COEN 432

WINTER 2018

“Wedding Seating Plan”

By: Ryan Nichols

29787739

For: Dr. Kharma

Due: Feb 1st

Part A

1.

Representation:

Representing the the guests at this wedding I have a class defined as *Person.* Person objects have a name, index according to their position in the csv file, table number, position number (seat position) and a list of preferences corresponding to all other Person instances index. Person objects have access to member functions print_preferences() mostly for testing, assign_table() and assign_position used to initially seat each guest and to keep track of their changing positions during cross over or mutations.

For tables at the wedding I have a class *Table* that has attributes size, table number and a list of guests seated at it. Table has access to member functions print_table() used during testing and seat_guest() used to during population initialization at which point guests are assigned their seats at random.

My population members are of a class called *Arrangement.* Arrangements have members tables which is a list of Table objects, table_seat_tuples which is a list of tuple pairs (table #, seat #). When this list is shuffled guests can be assigned their seats at random by pulling from this list. Arrangements also have number_tables, table_size and guest_list which is a copy of the list of guests (Person) created from the csv input file and of course Arrangements finally has a fitness attribute.

Arrangements more important member functions are get_fitness() which I'll discuss in depth later, get_penalty() used in the former, output_csv() which provides the solution file, check_same_table() which does what you'd think when passed two Person objects, same for check_next_to() which makes a call to the previous, seat_guests() which takes advantage of

the shuffled list of tuples mentioned earlier and finally two methods that deserve some discussion.

If I had to start this project again I would probably leave out the Table class and represent all tables as a single list being a member of Arrangement. The actual genotype here is a list of *Table* objects which were difficult to work with during crossover. It didn't seem realistically feasible to perform PMX crossover on multiple lists so I included member functions in class Arrangement export_master_list() and import_master(). The former exports a single list of the guests at all tables, preserving their order and any empty seats. The later does the reverse and takes a list in that format and breaks it back into the correct number of tables and assigns this imported list to its tables data member. This made my program more complex than was necessary but I decided to deal with the complexity and keep moving forward rather than restart with the more ideal representation. In the end I guess you'd consider the multiple tables translated into one list as my genotype and in fact it worked well with PMX crossover and a basic mutation.

Finally there is my class *Population.* In this file population.py I included some global variables at the top so that it would be easy to test some different parameters such as population size, window size etc. Population instances have members arrangement_list which holds a number of Arrangements according to the population size, table_size, number_tables and a copy of the guest list as guest_list.

Populations more important member functions are evolve_one_generation() does what you think and each part of it will be be discussed on other sections. The same can be said for pmx_crossover and a function which it calls named go_opposite_get_index() as well as mutate(). That leaves get_best_solution() for finding the best Arrangement from the Population instance and finally add_randomness() which takes a sorted list and an upper

bound on the number of possible swaps to be made, then gets a random number within that range and makes the swaps.

Initialization

In my main.py file I read the settings.txt file and determine the number of guests and table size for the simulation. I also read the input csv file and create a list of guests that will be passed to the constructor of my one Population instance. I only need one because it's list of arrangements evolves. In this file I also determine the number of tables required and pass that value to the constructor mentioned.

In Population's constructor I initialize the population by creating a number of Arrangements and for each I call their create_tables() and and seat_guests() member functions so guests are seated at random.

Population

Was discussed at length in the representation section.

Parent Selection Mechanism

This takes place in Population's member function evolve_one_generation() and is marked with a comment. The pseudo code is as follows:

Take a window from the list of Arrangements according to the window size
For each Arrangement in window
    Update the Arrangement's fitness value
Sort the window according to each Arrangement's fitness
Take the two Arrangements with the lowest values

Fitness Evaluation

Fitness evaluation takes place in one of Arrangement's member functions so Arrangment objects update their own fitness. The pseudo code is as follows:

Initialize total penalty to zero

For each table
For each guest at that table
If that guest is not actually an empty seat
For each table
For each guest at that table
If the two guests are not the same and the inner loop guest is not actaully an empty seat
Get the preference between the two guests
Determine if there should be a penalty and add this to total penalty if so

Set the Arrangment's fitness to the total penaly divided by two since we've counted each penalty twice

I will not go into the details of the function get_penalty() since the algorithm is given in the assignment description except to say that it was implemented in an if elif statement since I've used Python for this assignment.

