not have been possible within the given amount of time. It should be noted, this API does not do any of the genetic portions of the assignment. Nor did it ever do the genetic algorithms that are discussed in this write up. We simply use it to obtain the class information about the user's requested classes. The only portion written in the Ruby API for this assignment was the depth first search that is discussed later. This was done due to the performance issues encountered in MATLAB. The code used to obtain the information from the REST server can be seen below. MATLAB has some excellent REST functions that made calling the server very straight forward.

```
% Downloads all desired classes from the WebTMS API
function all_classes = DownloadClasses( classes )
% Downloads all desired classes from the WebTMS API

term = 'Winter';
term_type = 'Quarter';
term_year = '15-16';
url = 'http://107.170.133.244/api/v1/search_webtms';
options = weboptions('MediaType','application/json');

all_classes = cell(size(classes,1));

for i= 1:size(classes)
    data = struct();
    data.term = term;
    data.term_type = term_type;
    data.term_year = term_year;
    data.classid = classes{i,1};
    %data.days = {'Monday','Tuesday'};
    data.instruction_types = {classes{i,2}};
    response = webwrite(url,data,options);
    all_classes{i} =  response;
end


end
```

Our Results

The group experimented with changing the preferences around to display the ability our program has to skew the results towards what the user actually wants. The first set of results is for users that desire no online classes, a tightly packed schedule, morning classes, and having no classes on Fridays. Here is the MATLAB GUI output for the best schedule:

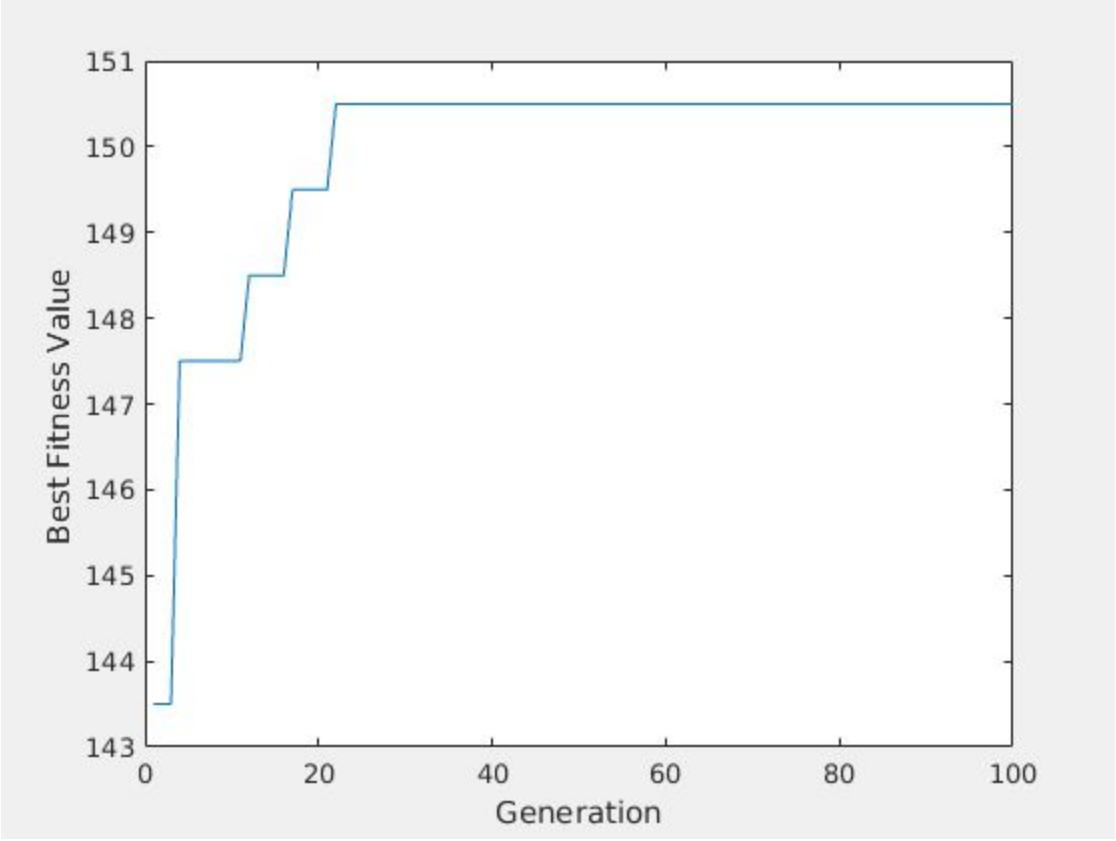
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Online
8:00am	ECON201	CHEM102	ECON201				
8:30am	ECON201	CHEM102	ECON201				
9:00am	ECON201	CHEM102	ECON201	ENGR102			
9:30am	ECON201	CHEM102	ECON201	ENGR102			
10:00am				ENGR102			
10:30am				ENGR102			
11:00am	ENGR102	CHEM102					
11:30am	ENGR102	CHEM102					
12:00pm	CHEM102		CHEM102		CHEM102		
12:30pm	CHEM102	CS283	CHEM102	CS283	CHEM102		
1:00pm		CS283		CS283			
1:30pm		CS283		CS283			
2:00pm							
2:30pm							
3:00pm							
3:30pm							
4:00pm							
4:30pm							
5:00pm							
5:30pm							
6:00pm							
6:30pm							
7:00pm							
7:30pm							
8:00pm							
8:30pm							
9:00pm							
9:30pm							
10:00pm							