Mutation and Crossover

Mutation is quite simple. Since in my representation I have lists of tables, which for crossover and mutation purposes are converted to a single list, mutation was achieved by determining how many mutations to execute by random but within a range determined by a parameter and for each mutation a pair of guests are chosen at random to switch places. Could be within a table or from one table to the next. The psuedo code for the function mutate() within object Population is as follows:

Given a list of guests and a max number of mutations

Determine a number of mutations within the range of the max

For each change in the range of changes to be made

Get a first random index from the list

Get a second random index from the list

Get the guest object from the first index

If it is not actually an empty seat

Assign this guest a new table number and seat number based on the second index

Get the guest object from the second random index

If it is not actaully an empty seat

Assign this guest a new table number and seat number based on the second index

Swap the two guest's positions regardless of whether or not they are empty seats

Crossover is more complex and my pmx_crossover() from Population calls another function named go_opposite_get_index(). I will give the psuedo code for this function separately and first to keep the pseudo code organized. The enter function is only there to assist in demonstrating recursion.

Enter function:

Given an original guest, original index, guest list1, guest list2, child list and table size

If the guest is not already in the child list

    Get the guest(2nd guest) from list1 at the given index

    If this guest is actually an empty seat

    Place the original guest in the child list at the original index

    Else

    Find the index(2nd index) in list2 of the 2nd guest

    If the child list at this index is an empty seat

        place the original guest here

Assign the original guest a table number and seat number

Accordingly

Else

Enter function again passing the original guest, 2nd index and same 3 lists

Now the pseudo code for the pmx_crossover():

Given a list1, list2 and table size

Create a child list of empty seats of size length of list 1

Get a random value of the segment start from range length of list1 - 2

Get a random value of the end of segment between the start value and the length of list1 -1

Copy the segment from list1 to the child list

For indices between segment start and end

If guest in list2 at the index is not empty seat

Call go_opposite_get_index passing guest, list1, list2, child list and table size

For each guest in list2

If this guest is not in the child list

Seat the guest in the first empyt seat in the child list

Update the guests table number and seat position accordingly

Return the child list from the function

Survivor Selection

Given two selected parents and two children resulting from crossover, in my evolve_one_generation() in Population I duplicated the guest lists of these four, *MUTANT_GROUPS* number of times and mutated those lists. Then I determined the fitness of each, sorted the entire list according to fitness. Then to increase diversity I swapped random positions in the list a random number of times within the range set by the parameter *LIMIT_CHANGES_SURVIVIOR_SELECTION.* The pseudo code is as follows.

Given a list of mutants

Sort the list by fitness

For within a randomly determined range set by a parameter

Pick two mutants and swap their positions

Take the window size -2 number of mutants from the bottom of the list and add these to a list that will become the next population

Add both parents to that list

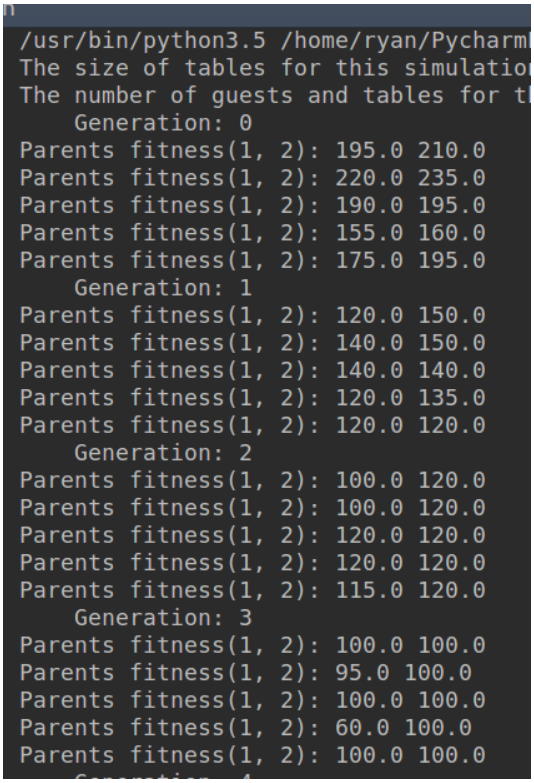
2.

Testing

The testing was done on the given preferences csv file with 15 guests and variations that I made of that file. Mostly just decreasing the number of guests so that I could more quickly determine if I was getting the correct fitness and to see the effect of fewer guests while trying to reach an overall fitness of zero.

I was able to achieve improvement from the original population which quickly levelled off at a value somewhat close to zero. It depended on the changing parameters and I was able to achieve a fitness of zero for a group of 7 guests taken from the csv file provided but just shortened.