As you can see from observing the schedule output, the classes are organized towards the beginning of the day and tightly compact. In addition to this, there are no online classes and only one class on Fridays. That seems like a rather well made schedule based on the given preferences. This only took about a minute to run too!

These are the hardcoded/defined user preferences that resulted in the previous schedule.

```
preferences = struct();
preferences.online_classes = 0;
preferences.tightly_packed = 10;
preferences.morning_classes = 10
preferences.no_classes = ['F'];
```

The plot below displays the our top schedule getting better over the generations.



The second set of results is for users that desire online classes, a tightly packed schedule, no morning classes, and having no classes on Wednesdays. Here is the MATLAB GUI output for the best schedule:

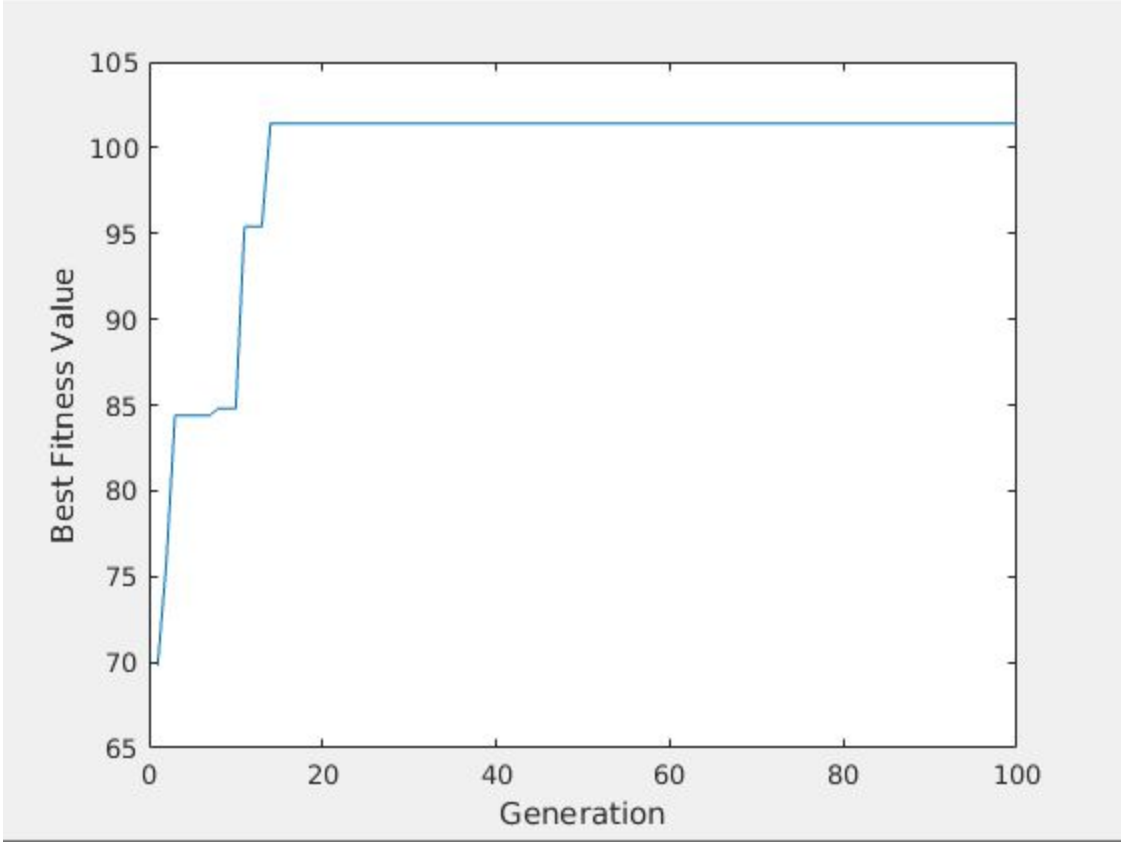
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Online
8:00am							ECON201
8:30am							
9:00am							
9:30am							
10:00am							
10:30am							
11:00am							
11:30am							
12:00pm			CHEM102				
12:30pm			CHEM102				
1:00pm		ENGR102	CHEM102				
1:30pm		ENGR102	CHEM102				
2:00pm	ENGR102	ENGR102	CHEM102				
2:30pm	ENGR102	ENGR102	CHEM102				
3:00pm							
3:30pm							
4:00pm	CHEM102		CHEM102		CHEM102		
4:30pm	CHEM102		CHEM102		CHEM102		
5:00pm							
5:30pm							
6:00pm							
6:30pm				CS283			
7:00pm				CS283			
7:30pm				CS283			
8:00pm				CS283			
8:30pm				CS283			
9:00pm				CS283			
9:30pm							
10:00pm							

This schedule is for the same classes as the previous example, but it's clearly different. We see the schedule biasing away from the early morning hours, but still remaining relatively compact. We also now have an ECON class online instead of during the week. Another aspect that we greatly desired.

These are the hardcoded/defined user preferences that resulted in the previous schedule.

```
preferences = struct();
preferences.online_classes = 10;
preferences.tightly_packed = 10;
preferences.morning_classes = 0;
preferences.no_classes = ['W'];
```

The plot below displays the our top schedule getting better over the generations.



As you can see between the two differing preferences, the schedules are dramatically different. Of course they don't fit all of the user's defined preferences exactly, but that is to be expected with any scheduling application. Not wanting to have classes on a certain day may not be possible if a class is only offered on a certain day.

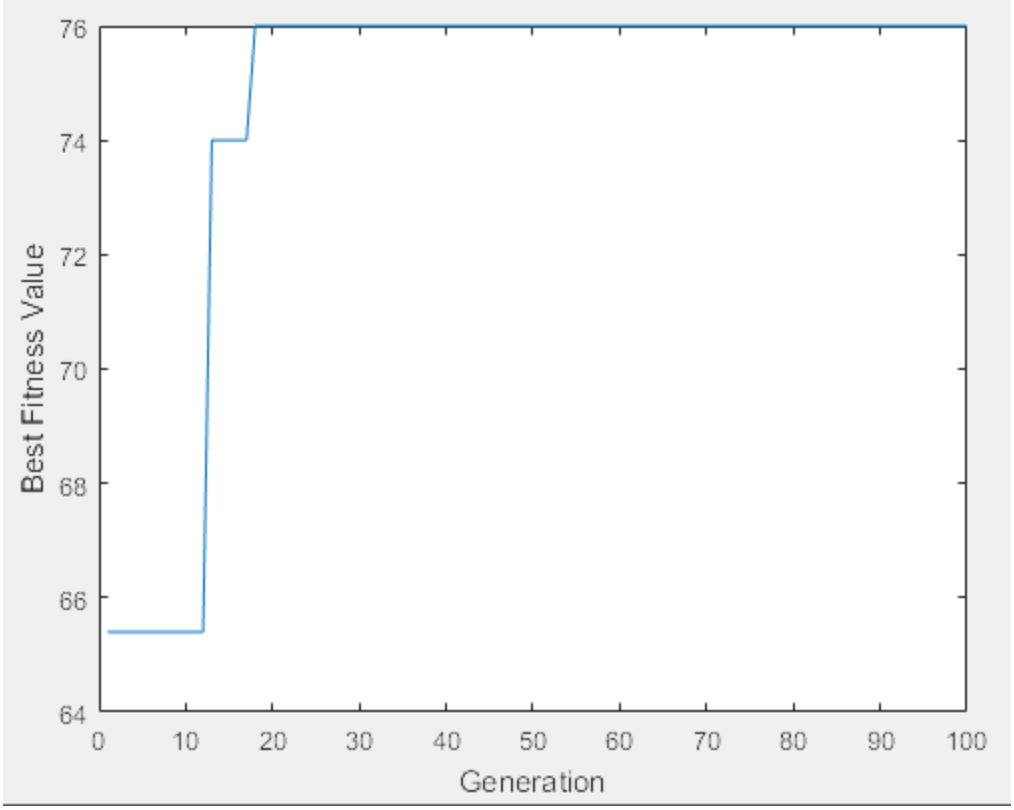
The group agrees that for an experimental program this does the task of creating generally favorable schedules that match user preferences very well and in a timely manner. 100 generations takes less than a minute to run on a modern computer!