For the given csv file with 15 guests a random sample of my output is below. In each generation I print the fitness of the parents since they are usually but not always the best from each window. With a population size of 20, a window size of 4, an upper limit to the number of mutations per genome of 10, an upper limit on the number of random swaps during survivor selection of 15 and creating 50 mutant groups from which to select survivors:



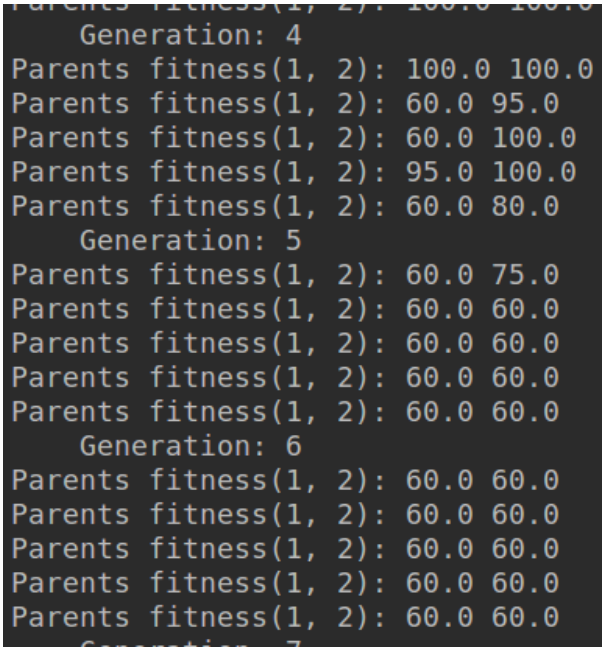


figure 1

figure 2

You can see that by the sixth generation the diversity, this is the diversity just at a glance as at this point in project I haven't implemented a function to actaully measure diversity, has been exhausted and I've reached as best a solution I can manage. Changing the parameters did not yield a better result than 60 for 15 guests.

Now with all other parameters the same besides changing the table size to 4, the number of guests to 7.

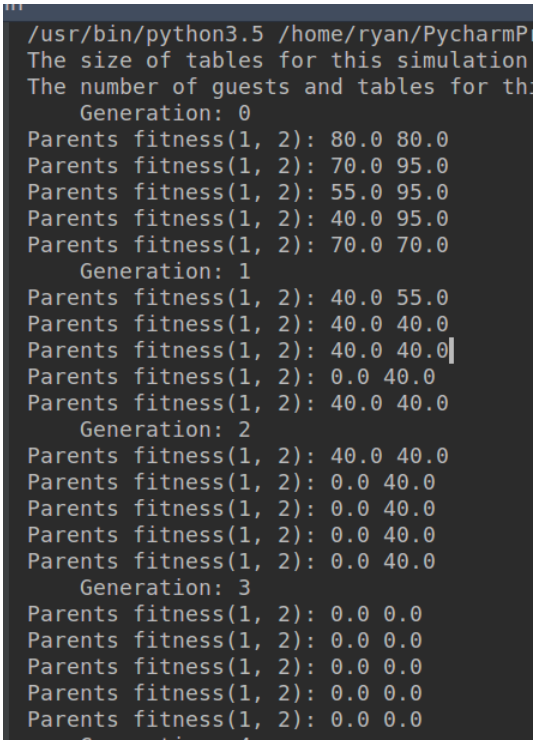


figure 3

You can see by the fourth generation I have achieved correct solutions.