Extra Results

Online classes, not tightly packed, morning classes, and no classes on Wednesday.

```
preferences = struct();
preferences.online_classes = 10;
preferences.tightly_packed = 0;
preferences.morning_classes = 10;
preferences.no_classes = ['W'];
```

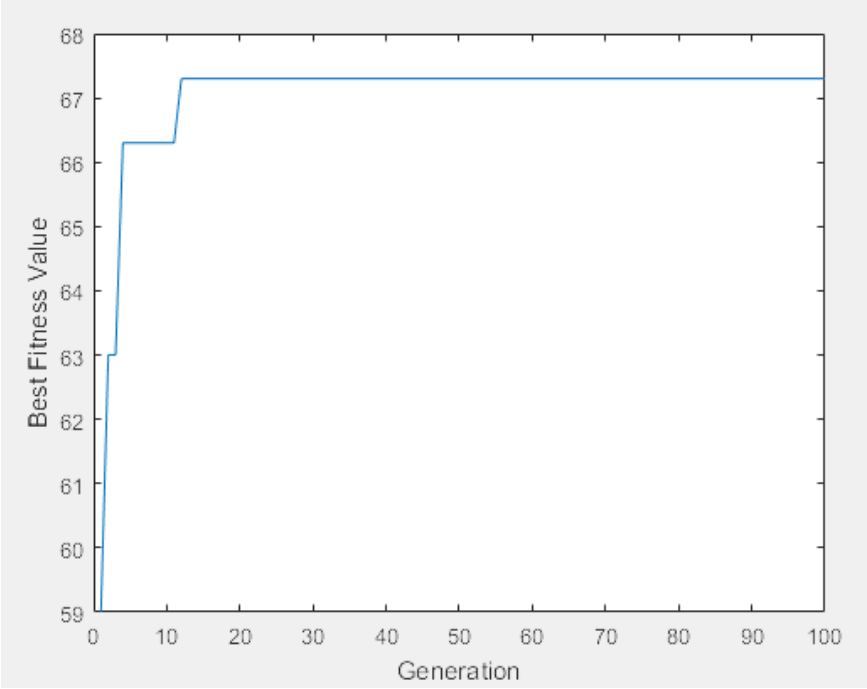
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Online
8:00am		CHEM102					ECON201
8:30am		CHEM102					
9:00am	CHEM102	CHEM102	CHEM102	CHEM102	CHEM102		
9:30am	CHEM102	CHEM102	CHEM102	CHEM102	CHEM102		
10:00am	ENGR102						
10:30am	ENGR102						
11:00am			ENGR102				
11:30am			ENGR102				
12:00pm			ENGR102				
12:30pm			ENGR102				
1:00pm							
1:30pm							
2:00pm							
2:30pm							
3:00pm							
3:30pm							
4:00pm							
4:30pm							
5:00pm							
5:30pm							
6:00pm							
6:30pm				CS283			
7:00pm				CS283			
7:30pm				CS283			
8:00pm				CS283			
8:30pm				CS283			
9:00pm				CS283			
9:30pm							
10:00pm							



No online classes, not tightly packed, no morning classes, and no classes on Monday.

```
preferences = struct();
preferences.online_classes = 0;
preferences.tightly_packed = 0;
preferences.morning_classes = 0;
preferences.no_classes = ['M'];
```

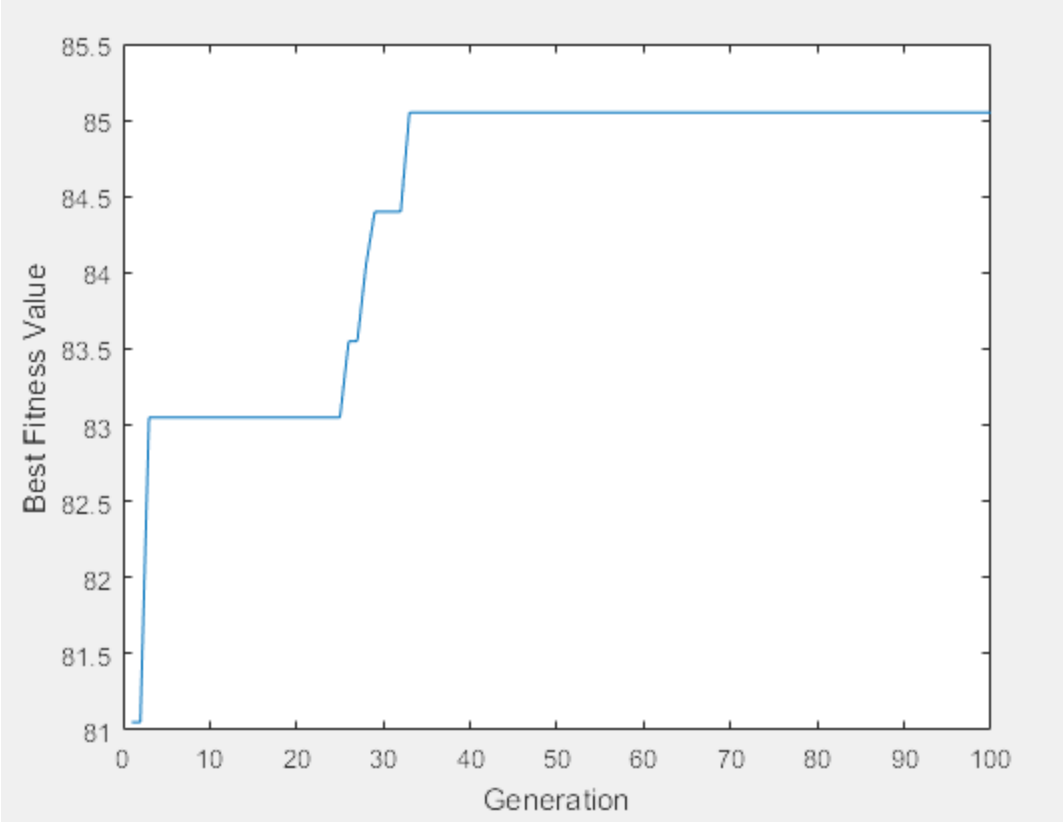
	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Online
8:00am							ECON201
8:30am							
9:00am							
9:30am							
10:00am							
10:30am							
11:00am							
11:30am							
12:00pm							
12:30pm							
1:00pm			ENGR102				
1:30pm			ENGR102				
2:00pm	ENGR102		ENGR102				
2:30pm	ENGR102		ENGR102				
3:00pm	CHEM102						
3:30pm	CHEM102						
4:00pm	CHEM102		CHEM102	CHEM102	CHEM102		
4:30pm	CHEM102		CHEM102	CHEM102	CHEM102		
5:00pm				CHEM102			
5:30pm				CHEM102			
6:00pm							
6:30pm		CS283					
7:00pm		CS283					
7:30pm		CS283					
8:00pm		CS283					
8:30pm		CS283					
9:00pm		CS283					
9:30pm							
10:00pm							



Middle of the road preferences for everything.

```
preferences = struct();
preferences.online_classes = 5;
preferences.tightly_packed = 5;
preferences.morning_classes = 5;
preferences.no_classes = ['F'];
```