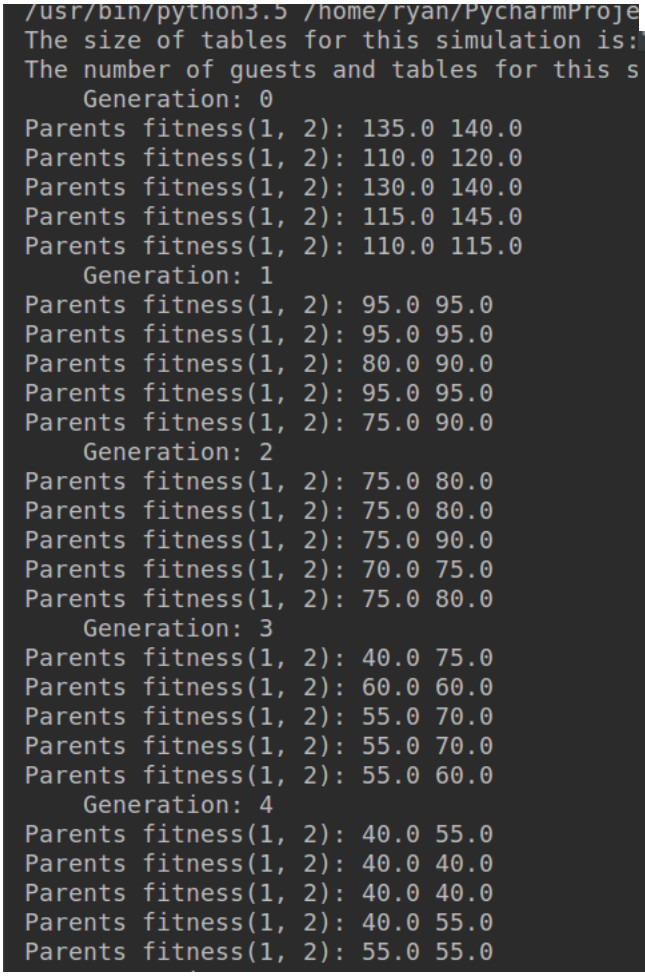


figure 4

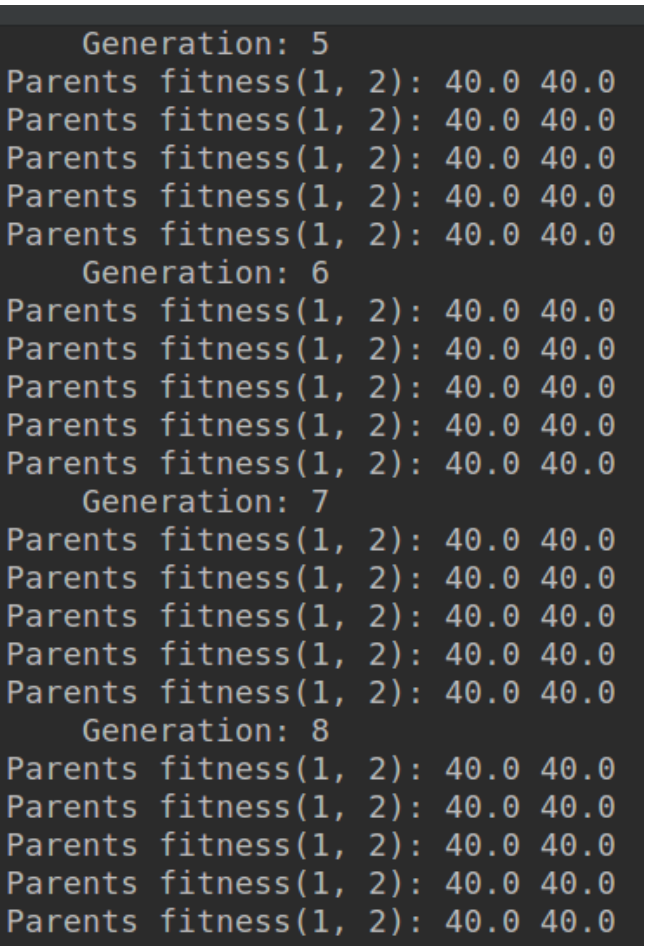


figure 5

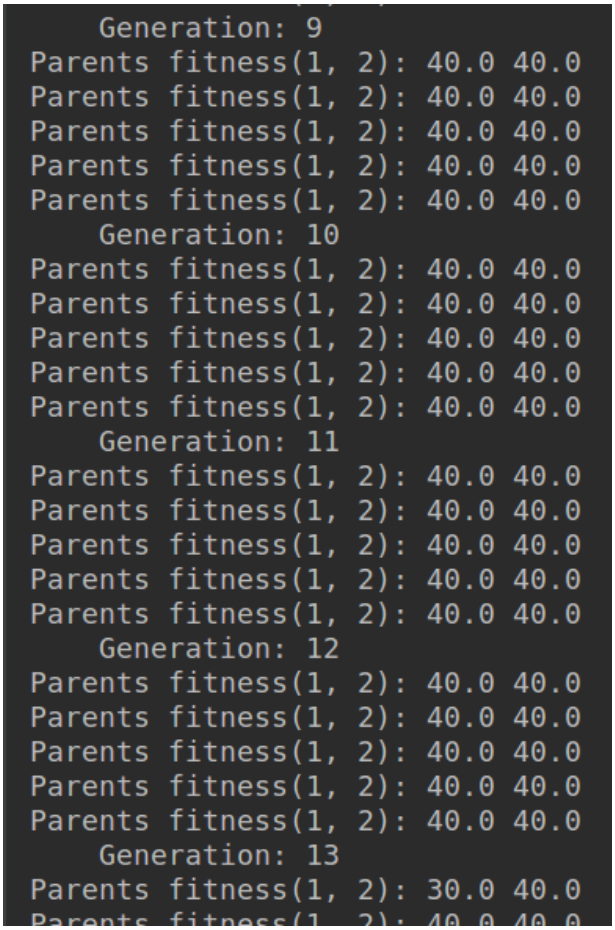


figure 6

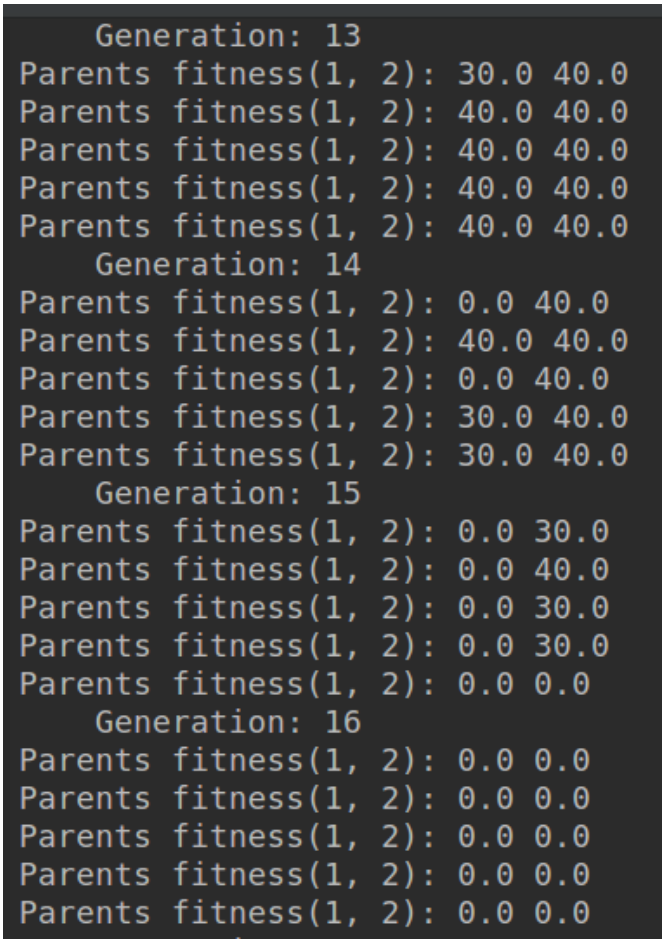


figure 7

You can see that around generation 10 and 11 it looks as though the progress has plateaued and then

Part B

The model I chose for diversity enhancement is a variation on crowding that I came up with myself. In part A for survivor selection I took mutants created from the offspring of two parents and put them in a list, then sorted them by fitness, added some randomness to their order and selected the lowest ordered mutants in the list until I had a window sized group and put these into the next population.
In my version of crowding I had two lists of mutants that want to move on to the next generation one for each parent. Which list the mutants go into depends on which parent they are more similar to. Then after sorting and adding randomness, when passing mutants on to the next generation we select from the two lists evenly. One from one list and one from the other. In this way mutants are only competing with those who are similar to their most similar parent between their two parents.
I wasn't able to noticeably improve diversity using this method.

For testing I implemented a function in class Population called test_diversity(), used to make sure that my algorithm for calculation the diversity between two arrangements was working correctly. It just sets up an arrangement which is the same as one provided in the problem description and then running the program to make sure I was getting the correct diversity.

Part C

Installation and Running Instructions

This program was developed using Python3 under Linux (Ubuntu) environment. I developed it using PyCharm and I've never used anything else so not entirely sure if its the same for all IDE's but you should be able to just take the files and import them into any IDE and run them. I have not made any modifications to them to make them runnable from the terminal like changing the chmod or anything like that so I think an IDE may be required. As a note I did not define a termination condition in my program other than to just let the program run to the end of the NUMBER_GENERATIONS you'll

find at the top of main.py which I set as 150 but for smaller number of guests in the 15 and under range you'll be able to get the best possible solution in many fewer generations, sometimes as low as 50.