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Online
8:00am							ECON201
8:30am							
9:00am							
9:30am							
10:00am				CHEM102			
10:30am				CHEM102			
11:00am				CHEM102	ENGR102		
11:30am				CHEM102	ENGR102		
12:00pm					ENGR102		
12:30pm					ENGR102		
1:00pm	CHEM102		CHEM102		CHEM102		
1:30pm	CHEM102		CHEM102		CHEM102		
2:00pm	ENGR102				CHEM102		
2:30pm	ENGR102				CHEM102		
3:00pm							
3:30pm							
4:00pm							
4:30pm							
5:00pm							
5:30pm							
6:00pm							
6:30pm		CS283					
7:00pm		CS283					
7:30pm		CS283					
8:00pm		CS283					
8:30pm		CS283					
9:00pm		CS283					
9:30pm							
10:00pm							



Crossover

The crossover is where the majority of the genetic portion of our implementation resides. To start, our crossover function had the added requirement that each child had to be a permutation of the parents' classes. Because of this, the group leaned on the concepts learned from the cryptography problem solved in homework 4.

The final crossover function is defined as follows:

A. For each class in the parents perform the following:

1. Select one of the two parents to copy the current class into the child.
2. Repeat for each class.

3. Give a 33% chance to mutate the child after the crossover.

An example print out from our program can be seen below:

```
Parent1
ECON201 Lecture M 6:00pm-9:50pm
CS432 Lecture W 6:00pm-8:50pm
CS283 Lecture TR 12:30pm-1:50pm
CHEM102 Lecture MWF 12:00pm-12:50pm
CHEM102 Lab W 10:00am-11:50am
CHEM102 Recitation/Discussion M 1:00pm-1:50pm
Parent2
ECON201 Lecture TR 12:00pm-1:50pm
CS432 Lecture W 6:00pm-8:50pm
CS283 Lecture R 6:30pm-9:20pm
CHEM102 Lecture MWF 4:00pm-4:50pm
CHEM102 Lab W 12:00pm-1:50pm
CHEM102 Recitation/Discussion W 10:00am-10:50am
Child
ECON201 Lecture TR 12:00pm-1:50pm
CS432 Lecture W 6:00pm-8:50pm
CS283 Lecture R 6:30pm-9:20pm
CHEM102 Lecture MWF 12:00pm-12:50pm
CHEM102 Lab W 10:00am-11:50am
CHEM102 Recitation/Discussion T 2:00pm-2:50pm
```

We can step through several of the classes in the child and see which parent they came from. ECON201 and CS432 clearly came from Parent2, while CS283 and CHEM102 Lecture and Lab came from Parent1. The one anomaly we find is the CHEM 102 Recitation/Discussion. There is no Tuesday section in either Parent1 and Parent2.

This is an example of our mutation at work. This position was swapped with a random CHEM102 Recitation/Discussion from the term master schedule.

Several other methods of crossover were attempted, but none were better than the previously discussed method. An example of another method attempted would be swapping a single class between parent 1 and 2 to result in two new children. This attempt made for an extremely slow program, but it did make progress. Another method attempted, simply for comparison, was to make the child a random permutation of one of the parents. It would draw a random class from the term master schedule for each class position. This is basically a random crossover and it resulted in the child not retaining any of the progress or health of the parents.

Mutation

The mutation is performed within our crossover function. The mutation has a 33% chance to occur. When a mutation occurs, we select a random class and replace with another random section of that class from the term master schedule. The code for this implementation can be seen below:

```
if randi(3,1,1) == 1
   classSwap = randi(size(child),1,1);
   nums = size(all_classes{classSwap},1);
   order_to_try = randperm(nums);
   for i = 1:nums
        if Fits(child,all_classes{classSwap}(order_to_try(i)))
            child{classSwap} = all_classes{classSwap}(order_to_try(i));
            break;
        end
   end
end
```

The group settled on a 33% chance to mutate based on some lengthy experimentation. Higher mutation rates resulted in seemingly random schedules and a less stable rate of ascent for the fitness function plot. ie: The schedules were not getting consistently better. Lower mutation rates caused the population to become stagnant. This means that the schedules resemble each other too much. This can be described as a similar problem to hill climbing. We reach a local maximum quicker when using a lower mutation rate. The mutation should be able to get us off of local maximum in order to find a more optimal maximum. The population would plateau the fitness score quicker and not be diverse enough. 33% mutation rate remained the best rate that gave us favorable and reliable schedules.

Fitness Function

The fitness function was built off of the letter frequency fitness function for the cryptographic problem in HW4. The general idea is to keep a running score. For each user preference, come up with a way to generate points based on whether the schedule matches the user's preference. The function's implementation became a simple analysis of the schedule to check for each of the allowed preferences. The higher the score, the better the schedule fits the user's preferences.

As you can see in the code below, we keep a fitness score that is summed to each of the preference scores. We have scoring for online classes, morning classes, tightly packed, and no classes preferences.

```
fitness = 0;

fitness = fitness + (preferences.online_classes * online_classes);
fitness = fitness + ((10-preferences.online_classes) * (size(classes,1) - online_classes))/2;

for i = 1:size(mapkeys,2)
    times = mapObj(char(mapkeys(i)));
    if times(3) ~= 0
        fitness = fitness + ((14-((times(2) - times(1))/100)) * preferences.tightly_packed)/10;
        fitness = fitness + ((14-((times(1)-800)/100)) * preferences.morning_classes)/10;
    end
end

for i = 1:size(preferences.no_classes)
    time = mapObj(char(preferences.no_classes(i)));
    if time(3) == 0
        fitness = fitness + 10;
    end
end
```

The group experimented with creating differing weights and restrictions on the scores. The end result is what you can see above. Each preference is scored equally based on a 10 point scoring scale. The final fitness function value can be summarized as summing of each of the individual preference scores:

$$F = F1 + F2 + F3 + \ldots + Fn$$

Comparison with Other Methods

One method we tried to look into was a brute force method where we generate every possible schedule and then run the fitness function to find the best possible schedule. The problem is the problem set gets exponentially bigger for every class you need. In our example, with 7 classes all with 4-49 sections available there are a staggering 24,300,864 possible combinations of classes. Trying to create all these and run a fitness function on all of them would take weeks, compared to our schedule

creator which creates a very good schedule in just minutes. Running at 1/10th a second per schedule:

24,300,864, / 10 per second / 60 seconds / 60 minutes / 24 hours = 27.7 Days!

In comparison, our program generates a sub optimal schedule within a minute! It should be noted that a brute force approach would result in an optimal solution if one existed. However, most students that are making a class schedule do not require a completely optimal schedule, nor do that have 27 days to wait on the creation of such a schedule.

To compare with more traditional search methods, the group implemented a depth first search (DFS) to find a schedule that matched what the user desired. We originally attempted to make this in MATLAB, but due to the sluggish nature of MATLAB on large datasets we could not get the program to work and finish correctly. Thus, we scrapped it. We switched over to writing a Ruby based DFS that ran on the server. The end result comparison was that the DFS was significantly slower than the genetic approach that we implemented. Our genetic approach is in a slower less efficient language too! This is clearly a successful program in the group's opinion.

Uses in Other Applications

Due to time restrictions, the group was unable to further extend this project's implementation to work for other scheduling applications. We have postulated several other real world examples for when this genetic scheduling technique would be effective. Drexel schedules work on the order of tens, but scheduling for an airport

works on the order of hundreds of thousands. The airports could set preferences and let the genetic algorithm run days or months in advance to build an optimal schedule. The same could be done for train scheduling or anything that has a large number of permutations. The main idea is that you make a program that finds a good enough, or suboptimal, schedule for us. It wouldn't need to run for days on end either. Depending on the dataset size atleast.

Further Modifications

Given more time, the group would've definitely liked to have seen this integrated and implemented into a Drexel scheduling app for either iPhone or Android. Due to the fast nature of the implementation this is an algorithm that could easily run on a mobile device. Several of these scheduling apps exist for Drexel, but most utilize a slow and cumbersome algorithm. The apps could also be branched off for other schools. This would require an API to give us the same information the Drexel WebTMS API does though.

On the MATLAB side, adding in prompts for the desired classes and preferences would be a nice addition, but the group ran out of time. Implementing those features would require input checking and a whole host of other details that were not related to the point of the project, genetic algorithms.

Conclusion

Overall, this is an excellent method for creating class schedules! These small scale experiments proved to be effective and a great way to build a favorable near optimal Drexel class schedule. We've shown a method for implementing a program that can solve the given problem for us instead of having the programmers solve the problem directly!